Микель Кохендерфер, Тим Уилер, Кайл Рэй

Алгоритмы принятия решений

ДМК ПРЕСС издательство

Системы автоматического принятия решений и поддержки принятия решений человеком широко используются в различных областях – от предотвращения столкновений самолетов до скрининга рака молочной железы. При разработке таких систем важно учитывать различные источники неопределенности, тщательно соблюдая баланс между несколькими целями.

Данная книга представляет собой полное введение в теорию алгоритмов принятия решений в условиях неопределенности, включая формулировки основных математических задач и методы их решения.

Среди рассматриваемых тем:

- принципы принятия решений в простых одношаговых задачах;
- последовательные решения в стохастической среде;
- взаимодействие с окружающей средой при отсутствии начальной модели поведения;
- решения при отсутствии информации о состоянии окружающей среды;
- решения в условиях неопределенности текущего и последующего состояния.

Основное внимание уделяется планированию и обучению с подкреплением, хотя некоторые из представленных методов основаны на элементах обучения с учителем и оптимизации.
Алгоритмы реализованы на языке программирования Julia.

Издание предназначено специалистам в области искусственного интеллекта и систем принятия решений, а также может быть полезно студентам и аспирантам.

ISBN 978-5-93700-187-0

Микель Кохендерфер, Тим Уилер, Кайл Рэй

Алгоритмы принятия решений

Algorithms for Decision Making

MYKEL J. KOCHENDERFER
TIM A. WHEELER
KYLE H. WRAY

The MIT Press
Cambridge, Massachusetts
London, England

Алгоритмы принятия решений

МИКЕЛЬ КОХЕНДЕРФЕР
ТИМ УИЛЕР
КАЙЛ РЭЙ

ДМК ПРЕСС
издательство
Москва, 2023

УДК 519.81
ББК 22.18
К75

Кохендерфер М., Уилер Т., Рэй К.
К75 Алгоритмы принятия решений / пер. с англ. В. С. Яценкова. – М.: ДМК Пресс, 2023. – 684 с.: ил.

ISBN 978-5-93700-187-0

Книга представляет собой введение в теорию алгоритмов принятия решений в условиях неопределенности, включая формулировки основных математических задач и методы их решения. Рассмотрены современные методы снижения вычислительной нагрузки и поиска оптимальных стратегий в различных сценариях – от простых регуляторов до стохастических многоагентных систем. Основное внимание уделяется планированию и обучению с подкреплением, хотя некоторые из представленных методов основаны на элементах обучения с учителем и оптимизации. Алгоритмы реализованы на языке программирования Julia.

Издание предназначено специалистам в области искусственного интеллекта и систем принятия решений, а также может быть полезно студентам и аспирантам.

УДК 519.81
ББК 22.18

ISBN 978-0-2620-4701-2 (англ.)

ISBN 978-5-93700-187-0 (рус.)

Содержание

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Manning Publications очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Предисловие

Эта книга представляет собой развернутый обзор, посвященный алгоритмам принятия решений в условиях неопределенности. Мы постарались охватить максимально широкий спектр тем, связанных с принятием решений, и познакомить читателей с формулировками основных математических задач и алгоритмами их решения. В книге приведены рисунки, примеры и упражнения, которые помогают раскрыть идеи, лежащие в основе различных подходов.

Книга предназначена для студентов старших курсов и аспирантов, а также специалистов в области искусственного интеллекта и систем принятия решений. Чтение этой книги требует определенного уровня математической подготовки и предполагает предварительное знакомство с многомерным исчислением, линейной алгеброй и теорией вероятностей. Некоторые вводные материалы представлены в приложениях. К дисциплинам, в которых книга будет особенно полезна, относятся математика, статистика, информатика, аэрокосмическая промышленность, электротехника и операционные процессы.

В основе этого учебника лежат алгоритмы, реализованные на языке программирования Julia. Мы обнаружили, что этот язык идеально подходит для описания алгоритмов в удобочитаемой форме. Приоритетом при разработке алгоритмических реализаций была простота понимания, а не эффективность кода. Для использования на практике могут понадобиться более оптимальные реализации. Разрешается бесплатно использовать любые фрагменты кода из этой книги с обязательным указанием ссылки на источник кода.

Микель Кохендерфер
Тим Уилер
Кайл Рэй
Стэнфорд, Калифорния,
28 февраля 2022 г.

Благодарности

Этот учебник написан на основе университетского курса «Принятие решений в условиях неопределенности», преподаваемого в Стэнфорде. Мы благодарны студентам и ассистентам преподавателей, которые помогали формировать курс в течение последних шести лет.

Авторы хотели бы поблагодарить многих людей, поделившихся ценными отзывами о ранних черновиках нашей рукописи, включая Дилана Асмара, Дрю Бэгнелла, Сафу Бахши, Эдварда Балабана, Джина Беттертона, Раунака Бхаттачарию, Келси Бинга, Максима Бутона, Остина Чана, Саймона Шовина, Шушмана Чоудхури, Джона Кокса, Мэттью Дейли, Викторию Дакс, Ричарда Дьюи, Деа Дрессель, Бена Дюпри, Торстейна Элиассена, Йоханнеса Фишера, Рушила Горадиа, Джайеша Гупту, Арека Джамгочяна, Рохана Капре, Марка Корена, Лиама Круза, Тора Латтимора, Бернарда Ланге, Ричи Ли, Шэна Ли, Майкла Литтмана, Роберта Мосса, Джошуа Отта, Ориану Пельтцер, Франческо Пикколи, Джеффри Сарнофф, Марка Шлихтинга, Рансалу Сенанаяке, Челси Сидран, Криса Стронга, Зака Санберга, Абия Тешоме, Александроса Цикаса, Кемаля Юра, Джоша Вольфа, Анила Йылдыза и Цзунчжана Чжана. Мы также хотели бы поблагодарить Сиднея Каца, Кунала Менду и Аяна Мукхопадхьяя за их вклад в обсуждение главы 1. Росс Александер разработал многие из упражнений, включенных в книгу. При подготовке этой рукописи к публикации нам было приятно работать с Элизабет Суэйзи из MIT Press.

При работе над этим учебником мы использовали различные пакеты программ с открытым исходным кодом (см. приложение G).

1 Введение

В нашей повседневной жизни многие важные задачи связаны с принятием решений в условиях неопределенности. К ним относятся, например, предотвращение столкновений самолетов, борьба с лесными пожарами и реагирование на стихийные бедствия. При разработке автоматизированных систем принятия решений или систем поддержки принятия решений важно учитывать различные источники неопределенности, тщательно соблюдая баланс между несколькими целями. Мы обсудим эти проблемы с вычислительной точки зрения, стремясь раскрыть теорию, лежащую в основе моделей принятия решений и вычислительных подходов. В этой главе представлена проблема принятия решений в условиях неопределенности, приведены некоторые примеры применения и очерчено пространство вычислительных подходов. Затем мы покажем, как различные дисциплины способствуют нашему пониманию разумного принятия решений, и определим области потенциального взаимодействия систем принятия решений и общества. Главу завершает краткий анонс оставшейся части книги.

1.1. Принятие решений

Агент – это сущность, которая действует на основе наблюдений за окружающей его средой. Агенты могут быть физическими объектами, такими как люди или роботы, или невещественными объектами, такими как системы поддержки принятия решений, которые полностью реализованы программно. Как показано на рис. 1.1, взаимодействие между агентом и окружающей средой следует *циклу «наблюдение-действие»*, который иногда называют *петлей*.

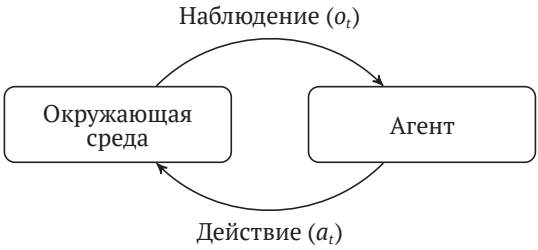


Рис. 1.1. Взаимодействие между агентом и его окружением

Агент в момент времени t получает *наблюдение* за окружающей средой, обозначенное как o_t. Наблюдения могут производиться, например, посредством биологических органов чувств, как у людей, или с помощью электронной сенсорной системы, такой как радар в системе управления воздушным движением. Наблюдения часто бывают неполными или зашумленными; например, люди могут не различить приближающийся самолет среди других отметок на экране, или радиолокационная система может пропустить обнаружение из-за электромагнитных помех. Затем агент выбирает *действие* a_t в процессе принятия решения.

Действие агента, такое как подача сигнала тревоги, может оказывать недетерминированное влияние на среду.

Мы будем говорить об агентах, которые разумно взаимодействуют для достижения своих целей на протяжении определенного времени. Учитывая прошлую последовательность наблюдений $o_1, \ldots, o_t$ и знание среды, агент должен выбрать действие, которое лучше всего способствует достижению его цели при наличии различных источников неопределенности[1], включая следующие:

- *неопределенность результата*, когда последствия наших действий не определены;
- *неопределенность модели*, когда наша модель задачи не определена;
- *неопределенность состояния*, когда истинное состояние окружающей среды остается неопределенным;
- *неопределенность взаимодействия*, когда поведение других агентов, взаимодействующих в окружающей среде, не определено.

Эта книга построена вокруг упомянутых четырех источников неопределенности. Принятие решений в условиях неопределенности занимает центральное место в области *искусственного интеллекта*[2], а также во многих других областях, как указано в разделе 1.4. Мы обсудим различные алгоритмы или описания вычислительных процессов для принятия решений, устойчивых к неопределенности.

1.2. Области применения

Обобщенную систему принятия решений, упомянутую в предыдущем разделе, можно применить к широкому кругу задач. В этом разделе мы рассмотрим несколько концептуальных примеров реального применения. Приложение F содержит описание других гипотетических задач, которые применяются в этой книге для наглядной демонстрации обсуждаемых алгоритмов.

---

[1] Здесь мы сосредоточимся на задачах дискретного времени. Задачи с непрерывным временем изучаются в области теории управления. См. D. E. Kirk, *Optimal Control Theory: An Introduction.* Prentice-Hall, 1970.

[2] Всестороннее введение в искусственный интеллект предоставлено в S. Russell, P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Pearson, 2021.

1.2.1. Предотвращение столкновения самолетов

Чтобы помочь авиационным диспетчерам предотвратить столкновение самолетов в воздухе, мы должны разработать систему, которая способна предупреждать пилотов о потенциальных угрозах и указывать им, какие маневры следует совершить[3]. Система взаимодействует с транспондерами самолетов, чтобы определить их положение с определенной степенью точности. Решить, какие указания дать пилотам, очень непросто. Существует неопределенность в том, как быстро пилоты отреагируют и насколько усердно они будут выполнять указания. Кроме того, существует неопределенность в поведении других самолетов. Необходимо, чтобы наша система выдавала предупреждения достаточно рано, оставляя пилотам запас времени для маневра, чтобы избежать столкновений, но, с другой стороны, система не должна выдавать предупреждения слишком рано, так как это приведет ко множеству ненужных маневров, которые сами по себе служат источником дополнительной опасности и неопределенности. Поскольку эта система будет использоваться постоянно во всем мире, нам требуется решение, обеспечивающее исключительно высокий уровень безопасности.

1.2.2. Автоматизированное вождение

Инженеры стремятся создать автономный автомобиль, который сможет безопасно передвигаться в обычных городских условиях[4]. Автономное транспортное средство должно принимать безопасные решения, полагаясь на набор собственных датчиков для восприятия окружающей среды. Одним из таких датчиков является *лидар*, который использует отражение лазерного луча для определения расстояний до препятствий и построения объемной картины окружающей обстановки. Другой тип датчика – это камера, которая с помощью алгоритмов компьютерного зрения может обнаруживать пешеходов и другие транспортные средства. Оба этих датчика несовершенны и чувствительны к шуму и помехам. Например, припаркованный грузовик может загораживать пешехода, собравшегося перейти дорогу по пешеходному переходу. Наша система должна предсказывать намерения и ожидаемые траектории движения других транспортных средств, пешеходов и прочих участников дорожного движения на основе их наблюдаемого поведения, чтобы автомобиль безопасно добрался до пункта назначения.

1.2.3. Скрининг рака молочной железы

Во всем мире рак молочной железы является наиболее распространенным видом рака у женщин. Раннее обнаружение рака молочной железы может помочь

---

[3] Это применение обсуждается в главе, написанной М. Кохендерфером для книги *Decision Making Under Uncertainty: Theory and Application*. MIT Press, 2015.

[4] Аналогичное применение представлено в докладе M. Bouton, A. Nakhaei, K. Fujimura, M. J. Kochenderfer, *Safe Reinforcement Learning with Scene Decomposition for Navigating Complex Urban Environments*, in IEEE Intelligent Vehicles Symposium (IV), 2019.

спасти жизни, а маммография (рентгеновское сканирование) является наиболее эффективным доступным методом скрининга. Тем не менее маммография сопряжена с потенциальными рисками, включая ложноположительные результаты, которые могут привести к ненужному и инвазивному последующему диагностическому обследованию. Многолетние исследования ученых из разных стран позволили выработать критерии и методики скрининга населения в зависимости от возраста, позволяющие найти компромисс между пользой и рисками от обследования. Наличие системы, которая может давать рекомендации на основе личных характеристик риска и истории скрининга, может привести к существенному улучшению здоровья определенных групп населения[5]. Успех такой системы можно оценить в сравнении с подходом тотального скрининга, используя критерии общего ожидаемого количества лет жизни с поправкой на качество, количество маммограмм, долю ложноположительных результатов и риска невыявленного рака, требующего оперативного вмешательства.

1.2.4. Доля инвестиций и распределение портфеля

Предположим, мы хотим построить систему, которая рекомендует людям, какую часть личного дохода они могут потратить по итогам года и сколько должны инвестировать[6]. Инвестиционный портфель может включать в себя акции и облигации с разным уровнем риска и ожидаемой доходности. Рост личных накоплений носит стохастический характер из-за неопределенности как в заработке, так и в инвестиционном доходе, часто увеличиваясь до тех пор, пока инвестор не достигает пенсионного возраста, а затем неуклонно снижается. Удовольствие, получаемое от потребления условной единицы богатства в течение года, обычно уменьшается с увеличением потребляемой суммы, что приводит к желанию оптимально распределить потребление и инвестирование на протяжении всей жизни человека.

1.2.5. Распределенное наблюдение за лесными пожарами

При тушении лесных пожаров одной из главных проблем является недостаточная осведомленность о текущей ситуации. Состояние пожара постоянно меняется под влиянием таких факторов, как ветер и распределение горючего материала в окружающей среде. Многие лесные пожары охватывают большие географические регионы. Один из подходов к наблюдению за лесными пожарами заключается в использовании группы дронов, оснащенных датчиками[7].

---

[5] Такая концепция предложена в статье T. Ayer, O. Alagoz, and N. K. Stout, *A POMDP Approach to Personalize Mammography Screening Decisions,* Operations Research, vol. 60, no. 5, pp. 1019–1034, 2012.

[6] Схожая задача была представлена в R. C. Merton, *Optimum Consumption and Portfolio Rules in a Continuous-Time Model, Journal of Economic Theory*, vol. 3, no. 4, pp. 373–413, 1971.

[7] Это применение было детально представлено в K. D. Julian, M. J. Kochenderfer, *Distributed Wildfire Surveillance with Autonomous Aircraft Using Deep Reinforcement Learning*, AIAA Journal of Guidance, Control, and Dynamics, vol. 42, no. 8, pp. 1768–1778, 2019.

Дальность действия датчиков отдельных дронов ограничена, но информация от группы дронов может быть объединена для получения единого снимка ситуации и принятия решений о распределении ресурсов. В идеале хотелось бы, чтобы члены группы самостоятельно определяли, как сотрудничать друг с другом для построения наилучшей информационной картины пожара. Эффективный мониторинг требует принятия решения о том, как маневрировать, чтобы охватить области, в которых новая информация может оказаться полезной, – тратить время на сканирование участков, про которые мы точно знаем, есть ли там пожар, было бы расточительно. Выявление предпочтительных областей для исследования требует рассуждений о стохастической эволюции пожара, основанных лишь на неполных знаниях о его текущем состоянии.

1.2.6. Исследование Марса

Марсоходы сделали важные открытия и значительно улучшили наше понимание Марса. Однако основным узким местом в научных исследованиях была связь между марсоходом и операторами на Земле. Для отправки информации датчиков с Марса на Землю и для отправки ответных команд с Земли на Марс может потребоваться до получаса. Кроме того, наведение антенн марсоходов и другие аспекты их работы необходимо планировать заранее, потому что окна радиообмена Земля–Марс ограничены из-за расположения орбитальных аппаратов, служащих ретрансляторами радиосигнала между планетами. Недавние исследования показали, что эффективность научно-исследовательских миссий может быть повышена в пять раз за счет достижения более высоких уровней автономности[8]. Люди-операторы по-прежнему будут обеспечивать руководство высокого уровня в соответствии с целями миссии, но марсоход получит возможность выбирать свои собственные краткосрочные цели, используя самую свежую информацию. Кроме того, желательно, чтобы марсоход адекватно реагировал на различные опасности и системные сбои без вмешательства человека.

1.3. Методы создания агентов

Существует множество методов создания агентов, принимающих решения. В зависимости от назначения агента и предметной области некоторые могут быть более подходящими, чем другие. Они отличаются обязанностями разработчика и задачами, которые подлежат автоматизации. В этой книге основное внимание уделено планированию и обучению с подкреплением, но некоторые методы будут включать в себя элементы обучения с учителем и оптимизации.

---

[8] Эту концепцию представили и оценили D. Gaines, G. Doran, M. Paton, B. Rothrock, J. Russino, R.Mackey, R. Anderson, R. Francis, C. Joswig, H. Justice, K. Kolcio, G. Rabideau, S. Schaffer, J. Sawoniewicz, A. Vasavada, V. Wong, K. Yu, A.-a. Agha-mohammadi, *Self-Reliant Rovers for Increased Mission Productivity, Journal of Field Robotics*, vol. 37, no. 7, pp. 1171–1196, 2020.

1.3.1. Явное программирование

Самый прямой метод создания агента, принимающего решения, состоит в том, чтобы предусмотреть все сценарии, в которых может оказаться агент, и явно запрограммировать ответное поведение агента. Подход с явным программированием может хорошо работать для простых задач, но он возлагает на разработчика большую нагрузку и ответственность за полноту выработанной стратегии. Специально для упрощения программирования агентов разработаны различные языки и среды программирования.

1.3.2. Обучение с учителем

В некоторых случаях проще показать агенту, что делать, чем писать программу, которую будет выполнять агент. Разработчик предоставляет набор обучающих примеров, и алгоритм автоматического обучения должен обобщить эти примеры, найдя связи и закономерности в обучающих данных. Этот подход известен как *обучение с учителем* и широко применяется для задач классификации. Данный метод иногда называют *поведенческим клонированием*, когда он применяется для обучения сопоставлению наблюдений с действиями. Поведенческое клонирование работает наиболее хорошо, когда опытный разработчик действительно знает наилучший план действий для представленного набора ситуаций. Хотя существует множество различных алгоритмов обучения, в новых ситуациях они, как правило, не могут работать лучше, чем разработчики-люди, предложившие обучающие примеры.

1.3.3. Оптимизация

При использовании метода *оптимизации* разработчик задает пространство возможных стратегий принятия решений и показатель качества работы агента, который необходимо максимизировать. Оценка эффективности стратегии принятия решений обычно предусматривает запуск серии прогонов модели. Затем алгоритм оптимизации ищет в этом пространстве оптимальную стратегию. Если пространство относительно невелико, а мера качества не имеет множества локальных оптимумов, то можно использовать различные методы локального или глобального поиска. Хотя обычно предполагают, что для запуска прогонов требуется знание динамической модели, его не используют для направленного поиска.

1.3.4. Планирование

Планирование – это форма оптимизации, в которой используется модель динамики задачи, помогающая вести поиск. На сегодняшний день в литературе описано множество различных задач планирования. Большинство исследований сосредоточены на детерминированных задачах. Для некоторых задач может быть приемлемо аппроксимировать динамику детерминированной

моделью. Предположение о детерминированной модели позволяет нам использовать методы, которые легче масштабировать для многомерных задач. Для некоторых задач принципиально важно учитывать будущую неопределенность. Эта книга полностью посвящена именно таким задачам.

1.3.5. Обучение с подкреплением

Обучение с подкреплением ослабляет предположение о том, что модель известна заранее. Вместо этого стратегия принятия решений изучается «на ходу» – во время взаимодействия агента с окружающей средой. Разработчик должен только предоставить меру качества решений; оптимизацией поведения агента занимается обучающий алгоритм. Одна из интересных сложностей, возникающих при обучении с подкреплением, заключается в том, что выбор действия влияет не только на немедленный успех агента в достижении своих целей, но и на способность агента узнавать об окружающей среде и определять особенности задачи, которыми можно воспользоваться.

1.4. История автоматизации принятия решений

Идея автоматизировать процесс принятия решений уходит своими корнями в мечты ранних философов, ученых, математиков и писателей. Древние греки начали упоминать автоматизацию в мифах и исторических записях еще в 800 г. до н. э. Слово αυτόματο впервые встречается в «Илиаде» Гомера, где содержится упоминание автоматических машин, включая механические треножники, используемые для обслуживания гостей за обедом[9]. В XVII веке философы предложили использовать логические правила для автоматического разрешения разногласий. Их идеи заложили основу для механических систем принятия решений.

Начиная с конца XVIII века изобретатели начали создавать автоматические машины для выполнения рутинной работы. В частности, ряд инноваций в текстильной промышленности привел к разработке автоматических ткацких станков, которые, в свою очередь, заложили основу для первых фабричных роботов[10]. В начале XIX века описания интеллектуальных машин для автоматизации труда стали появляться в научно-фантастических романах. Слово «робот» впервые появилось в пьесе чешского писателя Карела Чапека «R.U.R.» (сокращение от чеш. Rossumovi univerzální roboti, «Россумские универсальные роботы») о машинах, которые могут выполнять работу вместо людей. Пьеса вдохновила других писателей-фантастов включить роботов в свои произведения. В середине XX века известный писатель и профессор Айзек Азимов изложил свое видение робототехники в своей знаменитой серии «Робот».

---

[9] S. Vasileiadou, D. Kalligeropoulos, N. Karcanias, Systems, *Modelling and Control in Ancient Greece: Part 1: Mythical Automata, Measurement and Control*, vol. 36, no. 3, pp. 76–80, 2003.

[10] N. J. Nilsson, *The Quest for Artificial Intelligence*. Cambridge University Press, 2009.

Практические реализации автоматизированного принятия решений хуже всего справляются с неопределенностью. Еще в 1991 г. Джордж Данциг, получивший известность благодаря разработке симплексного алгоритма, заявил:

> Оглядываясь назад, интересно отметить, что первоначальная задача, с которой началось мое исследование, до сих пор не решена, а именно задача планирования или составления расписания действий, особенно динамического планирования в условиях неопределенности. Успешное решение такой задачи могло бы (в конечном итоге за счет лучшего планирования) способствовать благополучию и стабильности в мире[11].

Хотя принятие решений в условиях неопределенности по-прежнему остается областью активных исследований, за последние несколько столетий исследователи и инженеры заметно приблизились к реализации идей древних мечтателей и философов. Современные алгоритмы принятия решений основаны на сближении концепций, возникших в самых разных дисциплинах, включая экономику, психологию, неврологию, информатику, инженерию, математику и исследование операций. Далее мы кратко рассмотрим некоторые основные достижения этих дисциплин. Своеобразное «перекрестное опыление» между дисциплинами уже не раз становилось источником передовых идей в области принятия решений. Вероятно, так будет и впредь.

1.4.1. Экономика

Экономика нуждается в моделях, имитирующих принятие решений человеком. Один из подходов к построению таких моделей опирается на теорию полезности, впервые представленную в конце XVIII века[12]. *Теория полезности* предоставляет методику для моделирования и сравнения желательности различных результатов. Например, полезность можно использовать для сравнения желательности денежных количеств. В книге «Основы законодательства» (1887) Иеремия Бентам отметил убывающую полезность богатства:

1. Каждой порции богатства соответствует порция счастья.
2. Из двух людей с неравным состоянием тот, у кого больше всего богатства, имеет больше всего счастья.
3. Избыток счастья богатого не будет столь велик, как избыток его богатства[13].

Объединив понятие полезности с понятием рационального принятия решений, экономисты середины XX века пришли к формулировке *принципа максимальной ожидаемой полезности*. Этот принцип является ключевой идеей при создании автономных агентов, принимающих решения. Из теории полезности впоследствии развилась *теория игр*, которая пытается описать поведение не-

---

[11] G. B. Dantzig, *Linear Programming, Operations Research*, vol. 50, no. 1, pp. 42–47, 2002.

[12] G. J. Stigler, The Development of Utility Theory. I, Journal of Political Economy, vol. 58, no. 4, pp. 307–327, 1950.

[13] J. Bentham, *Theory of Legislation*. Trübner & Company, 1887.

скольких агентов, действующих в присутствии друг друга и стремящихся максимизировать свою выгоду[14].

1.4.2. Психология

Психологи также изучают принятие решений человеком, как правило, с точки зрения человеческого поведения. Психологи с XIX века разрабатывают теории обучения методом проб и ошибок, изучая реакцию животных на раздражители. Исследователи заметили, что животные склонны принимать решения, основываясь на удовлетворении или дискомфорте, которые они испытали в предыдущих подобных ситуациях. Русский психолог Иван Павлов объединил эту идею с концепцией *подкрепления* после наблюдения за характером слюноотделения у собак при кормлении. Психологи обнаружили, что модель поведения можно усилить или ослабить, используя постоянное подкрепление в виде определенного стимула. В середине XX века математик и ученый-компьютерщик Алан Тьюринг высказал мысль о том, что машины могут учиться таким же образом:

> Превращение машины в универсальную машину было бы наиболее впечатляющим, если бы механизмы взаимодействия обходились очень ограниченными входными данными. Обучение человеческого ребенка во многом зависит от системы поощрений и наказаний, и это наводит на мысль, что возможно осуществить обучение машины только с двумя взаимоисключающими входами: одним для «удовольствия» или «вознаграждения» (R) и другим для «боли» или «наказания» (P)[15].

Исследования психологов легли в основу концепции *обучения с подкреплением* – важнейшего метода, применяемого для обучения агентов принятию решений в условиях неопределенности[16].

1.4.3. Нейробиология

Психологи изучают человеческое поведение в том виде, в каком оно наблюдается извне, а нейробиологи сосредоточены на внутренних биологических процессах, формирующих поведение. В конце XIX века ученые обнаружили, что мозг состоит из сети взаимосвязанных нейронов, которая отвечает за его способность воспринимать и осознавать окружающий мир. Пионер искусственного интеллекта Нильс Нильссон описывает значение этого открытия для систем принятия решений следующим образом:

> Поскольку именно мозг животного отвечает за преобразование сенсорной информации в действие, вполне очевидно, что в работах нейрофи-

---

[14] O. Morgenstern, J. von Neumann, *Theory of Games and Economic Behavior*. Princeton University Press, 1953.

[15] M. Turing, *Intelligent Machinery*, National Physical Laboratory, Report, 1948.

[16] R. S. Sutton, A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. MIT Press, 2018.

> зиологов и нейроанатомов, изучающих мозг и его основные компоненты, нейроны, можно найти несколько хороших идей[17].

В 1940-х годах исследователи впервые предложили рассматривать нейроны как отдельные «логические единицы», способные выполнять вычислительные операции при объединении в сеть. Эта идея легла в основу нейронных сетей, которые используются в области искусственного интеллекта для выполнения множества сложных задач.

1.4.4. Информатика

В середине XX века ученые-компьютерщики начали рассматривать проблему разумного принятия решений как задачу организации символических операций с помощью формальной логики. Компьютерная программа Logic Theorist, написанная в середине XX века для выполнения автоматических рассуждений, использовала этот способ формального мышления для доказательства математических теорем. Герберт Саймон, один из ее изобретателей, подчеркнул особую символическую природу программы, связав ее с человеческим разумом:

> Мы создали компьютерную программу, способную мыслить нечисловыми категориями, и тем самым решили извечную проблему разума и тела, объяснив, как система, состоящая из материи, может обладать свойствами разума[18].

Эти символические системы в значительной степени полагались на человеческий опыт. Альтернативный подход к интеллекту, называемый *коннекционизмом*, отчасти вдохновлен достижениями в области нейробиологии и сосредоточен на использовании искусственных нейронных сетей в качестве субстрата для зарождения искусственного интеллекта. Зная, что нейронные сети можно обучить распознаванию образов, коннекционисты пытаются научить машину разумному поведению на основе данных или опыта, а не жестко запрограммированных знаний экспертов. Идея коннекционизма лежит в основе успеха AlphaGo – программы, которая разгромила чемпионов игры в го, – а также большей части интеллектуальной составляющей автономных транспортных средств. Алгоритмы, сочетающие символический и коннекционистский подходы, остаются предметом активных исследований и сегодня.

1.4.5. Инженерия

Специалисты в области инженерии стараются научить физические системы, такие как роботы, принимать разумные решения. Всемирно известный робототехник Себастьян Трун описывает компоненты этих систем следующим образом:

---

[17] N. J. Nilsson, *The Quest for Artificial Intelligence*. Cambridge University Press, 2009.

[18] Цитируется по J. Agar, *Science in the 20th Century and Beyond*. Polity, 2012.

> Общим для систем робототехники является то, что они находятся в физическом мире, воспринимают свою среду с помощью датчиков и манипулируют своей средой с помощью движущихся объектов[19].

При проектировании робототехнических систем инженеры оперируют такими понятиями, как *восприятие*, *планирование* и *действие*. Физические системы воспринимают мир, используя свои датчики для формирования представления о характерных чертах окружающей их среды. Восприятие как отдельная область исследований сосредоточено на использовании показаний датчиков для построения представления о состоянии мира. Планирование требует рассуждений о способах выполнения задач, для которых предназначена система. Процесс планирования стал возможным благодаря достижениям вычислительной техники на протяжении нескольких последних десятилетий[20]. Как только план сформирован, автономный агент должен выполнять его в реальном мире. Для реализации плана нужны как аппаратные средства (в виде исполнительных механизмов), так и алгоритмы для управления исполнительными механизмами и подавления возмущений. *Теория управления* сосредоточена на стабилизации механических систем посредством управления с обратной связью[21]. Автоматические системы управления широко используются в промышленности, от регулирования температуры в печи до навигации аэрокосмических систем.

1.4.6. Математика

Агент должен иметь возможность количественно оценить свою неопределенность, чтобы принимать обоснованные решения. Теория принятия решений в значительной степени зависит от теории вероятностей, рассматриваемой в контексте конкретной задачи. В частности, для этой книги важную роль играет байесовская статистика. В 1763 г. была опубликована посмертная статья Томаса Байеса, содержащая выкладки, которые позже стали известны как *правило Байеса*. Его подход к вероятностному выводу то приобретал, то терял популярность до середины XX века, когда исследователи обнаружили, что байесовские методы дают существенный выигрыш в ряде ситуаций[22]. Математик Бернард Купман нашел практическое применение теории во время Второй мировой войны:

> Каждая операция, связанная с поиском, сопряжена с неопределенностью; ее можно трактовать количественно только с точки зрения [...] вероятности. Сейчас это можно рассматривать как трюизм, но, похоже, развитие оперативного анализа во время Второй мировой войны позволило понять ее практическое значение[23].

---

[19] S. Thrun, *Probabilistic Robotics, Communications of the ACM*, vol. 45, no. 3, pp. 52–57, 2002.

[20] G. E. Moore, *Cramming More Components onto Integrated Circuits, Electronics*, vol. 38, no. 8, pp. 114–117, 1965.

[21] D. A. Mindell, *Between Human and Machine: Feedback, Control, and Computing Before Cybernetics*. JHU Press, 2002.

[22] W. M. Bolstad and J. M. Curran, *Introduction to Bayesian Statistics*. Wiley, 2016.

[23] B. O. Koopman, *Search and Screening: General Principles with Historical Applications*. Pergamon Press, 1980.

Методы выборки (иногда называемые *методами Монте-Карло*), разработанные в начале XX века для крупномасштабных расчетов в рамках Манхэттенского проекта, сделали возможными некоторые методы логического вывода, которые ранее были недостижимы. Они послужили основой для байесовских сетей, популярность которых значительно возросла в конце XX века благодаря исследованиям в области искусственного интеллекта.

1.4.7. Исследование операций

Исследование операций (operations research, операционный анализ) связано с поиском оптимальных решений задач, таких как распределение ресурсов, инвестиции в активы и планирование технического обслуживания. В конце XIX века исследователи начали изучать применение математического и научного анализов к производству товаров и услуг. Исследования ускорились во время промышленной революции, когда компании начали подразделять свое управление на отделы, отвечающие за отдельные аспекты общих решений. Во время Второй мировой войны оптимизация решений применялась к распределению ресурсов в армии. Когда война закончилась, предприятия обнаружили, что те же концепции исследования операций, которые ранее использовались для принятия военных решений, могут пригодиться для оптимизации бизнес-решений. Это послужило толчком к развитию науки управления, как описал теоретик организационной деятельности Гарольд Кунц:

> Твердое убеждение этой группы состоит в том, что если управление, или организация, или планирование, или принятие решений являются логическим процессом, то его можно выразить в виде математических символов и отношений. Центральным подходом этой школы является модель, поскольку именно с помощью этих приемов проблема выражается в ее основных отношениях и в терминах выбранных целей или задач[24].

Стремление лучше моделировать и понимать бизнес-решения привело к развитию ряда современных концепций, таких как *линейное программирование*, *динамическое программирование* и *теория очередей*[25].

1.5. Воздействие на общество

Алгоритмические подходы к принятию решений уже изменили общество и, вероятно, продолжат это делать в будущем. Далее мы кратко расскажем о нескольких способах, которыми алгоритмы принятия решений приносят пользу обществу, и о проблемах, которые возникают на этом пути[26].

---

[24] H. Koontz, *The Management Theory Jungle*, Academy of Management Journal, vol. 4, no. 3, pp. 174–188, 1961.

[25] F. S. Hillier, *Introduction to Operations Research*. McGraw-Hill, 2012.

[26] Более подробное обсуждение представлено в Z. R. Shi, C. Wang, F. Fang, *Artificial Intelligence for Social Good: A Survey*, 2020. arXiv: 2001.01818v1.

Алгоритмические подходы способствуют сохранению благоприятной экологической обстановки. Например, в контексте энергосбережения байесовская оптимизация применяется к автоматизированным домашним системам управления энергопотреблением. Алгоритмы из области многоагентных систем используются для прогнозирования работы интеллектуальных сетей, управления биржами обмена энергией и прогнозирования выработки электроэнергии солнечными панелями на крышах зданий. Также были разработаны алгоритмы для защиты биоразнообразия. Например, нейронные сети нашли применение в автоматизации учета диких животных, подходы теории игр используются для борьбы с браконьерством в лесах, а методы оптимизации применяются для распределения ресурсов среды обитания.

За последние десятилетия алгоритмы принятия решений получили распространение в медицине. Такие алгоритмы полезны для прикрепления жителей к оптимально расположенным районным больницам и сопоставления доноров органов с нуждающимися пациентами. Первым применением байесовских сетей, которое мы рассмотрим в этой книге, была диагностика заболеваний. С тех пор байесовские сети широко используются в медицине для диагностики и прогнозирования различных заболеваний. Глубокое обучение совершило революцию в области обработки медицинских изображений, и алгоритмические идеи сыграли важную роль в понимании распространения болезней.

Алгоритмы позволили нам понять принципы роста городских территорий и облегчить их проектирование. Алгоритмы, управляемые данными, широко используются для улучшения общественной инфраструктуры. Например, стохастические процессы применяют для прогнозирования отказов в водопроводах, глубокое обучение улучшило управление дорожным движением, а марковские процессы принятия решений и методы Монте-Карло использовались для улучшения реагирования на чрезвычайные ситуации. Благодаря применению децентрализованных мультиагентных алгоритмов удалось оптимизировать маршруты передвижения, а методы планирования пути использовались для оптимизации доставки товаров. Алгоритмы принятия решений широко применяются в автономных автомобилях и для повышения безопасности самолетов.

Алгоритмы оптимизации решений способны усиливать эффект от действий своих пользователей независимо от их намерений. Если целью пользователя этих алгоритмов, например, является распространение дезинформации во время политических выборов, то процессы оптимизации будут способствовать обману избирателей. Однако аналогичные алгоритмы можно использовать для мониторинга и противодействия распространению ложной информации. Иногда реализация алгоритмов принятия решений может привести к последствиям, которые их пользователи не предполагали[27].

Хотя алгоритмы принятия решений могут принести значительную пользу, существуют и проблемы, связанные с их особенностями и недостатками.

---

[27] Более общие соображения представлены в B. Christian, *The Alignment Problem*. Norton & Company, 2020. См. также D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, D. Mané, *Concrete Problems in AI Safety*, 2016. arXiv: 1606.06565v2.

Алгоритмы, построенные на основе данных, часто страдают от врожденной предвзятости и слепых зон из-за некорректного способа сбора данных. Алгоритмы становятся все более значимой частью нашей жизни. Поэтому важно понимать, как можно снизить риск предвзятости и справедливо распределить преимущества использования алгоритмов. Алгоритмы также могут быть уязвимы для манипулирования злоумышленниками, и очень важно с самого начала разрабатывать алгоритмы, устойчивые к таким атакам. Также важно установить моральные и правовые рамки для предотвращения непреднамеренных последствий и распределения ответственности.

1.6. Краткий обзор содержания книги

Эта книга состоит из пяти частей. В первой части рассматривается проблема рассуждений о неопределенности и целях при принятии одиночных простых решений в конкретный момент времени. Вторая часть расширяет процесс принятия решений до последовательных задач, где мы должны принять последовательность решений в ответ на информацию о результатах наших предыдущих последовательных действий. Третья часть касается *неопределенности модели* – ситуации, когда мы не имеем детерминированной модели и должны выстроить ее через взаимодействие с окружающей средой. Четвертая часть рассматривает *неопределенность состояния*, когда несовершенная информация о восприятии не позволяет нам узнать полное состояние окружающей среды. В заключительной пятой части обсуждаются способы принятия решений с участием нескольких агентов.

1.6.1. Вероятностное рассуждение

Рациональное принятие решений требует рассуждений о нашей неопределенности и целях. Эта часть книги начинается с пояснений о том, как представить неопределенность в виде распределения вероятностей. Реальные задачи требуют понимания распределений по многим переменным. Мы покажем, как строить такие модели распределений, как использовать их для получения выводов и как узнать их параметры и структуру из данных. Затем мы вводим основы *теории полезности* и поясняем, как она формирует основу для рационального принятия решений в условиях неопределенности с помощью принципа максимальной ожидаемой полезности. Далее обсудим, как понятия теории полезности могут быть включены в вероятностные графовые модели для формирования так называемых *сетей принятия решений*.

1.6.2. Многостадийные задачи

Многие важные проблемы требуют принятия ряда последовательных решений. Принцип максимальной ожидаемой полезности применим и в этом случае, но оптимальное принятие последовательных решений требует рассуж-

дений о будущих последовательностях действий и наблюдений. В этой части книги будут рассмотрены проблемы последовательного принятия решений в стохастической среде, где результаты наших действий неопределенные. Мы сосредоточимся на общей формулировке задач последовательного принятия решений, предполагая, что модель известна и что окружающая среда полностью наблюдаема. Позже мы ослабим оба этих предположения. Мы начнем со знакомства с *марковским процессом принятия решений* (MDP) – стандартной математической моделью для многостадийных задач принятия решений. Мы обсудим несколько подходов к поиску точных решений таких задач. Поскольку сложные задачи иногда не позволяют находить эффективные точные решения, мы обсудим ряд методов поиска приближенного решения, а также метод, основанный на прямом поиске в пространстве параметрических стратегий принятия решений. Наконец, разберем способы проверки того, что наши стратегии принятия решений будут работать должным образом при развертывании в реальном мире.

1.6.3. Неопределенность модели

В нашем обсуждении задач последовательного принятия решений до этого момента мы предполагали, что модели перехода и вознаграждения известны. Однако во многих задачах динамика и вознаграждение точно неизвестны, и агент должен научиться действовать на основе опыта. Наблюдая за результатами своих действий в виде переходов состояний и вознаграждений, агент должен научиться выбирать действия, которые максимизируют его долгосрочное накопление вознаграждений. Решение задач, в которых присутствует неопределенность модели, является предметом обучения с подкреплением и рассматривается в этой части книги. Мы обсудим несколько проблем, связанных с устранением неопределенности модели. Во-первых, агент должен найти тонкий баланс между поиском новых факторов окружающей среды и использованием знаний, полученных в результате опыта. Во-вторых, вознаграждение может быть получено спустя долгое время после того, как важные решения были приняты, так что более поздние вознаграждения должны быть правильно соотнесены с ранними решениями. В-третьих, агент должен уметь строить обобщения на основе ограниченного опыта. Мы рассмотрим теорию и некоторые из ключевых алгоритмов, соответствующих этим требованиям.

1.6.4. Неопределенность состояния

В этой части книги мы расширим понятие неопределенности, включив в него состояние. Вместо точного наблюдения за состоянием мы получаем наблюдения, имеющие только вероятностную связь с состоянием. Такие задачи можно смоделировать как *частично наблюдаемый марковский процесс принятия решений* (partially observable Markov decision process, POMDP). Общий подход к реализации POMDP включает в себя вывод распределения по убеждениям, исходя из состояния на текущем временном шаге, а затем применение стратегии,

которая сопоставляет убеждения с действиями. Эта часть начинается с обсуждения того, как обновить наше распределение убеждений с учетом прошлой последовательности наблюдений и действий. Затем обсуждаются различные точные и приближенные методы решения POMDP.

1.6.5. Мультиагентные системы

До этого момента в нашей среде за принятие решений отвечал только один агент. В пятой части книги мы расширяем понятия и методы из предыдущих четырех частей на систему из нескольких агентов, обсуждая проблемы, возникающие из-за неопределенности взаимодействия между агентами. Мы начнем с рассмотрения простых игр, в которых группа агентов одновременно выбирает действие. Результатом является индивидуальное вознаграждение для каждого агента на основе комбинированного совместного действия. *Марковская игра* (Markov game, MG) представляет собой обобщение как простых игр для нескольких состояний, так и MDP для нескольких агентов. Следовательно, агенты выбирают действия, которые могут стохастически изменить состояние общей среды. Из-за неопределенности стратегии других агентов алгоритмам MG приходится использовать обучение с подкреплением. *Частично наблюдаемая марковская игра* (partially observable Markov game, POMG) вводит неопределенность состояния, дополнительно обобщая MG и POMDP, поскольку теперь агенты получают только зашумленные локальные наблюдения. *Децентрализованный частично наблюдаемый марковский процесс принятия решений* (Dec-POMDP) фокусирует POMG на совместной многоагентной команде, где агенты получают общее вознаграждение. В этой части книги представлены четыре вышеупомянутые категории задач и рассмотрены точные и приближенные алгоритмы их решения.

Часть I

Вероятностные рассуждения

Рациональное принятие решений требует ясного понимания неопределенности, с которой мы сталкиваемся, и целей, к которым мы стремимся. Неопределенность возникает из-за практических и теоретических ограничений нашей способности предсказывать будущие события. Например, точное предсказание того, как человек-оператор отреагирует на совет системы поддержки принятия решений, потребует, среди прочего, наличия подробной модели работы человеческого мозга. Даже траектории движения спутников бывает трудно предсказать. Хотя ньютоновская физика позволяет предсказывать их с высокой точностью, спонтанные отказы двигателей ориентации могут привести к большим отклонениям от номинального пути, и даже небольшие неточности могут со временем накапливаться. Для достижения своих целей надежная система принятия решений должна учитывать различные источники неопределенности в текущем состоянии мира и будущих событиях. Эта часть книги начинается с рассказа о том, как представляют неопределенность с помощью распределений вероятностей. Для решения реальных задач необходимо знать распределения вероятностей по многим переменным. Мы обсудим, как создавать модели распределения вероятностей, использовать их для выводов и изучать их параметры и структуру на основе данных. Далее мы вводим основы теории полезности и показываем, как она формирует основу для рационального принятия решений в условиях неопределенности. Теорию полезности можно применять в упомянутых ранее вероятностных графовых моделях для формирования так называемых сетей принятия решений. Мы сосредоточимся на одношаговых решениях, оставив обсуждение проблем последовательных решений для следующей части книги.

2 Формальное представление неопределенности

Вычислительный подход к неопределенности требует формального представления. В этой главе пойдет речь о том, как представлять неопределенность[1]. Мы начнем с понятия *степени доверия* (degree of belief) и покажем, как набор аксиом приводит нас к использованию распределения вероятностей для количественной оценки неопределенности[2]. Мы рассмотрим несколько полезных форм распределений как по дискретным, так и по непрерывным переменным. Поскольку многие важные задачи связаны с распределением вероятностей по большому количеству переменных, мы обсудим способ эффективного представления совместных распределений, основанный на факторе независимости между переменными.

2.1. Степени доверия и вероятности

В задачах, связанных с неопределенностью, важно уметь сравнивать правдоподобие различных утверждений. Допустим, нам нужно выразить свое представление о том, что утверждение A более правдоподобно, чем утверждение B. Если A представляет собой «отказ исполнительного механизма», а B представляет «отказ датчика», мы можем написать $A \succ B$. Используя это основное отношение $\succ$, мы можем определить несколько других отношений:

$A \prec B$ тогда и только тогда, когда $B \succ A$, (2.1)

$A \sim B$ тогда и только тогда, когда ни $A \succ B$, ни $B \succ A$, (2.2)

$A \succcurlyeq B$ тогда и только тогда, когда $A \succ B$ или $A \sim B$, (2.3)

$A \preccurlyeq B$ тогда и только тогда, когда $B \succ A$ или $A \sim B$. (2.4)

Следует сделать некоторые предположения об отношениях, установленных операторами $\succ$, $\sim$ и $\prec$. Предположение об *универсальной сравнимости* требу-

---

[1] Детальное описание различных подходов к представлению неопределенности дано в монографии F. Cuzzolin, *The Geometry of Uncertainty.* Springer, 2021.

[2] Для более полного знакомства с темой см. E. T. Jaynes, *Probability Theory: The Logic of Science*. Cambridge University Press, 2003.

ет выполнения ровно одного из следующих условий: $A \succ B$, $A \sim B$ или $A \prec B$. Предположение о *транзитивности* требует, чтобы если $A \succcurlyeq B$ и $B \succcurlyeq C$, то $A \succcurlyeq C$. Предположения об универсальной сравнимости и транзитивности дают нам возможность представить правдоподобие с помощью функции P, принимающей вещественные значения и обладающей следующими двумя свойствами[3]:

$P(A) > P(B)$ тогда и только тогда, когда $A \succ B$; (2.5)

$P(A) = P(B)$ тогда и только тогда, когда $A \sim B$. (2.6)

Если мы сделаем ряд дополнительных предположений[4] о форме P, то сможем показать, что P должна удовлетворять основным аксиомам вероятности (приложение А.2). Если мы абсолютно уверены в A, то $P(A) = 1$. Если мы считаем, что A невозможно, то $P(A) = 0$. Неуверенность в истинности A представлена значениями между двумя этими экстремумами. Следовательно, *вероятностная мера* (probability mass) должна лежать между 0 и 1, при этом $0 \leqslant P(A) \leqslant 1$.

2.2. Распределения вероятностей

Распределение вероятностей присваивает вероятности различным исходам[5]. Существуют разные способы представления распределений вероятностей в зависимости от того, о каких исходах идет речь – дискретных или непрерывных.

2.2.1. Дискретные распределения вероятностей

Дискретное распределение вероятностей – это распределение по дискретному набору значений. Мы можем представить такое распределение как *функцию вероятности*, которая присваивает вероятность каждому возможному присвоению своей входной переменной значения. Например, предположим, что у нас есть переменная X, которая может принимать одно из n значений: $1, \ldots, n$, или, используя запись с двоеточием, $1{:}n$[6]. Распределение, связанное с X, определяет n вероятностей различных присвоений значений этой переменной, в частно-

---

[3] См. E. T. Jaynes, *Probability Theory: The Logic of Science.* Cambridge University Press, 2003.

[4] Аксиоматизация субъективной вероятности дана в работе P. C. Fishburn, *The Axioms of Subjective Probability*, Statistical Science, vol. 1, no. 3, pp. 335–345, 1986. Более поздняя аксиоматизация содержится в M. J. Dupré, F. J. Tipler, *New Axioms for Rigorous Bayesian Probability*, Bayesian Analysis, vol. 4, нет. 3, стр. 599–606, 2009.

[5] Хорошее введение в теорию вероятностей дано в D. P. Bertsekas, J. N. Tsitsiklis, *Introduction to Probability.* Athena Scientific, 2002.

[6] Мы будем часто использовать это двоеточие для компактности. В других книгах иногда используется обозначение $[1 \ldots n]$ для целых интервалов от 1 до n. Мы также будем использовать это обозначение с двоеточием для индексов векторов и матриц. Например, $x_{1:n}$ представляет $x_1, \ldots, x_n$. Обозначение двоеточия иногда используется в языках программирования, таких как Julia и MATLAB.

сти $P(X = 1), \ldots, P(X = n)$. На рис. 2.1 показан пример дискретного распределения вероятностей.

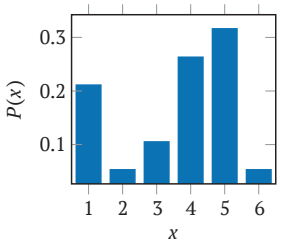


Рис. 2.1. Функция вероятности для распределения на интервале 1:6

Существуют ограничения на вероятностные меры, связанные с дискретными распределениями. Их значения должны в сумме равняться 1:

$$\sum_{i=1}^{n} P(X = i) = 1. \tag{2.7}$$

Для удобства записи мы будем использовать строчные буквы и надстрочные индексы в качестве сокращений при обсуждении присвоения значений переменным. Например, $P(x^3)$ – это сокращенная форма записи $P(X = 3)$. Если X – *двоичная переменная*, она может принимать значение *true* (истина) или *false* (ложь)[7]. Мы будем использовать 0 для представления *false* и 1 для представления *true*. Например, мы используем запись $P(x^0)$ для представления вероятности того, что X ложно.

Параметры распределения определяют вероятности, связанные с различными значениями дискретной переменной. Например, если мы используем X для представления исхода броска шестигранной кости, то мы получим $P(x^1) = \theta_1$, …, $P(x^6) = \theta_6$, где $\theta_{1:6}$ обозначает шесть параметров распределения. Однако нам достаточно пяти *независимых* параметров, чтобы однозначно указать распределение по исходам, потому что мы знаем, что сумма распределения должна равняться 1.

2.2.2. Непрерывные распределения вероятностей

Непрерывное распределение вероятностей – это распределение по непрерывному набору значений. Представление распределения по непрерывной переменной выглядит несколько сложнее, чем для дискретной переменной. Например, во многих непрерывных распределениях вероятность того, что переменная примет конкретное значение, бесконечно мала. Одним из способов представления непрерывного распределения вероятностей является использование *функции плотности вероятности* (рис. 2.2), обозначаемой строчными буквами. Если $p(x)$ – функция плотности вероятности по X, то $p(x)dx$ – это вероятность

[7] Julia, как и многие другие языки программирования, аналогичным образом обрабатывает логические значения как 0 и 1 в числовых операциях.

того, что X попадает в интервал $(x, x + dx)$ при $dx \to 0$. Подобно тому, как вероятностные меры, связанные с дискретным распределением, должны в сумме давать 1, интеграл функции плотности вероятности $p(x)$ должен быть равен 1:

$$\int_{-\infty}^{\infty} p(x)dx = 1. \tag{2.8}$$

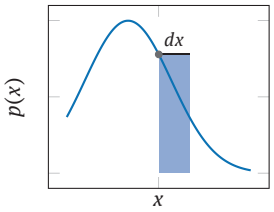


Рис. 2.2. Функции плотности вероятности используются для представления непрерывных распределений вероятностей. Если $p(x)$ – плотность вероятности, то $p(x)dx$, обозначенная площадью синего прямоугольника, – это вероятность того, что выборка из случайной величины попадает в интервал $(x, x + dx)$ при $dx \to 0$

Другой способ представить непрерывное распределение – использовать *кумулятивную функцию распределения* (рис. 2.3), которая определяет вероятностную меру, связанную со значениями ниже некоторого порога. Если у нас есть кумулятивная функция распределения P, связанная с переменной X, то $P(x)$ представляет вероятностную меру, связанную с X, принимающей значение, меньшее или равное x. Кумулятивная функция распределения может быть определена через функцию плотности вероятности p следующим образом:

$$\mathrm{cdf}_X(x) = P(X \leq x) = \int_{-\infty}^{\infty} p(x')dx'. \tag{2.9}$$

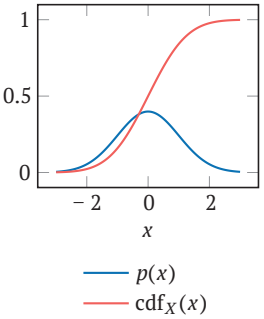


Рис. 2.3. Функция плотности вероятности и кумулятивная функция распределения для стандартного распределения Гаусса

С кумулятивной функцией распределения связана *квантильная функция*, также называемая *обратной кумулятивной функцией распределения* (рис. 2.4). Значение $\mathrm{quantile}_X(\alpha)$ – это значение x, такое что $P(X \leqslant x) = \alpha$. Другими словами, квантильная функция возвращает минимальное значение x, кумулятивное значение распределения которого больше или равно α. Разумеется, $0 < \alpha < 1$.

Существует множество различных параметризованных семейств распределений. Мы рассматриваем некоторые из них в приложении B. Простое семейство распределений – это равномерное распределение $\mathcal{U}(a, b)$, при котором плотность вероятности равномерно распределена между a и b и равна нулю в других местах. Следовательно, функция плотности вероятности равна $p(x) =$

$1/(b-a)$ для x, лежащего в интервале $[a, b]$. Для представления плотности в точке x мы можем использовать запись $\mathcal{U}(x|a, b)$[8]. *Носителем* (несущим множеством) распределения является набор значений, которым присвоена ненулевая плотность. В случае $\mathcal{U}(a, b)$ носителем является интервал $[a, b]$ (пример 2.1).

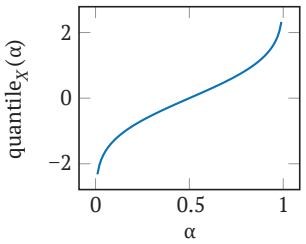


Рис. 2.4. Квантильная функция для стандартного распределения Гаусса

Пример 2.1. Пример равномерного распределения с нижней границей 0 и верхней границей 10

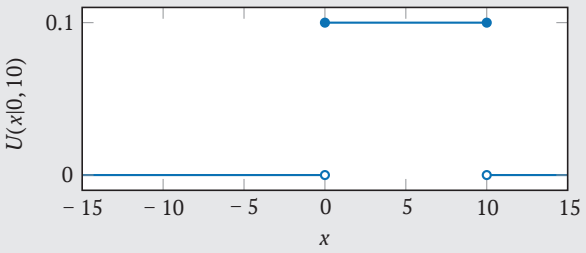
Равномерное распределение $\mathcal{U}(0, 10)$ присваивает равную вероятность всем значениям в диапазоне [0, 10] со следующей функцией плотности вероятности:

$$\mathcal{U}(x \mid 0,10) = \begin{cases} 1/10, & \text{если } 0 \le x \le 10 \\ 0 & \text{для остальных значений} \end{cases}. \tag{2.10}$$

Вероятность того, что случайная выборка из этого распределения равна константе π, практически нулевая. Однако мы можем определить ненулевые вероятности для выборок, находящихся в пределах некоторого интервала, допустим [3, 5]. Вероятность того, что выборка находится между 3 и 5, с учетом построенного здесь распределения, равна:

$$\int_3^5 \mathcal{U}(x \mid 0,10)dx = \frac{5-3}{10} = \frac{1}{5}. \tag{2.11}$$

Носителем этого распределения является интервал [0,10].



[8] В некоторых публикациях для разделения параметров распределения используется точка с запятой. Например, можно также написать $\mathcal{U}(x; a, b)$.

Другим распространенным распределением непрерывных переменных является распределение Гаусса (также называемое *нормальным распределением*). Распределение Гаусса характеризуется средним значением μ и дисперсией σ^2:

$$p(x) = \mathcal{N}(x|\mu, \sigma^2). \tag{2.12}$$

Здесь σ – *стандартное отклонение*, представляющее собой квадратный корень из дисперсии. Дисперсию также часто обозначают буквой ν. Мы используем запись $\mathcal{N}(\mu, \sigma^2)$ как обозначение гауссова распределения с параметрами μ и σ^2 и $\mathcal{N}(x|\mu, \sigma^2)$, представляющего плотность вероятности в точке x:

$$\mathcal{N}(x|\mu,\sigma^2) = \frac{1}{\sigma}\varphi\left(\frac{x-\mu}{\sigma}\right), \tag{2.13}$$

где φ – *функция плотности стандартного нормального распределения вероятностей*:

$$\varphi(x) = \frac{1}{\sqrt{2\pi}}\exp\left(-\frac{x^2}{2}\right). \tag{2.14}$$

В приложении В показаны графики функций плотности нормального распределения вероятностей с различными параметрами.

Хотя распределение Гаусса широко распространено и удобно в использовании, поскольку оно определяется только двумя параметрами и упрощает вычисления и вывод, оно имеет некоторые ограничения. В частности, оно присваивает ненулевую вероятность большим положительным и отрицательным значениям, что может не соответствовать реальной величине, которую мы пытаемся смоделировать. Например, очевидно, что не следует присваивать ненулевые вероятности самолетам, летящим под землей или за пределами земной атмосферы. Мы можем использовать *усеченное распределение Гаусса* (рис. 2.5), чтобы ограничить опорный интервал распределения, т. е. диапазон значений, которым присвоены ненулевые вероятности. Функция плотности вероятности определяется выражением

$$\mathcal{N}(x|\mu,\sigma^2,a,b) = \frac{\frac{1}{\sigma}\varphi\left(\frac{x-\mu}{\sigma}\right)}{\Phi\left(\frac{b-\mu}{\sigma}\right) - \Phi\left(\frac{a-\mu}{\sigma}\right)}, \tag{2.15}$$

когда x находится в интервале (a, b).

Функция Φ представляет собой *стандартную нормальную кумулятивную функцию распределения*, определяемую выражением

$$\Phi(x) = \int_{-\infty}^{\infty}\varphi(x')dx'. \tag{2.16}$$

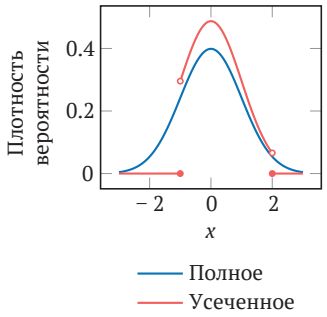


Рис. 2.5. Функции плотности вероятности для единичного распределения Гаусса и того же распределения, усеченного между –1 и 2

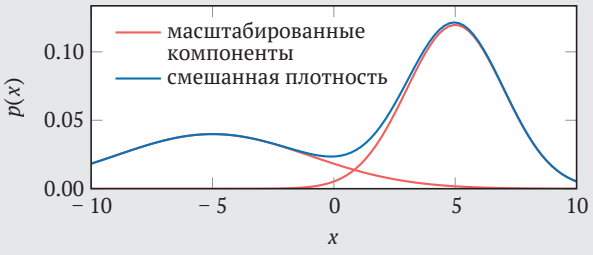
Распределение Гаусса является *унимодальным*, т. е. в распределении есть точка, относительно которой плотность увеличивается с одной стороны и уменьшается с другой. Существуют также различные способы представления непрерывных *мультимодальных* распределений. Один из способов – использовать *смешанную модель* (mixture model), которая представляет собой смесь нескольких распределений. Мы смешиваем набор одномодальных распределений, чтобы получить мультимодальное распределение. *Смешанная модель Гаусса* (Gaussian mixture model) – это модель, которая представляет собой просто средневзвешенное значение различных распределений Гаусса. Параметры такой модели состоят из параметров компонентов гауссова распределения $\mu_{1:n}$, $\sigma^2_{1:n}$, а также их веса $\rho_{1:n}$. Плотность определяется выражением

$$p(x|\mu_{1:n},\sigma^2_{1:n},\rho_{1:n}) = \sum_{i=1}^{n}\rho_1\mathcal{N}(x|\mu_i,\sigma_i^2), \tag{2.17}$$

где веса должны в сумме равняться 1. В примере 2.2 показана смешанная модель Гаусса с двумя компонентами.

Пример 2.2. Пример смешанной модели Гаусса

Создадим смешанную модель Гаусса с компонентами $\mu_1 = 5$, $\sigma_1 = 2$ и $\mu_2 = -5$, $\sigma_2 = 4$, с весами $\rho_1 = 0.6$ и $\rho_2 = 0.4$ соответственно. На рисунке ниже изображены графики плотности вероятностей двух компонентов модели, масштабированные по их весу:



Другой подход к представлению мультимодальных непрерывных распределений заключается в дискретизации. Например, мы можем представить распределение по непрерывной переменной как *кусочно-равномерную аппроксимацию*. Плотность определяется в границах промежутка, опирающегося на отрезок («кусок»), и с каждым промежутком связана вероятностная мера. Такое кусочно-равномерное распределение является разновидностью смешанной модели, в которой компоненты представляют собой равномерные распределения.

2.3. Совместные распределения

Совместное распределение – это распределение вероятностей по нескольким переменным. Распределение по одной переменной называется *одномерным*, а распределение по нескольким переменным называется *многомерным*. Если у нас есть совместное распределение по двум дискретным переменным X и Y, то $P(x, y)$ обозначает вероятность того, что одновременно и $X = x$, и $Y = y$.

Из совместного распределения мы можем вычислить *маргинальное (частное) распределение* переменной или набора переменных, суммируя все другие переменные в соответствии с так называемым *законом полной вероятности*[9]:

$$P(x) = \sum_y P(x, y). \tag{2.18}$$

Запомните этот прием – мы будем использовать его на протяжении всей книги.

Принятие решений в реальном мире часто требует рассуждений о совместных распределениях, включающих множество переменных. Иногда существуют сложные отношения между переменными, которые важно уметь представить формально. Существуют разные способы представления совместных распределений в зависимости от того, включают ли переменные дискретные или непрерывные значения.

2.3.1. Дискретные совместные распределения

Если переменные дискретны, совместное распределение может быть представлено таблицей наподобие табл. 2.1. В ней перечислены все возможные сочетания значений трех двоичных переменных. Каждая переменная может принимать значение только 0 или 1, что дает $2^3 = 8$ возможных назначений. Как и в случае с другими дискретными распределениями, вероятности в таблице должны в сумме равняться 1. Из этого следует, что хотя в таблице восемь

[9] Если наше распределение непрерывно, то при маргинализации мы интегрируем другие переменные. Например: $p(x) = \int p(x, y)dy$.

записей, только семь из них независимы. Если θ_i представляет вероятность в i-й строке таблицы, то нам достаточно параметров $\theta_1, \ldots, \theta_7$, потому что мы знаем, что $\theta_8 = 1 - (\theta_1 + \ldots + \theta_7)$.

X	Y	Z	$P(X, Y, Z)$
0	0	0	0.08
0	0	1	0.31
0	1	0	0.09
0	1	1	0.37
1	0	0	0.01
1	0	1	0.05
1	1	0	0.02
1	1	1	0.07

Таблица 2.1. Пример дискретного совместного распределения с участием бинарных переменных X, Y и Z

Если у нас есть n бинарных переменных, то нам нужно как минимум $2^n - 1$ независимых параметров, чтобы задать совместное распределение. Этот экспоненциальный рост числа параметров затрудняет хранение распределения в памяти компьютера. В некоторых случаях мы можем предположить, что наши переменные независимы, т. е. реализация одной переменной не влияет на распределение вероятностей другой. Если X и Y независимы, что иногда записывают как $X \perp Y$, то мы знаем, что $P(x, y) = P(x)P(y)$ для всех x и y. Предположим, что у нас есть двоичные переменные $X_1, \ldots, X_n$, которые не зависят друг от друга, в результате чего $P(x_{1:n}) = \prod_i P(x_i)$. Это разложение на множители позволяет нам представить совместное распределение, используя всего n независимых параметров вместо $2^n - 1$, необходимых, когда мы не можем предположить независимость (табл. 2.2). Предположение о независимости переменных может привести к огромной экономии с точки зрения сложности представления, но часто эта предпосылка оказывается ошибочной.

X	$P(X)$
0	0.85
1	0.15

Y	$P(Y)$
0	0.45
1	0.55

Z	$P(Z)$
0	0.20
1	0.80

Таблица 2.2. Если мы знаем, что переменные в табл. 2.1 независимы, мы можем представить $P(x, y, z)$, используя произведение $P(x)P(y)P(z)$. Такое представление требует только одного параметра для каждого из трех одномерных распределений

Мы можем представить совместное распределение в виде факторов. *Фактор* φ – это функция, которая ставит в соответствие каждому возможному набору значений некого множества переменных действительное число (значение фактора). Значения фактора, представляющие распределение вероятностей, должны быть неотрицательными. Фактор с неотрицательными значениями можно нормализовать так, чтобы он представлял собой распределение вероятностей. В алгоритме 2.1 реализованы дискретные факторы, а пример 2.3 демонстрирует, как они работают.

Другой подход к сокращению памяти, необходимой для представления совместных распределений с повторяющимися значениями, заключается в использовании *дерева решений*. Дерево решений, включающее три дискретные

переменные, показано в примере 2.4. Хотя в этом примере экономия по количеству параметров выглядит незначительной, она может стать весьма существенной, когда имеется много переменных и много повторяющихся значений.

Алгоритм 2.1. Типы и функции, относящиеся к работе с факторами в случае набора дискретных переменных. У переменной есть имя (представленное в виде символа) и целочисленное значение. *Назначение* (assignment) – это сопоставление имен переменных с целыми числами от 1 до m. Фактор определяется через *факторную таблицу*, которая присваивает значения различным назначениям, включающим множество переменных, и представляет собой сопоставление назначений с действительными значениями. Это сопоставление называется *словарем*. Любые назначения, не содержащиеся в словаре, устанавливаются равными 0. В этот блок кода также включены некоторые служебные функции для возврата имен переменных, связанных с фактором, выбора подмножества назначения, перечисления возможных назначений и нормализации факторов. Как сказано в приложении G.3.3, `product` выполняет декартово произведение набора коллекций. Он импортируется из `Base.Iterators`

```
struct Variable
    name::Symbol
    r::Int # количество возможных значений
end

const Assignment = Dict{Symbol,Int}
const FactorTable = Dict{Assignment,Float64}

struct Factor
    vars::Vector{Variable}
    table::FactorTable
end

variablenames(φ::Factor) = [var.name for var in φ.vars]

select(a::Assignment, varnames::Vector{Symbol}) =
    Assignment(n=>a[n] for n in varnames)

function assignments(vars::AbstractVector{Variable})
    names = [var.name for var in vars]
    return vec([Assignment(n=>v for (n,v) in zip(names, values))
                for values in product((1:v.r for v in vars)...)])
end

function normalize!(φ::Factor)
    z = sum(p for (a,p) in φ.table)
    for (a,p) in φ.table
       φ.table[a] = p/z
    end
    return φ
end
```

Пример 2.3. Построение дискретного фактора. *При построении таблицы факторов с использованием именованных кортежей используются функции полезности, определенные в приложении G.5*

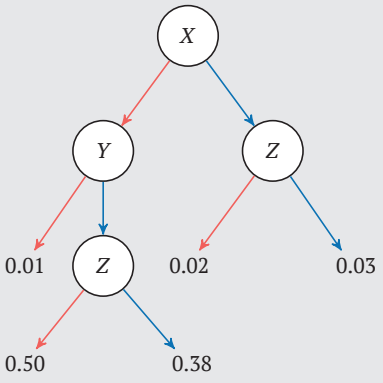
Мы можем создать экземпляр таблицы из табл. 2.1, используя тип `Factor` и следующий код:

```
# требуются вспомогательные функции из приложения G.5
X = Variable(:x, 2)
Y = Variable(:y, 2)
Z = Variable(:z, 2)
ϕ = Factor([X, Y, Z], FactorTable(
    (x=1, y=1, z=1) => 0.08, (x=1, y=1, z=2) => 0.31,
    (x=1, y=2, z=1) => 0.09, (x=1, y=2, z=2) => 0.37,
    (x=2, y=1, z=1) => 0.01, (x=2, y=1, z=2) => 0.05,
    (x=2, y=2, z=1) => 0.02, (x=2, y=2, z=2) => 0.07,
))
```

Пример 2.4. Дерево решений может быть более эффективным представлением совместного распределения, чем таблица

Предположим, у нас есть следующая таблица, представляющая совместное распределение вероятностей. Мы можем использовать изображенное справа от нее дерево решений, чтобы более компактно представить значения в таблице. Красные стрелки соответствуют решению, когда переменная равна 0, а синие – когда переменная равна 1. Вместо восьми вероятностей мы сохраняем только пять вместе с представлением дерева.

X	Y	Z	$P(X, Y, Z)$
0	0	0	0.01
0	0	1	0.01
0	1	0	0.50
0	1	1	0.38
1	0	0	0.02
1	0	1	0.03
1	1	0	0.02
1	1	1	0.03



2.3.2. Непрерывное совместное распределение

Мы также можем определить совместные распределения по непрерывным переменным. Довольно простым распределением является *многомерное рав-*

номерное распределение, которое задает постоянную плотность вероятности везде, где есть несущее множество. Мы можем использовать $\mathcal{U}(\mathbf{a}, \mathbf{b})$ для представления равномерного распределения по *рамке* (box), которая образована декартовым произведением интервалов, где i-й интервал равен $[a_i, b_i]$. Это семейство равномерных распределений представляет собой особый тип многомерного произведения распределений, а именно распределение, определяемое через произведение одномерных распределений. В этом случае

$$P(x) = \sum_y P(x, y). \tag{2.19}$$

Мы можем создать смешанную модель из взвешенного набора многомерных равномерных распределений точно так же, как и с одномерными распределениями. Если у нас есть совместное распределение по n переменным и k компонентам смеси, нам нужно определить $k(2n + 1) - 1$ независимых параметров. Для каждой из k компонент нам необходимо определить верхнюю и нижнюю границы для каждой из переменных, а также их веса. Мы вычитаем 1, потому что в сумме веса должны быть равны 1. На рис. 2.6 показан пример, который может быть представлен пятью компонентами.

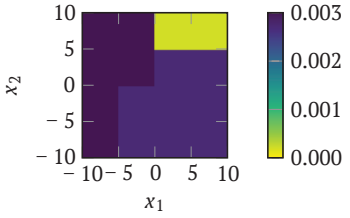


Рис. 2.6. Функция плотности для смеси многомерных равномерных распределений

Функции равномерной плотности часто получают путем независимой дискретизации каждой из переменных. При дискретизации формируется набор границ интервалов для каждой переменной. Эти границы образуют сетку из переменных. Затем мы сопоставляем равномерную плотность вероятности с каждой ячейкой сетки. Границы интервалов не обязательно должны быть распределены равномерно. В некоторых случаях желательно иметь увеличенное разрешение около определенных значений. С разными переменными могут быть связаны разные границы интервалов. Если имеется n переменных и m интервалов для каждой переменной, то нам потребуется $m^n - 1$ независимый параметр, чтобы определить распределение, – в дополнение к значениям, определяющим границы интервалов.

В некоторых случаях более эффективным с точки зрения использования памяти может быть представление непрерывного совместного распределения в виде дерева решений способом, подобным тому, что мы обсуждали для дискретных совместных распределений. Внутренние узлы сравнивают переменные с пороговыми значениями, а листовые (конечные) узлы представляют собой значения плотности. На рис. 2.7 показано дерево решений, соответствующее функции плотности на рис. 2.6.

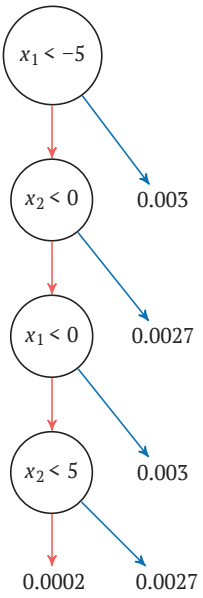


Рис. 2.7. Пример дерева решений, представляющего кусочно-линейную совместную плотность вероятности, определенную по x_1 и x_2 на интервале $[-10, 10]^2$

Другим полезным распределением является *многомерное распределение Гаусса* с функцией плотности

$$\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{n/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathsf{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right), \tag{2.20}$$

где x принадлежит $\mathbb{R}$, $\boldsymbol{\mu}$ – *вектор математического ожидания* (mean vector), а $\boldsymbol{\Sigma}$ – *матрица ковариаций* (матрица дисперсий). Приведенная здесь функция плотности требует, чтобы $\boldsymbol{\Sigma}$ была *неотрицательно определенной*[10]. Количество независимых параметров равно $n + (n + 1)n/2$, количество компонентов в $\boldsymbol{\mu}$ добавляется к количеству компонентов в верхнем треугольнике матрицы $\boldsymbol{\Sigma}$[11]. В приложении В показаны графики различных многомерных функций гауссова распределения. Мы также можем определить *многомерные смешанные модели Гаусса*. На рис. 2.8 показан пример с тремя компонентами.

Если у нас есть многомерная гауссиана со всеми независимыми переменными, то матрица ковариаций $\boldsymbol{\Sigma}$ является диагональной только с n независимыми параметрами. Фактически мы можем записать функцию плотности как произведение одномерных гауссовых плотностей:

$$\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \prod_i \mathcal{N}(x_i|\mu_i, \Sigma_{ii}). \tag{2.21}$$

---

[10] Это определение рассмотрено в приложении А.5.

[11] Если мы знаем параметры в верхнем треугольнике E, мы знаем параметры и в нижнем треугольнике, потому что E симметрично.

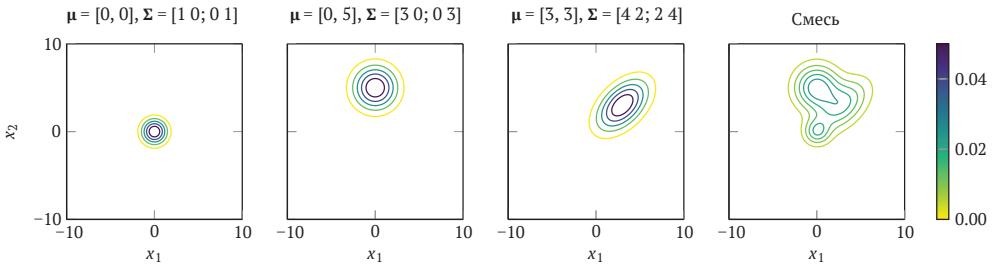


Рис. 2.8. Многомерная смешанная модель Гаусса с тремя компонентами. Компоненты смешиваются с весами 0.1, 0.5 и 0.4 соответственно

2.4. Условные распределения

Предыдущий раздел был основан на идее независимости переменных, которая помогает уменьшить количество параметров, используемых для определения совместного распределения. Однако, как уже упоминалось, независимость переменных может быть слишком сильным предположением. В этом разделе будет представлена идея *условной независимости*, которая может помочь уменьшить количество независимых параметров, не делая при этом столь строгих предположений. Прежде чем обсуждать условную независимость, мы сначала введем понятие *условного распределения*, которое представляет собой распределение по переменной при заданном значении одной или нескольких других.

Определение *условной вероятности* гласит, что

$$P(x|y) = \frac{P(x,y)}{P(y)}, \tag{2.22}$$

где $P(x|y)$ читается как «вероятность x при заданном y». В некоторых случаях y принято называть *свидетельством* (evidence), или *наблюдением*.

Поскольку условное распределение вероятностей – это распределение вероятностей по одной или нескольким переменным с учетом некоторых известных данных, мы знаем, что

$$\sum_x P(x|y) = 1 \tag{2.23}$$

для дискретного X. Если X непрерывно, то единице равен интеграл.

Мы можем подставить определение условной вероятности в уравнение (2.18), чтобы получить несколько иную форму закона полной вероятности:

$$P(x) = \sum_y P(x|y)P(y) \tag{2.24}$$

для дискретного распределения.

Еще одним полезным соотношением, которое следует из определения условной вероятности, является *правило Байеса*[12]:

$$P(x|y) = \frac{P(y|x)P(x)}{P(y)}. \tag{2.25}$$

Если у нас есть представление условного распределения $P(y|x)$, мы можем применить правило Байеса и поменять местами y и x, чтобы получить условное распределение $P(x|y)$.

Далее мы обсудим различные способы представления условных распределений вероятностей по дискретным и непрерывным переменным.

2.4.1. Дискретные модели условных распределений

Условное распределение вероятностей по дискретным переменным может быть представлено с помощью таблицы. По сути, мы можем воспользоваться тем же дискретным факторным представлением, которое использовали в разделе 2.3.1 для совместных распределений. В табл. 2.3 показан пример условного распределения $P(X|Y, Z)$ со всеми двоичными переменными. В отличие от сводной таблицы (например, табл. 2.1), столбец, содержащий вероятности, не обязательно должен в сумме давать 1. Однако если мы суммируем вероятности, которые согласуются с нашими условиями, мы должны получить 1. Например, при условии y^0 и z^0 мы имеем

$$P(x^0|y^0, z^0) + P(x^1|y^0, z^0) = 0.08 + 0.92 = 1. \tag{2.26}$$

Таблицы условной вероятности могут стать довольно большими. Если бы мы создали таблицу, подобную табл. 2.3, в которой все переменные могут принимать m значений при обусловливании n переменных, то в ней было бы m^{n+1} строк. Однако поскольку m значений переменной, которую мы обусловливаем, должны в сумме равняться 1, имеется ровно $(m - 1)m^n$ независимых параметров. По-прежнему наблюдается экспоненциальный рост числа переменных, которые служат условием. Когда в таблице условной вероятности много повторяющихся значений, дерево решений (представленное в разделе 2.3.1) может быть более эффективным представлением.

---

[12] Названа в честь английского статистика и пресвитерианского священника Томаса Байеса (ок. 1701–1761), который сформулировал эту теорему. Историческое подтверждение представлено в S. B. McGrayne, *The Theory That Would Not Die.* Yale University Press, 2011.

X	Y	Z	$P(X\|Y, Z)$
0	0	0	0.08
0	0	1	0.15
0	1	0	0.05
0	1	1	0.10
1	0	0	0.92
1	0	1	0.85
1	1	0	0.95
1	1	1	0.90

Таблица 2.3. Пример табличного представления условного распределения с использованием двоичных переменных *X*, *Y* и *Z*

2.4.2. Условные модели Гаусса

Условная модель Гаусса может представлять распределение по непрерывной переменной с учетом одной или нескольких дискретных переменных. Например, если у нас есть непрерывная переменная X и дискретная переменная Y со значениями $1{:}n$, мы можем определить условную гауссову модель следующим образом[13]:

$$p(x|y) = \begin{cases} \mathcal{N}(x|\mu_1, \sigma_1^2), & \text{если } y^1 \\ \vdots & \\ \mathcal{N}(x|\mu_n, \sigma_n^2), & \text{если } y^n \end{cases} \tag{2.27}$$

с вектором параметров $\boldsymbol{\theta} = [\mu_{1:n}, \sigma_{1:n}]$. Все $2n$ этих параметров могут изменяться независимо. Если мы хотим объявить условия по нескольким дискретным переменным, нам просто нужно добавить больше случаев и связанных параметров.

2.4.3. Линейные модели Гаусса

Линейная модель Гаусса $P(X|Y)$ представляет распределение по непрерывной переменной X как нормальное распределение со средним значением, являющимся линейной функцией значения непрерывной переменной Y. Условная функция плотности имеет вид

$$p(x|y) = \mathcal{N}(x|my + b, \sigma^2) \tag{2.28}$$

с параметрами $\boldsymbol{\theta} = [m, b, \sigma]$. Среднее значение является линейной функцией y, определяемой параметрами m и b. Дисперсия постоянная. На рис. 2.9 показан пример такой модели.

---

[13] Это определение относится к смеси одномерных гауссианов, но данное понятие можно легко обобщить на смесь многомерных гауссианов.

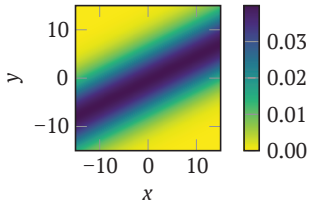


Рис. 2.9. Линейная модель Гаусса $p(x|y) = \mathcal{N}(x|2y + 1, 10^2)$

2.4.4. Условные линейные модели Гаусса

Условная линейная модель Гаусса сочетает в себе идеи условной и линейной моделей Гаусса и дает возможность обусловливать непрерывную переменную как дискретными, так и непрерывными переменными. Предположим, что мы хотим представить $p(X|Y, Z)$, где X и Y непрерывны, а Z принимает дискретные значения из интервала $1{:}n$. Тогда условная функция плотности может быть записана в следующем виде:

$$p(x|y,z) = \begin{cases} \mathcal{N}(x|m_1 y + b_1, \sigma_1^2), & \text{если } z^1 \\ \vdots & \\ \mathcal{N}(x|m_n y + b_n, \sigma_n^2), & \text{если } z^n \end{cases}. \tag{2.29}$$

Здесь вектор параметров $\boldsymbol{\theta} = [m_{1:n}, b_{1:n}, \sigma_{1:n}]$ состоит из $3n$ компонент.

2.4.5. Сигмовидные модели

Для представления распределения по бинарной переменной, зависящей от непрерывной переменной, можно использовать *сигмовидную модель*[14]. Допустим, нам нужно представить распределение $P(x^1|y)$, где x – бинарная переменная, а y – непрерывная. Конечно, мы могли бы просто установить порог θ и сказать, что $P(x^1|y) = 0$, если $y < \theta$, и $P(x^1|y) = 1$ в противном случае. Однако во многих реальных сценариях нам не подходит такой жесткий порог, который приводит к присвоению нулевой вероятности x^1 для определенных значений y.

Вместо жесткого порога мы могли бы использовать *мягкий порог*, который присваивает низкие вероятности, когда условная переменная ниже порога, и высокие вероятности, когда она выше порога. Одним из способов представления мягкого порога является использование *логит-модели*, которая дает сигмовидную кривую:

$$P(x^1|y) = \frac{1}{1 + \exp\left(-2\frac{y-\theta_1}{\theta_2}\right)}. \tag{2.30}$$

[14] Сигмоидой называют S-образную кривую. Существуют разные способы математического определения такой кривой, но мы сосредоточимся на логит-модели.

Параметр θ_1 управляет расположением порога, а параметр θ_2 управляет «мягкостью», или разбросом, вероятностей. На рис. 2.10 показан график распределения $P(x^1|y)$ с логит-моделью.

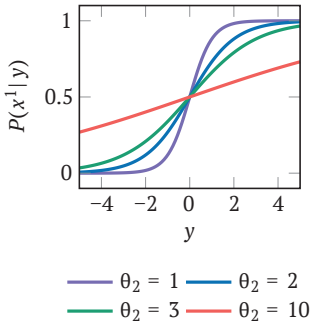


Рис. 2.10. Логит-модель с $\theta_1 = 0$ и разными значениями θ_2

2.4.6. Детерминированные переменные

Некоторые задачи могут содержать *детерминированную переменную*, значение которой является фиксированным при заданных условиях. Другими словами, мы присваиваем вероятность 1 значению, которое является детерминированной функцией от условий. В принципе, для представления дискретной детерминированной переменной можно использовать таблицу условной вероятности, но это расточительно. Ведь только один экземпляр переменной будет иметь вероятность 1 для каждого родительского экземпляра, а остальные записи будут равны 0. Мы можем использовать эту разреженность для более компактного представления. В данной книге алгоритмы, использующие дискретные факторы, рассматривают любые назначения, отсутствующие в таблице факторов, как нулевые, поэтому мы должны хранить только назначения, которые имеют ненулевую вероятность.

2.5. Байесовские сети

Для представления совместного распределения вероятностей можно использовать *байесовскую сеть*[15]. Структура байесовской сети описывается ориентированным ациклическим графом, состоящим из узлов и ориентированных ребер[16]. Каждый узел сети соответствует переменной. Направленные ребра соединяют пары вершин, при этом циклы в графе запрещены. Направленные

[15] Подробное описание байесовских сетей и других форм вероятностных графических моделей см. в D. Koller, N. Friedman, *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

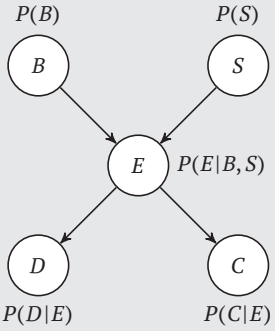
[16] Общая терминология в области графов представлена в приложении А.16.

ребра указывают на прямые вероятностные отношения[17]. С каждым узлом X_i связано условное распределение $P(X_i|\text{Pa}(X_i))$, где $\text{Pa}(X_i)$ представляет родителей X_i в графе. Алгоритм 2.2 реализует структуру данных байесовской сети. Пример 2.5 иллюстрирует применение байесовских сетей к задаче мониторинга состояния спутников.

Алгоритм 2.2. Представление дискретной байесовской сети через набор переменных, факторы и граф. Графовая структура данных представлена в файле `Graphs.jl`

```
struct BayesianNetwork
    vars::Vector{Variable}
    factors::Vector{Factor}
    graph::SimpleDiGraph{Int64}
end
```

Пример 2.5. Байесовская сеть, представляющая задачу мониторинга состояния спутников. Здесь показана структура сети, представленная в виде ориентированного ациклического графа. С каждым узлом связано условное распределение вероятностей



B отказ батареи
S отказ солнечной панели
E отказ системы электропитания
D отклонение от траектории
C потеря связи

На рисунке представлена байесовская сеть для задачи мониторинга состояния спутников с пятью бинарными переменными. К счастью, отказ батареи и выход из строя солнечной панели случаются редко, хотя вероятность отказа солнечной панели несколько выше, чем отказа батареи. Отказ батареи или панели может привести к отказу системы электропитания. Могут быть и другие причины отказа системы электропитания, помимо отказа батареи или солнечной панели, например проблема с блоком управления

[17] В причинно-следственных сетях направление ребер указывает на причинно-следственные связи между переменными. Однако в общих байесовских сетях причинность не требуется. J. Pearl, Causality: Models, Reasoning, and Inference, 2nd ed. Cambridge University Press, 2009.

питанием. Отказ системы электропитания может привести к отклонению траектории, которое можно наблюдать с Земли в телескоп, а также к потере связи, которая прерывает передачу телеметрических данных и данных миссии на различные наземные станции. Другие аномалии, не связанные с электропитанием, тоже могут привести к отклонению от траектории и потере связи.

С каждой из пяти переменных связаны пять распределений условной вероятности. Поскольку B и S не имеют родителей, нам нужно указать только $P(B)$ и $P(S)$. Фрагмент показанного ниже кода создает структуру байесовской сети с примерами значений для элементов связанных таблиц факторов. Кортежи в таблицах факторов индексируют области значений переменных, то есть {0,1} для всех переменных. Например, `(e=2, b=1, s=1)` соответствует (e^1, b^0, s^0).

```
# требуются вспомогательные функции из приложения G.5
B = Variable(:b, 2); S = Variable(:s, 2)
E = Variable(:e, 2)
D = Variable(:d, 2); C = Variable(:c, 2)
vars = [B, S, E, D, C]
factors = [
    Factor([B], FactorTable((b=1,) => 0.99, (b=2,) => 0.01)),
    Factor([S], FactorTable((s=1,) => 0.98, (s=2,) => 0.02)),
    Factor([E,B,S], FactorTable(
        (e=1,b=1,s=1) => 0.90, (e=1,b=1,s=2) => 0.04,
        (e=1,b=2,s=1) => 0.05, (e=1,b=2,s=2) => 0.01,
        (e=2,b=1,s=1) => 0.10, (e=2,b=1,s=2) => 0.96,
        (e=2,b=2,s=1) => 0.95, (e=2,b=2,s=2) => 0.99)),
    Factor([D, E], FactorTable(
        (d=1,e=1) => 0.96, (d=1,e=2) => 0.03,
        (d=2,e=1) => 0.04, (d=2,e=2) => 0.97)),
    Factor([C, E], FactorTable(
        (c=1,e=1) => 0.98, (c=1,e=2) => 0.01,
        (c=2,e=1) => 0.02, (c=2,e=2) => 0.99))
]
graph = SimpleDiGraph(5)
add_edge!(graph, 1, 3); add_edge!(graph, 2, 3)
add_edge!(graph, 3, 4); add_edge!(graph, 3, 5)
bn = BayesianNetwork(vars, factors, graph)
```

Цепное правило для байесовских сетей определяет построение совместного распределения из локальных распределений условной вероятности. Предположим, что у нас есть переменные $X_{1:n}$ и мы хотим вычислить вероятность присвоения всем этим переменным определенных значений $P(x_{1:n})$. В соответствии с цепным правилом

$$P(x_{1:n}) = \prod_{i=1}^{n} P(x_i|\text{pa}(x_i)), \tag{2.31}$$

где $\text{pa}(x_i)$ – присвоение определенного значения родителю X_i. Алгоритм 2.3 демонстрирует реализацию байесовских сетей с условными распределениями вероятностей, представленными в виде дискретных факторов.

Алгоритм 2.3. Функция для оценки вероятности присвоения определенных значений в заданной байесовской сети `bn`. Например, если `bn` определена в примере 2.5, тогда `a = (b=1,s=1,e=1,d=2,c=1) probability(bn, Assignment(a))` возвращает `0,034228655999999996`

```
function probability(bn::BayesianNetwork, assignment)
    subassignment(φ) = select(assignment, variablenames(φ))
    probability(φ) = get(φ.table, subassignment(φ), 0.0)
    return prod(probability(φ) for φ in bn.factors)
end
```

В примере с мониторингом спутника предположим, что мы хотим вычислить вероятность того, что все в порядке; то есть $P(b^0, s^0, e^0, d^0, c^0)$. Из цепного правила следует, что

$$P(b^0, s^0, e^0, d^0, c^0) = P(b^0)P(s^0)P(e^0|b^0, s^0)P(d^0|e^0)P(c^0|e^0). \tag{2.32}$$

Чтобы полностью определить совместное распределение по пяти переменным B, S, E, D и C, нам потребуется $2^5 - 1 = 31$ независимый параметр. Структура нашей байесовской сети позволяет нам определить совместное распределение, используя только $1 + 1 + 4 + 2 + 2 = 10$ независимых параметров. В данном случае разница между 10 и 31 не выглядит значительной, но экономия может быстро стать огромной в более крупных байесовских сетях. Достоинство байесовских сетей заключается в их способности уменьшать количество параметров, необходимых для определения совместного распределения вероятностей.

2.6. Условная независимость

Причина удивительной способности байесовской сети представлять совместное распределение с меньшим количеством независимых параметров, чем обычно требуется, заключается в предположениях об условной независимости, закодированных в ее графовой структуре[18]. *Условная независимость* является обобщением понятия независимости, введенного в разделе 2.3.1. Переменные X и Y условно независимы при заданном Z тогда и только тогда, когда $P(X, Y|Z) = P(X|Z)P(Y|Z)$. Утверждение, что X и Y условно независимы при заданном Z, записывается как $(X \perp Y|Z)$. Из этого определения можно показать, что $(X \perp Y|Z)$ тогда и только тогда, когда $P(X|Z) = P(X|Y, Z)$. При заданном Z информа-

[18] Если предположения об условной независимости, сделанные байесовской сетью, неверны, мы рискуем неправильно смоделировать совместное распределение, о чем пойдет речь в главе 5.

ция об Y не дает дополнительной информации об X, и наоборот. Иллюстрация этого положения представлена в примере 2.6.

Пример 2.6. Условная независимость в задаче мониторинга спутников

Предположим, что наличие отклонения траектории спутника (D) условно не зависит от того, есть ли у нас потеря связи (C), если известно об отказе системы электропитания (E). Мы можем записать это так: ($D \perp C \mid E$). Если мы знаем, что у нас произошел сбой электропитания, то наблюдаемая потеря связи никак не влияет на наше убеждение в наличии отклонения от траектории. У нас может быть повышенное ожидание отклонения траектории, но это только потому, что мы знаем об отказе системы электропитания.

Мы можем использовать набор правил, чтобы определить, подразумевает ли структура байесовской сети условную независимость двух переменных при заданном наборе других переменных[19]. Предположим, что мы хотим проверить, подразумевается ли выполнение ($A \perp B \mid C$) структурой сети, где C – набор обусловливающих переменных (свидетельств). Мы должны проверить все возможные ненаправленные пути из A в B на наличие так называемого *d-разделения*. Путь между A и B имеет d-разделение по C, если верно любое из следующих условий:

1. Путь содержит *цепочку* узлов $X \rightarrow Y \rightarrow Z$, такую, что Y находится в C.
2. Путь содержит *вилку* $X \leftarrow Y \rightarrow Z$, такую, что Y лежит в C.
3. Путь содержит *перевернутую вилку* (также называемую *v-структурой*), $X \rightarrow Y \leftarrow Z$, такую, что Y не находится в C и ни один потомок Y не находится в C. Пример 2.7 иллюстрирует это правило.

Мы говорим, что A и B d-разделены по C, если все пути между A и B d-разделены по C. Из этого d-разделения следует, что ($A \perp B \mid C$)[20]. Пример 2.8 демонстрирует процесс проверки того, подразумевает ли граф определенное предположение об условной независимости.

Иногда для обозначения минимального набора узлов, которые, если бы их значения были известны, делали бы X условно независимым от всех других узлов, используют термин *марковское одеяло* узла X[21]. Марковское одеяло конкретного узла состоит из его родителей, его потомков и других родителей его потомков.

---

[19] Даже если структура сети не предполагает условной независимости, условная независимость все же может иметь место благодаря выбору условных распределений вероятностей. См. упражнение 2.10.

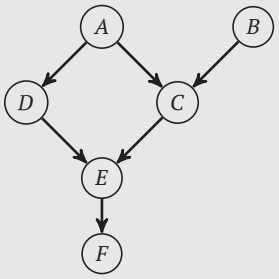
[20] Алгоритм эффективного определения d-разделения немного сложен. См. алгоритм 3.1 в D. Koller and N. Friedman, *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

[21] Названо в честь русского математика А.А. Маркова (1856–1922). J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

Пример 2.7. Соображения, лежащие в основе предположений об условной независимости, подразумеваемые (и не подразумеваемые) в цепочках, вилках и перевернутых вилках

Если у нас есть $X \rightarrow Y \rightarrow Z$ (цепочка) или $X \leftarrow Y \rightarrow Z$ (вилка) при заданном Y, то X и Z условно независимы, то есть $P(X|Y, Z) = P(X|Y)$. Интересно, что если бы направления стрелок были немного другими, $X \rightarrow Y \leftarrow Z$ (перевернутая вилка), то X и Z могли бы уже не быть условно независимыми при заданном Y. Другими словами, может быть так, что $P(B|E) = P(B|S, E)$. Для лучшего понимания рассмотрим перевернутую вилку пути от отказа батареи B к отказу солнечной панели S через отказ системы электропитания E. Предположим, мы знаем, что у нас есть отказ системы электропитания. Если мы знаем, что батарея исправна, то мы более склонны полагать, что отказала солнечная панель, потому что это альтернативная причина отказа электропитания. И наоборот, если мы узнаем, что у нас действительно вышла из строя батарея, наша уверенность в том, что неисправна солнечная панель, уменьшится. Этот эффект называется *объяснением*. Наблюдение за отказом солнечной панели объясняет причину отказа системы электропитания.

Пример 2.8. Допущения условной независимости, подразумеваемые приведенной ниже графовой структурой



Предположим, что мы хотим определить, подразумевает ли сеть, показанная на этом рисунке, что $(D \perp B \mid F)$. Есть два ненаправленных пути из D в B. Нам нужно проверить оба пути на d-разделение.

Путь $D \leftarrow A \rightarrow C \leftarrow B$ включает вилку $D \leftarrow A \rightarrow C$, за которой следует перевернутая вилка $A \rightarrow C \leftarrow B$. В точке A нет обусловливания, поэтому нет d-разделения от вилки. Поскольку F является потомком C, вдоль перевернутой вилки тоже нет d-разделения. Следовательно, на этом пути нет d-разделения.

Второй путь, $D \rightarrow E \leftarrow C \leftarrow B$, содержит перевернутую вилку $D \rightarrow E \leftarrow C$ и цепочку $E \leftarrow C \leftarrow B$. Поскольку F является потомком E, вдоль перевернутой вилки нет d-разделения. А поскольку вдоль цепной части этого пути также нет d-разделения, то нет d-разделения и на втором пути от D до B.

Чтобы D и B были условно независимыми при заданном F, должно присутствовать d-разделение на всех ненаправленных путях из D в B. Но в данном случае ни один из двух путей не имеет d-разделения. Следовательно, в структуре данной сети условная независимость не заложена.

2.7. Заключение

- Представление неопределенности в виде распределения вероятностей опирается на набор аксиом, связанных со сравнением правдоподобия различных утверждений.
- Существует множество семейств как дискретных, так и непрерывных распределений вероятностей.
- Непрерывные распределения вероятностей могут быть представлены функциями плотности.
- Семейства распределений вероятностей можно смешивать для создания более гибких распределений.
- Совместные распределения – это распределения по нескольким переменным.
- Условные распределения – это распределения по одной или нескольким переменным при заданных значениях обусловливающих переменных.
- Байесовская сеть определяется графовой структурой и набором условных распределений.
- Благодаря структуре байесовской сети мы можем представлять совместные распределения с меньшим количеством параметров из-за предположений об условной независимости.

2.8. Упражнения

Упражнение 2.1. Рассмотрим непрерывную случайную величину X, которая следует *экспоненциальному распределению*, параметризованному λ с плотностью вероятности $p(x|\lambda) = \lambda\exp(-\lambda x)$ с неотрицательным опорным интервалом. Вычислите кумулятивную функцию распределения X.

Решение. Начнем с определения кумулятивной функции распределения. Поскольку интервал распределения ограничен снизу по $x = 0$, в интервале $(-\infty, 0)$ отсутствует вероятностная мера, что позволяет привести нижнюю границу интеграла к 0. После вычисления интеграла получаем $\mathrm{cdf}_X(x)$:

$$\mathrm{cdf}_X(x) = \int_{-\infty}^{x} p(x')dx';$$

$$\mathrm{cdf}_X(x) = \int_0^x \lambda e^{-\lambda x'} dx';$$

$$\mathrm{cdf}_X(x) = -e^{-\lambda x'}\Big|_0^x;$$

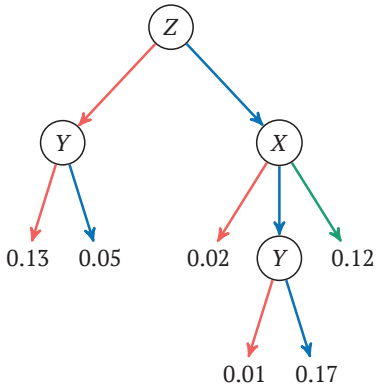
$$\mathrm{cdf}_X(x) = 1 - e^{-\lambda x}.$$

Упражнение 2.2. Какие пять компонентов смешанного распределения вы можете предложить для функции плотности на рис. 2.6? (Есть несколько правильных решений.)

Решение. Один из вариантов: $\mathcal{U}([-10,-10], [-5, 10])$, $\mathcal{U}([-5, 0], [0, 10])$, $\mathcal{U}([-5,-10], [0, 0])$, $\mathcal{U}([0,-10], [10, 5])$, $\mathcal{U}([0, 5], [10, 10])$.

Упражнение 2.3. Опираясь на представленное ниже табличное представление $P(X, Y, Z)$, сгенерируйте эквивалентное компактное представление дерева решений:

X	Y	Z	$P(X, Y, Z)$
0	0	0	0.13
0	0	1	0.02
0	1	0	0.05
0	1	1	0.02
1	0	0	0.13
1	0	1	0.01
1	1	0	0.05
1	1	1	0.17
2	0	0	0.13
2	0	1	0.12
2	1	0	0.05
2	1	1	0.12



Решение. Начнем с наиболее часто встречающихся вероятностей: 0.13 – когда $Z = 0$ и $Y = 0$, и 0.05 – когда $Z = 0$ и $Y = 1$. Сделаем Z корнем нашего дерева решений, и когда $Z = 0$, мы переходим к узлу Y. В зависимости от значения Y мы переходим либо к 0.13, либо к 0.05. Далее строим ветки для случаев, когда $Z = 1$. Наиболее распространенными являются вероятности 0.02, что имеет место, когда $Z = 1$ и $X = 0$, и 0.12 – когда $Z = 1$ и $X = 2$. Итак, когда $Z = 1$, мы переходим к узлу X. В зависимости от значения X (0, 1, 2) мы переходим к значению 0.02, узлу Y или значению 0.12 соответственно. Наконец, в зависимости от значения Y мы переходим либо к 0.01, либо к 0.17.

Упражнение 2.4. Предположим, что мы хотим определить многомерную смешанную модель Гаусса с тремя компонентами, определенными по четырем переменным. Нам нужно, чтобы два из трех нормальных распределений предполагали независимость между четырьмя переменными, в то время как третье нормальное распределение было определено без каких-либо предположений о независимости. Сколько независимых параметров необходимо для определения этой смешанной модели?

Решение. Для нормального распределения по четырем переменным ($n = 4$) с предположениями о независимости нам нужно указать $n + n = 2n = 8$ независимых параметров: 4 параметра для вектора математического ожидания и 4 параметра для матрицы ковариаций (что эквивалентно параметрам матожидания и дисперсии четырех независимых одномерных нормальных распределений). Для нормального распределения по четырем переменным без предположений о независимости нам нужно указать $n + n(n + 1)/2 = 14$ независимых параметров: 4 параметра для вектора математического ожидания и 10 параметров для матрицы ковариаций. Кроме того, для наших трех компонентов смешанной модели ($k = 3$) нам нужно указать $k - 1 = 2$ независимых параметра для весов. Таким образом, нам нужно $2(8) + 1(14) + 2 = 32$ независимых параметра, чтобы задать это смешанное распределение.

Упражнение 2.5. У нас есть три независимые переменные $X_{1:3}$, определяемые кусочно-равномерными плотностями с 4, 7 и 3 границами интервалов соответственно. Сколько независимых параметров потребуется для задания их совместного распределения?

Решение. Если у нас есть кусочно-равномерная плотность с m границами, значит, есть $m - 1$ интервалов и $m - 2$ независимых параметров. В данном случае требуется $(4 - 2) + (7 - 2) + (3 - 2) = 8$ независимых параметров.

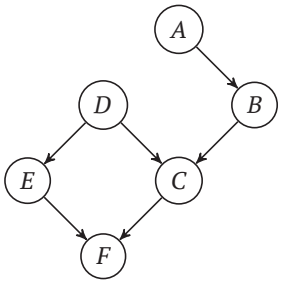
Упражнение 2.6. Предположим, что у нас есть четыре непрерывные случайные величины, X_1, X_2, Y_1 и Y_2, и мы хотим построить линейную гауссову модель $X = X_{1:2}$ при заданном $Y = Y_{1:2}$; то есть $p(X|Y)$. Сколько независимых параметров требуется для этой модели?

Решение. В этом случае наш вектор математического ожидания для нормального распределения является двухмерным и требует четырех независимых параметров для матрицы преобразования $\mathbf{M}$ и двух независимых параметров для вектора смещения $\mathbf{b}$. Нам также потребуются три независимых параметра для матрицы ковариаций $\mathbf{\Sigma}$. Всего нам нужно $4 + 2 + 3 = 9$ независимых параметров для определения этой модели:

$$p(\mathbf{x}|\mathbf{y}) = \mathcal{N}(\mathbf{x}|\mathbf{M}\mathbf{y} + \mathbf{b}, \mathbf{\Sigma}).$$

Упражнение 2.7. Сколько независимых параметров у изображенной ниже байесовской сети, в которой каждый узел может принимать одно из четырех значений? Каково процентное сокращение количества независимых парамет-

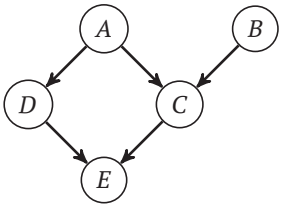
ров, необходимых при использовании этой байесовской сети, по сравнению с использованием полной сводной таблицы вероятностей?



Решение. Количество независимых параметров для каждого узла равно $(k-1)k^m$, где k – количество значений, которые может принимать узел, а m – количество родителей, которые есть у этого узла. Переменная A имеет 3, B имеет 12, C имеет 48, D имеет 3, E имеет 12, а F имеет 48 независимых параметров. Всего у этой байесовской сети 126 независимых параметров.

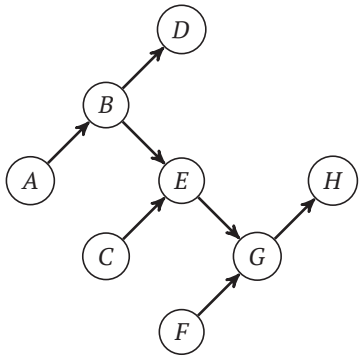
Количество независимых параметров, необходимых для построения сводной таблицы вероятностей по n переменным, которые могут принимать k значений, равно $k^n - 1$. Следовательно, для сводной таблицы вероятностей потребуется $46 - 1 = 4096 - 1 = 4095$ независимых параметров. Процентное сокращение количества требуемых независимых параметров составляет $(4095 - 126)/4095 \approx 96.9\ \%$.

Упражнение 2.8. Является ли узел A d-отделенным от E при заданном C для изображенной ниже байесовской сети?



Решение. Есть два пути из A в E: $A \to D \to E$ и $A \to C \to E$. По второму пути есть d-разделение, а по первому – нет. Следовательно, A не является d-отделенным от E при заданном C.

Упражнение 2.9. Определите марковское одеяло узла B для изображенной ниже байесовской сети:



Решение. Пути из B в A могут быть d-разделены только при заданном A. Пути из B в D могут быть d-разделены только при заданном D. Пути из B в E и одновременно F, G и H могут быть эффективно d-разделены заданным E. Пути от B к C естественно d-разделены из-за v-структуры; однако поскольку узел E должен содержаться в нашем марковском одеяле, пути от B к C при заданном E могут быть d-разделены только при заданном C. Таким образом, марковское одеяло узла B имеет состав $\{A, C, D, E\}$.

Упражнение 2.10. Может ли A быть независимым от B в байесовской сети со структурой $A \rightarrow B$?

Решение. Наличие прямой стрелки от A к B указывает на то, что независимость не подразумевается по умолчанию. Однако это не означает, что они не являются независимыми на самом деле. На независимость A и B влияет выбор таблиц условной вероятности. Мы можем подобрать таблицы так, чтобы обеспечить независимость. Например, предположим, что обе переменные являются бинарными и $P(a) = 0.5$ равномерна, а $P(b|a) = 0.5$. Ясно, что $P(A)P(B|A) = P(A)P(B)$, а значит, они независимы.

3 Вероятностный вывод

В предыдущей главе говорилось о формальных представлениях распределений вероятностей. Далее будет показано, как использовать эти представления для *вероятностного вывода*, который представляет собой определение распределения по одной или нескольким ненаблюдаемым переменным, исходя из значений, связанных с набором наблюдаемых переменных. Мы начнем с определения методов *точного вывода* (exact inference). Поскольку при определенной структуре сети точный вывод может оказаться затруднительным с точки зрения вычислений, мы также обсудим несколько алгоритмов приближенного вывода.

3.1. Вывод в байесовских сетях

В задачах вывода мы должны вывести распределение вероятности по *переменным запроса* (query variable), учитывая некоторые *наблюдаемые переменные* (evidence variable). Остальные узлы называются *скрытыми переменными* (hidden variable). Распределение, построенное по переменным запроса, исходя из наблюдаемых переменных, часто называют *апостериорным распределением* (posterior distribution).

Чтобы перейти к примеру вычислений, необходимых для вывода, вспомним байесовскую сеть из примера 2.5, структура которой воспроизведена на рис. 3.1. Предположим, у нас есть B в качестве переменной запроса и наблюдаемые переменные $D = 1$ и $C = 1$. Задача вывода заключается в вычислении $P(b^1|d^1, c^1)$, что соответствует вычислению вероятности отказа батареи при наблюдаемом отклонении траектории и потере связи.

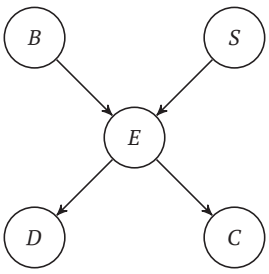


Рис. 3.1. Структура байесовской сети из примера 2.5

Из определения условной вероятности, представленного в уравнении (2.22), мы знаем, что

$$P(b^1|d^1,c^1) = \frac{P(b^1,d^1,c^1)}{P(d^1,c^1)}. \tag{3.1}$$

Чтобы найти числитель, применим *маргинализацию* (marginalization) – суммирование незадействованных переменных (в данном случае S и E):

$$P(b^1,d^1,c^1) = \sum_s \sum_e P(b^1,s,e,d^1,c^1). \tag{3.2}$$

Мы знаем из цепного правила для байесовских сетей, представленного в уравнении (2.31), что

$$P(b^1, s, e, d^1, c^1) = P(b^1)P(s)P(e|b^1, s)P(d^1|e)P(c^1|e). \tag{3.3}$$

Все компоненты в правой части уравнения указаны в условных распределениях вероятностей, связанных с узлами в байесовской сети. Мы можем вычислить знаменатель в уравнении (3.1), используя тот же подход, но с дополнительным суммированием по значениям для B.

Этот процесс, включающий применение определения условной вероятности, маргинализацию и применение цепного правила, можно использовать для выполнения точного вывода в любой байесовской сети. Мы можем реализовать точный вывод, используя *факторы*. Напомним, что факторы представляют собой дискретные многовариантные распределения. Для этого мы применяем следующие три операции над факторами:

- *перемножение факторов* (алгоритм 3.1) позволяет объединить два фактора и получить более крупный фактор. Если у нас есть $\varphi(X, Y)$ и $\psi(X, Z)$, то и $\varphi \cdot \psi$ будет включать X, Y и Z: $(\varphi \cdot \psi)(x, y, z) = \varphi(x, y)\psi(y, z)$. Произведение факторов проиллюстрировано примером 3.1;
- *факторная маргинализация* (алгоритм 3.2) применяется для суммирования определенной переменной из всей таблицы факторов, удаляя ее из результирующей области. Этот процесс проиллюстрирован примером 3.2;
- *факторное обусловливание* (алгоритм 3.3) применяется по отношению к некоторому наблюдению, чтобы удалить все строки в таблице, несовместимые с этим свидетельством. Пример 3.3 демонстрирует факторное обусловливание.

Все вышеупомянутые операции с факторами используются в алгоритме 3.4 для выполнения точного вывода. Он начинается с вычисления произведения всех факторов, обусловливания наблюдений, исключения скрытых переменных и нормализации. Одной из потенциальных проблем этого подхода является размер произведения всех факторов. Размер произведения равен произведению числа значений, которые может принимать каждая переменная. В примере со спутником имеется только $2^5 = 32$ возможные комбинации значений переменных, но многие интересные задачи будут иметь произведение факторов, которое слишком велико для прямого перечисления.

Алгоритм 3.1. Реализация перемножения факторов. В результате образуется фактор, представляющий совместное распределение двух меньших факторов φ и ψ. Если мы хотим вычислить факторное произведение φ и ψ, мы просто пишем φ*ψ

```
function Base.:*(φ::Factor, ψ::Factor)
    φnames = variablenames(φ)
    ψnames = variablenames(ψ)
    ψonly = setdiff(ψ.vars, φ.vars)
    table = FactorTable()
    for (φa,φp) in φ.table
        for a in assignments(ψonly)
            a = merge(φa, a)
            ψa = select(a, ψnames)
            table[a] = φp * get(ψ.table, ψa, 0.0)
        end
    end
    vars = vcat(φ.vars, ψonly)
    return Factor(vars, table)
end
```

Пример 3.1. Иллюстрация произведения факторов, представляющих $\varphi_1(X, Y)$ и $\varphi_2(Y, Z)$ для получения фактора, представляющего $\varphi_3(X, Y, Z)$

Произведение двух факторов образует новый фактор по объединению их переменных. Ниже мы создаем новый фактор из двух факторов, которые имеют общую переменную:

X	Y	$\varphi_1(X, Y)$
0	0	0.3
0	1	0.4
1	0	0.2
1	1	0.1

X	Y	$\varphi_2(X, Y)$
0	0	0.2
0	1	0.0
1	0	0.3
1	1	0.5

X	Y	Z	$\varphi_3(X, Y, Z)$
0	0	0	0.06
0	0	1	0.00
0	1	0	0.12
0	1	1	0.20
1	0	0	0.04
1	0	1	0.00
1	1	0	0.03
1	1	1	0.05

Алгоритм 3.2. Метод маргинализации переменной `name` из фактора φ

```
function marginalize(φ::Factor, name)
    table = FactorTable()
    for (a, p) in φ.table
        a′ = delete!(copy(a), name)
        table[a′] = get(table, a′, 0.0) + p
     end
     vars = filter(v -> v.name != name, φ.vars)
     return Factor(vars, table)
end
```

Пример 3.2. Иллюстрация маргинализации факторов

Вспомним совместное распределение вероятностей $P(X, Y, Z)$ из табл. 2.1. Мы можем маргинализировать Y, просуммировав вероятности в строках, которые имеют совпадающие значения для X и Z:

X	Y	Z	$\varphi(X, Y, Z)$
0	0	0	0.08
0	0	1	0.31
0	1	0	0.09
0	1	1	0.37
1	0	0	0.01
1	0	1	0.05
1	1	0	0.02
1	1	1	0.07

X	Y	$\varphi(X, Z)$
0	0	0.17
0	1	0.68
1	0	0.03
1	1	0.12

Алгоритм 3.3. Два метода факторного обусловливания при известных наблюдениях. Первый принимает фактор φ и возвращает новый фактор, записи таблицы которого согласуются с переменной `name`, имеющей значение `value`. Второй принимает фактор φ и применяет наблюдение в виде именованного кортежа. Метод `in_scope` возвращает значение `true`, если переменная с именем `name` относится к фактору φ

```
in_scope(name, φ) = any(name == v.name for v in φ.vars)

function condition(φ::Factor, name, value)
    if !in_scope(name, φ)
        return φ
    end
    table = FactorTable()
    for (a, p) in φ.table
```

```
        if a[name] == value
            table[delete!(copy(a), name)] = p
        end
    end
    vars = filter(v -> v.name != name, φ.vars)
    return Factor(vars, table)
end

function condition(φ::Factor, evidence)
    for (name, value) in pairs(evidence)
       φ = condition(φ, name, value)
    end
    return φ
end
```

Пример 3.3. Иллюстрация внесения наблюдения в фактор, в данном случае *Y*. Полученные значения необходимо заново нормировать

Факторное обусловливание подразумевает отбрасывание любых строк, несовместимых с наблюдениями. У нас есть факторы из табл. 2.1, и мы задали условие $Y = 1$. Все строки, для которых $Y \neq 1$, удаляются:

X	Y	Z	$\varphi(X, Y, Z)$
0	0	0	0.08
0	0	1	0.31
0	1	0	0.09
0	1	1	0.37
1	0	0	0.01
1	0	1	0.05
1	1	0	0.02
1	1	1	0.07

$Y = 1$

X	Y	$\varphi(X, Z)$
0	0	0.09
0	1	0.37
1	0	0.03
1	1	0.07

Алгоритм 3.4. Наивный алгоритм точного вывода для дискретной байесовской сети `bn`, который принимает в качестве входных данных набор имен переменных запроса `query` и `evidence`, связывающих значения с наблюдаемыми переменными. Алгоритм вычисляет совместное распределение по переменным запроса в виде фактора. Мы вводим тип `ExactInference`, чтобы можно было вызывать `infer` с различными методами вывода, которые будут рассмотрены в оставшейся части этой главы

```
struct ExactInference end

function infer(M::ExactInference, bn, query, evidence)
   φ = prod(bn.factors)
   φ = condition(φ, evidence)
```

```
    for name in setdiff(variablenames(φ), query)
        φ = marginalize(φ, name)
    end
    return normalize!(φ)
end
```

3.2. Вывод в наивных байесовских моделях

В предыдущем разделе был представлен обобщенный метод выполнения точного вывода в произвольной байесовской сети. В этом разделе обсуждается, как этот же метод можно использовать для решения задач *классификации* с помощью особой разновидности байесовской сети, известной как *наивная байесовская модель*. Эта структура представлена на рис. 3.2. Эквивалентное, но более компактное представление показано на рис. 3.3 с использованием *панели* (plate), изображенной здесь в виде закругленного прямоугольника. Надпись $i = 1{:}n$ в нижней части рамки указывает, что i в нижнем индексе имени переменной проходит по значениям от 1 до n.

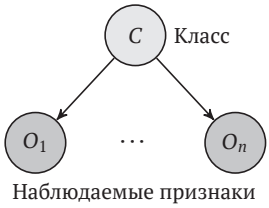


Рис. 3.2. Наивная байесовская модель

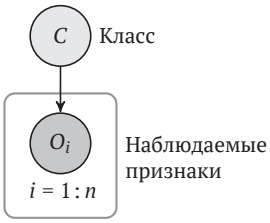


Рис. 3.3. Пластинчатое представление наивной байесовской модели

В наивной байесовской модели класс C является переменной запроса, а признаки $O_{1:n}$ – наблюдения. Наивная байесовская модель получила такое название, потому что она предполагает условную независимость между переменными наблюдений данного класса. Используя обозначения, введенные в разделе 2.6, мы можем сказать, что $(O_i \perp O_j | C)$ для всех $i = j$. Конечно, если эти предположения об условной независимости не выполняются, мы можем добавить необходимые направленные ребра между наблюдаемыми признаками.

Мы должны указать *априорное распределение* $P(C)$ и *класс-условное распределение* $P(O_i|C)$. Как и в предыдущем разделе, мы можем применить цепное правило для вычисления совместного распределения:

$$P(c, o_{1:n}) = P(c)\prod_{i=1}^{n} P(o_i|c). \tag{3.4}$$

Наша задача классификации включает вычисление условной вероятности $P(c|o_{1:n})$. Из определения условной вероятности имеем

$$P(c|o_{1:n}) = \frac{P(c, o_{1:n})}{P(o_{1:n})}. \tag{3.5}$$

Мы можем вычислить знаменатель, маргинализируя совместное распределение:

$$P(o_{1:n}) = \sum_c P(c, o_{1:n}). \tag{3.6}$$

Знаменатель в уравнении (3.5) не является функцией C и поэтому может рассматриваться как константа. Следовательно, мы можем написать

$$P(c|o_{1:n}) = kP(c, o_{1:n}), \tag{3.7}$$

где k – *нормировочная постоянная*, такая что $\sum_c P(c|o_{1:n}) = 1$. Мы часто опускаем k и пишем

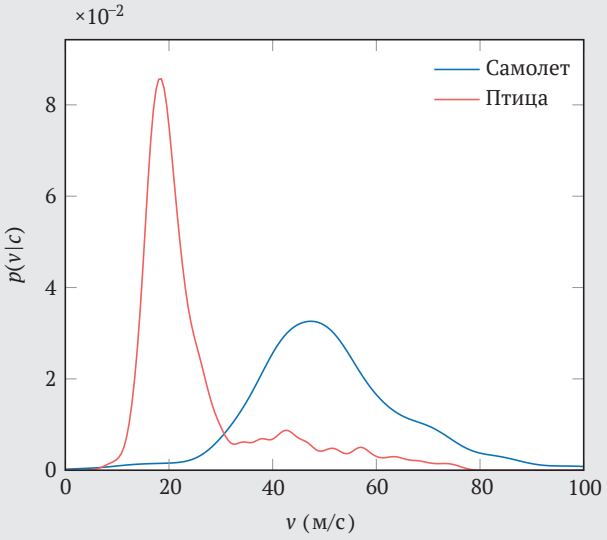
$$P(c|o_{1:n}) \propto P(c, o_{1:n}), \tag{3.8}$$

где *символ пропорциональности* $\propto$ используется для обозначения того, что левая сторона пропорциональна правой стороне. Пример 3.4 иллюстрирует, как наивный байесовский вывод может быть применен к классификации радарных следов.

Мы можем использовать этот метод, чтобы вывести распределение по классам, но во многих случаях перед нами стоит задача определить принадлежность к какому-то определенному классу. Обычно выполняют классификацию по классу с наивысшей апостериорной вероятностью $\text{argmax}_c P(c|o_{1:n})$. Однако выбор класса на самом деле является задачей принятия решения, в которой часто приходится учитывать последствия неправильной классификации. Например, если мы намерены использовать наш классификатор в системе управления воздушным движением для отсеивания радарных меток, не являющихся самолетами, то можем позволить себе время от времени пропускать через наш фильтр несколько птиц и другие мешающие метки (т. е. допустить ложную тревогу). Однако нам очень важно избежать отсеивания реальных самолетов, потому что это может привести к столкновению. В этом случае мы, вероятно, будем классифицировать радарную метку как птицу только в том случае, если апостериорная вероятность очень близка к 1. Проблемы принятия решений будут обсуждаться в главе 6.

Пример 3.4. Классификация радарных сигналов на соответствие птице или самолету

Предположим, у нас есть радиолокационный след метки на экране радара и мы хотим определить, кому он принадлежит – птице или самолету. Мы основываем наш вывод на скорости полета и количестве изменений курса. Первый параметр формирует наше мнение о том, является цель птицей или самолетом, при отсутствии какой-либо информации о траектории. Вот примеры распределений скорости полета v в зависимости от принадлежности к классу, полученные из радиолокационных данных:



Предположим, из цепного правила мы определили, что

P(птица, небольшие колебания курса) = 0.03;
P(самолет, небольшие колебания курса) = 0.01.

Конечно, эти вероятности в сумме не равны 1. Чтобы определить вероятность того, что объектом является птица, мы должны выполнить следующее вычисление:

$$\text{P(птица}|\text{медленная, небольшие колебания курса)} = \frac{0.03}{0.03+0.01} = 0.42.$$

3.3. Исключение переменной суммированием-перемножением

Для эффективного вывода в более сложных байесовских сетях можно использовать различные методы. Один из методов – *исключение переменной суммированием-перемножением* (sum-product variable elimination), при котором исключаются скрытые переменные (суммирование) с применением цепного правила (перемножение). Выгоднее маргинализировать переменные как можно раньше, чтобы избежать создания больших факторов.

Проиллюстрируем алгоритм исключения переменных, вычислив распределение $P(B|d^1, c^1)$ для байесовской сети на рис. 3.1. Условные распределения вероятностей, соответствующие узлам в сети, могут быть представлены следующими факторами:

$$\varphi_1(B), \varphi_2(S), \varphi_3(E, B, S), \varphi_4(D, E), \varphi_5(C, E). \tag{3.9}$$

Поскольку D и C являются наблюдаемыми переменными, последние два фактора можно заменить на $\varphi_6(E)$ и $\varphi_7(E)$, задав $D = 1$ и $C = 1$.

Затем мы последовательно удаляем скрытые переменные. Для выбора порядка удаления можно использовать разные стратегии, но в этом примере мы произвольно выбираем сначала E, а затем S. Чтобы исключить E, мы берем произведение всех факторов, включающих E, а затем маргинализируем E, чтобы получить новый фактор:

$$\varphi_8(B, S) = \sum_e \varphi_3(e, B, S)\varphi_6(e)\varphi_7(e). \tag{3.10}$$

Теперь мы можем отбросить φ_3, φ_6 и φ_7, потому что вся необходимая нам информация содержится в φ_8.

Затем мы исключаем S. Опять же, мы собираем все оставшиеся факторы, которые включают S, и исключаем S из произведения этих факторов:

$$\varphi_9(B) = \sum_s \varphi_2(s)\varphi_8(B, s). \tag{3.11}$$

Мы отбрасываем φ_2 и φ_8 и остаемся с $\varphi_1(B)$ и $\varphi_9(B)$. Наконец, мы перемножаем их и нормализуем результат, чтобы получить фактор, представляющий вероятность $P(B|d^1, c^1)$.

Эта процедура эквивалентна вычислению следующего выражения:

$$P(B|d^1, c^1) \propto \varphi_1(B)\sum_s \left(\varphi_3(e|B, s)\varphi_4(d^1|e)\varphi_5(c^1|e)\right). \tag{3.12}$$

Она дает тот же результат, но более эффективна, чем наивная процедура вычисления произведения всех факторов с последующей маргинализацией:

$$P(B|d^1, c^1) \propto \sum_s \sum_e \varphi_1(B)\varphi_2(s)\varphi_3(e|B, s)\varphi_4(d^1|e)\varphi_5(c^1|e). \tag{3.13}$$

Метод исключения переменной путем суммирования-перемножения реализован в алгоритме 3.5. Он принимает в качестве входных данных байесовскую сеть, набор переменных запроса, список наблюдаемых значений и порядок переменных. Сначала мы присваиваем все наблюдаемые значения. Затем для каждой переменной мы перемножаем все факторы, которые ее содержат, и исключаем эту переменную. Этот новый фактор заменяет прежние факторы, и мы повторяем процесс для следующей переменной.

Во многих случаях исключение переменных позволяет сделать вывод за время, которое линейно увеличивается с ростом размера сети, но в худшем случае время нарастает по экспоненциальному закону. На объем вычислений существенно влияет порядок исключения переменных. Выбор оптимального порядка исключения является *NP-трудным*[1]. Это означает, что в худшем случае решение невозможно найти за полиномиальное время (раздел 3.5). Даже если мы нашли оптимальный порядок, исключение переменных может потребовать экспоненциально растущего объема вычислений. Эвристические подходы к исключению переменных обычно пытаются минимизировать количество переменных, участвующих в промежуточных факторах, генерируемых алгоритмом.

Алгоритм 3.5. Реализация алгоритма исключения переменных, который использует байесовскую сеть `bn`, список переменных запроса `query` и наблюдения `evidence`. Переменные обрабатываются в порядке, указанном в `ordering`

```
struct VariableElimination
    ordering # массив переменных индексов
end

function infer(M::VariableElimination, bn, query, evidence)
    Φ = [condition(φ, evidence) for φ in bn.factors]
    for i in M.ordering
        name = bn.vars[i].name
        if name ∉ query
            inds = findall(φ->in_scope(name, φ), Φ)
            if !isempty(inds)
                φ = prod(Φ[inds])
                deleteat!(Φ, inds)
                φ = marginalize(φ, name)
                push!(Φ, φ)
            end
        end
    end
    return normalize!(prod(Φ))
end
```

[1] S. Arnborg, D. G. Corneil, A. Proskurowski, *Complexity of Finding Embeddings in a k-Tree*, SIAM Journal on Algebraic Discrete Methods, vol. 8, no. 2, pp. 277–284, 1987.

3.4. Распространение доверия

Механизм вывода, известный как *распространение доверия* (belief propagation), работает путем распространения «сообщений» по сети с использованием алгоритма суммирования для вычисления предельных распределений переменных запроса[2]. Время вычислений масштабируется линейно, но алгоритм дает точный ответ только в том случае, если в сети нет ненаправленных циклов. Если в сети есть ненаправленные циклы, то ее можно преобразовать в дерево, объединив несколько переменных в отдельные узлы с помощью так называемого *алгоритма дерева сочленений* (junction tree algorithm). Если количество переменных, которые необходимо объединить в какой-либо один узел в результирующей сети, невелико, то вывод можно получить с высокой вычислительной эффективностью. Вариант алгоритма, известный как *циклическое распространение доверия*, может обеспечить приближенные решения в сетях с ненаправленными циклами. Хотя этот подход не дает никаких гарантий и не всегда сходится, на практике он может работать довольно хорошо[3].

3.5. Вычислительная сложность

Мы можем показать, что вывод в байесовских сетях является NP-трудным, используя NP-полную задачу под названием 3SAT[4]. Байесовскую сеть легко построить из произвольной задачи 3SAT. Например, рассмотрим следующую формулу 3SAT[5]:

$$F(x_1, x_2, x_3, x_4) = \begin{array}{ccccccc} (& x_1 & \vee & x_2 & \vee & x_3 &) \\ (& \neg x_1 & \vee & \neg x_2 & \vee & x_3 &) \\ (& x_2 & \vee & \neg x_3 & \vee & x_4 &) \end{array} \begin{array}{c} \wedge \\ \wedge \\ \end{array}, \tag{3.14}$$

где ¬ представляет собой *логическое отрицание* (НЕ), ∧ представляет собой *логическую конъюнкцию* (И), а ∨ обозначает *логическую дизъюнкцию* (ИЛИ). Формула состоит из конъюнкции *клауз* (clause), которые, в свою очередь, являются дизъюнкциями *литералов*. Литерал – это просто переменная или ее отрицание.

---

[2] Учебное пособие по алгоритму суммирования-перемножения с обсуждением его связей со многими другими алгоритмами, разработанными отдельными группами: F. Kschischang, B. Frey, H.-A. Loeliger, Factor Graphs and the Sum-Product Algorithm, IEEE Transactions on Information Theory, vol. 47, no. 2, pp. 498–519, 2001.

[3] Распространение доверия и связанные с ним алгоритмы подробно описаны в книге D. Barber, *Bayesian Reasoning and Machine Learning.* Cambridge University Press, 2012.

[4] G. F. Cooper, *The Computational Complexity of Probabilistic Inference Using Bayesian Belief Networks*, Artificial Intelligence, vol. 42, no. 2–3, pp. 393–405, 1990. Байесовская сеть в этом разделе построена на основе упомянутой книги. В приложении C приводится краткий обзор классов сложности.

[5] Эта формула также используется в примере C.3 приложения C.

На рис. 3.4 показано соответствующее представление байесовской сети. Переменные представлены распределением $X_{1:4}$, а клаузы – распределением $C_{1:3}$. Распределения по переменным равномерны. Узлы, представляющие клаузы, имеют в качестве родителей причастные переменные. Поскольку это задача 3SAT, у каждого узла клаузы есть ровно три родителя. Каждый узел клаузы присваивает вероятность 0 назначениям, которые не удовлетворяют условию, и вероятность 1 всем удовлетворяющим назначениям. Остальные узлы присваивают вероятность 1 истине, если все их родители истинны. Исходная задача разрешима тогда и только тогда, когда $P(y^1) > 0$. Следовательно, вывод в байесовских сетях не менее сложен, чем 3SAT.

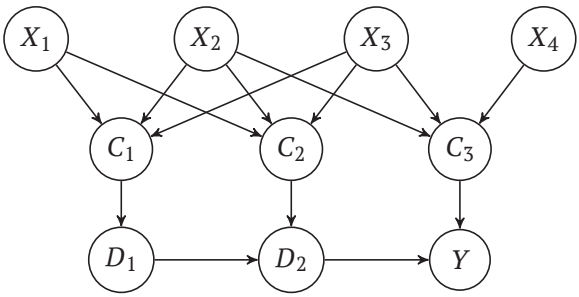


Рис. 3.4. Байесовская сеть, представляющая задачу 3SAT

Итак, мы знаем, что вывод в байесовских сетях является NP-трудным, и можем не тратить время на поиск эффективного и точного алгоритма вывода, который работает во всех байесовских сетях. Поэтому исследования за последние пару десятилетий были сосредоточены на приближенных методах вывода, которые обсуждаются далее.

3.6. Прямая выборка

В связи с тем, что точный вывод не поддается вычислительной обработке, было разработано множество методов аппроксимации. Один из простейших методов вывода основан на *прямой выборке* (direct sampling), когда для получения оценки вероятности используются случайные выборки из совместного распределения[6]. Чтобы продемонстрировать этот момент, предположим, что мы хотим вывести $P(b^1|d^1, c^1)$, имея n выборок из совместного распределения $P(b, s, e, d, c)$. Мы используем верхние индексы в скобках для обозначения индекса выборки, то есть $(b^{(i)}, s^{(i)}, e^{(i)}, d^{(i)}, c^{(i)})$ для i-й выборки. Прямая выборочная оценка вычисляется следующим образом:

[6] Иногда подходы, включающие случайную выборку, называют методами Монте-Карло. Название происходит от казино Монте-Карло в Монако. Введение в рандомизированные алгоритмы и их применение в различных предметных областях представлено в книге R. Motwani, P. Raghavan, *Randomized Algorithms.* Cambridge University Press, 1995.

$$P(b^1 | d^1, c^1) \approx \frac{\sum_i (b^{(i)} = 1 \wedge c^{(i)} = 1 \wedge d^{(i)} = 1)}{\sum_i (d^{(i)} = 1 \wedge c^{(i)} = 1)}. \tag{3.15}$$

Мы используем соглашение, согласно которому логическое утверждение в круглых скобках обрабатывается численно как 1, когда оно истинно, и 0, когда оно ложно. Числитель – это количество выборок, соответствующих параметрам b, d и c, равным 1, а знаменатель – это количество выборок, соответствующих параметрам d и c, равным 1.

Процесс выборки из совместного распределения, представленного байесовской сетью, несложен. На первом шаге мы выполняем *топологическую сортировку* (topological sort) узлов в байесовской сети. Топологическая сортировка узлов в ориентированном ациклическом графе – это упорядоченный список, такой, что если существует ребро $A \rightarrow B$, то A предшествует B в списке[7]. Например, топологической сортировкой для сети на рис. 3.1 является список B, S, E, D, C. Топологическая сортировка всегда существует, но не всегда уникальна. Еще одна топологическая сортировка той же сети – S, B, E, C, D.

Выполнив топологическую сортировку, мы можем начать делать выборку из условных распределений вероятностей. Алгоритм 3.6 показывает, как извлекать выборку из байесовской сети с заданным порядком $X_{i:n}$, который представлен топологической сортировкой. Мы извлекаем выборку из условного распределения, связанного с X_i, с учетом значений родителей, которые уже были назначены. Поскольку $X_{i:n}$ является топологической сортировкой, мы знаем, что все родители X_i уже существуют, что позволяет выполнить эту выборку. Прямая выборка реализована в алгоритме 3.7 и продемонстрирована в примере 3.5.

Алгоритм 3.6. Метод выборки назначения из байесовской сети `bn`. Мы также предоставляем метод выборки назначения из фактора φ

```
function Base.rand(φ::Factor)
    tot, p, w = 0.0, rand(), sum(values(φ.table))
    for (a,v) in φ.table
        tot += v/w
        if tot >= p
            return a
        end
    end
    return Assignment()
end

function Base.rand(bn::BayesianNetwork)
    a = Assignment()
```

---

[7] A. B. Kahn, «Topological Sorting of Large Networks», Communications of the ACM, vol. 5, no. 11, pp. 558–562, 1962. Реализация топологической сортировки предоставляется в пакете `Graphs.jl`.

```
    for i in topological_sort(bn.graph)
        name, φ = bn.vars[i].name, bn.factors[i]
        a[name] = rand(condition(φ, a))[name]
    end
    return a
end
```

Пример 3.5. Пример использования прямой выборки из байесовской сети для логического вывода

Предположим, мы берем 10 случайных выборок из сети на рис. 3.1. Нас интересует вывод $P(b^1|d^1, c^1)$. Только 2 из 10 выборок (указаны в таблице) согласуются с наблюдениями d^1 и c^1. В одной из этих выборок $b = 1$, а в другой $b = 0$. Из этих выборок мы делаем вывод, что $P(b^1|d^1, c^1) = 0{,}5$. Конечно, для точной оценки $P(b^1|d^1, c^1)$ нам следует использовать больше, чем 2 выборки.

B	*S*	*E*	*D*	*C*	
0	0	1	1	0	
0	0	0	0	0	
1	0	1	0	0	
1	0	1	1	1	←
0	0	0	0	0	
0	0	0	1	0	
0	0	0	0	1	
0	1	1	1	1	←
0	0	0	0	0	
0	0	0	1	0	

Алгоритм 3.7. Метод вывода путем прямой выборки, который использует байесовскую сеть `bn`, список переменных запроса `query` и наблюдения `evidence`. Метод берет `m` выборок из байесовской сети и сохраняет те выборки, которые согласуются с наблюдениями. Метод возвращает фактор для переменных запроса. Этот метод может дать сбой, если не будут найдены выборки, удовлетворяющие наблюдениям

```
struct DirectSampling
    m # количество выборок
end

function infer(M::DirectSampling, bn, query, evidence)
    table = FactorTable()
    for i in 1:(M.m)
        a = rand(bn)
```

```
        if all(a[k] == v for (k,v) in pairs(evidence))
            b = select(a, query)
            table[b] = get(table, b, 0) + 1
        end
    end
    vars = filter(v->v.name ∉ query, bn.vars)
    return normalize!(Factor(vars, table))
end
```

3.7. Выборка, взвешенная по правдоподобию

Проблема прямой выборки заключается в том, что мы можем напрасно потратить время на создание выборок, не соответствующих наблюдениям, особенно если наблюдения маловероятны. Альтернативный подход называется *выборкой, взвешенной по правдоподобию* (likelihood weighted sampling), и заключается в создании взвешенных выборок, согласующихся с наблюдениями.

В качестве иллюстрации метода мы снова попытаемся вывести $P(b^1|d^1, c^1)$. У нас есть набор из n выборок, где i-я выборка снова обозначается $(b^{(i)}, s^{(i)}, e^{(i)}, d^{(i)}, c^{(i)})$. Вес i-й выборки равен w_i. Взвешенная оценка вычисляется следующим образом:

$$P(b^1|d^1,c^1) \approx \frac{\sum_i w_i(b^{(i)}=1 \wedge d^{(i)}=1 \wedge c^{(i)}=1)}{\sum_i w_i(d^{(i)}=1 \wedge c^{(i)}=1)} \tag{3.16}$$

$$= \frac{\sum_i w_i(b^{(i)}=1)}{\sum_i w_i}. \tag{3.17}$$

Чтобы сгенерировать эти взвешенные выборки, мы снова начинаем с топологической сортировки и последующей выборки из условных распределений. Единственное отличие взвешенного правдоподобия заключается в том, как мы обрабатываем наблюдения. Вместо выборки их значений из условного распределения мы присваиваем переменным наблюдаемые значения и соответствующим образом подбираем вес выборки. Вес выборки – это просто произведение условных вероятностей в наблюдаемых узлах. Выборка, взвешенная по правдоподобию, реализована в алгоритме 3.8. Пример 3.6 иллюстрирует вывод с выборкой, взвешенной по правдоподобию.

Алгоритм 3.8. Метод логического вывода с использованием выборки, взвешенной по правдоподобию, который использует байесовскую сеть `bn`, список переменных запроса `query` и наблюдения `evidence`. Метод извлекает m выборок из байесовской сети, но по возможности устанавливает значения на основе наблюдений, отслеживая при этом условную вероятность. Эти вероятности используются для взвешивания выборок таким образом, чтобы окончательный вывод был наиболее точным. Метод возвращает фактор для переменных запроса

```
struct LikelihoodWeightedSampling
    m # количество выборок
end

function infer(M::LikelihoodWeightedSampling, bn, query, evidence)
    table = FactorTable()
    ordering = topological_sort(bn.graph)
    for i in 1:(M.m)
        a, w = Assignment(), 1.0
        for j in ordering
            name, φ = bn.vars[j].name, bn.factors[j]
            if haskey(evidence, name)
                a[name] = evidence[name]
                w *= φ.table[select(a, variablenames(φ))]
            else
                a[name] = rand(condition(φ, a))[name]
            end
        end
        b = select(a, query)
        table[b] = get(table, b, 0) + w
    end
    vars = filter(v->v.name ∈ query, bn.vars)
    return normalize!(Factor(vars, table))
end
```

Пример 3.6. Выборки из байесовской сети, взвешенные по правдоподобию

В приведенной ниже таблице показаны пять взвешенных по правдоподобию выборок из сети на рис. 3.1. Мы делаем выборку из $P(B)$, $P(S)$ и $P(E \mid B, S)$, как и при прямой выборке. Приходя к D и C, мы присваиваем $D = 1$ и $C = 1$. Если в этой выборке $E = 1$, то вес равен $P(d^1|e^1)P(c^1|e^1)$; в противном случае вес равен $P(d^1|e^0)P(c^1|e^0)$. Если мы предположим, что

$P(d^1|e^1)P(c^1|e^1) = 0{,}95$,
$P(d^1|e^0)P(c^1|e^0) = 0{,}01$,

то мы можем аппроксимировать вывод из выборок в таблице:

$$P(b^1|d^1,c^1) = \frac{0.95}{0.95+0.95+0.01+0.01+0.95} \approx 0.331.$$

B	S	E	D	C	Вес
1	0	1	1	1	$P(d^1\|e^1)P(c^1\|e^1)$
0	1	1	1	1	$P(d^1\|e^1)P(c^1\|e^1)$
0	0	0	1	1	$P(d^1\|e^0)P(c^1\|e^0)$
0	0	0	1	1	$P(d^1\|e^0)P(c^1\|e^0)$
0	0	1	1	1	$P(d^1\|e^1)P(c^1\|e^1)$

Хотя взвешивание по правдоподобию делает так, что все выборки согласуются с наблюдениями, иногда оно может быть нерациональным. Рассмотрим простую байесовскую сеть обнаружения химических веществ, показанную на рис. 3.5, и предположим, что мы обнаружили интересующее химическое вещество. Мы хотим вывести $P(c^1|d^1)$. Поскольку эта сеть небольшая, мы можем легко точно вычислить эту вероятность, используя правило Байеса:

$$P(c^1|d^1) = \frac{P(d^1|c^1)P(c^1)}{P(d^1|c^1)P(c^1)+P(d^1|c^0)P(c^0)} \tag{3.18}$$

$$= \frac{0.999\times 0.001}{0.999\times 0.001+0.001\times 0.999} \tag{3.19}$$

$$= 0.5. \tag{3.20}$$

Если мы используем взвешивание по правдоподобию, то 99.9 % выборок будут иметь $C = 0$ с весом 0.001. Пока мы не получим выборку $C = 1$, связанную с весом 0.999, наша оценка $P(c^1|d^1)$ будет равна 0.

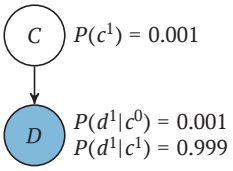


Рис. 3.5. Обнаружение химического вещества: байесовская сеть, где C указывает, *присутствует* ли химическое вещество, а D указывает, *обнаружено* ли химическое вещество

3.8. Выборка Гиббса

Альтернативным подходом к выводу является использование *выборки Гиббса* (Gibbs sampling)[8], которая является разновидностью метода Монте-Карло с цепями Маркова. Данный метод заключается в извлечении выборок без взвешивания, согласующихся с наблюдениями. Из этих выборок мы можем вывести распределения для переменных запроса.

Метод предусматривает генерацию последовательности выборок, начиная с исходной выборки $x_{1:n}^{(1)}$, сгенерированной случайным образом с переменными наблюдения, которым присвоены их наблюдаемые значения. Здесь k-я выборка $x_{1:n}^{(k)}$ вероятностно зависит от предыдущей выборки $x_{1:n}^{(k-1)}$. Модифицируем $x_{1:n}^{(k-1)}$, чтобы получить $x_{1:n}^{(k)}$ следующим образом. Используя любой порядок ненаблюдаемых переменных, который не обязательно должен быть топологическим, выбираем $x_i^{(k)}$ из распределения, представленного вероятностями $P(X_i|x_{-i}^{(k)})$. Здесь $x_{-i}^{(k)}$ представляет значения всех других переменных, кроме X_i, в выборке k. Выборку из $P(X_i|x_{-i}^{(k)})$ можно сделать эффективно, потому что нам нужно рассмотреть только марковское одеяло переменной X_i (раздел 2.6).

В отличие от предыдущих методов, в данном случае выборки не являются независимыми. Однако можно доказать, что в пределе выборки берутся точно из совместного распределения по ненаблюдаемым переменным с учетом наблюдений. В алгоритме 3.9 показано вычисление коэффициента для $P(X_i|x_{-i})$. Выборка Гиббса реализована в алгоритме 3.10.

Алгоритм 3.9. Метод вычисления $P(X_i|x_{-i})$ для байесовской сети `bn` при текущем назначении `a`

```
function blanket(bn, a, i)
    name = bn.vars[i].name
    val = a[name]
    a = delete!(copy(a), name)
    Φ = filter(φ -> in_scope(name, φ), bn.factors)
    φ = prod(condition(φ, a) for φ in Φ)
    return normalize!(φ)
end
```

---

[8] Назван в честь американского ученого Джозайи Уилларда Гиббса (1839–1903), который вместе с Джеймсом Клерком Максвеллом и Людвигом Больцманом создал область статистической механики.

Алгоритм 3.10. Реализация выборки Гиббса для байесовской сети `bn` с наблюдением `evidence` и порядком `ordering`. Метод итеративно обновляет назначение `a` в течение `m` итераций

```
function update_gibbs_sample!(a, bn, evidence, ordering)
    for i in ordering
        name = bn.vars[i].name
        if !haskey(evidence, name)
            b = blanket(bn, a, i)
            a[name] = rand(b)[name]
        end
    end
end

function gibbs_sample!(a, bn, evidence, ordering, m)
    for j in 1:m
        update_gibbs_sample!(a, bn, evidence, ordering)
    end
end

struct GibbsSampling
    m_samples # кол-во выборок для использования
    m_burnin # кол-во отбрасываемых выборок
    m_skip # кол-во пропускаемых выборок
    ordering # массив переменных индексов
end

function infer(M::GibbsSampling, bn, query, evidence)
    table = FactorTable()
    a = merge(rand(bn), evidence)
    gibbs_sample!(a, bn, evidence, M.ordering, M.m_burnin)
    for i in 1:(M.m_samples)
        gibbs_sample!(a, bn, evidence, M.ordering, M.m_skip)
        b = select(a, query)
        table[b] = get(table, b, 0) + 1
    end
    vars = filter(v->v.name ∈ query, bn.vars)
    return normalize!(Factor(vars, table))
end
```

К нашему рабочему примеру можно применить выборку Гиббса. Мы можем использовать наши m выборок для оценки следующим образом:

$$P(b^1|d^1,c^1) \approx \frac{1}{m}\sum_i (b^{(i)} = 1). \tag{3.21}$$

На рис. 3.6 сравнивается сходимость оценки $P(c^1|d^1)$ в сети обнаружения химических веществ с использованием трех выборок – прямой, взвешенной по правдоподобию и выборки Гиббса. Дольше всего сходится прямая выборка.

Кривая прямой выборки имеет длительные периоды, в течение которых оценка не меняется, поскольку выборки не согласуются с наблюдениями. В этом примере выборка, взвешенная по правдоподобию, сходится быстрее. Всплески возникают при генерации выборки с $C = 1$, а затем постепенно уменьшаются. Выборка Гиббса в этом примере быстро сходится к истинному значению 0,5.

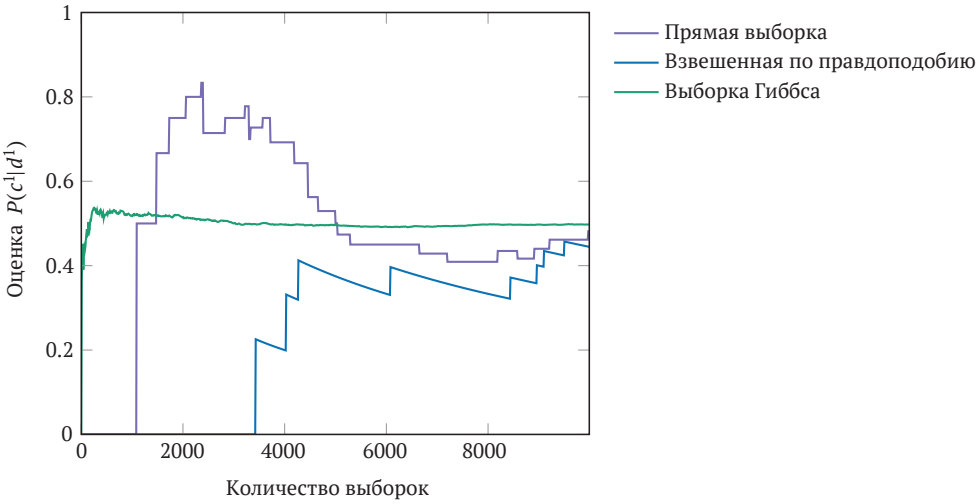


Рис. 3.6. Сравнение методов вывода на основе выборки в сети обнаружения химических веществ. Взвешенные по правдоподобию и прямые выборки имеют плохую сходимость из-за редкости событий, тогда как выборка Гиббса может эффективно сходиться к истинному значению даже без периода приработки или прореживания

Как упоминалось ранее, выборка Гиббса, как и другие методы Монте-Карло с цепями Маркова, в пределе производит выборки из желаемого распределения. На практике мы должны запускать выборку Гиббса в течение некоторого времени, называемого *периодом приработки* (burn-in period), прежде чем она сойдется к устойчивому распределению. Образцы, полученные во время приработки, обычно отбрасывают. Если необходимо использовать много выборок из одной серии выборок Гиббса, обычно выборки прореживают, оставляя только каждую h-ю выборку из-за потенциальной корреляции между выборками.

3.9. Вывод в гауссовых моделях

Если совместное распределение является гауссовым, мы можем выполнить точный вывод аналитически. Две совместные гауссовы случайные величины **a** и **b** могут быть записаны так:

$$\begin{bmatrix} \mathbf{a} \\ \mathbf{b} \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \boldsymbol{\mu}_\mathbf{a} \\ \boldsymbol{\mu}_\mathbf{b} \end{bmatrix}, \begin{bmatrix} \mathbf{A} & \mathbf{C} \\ \mathbf{C}^\top & \mathbf{B} \end{bmatrix} \right). \tag{3.22}$$

Частное распределение многомерного гауссиана также является гауссовым:

$$\mathbf{a} \sim \mathcal{N}(\boldsymbol{\mu}_a, \mathbf{A}). \tag{3.23}$$

Условное распределение многомерного гауссиана тоже является гауссовым с удобным решением в аналитическом виде:

$$p(\mathbf{a}|\mathbf{b}) = \mathcal{N}(\mathbf{a}|\boldsymbol{\mu}_{a|b}, \boldsymbol{\Sigma}_{a|b}); \tag{3.24}$$

$$\boldsymbol{\mu}_{a|b} = \boldsymbol{\mu}_a + \mathbf{C}\mathbf{B}^{-1}(\mathbf{b} - \boldsymbol{\mu}_b); \tag{3.25}$$

$$\boldsymbol{\Sigma}_{a|b} = \mathbf{A} - \mathbf{C}\mathbf{B}^{-1}\mathbf{C}^{\mathsf{T}}. \tag{3.26}$$

Алгоритм 3.11 демонстрирует использование этих уравнений для вывода распределения по набору переменных запроса с учетом наблюдений. В примере 3.7 показано, как извлечь частные и условные распределения из многомерного гауссова распределения.

Алгоритм 3.11. Вывод в многомерном гауссовом распределении `D`. Целочисленные векторы задают переменные запроса в аргументе `query` и переменные наблюдений в аргументе `evidencevars`. Значения переменных наблюдений содержатся в векторе `evidence`. Пакет `Distributions.jl` определяет распределение `MvNormal`

```
function infer(D::MvNormal, query, evidencevars, evidence)
    μ, Σ = D.μ, D.Σ.mat
    b, μa, μb = evidence, μ[query], μ[evidencevars]
    A = Σ[query,query]
    B = Σ[evidencevars,evidencevars]
    C = Σ[query,evidencevars]
    μ = μ[query] + C * (B\(b - μb))
    Σ = A - C * (B \ C')
    return MvNormal(μ, Σ)
end
```

Пример 3.7. Частные и условные распределения для многомерного гауссиана

Допустим

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 3 & 1 \\ 1 & 2 \end{bmatrix}\right).$$

Частное распределение для x_1 – $\mathcal{N}(0, 3)$, а для x_2 – $\mathcal{N}(1, 2)$.

Условное распределение для x_1 при $x_2 = 2$ равно

$$\mu_{x_1|x_2=2} = 0 + 1 \cdot 2^{-1} \cdot (2 - 1) = 0.5;$$

$$\Sigma_{x_1|x_2=2} = 3 - 1 \cdot 2^{-1} \cdot 1 = 2.5.$$

Мы можем выполнить этот расчет вывода при помощи алгоритма 3.11, построив совместное распределение

D = MvNormal([0.0,1.0],[3.0 1.0; 1.0 2.0]),

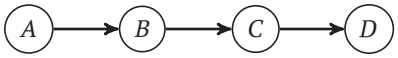
а затем выполнив вызов `infer(D, [1], [2], [2.0])`.

3.10. Заключение

- Вывод представляет собой определение вероятности переменных запроса при наличии некоторых наблюдений.
- Точный вывод можно сделать, вычислив совместное распределение по переменным, задав наблюдения и исключив любые скрытые переменные.
- Вывод можно эффективно делать в наивных байесовских моделях, в которых единственная родительская переменная влияет на множество условно независимых дочерних элементов.
- Алгоритм исключения переменных может повысить эффективность точного вывода за счет последовательного исключения переменных.
- Распространение доверия представляет собой еще один метод логического вывода, при котором информация итеративно передается между факторами для получения результата.
- Путем сведения к проблеме 3SAT можно показать, что вывод в байесовской сети является NP-трудным, что объясняет потребность в разработке методов приближенного вывода.
- Приблизительный вывод можно получить путем прямой выборки из совместного распределения, представленного байесовской сетью, но это может потребовать отбрасывания многих выборок, которые не согласуются с данными.
- Выборка, взвешенная по правдоподобию, может сократить объем вычислений, необходимых для приблизительного вывода, за счет создания только выборок, которые согласуются с фактическими данными, и соответствующего взвешивания каждой выборки.
- Выборка Гиббса формирует ряд невзвешенных выборок, которые согласуются с наблюдениями и могут значительно ускорить приблизительный вывод.
- Когда совместное распределение является гауссовым, точный вывод можно эффективно получить с помощью матричных операций.

3.11. Упражнения

Упражнение 3.1. Исходя из изображенной ниже байесовской сети и связанных с ней условных распределений вероятностей, напишите уравнение точного вывода для запроса $P(a^1|d^1)$.



Решение. Сначала развернем выражение вывода, используя определение условной вероятности:

$$P(a^1|d^1) = \frac{P(a^1, d^1)}{P(d^1)}.$$

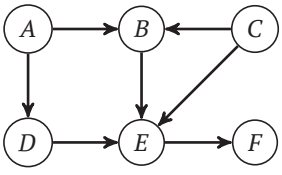
Мы можем переписать числитель как маргинализацию скрытых переменных, а знаменатель – как маргинализацию скрытых переменных и переменных запроса:

$$P(a^1|d^1) = \frac{\sum_b \sum_c P(a^1, b, c, d^1)}{\sum_a \sum_b \sum_c P(a, b, c, d^1)}.$$

Определение совместной вероятности как в числителе, так и в знаменателе можно переписать с использованием цепного правила для байесовских сетей, а полученное уравнение можно упростить, удалив константы из сумм:

$$\begin{aligned} P(a^1|d^1) &= \frac{\sum_b \sum_c P(a^1)P(b|a^1)P(c|b)P(d^1|c)}{\sum_a \sum_b \sum_c P(a)P(b|a)P(c|b)P(d^1|c)} \\ &= \frac{P(a^1)\sum_b \sum_c P(b|a^1)P(c|b)P(d^1|c)}{\sum_a \sum_b \sum_c P(a)P(b|a)P(c|b)P(d^1|c)} \\ &= \frac{P(a^1)\sum_b P(b|a^1)\sum_c P(c|b)P(d^1|c)}{\sum_a P(a)\sum_b \sum_c P(b|a)P(c|b)P(d^1|c)}. \end{aligned}$$

Упражнение 3.2. Исходя из изображенной ниже байесовской сети и связанных с ней условных распределений вероятностей, напишите уравнение точного вывода для запроса $P(c^1, d^1 \mid a^0, f^1)$.



Решение. Сначала мы расширяем выражение вывода, используя определение условного выражения.

$$P(c^1, d^1 | a^0, f^1) = \frac{P(a^0, c^1, d^1, f^1)}{P(a^0, f^1)}.$$

Мы можем переписать числитель как маргинализацию скрытых переменных, а знаменатель – как маргинализацию скрытых переменных и переменных запроса:

$$P(c^1, d^1 | a^0, f^1) = \frac{\sum_b \sum_e P(a^0, b, c^1, d^1, e, f^1)}{\sum_b \sum_c \sum_d \sum_e P(a^0, b, c, d, e, f^1)}.$$

Определение совместной вероятности как в числителе, так и в знаменателе можно переписать с использованием цепного правила для байесовских сетей, а полученное уравнение можно упростить, удалив константы из сумм. Обратите внимание, что существует несколько возможных порядков суммирования в окончательном уравнении:

$$\begin{aligned} P(c^1, d^1 | a^0, f^1) &= \frac{\sum_b \sum_e P(a^0) P(b|a^0, c^1) P(c^1) P(d^1|a^0) P(e|b, c^1, d^1) P(f^1|e)}{\sum_b \sum_c \sum_d \sum_e P(a^0) P(b|a^0, c) P(c) P(d|a^0) P(e|b, c, d) P(f^1|e)} \\ &= \frac{P(a^0) P(c^1) P(d^1|a^0) \sum_b \sum_e P(b|a^0, c^1) P(e|b, c^1, d^1) P(f^1|e)}{P(a^0) \sum_b \sum_c \sum_d \sum_e P(b|a^0, c) P(c) P(d|a^0) P(e|b, c, d) P(f^1|e)} \\ &= \frac{P(c^1) P(d^1|a^0) \sum_b P(b|a^0, c^1) \sum_e P(e|b, c^1, d^1) P(f^1|e)}{\sum_c P(c) \sum_b P(b|a^0, c) \sum_d P(d|a^0) \sum_e P(e|b, c, d) P(f^1|e)}. \end{aligned}$$

Упражнение 3.3. Предположим, что мы разрабатываем систему обнаружения объектов для автономного городского автомобиля. Сенсорная система нашего транспортного средства сообщает размер объекта S (маленький, средний или большой) и скорость V (медленно, умеренно или быстро). Мы хотим разработать модель, которая будет определять класс C объекта – транспортное средство, пешеход или мяч – с учетом размеров и скорости объекта. Если мы располагаем наивной байесовской моделью с априорными и условными распределениями класса в таблицах ниже, к какому классу относится обнаруженный объект с учетом наблюдений S = средний и V = низкая?

C	$P(C)$
Автомобиль	0.80
Пешеход	0.19
Мяч	0.01

C	S	$P(S\|C)$
Автомобиль	Маленький	0.001
Автомобиль	Средний	0.009
Автомобиль	Большой	0.990
Пешеход	Маленький	0.200
Пешеход	Средний	0.750
Пешеход	Большой	0.050
Мяч	Маленький	0.800
Мяч	Средний	0.199
Мяч	Большой	0.001

C	V	$P(V\|C)$
Автомобиль	Медленно	0.2
Автомобиль	Умеренно	0.2
Автомобиль	Быстро	0.6
Пешеход	Медленно	0.5
Пешеход	Умеренно	0.4
Пешеход	Быстро	0.1
Мяч	Медленно	0.4
Мяч	Умеренно	0.4
Мяч	Быстро	0.2

Решение. Чтобы вычислить апостериорное распределение $P(c|o_{1:n})$, мы используем определение совместного распределения для наивной байесовской модели в уравнении (3.4):

$$P(c|o_{1:n}) \propto P(c)\prod_{i=1}^{n} P(o_i|c);$$

P(автомобиль|средний, медленно) $\propto$ P(автомобиль)P(S = средний|автомобиль) P(низкий|автомобиль);

P(автомобиль|средний, медленно) $\propto$ (0.80)(0.009)(0.2) = 0.00144;

P(пешеход|средний, медленно) $\propto$ P(пешеход)P(S = средний|пешеход)P(V = медленно|пешеход);

P(пешеход|средний, медленно) $\propto$ (0.19)(0.75)(0.5) = 0.07125;

P(мяч|средний, медленно) $\propto$ P(мяч)P(S = средний|мяч)P(V = медленно|мяч);

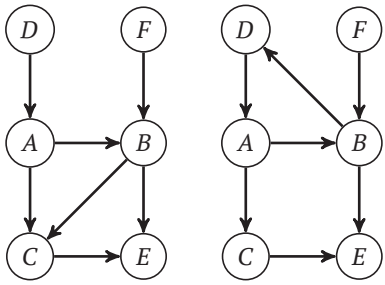
P(мяч|средний, медленно) $\propto$ (0.01)(0.199)(0.4) = 0.000796.

Поскольку вероятность P(пешеход|средний, медленно) имеет наибольшее значение, объект классифицируется как пешеход.

Упражнение 3.4. Исходя из формулы 3SAT в уравнении (3.14) и структуры байесовской сети на рис. 3.4, найдите значения $P(c_3^1|x_2^1, x_3^0, x_4^1)$ и $P(y^1|d_2^1, c_3^0)$.

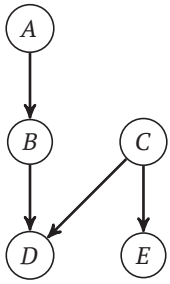
Решение. Мы имеем $P(c_3^1|x_2^1, x_3^0, x_4^1)$, потому что x_2^1, x_3^0, x_4^1 делают третью клаузу истинной, и $P(y^1|d_2^1, c_3^0) = 0$, потому что $Y = 1$ требует, чтобы и D_2, и C_3 были истинными.

Упражнение 3.5. Выполните топологическую сортировку для каждого из следующих ориентированных графов:



Решение. Существуют три допустимые топологические сортировки для первого ориентированного графа (байесовская сеть): (F, D, A, B, C, E), (D, A, F, B, C, E) и (D, F, A, B, C, E). Для второго ориентированного графа нет допустимых топологических сортировок, поскольку он циклический.

Упражнение 3.6. Предположим, что у нас есть изображенная ниже байесовская сеть, и мы нуждаемся в создании аппроксимации вывода $P(e^1|b^0, d^1)$ с использованием выборки, взвешенной по правдоподобию. Исходя из следующих выборок, запишите выражения для каждого веса выборки. Кроме того, напишите уравнение для оценки $P(e^1|b^0, d^1)$ в виде веса выборки w_i.



A	B	C	D	E
0	0	0	1	0
1	0	0	1	0
0	0	0	1	1
1	0	1	1	1
0	0	1	1	0
1	0	1	1	1

Решение. Для выборки, взвешенной по правдоподобию, веса выборки являются произведением распределений переменных наблюдений, обусловленных значениями их родителей. Следовательно, веса имеют общий вид $P(b^0|a)P(d^1|b^0, c)$. Затем мы сопоставляем каждое из значений для каждой выборки из совместного распределения:

A	B	C	D	E	Вес
0	0	0	1	0	$P(b^0\|a^0)P(d^1\|b^0, c^0)$
1	0	0	1	0	$P(b^0\|a^1)P(d^1\|b^0, c^0)$
0	0	0	1	1	$P(b^0\|a^0)P(d^1\|b^0, c^0)$
1	0	1	1	1	$P(b^0\|a^1)P(d^1\|b^0, c^1)$
0	0	1	1	0	$P(b^0\|a^0)P(d^1\|b^0, c^1)$
1	0	1	1	0	$P(b^0\|a^1)P(d^1\|b^0, c^1)$

Чтобы вычислить $P(e^1|b^0, d^1)$, нам просто нужно просуммировать веса выборок, соответствующие переменной запроса, и разделить результат на сумму всех весов:

$$P(e^1|b^0,d^1)=\frac{\sum_i w_i(e^{(i)}=1)}{\sum_i w_i}=\frac{w_3+w_4+w_6}{w_1+w_2+w_3+w_4+w_5+w_6}.$$

Упражнение 3.7. Каждый год мы получаем набор данных – оценки школьников за экзамены по математике M, чтению R и письму W по 100-балльной системе. Используя данные за предыдущие годы, мы создаем следующее распределение:

$$\begin{bmatrix} M \\ R \\ W \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 81 \\ 82 \\ 80 \end{bmatrix}, \begin{bmatrix} 25 & -9 & -16 \\ -9 & 36 & 16 \\ -16 & 16 & 36 \end{bmatrix}\right).$$

Вычислите параметры условного распределения баллов учащегося по математике и чтению при условии, что за экзамен по письму он получил 90 баллов.

Решение. Если мы примем, что $\mathbf{a}$ представляет собой вектор оценок по математике и чтению, а $\mathbf{b}$ представляет собой оценку по письму, совместное и условное распределения будут следующими:

$$\begin{bmatrix} \mathbf{a} \\ \mathbf{b} \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \boldsymbol{\mu}_a \\ \boldsymbol{\mu}_b \end{bmatrix}, \begin{bmatrix} \mathbf{A} & \mathbf{C} \\ \mathbf{C}^\top & \mathbf{B} \end{bmatrix}\right);$$

$p(\mathbf{a}|\mathbf{b}) = \mathcal{N}(\mathbf{a}|\boldsymbol{\mu}_{a|b}, \boldsymbol{\Sigma}_{a|b})$;

$\boldsymbol{\mu}_{a|b} = \boldsymbol{\mu}_a + \mathbf{C}\mathbf{B}^{-1}(\mathbf{b} - \boldsymbol{\mu}_b)$;

$\boldsymbol{\Sigma}_{a|b} = \mathbf{A} - \mathbf{C}\mathbf{B}^{-1}\mathbf{C}^\top$.

В условии нам даны следующие определения:

$$\boldsymbol{\mu}_a = \begin{bmatrix} 81 \\ 82 \end{bmatrix}, \quad \boldsymbol{\mu}_b = [80], \quad \mathbf{A} = \begin{bmatrix} 25 & -9 \\ -9 & 36 \end{bmatrix}, \quad \mathbf{B} = [36], \quad \mathbf{C} = \begin{bmatrix} -16 \\ 16 \end{bmatrix}.$$

Таким образом, мы получаем следующие параметры условного распределения при $\mathbf{b} = W = 90$:

$$\boldsymbol{\mu}_{M,R|W=90} = \begin{bmatrix} 81 \\ 82 \end{bmatrix} + \begin{bmatrix} -16 \\ 16 \end{bmatrix} \frac{1}{36}(90 - 80) \approx \begin{bmatrix} 76.5 \\ 86.4 \end{bmatrix};$$

$$\boldsymbol{\Sigma}_{M,R|W=90} = \begin{bmatrix} 25 & -9 \\ -9 & 36 \end{bmatrix} - \begin{bmatrix} -16 \\ 16 \end{bmatrix} \frac{1}{36} \begin{bmatrix} -16 & 16 \end{bmatrix} \approx \begin{bmatrix} 25 & -9 \\ -9 & 36 \end{bmatrix} - \begin{bmatrix} 7.1 & -7.1 \\ -7.1 & 7.1 \end{bmatrix} = \begin{bmatrix} 17.9 & -1.9 \\ -1.9 & 28.9 \end{bmatrix}.$$

Исходя из имеющегося условного распределения и из того, что ученик набрал 90 баллов за тест по письму, мы ожидаем, что он получит 76.5 балла за тест по математике со стандартным отклонением $\sqrt{17.9}$ и 86.4 балла за тест по чтению со стандартным отклонением $\sqrt{28.9}$.

4 Параметрическое обучение

До сих пор мы предполагали, что параметры и структура наших вероятностных моделей известны. В этой главе рассматривается задача обучения, или подбора параметров модели по наблюдаемым данным[1]. Мы начнем со знакомства с подходом, в котором находят параметры модели, обеспечивающие *максимальную вероятность наблюдения данных* (*максимальное правдоподобие*). После обсуждения ограничений такого подхода мы рассмотрим альтернативный байесовский подход, в котором начинают с распределения вероятностей по неизвестным параметрам, а затем обновляют это распределение на основе наблюдаемых данных с использованием законов вероятности. Главу завершает обсуждение вероятностных моделей, которые не ограничиваются фиксированным числом параметров.

4.1. Обучение по критерию максимального правдоподобия

При обучении параметров модели по критерию максимального правдоподобия мы пытаемся найти параметры распределения, которые максимизируют вероятность наблюдения данных. Если θ представляет параметры распределения, то *оценка максимального правдоподобия* вычисляется следующим образом:

$$\hat{\theta} = \arg\max_{\theta} P(D|\theta), \tag{4.1}$$

где $P(D|\theta)$ – это вероятность, которую наша стохастическая модель присваивает данным D, наблюдаемым, когда параметры модели принимают значение θ[2]. Здесь и далее мы будем использовать циркумфлекс ($\hat{\cdot}$), который часто называют «домиком» или «крышкой», для обозначения оценки параметра.

---

[1] В этой главе основное внимание уделяется обучению параметров модели на основе данных, что является важным компонентом области машинного обучения. Содержательное введение в эту область представлено в K. P. Murphy, *Probabilistic Machine Learning: An Introduction*. MIT Press, 2022.

[2] Здесь мы пишем $P(D|\theta)$, как будто это масса вероятности, связанная с дискретным распределением. Однако наша вероятностная модель может быть непрерывной, и в этом случае мы работаем с плотностью.

Есть две проблемы, связанные с изучением параметра максимального правдоподобия. Одна из них заключается в выборе подходящей вероятностной модели, с помощью которой мы определяем $P(D|\theta)$. Мы часто предполагаем, что выборки в наших данных D распределены *независимо и одинаково*, что означает, что наши выборки $D = o_{1:m}$ взяты из распределения $o_i \sim P(\cdot|\theta)$ с

$$P(D|\theta) = \prod_i P(o_i|\theta). \tag{4.2}$$

Стохастические модели могут быть основаны, например, на категориальных или гауссовых распределениях, упомянутых в предыдущих главах.

Другая проблема заключается в выполнении максимизации в уравнении (4.3). Для многих распространенных стохастических моделей мы можем выполнить эту оптимизацию аналитически. Но бывают и затруднительные случаи. Обычный подход заключается в максимизации *логарифмического правдоподобия* (log-likelihood), часто обозначаемого как $\ell(\theta)$. Поскольку логарифмическое преобразование монотонно возрастает, максимизация логарифмического правдоподобия эквивалентна максимизации правдоподобия[3]:

$$\hat{\theta} = \arg\max_{\theta} \sum_i \log P(o_i|\theta). \tag{4.3}$$

Вычисление суммы логарифмических правдоподобий обычно обладает гораздо более высокой численной устойчивостью по сравнению с перемножением малых вероятностных масс или плотностей. В оставшейся части этого раздела будет показано, как оптимизировать уравнение (4.3) для различных типов распределений.

4.1.1. Оценки максимального правдоподобия для категориальных распределений

Предположим, что случайная величина C показывает, приведет ли полет к столкновению в воздухе, и нас интересует оценка распределения $P(C)$. Поскольку C принимает значение 0 или 1, достаточно оценить параметр $\theta = P(c^1)$. Фактически нам нужно вывести θ из данных D. Допустим, у нас есть историческая база данных, охватывающая предыдущее десятилетие и включающая m полетов, из которых n закончились столкновением в воздухе. Здравый смысл, конечно же, подсказывает нам, что хорошей оценкой θ на основе таких данных D является n/m. Исходя из предположения о независимости исходов между полетами, мы можем найти вероятность последовательности из m исходов в D с n столкновениями в воздухе следующим образом:

$$P(D|\theta) = \theta^n(1 - \theta)^{m-n}. \tag{4.4}$$

---

[3] Хотя в этом уравнении не имеет значения, максимизируем ли мы натуральный логарифм (по основанию e) или десятичный логарифм (по основанию 10), в этой книге мы будем использовать запись $\log(x)$ для обозначения логарифма x по основанию e.

Оценка максимального правдоподобия $\hat{\theta}$ – это значение θ, которое максимизирует уравнение (4.4), что эквивалентно максимизации логарифмического правдоподобия:

$$\ell(\theta) = \log(\theta^n(1-\theta)^{m-n} \tag{4.5}$$

$$= n\log\theta + (m-n)\log(1-\theta). \tag{4.6}$$

Мы можем найти максимум функции обычным способом, приравняв первую производную ℓ к нулю, а затем найдя θ. Производная выглядит так:

$$\frac{\partial}{\partial\theta}\ell(\theta) = \frac{n}{\theta} - \frac{m-n}{1-\theta}. \tag{4.7}$$

Мы можем найти $\hat{\theta}$, приравняв эту производную к нулю:

$$\frac{n}{\hat{\theta}} - \frac{m-n}{1-\hat{\theta}} = 0. \tag{4.8}$$

Выполнив простые алгебраические преобразования, мы находим, что действительно $\hat{\theta} = n/m$.

Вычисление оценки максимального правдоподобия для переменной X, которая может принимать k значений, также не вызывает затруднений. Если $n_{1:k}$ – это наблюдаемые точки данных для k различных значений, то оценка максимального правдоподобия для $P(x^i|n_{1:k})$ определяется выражением

$$\hat{\theta}_i = \frac{n_i}{\sum_{j=1}^{k} n_j}. \tag{4.9}$$

4.1.2. Оценки максимального правдоподобия для распределений Гаусса

В распределении Гаусса логарифмическое правдоподобие среднего μ и дисперсии σ^2 для m выборок определяется выражением

$$\ell(\mu,\sigma^2) \propto -m\log\sigma - \frac{\sum_i (o_i-\mu)^2}{2\sigma^2}. \tag{4.10}$$

Снова приравняем производные по параметрам к нулю и найдем оценку максимального правдоподобия, решив следующие уравнения:

$$\frac{\partial}{\partial\mu}\ell(\mu,\sigma^2) = \frac{\sum_i (o_i-\hat{\mu})}{\hat{\sigma}^2} = 0; \tag{4.11}$$

$$\frac{\partial}{\partial \sigma}\ell(\mu,\sigma^2) = -\frac{m}{\hat{\sigma}} + \frac{\sum_i (o_i - \hat{\mu})^2}{\hat{\sigma}^3} = 0. \tag{4.12}$$

После алгебраических преобразований получаем:

$$\hat{\mu} = \frac{\sum_i o_i}{m}; \qquad \hat{\sigma}^2 = \frac{\sum_i (o_i - \hat{\mu})^2}{m}. \tag{4.13}$$

На рис. 4.1 показан пример подгонки кривой распределения Гаусса к данным.

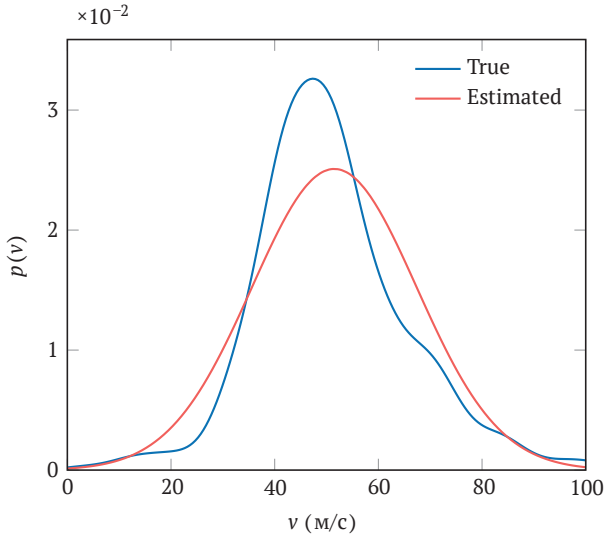


Рис. 4.1. Предположим, что у нас есть измерения скорости полета $o_{1:m}$ для m траекторий самолета и мы хотим обучить модель Гаусса. Здесь изображена гауссиана с оценками максимального правдоподобия $\hat{\mu} = 51.5$ м/с и $\hat{\sigma} = 15.9$ м/с. Для сравнения показано «истинное» распределение. В данном случае распределение Гаусса является вполне разумным приближением к истинному распределению

4.1.3. Оценки максимального правдоподобия для байесовских сетей

Мы можем применить метод обучения параметра максимального правдоподобия к байесовским сетям. Теперь мы будем предполагать, что наша сеть состоит из набора n дискретных переменных, которые мы обозначаем как $X_{1:n}$. Наши данные $D = \{\mathbf{o}_1, \ldots, \mathbf{o}_m\}$ состоят из наблюдаемых выборок этих перемен-

ных. В нашей сети со структурой G обозначим за r_i количество экземпляров X_i, а q_i – количество экземпляров родителей X_i. Если у X_i нет родителей, то $q_i = 1$. В свою очередь, j-й экземпляр родителей X_i обозначается как $\boldsymbol{\pi}_{ij}$.

Таблица коэффициентов (factor table) для X_i, таким образом, содержит элементы $r_i q_i$, в результате чего в нашей байесовской сети начитывается $\sum_{i=1}^{n} r_i q_i$ параметров. Каждый параметр записывается как θ_{ijk} и определяется как

$$P(X_i = k|\boldsymbol{\pi}_{ij}) = \theta_{ijk}. \tag{4.14}$$

Хотя имеется $\sum_{i=1}^{n} r_i q_i$ параметров, только $\sum_{i=1}^{n} (r_i - 1) q_i$ из них являются независимыми. Мы используем $\boldsymbol{\theta}$ для обозначения множества всех параметров.

Далее, мы обозначим за m_{ijk} количество раз, когда $X_i = k$ для данного родительского экземпляра j в наборе данных. В алгоритме 4.1 реализована функция для извлечения этих *подсчетов* (count) или статистики из набора данных. Правдоподобие выражается через m_{ijk} следующим образом:

$$P(D|\boldsymbol{\theta}, G) = \prod_{i=1}^{n} \prod_{j=1}^{q_i} \prod_{k=1}^{r_i} \theta_{ijk}^{m_{ijk}}. \tag{4.15}$$

Подобно оценке максимального правдоподобия для одномерного распределения в уравнении (4.9), оценка максимального правдоподобия в нашей модели дискретной байесовской сети имеет вид

$$\hat{\theta}_{ijk} = \frac{m_{ijk}}{\sum_{k'} m_{ijk}}. \tag{4.16}$$

Пример 4.1 иллюстрирует процесс извлечения статистической информации из набора данных.

Алгоритм 4.1. Функция для извлечения статистической информации из дискретного набора данных D, предполагающая байесовскую сеть с переменными `vars` и структурой `G`. Набор данных представляет собой матрицу размера $n \times m$, где n – количество переменных, а m – количество точек данных. Эта функция возвращает массив `M` длины n. i-й компонент состоит из матрицы подсчетов $q_i \times r_i$. Функция `sub2ind(siz, x)` возвращает линейный индекс в массиве с размерами, заданными в `siz` для заданных координат `x`. Он используется для определения того, какой родительский экземпляр имеет отношение к конкретной точке данных и переменной

```
function sub2ind(siz, x)
    k = vcat(1, cumprod(siz[1:end-1]))
    return dot(k, x .- 1) + 1
end
```

```
function statistics(vars, G, D::Matrix{Int})
    n = size(D, 1)
    r = [vars[i].r for i in 1:n]
    q = [prod([r[j] for j in inneighbors(G,i)]) for i in 1:n]
    M = [zeros(q[i], r[i]) for i in 1:n]
    for o in eachcol(D)
        for i in 1:n
            k = o[i]
            parents = inneighbors(G,i)
            j = 1
            if !isempty(parents)
                 j = sub2ind(r[parents], o[parents])
            end
            M[i][j,k] += 1.0
        end
    end
    return M
end
```

Пример 4.1. Использование функции `statistics` для извлечения статистической информации из набора данных. Байесовское параметрическое обучение можно использовать, чтобы избежать значений `NAN`, но мы должны указать априорные значения

Предположим, что у нас есть небольшая сеть $A \rightarrow B \leftarrow C$ и мы хотим извлечь статистическую информацию из матрицы данных `D`. Мы можем использовать следующий код:

```
G = SimpleDiGraph(3)
add_edge!(G, 1, 2)
add_edge!(G, 3, 2)
vars = [Variable(:A,2), Variable(:B,2), Variable(:C,2)]
D = [1 2 2 1; 1 2 2 1; 2 2 2 2]
M = statistics(vars, G, D)
```

Выход представляет собой массив `M`, состоящий из этих трех матриц подсчетов, каждая из которых имеет размер $q_i \times r_i$:

$$\begin{bmatrix}2 & 2\end{bmatrix}\begin{bmatrix}0 & 0\\ 0 & 0\\ 2 & 0\\ 0 & 2\end{bmatrix}\begin{bmatrix}0 & 4\end{bmatrix}.$$

Мы можем вычислить оценку максимального правдоподобия, нормализовав строки в матрицах массива `M`:

```
θ = [mapslices(x->normalize(x,1), Mi, dims=2) for Mi in M]
```

– и получаем в итоге

$$\begin{bmatrix}0.5 & 0.5\end{bmatrix}\begin{bmatrix}\text{NAN} & \text{NAN}\\ \text{NAN} & \text{NAN}\\ 1 & 0\\ 0 & 1\end{bmatrix}\begin{bmatrix}0 & 1\end{bmatrix}.$$

Здесь первый и второй родительские экземпляры второй переменной `B` приводят к оценкам `NAN` (not a number, «не число»). Поскольку в данных нет наблюдений за этими двумя экземплярами родителей, знаменатель уравнения (4.16) равен нулю, что делает оценку параметра неопределенной. Большинство других параметров не являются `NAN`. Например, параметр $\theta_{112} = 0.5$ означает, что оценка максимального правдоподобия $P(a^2)$ равна 0.5.

4.2. Байесовское параметрическое обучение

Байесовское параметрическое обучение (Bayesian parameter learning) устраняет некоторые недостатки оценки максимального правдоподобия, особенно когда количество данных ограничено. Например, предположим, что наша база данных по безопасности полетов была ограничена событиями прошлой недели, и за это время не случилось зарегистрированных столкновений в воздухе. Если θ – это вероятность того, что полет приведет к столкновению в воздухе, то оценка максимальной вероятности составит $\hat{\theta} = 0$. Вера в то, что вероятность столкновения в воздухе равна нулю, не является разумным выводом, если только наша предыдущая гипотеза не заключалась, например, в том, что все полеты абсолютно безопасны.

Байесовский подход к параметрическому обучению подразумевает оценку $p(\theta|D)$, апостериорного распределения по θ при наличии данных D. Вместо получения точечной оценки θ, как при оценке максимального правдоподобия, мы получаем распределение. Это распределение позволяет нам количественно определить неопределенность относительно истинного значения θ. Мы можем преобразовать это распределение в точечную оценку, вычислив математическое ожидание:

$$\hat{\theta} = \mathbb{E}_{\theta\sim p(\cdot|D)}[\theta] = \int \theta p(\theta|D)d\theta. \tag{4.17}$$

Однако в некоторых случаях математическое ожидание может оказаться неприемлемым, как показано на рис. 4.2. Альтернативой является использование *максимальной апостериорной оценки* (maximum a posteriori estimate):

$$\hat{\theta} = \arg\max_{\theta} p(\theta|D). \tag{4.18}$$

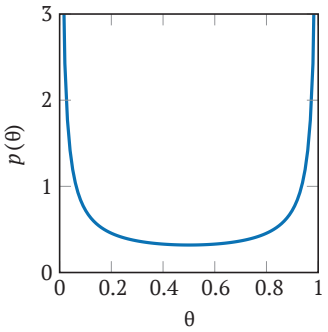


Рис. 4.2. Пример распределения, для которого ожидаемое значение θ не является хорошей оценкой. Ожидаемое значение 0.5 имеет меньшую плотность, чем экстремальные значения 0 или 1. Данное распределение является бета-распределением с параметрами (0.2, 0.2). Мы рассмотрим этот тип распределения немного позже

Эта оценка соответствует значению θ, которому присваивается наибольшая плотность. Его часто называют *модой* (mode) распределения. Как показано на рис. 4.2, мода может быть неуникальной.

Байесовское параметрическое обучение можно рассматривать как вывод в байесовской сети со структурой, показанной на рис. 4.3, которая предполагает, что наблюдаемые переменные условно независимы друг от друга. Как и в любой байесовской сети, мы должны определить $p(\theta)$ и $P(O_i|\theta)$. Мы часто используем равномерное априорное распределение вероятности $p(\theta)$. В оставшейся части этого раздела мы рассмотрим байесовское параметрическое обучение различных моделей $P(O_i|\theta)$.

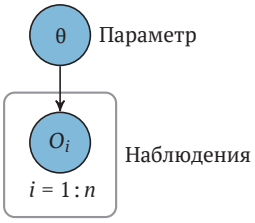


Рис. 4.3. Байесовская сеть, иллюстрирующая параметрическое обучение

4.2.1. Байесовское обучение для бинарных распределений

Предположим, мы хотим узнать параметры бинарного распределения. Здесь мы будем использовать $P(o^1|\theta) = \theta$. Чтобы вывести распределение по θ в байесовской сети, показанной на рис. 4.3, мы можем применить стандартный метод вывода, обсуждавшийся в главе 3. В данном случае мы предполагаем равномерное априорное распределение:

$$p(\theta|o_{1:m}) \propto p(\theta|o_{1:m}) \tag{4.19}$$

$$= p(\theta)\prod_{i=1}^{m} P(o_i|\theta) \tag{4.20}$$

$$= \prod_{i=1}^{m} P(o_i|\theta) \tag{4.21}$$

$$= \prod_{i=1}^{m} \theta^{o_i}(1-\theta)^{1-o_i} \tag{4.22}$$

$$= \theta^n(1-\theta)^{m-n}. \tag{4.23}$$

Апостериорное распределение пропорционально $\theta^n(1-\theta)^{m-n}$, где n – количество раз, когда $O_i = 1$. Чтобы найти нормировочную постоянную, мы интегрируем

$$\int_0^1 \theta^n(1-\theta)^{m-n}d\theta = \frac{\Gamma(n+1)\Gamma(m-n+1)}{\Gamma(m+2)}, \tag{4.24}$$

где Γ – *гамма-функция*. Гамма-функция используется для обобщения понятия факториала на множества действительных и комплексных значений аргумента. Если m – целое число, то $\Gamma(m) = (m-1)!$. С учетом нормировки получаем

$$p(\theta|o_{1:m}) = \frac{\Gamma(m+2)}{\Gamma(n+1)\Gamma(m-n+1)}\theta^n(1-\theta)^{m-n} \tag{4.25}$$

$$= \text{Beta}(\theta|n+1, m-n+1). \tag{4.26}$$

Бета-распределение Beta(α, β) определяется параметрами α и β. Различные кривые для этого распределения показаны на рис. 4.4. Распределение Beta(1,1) соответствует равномерному распределению от 0 до 1.

Распределение Beta(α, β) имеет среднее

$$\frac{\alpha}{\alpha+\beta}. \tag{4.27}$$

Когда α и β оба больше 1, мода имеет вид

$$\frac{\alpha-1}{\alpha+\beta-2}. \tag{4.28}$$

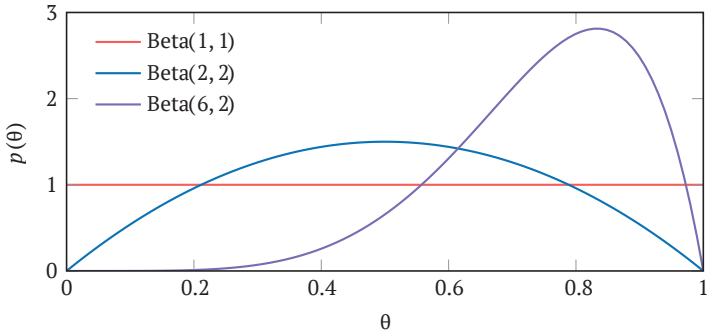


Рис. 4.4. Наложение кривых нескольких бета-распределений

Нужно отметить одно удобное свойство: если бета-распределение используется в качестве априорного бинарного распределения параметров, тогда апостериорное распределение также является бета-распределением. В частности, если априорное распределение задано как Beta(α, β) и мы делаем наблюдение o_i, то мы получаем апостериорное распределение Beta(α + 1, β), если $o_i = 1$,

и Beta(α, β + 1), если $o_i = 0$. Следовательно, если мы начали с априорного распределения, заданного Beta(α, β), и наши данные показали, что на m полетов приходится n столкновений, то апостериорное распределение вероятностей было бы задано как Beta(α + n, β + m – n). Параметры α и β априорного распределения иногда называют *псевдоподсчетами*, потому что они обрабатываются аналогично наблюдаемым подсчетам двух классов исходов в апостериорном распределении, хотя псевдоподсчеты не обязательно должны быть целыми числами.

Выбор априорного распределения вероятности, в принципе, должен осуществляться без знания данных, используемых для вычисления апостериорного распределения. Равномерные априорные распределения часто хорошо работают на практике, хотя при наличии экспертных знаний от них можно перейти к иным априорным распределениям. Например, предположим, что у нас есть слегка несимметричная монета и мы хотим оценить θ – вероятность того, что выпадет решка. Мы начинаем с предположения θ, которое приблизительно равно 0.5, а затем начинаем подбрасывать монету. Вместо того чтобы начинать с равномерного априорного распределения Beta(1, 1), мы могли бы использовать распределение Beta(2, 2) (показанное на рис. 4.4), которое придает больший вес значениям, близким к 0.5. Если бы мы были более уверены в оценке, близкой к 0.5, то могли бы уменьшить дисперсию априорного распределения, увеличив псевдоподсчеты. Априорное распределение Beta(10, 10) имеет гораздо более выраженный пик, чем Beta(2, 2). В целом, однако, значимость априорного распределения вероятности уменьшается с ростом объема данных, используемых для вычисления апостериорного распределения. Если мы наблюдаем m подбрасываний и выпало n орлов, то разница между Beta(1 + n, 1 + m – n) и Beta(10 + n, 10 + m – n) становится незначительнойпри условии исчисления количества подбрасываний в тысячах.

4.2.2. Байесовское обучение для категориальных распределений

Распределение Дирихле[4] является обобщением бета-распределения и может применяться для оценки параметров категориальных распределений. Предположим, что X – дискретная случайная величина, принимающая целые значения от 1 до n. Определим параметры распределения как $\theta_{1:n}$, где $P(x_i) = \theta_i$. Конечно, сумма параметров должна равняться 1, поэтому только первые n – 1 параметров независимы. Распределение Дирихле применяется для представления как априорного, так и апостериорного распределения и параметризуется как $\alpha_{1:n}$. Плотность определяется выражением

$$\mathrm{Dir}(\theta_{1:n}|\alpha_{1:n}) = \frac{\Gamma(\alpha_0)}{\prod_{i=1}^{n}\Gamma(\alpha_i)}\prod_{i=1}^{n}\theta_i^{\alpha_i-1}, \tag{4.29}$$

[4] Это распределение названо в честь немецкого математика Иоганна Петера Густава Лежена Дирихле (1805–1859).

где α_0 обозначает сумму параметров $\alpha_{1:n}$[5]. Если $n = 2$, то легко видеть, что уравнение (4.29) эквивалентно бета-распределению.

Обычно применяется равномерное априорное распределение, когда все параметры Дирихле $\alpha_{1:n}$ устанавливаются равными 1. Как и в случае с бета-распределением, параметры Дирихле часто называют *псевдоподсчетами*. Если априорное значение для $\theta_{1:n}$ задано как $\mathrm{Dir}(\alpha_{1:n})$ и имеется m_i наблюдений $X = i$, то апостериорное распределение определяется уравнением

$$p(\theta_{1:n}|\alpha_{1:n}, m_{1:n}) = \mathrm{Dir}(\theta_{1:n}|\alpha_1 + m_1, \ldots, \alpha_n + m_n). \tag{4.30}$$

Распределение $\mathrm{Dir}(\alpha_{1:n})$ имеет вектор средних значений, i-я компонента которого равна

$$\frac{\alpha_i}{\sum_{j=1}^{n} \alpha_j}. \tag{4.31}$$

При $\alpha_i > 1$ i-я компонента моды равна

$$\frac{\alpha_i - 1}{\sum_{j=1}^{n} \alpha_j - n}. \tag{4.32}$$

Мы выяснили, что оценка параметров байесовских распределений вероятности для бинарных и дискретных случайных величин не вызывает труда, поскольку это всего лишь подсчет различных исходов в данных. Правило Байеса можно использовать для вывода распределения параметров других параметризуемых распределений. В некоторых случаях – в зависимости от выбора априорного распределения и формы параметризуемого распределения – вычисление апостериорного распределения по пространству параметров также может быть выполнено аналитически.

4.2.3. Байесовское обучение для байесовских сетей

Мы можем применить байесовское параметрическое обучение к дискретным байесовским сетям. Априорное распределение байесовской сети с параметрами $\boldsymbol{\theta}$ можно разложить на множители следующим образом:

$$p(\boldsymbol{\theta}|G) = \prod_{i=1}^{n} \prod_{j=1}^{q_i} p(\boldsymbol{\theta}_{ij}), \tag{4.33}$$

где $\boldsymbol{\theta}_{ij} = (\theta_{ij1}, \ldots, \theta_{ijr_i})$. Можно показать, что априорное распределение $p(\boldsymbol{\theta}_{ij})$ при некоторых слабых предположениях следует распределению Дирихле $\mathrm{Dir}(\alpha_{ij1}, \ldots, \alpha_{ijr_i})$. В алгоритме 4.2 реализовано создание структуры данных из α_{ijk}, где все элементы равны 1, что соответствует равномерному априорному распределению.

---

[5] В приложении В представлены графики плотности распределения Дирихле для различных параметров.

После наблюдения за данными в форме подсчетов m_{ijk} (как представлено в разделе 4.1.3) апостериорное распределение имеет вид

$$p(\boldsymbol{\theta}_{ij}|\alpha_{ij}, m_{ij}) = \mathrm{Dir}(\boldsymbol{\theta}_{ij}|\alpha_{ij1} + m_{ij1}, \ldots, \alpha_{ijr_i} + m_{ijr_i}) \tag{4.34}$$

аналогично уравнению (4.30). Этот процесс продемонстрирован в примере 4.2.

Алгоритм 4.2. Функция для создания априорного распределения α_{ijk}, где все элементы равны 1. Массив матриц, который возвращает эта функция, имеет ту же форму, что и статистическая информация, сгенерированная алгоритмом 4.1. Для определения соответствующих размерностей функция принимает на вход список переменных `vars` и структуру `G`

```
function prior(vars, G)
    n = length(vars)
    r = [vars[i].r for i in 1:n]
    q = [prod([r[j] for j in inneighbors(G,i)]) for i in 1:n]
    return [ones(q[i], r[i]) for i in 1:n]
end
```

Пример 4.2. Вычисление апостериорных параметров в байесовской сети. Обратите внимание, что, в отличие от примера 4.1, здесь у нас нет значений `NAN`

Мы можем вычислить параметры апостериорного распределения, относящегося к байесовской сети, путем простого сложения априорных параметров и подсчетов (уравнение (4.34)). Если мы используем матрицу подсчетов `M`, полученную в примере 4.1, то можем добавить ее к матрицам априорных параметров `α = prior(vars, G)`, чтобы получить набор апостериорных параметров `M + α`:

$$\begin{bmatrix}3 & 3\end{bmatrix} \begin{bmatrix}1 & 1\\ 1 & 1\\ 3 & 1\\ 1 & 3\end{bmatrix} \begin{bmatrix}1 & 5\end{bmatrix}.$$

4.3. Непараметрическое обучение

В предыдущих двух разделах предполагалось, что вероятностная модель имеет фиксированную форму и что из данных должен быть получен фиксированный набор параметров. Альтернативный подход основан на *непараметрических* методах, в которых количество параметров зависит от объема данных. Распространенным непараметрическим методом является *ядерная оценка плотности*

(Метод окна Розенблатта–Парзена. – *Прим. перев.*), реализованная в алгоритме 4.3. Если даны наблюдения $o_{1:m}$, ядерная оценка плотности выглядит следующим образом:

$$p(x) = \frac{1}{m}\sum_{i=1}^{m}\varphi(x - o_i), \tag{4.35}$$

где φ – *кернфункция* (kernel function, ядерная функция), интеграл которой равен 1. Кернфункция используется для присвоения большей плотности значениям вблизи наблюдаемых точек данных. Кернфункция обычно симметрична, т. е. $\varphi(x) = \varphi(-x)$. Распространенным ядром является распределение Гаусса с нулевым средним. Когда используется такое ядро, стандартное отклонение часто называют *полосой пропускания* (bandwith), которую можно настроить для управления гладкостью функции плотности. Большая ширина полосы обычно приводит к более плавному распределению плотности. Для выбора подходящей полосы пропускания по имеющимся данным могут применяться байесовские методы. Влияние выбора полосы пропускания наглядно показано на рис. 4.5.

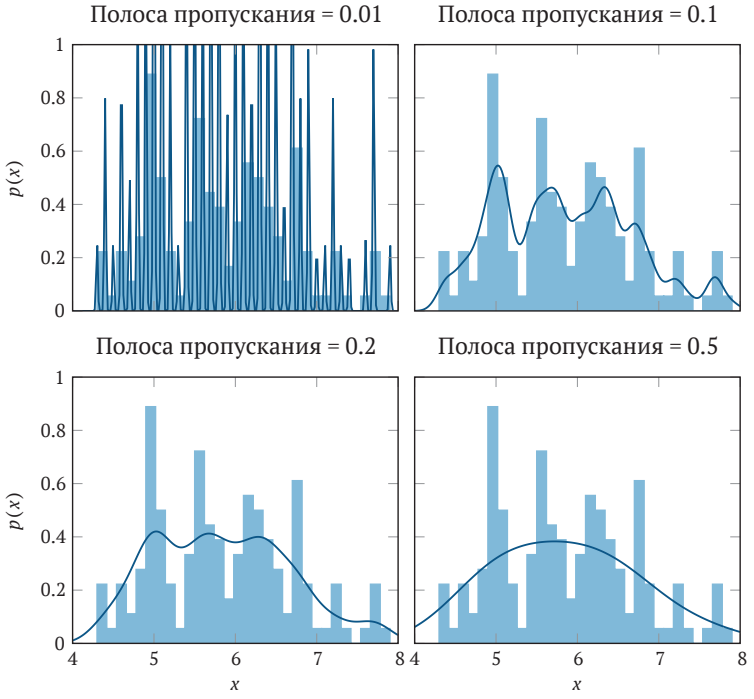


Рис. 4.5. Ядерная оценка плотности применялась к одному и тому же набору данных с использованием гауссовых ядер с нулевым средним и различной шириной полосы пропускания. На гистограмме синим цветом показаны частоты базовых наборов данных, а черными линиями – плотность вероятности из ядерной оценки плотности. Более широкие полосы пропускания сглаживают оценку, тогда как меньшие полосы пропускания могут вызвать переобучение по конкретным образцам

Алгоритм 4.3. Метод `gaussian_kernel` возвращает ядро Гаусса с нулевым средним значением $\varphi(x)$ и полосой пропускания `b`. Также для ядра φ и списка наблюдений `O` реализована ядерная оценка плотности

```
gaussian_kernel(b) = x->pdf(Normal(0,b), x)

function kernel_density_estimate(φ, O)
    return x -> sum([φ(x - o) for o in O])/length(O)
end
```

4.4. Обучение с отсутствующими данными

При обучении параметров вероятностной модели мы можем столкнуться с ситуацией, когда в данных отсутствуют отдельные записи[6]. Например, если мы проводим опрос, некоторые респонденты могут отказаться отвечать на определенные вопросы. В табл. 4.1 показан пример набора данных с отсутствующими элементами, включающими три бинарные переменные: *A*, *B* и *C*. Один из подходов к обработке отсутствующих данных заключается в отбрасывании всех *неполных* экземпляров, в которых имеется одна или несколько отсутствующих записей. В зависимости от того, сколько данных отсутствует, нам, возможно, придется отбросить большую их часть. В табл. 4.1 нам пришлось бы отбросить все строки, кроме первой, что вряд ли приемлемо.

A	*B*	*C*
1	1	0
?	1	1
1	?	?
?	?	?

Таблица 4.1. Пример данных, состоящих из четырех экземпляров с шестью пропущенными записями

Мы можем обучить параметры модели даже на неполных данных, используя метод максимального правдоподобия или байесовский подход. Если взять байесовский подход максимума апостериорного распределения, то нам нужно вычислить оценку

$$\hat{\theta} = \arg\max_{\theta} p(\theta \mid D_{\text{obs}}) \tag{4.36}$$

$$= \arg\max_{\theta} \sum_{D_{\text{mis}}} p(\theta | D_{\text{obs}}, D_{\text{mis}}) P(D_{\text{mis}} | D_{\text{obs}}), \tag{4.37}$$

[6] Обучение с отсутствующими данными является предметом большого количества публикаций. Всестороннее введение и обзор предоставлены в G. Molenberghs, G. Fitzmaurice, M. G. Kenward, A. Tsiatis, G. Verbeke, eds., *Handbook of Missing Data Methodology*. CRC Press, 2014.

где D_{obs} и D_{mis} состоят из всех наблюдаемых и отсутствующих данных соответственно. Если данные непрерывны, то сумму следует заменить интегралом. Маргинализация по отсутствующим данным может быть дорогостоящей в вычислительном отношении. Она также влияет на вычислительную гибкость байесовского подхода.

В этом разделе обсуждаются два общих подхода к обучению с отсутствующими данными без необходимости перечисления всех возможных комбинаций отсутствующих значений. Первый подход предусматривает изучение параметров распределения с использованием предсказанных значений отсутствующих записей. Второй основан на итеративном улучшении оценок параметров.

Мы будем рассматривать ситуацию, когда данные *отсутствуют случайным образом*, а это означает, что вероятность отсутствия записи не зависит от ее значения при заданных значениях наблюдаемых переменных. Примером ситуации, которая *не* соответствует этому допущению, могут быть радиолокационные данные, содержащие измерения расстояния до цели, но измерения могут отсутствовать либо из-за шума, либо из-за того, что цель находится за пределами диапазона обнаружения. В таком случае отсутствие записи с большой вероятностью будет обусловлено наблюдаемыми переменными уровня помех и расстояния. Анализ обусловленного отсутствия данных требует применения особых моделей и алгоритмов, отличных от тех, что мы здесь обсуждаем[7].

4.4.1. Подстановка данных

Альтернативой отбрасыванию неполных записей является подстановка отсутствующих значений в записях. *Подстановка данных* – это процесс вывода значений отсутствующих записей. Один из способов реализовать подстановку – это аппроксимация уравнения (4.37), где мы находим

$$\hat{D}_{\text{mis}} = \arg\max_{D_{\text{mis}}} p(D_{\text{mis}} | D_{\text{obs}}). \tag{4.38}$$

Получив значения для подстановки вместо отсутствующих, мы можем использовать эти данные для нахождения максимума апостериорного распределения:

$$\hat{\theta} = \arg\max_{\theta} p(\theta | D_{\text{obs}}) \approx \arg\max_{\theta} p(\theta | D_{\text{obs}}, \hat{D}_{\text{mis}}). \tag{4.39}$$

В качестве альтернативы можно использовать метод максимального правдоподобия.

Решение уравнения (4.38) может быть сложным с вычислительной точки зрения. Один простой подход к дискретным наборам данных состоит в том,

---

[7] Различные механизмы отсутствия и связанные с ними методы вывода рассмотрены в R. J. A. Little, D. B. Rubin, *Statistical Analysis with Missing Data,* 3rd ed. Wiley, 2020.

чтобы заменить отсутствующие записи наиболее часто наблюдаемым значением, называемым *предельной модой* (marginal mode). Например, в табл. 4.1 мы могли бы заменить все отсутствующие значения для столбца *A* на его предельную моду, равную 1.

Непрерывные данные часто не имеют повторов. Однако мы можем подогнать распределение к непрерывным значениям, а затем использовать моду полученного распределения. Например, мы можем подобрать распределение Гаусса по данным в табл. 4.2, а затем подставить вместо недостающих записей среднее наблюдаемых значений соответствующей переменной. Верхний левый график на рис. 4.6 иллюстрирует применение этого подхода к двумерным данным. Красные линии показывают, как записи с отсутствующим первым или вторым компонентом соотносятся с их подстановочными аналогами. Затем мы можем использовать наблюдаемые и подставленные данные, чтобы получить оценку максимального правдоподобия параметров совместного гауссова распределения. Как видно на рисунке, этот метод подстановки не всегда дает разумные прогнозы, а обученная модель довольно плохого качества.

A	*B*	*C*
−6.5	0.9	4.2
?	4.4	9.2
7.8	?	?
?	?	?

Таблица 4.2. Пример данных с непрерывными значениями

Часто мы можем добиться большего успеха, учитывая вероятностные отношения между наблюдаемыми и ненаблюдаемыми переменными. На рис. 4.6 четко прослеживается корреляция между двумя переменными; следовательно, знание значения одной переменной может помочь предсказать значение другой переменной. Обычный подход к подстановке, называемый *подстановкой по методу ближайшего соседа*, заключается в использовании значений, связанных с экземпляром, который является ближайшим в соответствии с мерой расстояния, применяемой для наблюдаемых переменных. Верхний правый график на рис. 4.6 иллюстрирует использование евклидова расстояния для нахождения подстановочных значений. Этот подход, как правило, обеспечивает более качественную подстановку и более точную подгонку полученных распределений.

Альтернативный подход заключается в том, чтобы подогнать распределение к полным наблюдаемым данным, а затем использовать это распределение для вывода пропущенных значений. Для вывода мы можем применить алгоритмы из предыдущей главы. Например, если мы располагаем дискретными данными и предположениями о структуре байесовской сети, можно воспользоваться исключением переменных или выборкой Гиббса, чтобы получить распределение по отсутствующим переменным из экземпляра наблюдаемых переменных. Исходя из этого распределения, мы можем использовать среднее значение или моду для подстановки пропущенных значений. В качестве альтернативы можно взять выборку из этого распределения. Если наши данные

непрерывны и предполагается, что в совокупности они являются гауссовыми, то для вывода апостериорного распределения по этим данным подходит алгоритм 3.11. Нижние графики на рис. 4.6 демонстрируют подстановку с использованием методов апостериорной моды и апостериорной выборки.

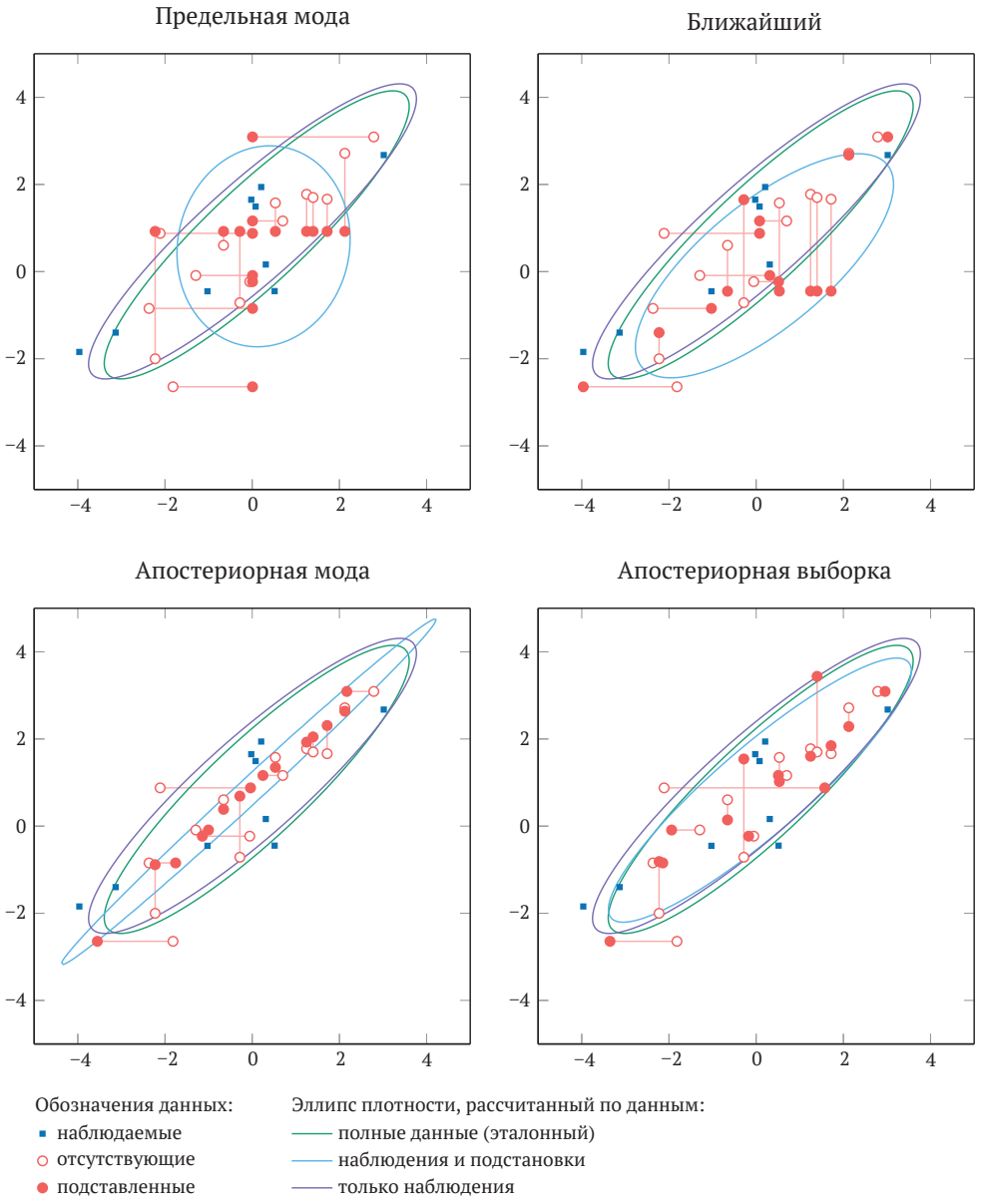


Рис. 4.6. Демонстрация методов подстановки.
Здесь эллипсы соответствуют точкам, где плотность оценки максимального правдоподобия совместного распределения равна 0.02

4.4.2. Алгоритм ожидания-максимизации

К категории методов *ожидания-максимизации* (expectation-maximization, EM) относится итеративное улучшение оценки параметра распределения $\hat{\theta}$[8]. Начальное значение $\hat{\theta}$ может быть предположением, случайно выбранным из априорного распределения или полученным с использованием одного из методов, рассмотренных в разделе 4.4.1. На каждой итерации мы выполняем процесс обновления $\hat{\theta}$, состоящий из двух шагов.

Сначала идет *шаг ожидания* (expectation step, E-step), где мы используем текущую оценку θ, чтобы вывести недостающие данные. Например, моделируя данные с помощью дискретной байесовской сети, мы можем использовать один из уже известных нам алгоритмов, чтобы вывести распределение отсутствующих записей для каждого экземпляра. При извлечении подсчета мы применяем взвешивание, пропорциональное вероятности заполняющего значения, как показано в примере 4.3. В случаях, когда пропущено много переменных, у нас может оказаться слишком много возможных вариантов заполняющих значений для перебора, что делает более привлекательным подход, основанный на выборке. Выборка также подходит в качестве метода аппроксимации, когда наши переменные непрерывны.

Затем следует *шаг максимизации* (maximization step, M-step), когда мы пытаемся найти новую оценку $\hat{\theta}$, максимизирующую правдоподобие завершенных данных. Если у нас есть дискретная байесовская сеть со взвешенными подсчетами в форме, показанной в примере 4.3, то мы можем выполнить оценку максимального правдоподобия, упомянутую ранее в этой главе. При наличии знания априорного распределения мы можем применить метод оценки с помощью апостериорного максимума.

Этот подход не гарантирует сходимости к параметрам модели, которые максимизируют вероятность наблюдаемых данных, но он может неплохо работать на практике. Чтобы уменьшить риск сходимости алгоритма только к локальному оптимуму, мы можем запустить алгоритм из множества различных начальных точек в пространстве параметров. Мы просто выбираем в конце результирующую оценку параметра, которая максимизирует правдоподобие.

Алгоритм ожидания-максимизации подходит даже для подстановки значений переменных, которые вообще не наблюдаются в данных. Такие переменные называются *скрытыми переменными* (latent variable). В качестве примера предположим, что у нас есть байесовская сеть $Z \rightarrow X$, где величина X непрерывна, а Z дискретна и может принимать одно из трех значений. Наша модель предполагает, что $p(x|z)$ является условным гауссовым распределением. Допустим, набор данных содержит значения только для X и не содержит для Z. Мы располагаем начальным $\hat{\theta}$ и используем его для вывода распределения вероятностей по значениям Z, исходя из значения X для каждой записи. Распределение заполняющих значений ввода затем используется для обновления

[8] Максимизация ожидания была введена в работе A. P. Dempster, N. M. Laird, D. B. Rubin, *Maximum Likelihood from Incomplete Data via the EM Algorithm,* Journal of the Royal Statistical Society, Series B (Methodological), vol. 39, no. 1, pp. 1–38, 1977.

нашей оценки параметров распределения вероятностей $P(Z)$ и $P(X|Z)$, как показано в примере 4.4. Итерации выполняют до сходимости, что часто происходит очень быстро. Параметры распределения, которые мы получаем в этом примере, определяют смешанную модель Гаусса, которая была рассмотрена в разделе 2.2.2.

Пример 4.3. Расширение неполного набора данных с использованием предполагаемых параметров модели

Предположим, что у нас есть бинарная байесовская сеть с $A \rightarrow B$. Начнем с предположения, что $\hat{\boldsymbol{\theta}}$ устанавливает

$$P(a^1) = 0.5, \qquad P(b^1|a^0) = 0.2, \qquad P(b^1|a^1) = 0.6.$$

Используя эти параметры, мы можем дополнить набор данных с пропущенными значениями (слева) до взвешенного набора данных со всеми возможными отдельными дополнениями (справа):

A	B
1	1
0	1
0	?
?	0

A	B	Вес
1	1	1
0	1	1
0	0	$1 - P(b^1\|a^0) = 0.8$
0	1	$P(b^1\|a^0) = 0.2$
0	0	$\alpha P(a^0)P(b^0\|a^0) = \alpha 0.4 = 2/3$
1	0	$\alpha P(a^1)P(b^0\|a^1) = \alpha 0.2 = 1/3$

Здесь α является константой нормализации, которая обеспечивает дополнение каждой записи до такой, чья сумма весов равна 1. Отсюда матрицы подсчетов имеют следующий вид:

$$\begin{bmatrix}(2+2/3) & (1+1/3)\end{bmatrix} \begin{bmatrix}(0.8+2/3) & 1.2 \\ 1/3 & 1\end{bmatrix}.$$

Пример 4.4. Максимизация ожидания применительно к обучению параметров гауссовой смешанной модели

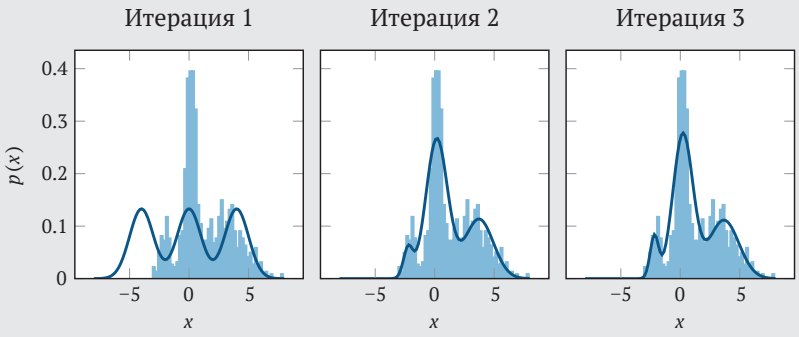
У нас есть байесовская сеть $Z \rightarrow X$, где Z – дискретная скрытая переменная с тремя значениями, а X представляет собой непрерывное распределение $p(x|z)$, моделируемое как условное гауссово распределение. Следовательно, у нас есть параметры, определяющие $P(z^1)$, $P(z^2)$ и $P(z^3)$, а также μ_i и σ_i для каждого из трех гауссовых распределений, связанных с различными значениями Z. В этом примере мы используем начальный вектор параметров $\hat{\boldsymbol{\theta}}$, который задает $P(z^i) = 1/3$ и $\sigma_i = 1$ для всех i, а также дополняем средние значения $\mu_1 = -4$, $\mu_2 = 0$ и $\mu_3 = 4$.

Предположим, что первый экземпляр в наших данных имеет $X = 4.2$. Нам нужно вычислить распределение по Z для этого экземпляра:

$$P(z^i|X = 4.2) = \frac{P(z^i)\mathcal{N}(4.2|\mu_i, \sigma_i^2)}{\sum_j P(z^i)\mathcal{N}(4.2|\mu_j, \sigma_j^2)}.$$

Аналогичным образом вычислим это распределение для всех экземпляров в наборе данных. Получив взвешенные дополнения, мы можем найти новую оценку $\hat{\boldsymbol{\theta}}$. Вычислим $P(z^i)$, взяв среднее значение по экземплярам в наборе данных. Для нахождения μ_i и σ_i воспользуемся средними значениями и стандартными отклонениями X по экземплярам в наборе данных, взвешенными по вероятности z^i, соответствующей различным экземплярам.

Будем выполнять итерации до схождения. На рисунке ниже показаны три итерации. Гистограмма сгенерирована из значений X. Темно-синей линией показана предполагаемая функция плотности распределения. К третьей итерации параметры нашей смешанной гауссовой модели достаточно точно соответствуют истинному распределению данных.



4.5. Заключение

- Параметрическое обучение представляет собой вывод параметров вероятностной модели из данных.
- Метод максимального правдоподобия в параметрическом обучении основан на максимизации функции правдоподобия, что для некоторых моделей может быть выполнено аналитически.
- Байесовский подход к параметрическому обучению основан на выводе распределения вероятностей по основному параметру с использованием правила Байеса.
- Бета-распределения и распределения Дирихле являются примерами байесовских априорных распределений, которые легко обновлять с помощью данных.

- В отличие от параметрического обучения, которое предполагает фиксированную параметризацию вероятностной модели, непараметрическое обучение использует представления, которые растут с увеличением объема данных.
- Для решения проблемы параметрического обучения в условиях неполных данных можно использовать, например, подстановку данных или ожидание-максимизацию, когда мы делаем выводы на основе наблюдаемых значений.

4.6. Упражнения

Упражнение 4.1. Предположим, что Анна выполняет штрафные броски в баскетболе. Пока мы не увидели ее игру, мы исходим из равномерного априорного распределения вероятности того, что она успешно попадет в корзину за один бросок. Затем мы видим, как она делает три броска, два из которых удачные. Какова, по вашему мнению, вероятность того, что следующий бросок будет удачным?

Решение. Обозначим вероятность попадания в корзину за θ. Поскольку мы начали с равномерного априорного распределения Beta(1, 1) и наблюдали два попадания и один промах, нашим апостериорным распределением будет Beta(1 + 2, 1 + 1) = Beta(3, 2). Далее вычислим вероятность попадания следующим образом:

$$P(\text{попадание}) = \int P(\text{попадание}|\theta)\text{Beta}(\theta|3.2)d\theta = \int \theta\text{Beta}(\theta|3.2)d\theta.$$

Это выражение является просто ожиданием (или средним) бета-распределения, которое дает нам $P(\text{попадание}) = 3/5$.

Упражнение 4.2. Рассмотрим непрерывную случайную величину X, которая следует *распределению Лапласа* с параметрами μ и b и с плотностью вероятности

$$p(x|\mu, b) = \frac{1}{2b}\exp\left(-\frac{|x-\mu|}{b}\right).$$

Вычислите оценки максимального правдоподобия параметров распределения Лапласа для набора данных D, состоящего из m независимых наблюдений $x_{1:m}$. Обратите внимание, что $\partial|u|/\partial x = \text{sign}(u)\partial u/\partial x$, где функция sign возвращает знак своего аргумента.

Решение. Поскольку наблюдения независимы, мы можем записать функцию логарифмического правдоподобия в виде суммы:

$$\begin{aligned}\ell(\mu, b) &= \sum_{i=1}^{m} \log\left[\frac{1}{2b}\exp\left(-\frac{|x_i-\mu|}{b}\right)\right] \\ &= -\sum_{i=1}^{m}\log 2b - \sum_{i=1}^{m}\frac{|x_i-\mu|}{b} \\ &= -m\log 2b - \frac{1}{b}\sum_{i=1}^{m}|x_i-\mu|.\end{aligned}$$

Чтобы получить оценки максимального правдоподобия истинных параметров μ и b, мы берем частные производные логарифмического правдоподобия по каждому из параметров, приравниваем их к нулю и решаем уравнения для каждого параметра. Сначала находим $\hat{\mu}$:

$$\begin{aligned}&\frac{\partial}{\partial\mu}\ell(\mu, b) = \frac{1}{\hat{b}}\sum_{i=1}^{m}\text{sign}(x_i-\mu); \\ &0 = \frac{1}{\hat{b}}\sum_{i=1}^{m}\text{sign}(x_i-\hat{\mu}); \\ &0 = \sum_{i=1}^{m}\text{sign}(x_i-\hat{\mu}); \\ &\hat{\mu} = \text{median}(x_{1:m}).\end{aligned}$$

Теперь найдем $\hat{b}$:

$$\begin{aligned}&\frac{\partial}{\partial b}\ell(\mu, b) = -\frac{m}{b} + \frac{1}{b^2}\sum_{i=1}^{m}|x_i-\hat{\mu}|; \\ &0 = -\frac{m}{b} + \frac{1}{\hat{b}^2}\sum_{i=1}^{m}|x_i-\hat{\mu}|; \\ &\frac{m}{\hat{b}} = \frac{1}{\hat{b}^2}\sum_{i=1}^{m}|x_i-\hat{\mu}|; \\ &\hat{b} = \frac{1}{m}\sum_{i=1}^{m}|x_i-\hat{\mu}|.\end{aligned}$$

Следовательно, оценки максимального правдоподобия для параметров распределения Лапласа равны $\hat{\mu}$ – медиане наблюдений и $\hat{b}$ – среднему значению абсолютных отклонений от медианы.

Упражнение 4.3. Рассмотрим применение оценки максимального правдоподобия к *цензурированным данным* (censored data), где некоторые измерения известны лишь частично. Предположим, что вы производите электродвигатели для дрона-квадрокоптера и хотите создать модель того, как долго они прослу-

жат до отказа (отказ хотя бы одного двигателя неизбежно приводит к падению дрона). Хотя существуют более подходящие распределения для моделирования надежности компонентов[9], мы будем использовать *экспоненциальное распределение*, параметризованное λ с функцией плотности вероятности $\lambda \exp(-\lambda x)$ и кумулятивной функцией распределения $1 - \exp(-\lambda x)$. Вы запустили пять дронов. У троих из них отказали моторы через 132 часа, 42 часа и 89 часов. Вы прекратили тестирование двух других после 200 часов безотказной работы. Следовательно, вы не знаете их срок службы – просто известно, что он превышает 200 часов. Какова оценка максимального правдоподобия для λ, исходя из этих данных?

Решение. Эта задача имеет $n = 3$ полностью наблюдаемых измерения и $m = 2$ цензурированных измерения. Будем использовать t_i для обозначения i-го полностью наблюдаемого измерения и $\underline{t}_j$ для j-го цензурированного измерения. Вероятность одиночного измерения, превышающего $\underline{t}_j$, является дополнением к кумулятивной функции распределения, которая представляет собой просто $\exp(-\lambda \underline{t}_j)$. Следовательно, правдоподобие данных вычисляется следующим образом:

$$\left(\prod_{i=1}^{n} \lambda e^{-\lambda t_i}\right)\left(\prod_{j=1}^{m} e^{-\lambda t_{-j}}\right).$$

Воспользуемся стандартным подходом максимизации логарифмического правдоподобия, которое определяется выражением

$$\ell(\lambda) = \sum_{i=1}^{n} (\log \lambda - \lambda t_i) + \sum_{j=1}^{m} -\lambda \underline{t}_j.$$

Производная по λ имеет следующий вид:

$$\frac{\partial \ell}{\partial \lambda} = \frac{n}{\lambda} - \sum_{i=1}^{n} t_i - \sum_{j=1}^{m} \underline{t}_j.$$

Приравняв эту производную к 0, мы можем решить уравнение для λ, чтобы найти оценку максимального правдоподобия:

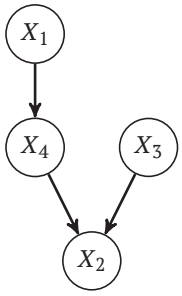
$$\hat{\lambda} = \frac{n}{\sum_{i=1}^{n} t_i + \sum_{j=1}^{m} t_{-j}} = \frac{3}{132 + 42 + 89 + 200 + 200} \approx 0.00452.$$

Среднее значение экспоненциального распределения равно $1/\lambda$. Отсюда следует, что среднее значение наработки мотора до отказа равно 221 часу.

---

[9] K. S. Trivedi, A. Bobbio, *Reliability and Availability Engineering.* Cambridge University Press, 2017.

Упражнение 4.4. У нас есть байесовская сеть, в которой переменные $X_{1:3}$ могут принимать значения из набора {1, 2}, а X_4 могут принимать значения из набора {1, 2, 3}. Пусть у нас есть набор данных D наблюдений $o_{1:m}$, показанный ниже. Найдите оценки максимального правдоподобия соответствующих параметров условного распределения $\boldsymbol{\theta}$.



$$D = \begin{bmatrix} 1 & 2 & 1 & 1 & 1 & 2 & 1 & 2 & 1 & 1 \\ 2 & 2 & 2 & 1 & 2 & 1 & 1 & 1 & 2 & 1 \\ 2 & 2 & 2 & 1 & 1 & 1 & 1 & 1 & 2 & 1 \\ 3 & 2 & 1 & 1 & 1 & 3 & 3 & 1 & 1 & 1 \end{bmatrix}$$

Решение. Сгенерируем матрицы подсчетов $\mathbf{M}_i$ размером $q_i \times r_i$ для каждого узла, перебирая набор данных и сохраняя подсчеты. Затем нормализуем каждую строку в матрицах, чтобы получить матрицы, содержащие оценки максимального правдоподобия параметров:

$$\mathbf{M}_1 = \begin{bmatrix} 7 & 3 \end{bmatrix}; \quad \mathbf{M}_2 = \begin{bmatrix} 3 & 1 \\ 0 & 0 \\ 2 & 0 \\ 0 & 2 \\ 0 & 1 \\ 0 & 1 \end{bmatrix}; \quad \mathbf{M}_3 = \begin{bmatrix} 6 & 4 \end{bmatrix}; \quad \mathbf{M}_4 = \begin{bmatrix} 5 & 0 & 2 \\ 1 & 1 & 1 \end{bmatrix};$$

$$\hat{\theta}_1 = \begin{bmatrix} 0.7 & 0.3 \end{bmatrix}; \quad \hat{\theta}_2 = \begin{bmatrix} 0.75 & 0.25 \\ \text{NAN} & \text{NAN} \\ 1.0 & 0.0 \\ 0.0 & 1.0 \\ 0.0 & 1.0 \\ 0.0 & 1.0 \end{bmatrix}; \quad \hat{\theta}_3 = \begin{bmatrix} 0.6 & 0.4 \end{bmatrix}; \quad \hat{\theta}_4 \approx \begin{bmatrix} 0.71 & 0.0 & 0.29 \\ 0.33 & 0.33 & 0.34 \end{bmatrix}.$$

Упражнение 4.5. У нас есть несимметричная монета, и мы хотим оценить параметр Бернулли φ, определяющий вероятность того, что выпадет орел. Пусть при первом броске выпал орел ($o_1 = 1$). Ответьте на следующие вопросы:

- Какова оценка φ по методу максимального правдоподобия?
- Какова максимальная апостериорная оценка φ, если исходить из равномерного априорного распределения?
- Каково ожидание апостериорного распределения по φ, если исходить из равномерного априорного распределения?

Решение. Поскольку при первом броске выпал орел, у нас $n = 1$ успех и $m = 1$ попытка.

- Оценка φ по методу максимального правдоподобия равна $n/m = 1$.
- При использовании равномерного априорного распределения Beta(1, 1) апостериорное распределение равно Beta(1 + n, 1 + m – n) = Beta(2, 1). Максимальная апостериорная оценка φ, или мода апостериорного распределения, равна

$$\frac{\alpha-1}{\alpha+\beta-2}=\frac{2-1}{2+1-2}=1.$$

- Среднее апостериорного распределения равно

$$\frac{\alpha}{\alpha+\beta}=\frac{2}{2+1}=\frac{2}{3}.$$

Упражнение 4.6. Предположим, у вас есть следующий набор данных с одним пропущенным значением. Какое значение будет подставлено с использованием метода предельной моды, если предположить, что маргинальное распределение является гауссовым? Какое значение будет подставлено с использованием метода ближайшего соседа?

X_1	X_2
0.5	1.0
?	0.3
–0.6	–0.3
0.1	0.2

Решение. Предполагая, что маргинальное распределение по X_1 является гауссовым, мы можем вычислить предельную моду, которая является средним гауссова распределения:

$$\mu=\frac{1}{m}\sum_{i=1}^{m}x_i=\frac{0.5-0.6+0.1}{3}=0.$$

Таким образом, для подстановки методом маргинальной моды мы получаем значение, равное 0. Для подстановки по методу ближайшего соседа ближайшей выборкой к $X_2 = 0.3$ будет четвертая выборка, поэтому вместо отсутствующего значения будет подставлено 0.1.

Упражнение 4.7. Предположим, у вас есть набор данных по двум переменным $X_{1:2}$ с несколькими пропущенными значениями. Мы предполагаем, что $X_{1:2}$ соответствуют совместному многомерному гауссову распределению, и используем полностью наблюдаемые выборки, чтобы подогнать модель к следующему распределению:

$$\begin{bmatrix} X_1 \\ X_2 \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 5 \\ 2 \end{bmatrix}, \begin{bmatrix} 4 & 1 \\ 1 & 2 \end{bmatrix}\right).$$

Какое значение будет подставлено в X_1 для выборки $X_2 = 1.5$ с использованием метода апостериорной моды? Из какого распределения нам нужно произвести выборку для подстановки методом апостериорной выборки?

Решение. Поскольку мы предположили, что $X_{1:2}$ соответствуют совместному гауссову распределению, апостериорное распределение по X_1 при заданном X_2 также является гауссовым, а его мода является средним апостериорного распределения. Мы можем вычислить среднее значение апостериорного распределения следующим образом:

$$p(x_1|x_2) = \mathcal{N}(x_1|\mu_{x_1|x_2}, \sigma^2_{x_1|x_2});$$

$$\mu_{x_1|x_2=1.5} = 5 + (1)(2)^{-1}(1.5 - 2) = 4.75.$$

Следовательно, в соответствии с методом апостериорной моды будет подставлено значение 4.75. Для подстановки методом апостериорной выборки следует выбрать значение $X_1 \sim \mathcal{N}(4.75, 3.5)$.

5 Структурное обучение

В предыдущих главах этой книги предполагалось, что структуры наших вероятностных моделей известны. В этой главе обсуждаются методы обучения структур моделей на основе данных[1]. Мы начнем с методов вычисления вероятности графовой структуры исходя из данных. Как правило, при построении модели необходимо максимизировать эту вероятность. Поскольку пространство возможных графовых структур обычно слишком обширно для прямого перебора, мы также обсудим способы эффективного поиска в этом пространстве.

5.1. Оценка байесовской сети

Нам необходимо иметь возможность оценивать сетевую структуру G исходя из того, насколько хорошо она моделирует данные D. Метод максимума апостериорной вероятности применительно к структурному обучению заключается в нахождении структуры G, при которой значение $P(G|D)$ максимально. Мы начнем с методов вычисления байесовской оценки на основе $P(G|D)$, свидетельствующей о том, насколько хорошо структура G моделирует данные. Затем рассмотрим методику поиска сети с наивысшей оценкой в пространстве сетей. Как и в случае вывода байесовских сетей, можно показать, что в общем случае обучение структуры байесовской сети является NP-трудным[2].

Вычислим $P(G|D)$, используя правило Байеса и закон полной вероятности:

$$P(G|D) \propto P(G)P(D|G) \tag{5.1}$$

$$= P(G)\int P(D|\boldsymbol{\theta},G)p(\boldsymbol{\theta}|G)\mathrm{d}\boldsymbol{\theta}, \tag{5.2}$$

---

[1] Обзоры методов обучения байесовской сетевой структуры можно найти в следующих учебниках: D. Koller, N. Friedman, *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009. R. E. Neapolitan, *Learning Bayesian Networks*. Prentice Hall, 2003.

[2] См. D. M. Chickering, *Learning Bayesian Networks is NP-Complete*, в Learning from Data: Artificial Intelligence and Statistics V, D. Fisher, H.-J. Lenz, eds., Springer, 1996, pp. 121–130. D. M. Chickering, D. Heckerman, C. Meek, *Large-Sample Learning of Bayesian Networks is NP-Hard*, Journal of Machine Learning Research, vol. 5, pp. 1287–1330, 2004.

где **θ** содержит сетевые параметры, представленные в предыдущей главе. Интегрирование по **θ** дает нам[3]

$$P(G|D) = P(G)\prod_{i=1}^{n}\prod_{j=1}^{q_i}\frac{\Gamma(\alpha_{ij0})}{\Gamma(\alpha_{ij0}+m_{ij0})}\prod_{k=1}^{r_i}\frac{\Gamma(\alpha_{ijk}+m_{ijk})}{\Gamma(\alpha_{ijk})}, \tag{5.3}$$

где значения α_{ijk} – это псевдоподсчеты, а m_{ijk} – подсчеты, как было определено в предыдущей главе. Мы также определяем значения

$$\alpha_{ij0} = \sum_{k=1}^{r_i}\alpha_{ijk}; \qquad m_{ij0} = \sum_{k=1}^{r_i} m_{ijk}. \tag{5.4}$$

Нахождение структуры G, обеспечивающей максимум уравнения (5.2), аналогично нахождению G, дающей максимальное значение *байесовской оценки* (Bayesian score):

$$\log P(G|D) = \log P(G) + \sum_{i=1}^{n}\sum_{j=1}^{q_i}\left(\log\left(\frac{\Gamma(\alpha_{ij0})}{\Gamma(\alpha_{ij0}+m_{ij0})}\right)\right) + \sum_{k=1}^{r_i}\left(\log\left(\frac{\Gamma(\alpha_{ijk}+m_{ijk})}{\Gamma(\alpha_{ijk})}\right)\right). \tag{5.5}$$

Байесовскую оценку удобнее находить численно, потому что проще складывать логарифмы небольших чисел, чем перемножать эти небольшие числа, рискуя вычислительной стабильностью. Многие программные библиотеки могут напрямую вычислять логарифм гамма-функции.

В литературе представлены исследования различных априорных графовых структур, хотя на практике часто используется равномерное априорное распределение, и в этом случае $\log P(G)$ можно исключить из расчета байесовской оценки в уравнении (5.5). Вычисление байесовской оценки реализовано в алгоритме 5.1.

Алгоритм 5.1. Алгоритм вычисления байесовской оценки для списка переменных `vars` и графа `G` с данными `D`. Этот метод использует равномерное априорное распределение $\alpha_{ijk} = 1$ для всех i, j и k, сгенерированное алгоритмом 4.2. Функция `loggamma` импортирована из `SpecialFunctions.jl`. Функции `statistics` и `prior` представлены в главе 4. Обратите внимание, что $\log(\Gamma(\alpha)/\Gamma(\alpha + m)) = \log\Gamma(\alpha) - \log\Gamma(\alpha + m)$ и что $\log\Gamma(1) = 0$

```
function bayesian_score_component(M, α)
    p = sum(loggamma.(α + M))
    p -= sum(loggamma.(α))
    p += sum(loggamma.(sum(α,dims=2)))
    p -= sum(loggamma.(sum(α,dims=2) + sum(M,dims=2)))
```

---

[3] Вывод приведен в приложении к G. F. Cooper and E. Herskovits, *A Bayesian Method for the Induction of Probabilistic Networks from Data*, Machine Learning , vol. 4, no. 9, pp. 309–347, 1992.

```
    return p
end

function bayesian_score(vars, G, D)
    n = length(vars)
    M = statistics(vars, G, D)
    α = prior(vars, G)
    return sum(bayesian_score_component(M[i], α[i]) for i in 1:n)
end
```

У оптимизации структуры сети на основе байесовской оценки есть полезный побочный эффект – мы можем найти правильный баланс сложности модели с учетом доступных данных. Нам не нужна модель, которая упускает важные взаимосвязи между переменными, но также не имеет смысла напрасно тратить усилия на обучение модели, имеющей слишком много параметров для доступных данных.

Чтобы показать на примере, как байесовская оценка помогает сбалансировать сложность модели, рассмотрим сеть, изображенную на рис. 5.1. Значение A слабо влияет на значение B, а C не зависит от других переменных. Мы делаем выборку из этой «истинной» модели для генерации данных D, а затем пытаемся изучить структуру модели. Существует 25 возможных сетевых структур, содержащих три переменные, но мы сосредоточимся на оценках моделей, изображенных на рис. 5.2.

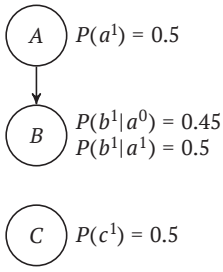


Рис. 5.1. Простая байесовская сеть для демонстрации того, как байесовская оценка помогает сбалансировать сложность модели

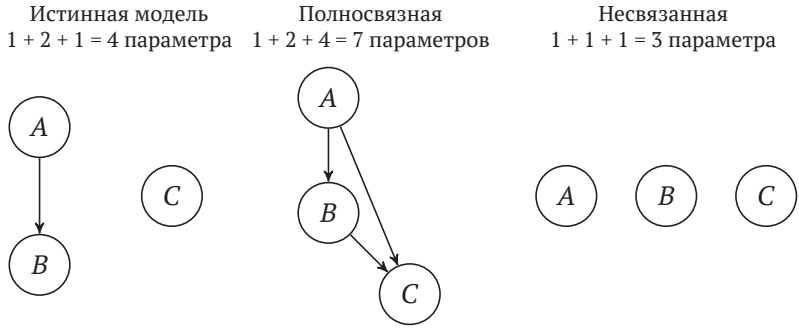


Рис. 5.2. Три структуры байесовской сети с разным уровнем сложности

На рис. 5.3 показано, как байесовские оценки полносвязных и несвязанных моделей соотносятся с истинной моделью по мере увеличения объема данных. При построении графика мы вычитали оценку истинной модели, поэтому значения выше 0 указывают на то, что текущая модель обеспечивает лучшее представление, чем истинная модель (на доступных данных). Из графика следует, что несвязанная модель работает лучше, чем истинная модель, когда имеется менее 5×10^3 выборок. Полносвязная модель никогда не работает лучше, чем истинная модель, но она начинает работать лучше, чем несвязанная модель, начиная примерно с 10^4 выборок, потому что набирается достаточно данных для точного вычисления ее семи независимых параметров.

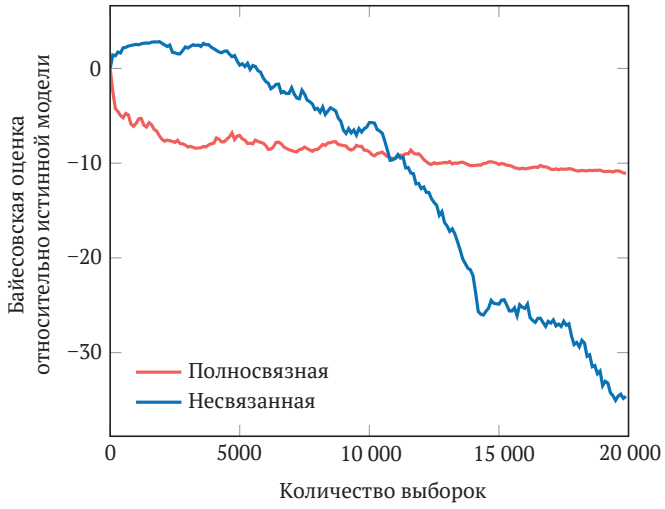


Рис. 5.3. Обучение байесовской сетевой структуры обеспечивает сбалансированную сложность модели для доступных данных. Полносвязная модель никогда не превосходит истинную модель, в то время как несвязанная модель в конечном итоге уступает по качеству, когда взято более 5×10^3 выборок. Это означает, что более простые модели могут превзойти сложные в условиях нехватки данных

5.2. Поиск ориентированного графа

При *поиске ориентированного графа* (directed graph search) мы ищем в пространстве ориентированных ациклических графов такой, при котором байесовская оценка максимальна. Пространство возможных байесовских сетевых структур растет сверхэкспоненциально[4]. При 10 узлах существует 4.2×10^{18} воз-

[4] R. W. Robinson, Counting Labeled Acyclic Digraphs, in *Ann Arbor Conference on Graph Theory*, 1973.

можных ориентированных ациклических графов. При 20 узлах их уже 2.4×10^{72}. За исключением байесовских сетей с несколькими узлами, мы не можем пройти перебором пространство возможных структур, чтобы найти сеть с наивысшей оценкой. Поэтому приходится полагаться на стратегию поиска.

К счастью, поиск – это общая проблема, и за долгие годы было изучено множество общих поисковых алгоритмов.

Одна из наиболее распространенных стратегий поиска называется K_2[5]. Поиск (алгоритм 5.2) выполняется за полиномиальное время, но не гарантирует нахождения глобально оптимальной структуры сети. Он может использовать любую функцию оценки, но на практике часто выбирают байесовскую оценку из-за ее способности обеспечить сбалансированную сложность модели для доступных данных. Поиск K_2 начинает с графа без направленных ребер, а затем перебирает переменные в соответствии с заданным порядком, жадно добавляя родителей к узлам таким образом, чтобы максимально увеличить оценку. Для K_2 характерно ограничение максимального количества родителей для любого узла, чтобы уменьшить объем вычислений. Исходный вариант алгоритма K_2 предполагает априорное равномерное распределение Дирихле с $\alpha_{ijk} = 1$ для всех i, j и k, но в принципе можно использовать любое априорное распределение.

Алгоритм 5.2. Поиск K_2 в пространстве ориентированных ациклических графов с использованием заданного порядка переменных. Этот порядок переменных задает топологический порядок в результирующем графе. Функция `fit` принимает упорядоченный список переменных `vars` и набор данных `D`. Метод начинается с пустого графа и итеративно добавляет следующего родителя, который максимально улучшает байесовскую оценку

```
struct K2Search
    ordering::Vector{Int} # variable ordering
end

function fit(method::K2Search, vars, D)
    G = SimpleDiGraph(length(vars))
    for (k,i) in enumerate(method.ordering[2:end])
        y = bayesian_score(vars, G, D)
        while true
            y_best, j_best = -Inf, 0
            for j in method.ordering[1:k]
                if !has_edge(G, j, i)
                    add_edge!(G, j, i)
                    y′ = bayesian_score(vars, G, D)
                    if y′ > y_best
                        y_best, j_best = y′, j
```

---

[5] Название происходит от того факта, что это эволюция системы Кутато. Алгоритм был представлен в G. F. Cooper, E. Herskovits, *A Bayesian Method for the Induction of Probabilistic Networks from Data*, Machine Learning, vol. 4, no. 9, pp. 309–347, 1992.

```
                end
                rem_edge!(G, j, i)
            end
        end
        if y_best > y
            y = y_best
            add_edge!(G, j_best, i)
        else
            break
        end
    end
end
return G
end
```

Общей стратегией поиска является *локальный поиск*, который иногда называют *восхождением по выпуклой поверхности*, или *поиском экстремума*. Алгоритм 5.3 иллюстрирует реализацию этого подхода. На первом шаге у нас есть начальный граф, а от него мы переходим к соседу с наивысшим результатом. Окрестность графа состоит из графов, которые находятся на расстоянии только одной базовой операции над графом. К базовым операциям относятся добавление ребра, удаление ребра и обращение направления ребра. Конечно, над текущим графом не всегда можно выполнить произвольную операцию, а операции, вводящие циклы в граф, недопустимы. Поиск продолжается до тех пор, пока среди соседей удается найти граф, оценка которого больше, чем у текущего графа.

Вероятностная версия локального поиска реализована в алгоритме 5.3. Вместо того чтобы генерировать всех соседей графа на каждой итерации, этот метод генерирует одного случайного соседа и переходит к нему, если его байесовская оценка больше, чем у текущего графа.

Алгоритм 5.3. Поиск в локальном ориентированном графе, который начинается с начального ориентированного графа `G` и перемещается к соседнему случайному графу всякий раз, когда его байесовская оценка больше. Этот процесс повторяется `k_max` итераций. Случайные соседи графа генерируются путем добавления или удаления одного ребра. Этот алгоритм можно расширить, включив в него изменение направления ребра на противоположное. Добавление ребер может привести к появлению циклов графа, и в этом случае мы присваиваем ему оценку $-\infty$

```
struct LocalDirectedGraphSearch
    G # initial graph
    k_max # number of iterations
end

function rand_graph_neighbor(G)
    n = nv(G)
```

```
    i = rand(1:n)
    j = mod1(i + rand(2:n)-1, n)
    G′ = copy(G)
    has_edge(G, i, j) ? rem_edge!(G′, i, j) : add_edge!(G′, i, j)
    return G′
end

function fit(method::LocalDirectedGraphSearch, vars, D)
    G = method.G
    y = bayesian_score(vars, G, D)
    for k in 1:method.k_max
        G′ = rand_graph_neighbor(G)
        y′ = is_cyclic(G′) ? -Inf : bayesian_score(vars, G′, D)
        if y′ > y
            y, G = y′, G′
        end
    end
    return G
end
```

Локальный поиск может застрять в *локальных оптимумах*, не позволяя найти глобально оптимальную структуру сети. Для решения проблемы глобальных оптимумов были предложены различные стратегии, в том числе следующие[6]:

- *произвольный перезапуск*. Как только локальные оптимумы будут найдены, просто перезапускают поиск в произвольной точке пространства поиска;
- *имитация отжига*. Вместо того чтобы всегда двигаться к соседу с наибольшей оценкой, поиск может посещать соседей с более низкой оценкой в соответствии с некоторой рандомизированной стратегией исследования. По мере продвижения поиска элемент случайности в исследовании пространства уменьшается в соответствии с определенным графиком. Этот подход получил свое название по аналогии с процессом отжига (снятия локальных напряжений) металла в металлургии;
- *генетические алгоритмы*. Процедура начинается с начальной случайной совокупности точек в пространстве поиска, представленных в виде двоичных строк. Каждый бит в строке указывает на наличие или отсутствие стрелки между двумя узлами. Таким образом, манипуляции со строками позволяют осуществлять поиск в пространстве ориентированных графов. Особи в популяции размножаются со скоростью, пропорциональной их оценке. Строки отдельных особей, выбранных для размножения, рекомбинируются случайным образом посредством *генетического скрещивания* (кроссовера), которое включает выбор точки скрещивания на двух случайно выбранных особях, а затем замену строк после этой точки. В популяцию также вводят мутации путем случайной инверсии битов в строках.

[6] Область оптимизации весьма обширна, и для достижения локальных оптимумов было разработано множество методов. Хороший обзор представлен в следующем учебнике: M. J. Kochenderfer, T. A. Wheeler, *Algorithms for Optimization*. MIT Press, 2019.

Процесс эволюции продолжается до тех пор, пока не будет найдена удовлетворительная точка в пространстве поиска;

- *меметические алгоритмы*. Этот метод, иногда называемый *генетическим локальным поиском*, представляет собой просто комбинацию генетических алгоритмов и локального поиска. После генетической рекомбинации к особям применяется локальный поиск;
- *поиск с запретами*. Предыдущие методы могут быть дополнены списком запретов, содержащим недавно посещенные точки в пространстве поиска. Алгоритм поиска избегает соседей, указанных в списке запретов.

Некоторые стратегии поиска могут работать лучше, чем другие, на определенных наборах данных, но в целом поиск глобальных оптимумов остается NP-трудным. Однако для многих практических применений не требуется глобально оптимальная сетевая структура и можно ограничиться локально оптимальной.

5.3. Марковские классы эквивалентности

Как было сказано ранее, структура байесовской сети кодирует набор предположений об условной независимости. При обучении структуры байесовской сети необходимо учитывать важный факт: два разных графа могут кодировать одни и те же предположения о независимости. В качестве простого примера: сеть с двумя переменными $A \rightarrow B$ имеет те же предположения о независимости, что и $A \leftarrow B$. Мы не можем определить направление ребра между узлами A и B только лишь на основе данных.

Если две сети кодируют одни и те же предположения об условной независимости, мы говорим, что они *эквивалентны по Маркову* (Markov equivalent). Можно доказать, что два графа эквивалентны по Маркову тогда и только тогда, когда они имеют 1) одни и те же ребра независимо от направления и 2) тот же набор имморальных *v*-структур. *Имморальная v-структура* (immoral v-structure) – это *v*-образная структура $X \rightarrow Y \leftarrow Z$, где X и Z не связаны напрямую, как показано на рис. 5.4. *Марковский класс эквивалентности* (Markov equivalence class) – это множество, содержащее все направленные ациклические графы, которые эквивалентны друг другу. Метод проверки эквивалентности по Маркову реализован в алгоритме 5.4.

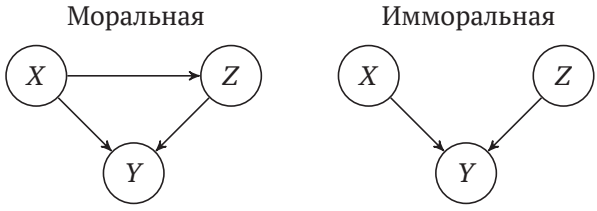


Рис. 5.4. Моральные и имморальные *v*-структуры

Как правило, двум структурам, принадлежащим к одному и тому же марковскому классу эквивалентности, могут быть присвоены разные оценки. Однако если байесовская оценка используется с априорным распределением Дирихле, таким что $k = \sum_j \sum_k \alpha_{ijk}$ является константой для всех i, то двум эквивалентным по Маркову структурам присваивается одна и та же оценка[7]. Такие априорные структуры называются *BDe*, а особым случаем является априорная структура *BDeu*[8], в которой $\alpha_{ijk} = k/(q_i r_i)$. Хотя обычно используемое равномерное априорное распределение $\alpha_{ijk} = 1$ не всегда приводит к тому, что идентичные оценки присваиваются структурам, состоящим в одном и том же классе эквивалентности, они часто довольно близки. Функция оценки, которая присваивает одинаковую оценку всем структурам одного класса, называется *эквивалентной оценкой*.

5.4. Поиск частично ориентированного графа

Класс марковской эквивалентности может быть представлен как *частично ориентированный граф* (partially directed graph), который иногда называют *существенным графом* (essential graph), или *шаблоном ориентированного ациклического графа*. Частично ориентированный граф может содержать как направленные, так и ненаправленные ребра. Пример частично ориентированного графа, кодирующего марковский класс эквивалентности, показан на рис. 5.5. Ориентированный ациклический граф G является членом марковского класса эквивалентности, закодированного частично ориентированным графом G', тогда и только тогда, когда G имеет те же ребра, что и G', независимо от направления, и имеет те же имморальные v-структуры, что и G'.

Алгоритм 5.4. Метод определения марковской эквивалентности ориентированных ациклических графов `G` и `H`. Функция `subsets` из `IterTools.jl` возвращает все подмножества данного набора и указанного размера

```
function are_markov_equivalent(G, H)
    if nv(G) != nv(H) || ne(G) != ne(H) ||
        !all(has_edge(H, e) || has_edge(H, reverse(e))
                                    for e in edges(G))
        return false
    end
    for (I, J) in [(G,H), (H,G)]
        for c in 1:nv(I)
            parents = inneighbors(I, c)
```

---

[7] Это было показано в D. Heckerman, D. Geiger, D. M. Chickering, *Learning Bayesian Networks: The Combination of Knowledge and Statistical Data*, Machine Learning, vol. 20, no. 3, pp. 197–243, 1995.

[8] W. L. Buntine, *Theory Refinement on Bayesian Networks*, in Conference on Uncertainty in Artificial Intelligence (UAI), 1991.

```
            for (a, b) in subsets(parents, 2)
                if !has_edge(I, a, b) && !has_edge(I, b, a) &&
                   !(has_edge(J, a, c) && has_edge(J, b, c))
                    return false
                end
            end
        end
    end

    return true
end
```

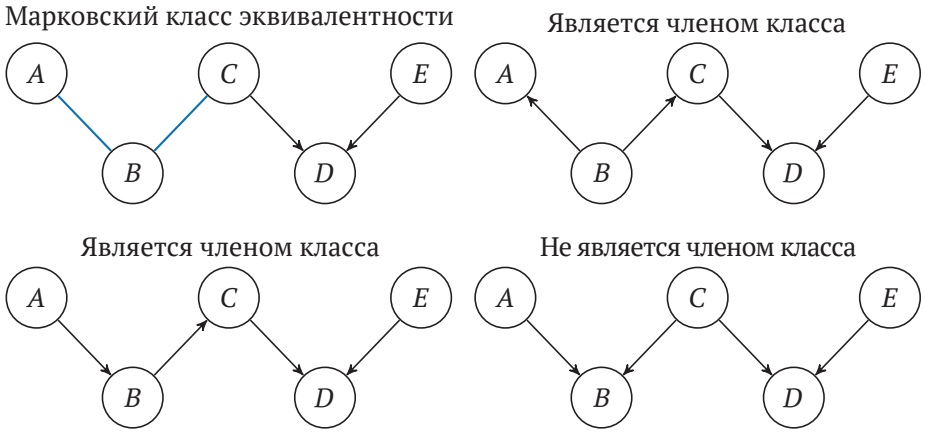


Рис. 5.5. Марковский класс эквивалентности и примеры структур, которые являются и не являются членами этого класса. Одна из структур не принадлежит к марковскому классу эквивалентности, поскольку вводит имморальную *v*-структуру $A \rightarrow B \leftarrow C$, которая отсутствует в частично ориентированном графе

Вместо поиска в пространстве ориентированных ациклических графов мы можем искать в пространстве марковских классов эквивалентности, представленных частично ориентированными графами[9]. Хотя такое пространство, конечно, меньше пространства ориентированных ациклических графов, оно не существенно меньше; отношение направленных ациклических графов к классам эквивалентности довольно быстро приближается к 3.7[10]. Проблема восхождения по выпуклой поверхности в пространстве направленных ациклических графов заключается в том, что окрестность может состоять из других графов, принадлежащих к тому же классу эквивалентности с той же оценкой, что может привести к застреванию поиска в локальном оптимуме. Поиск в пространстве

[9] Подробности поиска в этом пространстве предоставлены в D. M. Chickering, *Learning Equivalence Classes of Bayesian-Network Structures*, Journal of Machine Learning Research, vol. 2, pp. 445–498, 2002.

[10] S. B. Gillispie, M. D. Perlman, *The Size Distribution for Markov Equivalence Classes of Acyclic Digraph Models*, Artificial Intelligence , vol. 141, no. 1–2, pp. 137–155, 2002.

классов эквивалентности позволяет нам переходить к различным ориентированным ациклическим графам вне текущего класса эквивалентности.

На практике можно использовать любую из общих стратегий поиска, представленных в разделе 5.2. Если используется какая-либо форма локального поиска, нам нужно определить локальные операции с графами, которые определяют окрестности графа. В качестве примера можно привести следующие операции:

- если ребра между X и Y не существует, добавить либо $X - Y$, либо $X \rightarrow Y$;
- если существует ребро $X - Y$ или $X \rightarrow Y$, то удалить ребро между X и Y;
- если существует ребро $X \rightarrow Y$, то изменить направление ребра, чтобы получить $X \leftarrow Y$;
- если существуют ребра $X - Y - Z$, то добавить $X \rightarrow Y \leftarrow Z$.

Чтобы оценить частично ориентированный граф, мы генерируем член его марковского класса эквивалентности и вычисляем оценку.

5.5. Заключение

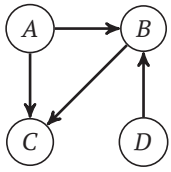
- Для подгонки байесовской сети к данным необходимо выбрать структуру байесовской сети, которая диктует условные зависимости между переменными.
- Байесовские подходы к структурному обучению максимизируют байесовскую оценку, которая связана с вероятностью графовой структуры при заданном наборе данных.
- Байесовская оценка применима как к простым структурам для небольших наборов данных, так и к более сложным структурам для больших наборов данных.
- Количество возможных структур растет суперэкспоненциально с ростом количества переменных, и поиск структуры, для которой байесовская оценка максимальна, является NP-сложной задачей.
- Алгоритмы поиска в пространстве ориентированных графов, такие как K_2 и локальный поиск, могут быть эффективными, но не гарантируют оптимальности.
- Такие методы, как поиск частично ориентированного графа, обходят пространство марковских классов эквивалентности, что может быть более эффективно, чем поиск в большем пространстве ориентированных ациклических графов.

5.6. Упражнения

Упражнение 5.1. Сколько соседей имеют ориентированный ациклический граф без ребер, у которого m вершин?

Решение. Из трех основных операций над графом мы можем только добавлять ребра. Мы можем добавить любое ребро к безреберному ориентированному ациклическому графу, и он останется ациклическим. Граф имеет $m(m-1) = m^2 - m$ возможных пар узлов и, следовательно, столько же соседей.

Упражнение 5.2. Сколько сетей находится по соседству со следующей байесовской сетью?

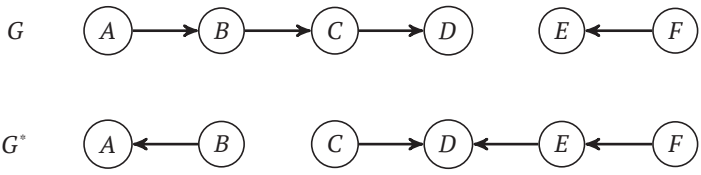


Решение. Мы можем выполнить следующие операции над графом:

- добавить $A \to D, D \to A, D \to C$;
- удалить $A \to B, A \to C, B \to C, D \to B$;
- обратить $A \to B, B \to C, D \to B$.

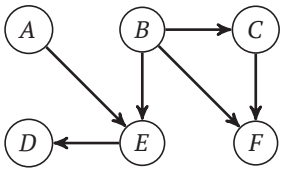
Следовательно, у этой байесовской сети есть 10 соседей.

Упражнение 5.3. Предположим, мы начинаем локальный поиск с байесовской сети G. Какое наименьшее количество итераций локального поиска может быть выполнено для сходимости к оптимальной байесовской сети G^*?

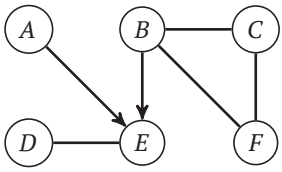


Решение. На каждой итерации локальный поиск может перемещаться из исходной сети в соседнюю, что соответствует не более чем одной операции над ребром исходной сети. Поскольку между ребрами сетей G и G^* есть три различия, для выполнения локального поиска из G потребуется как минимум три итерации, чтобы достичь G^*. Одна из возможных минимальных последовательностей итераций локального поиска выглядит так: обращение $A \to B$, удаление $B \to C$ и добавление $E \to D$. Мы предполагаем, что графы, сформированные с помощью этих операций с ребрами, дали самые высокие байесовские оценки среди всех графов в рассматриваемой окрестности.

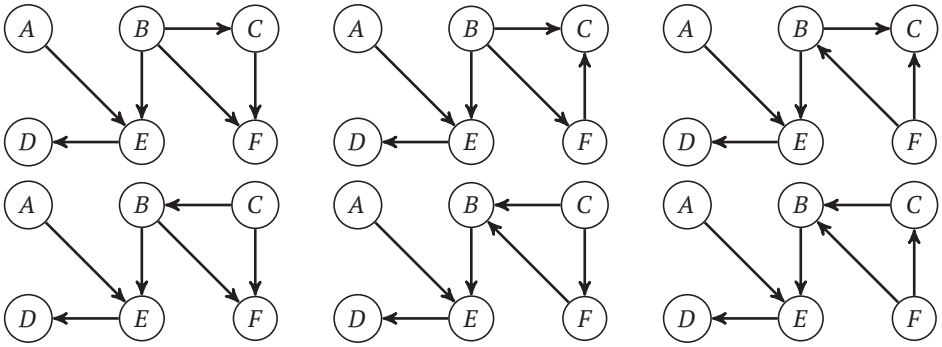
Упражнение 5.4. Постройте частично ориентированный ациклический граф, представляющий марковский класс эквивалентности следующей байесовской сети. Сколько графов входит в этот класс?



Решение. Марковский класс эквивалентности может быть представлен следующим частично ориентированным ациклическим графом:

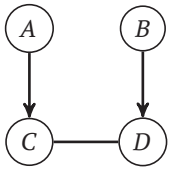


В этот класс эквивалентности входят шесть следующих сетей:



Упражнение 5.5. Приведите пример частично ориентированного ациклического графа с четырьмя вершинами, не определяющего непустой марковский класс эквивалентности.

Решение. Рассмотрим следующий частично ориентированный ациклический граф:



Мы не можем заменить ненаправленное ребро на направленное, потому что это приведет к появлению новой *v*-структуры.

6 Простые решения

В этой главе вводится понятие *простых решений* (simple decision), когда мы принимаем единственное решение в условиях неопределенности[1]. Мы изучим проблему принятия решений с точки зрения *теории полезности* (utility theory), которая подразумевает моделирование предпочтений агента как *функции действительных значений* (real-valued function), в отличие от неопределенных исходов[2]. Эта глава начинается с обсуждения того, как небольшой набор ограничений на *рациональные предпочтения* (rational preference) может привести к получению функции полезности, поскольку ее можно вывести из последовательности запросов о предпочтениях. Затем мы вводим принцип *максимальной ожидаемой полезности* как определение рациональности – центральное понятие *теории принятия решений* (decision theory), которое будет служить ключевым критерием принятия решений в этой книге[3]. Далее будет показано, как задачи принятия решений могут быть представлены в виде сетей решений, и представлен алгоритм достижения оптимального решения. В этой главе также вводится понятие *полезности информации* (value of information), которое определяет полезность, полученную за счет наблюдения за дополнительными переменными. Глава завершается кратким обсуждением того, почему принятие решений человеком не всегда согласуется с принципом максимальной ожидаемой полезности.

6.1. Ограничения рациональных предпочтений

Мы начали обсуждение неопределенности в главе 2 с указания на необходимость сравнивать степень *уверенности* в различных утверждениях. В этой главе мы будем говорить об умении сравнивать степень *желательности* двух

---

[1] Простые решения выглядят намного легче по сравнению с последовательными задачами, которым посвящена оставшаяся часть книги. Однако простые решения не обязательно легко достижимы.

[2] Шумейкер дает обзор развития теории полезности в P. J. H. Schoemaker, *The Expected Utility Model: Its Variants, Purposes, Evidence and Limitations*, Journal of Economic Literature, vol. 20, no. 2, pp. 529–563, 1982. Fishburn surveys the field. См. также P. C. Fishburn, *Utility Theory*, Management Science, vol. 14, no. 5, pp. 335–378, 1968.

[3] Обзор области теории принятия решений представлен в M. Peterson, *An Introduction to Decision Theory*. Cambridge University Press, 2009.

разных исходов. Мы заявляем о наших предпочтениях, используя следующие операторы:

- $A \succ B$, если мы предпочитаем A, а не B;
- $A \sim B$, если нет предпочтений между A и B;
- A $\succcurlyeq$ B, если мы предпочитаем A, а не B, или разницы нет.

Как убеждения, так и предпочтения могут быть субъективными.

Помимо сравнения событий, наши операторы предпочтений можно использовать для сравнения предпочтений относительно неопределенных исходов. *Лотерея* (lottery) – это набор вероятностей, соотнесенных с набором исходов. Например, если $S_{1:n}$ – набор исходов, а $p_{1:n}$ – связанные с ними вероятности, то лотерея, охватывающая эти исходы и вероятности, записывается как

$$[S_1 : p_1; \ldots; S_n : p_n]. \tag{6.1}$$

Существование действительной меры полезности вытекает из набора предположений о предпочтениях[4]. Исходя из функции полезности, можно дать определение того, что значит «принимать рациональные решения в условиях неопределенности». Точно так же, как ранее мы наложили ряд ограничений на убеждения, мы наложим некоторые ограничения на предпочтения[5]:

- *полнота*. Выполняется ровно одно из следующих утверждений: $A \succ B$, $B \succ A$ или $A \sim B$;
- *транзитивность*. Если $A \succcurlyeq B$ и $B \succcurlyeq C$, то $A \succcurlyeq C$;
- *непрерывность*. Если $A \succcurlyeq C \succcurlyeq B$, то существует вероятность p такая, что $[A : p; B : 1 - p] \sim C$;
- *независимость*. Если $A \succ B$, то для любого C и вероятности p справедливо $[A : p; C : 1 - p] \succcurlyeq [B : p; C : 1 - p]$.

Это ограничения, налагаемые на рациональные предпочтения. Они ничего не говорят о предпочтениях реальных людей; на самом деле есть веские доказательства того, что люди не всегда рациональны (этот вопрос обсуждается далее в разделе 6.7). Наша цель в этой книге – определить рациональное принятие решений с вычислительной точки зрения, чтобы мы могли создавать полезные системы. Возможное расширение этой теории до понимания процесса принятия решений человеком представляет лишь второстепенный интерес.

---

[4] Теория ожидаемой полезности была введена швейцарским математиком и физиком Даниэлем Бернулли (1700–1782) в 1738 г.: D. Bernoulli, *Exposition of a New Theory on the Measurement of Risk*, Econometrica, vol. 22, no. 1, pp. 23–36, 1954.

[5] Эти ограничения иногда именуют *аксиомами фон Неймана–Моргенштерна*, названными в честь венгерско-американского математика и физика Джона фон Неймана (1903–1957) и австрийско-американского экономиста Оскара Моргенштерна (1902–1977), которые сформулировали разные варианты этих аксиом. См. J. von Neumann, O. Morgenstern, *Theory of Games and Economic Behavior*. Princeton University Press, 1944. Критика этих аксиом представлена в P. Anand, *Are the Preference Axioms Really Rational?* Theory and Decision, vol. 23, no. 2, pp. 189–214, 1987.

6.2. Функции полезности

Точно так же, как ограничения на сравнение правдоподобия различных утверждений приводят к понятию действительной меры вероятности, ограничения на рациональные предпочтения приводят к понятию действительной меры *полезности*. Из наших ограничений на рациональные предпочтения следует, что существует действительная функция полезности U такая, что

- $U(A) > U(B)$ тогда и только тогда, когда $A \succ B$;
- $U(A) = U(B)$ тогда и только тогда, когда $A \sim B$.

Функция полезности уникальна с точностью до *положительного аффинного преобразования*. Другими словами, для любых констант $m > 0$ и b равенство $U'(S) = mU(S) + b$ справедливо тогда и только тогда, когда предпочтения, основанные на U', такие же, как и U. Полезности подобны температурам: вы можете успешно сравнивать температуры, используя шкалы Кельвина, Цельсия или Фаренгейта, потому что все они являются аффинными преобразованиями друг друга.

Из ограничений на рациональные предпочтения следует, что полезность лотереи определяется как

$$U([S_1 : p_1; \ldots; S_n : p_n]) = \sum_{i=1}^{n} p_i U(S_i). \tag{6.2}$$

В примере 6.1 это уравнение применяется для вычисления полезности исходов, связанных с системой предотвращения столкновений.

Пример 6.1. Лотерея на основе исходов системы предотвращения столкновений

Предположим, что мы строим систему предотвращения столкновений. Исход столкновения с воздушным судном определяется тем, выдает ли система предупреждение (A) и происходит ли столкновение (C). Поскольку A и C принимают бинарные значения, возможны четыре исхода. Пока наши предпочтения рациональны, мы можем записать нашу функцию полезности в пространстве возможных лотерей с помощью четырех параметров: $U(a^0, c^0)$, $U(a^1, c^0)$, $U(a^0, c^1)$ и $U(a^1, c^1)$. Например,

$U([a^0, c^0 : 0.5; a^1, c^0 : 0.3; a^0, c^1 : 0.1; a^1, c^1 : 0.1])$

равно

$0.5U(a^0, c^0) + 0.3U(a^1, c^0) + 0.1U(a^0, c^1) + 0.1U(a^1, c^1)$.

Если функция полезности ограничена, то мы можем определить *нормализованную функцию полезности*, где наилучшему возможному результату присваивается полезность 1, а наихудшему возможному результату присваивает-

ся полезность 0. Полезность каждого из других результатов масштабируется и транслируется по мере необходимости.

6.3. Выявление полезности

При построении системы принятия решений или поддержки принятия решений часто имеет смысл вывести функцию полезности для человека или группы людей. Такой подход называется *выявлением полезности* (utility elicitation), или *выявлением предпочтений* (preference elicitation)[6]. Один из способов сделать это – принять полезность наихудшего результата $\underline{S}$ равной 0, а наилучшего результата $\bar{S}$ – равной 1. Пока полезности исходов ограничены, мы можем транслировать и масштабировать их, не изменяя наши предпочтения. Если нужно определить полезность исхода S, то мы определяем вероятность p так, что $S \sim [\bar{S}{:}p; \underline{S}{:}1 - p]$. Отсюда следует, что $U(S) = p$. В примере 6.2 этот процесс применяется для определения функции полезности, связанной с проблемой предотвращения столкновений.

Пример 6.2. Применение процесса выявления полезности в системе предотвращения столкновений

В нашем примере предотвращения столкновений наилучшее возможное событие – это отсутствие предупреждения и отсутствие столкновения, поэтому мы принимаем $U(a^0, c^0) = 1$. Наихудшее возможное событие – это наличие предупреждения и столкновения, поэтому мы принимаем $U(a^1, c^1) = 0$. Определим лотерею $L(p)$ как $[a^0, c^0{:}p; a^1, c^1{:}1 - p]$. Чтобы определить $U(a^1, c^0)$, нужно найти такую p, что $(a^1, c^0) \sim L(p)$. Аналогично, чтобы определить $U(a^0, c^1)$, находим такую p, что $(a^0, c^1) \sim L(p)$.

Может возникнуть соблазн использовать для вывода функций полезности критерии денежной полезности. Например, если мы создаем систему поддержки принятия решений для борьбы с лесными пожарами, инстинктивно хочется определить функцию полезности с точки зрения убытков, связанных с повреждением имущества, и затрат на развертывание ресурсов пожаротушения. Однако в экономике хорошо известно, что полезность богатства в общем случае нелинейна[7]. Если бы существовала линейная зависимость между полезностью и богатством, то решения принимались бы исключительно с точки зрения максимизации ожидаемой денежной полезности. Попытка максимизировать ожидаемую денежную полезность приведет к отказу от института

[6] Различные методы извлечения полезности рассмотрены в P. H. Farquhar, *Utility Assessment Methods*, Management Science, vol. 30, no. 11, pp. 1283–1300, 1984.

[7] H. Markowitz, *The Utility of Wealth*, Journal of Political Economy, vol. 60, no. 2, pp. 151–158, 1952.

страхования, потому что ожидаемая денежная полезность страховых полисов обычно отрицательна.

Вместо того чтобы пытаться максимизировать ожидаемое богатство как таковое, обычно стараются максимизировать ожидаемую полезность богатства. Конечно, у разных людей разное восприятие полезности на этот счет. На рис. 6.1 показан пример функции полезности. Для небольшого богатства кривая почти линейна, поэтому 100 долларов примерно в два раза лучше, чем 50 долларов. Однако с ростом богатства наклон кривой заметно уменьшается; в конце концов, разница в 1000 долларов для миллиардера значит намного меньше, чем для обычного человека. Эффект сглаживания кривой полезности иногда называют *убывающей предельной полезностью*.

При обсуждении денежных функций полезности часто используют три упомянутых ниже термина. Чтобы проиллюстрировать их, предположим, что исход A представляет получение 50 долларов, а B представляет 50%-ный шанс выиграть 100 долларов.

- *Нейтралитет к риску*. Функция полезности линейна. Нет никакого предпочтения между 50 долларами и шансом 50 % выиграть 100 долларов ($A \sim B$).
- *Стремление к риску*. Функция полезности выпуклая. Предпочтение отдается шансу 50 % выиграть 100 долларов ($A \prec B$).
- *Избегание риска*. Функция полезности вогнутая. Предпочтение отдается гарантированным 50 долларам ($A \succ B$).

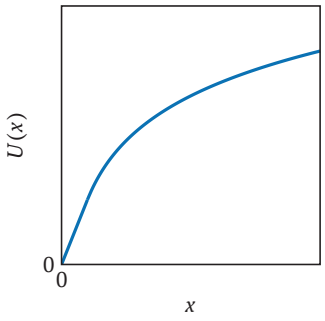


Рис. 6.1. Полезность богатства x часто моделируется как линейная для небольших значений, а затем вогнутая для больших значений, демонстрируя неприятие риска

Существует несколько общих функциональных форм для моделирования неприятия риска в случае скалярных величин[8], таких как богатство или наличие больничных коек. Одна из них – *квадратичная полезность*:

$$U(x) = \lambda x - x^2, \tag{6.3}$$

где параметр $\lambda > 0$ определяет степень неприятия риска. Поскольку обычно нам нужно, чтобы эта функция полезности монотонно возрастала при моделирова-

[8] Эти функциональные формы хорошо изучены в области экономики и финансов: J. E. Ingersoll, *Theory of Financial Decision Making*. Rowman and Littlefield Publishers, 1987.

нии полезности таких величин, как богатство, принято ограничивать эту функцию значением $x = \lambda/2$. После этого полезность начинает уменьшаться. Другой простой формой является *экспоненциальная полезность*:

$$U(x) = 1 - e^{-\lambda x}, \tag{6.4}$$

где $\lambda > 0$. Хотя это удобная математическая форма, обычно она не рассматривается как правдоподобная модель полезности богатства. Альтернативой является *энергетическая полезность* (power utility):

$$U(x) = \frac{x^{1-\lambda} - 1}{1-\lambda}, \tag{6.5}$$

где $\lambda \geqslant 0$ и $\lambda \neq 1$. *Логарифмическая полезность*:

$$U(x) = \log x, \tag{6.6}$$

где $x > 0$ можно рассматривать как частный случай энергетической полезности, где $\lambda \to 1$. На рис. 6.2 показан график функции энергетической полезности вместе с частным случаем логарифмической полезности.

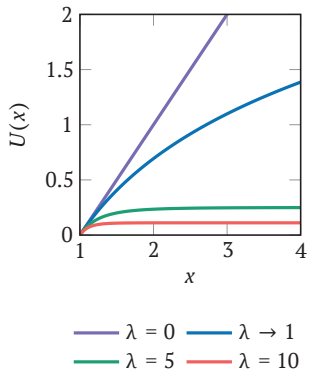


Рис. 6.2. Функции энергетической полезности

6.4. Принцип максимальной ожидаемой полезности

Нас интересует проблема принятия рациональных решений при несовершенном знании состояния мира. Предположим, что у нас есть вероятностная модель $P(s'|o, a)$, которая представляет вероятность того, что состояние мира будет s', при условии что мы наблюдаем o и предпринимаем действие a. У нас есть функция полезности $U(s')$, которая кодирует наши предпочтения в пространстве исходов. Ожидаемая полезность (expected utility, EU) действия a при заданном наблюдении o определяется выражением

$$EU(a|o) = \sum_{s'} P(s'|a,o)U(s'). \tag{6.7}$$

Принцип *максимальной ожидаемой полезности* гласит, что *рациональный агент* должен выбрать действие, максимизирующее ожидаемую полезность:

$$a^* = \arg\max_a EU(a|o). \tag{6.8}$$

Поскольку нас интересует построение рациональных агентов, уравнение (6.8) играет центральную роль в этой книге[9]. Пример 6.3 иллюстрирует применение этого принципа к задаче выбора простого решения.

Пример 6.3. Применение принципа максимальной ожидаемой полезности к простому решению взять с собой зонт

Предположим, что вы пытаетесь решить, брать ли с собой в поездку зонт, учитывая прогноз погоды в пункте назначения. Вы наблюдаете прогноз o, который может предсказывать либо дождь, либо солнце. Выбор вашего действия a состоит в том, чтобы либо взять зонт в дорогу, либо оставить его дома. Результирующее состояние s' является комбинацией вашего решения и фактической погоды в пункте назначения. Ваша вероятностная модель выглядит следующим образом:

Наблюдение o	Действие a	Состояние s'	$P(s'\|a, o)$
прогноз: дождь	взять зонт	дождь, есть зонт	0.9
прогноз: дождь	оставить зонт	дождь, нет зонта	0.9
прогноз: дождь	взять зонт	ясно, есть зонт	0.1
прогноз: дождь	оставить зонт	ясно, нет зонта	0.1
прогноз: ясно	взять зонт	дождь, есть зонт	0.2
прогноз: ясно	оставить зонт	дождь, нет зонта	0.2
прогноз: ясно	взять зонт	ясно, есть зонт	0.8
прогноз: ясно	оставить зонт	ясно, нет зонта	0.8

Как показано в таблице, вы предполагаете, что прогноз несовершенен; предсказание дождя сбывается в 90 % случаев, а предсказание солнца – в 80 % случаев. Кроме того, вы предполагаете, что наличие зонта не влияет на погоду, хотя некоторые могут усомниться в этом предположении. Функция полезности выглядит следующим образом:

[9] Важность принципа максимальной ожидаемой полезности для области искусственного интеллекта обсуждается в S. Russell, P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Pearson, 2021.

Состояние s'	$U(s')$
дождь, есть зонт	−0.1
дождь, нет зонта	−1
ясно, есть зонт	0.9
ясно, нет зонта	1

Вы можете вычислить ожидаемую полезность действия «взять зонт», если спрогнозирован дождь, используя уравнение (6.7):

$EU(\text{взять зонт} \mid \text{спрогнозирован дождь}) = 0.9 \times (-0.1) + 0.1 \times 0.9 = 0.$

Точно так же вы можете вычислить ожидаемую полезность действия «оставить зонт», если спрогнозирован дождь, используя уравнение (6.7):

$EU(\text{оставить зонт} \mid \text{спрогнозирован дождь}) = 0.9 \times (-1) + 0.1 \times 1 = -0.8.$

Следовательно, вам нужно взять с собой зонт.

6.5. Сети принятия решений

Сеть принятия решений (decision network), иногда называемая *диаграммой влияния* (influence diagram), представляет собой обобщение байесовской сети, включающее в себя узлы действия и полезности, и компактно изображает модели вероятности и полезности, определяющие проблему принятия решения[10]. Пространства состояния, действия и наблюдения, упомянутые в предыдущем разделе, могут быть факторизованы, а структура сети принятия решений фиксирует отношения между различными компонентами.

Сети принятия решений состоят из трех типов узлов:

- *узел вероятности* – соответствует случайной величине и обозначается кружком;
- *узел принятия решения* (*узел действия*) соответствует переменной решения и обозначается квадратом;
- *узел полезности* соответствует переменной полезности, обозначается ромбом и не может иметь потомков.

Существуют также три вида направленных ребер:

- *условное ребро* заканчивается узлом вероятности и указывает, что неопределенность в этом узле вероятности обусловлена значениями всех его родителей;
- *информационное ребро* заканчивается узлом принятия решения и указывает, что решение, связанное с этим узлом, принимается с учетом значе-

[10] Подробное обсуждение сетей принятия решений можно найти в F. V. Jensen, T. D. Nielsen, *Bayesian Networks and Decision Graphs*, 2nd ed. Springer, 2007.

ний его родителей (эти ребра часто изображены пунктирными линиями и иногда для простоты не показаны на диаграммах);

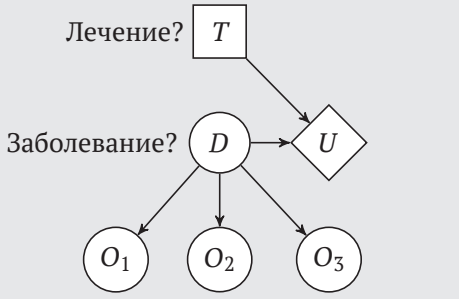
- *функциональное ребро* заканчивается узлом полезности и указывает, что этот узел определяется исходами своих родителей.

Подобно байесовским сетям, сети принятия решений не могут иметь циклов. Полезность, связанная с действием, равна сумме значений во всех узлах полезности. В примере 6.4 показано, как сеть принятия решений моделирует задачу лечения заболевания на основе результатов диагностических тестов.

Пример 6.4. Пример сети принятия решений, моделирующей решение о необходимости лечения заболевания с учетом информации, полученной в результате диагностических тестов

У нас есть набор результатов диагностических тестов, которые могут указывать на наличие определенного заболевания. Исходя из наших знаний о тестах, нам нужно решить, применять ли лечение. Полезность является функцией от того, применяется ли лечение и действительно ли присутствует болезнь. Условные ребра соединяют D с наблюдениями O_1, O_2 и O_3. Информационные ребра не показаны на иллюстрации явно, но они соединяют наблюдения с T. Функциональные ребра соединяют T и D с U.

T	D	$U(T, D)$
0	0	0
0	1	−10
1	0	−1
1	1	−1



Прямой подход к задаче (алгоритм 6.1) требует перебора всех возможных экземпляров решения, чтобы найти такое, которое максимизирует ожидаемую полезность. Для каждого экземпляра мы оцениваем связанную с ним ожидаемую полезность. Для этого мы начинаем с создания экземпляров узлов принятия решения и наблюдаемых узлов вероятности, а затем применяем любой алгоритм вывода для вычисления апостериорного распределения, исходя из входных данных функции полезности. Ожидаемая полезность представляет собой сумму значений в узлах полезности. В примере 6.5 показано, как этот процесс можно применить к нашему текущему примеру.

Алгоритм 6.1. Простой подход к задаче в виде сети принятия решений. Сеть принятия решений – это байесовская сеть с переменными случайности, решения и полезности. Переменные полезности рассматриваются как детерминированные. Поскольку переменные в нашей байесовской сети принимают значения $1{:}r_i$, переменные полезности сопоставляются с реальными значениями с помощью поля `utilities`. Например, если у нас есть служебная переменная `:u1`, то i-й полезностью, связанной с этой переменной, является `utilities[:u1][i]`. Функция `solve` принимает в качестве входных данных структуру `SimpleProblem`, `evidence` и метод вывода, а возвращает наилучшие значения переменных решения и связанную с ним ожидаемую полезность

```
struct SimpleProblem
    bn::BayesianNetwork
    chance_vars::Vector{Variable}
    decision_vars::Vector{Variable}
    utility_vars::Vector{Variable}
    utilities::Dict{Symbol, Vector{Float64}}
end

function solve(𝒫::SimpleProblem, evidence, M)
    query = [var.name for var in 𝒫.utility_vars]
    U(a) = sum(𝒫.utilities[uname][a[uname]] for uname in query)
    best = (a=nothing, u=-Inf)
    for assignment in assignments(𝒫.decision_vars)
        evidence = merge(evidence, assignment)
       φ = infer(M, 𝒫.bn, query, evidence)
        u = sum(p*U(a) for (a, p) in φ.table)
        if u > best.u
            best = (a=assignment, u=u)
        end
    end
    return best
end
```

Было разработано множество методов, делающих оценку сетей принятия решений более эффективной[11]. Один из таких методов предусматривает удаление узлов решения и вероятности из сетей принятия решений, если они не имеют потомков, образованных условными, информационными или функциональными ребрами. В примере 6.5 мы можем удалить O_2 и O_3, потому что у них нет потомков. Мы не можем удалить O_1, поскольку существует информационное ребро от O_1 до T (хотя оно не изображено явно).

[11] R. D. Shachter, *Evaluating Influence Diagrams*, Operations Research, vol. 34, no. 6, pp. 871–882, 1986. R. D. Shachter, *Probabilistic Inference and Influence Diagrams*, Operations Research, vol. 36, no. 4, pp. 589–604,1988.

Пример 6.5. Сеть принятия решений в задаче диагностики заболеваний

Воспользуемся уравнением (6.7) для вычисления ожидаемой полезности лечения болезни в соответствии с сетью принятия решений из примера 6.4. Предположим, что у нас есть результат только первого диагностического теста и он оказался положительным. Если бы мы хотели сделать знание первого диагностического теста явным на диаграмме, то провели бы информационное ребро от O_1 до T:

$$EU(t^1|o_1^1) = \sum_{o_3}\sum_{o_2}\sum_{d} P(d, o_2, o_3|t^1, o_1^1)U(t^1, d, o_1^1, o_2, o_3).$$

Воспользуемся цепным правилом для байесовских сетей и определением условной вероятности для вычисления $P(d, o_2, o_3|t^1, o_1^1)$. Поскольку узел полезности зависит только от того, присутствует ли болезнь и лечим ли мы ее, мы можем упростить $U(t^1, d, o_1^1, o_2, o_3)$ до $U(t^1, d)$. Следовательно,

$$EU(t^1|o_1^1) = \sum_{d} P(d|t^1, o_1^1)U(t^1, d).$$

Любой из точных или приближенных методов вывода, представленных в предыдущей главе, можно использовать для оценки $P(d|t^1, o_1^1)$. Чтобы решить, применять ли лечение, мы вычисляем $P(d|t^1, o_1^1)$ и $EU(t^0|o_1^1)$ и принимаем решение, обеспечивающее наибольшую ожидаемую полезность.

6.6. Полезность информации

Мы принимаем решения на основе наблюдений. Во многих случаях возникает естественное желание количественно определить полезность информации, то есть узнать, в какой мере наблюдение за дополнительными переменными повысит полезность[12]. Например, в случае лечения заболеваний в примере 6.5 мы предположили, что наблюдали только o_1^1. Зная положительный результат лишь одного диагностического теста, мы можем отказаться от лечения. Тем не менее было бы полезно провести дополнительные диагностические тесты, чтобы снизить риск отказа от лечения заболевания, которое действительно присутствует.

При вычислении полезности информации мы будем использовать $EU^*(o)$ для обозначения ожидаемой полезности оптимального действия при задан-

[12] R. A. Howard, *Information Value Theory*, IEEE Transactions on Systems Science and Cybernetics, vol. 2, no. 1, pp. 22–26, 1966. Применение сетей принятия решений рассмотрено в S. L. Dittmer, F. V. Jensen, *Myopic Value of In-formation in Influence Diagrams*, Conference on Uncertainty in Artificial Intelligence (UAI), 1997. R. D. Shachter, *Efficient Value of Information Computation*, Conference on Uncertainty in Artificial Intelligence (UAI), 1999.

ном наблюдении o. Полезность информации о переменной O' при заданном o равна

$$VOI(O'|o) = \left(\sum_{o'} P(o'|o) EU^*(o, o') \right) - EU^*(o). \tag{6.9}$$

Другими словами, полезность информации о переменной – это увеличение ожидаемой полезности, если эта переменная наблюдается. Реализация данного подхода представлена в алгоритме 6.2.

Алгоритм 6.2. Метод оценки сети принятия решений, который использует простую задачу $\mathcal{P}$, список переменных запроса `query`, словарь, содержащий наблюдаемые вероятностные переменные и их значения `evidence`, а также стратегию вывода `M`. Метод возвращает решение, которое максимизирует ожидаемую полезность с учетом свидетельств

```
function value_of_information(𝒫, query, evidence, M)
    φ = infer(M, 𝒫.bn, query, evidence)
    voi = -solve(𝒫, evidence, M).u
    query_vars = filter(v->v.name ∈ query, 𝒫.chance_vars)
    for o′ in assignments(query_vars)
        oo′ = merge(evidence, o′)
        p = φ.table[o′]
        voi += p*solve(𝒫, oo′, M).u
    end
    return voi
end
```

Полезность информации никогда не бывает отрицательной. Ожидаемая полезность может увеличиться только в том случае, если дополнительные наблюдения могут привести к другим оптимальным решениям. Если наблюдение за новой переменной O' не влияет на выбор действия, то $EU^*(o, o') = EU^*(o)$ для всех o', и в этом случае уравнение (6.9) равно нулю. Например, если оптимальное решение состоит в том, чтобы лечить заболевание независимо от результата диагностического теста, то полезность наблюдения за результатом нового теста равна нулю.

Полезность информации отражает только увеличение ожидаемой полезности от наблюдения. С конкретным наблюдением могут быть связаны дополнительные затраты. Одни диагностические тесты могут быть простыми и недорогими, например измерение температуры тела; но другие являются намного более дорогостоящими и инвазивными, например пункция спинномозговой жидкости. Разумеется, полезность информации, полученной при помощи пункции, может быть намного выше, чем при измерении температуры, но следует учитывать стоимость тестов.

Полезность информации – важная и часто используемая метрика для выбора объекта наблюдения. Иногда значение этой метрики применяют для

определения правильной последовательности наблюдений. После каждого наблюдения определяют полезность информации оставшихся ненаблюдаемых переменных. Затем для наблюдения выбирают ненаблюдаемую переменную с наибольшей информативностью. Если новые наблюдения влекут за собой дополнительные затраты, то их вычитают из полезности информации при выборе следующей наблюдаемой переменной. Процесс продолжается до тех пор, пока удается найти новую переменную для наблюдения. Затем выбирают оптимальное действие. Этот способ не обязательно отражает действительно оптимальную последовательность наблюдений. Оптимальный порядок выбора наблюдений можно определить с помощью методов последовательного принятия решений, представленных в последующих главах.

6.7. Иррациональность

Теория принятия решений относится к разряду *нормативных*, она не является *дескриптивной теорией*, объясняющей и предсказывающей человеческое поведение. Человеческие суждения и предпочтения часто не следуют правилам рациональности, изложенным в разделе 6.1[13]. Даже эксперты-люди могут иметь весьма непоследовательный набор предпочтений, что может создать проблемы при разработке системы поддержки принятия решений, которая пытается максимизировать ожидаемую полезность.

Пример 6.6 показывает, что определенность часто преувеличивает несомненные потери по сравнению с просто вероятными потерями. Этот эффект проявляется и по отношению к выигрышу. Меньший, но гарантированный выигрыш для большинства людей предпочтительнее гораздо большего выигрыша, который является лишь возможным с определенной вероятностью, и это явление существенно нарушает аксиомы рациональности.

Пример 6.7 демонстрирует *эффект фрейминга*, или, как его еще называют, *эффект влияния рамок восприятия* (framing effect), когда люди по-разному выбирают варианты в зависимости от того, представлены ли они как потеря или как выигрыш. К отклонениям от предписаний теории полезности могут привести и многие другие когнитивные искажения[14]. Особенно аккуратно следует относиться к попыткам получения функций полезности от людей-экспертов при создании систем поддержки принятия решений. Хотя рекомендации системы поддержки принятия решений могут быть рациональными с формальной точки зрения, они не всегда точно отражают предпочтения человека в определенных ситуациях.

---

[13] Канеман и Тверски критикуют теорию ожидаемой полезности и вводят альтернативную модель, называемую *теорией перспектив*, которая, по-видимому, больше соответствует человеческому поведению. D. Kahneman, A. Tversky, *Prospect Theory: An Analysis of Decision Under Risk*, Econometrica, vol. 47, no. 2, pp. 263–292,1979.

[14] В нескольких недавних книгах обсуждается очевидная человеческая иррациональность. *D. Ariely, Predictably Irrational: The Hidden Forces That Shape Our Decisions*. Harper, 2008. J. Lehrer, *How We Decide*. Houghton Mifflin, 2009.

Пример 6.6. Эксперимент, демонстрирующий, что людям свойственно преувеличивать потери, которые являются определенными, по сравнению с потерями, которые просто вероятны. По статье A. Tversky, D. Kahneman, *The Framing of Decisions and the Psychology of Choice*, Science, vol. 211, no. 4481, pp. 453–458, 1981

Тверски и Канеман изучили предпочтения студентов университетов, которые отвечали на вопросы анкеты. Они представили студентам вопросы, касающиеся последствий эпидемии. Студентам предложили выбрать один из двух вариантов исхода эпидемии:

- *A*: 100%-ный шанс потерять 75 жизней;
- *B*: 80%-ный шанс потерять 100 жизней.

Наиболее предпочтителен вариант *B*. Из уравнения (6.2) мы знаем, что

$$U(\text{потеря } 75) < 0.8U(\text{потеря } 100). \tag{6.10}$$

Затем им было предложено выбрать один из двух следующих исходов:

- *C*: 10%-ный шанс потерять 75 жизней;
- *D*: 8%-ный шанс потерять 100 жизней.

В этой паре более предпочтительным оказался вариант *C*. Следовательно, $0.1U(\text{потеря } 75) > 0.08U(\text{потеря } 100)$. Умножаем обе части неравенства на 10 и получаем

$$U(\text{потеря } 75) > 0.8U(\text{потеря } 100). \tag{6.11}$$

Очевидно, что уравнения (6.10) и (6.11) приводят к противоречию. Заметим, что мы не делали никаких предположений относительно фактического значения $U(\text{потеря } 75)$ и $U(\text{потеря } 100)$ – мы даже не предполагали, что потеря 100 жизней хуже, чем потеря 75 жизней. Поскольку уравнение (6.2) следует непосредственно из аксиом фон Неймана–Моргенштерна, приведенных в разделе 6.1, происходит нарушение по крайней мере одной из аксиом, даже если многие люди, выбирающие варианты *B* и *C*, находят эти аксиомы правильными.

Пример 6.7. Эксперимент, демонстрирующий эффект фрейминга. По статье A. Tversky, D. Kahneman, *The Framing of Decisions and the Psychology of Choice*, Science, vol. 211, no. 4481, pp. 453–458, 1981

Тверски и Канеман продемонстрировали эффект фрейминга, используя гипотетический сценарий, в котором ожидается, что эпидемия убьет 600 человек. Они предложили студентам следующие два исхода:

- *E*: 200 человек будут спасены;
- *F*: вероятность 1/3, что 600 человек будут спасены, и вероятность 2/3, что никто не будет спасен.

Большинство студентов предпочли исход E. Затем их попросили выбрать между следующими исходами:

- G: погибнет 400 человек;
- H: вероятность 1/3, что никто не умрет, и вероятность 2/3, что погибнет 600 человек.

Большинство студентов предпочли исход H, хотя ранее выбранный исход E эквивалентен G, а F эквивалентен H. Это несоответствие связано исключительно с формулировкой вопроса.

6.8. Заключение

- Рациональное принятие решений сочетает в себе теории вероятности и полезности.
- Существование функции полезности следует из ограничений на рациональные предпочтения.
- Рациональное решение – это такое решение, при котором ожидаемая полезность максимальна.
- Задачи принятия решений можно моделировать с помощью сетей принятия решений, которые являются расширенными байесовскими сетями, включающими действия и полезности.
- Получение простого решения требует логического вывода в байесовских сетях и поэтому является NP-трудным.
- Полезность информации отражает прирост ожидаемой полезности в случае наблюдения новой переменной.
- Люди не всегда рациональны.

6.9. Упражнения

Упражнение 6.1. Предположим, что у нас есть функция полезности $U(s)$ с конечным максимальным значением $\overline{U}$ и конечным минимальным значением $\underline{U}$. Какова соответствующая нормализованная функция полезности $\hat{U}(s)$, сохраняющая те же предпочтения?

Решение. Нормализованная функция полезности имеет максимальное значение 1 и минимальное значение 0. Предпочтения сохраняются при аффинных преобразованиях, поэтому мы определяем аффинное преобразование $U(s)$, которое подгоняет значения к границам единичного диапазона. Это преобразование выглядит следующим образом:

$$\hat{U}(s) = \frac{U(s) - \underline{U}}{\bar{U} - \underline{U}} = \frac{1}{\bar{U} - \underline{U}} U(s) - \frac{\underline{U}}{\bar{U} - \underline{U}}.$$

Упражнение 6.2. Если A ≽ C ≽ B и полезности каждого исхода равны $U(A) = 450$, $U(B) = -150$ и $U(C) = 60$, какая лотерея по A и B сделает эти два исхода эквивалентными исходу C?

Решение. Лотерея между A и B определяется как $[A{:}p; B{:}1 - p]$. Чтобы обеспечить безразличие выбора между этой лотереей и C (то есть $[A{:}p; B{:}1 - p] \sim C$), мы должны выполнить условие $U([A{:}p; B{:}1 - p]) = U(C)$. Таким образом, мы должны вычислить значение вероятности p, удовлетворяющее равенству

$$U([A{:}p; B{:}1 - p]) = U(C);$$

$$pU(A) + (1 - p)U(B) = U(C);$$

$$p = \frac{U(C) - U(B)}{U(A) - U(B)};$$

$$p = \frac{60 - (-150)}{450 - (-150)} = 0.35.$$

Отсюда следует, что лотерея исходов $[A{:}0.35; B{:}0.65]$ эквивалентна исходу C, как и требовалось.

Упражнение 6.3. Предположим, что функция полезности U по трем исходам A, B и C дает значения $U(A) = 5$, $U(B) = 20$ и $U(C) = 0$. Вам предоставляется выбор между лотереей, которая дает 50%-ную вероятность B и 50%-ную вероятность C, и лотереей, которая гарантирует 100%-ную вероятность A. Вычислите предпочтительную лотерею и покажите, что при положительном аффинном преобразовании с $m = 2$ и $b = 30$ ваше предпочтение лотереи не изменится.

Решение. Первая лотерея определяется как $[A{:}0.0; B{:}0.5; C{:}0.5]$, а вторая лотерея – как $[A{:}1.0; B{:}0.0; C{:}0.0]$. Исходные полезности для каждой лотереи задаются выражениями

$$U([A{:}0.0; B{:}0.5; C{:}0.5]) = 0.0U(A) + 0.5U(B) + 0.5U(C) = 10;$$

$$U([A{:}1.0; B{:}0.0; C{:}0.0]) = 1.0U(A) + 0.0U(B) + 0.0U(C) = 5.$$

Таким образом, поскольку $U([A{:}0.0; B{:}0.5; C{:}0.5]) > U([A{:}1.0; B{:}0.0; C{:}0.0])$, вам следует предпочесть первую лотерею. При положительном аффинном преобразовании $m = 2$ и $b = 30$ наши новые полезности могут быть вычислены как $U' = 2U + 30$. Тогда новые полезности равны $U'(A) = 40$, $U'(B) = 70$ и $U'(C) = 30$. Новые полезности для каждой лотереи равны

$$U'([A{:}0.0; B{:}0.5; C{:}0.5]) = 0.0U'(A) + 0.5U'(B) + 0.5U'(C) = 50;$$

$U'([A{:}1.0; B{:}0.0; C{:}0.0]) = 1.0U'(A) + 0.0U'(B) + 0.0U'(C) = 40.$

Поскольку $U'([A{:}0.0; B{:}0.5; C{:}0.5]) > U'([A{:}1.0; B{:}0.0; C{:}0.0])$, вы сохраняете предпочтение первой лотереи.

Упражнение 6.4. Докажите, что функция энергетической полезности в уравнении (6.5) приводит к избеганию риска для всех $x > 0$ и $\lambda > 0$, где $\lambda \neq 1$.

Решение. Избегание риска подразумевает, что функция полезности вогнута, а это означает, что вторая производная функции полезности должна быть отрицательной. Функция полезности и ее производные вычисляются следующим образом:

$$U(x) = \frac{x^{1-\lambda} - 1}{1-\lambda};$$

$$\frac{dU}{dx} = \frac{1}{x^{\lambda}};$$

$$\frac{d^2U}{dx^2} = \frac{-\lambda}{x^{\lambda+1}}.$$

Поскольку $x^{\lambda+1}$ при $x > 0, \lambda > 0$ и $\lambda \neq 1$ – положительное число, возведенное в положительную степень, оно гарантированно будет положительным. Умножение его на $-\lambda$ гарантирует, что вторая производная отрицательна. Таким образом, для всех $x > 0$ и $\lambda > 0, \lambda \neq 1$ функция полезности ведет к избеганию риска.

Упражнение 6.5. Используя условия, приведенные в примере 6.3, вычислите ожидаемую полезность того, что вы возьмете зонт, если ожидается солнце, и ожидаемую полезность того, что вы оставите зонт, если ожидается солнце. Какое действие максимизирует ожидаемую полезность, учитывая, что прогноз погоды обещает солнце?

Решение:

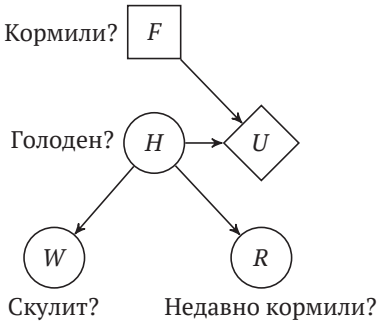
$EU(\text{взять зонт} \mid \text{ожидается солнце}) = 0.2 \times (-0.1) + 0.8 \times 0.9 = 0.7.$

$EU(\text{оставить зонт} \mid \text{ожидается солнце}) = 0.2 \times (-1.0) + 0.8 \times 1.0 = 0.6.$

Действие, которое максимизирует ожидаемую полезность, если прогноз погоды обещает солнце, – это взять зонт!

Упражнение 6.6. Предположим, что вы пытаетесь найти оптимальное решение, кормить ли (F) вашего щенка, основываясь на вероятности того, что щенок голоден (H). Вы можете наблюдать, скулит ли щенок (W) и кормил ли его кто-то недавно (R). Полезность каждой комбинации кормления и наблюдения признаков голода, а также схема сети принятия решений приведены ниже:

F	H	$U(F, H)$
0	0	0.0
0	1	−1.0
1	0	−0.5
1	1	−0.1



Учитывая, что $P(h^1|w^1) = 0.78$, если мы наблюдаем скуление щенка (w^1), какова ожидаемая полезность *отказа от кормления* щенка (f^0) и кормления щенка (f^1)? Каким будет оптимальное действие?

Решение. Начнем с определения ожидаемой полезности и отметим, что полезность зависит только от H и F:

$$EU(f^0|w^1) = \sum_h P(h|w^1)U(f^0, h).$$

Теперь мы можем вычислить ожидаемую полезность кормления щенка, при условии что он скулит, и ожидаемую полезность отказа от кормления щенка, при условии что он скулит:

$$EU(f^0|w^1) = 0.22 \times 0.0 + 0.78 \times (-1.0) = -0.78;$$

$$EU(f^1|w^1) = 0.22 \times (-0.5) + 0.78 \times (-0.1) = -0.188.$$

Таким образом, оптимальное действие – накормить щенка (f^1), так как это максимизирует ожидаемую полезность $EU^*(w^1) = -0.188$.

Упражнение 6.7. Исходя из результатов упражнения 6.6 и при условии, что $P(r^1|w^1) = 0.2$, $P(h^1|w^1, r^0) = 0.9$ и $P(h^1|w^1, r^1) = 0.3$, найдите полезность полученной от кого-то еще информации о том, что щенка недавно покормили, учитывая, что щенок все равно скулит (w^1).

Решение. Вам необходимо вычислить следующее значение:

$$VOI(R|w^1) = \left(\sum_r P(r|w^1)EU^*(w^1, r)\right) - EU^*(w^1).$$

Начнем с вычисления $EU(f|w^1, r)$ для всех f и r. Следуя тому же выводу, что и в упражнении 6.6, имеем:

$$EU(f^0|w^1, r^0) = \sum_h P(h|w^1, r^0)U(f^0, h).$$

Таким образом, для каждой комбинации F и R мы имеем следующие ожидаемые полезности:

$$EU(f^0|w^1,r^0) = \sum_h P(h|w^1,r^0)U(f^0,h) = 0.1\times 0.0 + 0.9\times -1.0 = -0.9;$$

$$EU(f^1|w^1,r^0) = \sum_h P(h|w^1,r^0)U(f^1,h) = 0.1\times -0.5 + 0.9\times -0.1 = -0.14;$$

$$EU(f^0|w^1,r^1) = \sum_h P(h|w^1,r^1)U(f^0,h) = 0.7\times 0.0 + 0.3\times -1.0 = -0.3;$$

$$EU(f^1|w^1,r^1) = \sum_h P(h|w^1,r^1)U(f^1,h) = 0.7\times -0.5 + 0.3\times -0.1 = -0.38.$$

Оптимальные ожидаемые полезности равны

$EU^*(w^1, r^0) = -0.14;$

$EU^*(w^1, r^1) = -0.3.$

Теперь мы можем вычислить полезность информации:

$VOI(R|w^1) = 0.8(-0.14) + 0.2(-0.3) - (-0.188) = 0.016.$

Часть II

Задачи последовательного принятия решений

До сих пор мы предполагали, что принимаем единственное решение в определенный момент времени, но многие важные задачи требуют принятия ряда последовательных решений. В этом случае тоже применяется принцип максимальной ожидаемой полезности, но оптимальный выбор последовательных решений требует рассуждений о будущих последовательностях действий и наблюдений. Во второй части книги рассмотрены проблемы последовательного принятия решений в стохастической среде. Мы сосредоточимся на общей формулировке задачи последовательного принятия решений с предположениями, что модель известна и что окружающая среда полностью наблюдаема. Позже мы ослабим оба этих предположения. Мы начнем с понятия марковского процесса принятия решений – стандартной математической модели для задач последовательного принятия решений. Мы обсудим несколько подходов к поиску точных решений. Поскольку масштабные задачи иногда не позволяют эффективно находить точные решения, мы обсудим различные методы поиска приближенных решений, а также метод прямого поиска в пространстве параметрических стратегий принятия решений. Наконец, рассмотрим способы проверки стратегий принятия решений, чтобы убедиться, что они будут работать должным образом при использовании в реальном мире.

7 Методы точного решения

В этой главе мы рассмотрим *марковский процесс принятия решений* (Markov decision process, MDP), применяемый для представления задач последовательного принятия решений, когда последствия наших действий не детерминированы[1]. Мы начнем с описания модели, которая определяет как стохастическую динамику системы, так и полезность, связанную с ее эволюцией. Для вычисления полезности, связанной со стратегией принятия решений, и для поиска оптимальной стратегии применяются различные алгоритмы. Сделав определенные предположения, мы можем найти точные решения задачи MDP. В последующих главах будут рассмотрены приближенные методы, которые, как правило, лучше масштабируются для решения более масштабных задач.

7.1. Марковские процессы принятия решений

Согласно MDP (алгоритм 7.1) мы выбираем действие a_t в момент времени t на основе наблюдаемого состояния s_t. Затем получаем вознаграждение r_t. *Пространство действий* $\mathcal{A}$ – это множество возможных действий, а *пространство состояний* $\mathcal{S}$ – это множество возможных состояний. Некоторые алгоритмы предполагают, что эти множества конечны, но в общем случае это не требуется. Состояние развивается стохастически на основе текущего состояния и действий, которые мы предпринимаем. Предположение о том, что следующее состояние зависит только от текущего состояния и действия, а не от какого-либо предшествующего состояния или действия, называется *допущением Маркова* (Markov assumption).

MDP можно представить с помощью сети принятия решений, как показано на рис. 7.1. Существуют информационные ребра (здесь не показаны) от $A_{1:t-1}$ и $S_{1:t}$ до A_t. Функция полезности разлагается на вознаграждения $R_{1:t}$. Мы сосредоточимся на *стационарных* MDP, в которых $P(S_{t+1}|S_t, A_t)$ и $P(R_t|S_t, A_t)$ не меняются со временем. Стационарные MDP могут быть компактно представлены динамической диаграммой решений, как показано на рис. 7.2. *Модель перехода состояний* $T(s'|s, a)$ представляет вероятность перехода из состояния s в s' после выполне-

[1] Такие модели первоначально изучались в 1950-х годах: R. E. Bellman, *Dynamic Programming*. Princeton University Press, 1957. Современную трактовку можно найти в книге M. L. Puterman, Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley, 2005.

ния действия *a*. *Функция вознаграждения* $R(s, a)$ представляет собой ожидаемое вознаграждение, полученное при выполнении действия *a* из состояния *s*.

Алгоритм 7.1. Структура данных для MDP. Позже мы будем использовать поле `TR` для выборки следующего состояния и вознаграждения с учетом текущего состояния и действия: `s′, r = TR(s, a)`. В математической нотации MDP иногда определяют в терминах кортежа, состоящего из различных компонентов MDP, записанных в виде $(\mathcal{S}, \mathcal{A}, T, R, \gamma)$

```
struct MDP
    γ  # коэффициент дисконтирования
    𝒮  # пространство состояний
    𝒜  # пространство действий
    T  # переходная функция
    R  # функция вознаграждения
    TR # выборка перехода и вознаграждения
end
```

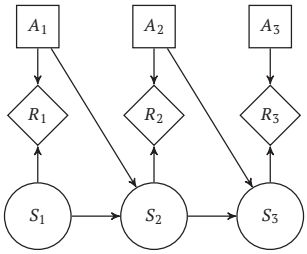


Рис. 7.1. Схема сети принятия решений MDP

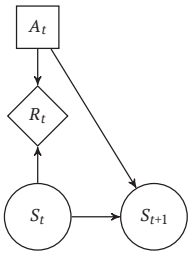


Рис. 7.2. Схема стационарной сети принятия решений MDP. Все MDP имеют эту общую структуру

Функция вознаграждения является детерминированной функцией от *s* и *a*, поскольку она представляет собой ожидание, но вознаграждение может генерироваться стохастически в окружающей среде или даже зависеть от результирующего следующего состояния[2]. В примере 7.1 показано, как формулируется задача предотвращения столкновений в виде MDP.

[2] Например, если вознаграждение зависит от следующего состояния, данного в $R(s, a, s')$, то функция ожидаемого вознаграждения будет иметь следующий вид:

$R(s,a) = \sum_{s'} T(s'|s,a)R(s,a,s')$.

Пример 7.1. Задача предотвращения столкновений самолетов, сформулированная как MDP. Многие другие практические приложения обсуждаются в D. J. White, *A Survey of Applications of Markov Decision Processes*, Journal of the Operational Research Society, vol. 44, no. 11, pp. 1073–1096, 1993

Задачу предотвращения столкновений самолетов можно сформулировать как MDP. Состояния представляют позиции и скорости двух самолетов – нашего и встречного, а действия отражают один из трех возможных вариантов – набор высоты, снижение или сохранение высоты. Мы получаем большое отрицательное вознаграждение за столкновение с другим самолетом и небольшое отрицательное вознаграждение за набор высоты или снижение.

Зная текущее состояние, мы должны решить, требуется ли маневр уклонения. Это сложная задача, потому что положение самолета изменяется вероятностно, а мы должны начать маневр достаточно рано, чтобы успеть избежать столкновения, но не слишком рано, чтобы избежать ненужного маневрирования.

Вознаграждения в MDP рассматриваются как компоненты аддитивно разложенной функции полезности. В задаче с *конечным горизонтом* (finite horizon) с n решениями полезность, связанная с последовательностью вознаграждений $r_{1:n}$, представляет собой простую сумму

$$\sum_{t=1}^{n} r_t. \tag{7.1}$$

Сумму вознаграждений иногда называют *доходом*.

В задаче с *бесконечным горизонтом* (infinite horizon), в которой количество решений не ограничено, сумма вознаграждений может стать бесконечной[3]. Существует несколько способов определить полезность с точки зрения индивидуальных вознаграждений в задачах с бесконечным горизонтом. Один из способов – ввести *коэффициент дисконтирования* (discount factor) γ, принимающий значения в интервале от 0 до 1. В этом случае полезность определяется выражением

$$\sum_{t=1}^{\infty} \gamma^{t-1} r_t. \tag{7.2}$$

Значение (7.2) иногда называют *дисконтированным доходом*. Пока $0 \leqslant \gamma < 1$ и вознаграждения конечны, полезность будет конечной. Наличие коэффициента дисконтирования приводит к тому, что вознаграждение в настоящем

[3] Предположим, что стратегия A приводит к вознаграждению в размере 1 за каждый временной шаг, а стратегия B приводит к вознаграждению в размере 100 за временной шаг. Интуитивно рациональный агент должен предпочесть стратегию B стратегии A, но обе они обеспечивают одну и ту же бесконечную ожидаемую полезность.

стоит больше, чем вознаграждение в будущем – идея, которая также широко применяется в экономике.

Другой способ определить полезность в задачах с бесконечным горизонтом – использовать *среднее вознаграждение* (average reward), также называемое *средним доходом* (average return) и вычисляемое по формуле

$$\lim_{n\to\infty}\frac{1}{n}\sum_{t=1}^{n} r_t. \tag{7.3}$$

Формула (7.3) выглядит привлекательно, поскольку нам не нужно выбирать коэффициент дисконтирования, но часто нет практической разницы между ней и формулой дисконтированного дохода с коэффициентом дисконтирования, близким к 1. Поскольку дисконтированный доход зачастую проще в вычислительном отношении, мы сосредоточимся на формуле (7.2).

Стратегия (policy) указывает нам, какое действие выбрать, учитывая прошлую историю состояний и действий. Действие, которое нужно выбрать в момент времени t, учитывая *историю* $h_t = (s_{1:t}, a_{1:t-1})$, записывается как $\pi_t(h_t)$. Поскольку будущие состояния и вознаграждения зависят только от текущего состояния и действия (как видно из предположений условной независимости на рис. 7.1), мы можем ограничить свое внимание стратегиями, которые зависят только от текущего состояния. Кроме того, мы сосредоточимся в первую очередь на *детерминированных стратегиях*, поскольку в MDP гарантированно существует оптимальная детерминированная стратегия. В последующих главах обсуждаются *стохастические стратегии*, где $\pi_t(a_t|s_t)$ обозначает вероятность, которую стратегия определяет для совершения действия a_t в состоянии s_t в момент времени t.

В задачах с бесконечным горизонтом со стационарными переходами и вознаграждениями мы можем дополнительно ограничить наше внимание *стационарными стратегиями*, которые не зависят от времени. Мы можем записать действие, связанное со стационарной стратегией π в состоянии s, как $\pi(s)$ без индекса времени. Однако в задачах с конечным горизонтом может быть полезно выбрать действие в зависимости от того, сколько временных шагов осталось. Например, при игре в баскетбол обычно нежелательно делать бросок с середины площадки, если до конца игры не осталось всего пары секунд. Мы можем заставить стационарные стратегии учитывать время, включив его в набор переменных состояния.

Ожидаемая полезность выполнения стратегии π из состояния s обозначается как $U^{\pi}(s)$. В контексте MDP функцию $U^{\pi}(s)$ часто называют *функцией ценности* (value function), но, чтобы не вносить путаницу, мы продолжим использовать прежнее, более общее название – *функция полезности*. Оптимальная стратегия π^* – это такая стратегия, при соблюдении которой достигается максимальная ожидаемая полезность[4]

[4] Это согласуется с принципом максимальной ожидаемой полезности, представленным в разделе 6.4.

$$\pi^*(s) = \arg\max_{\pi} U^{\pi}(s) \tag{7.4}$$

для всех состояний s. Функция полезности, связанная с оптимальной стратегией π^*, называется *оптимальной функцией полезности* (optimal value function) и обозначается как U^*.

Оптимальную стратегию можно найти, используя особую технику вычислений, называемую *динамическим программированием*[5], которая представляет собой упрощение сложной задачи путем ее рекурсивного разбиения на более простые подзадачи. Хотя мы сосредоточимся на алгоритмах динамического программирования для MDP, динамическое программирование – это общий подход, который можно применять к большому количеству других задач. Например, динамическое программирование можно использовать для вычисления последовательности Фибоначчи и поиска самой длинной общей подпоследовательности между двумя строками[6]. В целом алгоритмы, использующие динамическое программирование для решения MDP, намного эффективнее, чем методы прямых вычислений.

7.2. Оценка стратегии

Прежде чем перейти к выводу оптимальной стратегии, обсудим *оценку стратегии* (policy evaluation), где мы вычисляем функцию полезности U^{π}. Оценка стратегии может выполняться итеративно. Если стратегия выполняется для одного шага, полезность равна $U_1^{\pi} = R(s, \pi(s))$. Дальнейшие шаги можно получить из *уравнения предпросмотра* (Lookahead equation; в теории алгоритмов термин lookahead означает предварительный просмотр как можно большего количества доступных вариантов действия или входных элементов перед принятием оптимального решения. – *Прим. перев.*):

$$U_{k=1}^{\pi}(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s'|s, \pi(s)) U^{\pi}(s'). \tag{7.5}$$

Это уравнение реализовано в алгоритме 7.2. Итеративная оценка стратегии реализована в алгоритме 7.3. Несколько итераций показаны на рис. 7.3.

---

[5] Термин «динамическое программирование» был введен американским математиком Ричардом Эрнестом Беллманом (1920–1984). Определение «динамическое» обозначает тот факт, что проблема меняется во времени, а «программирование» относится к методологии поиска оптимальной программы или стратегии решения. R. Bellman, Eye of the Hurricane: An Autobiography. World Scientific, 1984.

[6] R. Bellman, *Eye of the Hurricane: An Autobiography*. World Scientific, 1984.

Алгоритм 7.2. Функции для вычисления предпросмотра состояния-действия из состояния `s` при заданном действии `a` с использованием оценки функции полезности `U` для MDP $\mathcal{P}$. Вторая версия алгоритма обрабатывает случай, когда `U` является вектором

```
function lookahead(𝒫::MDP, U, s, a)
    𝒮, T, R, γ = 𝒫.𝒮, 𝒫.T, 𝒫.R, 𝒫.γ
    return R(s,a) + γ*sum(T(s,a,s′)*U(s′) for s′ in 𝒮)
end
function lookahead(𝒫::MDP, U::Vector, s, a)
    𝒮, T, R, γ = 𝒫.𝒮, 𝒫.T, 𝒫.R, 𝒫.γ
    return R(s,a) + γ*sum(T(s,a,s′)*U[i] for (i,s′) in enumerate(𝒮))
end
```

Алгоритм 7.3. Итеративная оценка стратегии, где мы вычисляем функцию полезности стратегии π для MDP $\mathcal{P}$ с дискретными пространствами состояний и действий, используя `k_max` итераций

```
function iterative_policy_evaluation(𝒫::MDP, π, k_max)
    𝒮, T, R, γ = 𝒫.𝒮, 𝒫.T, 𝒫.R, 𝒫.γ
    U = [0.0 for s in 𝒮]
    for k in 1:k_max
        U = [lookahead(𝒫, U, s, π(s)) for s in 𝒮]
    end
    return U
end
```

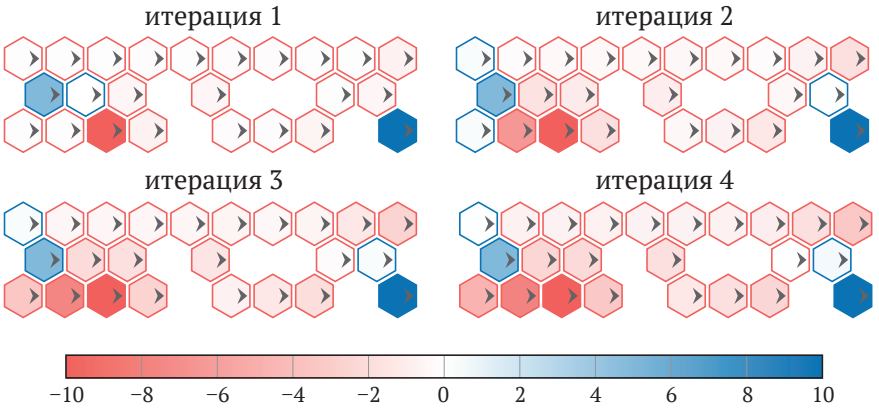


Рис. 7.3. Итеративная оценка стратегии, используемая для оценки стратегии движения на восток в задаче гексамира (представлена в приложении F.1). Стрелки указывают направление, рекомендованное стратегией (т. е. всегда двигаться на восток), а цвета обозначают значения, связанные с состояниями. Значения меняются с каждой итерацией

Функция полезности U^π может быть вычислена с произвольной точностью при наличии достаточного количества итераций уравнения следующего шага. Сходимость гарантирована, поскольку обновление в уравнении (7.5) представляет собой *сжатое отображение* (contraction mapping, рассмотрено в приложении A.15)[7].

При сходимости выполняется следующее равенство:

$$U^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s'|s, \pi(s)) U^\pi(s'). \tag{7.6}$$

Это равенство называется *уравнением ожидания Беллмана*[8].

Оценка стратегии может быть выполнена без итерации путем непосредственного решения системы уравнений в уравнении ожидания Беллмана. Уравнение (7.6) определяет множество $|S|$ линейных уравнений с $|S|$ неизвестных, соответствующих полезности в каждом состоянии. Один из способов решить эту систему уравнений – сначала преобразовать ее в матричную форму:

$$\mathbf{U}^\pi = \mathbf{R}^\pi + \gamma \mathbf{T}^\pi \mathbf{U}^\pi, \tag{7.7}$$

где $\mathbf{U}^\pi$ и $\mathbf{R}^\pi$ – функции полезности и вознаграждения, представленные в векторной форме, где вектор имеет $|\mathcal{S}|$ компонент. Матрица $\mathbf{T}^\pi$ размером $|\mathcal{S}| \times |\mathcal{S}|$ содержит вероятности перехода состояний, где T_{ij}^π – вероятность перехода из i-го состояния в j-е.

Функцию полезности находят следующим образом:

$$\mathbf{U}^\pi - \gamma \mathbf{T}^\pi \mathbf{U}^\pi = \mathbf{R}^\pi; \tag{7.8}$$

$$(\mathbf{I} - \gamma \mathbf{T}^\pi)\mathbf{U}^\pi = \mathbf{R}^\pi; \tag{7.9}$$

$$\mathbf{U}^\pi = (\mathbf{I} - \gamma \mathbf{T}^\pi)^{-1} \mathbf{R}^\pi. \tag{7.10}$$

Этот метод реализован в алгоритме 7.4. Нахождение $\mathbf{U}^\pi$ таким способом занимает $O(|\mathcal{S}|^3)$ времени. Метод используется для оценки стратегии на рис. 7.4.

Алгоритм 7.4. Метод точной оценки стратегии, который вычисляет функцию полезности стратегии π для MDP $\mathcal{P}$ с дискретными пространствами состояний и действий

```
function policy_evaluation(𝒫::MDP, π)
    𝒮, R, T, γ = 𝒫.𝒮, 𝒫.R, 𝒫.T, 𝒫.γ
    R′ = [R(s, π(s)) for s in 𝒮]
    T′ = [T(s, π(s), s′) for s in 𝒮, s′ in 𝒮]
    return (I - γ*T′)\R′
end
```

---

[7] См. упражнение 7.12.

[8] Это уравнение названо в честь Ричарда Беллмана, одного из пионеров динамического программирования. R. E. Bellman, *Dynamic Programming*. Princeton University Press, 1957.

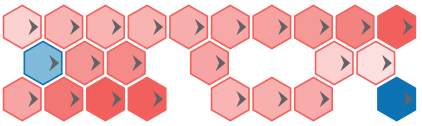


Рис. 7.4. Точная оценка стратегии, используемая для оценки стратегии движения на восток в задаче гексамира. Точное решение дает более низкие значения, чем те, что получились в первых нескольких шагах итеративной оценки стратегии согласно рис. 7.3. Если бы мы запускали итеративную оценку стратегии для большего количества итераций, она сходилась бы к точному значению функции полезности

7.3. Нахождение стратегии через функцию полезности

В предыдущем разделе говорилось о том, как найти функцию полезности, связанную со стратегией. В этом разделе будет показано, как извлечь стратегию из функции полезности, которую мы позже будем использовать при создании оптимальных стратегий. Имея функцию полезности U, которая может совпадать или не совпадать с функцией оптимальной полезности, мы можем построить стратегию π, которая максимизирует уравнение следующего шага, представленное в (7.5):

$$\pi(s) = \arg\max_a \left(R(s,a) + \gamma \sum_{s'} T(s'|s,a) U(s') \right). \tag{7.11}$$

Мы будем называть это *жадной стратегией* (greedy policy) по отношению к U. Если U – оптимальная функция полезности, то извлеченная стратегия оптимальна. Алгоритм 7.5 реализует эту идею.

Алгоритм 7.5. Стратегия функции полезности, извлеченная из функции полезности `U` для MDP $\mathcal{P}$. Функция `greedy` также будет использоваться в других алгоритмах

```
struct ValueFunctionPolicy
    𝒫 # задача
    U  # функция полезности
end

function greedy(𝒫::MDP, U, s)
    u, a = findmax(a->lookahead(𝒫, U, s, a), 𝒫.𝒜)
    return (a=a, u=u)
end

(π::ValueFunctionPolicy)(s) = greedy(π.𝒫, π.U, s).a
```

Альтернативным способом представления стратегии является использование *функции полезности действия* (action value function), иногда называемой *Q-функцией*. Функция полезности действия представляет собой ожидаемый доход при старте в состоянии s, выполнении действия a и последующем продолжении жадной стратегии по отношению к Q:

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} T(s'|s,a)U(s'). \tag{7.12}$$

Из этой функции полезности действия мы можем получить функцию полезности,

$$U(s) = \max_a Q(s,a), \tag{7.13}$$

а также стратегию

$$\pi(s) = \arg\max_a Q(s,a). \tag{7.14}$$

Явное хранение Q для дискретных задач занимает $\mathcal{O}(|\mathcal{S}|\times|\mathcal{A}|)$ памяти вместо $\mathcal{O}(|\mathcal{S}|)$ памяти для хранения U, но нам не нужно использовать R и T для извлечения стратегии.

Стратегии также могут быть представлены с использованием *функции преимущества* (advantage function), которая количественно определяет преимущество выполнения действия по сравнению с жадным действием. Эта функция определяется как разность между Q и U:

$$A(s, a) = Q(s, a) - U(s). \tag{7.15}$$

Действия в соответствии с жадной стратегией имеют нулевое преимущество, а нежадные действия имеют отрицательное преимущество. Некоторые алгоритмы, которые мы обсудим позже в этой книге, используют U-представление, а другие будут использовать Q или A.

7.4. Итерация по стратегиям

Итерация по стратегиям (алгоритм 7.6) – один из способов вычисления оптимальной стратегии. Она заключается в итеративном чередовании между оценкой стратегии (раздел 7.2) и улучшением стратегии с помощью метода жадной стратегии (алгоритм 7.5). Итерация по стратегиям гарантированно сходится при любой исходной стратегии. Она сходится за конечное число итераций, потому что существует конечное число стратегий, и каждая итерация улучшает стратегию, если ее можно улучшить. Хотя количество возможных стратегий экспоненциально зависит от количества состояний, итерация часто быстро сходится. На рис. 7.5 показана итерация по стратегиям для решения задачи гексамира.

Алгоритм 7.6. Итерация по стратегиям, которая пошагово улучшает исходную стратегию π для получения оптимальной стратегии MDP $\mathcal{P}$ с дискретными пространствами состояний и действий

```
struct PolicyIteration
    π # initial policy
    k_max # maximum number of iterations
end

function solve(M::PolicyIteration, 𝒫::MDP)
    π, 𝒮 = M.π, 𝒫.𝒮
    for k = 1:M.k_max
        U = policy_evaluation(𝒫, π)
        π′ = ValueFunctionPolicy(𝒫, U)
        if all(π(s) == π′(s) for s in 𝒮)
            break
        end
        π = π′
    end
    return π
end
```

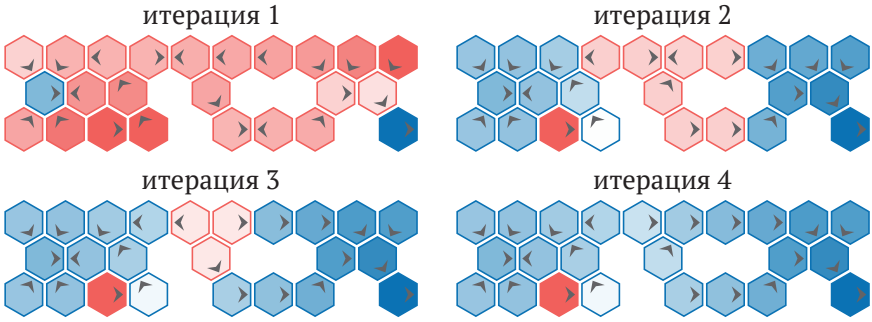


Рис. 7.5. Итерация по стратегиям, используемая для итеративного улучшения изначально направленной на восток стратегии в задаче гексамира с целью получения оптимальной стратегии. В первой итерации мы видим функцию полезности, связанную со стратегией движения на восток, и стрелки, указывающие на стратегию, которая является жадной по отношению к этой функции полезности. Итерация по стратегиям сходится в четыре шага; если бы мы выполнили пять или более итераций, мы бы получили ту же стратегию

Итерация по стратегиям часто бывает вычислительно затратной, потому что в каждой итерации мы должны вычислять оценку стратегии. Вариант итерационного поиска стратегии, называемый *модифицированной итерацией по стратегиям* (modified policy iteration)[9], аппроксимирует функцию полезности,

[9] M. L. Puterman, M. C. Shin, *Modified Policy Iteration Algorithms for Discounted Markov Decision Problems*, Management Science, vol. 24, no. 11, pp. 1127–1137, 1978.

используя итеративную оценку стратегии вместо точной оценки. Мы можем выбрать количество итераций оценки стратегии между шагами улучшения. Если мы используем только одну итерацию между шагами, то этот подход идентичен итерации по критерию.

7.5. Итерация по критерию

Итерация по критерию (value iteration) – это альтернатива итерации по стратегиям, которая часто используется из-за ее простоты. В отличие от улучшения стратегии, итерация по критерию напрямую обновляет функцию полезности. Она начинается с любой ограниченной функции полезности U, подразумевая, что $|U(s)| < \infty$ для всех s. Одним из вариантов инициализации является $U(s) = 0$ для всех s.

Функцию полезности можно улучшить, применив *оператор Беллмана* (Bellman backup), также называемый *обновлением Беллмана* (Bellman update)[10]:

$$U_{k+1}(s) = \max_a \left(R(s,a) + \gamma \sum_{s'} T(s'|s,a) U_k(s') \right). \tag{7.16}$$

Эта процедура реализована в алгоритме 7.7.

Алгоритм 7.7. Оператор Беллмана, применяемый к MDP $\mathcal{P}$, который улучшает функцию полезности `U` в состоянии `s`

```
function backup(𝒫::MDP, U, s)
    return maximum(lookahead(𝒫, U, s, a) for a in 𝒫.𝒜)
end
```

Многократное применение этого оператора гарантированно даст сходимость к оптимальной функции полезности. Как и при итерации по стратегиям, мы можем использовать тот факт, что оператор представляет собой отображение сжатия, чтобы доказать сходимость[11]. Гарантируется, что эта оптимальная стратегия удовлетворяет *уравнению оптимальности Беллмана*:

$$U^*(s) = \max_a \left(R(s,a) + \gamma \sum_{s'} T(s'|s,a) U^*(s') \right). \tag{7.17}$$

Дальнейшие применения оператора Беллмана после выполнения этого равенства не изменяют функцию полезности. Оптимальная стратегия может быть получена из U^* с помощью уравнения (7.11). Итерация по критериям реализована в алгоритме 7.8 и применяется к задаче гексамира на рис. 7.6.

[10] Это действие можно условно назвать операцией резервного копирования, потому что она возвращает информацию в исходное состояние из ее будущих состояний.

[11] См. упражнение 7.13.

Реализация в алгоритме 7.8 останавливается после фиксированного числа итераций, но также итерации часто прекращают досрочно, основываясь на максимальном изменении значения $\|U_{k+1} - U_k\|_\infty$, называемого *невязкой Беллмана* (Bellman residual). Если невязка Беллмана оказывается ниже порога δ, то итерации прекращаются. Невязка Беллмана для δ гарантирует, что оптимальная функция полезности, полученная итерацией по критерию, находится в пределах $\varepsilon = \delta\gamma/(1 - \gamma)$ от U^*[12]. Коэффициенты дисконтирования, близкие к 1, значительно увеличивают эту ошибку, что замедляет сходимость. Если мы сильно дисконтируем предстоящее вознаграждение (γ приближается к 0), то нам не понадобится много итераций в будущем. Этот эффект продемонстрирован в примере 7.2.

Зная максимальное отклонение ожидаемой функции полезности от оптимальной функции полезности $\|U_k - U^*\|_\infty < \varepsilon$, мы можем ограничить максимальное отклонение вознаграждения, полученного благодаря извлеченной стратегии π, от вознаграждения при оптимальной стратегии π^*. Эти *потери стратегии* (policy loss) $\|U^\pi - U^*\|_\infty$ ограничены величиной $2\varepsilon\gamma/(1 - \gamma)$[13].

Алгоритм 7.8. Итерация по критерию, которая пошагово улучшает функцию полезности `U` для получения оптимальной стратегии MDP $\mathcal{P}$ с дискретными пространствами состояний и действий. Вычисления завершаются после `k_max` итераций

```
struct ValueIteration
    k_max # максимальное количество итераций
end

function solve(M::ValueIteration, 𝒫::MDP)
    U = [0.0 for s in 𝒫.𝒮]
    for k = 1:M.k_max
        U = [backup(𝒫, U, s) for s in 𝒫.𝒮]
    end
    return ValueFunctionPolicy(𝒫, U)
end
```

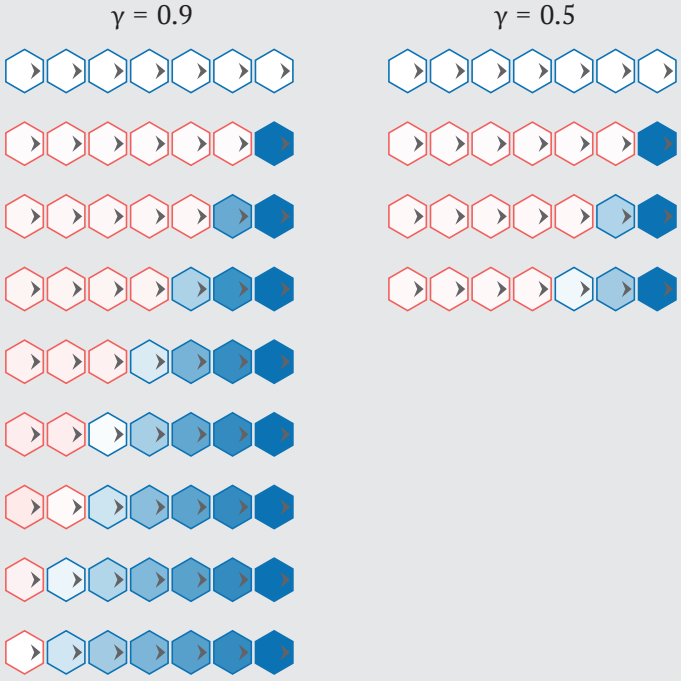
---

[12] См. упражнение 7.8.

[13] S. P. Singh and R. C. Yee, *An Upper Bound on the Loss from Approximate Optimal-Value Functions*, Machine Learning, vol. 16, no. 3, pp. 227–233, 1994.

Пример 7.2. Влияние коэффициента дисконтирования на сходимость итерации по критерию. В каждом случае итерация выполнялась до тех пор, пока невязка Беллмана не стала меньше 1

Рассмотрим простой вариант задачи о гексамире, состоящем из прямой линии плиток с одной потребляющей (конечной) плиткой в конце, дающей вознаграждение, равное 10. Коэффициент дисконтирования напрямую влияет на скорость, с которой вознаграждение от потребляющей плитки распространяется вниз по линии к другой плитке и, следовательно, как быстро сходится итерация по критерию.



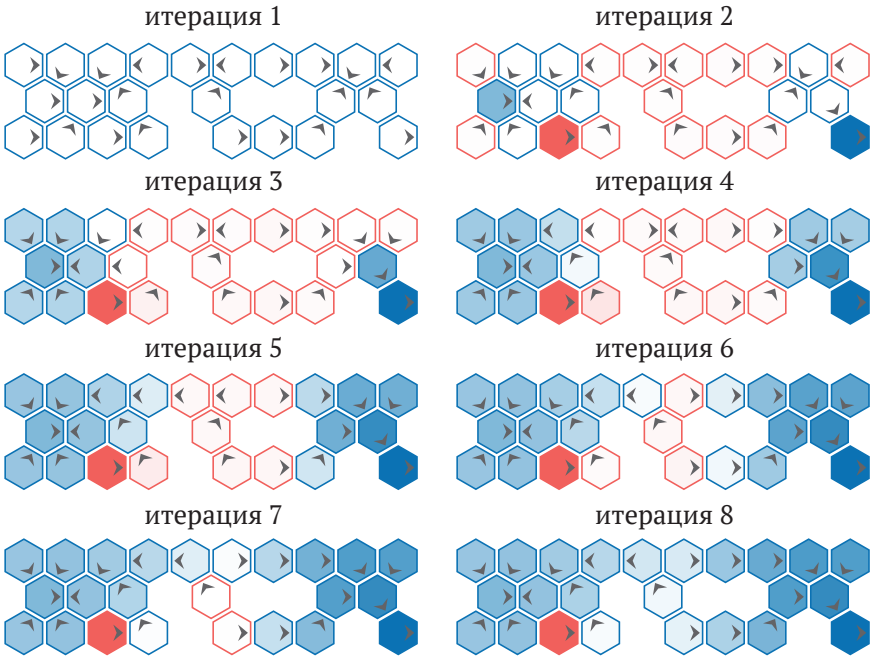


Рис. 7.6. Итерация по критерию в задаче гексамира для получения оптимальной стратегии. Каждый шестиугольник окрашен в соответствии с функцией полезности, а стрелки указывают стратегию, которая является жадной по отношению к этой функции полезности

7.6. Асинхронная итерация по критерию

Итерация по критерию, как правило, требует больших вычислительных ресурсов, поскольку каждое значение в функции полезности U_k обновляется на каждой итерации для получения U_{k+1}. При асинхронной итерации на каждом шаге обновляется только подмножество состояний. Асинхронная итерация по критерию по-прежнему гарантированно сойдется к оптимальной функции полезности, при условии что каждое состояние обновляется бесконечное количество раз.

Один из распространенных методов асинхронной итерации по критерию – *итерация Гаусса–Зейделя* (алгоритм 7.9), просматривает порядок состояний и применяет оператор Беллмана по месту (без создания в памяти копии объектов):

$$U(s) \leftarrow \max_a \left(R(s,a) + \gamma \sum_{s'} T(s'|s,a) U(s') \right). \tag{7.18}$$

Экономия вычислительных ресурсов заключается в том, что не нужно создавать в памяти вторую функцию-значение при каждой итерации. Итерация Гаусса–Зейделя может сходиться быстрее, чем стандартная итерация по критерию, но это зависит от выбранного порядка[14]. В некоторых задачах состояние содержит индекс времени, который детерминистически увеличивается по оси времени. Если мы применяем итерацию Гаусса–Зейделя, начиная с последнего временного индекса, и движемся в обратном направлении, этот процесс иногда называют *обратно-индуктивной итерацией по критерию* (backward induction value iteration). Пример влияния порядка состояний приведен в примере 7.3.

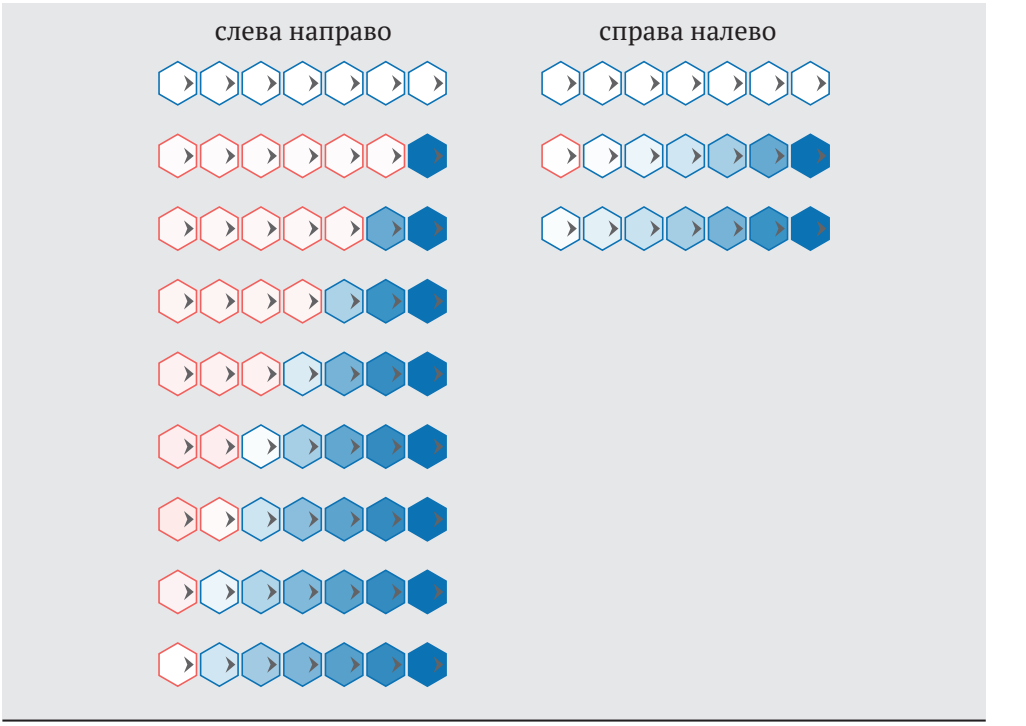
Алгоритм 7.9. Асинхронная итерация по критерию, которая обновляет состояния иначе, чем стандартная итерация по критерию, что часто экономит время вычислений. Вычисления завершаются после `k_max` итераций

```
struct GaussSeidelValueIteration
    k_max # максимальное количество итераций
end

function solve(M::GaussSeidelValueIteration, 𝒫::MDP)
    U = [0.0 for s in 𝒫.𝒮]
    for k = 1:M.k_max
        for (i, s) in enumerate(𝒫.𝒮)
            U[i] = backup(𝒫, U, s)
        end
    end
    return ValueFunctionPolicy(𝒫, U)
end
```

Пример 7.3. Влияние порядка следования состояний на сходимость асинхронной итерации по критерию. В этом случае оценка справа налево позволяет добиться сходимости за гораздо меньшее количество итераций

Рассмотрим линейную вариацию задачи гексамира из примера 7.2. Мы можем решить ту же задачу, используя асинхронную итерацию по критерию. Порядок следования состояний напрямую влияет на скорость, с которой награда от потребляющей плитки распространяется вниз по линии к другим плиткам, и, следовательно, на то, насколько быстро метод сходится.

[14] Неудачный порядок в итерации по критерию Гаусса–Зейделя не может привести к тому, что алгоритм будет работать медленнее, чем стандартная итерация по критерию.



7.7. Представление задачи в виде линейной программы

Задача поиска оптимальной стратегии может быть сформулирована как *линейная программа*, которая представляет собой задачу оптимизации с линейной целевой функцией и набором линейных ограничений равенства или неравенства. Представив задачу в виде линейной программы, далее можно использовать один из многих решателей линейного программирования[15].

Чтобы продемонстрировать преобразование уравнения оптимальности Беллмана в линейную программу, мы начнем с замены равенства в уравнении оптимальности Беллмана набором ограничений-неравенств при минимизации $U(s)$ в каждом состоянии s[16]:

[15] Обзор линейного программирования представлен в R. Vanderbei, *Linear Programming, Foundations and Extensions*, 4th ed. Springer, 2014.

[16] Разумно было бы понизить значение $U(s)$ во всех состояниях s, чтобы преобразовать ограничения неравенства в ограничения равенства. Следовательно, мы минимизируем сумму всех полезностей.

$$\text{минимизировать} \sum_s U(s),$$

$$\text{при условии что } U(s) \geq \max_a \left(R(s,a) + \gamma \sum_{s'} T(s'|s,a) U(s') \right) \text{ для всех } s. \tag{7.19}$$

Переменными в оптимизации являются полезности в каждом состоянии. Зная эти полезности, мы можем определить оптимальную стратегию, используя уравнение (7.11).

Операцию поиска максимума в ограничениях неравенства можно заменить набором линейных ограничений, что делает программу линейной:

$$\text{минимизировать} \sum_s U(s),$$

$$\text{при условии что } U(s) \geq R(s,a) + \gamma \sum_{s'} T(s'|s,a) U(s') \text{ для всех } s \text{ и } a. \tag{7.20}$$

В линейной программе, показанной в уравнении (7.20), количество переменных равно количеству состояний, а количество ограничений равно количеству состояний, умноженному на количество действий. Поскольку линейные программы могут быть выполнены за полиномиальное время[17], задачи MDP также могут быть решены за полиномиальное время. Хотя подход линейного программирования обеспечивает гарантию асимптотической сложности, на практике часто бывает выгоднее просто использовать итерацию по критерию. В алгоритме 7.10 показана реализация решения дискретной задачи методом линейного программирования.

Алгоритм 7.10. Метод решения задачи дискретной MDP с использованием линейного программирования. Для удобства задания линейной программы определим функцию преобразования MDP в тензорную форму, где состояния и действия состоят из целых индексов, функция вознаграждения – матрица, а функция перехода – трехмерный тензор. Она использует пакет `JuMP.jl` для математического программирования. В качестве оптимизатора применяется `GLPK.jl`, но вместо него можно использовать и другие оптимизаторы. Чтобы использовать этот метод, мы также определяем поведение решения по умолчанию для MDP

```
struct LinearProgramFormulation end

function tensorform(𝒫::MDP)
    𝒮, 𝒜, R, T = 𝒫.𝒮, 𝒫.𝒜, 𝒫.R, 𝒫.T
    𝒮′ = eachindex(𝒮)
    𝒜′ = eachindex(𝒜)
```

---

[17] Это доказал Л. Г. Хачиян, см.: Полиномиальные алгоритмы в линейном программировании // Журнал вычислительной математики и математической физики СССР. Т. 1. С. 20; № 1. С. 53–72, 1980. На практике современные алгоритмы, как правило, более эффективны.

```
    R′ = [R(s,a) for s in 𝒮, a in 𝒜]
    T′ = [T(s,a,s′) for s in 𝒮, a in 𝒜, s′ in 𝒮]
    return 𝒮′, 𝒜′, R′, T′
end

solve(𝒫::MDP) = solve(LinearProgramFormulation(), 𝒫)

function solve(M::LinearProgramFormulation, 𝒫::MDP)
    𝒮, 𝒜, R, T = tensorform(𝒫)
    model = Model(GLPK.Optimizer)
    @variable(model, U[𝒮])
    @objective(model, Min, sum(U))
    @constraint(model, [s=𝒮,a=𝒜], U[s] ≥ R[s,a] + 𝒫.γ*T[s,a,:]⋅U)
    optimize!(model)
    return ValueFunctionPolicy(𝒫, value.(U))
end
```

7.8. Линейные системы с квадратичным вознаграждением

До сих пор мы предполагали дискретные пространства состояний и действий. В данном разделе мы ослабим это предположение, позволив использовать непрерывные векторные состояния и действия. Уравнение оптимальности Беллмана для дискретных задач можно преобразовать следующим образом[18]:

$$U_{h+1}(\mathbf{s}) = \max_{\mathbf{a}} \left(R(\mathbf{s}, \mathbf{a}) + \int T(\mathbf{s}' | \mathbf{s}, \mathbf{a}) U_h(\mathbf{s}') d\mathbf{s}' \right), \tag{7.21}$$

где s и a в уравнении (7.16) заменены их векторными эквивалентами, сумма заменена интегралом, а T дает плотность, а не массу вероятности. Уравнение (7.21) для произвольного непрерывного распределения переходов и функции вознаграждения является вычислительно сложным.

В некоторых случаях для MDP с непрерывным пространством состояний и действий существуют точные методы решения[19]. В частности, если задача имеет *линейную динамику* и *квадратичное вознаграждение*, то оптимальная стратегия эффективно находится аналитически. Такая система известна в теории управления как *линейный квадратичный регулятор* (linear quadratic regulator, LQR) и хорошо изучена[20].

---

[18] В этом разделе предполагается, что задача не дисконтирована и имеет конечный горизонт, но эти уравнения можно легко обобщить.

[19] Подробный обзор см. в главе 4 тома 1 книги D. P. Bertsekas, *Dynamic Programming and Optimal Control*. Athena Scientific, 2007.

[20] Краткое изложение LQR и других связанных задач управления см. в A. Shaiju, I. R. Petersen, *Formulas for Discrete Time LQR, LQG, LEQG and Minimax LQG Optimal Control Problems*, IFAC Proceedings Volumes, vol. 41, no. 2, pp. 8773–8778, 2008.

Задача имеет линейную динамику, если следующее состояние $\mathbf{s}'$ после выполнения действия $\mathbf{a}$ из состояния $\mathbf{s}$ определяется уравнением вида:

$$\mathbf{s}' = \mathbf{T}_s\mathbf{s} + \mathbf{T}_a\mathbf{a} + \mathbf{w}, \tag{7.22}$$

где $\mathbf{T}_s$ и $\mathbf{T}_a$ – матрицы, а $\mathbf{w}$ – случайное возмущение, взятое из распределения с конечной дисперсией с нулевым средним значением, которое не зависит от $\mathbf{s}$ и $\mathbf{a}$. Одним из распространенных вариантов является многомерное гауссово распределение.

Функция вознаграждения является квадратичной, если ее можно записать в следующем виде[21]:

$$R(\mathbf{s}, \mathbf{a}) = \mathbf{s}^\top\mathbf{R}_s\mathbf{s} + \mathbf{a}^\top\mathbf{R}_a\mathbf{a}, \tag{7.23}$$

где $\mathbf{R}_s$ и $\mathbf{R}_a$ – матрицы, которые определяют, как комбинации компонентов состояния и действия способствуют вознаграждению. Дополнительно потребуем, чтобы $\mathbf{R}_s$ была отрицательно полуопределенной, а $\mathbf{R}_a$ – отрицательно определенной. Такая функция вознаграждения *наказывает* состояния и действия, которые отклоняются от 0.

Задачи с линейной динамикой и квадратичным вознаграждением обычно встречаются в теории управления, где часто пытаются регулировать процесс таким образом, чтобы он не отклонялся далеко от желаемого значения. Квадратичная стоимость назначает гораздо более высокую стоимость состояниям, удаленным от исходного состояния, чем тем, которые находятся рядом с ним. Оптимальная стратегия для задачи с линейной динамикой и квадратичным вознаграждением представляет собой аналитическое решение. Многие задачи MDP можно аппроксимировать линейно-квадратичными MDP и решить аналитически, что часто приводит к разумной стратегии для исходной задачи.

Подстановка функций перехода и вознаграждения в уравнение (7.21) дает

$$U_{h+1}(\mathbf{s}) = \max_{\mathbf{a}}\left(\mathbf{s}^\top\mathbf{R}_s\mathbf{s} + \mathbf{a}^\top\mathbf{R}_a\mathbf{a} + \int p(\mathbf{w})U_h(\mathbf{T}_s\mathbf{s} + \mathbf{T}_a\mathbf{a} + \mathbf{w})\mathrm{d}\mathbf{w}\right), \tag{7.24}$$

где $p(\mathbf{w})$ – плотность вероятности случайного возмущения с нулевым средним значением $\mathbf{w}$.

Оптимальная одношаговая функция полезности имеет вид:

$$U_1(\mathbf{s}) = \max_{\mathbf{a}}\left(\mathbf{s}^\top\mathbf{R}_s\mathbf{s} + \mathbf{a}^\top\mathbf{R}_a\mathbf{a}\right) = \mathbf{s}^\top\mathbf{R}_s\mathbf{s}, \tag{7.25}$$

где оптимальное действие $\mathbf{a} = 0$.

По индукции покажем, что $U_h(\mathbf{s})$ имеет квадратичную форму $\mathbf{s}^\top\mathbf{V}_h\mathbf{s} + q_h$ с симметричными матрицами $\mathbf{V}_h$. Для одношаговой функции полезности $\mathbf{V}_1 = \mathbf{R}_s$ и $q_1 = 0$. Подстановка этой квадратичной формы в уравнение (7.24) дает

---

[21] Также можно включить третий член, $2\mathbf{s}^\top\mathbf{R}_{sa}\mathbf{a}$. Пример см. в Shaiju and Petersen (2008).

$$U_{h+1}(\mathbf{s}) = \mathbf{s}^\top \mathbf{R}_s \mathbf{s} + \max_{\mathbf{a}} \left(\mathbf{a}^\top \mathbf{R}_a \mathbf{a} + \int p(\mathbf{w}) (\mathbf{T}_s \mathbf{s} + \mathbf{T}_a \mathbf{a} + \mathbf{w})^\top \mathbf{V}_h (\mathbf{T}_s \mathbf{s} + \mathbf{T}_a \mathbf{a} + \mathbf{w}) + q_h \right). \quad (7.26)$$

Это уравнение можно упростить, развернув и воспользовавшись тем, что $\int p(\mathbf{w}) \mathrm{d}\mathbf{w} = 1$ и $\int \mathbf{w} p(\mathbf{w}) \mathrm{d}\mathbf{w} = 0$:

$$\begin{aligned} U_{h+1}(\mathbf{s}) = {} & \mathbf{s}^\top \mathbf{R}_s \mathbf{s} + \mathbf{s}^\top \mathbf{T}_s^\top \mathbf{V}_h \mathbf{T}_s \mathbf{S} \\ & + \max_{\mathbf{a}} \left(\mathbf{a}^\top \mathbf{R}_a \mathbf{a} + 2\mathbf{s}^\top \mathbf{T}_s^\top \mathbf{V}_h \mathbf{T}_a \mathbf{a} + \mathbf{a}^\top \mathbf{T}_a^\top \mathbf{V}_h \mathbf{T}_a \mathbf{a} \right) \\ & + \int p(\mathbf{w}) (\mathbf{w}^\top \mathbf{V}_h \mathbf{w}) \mathrm{d}\mathbf{w} + q_h . \end{aligned} \quad (7.27)$$

Мы можем получить оптимальное действие, продифференцировав по $\mathbf{a}$ и приравняв к 0[22]:

$$\begin{aligned} 0 &= (\mathbf{R}_a + \mathbf{R}_a^\top)\mathbf{a} + 2\mathbf{T}_a^\top \mathbf{V}_h \mathbf{T}_s \mathbf{S} + \left(\mathbf{T}_a^\top \mathbf{V}_h \mathbf{T}_a + (\mathbf{T}_a^\top \mathbf{V}_h \mathbf{T}_a)^\top \right) \mathbf{a} \\ &= 2\mathbf{R}_a \mathbf{a} + 2\mathbf{T}_a^\top \mathbf{V}_h \mathbf{T}_s \mathbf{s} + 2\mathbf{T}_a^\top \mathbf{V}_h \mathbf{T}_a \mathbf{a}. \end{aligned} \quad (7.28)$$

Решение для оптимального действия дает[23]

$$\mathbf{a} = -(\mathbf{R}_a + \mathbf{T}_a^\top \mathbf{V}_h \mathbf{T}_a)^{-1} \mathbf{T}_a^\top \mathbf{V}_h \mathbf{T}_s \mathbf{s}. \quad (7.29)$$

Подстановка оптимального действия в $U_{h+1}(\mathbf{s})$ дает искомую квадратичную форму[24], где

$$\mathbf{V}_{h+1} = \mathbf{R}_s + \mathbf{T}_s^\top \mathbf{V}_h^\top \mathbf{T}_s - (\mathbf{T}_a^\top \mathbf{V}_h \mathbf{T}_s)^\top (\mathbf{R}_a + \mathbf{T}_a^\top \mathbf{V}_h \mathbf{T}_a)^{-1} (\mathbf{T}_a^\top \mathbf{V}_h \mathbf{T}_s), \quad (7.30)$$

а также

$$q_{h+1} = \sum_{i=1}^{h} \mathbb{E}_{\mathbf{w}}[\mathbf{w}^\top \mathbf{V}_i \mathbf{w}]. \quad (7.31)$$

Если $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma})$, то

$$q_{h+1} = \sum_{i=1}^{h} \mathrm{Tr}(\boldsymbol{\Sigma} \mathbf{V}_i). \quad (7.32)$$

Мы можем вычислить $\mathbf{V}_h$ и q_h до любого горизонта h, начиная с $\mathbf{V}_1 = \mathbf{R}_s$ и $q_1 = 0$ и выполняя итерации с помощью уравнений (7.30) и (7.31). Оптимальное действие для h-шаговой стратегии следует непосредственно из уравнения (7.29):

---

[22] Вспомним, что $\nabla_{\mathbf{x}} \mathbf{A}\mathbf{x} = \mathbf{A}^\top \quad \nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A}\mathbf{x} = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x}$.

[23] Матрица $\mathbf{R}_a + \mathbf{T}_a^\top \mathbf{V}_h \mathbf{T}_a$ отрицательно определена и, следовательно, обратима.

[24] Это уравнение иногда называют *уравнением Риккати с дискретным временем*, названным в честь венецианского математика Якопо Риккати (1676–1754).

$$\pi_h(\mathbf{s}) = -(\mathbf{T}_a^\top \mathbf{V}_{h-1} \mathbf{T}_a + \mathbf{R}_a)^{-1} \mathbf{T}_a^\top \mathbf{V}_{h-1} \mathbf{T}_s \mathbf{s}. \tag{7.33}$$

Обратите внимание, что оптимальное действие не зависит от распределения возмущения с нулевым средним[25]. Однако дисперсия возмущения влияет на ожидаемую полезность. Реализация метода представлена в алгоритме 7.11. Пример 7.4 демонстрирует этот процесс на простой задаче с линейной гауссовой динамикой.

Алгоритм 7.11. Метод, который вычисляет оптимальную стратегию для горизонта MDP с количеством шагов `h_max` со стохастической линейной динамикой, параметризованной матрицами `Ts` и `Ta`, и квадратичным вознаграждением, параметризованным матрицами `Rs` и `Ra`. Метод возвращает вектор стратегий, где запись h производит оптимальное первое действие в h-шаговой стратегии

```
struct LinearQuadraticProblem
    Ts # матрица перехода с учетом состояния
    Ta # матрица перехода с учетом действия
    Rs # матрица вознаграждения с учетом состояния (отрицательно полуопределенная)
    Ra # матрица вознаграждения с учетом действия (отрицательно определенная)
    h_max # horizon
end

function solve(𝒫::LinearQuadraticProblem)
    Ts, Ta, Rs, Ra, h_max = 𝒫.Ts, 𝒫.Ta, 𝒫.Rs, 𝒫.Ra, 𝒫.h_max
    V = zeros(size(Rs))
    ns = Any[s -> zeros(size(Ta, 2))]
    for h in 2:h_max
        V = Ts'*(V - V*Ta*((Ta'*V*Ta + Ra) \ Ta'*V))*Ts + Rs
        L = -(Ta'*V*Ta + Ra) \ Ta' * V * Ts
        push!(ns, s -> L*s)
    end
    return ns
end
```

Пример 7.4. Решение MDP с конечным горизонтом с линейной функцией перехода и квадратичным вознаграждением. На рисунке ниже показано развитие системы от [–10, 0]. Синие контурные линии показывают распределение Гаусса по состояниям на каждой итерации. Начальное убеждение круглое, но оно искажается до овальной формы, когда мы распространяем его вперед с помощью фильтра Калмана

Рассмотрим задачу непрерывного MDP, где состояние состоит из скалярной позиции и скорости $s = [x, v]$. Действия представляют собой скалярные ускорения a, каждое из которых выполняется в течение временного шага $\Delta t = 1$.

[25] В этом случае мы можем заменить случайные возмущения их ожидаемым значением без изменения оптимальной стратегии. Это свойство известно как *эквивалентность достоверности* (certainty equivalence).

Найдем оптимальную пятишаговую стратегию из $s_0 = [-10, 0]$ с учетом квадратичного вознаграждения:

$R(\mathbf{s}, a) = -x^2 - v^2 - 0.5a^2$,

так что система стремится к покою при $\mathbf{s} = \mathbf{0}$.

Динамика перехода может быть представлена матрицами

$$\begin{bmatrix} x' \\ v' \end{bmatrix} = \begin{bmatrix} x + v\Delta t + \frac{1}{2}a\Delta t^2 + w_1 \\ v + a\Delta t + w_2 \end{bmatrix} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} + \begin{bmatrix} 0.5\Delta t^2 \\ \Delta t \end{bmatrix} [a] + \mathbf{w},$$

где $\mathbf{w}$ взято из многомерного распределения Гаусса с нулевым средним и ковариацией $0{,}1\mathbf{I}$.

Матрицы вознаграждения: $\mathbf{R}_s = -\mathbf{I}$ и $\mathbf{R}_a = -[0{,}5]$.

Результирующие оптимальные стратегии таковы:

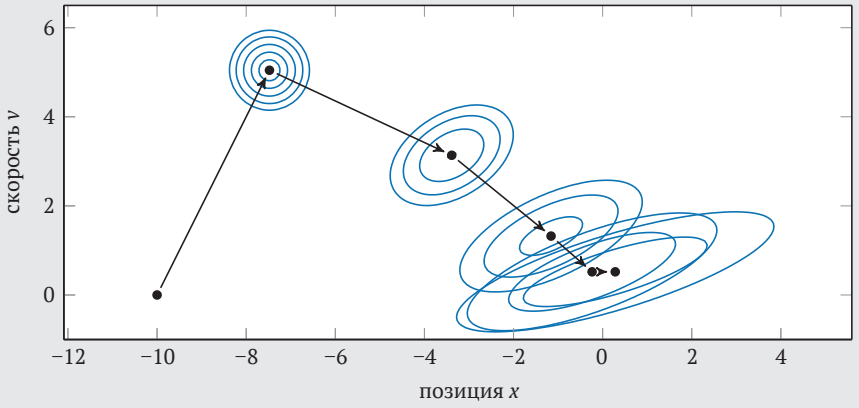
$\pi_1(\mathbf{s})$ =[0 0] $\mathbf{s}$;

$\pi_2(\mathbf{s})$ =[−0,286 −0,857] $\mathbf{s}$;

$\pi_3(\mathbf{s})$ =[−0,462 −1,077] $\mathbf{s}$;

$\pi_4(\mathbf{s})$ =[−0,499 −1,118] $\mathbf{s}$;

$\pi_5(\mathbf{s})$ =[−0,504 −1,124] $\mathbf{s}$.



7.9. Заключение

- Точное решение задачи дискретных MDP с ограниченным вознаграждением можно найти с помощью динамического программирования.
- Точная оценка стратегии для таких задач может быть выполнена посредством обращения матрицы или аппроксимирована с помощью итеративного алгоритма.

- Итерацию по стратегиям можно использовать для поиска оптимальных стратегий путем многократных итеративных переходов между оценкой стратегии и ее улучшением.
- Итерация по критериям и асинхронная итерация по критериям экономят вычисления за счет прямой итерации функции полезности.
- Задача поиска оптимальной стратегии может быть оформлена в виде линейной программы и решена за полиномиальное время.
- Непрерывные задачи с линейными функциями перехода и квадратичными вознаграждениями могут иметь точное решение.

7.10. Упражнения

Упражнение 7.1. Покажите, что для бесконечной последовательности постоянных вознаграждений ($r_t = r$ для всех t) дисконтированная доходность на бесконечном горизонте сходится к $r/(1 - \gamma)$.

Решение. Мы можем доказать, что бесконечная последовательность дисконтированных постоянных вознаграждений сходится к $r/(1 - \gamma)$ за следующие шаги:

$$\sum_{t=1}^{\infty} \gamma^{t-1} r_t = r + \gamma^1 r + \gamma^2 r + \cdots = r + \gamma \sum_{t=1}^{\infty} \gamma^{t-1} r_t.$$

Перенесем суммирование в левую часть и вынесем за знак суммы $(1 - \gamma)$:

$$(1-\gamma)\sum_{t=1}^{\infty} \gamma^{t-1} r = r;$$

$$\sum_{t=1}^{\infty} \gamma^{t-1} r = \frac{r}{1-\gamma}.$$

Упражнение 7.2. Предположим, у нас есть MDP, состоящий из пяти состояний $s_{1:5}$ и двух действий: остаться (a_S) и продолжить (a_C). У нас также есть следующие условия:

$T(s_i|s_i, a_S) = 1$ для $i \in \{1, 2, 3, 4\}$;

$T(s_{i+1}|s_i, a_C) = 1$ для $i \in \{1, 2, 3, 4\}$;

$T(s_5|s_5, a) = 1$ для всех действий a;

$R(s_i, a) = 0$ для $i \in \{1, 2, 3, 5\}$ и для всех действий a;

$R(s_4, a_S) = 0$;

$R(s_4, a_C) = 10$.

Чему равен коэффициент дисконтирования γ, если оптимальное значение $U^*(s_1) = 1$?

Решение. Оптимальное значение $U^*(s_1)$ связано со следованием оптимальной стратегии n^*, начиная с s_1. При заданной модели перехода оптимальная стратегия с началом в s_1 состоит в том, чтобы продолжаться до достижения s_5, что является конечным состоянием, когда мы больше не можем переходить в другое состояние или накапливать дополнительное вознаграждение. Таким образом, оптимальное значение s_1 можно вычислить как

$$U^*(s_1) = \sum_{t=1}^{\infty} \gamma^{t-1} r_t;$$

$$U^*(s_1) = R(s_1, a_C) + \gamma^1 R(s_2, a_C) + \gamma^2 R(s_3, a_C) + \gamma^3 R(s_4, a_C) + \gamma^4 R(s_5, a_C) + \cdots;$$

$$U^*(s_1) = 0 + \gamma^1 \times 0 + \gamma^2 \times 0 + \gamma^3 \times 10 + \gamma^4 \times 0 + 0;$$

$$1 = 10\gamma^3.$$

Следовательно, коэффициент дисконтирования $\gamma = 0{,}1^{1/3} \approx 0{,}464$.

Упражнение 7.3. Какова временна́я сложность выполнения k шагов итеративной оценки стратегии?

Решение. Для итеративной оценки стратегии необходимо вычислить уравнение следующего шага:

$$U^{\pi}_{k+1}(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s' | s, \pi(s)) U^{\pi}_k(s').$$

Обновление значения в одном состоянии требует суммирования всех $|\mathcal{S}|$ состояний. Для одной итерации по всем состояниям мы должны выполнить эту операцию $|\mathcal{S}|$ раз. Таким образом, временна́я сложность k шагов итеративной оценки стратегии составляет $O(k|\mathcal{S}|^2)$.

Упражнение 7.4. Предположим, что у нас есть MDP с шестью состояниями $s_{1:6}$ и четырьмя действиями $a_{1:4}$. Используя следующую табличную форму функции полезности действия $Q(s, a)$, вычислите $U(s)$, $\pi(s)$ и $A(s, a)$.

$Q(s, a)$	a_1	a_2	a_3	a_4
s_1	0.41	0.46	0.37	0.37
s_2	0.50	0.55	0.46	0.37
s_3	0.60	0.50	0.38	0.44
s_4	0.41	0.50	0.33	0.41
s_5	0.50	0.60	0.41	0.39
s_6	0.71	0.70	0.61	0.59

Решение. Мы можем вычислить $U(s)$, $\pi(s)$ и $A(s, a)$, используя следующие уравнения:

$$U(s) = \max_a Q(s,a); \qquad \pi(s) = \arg\max_a Q(s,a); \qquad A(s,a) = Q(s,a) - U(s).$$

s	$U(s)$	$\pi(s)$	$A(s, a_1)$	$A(s, a_2)$	$A(s, a_3)$	$A(s, a_4)$
s_1	0.46	a_2	−0.05	0.00	−0.09	−0.09
s_2	0.55	a_2	−0.05	0.00	−0.09	−0.18
s_3	0.60	a_1	0.00	−0.10	−0.22	−0.16
s_4	0.50	a_2	−0.09	0.00	−0.17	−0.09
s_5	0.60	a_2	−0.10	0.00	−0.19	−0.21
s_6	0.71	a_1	0.00	-0.01	−0.10	−0.12

Упражнение 7.5. Предположим, что у нас есть прямолинейный гексамир с тремя плитками (приложение F.1), где крайняя правая плитка является поглощающим состоянием. Когда мы совершаем какое-либо действие в крайнем правом состоянии, мы получаем вознаграждение в размере 10 и переносимся в четвертое конечное состояние, где мы больше не получаем никакого вознаграждения. Используйте коэффициент дисконтирования $\gamma = 0.9$ и выполните один шаг итерации по стратегиям, где исходная стратегия n заставляет нас двигаться на восток в первой плитке, на северо-восток во второй плитке и на юго-запад в третьей плитке. Для этапа оценки стратегии составьте матрицу перехода $\mathbf{T}^\pi$ и вектор вознаграждения $\mathbf{R}^\pi$, а затем найдите функцию полезности бесконечного горизонта $\mathbf{U}^\pi$ напрямую, используя инверсию матрицы. Для шага улучшения стратегии вычислите обновленную стратегию π', максимизируя уравнение следующего шага.

Решение. На этапе оценки стратегии воспользуемся уравнением (7.10), которое для удобства повторим здесь:

$$\mathbf{U}^\pi = (\mathbf{I} - \gamma\mathbf{T}^\pi)^{-1}\mathbf{R}^\pi.$$

Формируя матрицу перехода $\mathbf{T}^\pi$ и вектор вознаграждения $\mathbf{R}^\pi$ с дополнительным состоянием для конечного состояния, мы можем решить для функции значения бесконечного горизонта $\mathbf{U}^\pi$[26]:

$$\mathbf{U}^\pi = \left(\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix} - (0.9)\begin{bmatrix}0.3&0.7&0&0\\0&0.85&0.15&0\\0&0&0&1\\0&0&0&1\end{bmatrix}\right)^{-1}\begin{bmatrix}-0.3\\-0.85\\10\\0\end{bmatrix} \approx \begin{bmatrix}1.425\\2.128\\10\\0\end{bmatrix}.$$

Для этапа улучшения стратегии мы применяем уравнение (7.11), используя обновленную функцию полезности. Действия в операторе arg max соответствуют a_E, a_{NE}, a_{NW}, a_W, a_{SW} и a_{SE}:

---

[26] Задача гексамира определяет $R(s, a, s')$, поэтому для получения записей для $\mathbf{R}^\pi$ мы должны вычислить $R(s,a) = \sum_{s'} T(s'|s,a)R(s,a,s')$.
Например, −0.3 получается из 30%-ной вероятности того, что движение на восток вызовет столкновение с границей со стоимостью −1.

$\pi(s_1) = \arg\max(1.425, 0.527, 0.283, 0.283, 0.283, 0.527) = a_E$;

$\pi(s_2) = \arg\max(6.575, 2.128, 0.970, 1.172, 0.970, 2.128) = a_E$;

$\pi(s_3) = \arg\max(10, 10, 10, 10, 10, 10)$ (все действия одинаково желательны).

Упражнение 7.6. Выполните два шага итерации значения для задачи, представленной в упражнении 7.5, начиная с функции начального значения $U_0(s) = 0$ для всех s.

Решение. Для итеративного обновления функции полезности нам нужно использовать оператор Беллмана (уравнение (7.16)). Действия в операторе max соответствуют a_E, a_{NE}, a_{NW}, a_W, a_{SW} и a_{SE}. При первой итерации функция полезности равна нулю для всех состояний, поэтому нам нужно учитывать только компонент вознаграждения:

$U_1(s_1) = \max(-0.3, -0.85, -1, -1, -1, -0.85) = -0.3$;

$U_1(s_2) = \max(-0.3, -0.85, -0.85, -0.3, -0.85, -0.85) = -0.3$;

$U_1(s_3) = \max(10, 10, 10, 10, 10, 10) = 10$.

Для второй итерации

$U_2(s_1) = \max(-0.57, -1.12, -1.27, -1.27, -1.27, -1.12) = -0.57$;

$U_2(s_2) = \max(5.919, 0.271, -1.12, -0.57, -1.12, 0.271) = 5.919$;

$U_2(s_3) = \max(10, 10, 10, 10, 10, 10) = 10$.

Упражнение 7.7. Примените один проход асинхронной итерации по критериям к задаче в упражнении 7.5, начиная с функции полезности $U_0(s) = 0$ для всех s. Обновите состояния справа налево.

Решение. Будем использовать оператор Беллмана (уравнение (7.16)) для итеративного обновления функции полезности для каждого состояния в соответствии с заданным порядком. Действия в операторе max() соответствуют a_E, a_{NE}, a_{NW}, a_W, a_{SW} и a_{SE}:

$U(s_3) = \max(10, 10, 10, 10, 10, 10) = 10$;

$U(s_2) = \max(6, 0.5, -0.85, -0.3, -0.85, 0.5) = 6$;

$U(s_1) = \max(3.48, -0.04, -1, -1, -1, -0.04) = 3.48$.

Упражнение 7.8. Докажите, что невязка Беллмана, равная δ, гарантирует, что функция полезности, полученная итерацией по критерию, находится в пределах $\delta\gamma/(1 - \gamma)$ от $U^*(s)$ в каждом состоянии s.

Решение. Предположим, что для данного U_k мы знаем, что $||U_k - U_{k-1}||_\infty < \delta$. Затем мы ограничиваем улучшение на следующей итерации:

$$\begin{aligned}U_{k+1}(s)-U_k(s) &= \max_a\left(R(s,a)+\gamma\sum_{s'}T(s'|s,a)U_k(s')\right)\\ &\quad-\max_a\left(R(s,a)+\gamma\sum_{s'}T(s'|s,a)U_{k-1}(s')\right)\\ &< \max_a\left(R(s,a)+\gamma\sum_{s'}T(s'|s,a)U_k(s')\right)\\ &\quad-\max_a\left(R(s,a)+\gamma\sum_{s'}T(s'|s,a)U_k(s')-\delta\right)\\ &= \delta\gamma.\end{aligned}$$

Сходным образом

$$\begin{aligned}U_{k+1}(s)-U_k(s) &> \max_a\left(R(s,a)+\gamma\sum_{s'}T(s'|s,a)U_k(s')\right)\\ &\quad-\max_a\left(R(s,a)+\gamma\sum_{s'}T(s'|s,a)(U_{k-1}(s')+\delta)\right)\\ &= -\delta\gamma.\end{aligned}$$

Таким образом, накопленное улучшение после бесконечных итераций ограничено

$$\|U^*(s)-U_k(s)\|_\infty < \sum_{i=1}^{\infty}\delta\gamma^i = \frac{\delta\gamma}{1-\gamma}.$$

Невязка Беллмана, равная δ, таким образом, гарантирует, что оптимальная функция полезности, полученная итерацией по критерию, находится в пределах $\delta\gamma/(1-\gamma)$ от U^*.

Упражнение 7.9. Предположим, что мы выполняем оценку экспертной стратегии, чтобы получить функцию полезности. Если жадные действия по отношению к функции полезности эквивалентны экспертной стратегии, то какой вывод мы можем сделать об экспертной стратегии?

Решение. Из уравнения оптимальности Беллмана мы знаем, что жадный алгоритм поиска оптимальной функции полезности является стационарным. Если жадная стратегия соответствует экспертной стратегии, то жадная стратегия оптимальна.

Упражнение 7.10. Переформулируйте задачу LQR с квадратичной функцией вознаграждения $R(\mathbf{s}, \mathbf{a}) = \mathbf{s}^\top\mathbf{R}_s\mathbf{s} + \mathbf{a}^\top\mathbf{R}_a\mathbf{a}$ таким образом, чтобы функция вознаграждения включала линейные члены по **s** и **a**.

Решение. Мы можем ввести дополнительное измерение состояния, которое всегда равно 1, что даст новую систему с линейной динамикой:

$$\begin{bmatrix} \mathbf{s}' \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{T}_s & \mathbf{0} \\ \mathbf{0}^\top & 1 \end{bmatrix} \begin{bmatrix} \mathbf{s} \\ 1 \end{bmatrix} + \mathbf{T}_a \mathbf{a}.$$

Функция вознаграждения дополненной системы теперь может иметь линейные компоненты вознаграждения за состояние:

$$\begin{bmatrix} \mathbf{s} \\ 1 \end{bmatrix}^\top \mathbf{R}_{\text{дополн.}} \begin{bmatrix} \mathbf{s} \\ 1 \end{bmatrix} = \mathbf{s}^\top \mathbf{R}_s \mathbf{s} + 2\mathbf{r}_{s,\text{линейн.}}^\top \mathbf{s} + r_{s,\text{скаляр.}}.$$

Точно так же мы можем дополнить систему измерением действия, которое всегда равно 1, чтобы получить линейные компоненты вознаграждения за действие.

Упражнение 7.11. Почему оптимальная стратегия, полученная в примере 7.4, производит действия с бóльшим размахом отклонений, когда горизонт больше?

Решение. Задача в примере 7.4 имеет квадратичное вознаграждение, которое наказывает отклонения от исходной точки. Чем дальше горизонт, тем большее отрицательное вознаграждение может быть накоплено, что делает более выгодным достижение источника раньше.

Упражнение 7.12. Докажите, что итерационная оценка стратегии сходится к решению уравнения (7.6).

Решение. Рассмотрим итеративную оценку, примененную к стратегии π, как указано в уравнении (7.5):

$$U_{k+1}^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s'|s, \pi(s) U_k^\pi(s').$$

Определим оператор B_π и перепишем его как $U_{k+1}^\pi = B_\pi U_k^\pi$. Мы можем показать, что B_π является сжимающим отображением:

$$\begin{aligned} B_\pi U^\pi(s) &= R(s, \pi(s)) + \gamma \sum_{s'} T(s'|s, \pi(s)) U^\pi(s') \\ &= R(s, \pi(s)) + \gamma \sum_{s'} T(s'|s, \pi(s)) (U^\pi(s') - \hat{U}^\pi(s') + \hat{U}^\pi(s')) \\ &= B_\pi \hat{U}^\pi(s) + \gamma \sum_{s'} T(s'|s, \pi(s)) (U^\pi(s') - \hat{U}^\pi(s')) \\ &\le B_\pi \hat{U}^\pi(s) + \gamma ||U^\pi - \hat{U}^\pi||_\infty. \end{aligned}$$

Следовательно, $||B_\pi U^\pi - B_\pi \hat{U}^\pi||_\infty \le \alpha ||U^\pi - \hat{U}^\pi||_\infty$ при $\alpha = \gamma$, откуда следует, что B_π – сжимающее отображение. Как обсуждалось в приложении А.15, $\lim_{t \to \infty} B_\pi^t U_1^\pi$ сходится к единственной неподвижной точке U^π, для которой $U^\pi = B_\pi U^\pi$.

Упражнение 7.13. Докажите, что итерация по критерию сходится к единственному решению.

Решение. Обновление полезности согласно уравнению (7.16) равно

$$U^{k+1}(s) = \max_a \left(R(s,a) + \gamma \sum_{s'} T(s'|s,a) U_k(s') \right).$$

Обозначим оператор Беллмана за B и перепишем применение оператора Беллмана как $U_{k+1} = BU_k$. Как и в предыдущей задаче, если B является сжимающим отображением, то повторное применение B к U будет сходиться к единственной фиксированной точке.

Мы можем показать, что B является сжимающим отображением:

$$\begin{aligned} BU(s) &= \max_a \left(R(s,a) + \gamma \sum_{s'} T(s'|s,a) U(s') \right) \\ &= \max_a \left(R(s,a) + \gamma \sum_{s'} T(s'|s,a)(U(s') - \hat{U}(s') + \hat{U}(s')) \right) \\ &\leq B\hat{U}(s) + \gamma \max_a \sum_{s'} T(s'|s,a)(U(s') - \hat{U}(s')) \\ &\leq B\hat{U}(s) + \alpha ||U - \hat{U}||_\infty, \end{aligned}$$

где $\alpha = \gamma \max_s \max_a \sum_{s'} T(s'|s,a)$, при $0 \leqslant \alpha < 1$. Следовательно, $\|BU - B\hat{U}\|_\infty \leq \alpha \|U - \hat{U}\|_\infty$, откуда следует, что B – сжимающее отображение.

Упражнение 7.14. Покажите, что точка, к которой сходится итерация по критерию, соответствует оптимальной функции полезности.

Решение. Пусть U будет функцией полезности, полученной в результате итерации по критерию. Мы хотим показать, что $U = U^*$. При сходимости имеем $BU = U$. Пусть U_0 – функция полезности, которая отображает все состояния в 0. Для любой стратегии π из определения B_π следует, что $B_\pi U_0 \leqslant BU_0$. Аналогично $B_\pi^t U_0 \leqslant B^t U_0$. Поскольку $B_{\pi^*}^t U_0 \to U^*$ и $B^t U_0 \to U$ при $t \to \infty$, отсюда следует $U^* \leqslant U$, что может иметь место только при $U = U^*$.

Упражнение 7.15. Предположим, что нам дана линейная задача Гаусса с возмущением $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma})$ и квадратичным вознаграждением. Покажите, что скалярный член функции полезности имеет вид:

$$q_{h+1} = \sum_{i=1}^{h} \mathbb{E}_{\mathbf{w}}[\mathbf{w}^\top \mathbf{V}_i \mathbf{w}] = \sum_{i=1}^{h} \mathrm{Tr}(\boldsymbol{\Sigma} \mathbf{V}_i).$$

Вы можете использовать *прием с эквивалентностью следа* (trace trick):

$$\mathbf{x}^\top \mathbf{A} \mathbf{x} = \mathrm{Tr}(\mathbf{x}^\top \mathbf{A} \mathbf{x}) = \mathrm{Tr}(\mathbf{A} \mathbf{x} \mathbf{x}^\top).$$

Решение. Уравнение в условии верно, если $\mathbb{E}_{\mathbf{w}}[\mathbf{w}^\top \mathbf{V}_i \mathbf{w}] = \mathrm{Tr}(\boldsymbol{\Sigma}\mathbf{V}_i)$. Докажем это следующим образом:

$$\begin{aligned}
\mathop{\mathbb{E}}_{\mathbf{w}\sim\mathcal{N}(\mathbf{0},\boldsymbol{\Sigma})}[\mathbf{w}^\top \mathbf{V}_i \mathbf{w}] &= \mathop{\mathbb{E}}_{\mathbf{w}\sim\mathcal{N}(\mathbf{0},\boldsymbol{\Sigma})}[\mathrm{Tr}(\mathbf{w}^\top \mathbf{V}_i \mathbf{w})] \\
&= \mathop{\mathbb{E}}_{\mathbf{w}\sim\mathcal{N}(\mathbf{0},\boldsymbol{\Sigma})}[\mathrm{Tr}(\mathbf{V}_i \mathbf{w}\mathbf{w}^\top)] \\
&= \mathrm{Tr}\left(\mathop{\mathbb{E}}_{\mathbf{w}\sim\mathcal{N}(\mathbf{0},\boldsymbol{\Sigma})}[\mathbf{V}_i \mathbf{w}\mathbf{w}^\top]\right) \\
&= \mathrm{Tr}\left(\mathbf{V}_i \mathop{\mathbb{E}}_{\mathbf{w}\sim\mathcal{N}(\mathbf{0},\boldsymbol{\Sigma})}[\mathbf{w}\mathbf{w}^\top]\right) \\
&= \mathrm{Tr}(\mathbf{V}_i\boldsymbol{\Sigma}) \\
&= \mathrm{Tr}(\boldsymbol{\Sigma}\mathbf{V}_i).
\end{aligned}$$

Упражнение 7.16. Какова роль скалярного члена q в оптимальной функции полезности LQR, представленной в уравнении (7.31)?

$$q_{h+1} = \sum_{i=1}^{h} \mathbb{E}_{\mathbf{w}}[\mathbf{w}^\top \mathbf{V}_i \mathbf{w}].$$

Решение. Матрица $\mathbf{M}$ является положительно определенной, если для всех ненулевых $\mathbf{x}$ выполняется неравенство $\mathbf{x}^\top\mathbf{M}\mathbf{x} > 0$. В уравнении (7.31) каждое $\mathbf{V}_i$ отрицательно полуопределено, поэтому $\mathbf{w}^\top\mathbf{V}\mathbf{w} \leqslant 0$ для всех $\mathbf{w}$. Таким образом, гарантируется, что эти члены q неположительны. Этого следует ожидать, так как невозможно получить положительное вознаграждение в задачах LQR, и вместо этого мы стремимся минимизировать затраты.

Скаляры q представляют собой смещения в квадратичной оптимальной функции полезности:

$$U(\mathbf{s}) = \mathbf{s}^\top\mathbf{V}\mathbf{s} + q.$$

Каждый q представляет базисное вознаграждение, вокруг которого колеблется член $\mathbf{s}^\top\mathbf{V}\mathbf{s}$. Мы знаем, что $\mathbf{V}$ отрицательно определен, поэтому $\mathbf{s}^\top\mathbf{V}\mathbf{s} \leqslant 0$, и, таким образом, q представляет собой ожидаемое вознаграждение, которое можно было бы получить, если бы вы находились в начальной точке $\mathbf{s} = 0$.

8 Приближенное вычисление функции полезности

До сих пор мы предполагали, что функция полезности представлена в виде таблицы. Но табличное представление применимо только для небольших дискретных задач. Задачи с большими пространствами состояний могут занимать нереально большой объем памяти, а точные методы, рассмотренные в предыдущей главе, могут потребовать практически не осуществимого объема вычислений. Для решения таких задач часто приходится прибегать к *приближенному динамическому программированию*[1]. Одним из способов аппроксимации решений, который мы рассмотрим в этой главе, является использование *аппроксимации функции полезности*. Мы обсудим различные подходы к аппроксимации функции полезности и способы использования динамического программирования для получения приближенно оптимальных стратегий.

8.1. Параметрические представления

Далее мы будем использовать $U_{\boldsymbol{\theta}}(s)$ для обозначения нашего *параметрического представления* функции полезности, где $\boldsymbol{\theta}$ – *вектор параметров*. Существует множество способов представления $U_{\boldsymbol{\theta}}(s)$, некоторые из них будут упомянуты далее в этой главе. Предполагая, что у нас есть такое приближение, мы можем найти действие π в соответствии с выражением

$$\pi(s) = \arg\max_a \left(R(s,a) + \gamma \sum_{s'} T(s'|s,a) U_{\boldsymbol{\theta}}(s') \right). \tag{8.1}$$

Различные аппроксимации функции полезности часто используются в задачах с непрерывными пространствами состояний, и в этом случае сумму в уравнении (8.1) заменяет интеграл. Интеграл можно аппроксимировать, используя выборки переходной модели.

Альтернативой уравнению (8.1) является аппроксимация функции полезности действия $Q(s, a)$. Используя $Q_{\boldsymbol{\theta}}(s, a)$ для представления нашей парамет-

---

[1] Более глубокое рассмотрение этой темы представлено в W. B. Powell, *Approximate Dynamic Programming: Solving the Curses of Dimensionality*, 2nd ed. Wiley, 2011. Соответствующие идеи могут быть позаимствованы из различных областей, как обсуждалось в W. B. Powell, *Reinforcement Learning and Stochastic Optimization*. Wiley, 2022.

рической аппроксимации, мы можем получить действие в соответствии с уравнением

$$\pi(s) = \arg\max_a Q_\theta(s, a). \tag{8.2}$$

В этой главе рассматривается применение динамического программирования к конечному набору состояний $\mathcal{S} = s_{1:m}$, чтобы получить параметрическую аппроксимацию функции полезности во всем пространстве состояний. Для создания этого набора могут применяться разные схемы. Если пространство состояний относительно низкоразмерное, мы можем определить сетку. Другой подход заключается в использовании случайной выборки из пространства состояний. Однако некоторые состояния встречаются с большей вероятностью, чем другие, и поэтому они более важны при построении функции полезности. Мы можем сместить выборку в сторону более важных состояний, выполнив симуляции с некоторой стратегией (возможно, изначально случайной) из правдоподобного набора начальных состояний.

Для улучшения нашей аппроксимации функции полезности в пространстве состояний $\mathcal{S}$ можно использовать итеративный подход. Мы поочередно улучшаем значения функции полезности в $\mathcal{S}$ с помощью динамического программирования и заново подгоняем нашу аппроксимацию в этих состояниях. Алгоритм 8.1 представляет реализацию, в которой шаг динамического программирования состоит из операторов Беллмана, как мы делали раньше при итерации по критерию (раздел 7.5). Аналогичный алгоритм может быть создан для аппроксимации значимости действия Q_θ[2].

Алгоритм 8.1. Аппроксимация итерации по критерию для MDP с аппроксимацией параметрической функции полезности `Uθ`. Мы применяем оператор Беллмана (как определено в алгоритме 7.7) в состояниях `S`, чтобы получить вектор полезности `U`. Затем вызываем метод `fit!(Uθ, S, U)`, который модифицирует параметрическое представление `Uθ`, чтобы обеспечить лучшее соответствие значений состояний в `S` полезностям в `U`. Различные параметрические аппроксимации имеют разные реализации `fit!`

```
struct ApproximateValueIteration
    Uθ    # начальные значения параметрической функции, которую реализует fit!
    S     # набор дискретных состояний для выполнения оператора Беллмана
    k_max # максимальное количество итераций
end

function solve(M::ApproximateValueIteration, 𝒫::MDP)
```

---

[2] Несколько других категорий подходов к оптимизации аппроксимаций функции полезности рассмотрены в A. Geramifard, T. J. Walsh, S. Tellex, G. Chowdhary, N. Roy, J. P. How, *A Tutorial on Linear Function Approximators for Dynamic Programming and Reinforcement Learning*, Foundations and Trends in Machine Learning, vol. 6, no. 4, pp. 375–451, 2013.

```
    Uθ, S, k_max = M.Uθ, M.S, M.k_max
    for k in 1:k_max
        U = [backup(𝒫, Uθ, s) for s in S]
        fit!(Uθ, S, U)
    end
    return ValueFunctionPolicy(𝒫, Uθ)
end
```

Все параметрические представления, обсуждаемые в этой главе, можно использовать с алгоритмом 8.1. В целом для совместимости с алгоритмом 8.1 представление должно допускать оценку $U_{\boldsymbol{\theta}}$ и подгонку $U_{\boldsymbol{\theta}}$ к ожиданиям полезности в точках набора $\mathcal{S}$.

Мы можем сгруппировать параметрические представления в две категории. К первой категории относятся методы *локальной аппроксимации*, где $\boldsymbol{\theta}$ соответствует значениям состояний в $\mathcal{S}$. Чтобы найти $U_{\boldsymbol{\theta}}(s)$ в произвольном состоянии s, мы берем взвешенную сумму полезностей, хранящихся в $\mathcal{S}$. Вторая категория включает методы *глобальной аппроксимации*, где $\boldsymbol{\theta}$ не имеет прямого отношения к значениям состояний в $\mathcal{S}$. На самом деле вектор $\boldsymbol{\theta}$ может иметь гораздо меньше или даже гораздо больше компонентов, чем состояний в $\mathcal{S}$.

Как локальную аппроксимацию, так и многие глобальные аппроксимации можно рассматривать как *аппроксимацию линейной функции* $U_{\boldsymbol{\theta}}(s) = \boldsymbol{\theta}^\top \boldsymbol{\beta}(s)$, где методы различаются тем, как они определяют векторную функцию $\boldsymbol{\beta}$. В методах локальной аппроксимации $\boldsymbol{\beta}(s)$ определяет, как взвешивать полезности состояний в $\mathcal{S}$, чтобы аппроксимировать полезность состояния s. Веса обычно неотрицательны и в сумме равны 1. Во многих методах глобальной аппроксимации $\boldsymbol{\beta}(s)$ рассматривают как набор базисных функций, которые линейно комбинируются для получения аппроксимации произвольного s.

Мы также можем аппроксимировать функцию полезности действия, используя линейную функцию $Q_{\boldsymbol{\theta}}(s, a) = \boldsymbol{\theta}^\top \boldsymbol{\beta}(s, a)$. В контексте локальных аппроксимаций мы можем обеспечить аппроксимации по непрерывным пространствам действий, выбрав конечное множество действий $A \subset \mathcal{A}$. Тогда наш вектор параметров $\boldsymbol{\theta}$ будет состоять из $|\mathcal{S}| \times |\mathcal{A}|$ компонентов, каждый из которых соответствует значению состояния-действия. Наша функция $\boldsymbol{\beta}(s, a)$ вернет вектор с тем же числом компонентов, который указывает, как взвесить наш конечный набор значений состояния-действия, чтобы получить ожидание полезности, связанное с состоянием s и действием a.

8.2. Аппроксимация по ближайшему соседу

Простой подход к локальной аппроксимации заключается в применении такого состояния в $\mathcal{S}$, которое является *ближайшим соседом* s. Чтобы использовать этот подход, нам нужна метрика расстояния (приложение A.3). Мы будем использовать запись $d(s, s')$ для обозначения расстояния между двумя состояниями s и s'. Приближенная функция полезности тогда имеет вид $U_{\boldsymbol{\theta}}(s) = \theta_i$, где

$i = \arg\min_{j \in 1:m} d(s_j, s)$. На рис. 8.1 показан пример функции полезности, представленной с использованием схемы ближайшего соседа.

Мы можем обобщить этот подход, чтобы усреднить полезности *k-ближайших соседей*. Этот подход по-прежнему приводит нас к кусочно-постоянным функциям полезности, но разные значения k могут привести к лучшим приближениям. На рис. 8.1 показаны примеры функций полезности, аппроксимированных различными значениями k. Реализация этого подхода представлена в алгоритме 8.2.

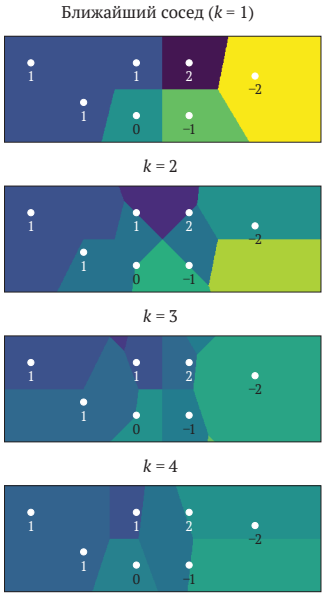


Рис. 8.1. Аппроксимация состояний в двумерном непрерывном пространстве состояний с использованием среднего значения полезности их k-ближайших соседей в соответствии с евклидовым расстоянием. Результирующая функция полезности является кусочно-постоянной

Алгоритм 8.2. Метод k-ближайших соседей, который аппроксимирует значение состояния `s` на основе `k` ближайших состояний в `S`, определяемых функцией расстояния `d`. Вектор `θ` содержит значения состояний в `S`. Большей эффективности можно добиться, используя специализированные структуры данных, такие как *kd*-деревья, реализованные в `NearestNeighbors.jl`

```
mutable struct NearestNeighborValueFunction
    k # количество соседей
    d # функция расстояния d(s, s′)
    S # набор дискретных состояний
    θ # вектор значений состояний в наборе S
end

function (U0::NearestNeighborValueFunction)(s)
    dists = [U0.d(s,s′) for s′ in U0.S]
    ind = sortperm(dists)[1:U0.k]
    return mean(U0.θ[i] for i in ind)
end
```

```
function fit!(U0::NearestNeighborValueFunction, S, U)
    U0.θ = U
    return U0
end
```

8.3. Ядерное сглаживание

Другим методом локальной аппроксимации является *ядерное сглаживание* (kernel smoothing), когда полезности состояний в $\mathcal{S}$ сглаживаются по всему пространству состояний. Этот метод требует определения *ядерной функции* (kernel function, кернфункция) $k(s, s')$, которая связывает пары состояний s и s'. Обычно нам нужно, чтобы значение $k(s, s')$ было выше для состояний, которые ближе друг к другу, потому что это подсказывает нам, как совместно взвешивать полезности, связанные с состояниями в $\mathcal{S}$. Этот метод приводит к следующему линейному приближению:

$$U_{\boldsymbol{\theta}}(s) = \sum_{i=1}^{m} \theta_i \beta_i(s) = \boldsymbol{\theta}^\top \boldsymbol{\beta}(s), \tag{8.3}$$

где

$$\beta_i(s) = \frac{k(s, s_i)}{\sum_{j=1}^{m} k(s, s_j)}. \tag{8.4}$$

Реализация метода представлена в алгоритме 8.3.

Алгоритм 8.3. Аппроксимация локально взвешенной функции полезности, определяемая ядерной функцией `k` и вектором полезностей `θ` в наборе состояний `S`

```
mutable struct LocallyWeightedValueFunction
    k # ядерная функция k(s, s′)
    S # набор дискретных состояний
    θ # вектор значений состояний в S
end

function (U0::LocallyWeightedValueFunction)(s)
    w = normalize([U0.k(s,s′) for s′ in U0.S], 1)
    return U0.θ ⋅ w
end

function fit!(U0::LocallyWeightedValueFunction, S, U)
    U0.θ = U
    return U0
end
```

Ядерную функцию можно определить разными способами. Мы можем определить ядро просто как обратное расстояние между состояниями:

$$k(s, s') = \max(d(s, s'), \varepsilon)^{-1}, \tag{8.5}$$

где ε – небольшая положительная константа, чтобы избежать деления на ноль при $s = s'$. На рис. 8.2 показаны аппроксимации с использованием нескольких функций расстояния. Видно, что ядерное сглаживание может привести к гладким аппроксимациям функции полезности, в отличие от метода k-ближайших соседей. На рис. 8.3 это ядро применяется к дискретной задаче гексамира и по-

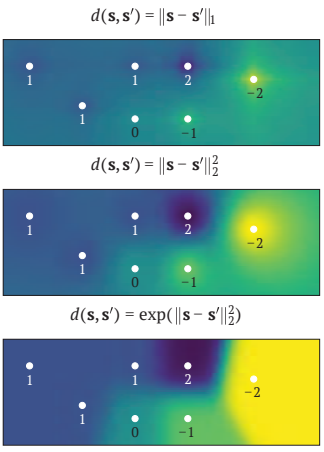


Рис. 8.2. Аппроксимация состояний в двумерном непрерывном пространстве путем присвоения значений на основе близости к нескольким состояниям с известными значениями. Аппроксимации строятся с использованием нескольких функций расстояния

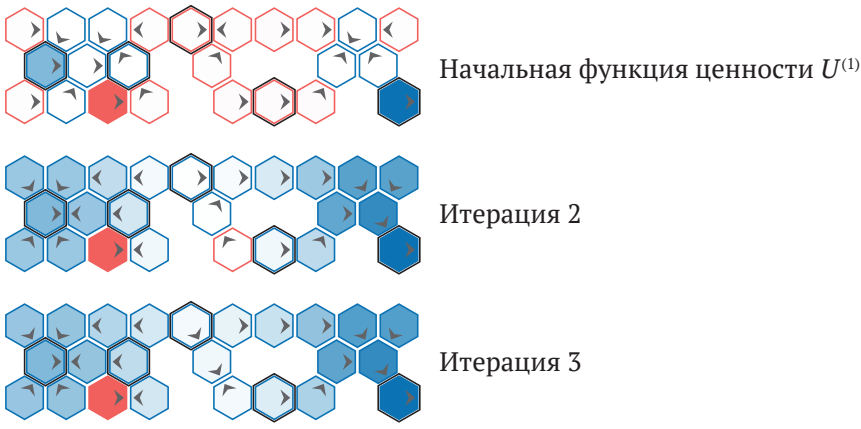


Рис. 8.3. Итерация по критерию при локальной аппроксимации, используемая для итеративного улучшения приближенной функции полезности в задаче гексамира. Пять описанных состояний используются для аппроксимации функции полезности. Значения остальных состояний аппроксимируются функцией расстояния $\|\mathbf{s} - \mathbf{s}'\|_2^2$. Итоговая стратегия разумна, но тем не менее неоптимальна. Положительное вознаграждение показано синим цветом, а отрицательное – красным

казан результат нескольких итераций приближенного значения (алгоритм 8.1). На рис. 8.4 показаны функция полезности и стратегия, полученные для *задачи горной машины* (mountain car problem, приложение F.4) с непрерывным пространством состояний.

Другим распространенным ядром является *ядро Гаусса* (Gaussian kernel):

$$k(s,s') = \exp\left(-\frac{d(s,s')^2}{2\sigma^2}\right), \tag{8.6}$$

где σ определяет степень сглаживания.

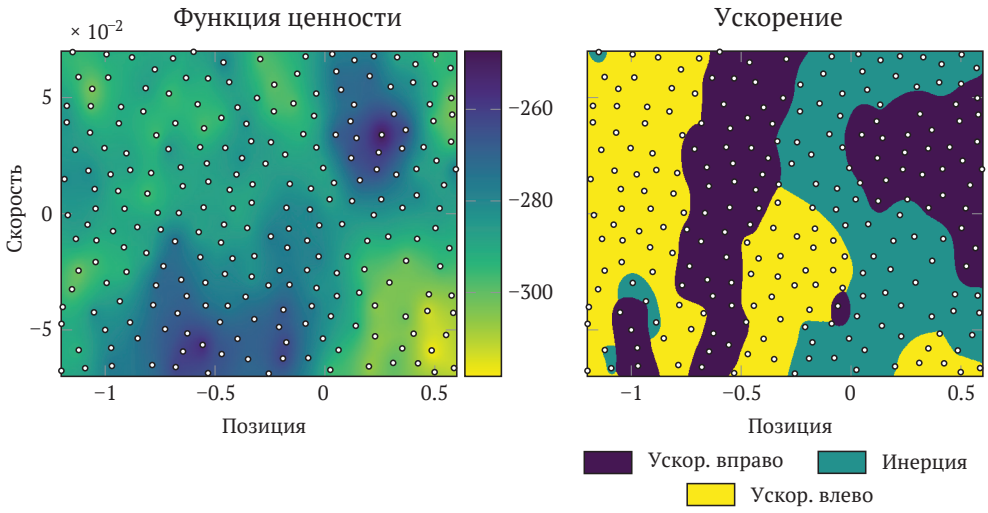


Рис. 8.4. Функция полезности и стратегия, полученные путем изучения полезности действия для конечного набора состояний (белые кружки) в задаче горной машины с использованием функции расстояния $||\mathbf{s} - \mathbf{s}'||_2 + 0.1$

8.4. Линейная интерполяция

Линейная интерполяция – еще один распространенный подход к локальной аппроксимации. Наиболее очевидным является одномерный случай, когда аппроксимированное значение для состояния s между двумя состояниями s_1 и s_2 равно

$$U_{\boldsymbol{\theta}}(s) = \alpha\theta_1 + (1 - \alpha)\theta_2, \tag{8.7}$$

где $\alpha = (s_2 - s)/(s_2 - s_1)$. Этот случай показан на рис. 8.5 и 8.6.

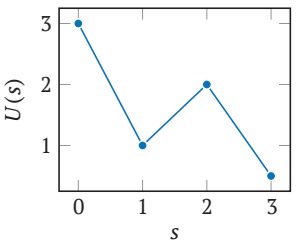


Рис. 8.5. Одномерная линейная интерполяция производит интерполированные значения вдоль сегмента линии, соединяющего две точки

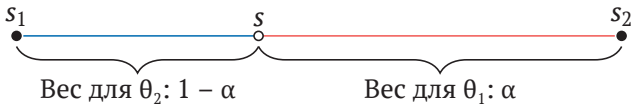
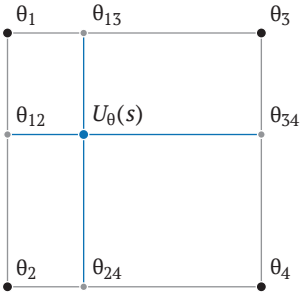


Рис. 8.6. Вес, присвоенный каждой точке в одномерном пространстве, пропорционален длине сегмента на противоположной стороне

Линейную интерполяцию можно распространить на многомерную сетку. В двумерном случае, называемом *билинейной интерполяцией*, мы интерполируем по четырем вершинам. Билинейная интерполяция выполняется посредством одномерной линейной интерполяции, по одному разу на каждой оси, что требует использования четырех состояний в вершинах сетки. Этот метод интерполяции показан на рис. 8.7.



θ_{12} = 1D-интерполяция между θ_1 и θ_2 вдоль вертикальной оси
θ_{24} = 1D-интерполяция между θ_2 и θ_4 вдоль горизонтальной оси
θ_{13} = 1D-интерполяция между θ_1 и θ_3 вдоль горизонтальной оси
θ_{34} = 1D-интерполяция между θ_3 и θ_4 вдоль вертикальной оси

$$U_{\boldsymbol{\theta}}(s) = \begin{cases} \text{1D-интерполяция между } \theta_{12} \text{ и } \theta_{34} \text{ вдоль гор. оси} \\ \text{или} \\ \text{1D-интерполяция между } \theta_{13} \text{ и } \theta_{24} \text{ вдоль верт. оси} \end{cases}_1$$

Рис. 8.7. Линейная интерполяция на двумерной сетке достигается путем линейной интерполяции по каждой оси в произвольном порядке

Если даны четыре вершины с координатами $s_1 = [x_1, y_1]$, $s_2 = [x_1, y_2]$, $s_3 = [x_2, y_1]$ и $s_4 = [x_2, y_2]$ и выборочное состояние $s = [x, y]$, то интерполированное значение равно

$$U_{\boldsymbol{\theta}}(s) = \alpha\theta_{12} + (1-\alpha)\theta_{34} \tag{8.8}$$

$$= \frac{x_2 - x}{x_2 - x_1}\theta_{12} + \frac{x - x_1}{x_2 - x_1}\theta_{34} \tag{8.9}$$

$$= \frac{x_2 - x}{x_2 - x_1}(\alpha\theta_1 + (1-\alpha)\theta_2) + \frac{x - x_1}{x_2 - x_1}(\alpha\theta_3 + (1-\alpha)\theta_4) \tag{8.10}$$

$$= \frac{x_2 - x}{x_2 - x_1}\left(\frac{y_2 - y}{y_2 - y_1}\theta_1 + \frac{y - y_1}{y_2 - y_1}\theta_2\right) + \frac{x - x_1}{x_2 - x_1}\left(\frac{y_2 - y}{y_2 - y_1}\theta_3 + \frac{y - y_1}{y_2 - y_1}\theta_4\right) \tag{8.11}$$

$$= \frac{(x_2 - x)(y_2 - y)}{(x_2 - x_1)(y_2 - y_1)}\theta_1 + \frac{(x_2 - x)(y - y_1)}{(x_2 - x_1)(y_2 - y_1)}\theta_2$$
$$+ \frac{(x - x_1)(y_2 - y)}{(x_2 - x_1)(y_2 - y_1)}\theta_3 + \frac{(x - x_1)(y - y_1)}{(x_2 - x_1)(y_2 - y_1)}\theta_4. \tag{8.12}$$

В результате интерполяции каждой вершине присваивается вес в соответствии с площадью противоположного ей квадранта, как показано на рис. 8.8.

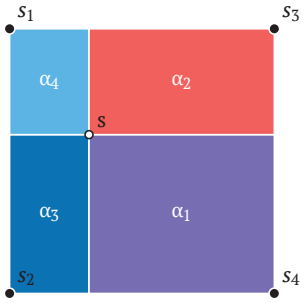


Рис. 8.8. Линейная интерполяция на двумерной сетке дает вклад каждой вершины, равный относительной площади противоположного ей квадранта: $U_\theta(s) = \alpha_1\theta_1 + \alpha_2\theta_2 + \alpha_3\theta_3 + \alpha_4\theta_4$

Полилинейная интерполяция (multilinear interpolation) в d измерениях аналогичным образом достигается путем линейной интерполяции вдоль каждой оси, что требует наличия 2^d вершин. В этом случае полезность каждой вершины взвешивается в соответствии с объемом противоположного гиперпрямоугольника. Полилинейная интерполяция реализована в алгоритме 8.4. Рисунок 8.9 демонстрирует этот подход в двумерном пространстве состояний.

Алгоритм 8.4. Реализация метода полилинейной интерполяции для вычисления вектора состояния `s` при известных значениях состояния `θ` по сетке, определяемой нижней левой вершиной `o` и вектором ширины `δ`. Все вершины сетки можно записать как `o + δ.*i` для некоторого несобственного целого вектора `i`. Пакет `Interpolations.jl` также предлагает полилинейные и другие методы интерполяции

```
mutable struct MultilinearValueFunction
    o # позиция левого нижнего угла
    δ # вектор ширины
    θ # вектор значений состояний в S
end

function (Uθ::MultilinearValueFunction)(s)
    o, δ, θ = Uθ.o, Uθ.δ, Uθ.θ
    Δ = (s - o)./δ
    # Многомерный индекс левой нижней ячейки
```

```
    i = min.(floor.(Int, Δ) .+ 1, size(θ) .- 1)
    vertex_index = similar(i)
    d = length(s)
    u = 0.0
    for vertex in 0:2^d-1
        weight = 1.0
        for j in 1:d
            # Проверка, равен ли 1 j-й бит
            if vertex & (1 << (j-1)) > 0
                vertex_index[j] = i[j] + 1
                weight *= Δ[j] - i[j] + 1
            else
                vertex_index[j] = i[j]
                weight *= i[j] - Δ[j]
            end
        end
        u += θ[vertex_index...]*weight
    end
    return u
end

function fit!(Uθ::MultilinearValueFunction, S, U)
    Uθ.θ = U
    return Uθ
end
```

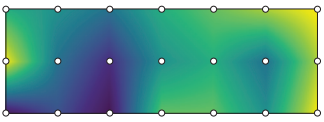


Рис. 8.9. Двумерная линейная интерполяция по сетке 3×7

8.5. Симплексная интерполяция

Полилинейная интерполяция становится неэффективной при больших размерностях. Вместо того чтобы взвешивать вклады 2^d точек, *симплексная интерполяция* (simplex interpolation) рассматривает только $d + 1$ точек в окрестности данного состояния, чтобы создать непрерывную поверхность, которая соответствует известным точкам выборки.

Мы начинаем с многомерной сетки и делим каждую ячейку на $d!$ *симплексов*, которые являются многомерными обобщениями треугольников, определяемых *выпуклой оболочкой* (convex hull) из $d + 1$ вершины. Этот процесс известен как *триангуляция Коксетера–Фрейденталя–Куна*[3] и гарантирует, что любые два симплекса, которые имеют общую грань, будут давать эквивалентные зна-

[3] A. W. Moore, *Simplicial Mesh Generation with Applications*, Ph.D. dissertation, Cornell University, 1992.

чения по всей грани, таким образом обеспечивая непрерывность при интерполяции, как показано на рис. 8.10.

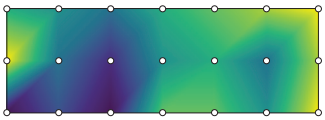


Рис. 8.10. Двумерная симплексная интерполяция по сетке 3×7

Для наглядности предположим, что мы переместили и масштабировали ячейку, содержащую состояние, таким образом, что самая нижняя вершина равна 0, а противоположная по диагонали вершина равна 1. Для каждой перестановки 1 существует симплекс 1:*d*. Симплекс, заданный перестановкой **p**, представляет собой множество точек **x**, удовлетворяющих неравенству

$$0 \le x_{p_1} \le x_{p_2} \le \cdots \le x_{p_3} \le 1. \tag{8.13}$$

На рис. 8.11 показаны симплексы, полученные для единичного куба.

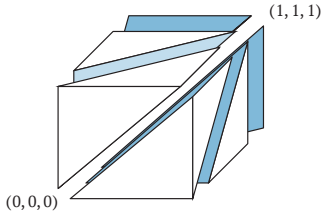
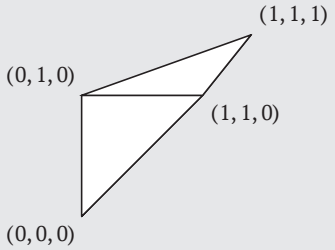


Рис. 8.11. Триангуляция единичного куба. На основе рис. 2.1 из работы A. W. Moore, *Simplicial Mesh Generation with Applications*, Ph. D. dissertation, Cornell University, 1992

В процессе симплексной интерполяции сначала транслируют и масштабируют вектор состояния **s** в единичный гиперкуб соответствующей ему ячейки, чтобы получить **s′**. Затем сортируют записи в **s′**, чтобы определить, какой симплекс содержит **s′**. Далее полезность **s′** может быть выражена уникальной линейной комбинацией вершин этого симплекса.

В примере 8.1 представлен пример симплексной интерполяции. Процесс реализован в алгоритме 8.5.

Пример 8.1. Симплексная интерполяция в трех измерениях



Рассмотрим трехмерный симплекс, заданный перестановкой **p** = [3, 1, 2], такой, что точки внутри симплекса удовлетворяют условию $0 \leqslant x_3 \leqslant x_1 \leqslant x_2 \leqslant 1$. Этот симплекс имеет вершины (0, 0, 0), (0, 1, 0), (1, 1, 0) и (1, 1, 1).

Таким образом, любая точка **s**, принадлежащая симплексу, может быть выражена с помощью взвешивания вершин:

$$\begin{bmatrix} s_1 \\ s_2 \\ s_3 \end{bmatrix} = w_1 \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + w_2 \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} + w_3 \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} + w_4 \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}.$$

Мы можем последовательно определить значения трех последних весов:

$w_4 = s_3$; $w_3 = s_1 - w_4$; $w_2 = s_2 - w_3 - w_4$.

Затем мы получаем w_1, добиваясь, чтобы сумма весов равнялась 1.

Если **s** = [0.3, 0.7, 0.2], то веса равны

$w_4 = 0.2$; $w_3 = 0.1$; $w_2 = 0.4$; $w_1 = 0.3$.

Алгоритм 8.5. Реализация метода симплексной интерполяции для нахождения значения вектора состояния `s` по известным значениям состояния `θ` на сетке, определяемой нижней левой вершиной `o` и вектором ширины `δ`. Все вершины сетки можно записать как `o + δ.*i` для некоторого неотрицательного целого вектора `i`. Симплексная интерполяция также реализована в общем пакете `GridInterpolations.jl`

```
mutable struct SimplexValueFunction
    o # позиция левого нижнего угла
    δ # вектор ширины
    θ # вектор значений состояний в S
end

function (Uθ::SimplexValueFunction)(s)
    Δ = (s - Uθ.o)./Uθ.δ
    # Многомерный индекс правой верхней ячейки
    i = min.(floor.(Int, Δ) .+ 1, size(Uθ.θ) .- 1) .+ 1
    u = 0.0
    s′ = (s - (Uθ.o + Uθ.δ.*(i.-2))) ./ Uθ.δ
    p = sortperm(s′) # сортировка по увеличению
    w_tot = 0.0
    for j in p
        w = s′[j] - w_tot
        u += w*Uθ.θ[i...]
        i[j] -= 1
        w_tot += w
    end
```

```
    u += (1 - w_tot)*U0.θ[i...]
    return u
end

function fit!(U0::SimplexValueFunction, S, U)
    U0.θ = U
    return U0
end
```

8.6. Линейная регрессия

Простым методом глобальной аппроксимации является *линейная регрессия*, где $U_\theta(s)$ – линейная комбинация *базисных функций* (basis functions), также обычно называемых *признаками* (feature). Эти базисные функции обычно являются нелинейной функцией состояния s и объединяются в векторную функцию $\boldsymbol{\beta}(s)$ или $\boldsymbol{\beta}(s, a)$, что приводит к приближениям

$$U_\theta(s) = \boldsymbol{\theta}^\top \boldsymbol{\beta}(s); \qquad Q_\theta(s, a) = \boldsymbol{\theta}^\top \boldsymbol{\beta}(s, a). \tag{8.14}$$

Хотя наша аппроксимация является линейной по отношению к базисным функциям, ее результат может быть нелинейным по отношению к основным переменным состояния, как показано на рис. 8.12. В примере 8.2 представлена глобальная линейная аппроксимация значений с использованием полиномиальных базисных функций для непрерывной задачи горной машины, которая приводит к нелинейной аппроксимации функции полезности относительно переменных состояния.

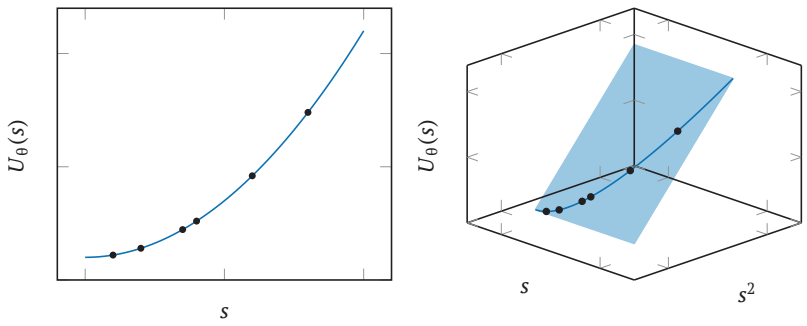


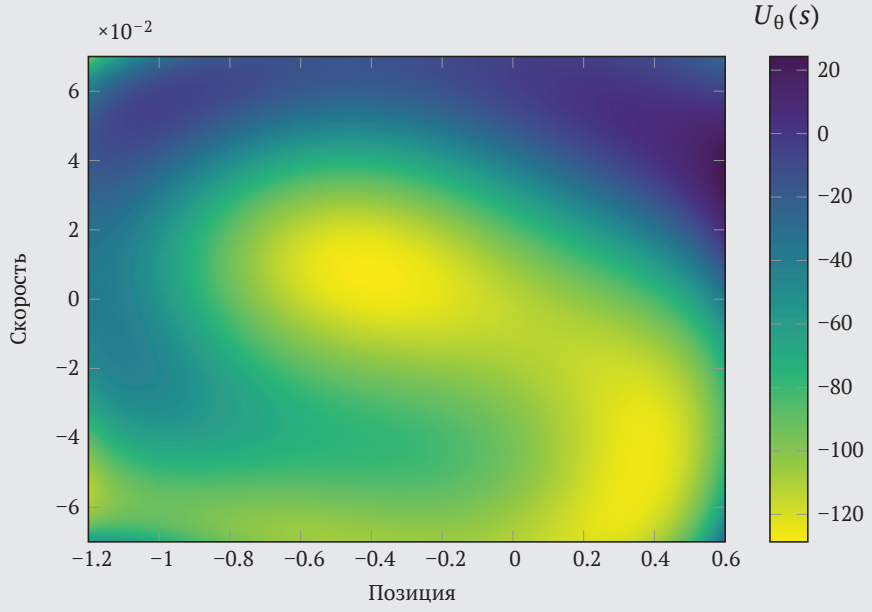
Рис. 8.12. Линейная регрессия с нелинейными базисными функциями является линейной в более высоких измерениях. Здесь полиномиальную регрессию можно рассматривать как линейную в трехмерном пространстве. Функция существует в плоскости, образованной ее основаниями, но не занимает всю плоскость, поскольку члены не являются независимыми

Пример 8.2. Использование линейной аппроксимации функции полезности в задаче горной машины. Выбор базисных функций имеет большое значение. Оптимальная функция полезности для машины нелинейна, имеет форму спирали и разрывы. Даже полиномы шестой степени не дают идеального соответствия

Мы можем аппроксимировать функцию полезности для задачи горной машины, используя линейное приближение. Задача имеет непрерывное пространство состояний с двумя измерениями, состоящими из позиции x и скорости v. Ниже показаны базисные функции до шестой степени:

$$\boldsymbol{\beta}(s) = \begin{bmatrix} 1, & & & & & & \\ x, & v, & & & & & \\ x^2, & xv, & v^2, & & & & \\ x^3, & x^2v, & xv^2, & v^3, & & & \\ x^4, & x^3v, & x^2v^2, & xv^3, & v^4, & & \\ x^5, & x^4v, & x^3v^2, & x^2v^3, & xv^4, & v^5, & \\ x^6, & x^5v, & x^4v^2, & x^3v^3, & x^2v^4, & xv^5, & v^6 \end{bmatrix}$$

Так выглядит график приближенной функции полезности, подогнанной по парам состояние-значение из экспертной стратегии:



Добавление большего количества базисных функций обычно улучшает способность достигать целевой полезности состояний в $\mathcal{S}$, но слишком большое количество базисных функций может привести к плохим аппроксимациям в других состояниях. В литературе описаны методы выбора подходящего набора базисных функций для регрессионной модели[4].

Подгонка линейных моделей включает в себя определение вектора $\boldsymbol{\theta}$, который минимизирует квадрат ошибки предсказаний в состояниях в $\mathcal{S} = s_{1:m}$. Если полезности, связанные с этими состояниями, обозначены как $u_{1:m}$, то нам необходимо найти $\boldsymbol{\theta}$, который минимизирует выражение

$$\sum_{i=1}^{m} (\hat{U}_{\boldsymbol{\theta}}(s_i) - u_i)^2 = \sum_{i=1}^{m} (\boldsymbol{\theta}^\top \boldsymbol{\beta}(s_i) - u_i)^2. \tag{8.15}$$

Оптимальный вектор $\boldsymbol{\theta}$ можно вычислить с помощью некоторых простых матричных операций. Сначала построим матрицу $\mathbf{X}$, где каждая из m строк $\mathbf{X}_i$ содержит $\boldsymbol{\beta}(s_i)^\top$[5]. Можно показать, что значение $\boldsymbol{\theta}$, минимизирующее квадрат ошибки, равно

$$\boldsymbol{\theta} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top u_{1:m} = \mathbf{X}^+ u_{1:m}, \tag{8.16}$$

где $\mathbf{X}^+$ – *псевдообратная матрица Мура–Пенроуза* от матрицы $\mathbf{X}$. Псевдообратную матрицу часто получают, сначала вычисляя *сингулярное разложение* $\mathrm{X} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{U}^*$. Тогда мы получаем

$$\mathbf{X}^+ = \mathbf{U}\boldsymbol{\Sigma}^+\mathbf{U}^*. \tag{8.17}$$

Псевдообратная матрица от диагональной матрицы $\boldsymbol{\Sigma}$ получается путем взятия обратной величины каждого ненулевого элемента диагонали и последующего транспонирования результата.

На рис. 8.13 показано, как полезности состояний в $\mathcal{S}$ подгоняются с использованием разных семейств базисных функций. Разный выбор базисных функций приводит к разным ошибкам.

Алгоритм 8.6 обеспечивает реализацию оценки и подгонки моделей линейной регрессии функции полезности. Пример 8.3 демонстрирует этот подход на примере задачи о горной машине.

---

[4] См. главу 14 книги M. J. Kochenderfer and T. A. Wheeler, *Algorithms for Optimization*. MIT Press, 2019, или главу 7 книги T. Hastie, R. Tibshirani, J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2nd ed. Springer Series in Statistics, 2001.

[5] Обзор математической базы, лежащей в основе линейной регрессии, а также более продвинутых методов, см. в T. Hastie, R. Tibshirani, J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2nd ed. Springer Series in Statistics, 2001.

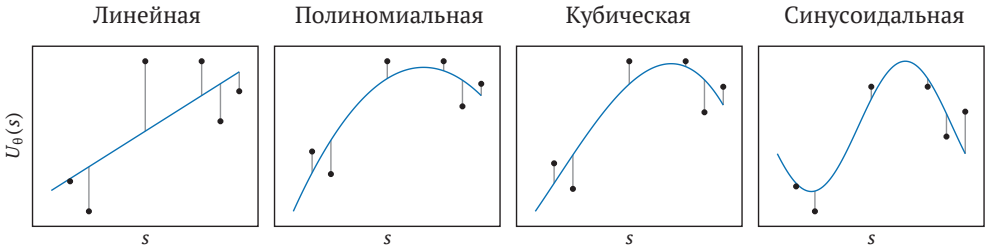


Рис. 8.13. Линейная регрессия с различными семействами базисных функций

Алгоритм 8.6. Аппроксимация функции значения линейной регрессии, заданная базисной векторной функцией `β` и вектором параметров `θ`. Функция `pinv` реализует псевдоинверсию. Julia, как и другие языки, поддерживает оператор обратной косой черты, что позволяет нам писать `X \ U` вместо `pinv(X)*U` в функции `fit!`

```
mutable struct LinearRegressionValueFunction
    β # функция базисного вектора
    θ # вектор параметров
end

function (Uθ::LinearRegressionValueFunction)(s)
    return Uθ.β(s) ⋅ Uθ.θ
end

function fit!(Uθ::LinearRegressionValueFunction, S, U)
    X = hcat([Uθ.β(s) for s in S]...)'
    Uθ.θ = pinv(X)*U
    return Uθ
end
```

Пример 8.3. Линейная регрессия с использованием базы Фурье, используемой для аппроксимации функции стоимости в задаче о горной машине (приложение F.4)

Показаны функции полезности (верхний ряд) и результирующие стратегии (нижний ряд). Глобально аппроксимированная функция полезности плохо совпадает с фактическими значениями, несмотря на использование базисных функций Фурье восьмого порядка. Полученная стратегия не является хорошим приближением к экспертной стратегии. Небольшой временной шаг в задаче о горной машине приводит к тому, что даже небольшие изменения в ландшафте функции полезности влияют на стратегию. Оптимальные функции полезности часто имеют сложную геометрию, которую трудно воплотить с помощью глобальных базисных функций.

Мы можем применить линейную регрессию, чтобы обучить функцию полезности для задачи горной машины. Оптимальная функция полезности

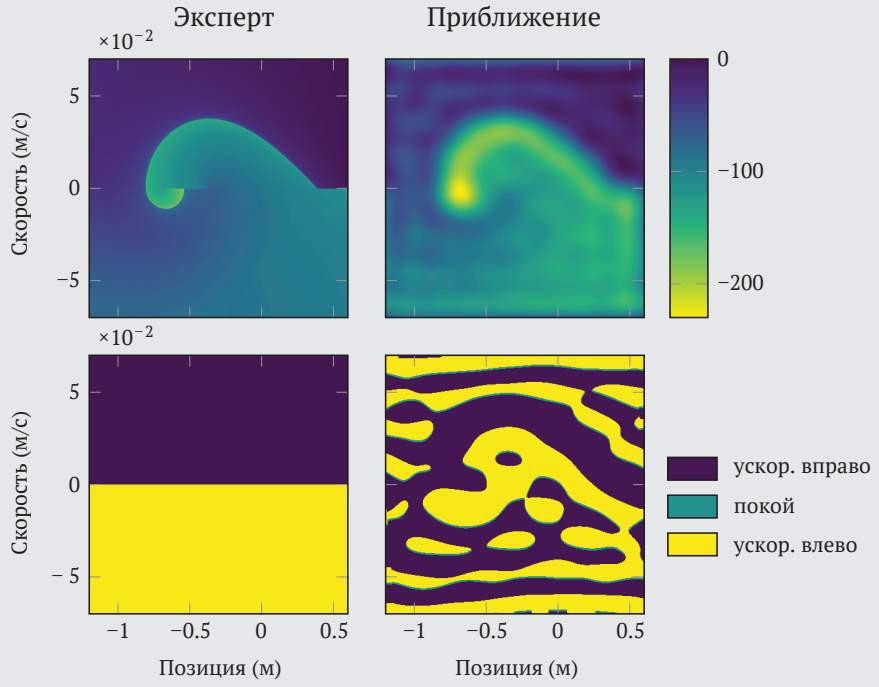
имеет форму спирали, которую трудно аппроксимировать полиномиальными базисными функциями (пример 8.2). Мы используем базисные функции Фурье, компоненты которых имеют следующий вид:

$b_0(x) = ½$;

$b_{s,i}(x) = \sin(2\pi ix/T)$ для $i = 1, 2, \ldots$;

$b_{c,i}(x) = \cos(2\pi ix/T)$ для $i = 1, 2, \ldots$,

где T – ширина домена компонента. Многомерные базисные функции Фурье представляют собой все комбинации одномерных компонентов по осям пространства состояний. Здесь мы используем приближение восьмого порядка, поэтому i принимает значения до 8. Экспертная стратегия состоит в том, чтобы ускоряться в направлении движения.



8.7. Регрессия на основе нейронной сети

Регрессия на основе нейронной сети избавляет нас от необходимости строить соответствующий набор базисных функций, как это требуется в линейной регрессии. Вместо этого для представления функции полезности используется *нейронная сеть*. Обзор нейронных сетей приводится в приложении D. Входными данными нейронной сети будут переменные состояния, а на вы-

ходе мы получим полезность. Параметры $\boldsymbol{\theta}$ будут соответствовать *весам* в нейронной сети.

Как сказано в приложении D, мы можем оптимизировать веса сети для достижения конкретной цели. В контексте приближенного динамического программирования мы хотели бы минимизировать ошибку наших предсказаний, как мы это делали в предыдущем разделе. Однако минимизация квадратичной ошибки не может быть достигнута с помощью простых матричных операций. Вместо этого нам обычно приходится применять методы оптимизации, такие как градиентный спуск. К счастью, вычисление градиента нейронных сетей можно точно выполнить с помощью *цепного правила* (правила дифференцирования сложной функции как цепочки производных).

8.8. Заключение

- Для решения масштабных или непрерывных задач мы можем попытаться найти приблизительные стратегии, представленные параметрическими моделями функции полезности.
- Подходы, рассмотренные в этой главе, предусматривают итеративное применение шагов динамического программирования в конечном наборе состояний и повторную подгонку параметрической аппроксимации.
- Методы локальной аппроксимации находят приближенную функцию полезности на основе значений близких состояний с известными значениями.
- К методам локальной аппроксимации относятся метод ближайшего соседа, сглаживание ядра, линейная и симплексная интерполяция.
- К методам глобальной аппроксимации относятся линейная регрессия и регрессия на основе нейронной сети.
- Нелинейные функции полезности можно получить при использовании линейной регрессии в сочетании с соответствующим выбором нелинейных базисных функций.
- Регрессия на основе нейронной сети избавляет нас от необходимости находить базисные функции, но ее обучение является более сложной задачей и обычно требует использования градиентного спуска для настройки параметрической аппроксимации функции полезности.

8.9. Упражнения

Упражнение 8.1. Методы аппроксимации функции полезности, представленные в этой главе, в основном предполагают непрерывные пространства состояний. Задача гексамира (приложение F.1) является дискретной, но большинство ее состояний можно отобразить в двухмерные местоположения. Однако у нее есть дополнительное конечное состояние, дающее нулевое вознаграждение, которое не имеет двумерного местоположения. Как можно изменить методы

аппроксимации непрерывной функции полезности, описанные в этой главе, для обработки такого состояния?

Решение. Задача гексамира требует от агента перемещаться по двумерной шестиугольной сетке. Однако агент может войти в единственное конечное состояние из нескольких разных плиток сетки. Это единственное конечное состояние представляет собой проблему для методов аппроксимации функции полезности, которые часто полагаются на близость для вывода значения состояния.

Конечное состояние можно спроецировать в то же пространство, что и другие состояния, возможно, далеко от них. Этот прием тем не менее привнесет некоторую форму близости в вычисление значения конечного состояния. Выбор одной позиции для состояния, которое должно быть равноудалено от нескольких предшествующих состояний, приводит к смещению.

Альтернатива состоит в том, чтобы рассматривать конечное состояние как частный случай. Ядерную функцию можно изменить так, чтобы возникло бесконечное расстояние между конечным состоянием и любыми другими состояниями.

Другой вариант – настроить задачу таким образом, чтобы у каждой плитки было конечное состояние, приносящее конечное вознаграждение. Каждое конечное состояние может совпадать со своим предшествующим состоянием, но смещено в дополнительном измерении. Это преобразование поддерживает близость за счет дополнительных состояний.

Упражнение 8.2. Табличное представление является частным случаем линейных приближенных функций полезности. Покажите, каким образом для любой дискретной задачи табличное представление может рассматриваться как линейная приближенная функция полезности.

Решение. Рассмотрим дискретный MDP с m состояниями $s_{1:m}$ и n действиями $a_{1:n}$. Табличное представление связывает значение с каждым состоянием или парой состояние-действие. Мы можем добиться того же самого, используя линейную приближенную функцию полезности. Мы ассоциируем индикаторную функцию с каждым состоянием или парой состояние-действие, при этом значение функции равно 1, когда входными данными является заданное состояние или пара состояние-действие, и 0 в противном случае:

$$\beta_i(s) = (s = s_i) = \begin{cases} 1, & \text{если } s = s_i \\ 0 & \text{в ином случае} \end{cases}$$

или

$$\beta_{ij}(s,a) = \big((s,a) = (s_i,a_j)\big) = \begin{cases} 1, & \text{если } (s,a) = (s_i,a_j) \\ 0 & \text{в ином случае} \end{cases}.$$

Упражнение 8.3. Предположим, что у нас есть задача с непрерывными пространствами состояний и действий и мы хотели бы построить как локальную,

так и глобальную аппроксимацию функции полезности действия $Q(s, a) = \boldsymbol{\theta}^\top \boldsymbol{\beta}(s, a)$. Для глобальной аппроксимации выберем базисные функции

$$\boldsymbol{\beta}(s, a) = [1, s, a, s^2, sa, a^2].$$

Если нам дан набор из 100 состояний $\mathcal{S} = s_{1:100}$ и набор из пяти действий $A = a_{1:5}$, сколько параметров находится в $\boldsymbol{\theta}$ для метода локальной аппроксимации? Сколько параметров находится в $\boldsymbol{\theta}$ для указанного метода глобальной аппроксимации?

Решение. В методах локальной аппроксимации значения состояния-действия являются параметрами. У нас будет $|\mathcal{S}| \times |A| = 100 \times 5 = 500$ параметров в $\boldsymbol{\theta}$. В методах глобальной аппроксимации параметрами являются коэффициенты базисных функций. Поскольку в $\boldsymbol{\beta}$(s, a) шесть компонент, у нас будет шесть параметров в $\boldsymbol{\theta}$.

Упражнение 8.4. Заданы состояния $s_1 = (4,5)$, $s_2 = (2,6)$ и $s_3 = (-1, -1)$ и соответствующие им значения $U(s_1) = 2$, $U(s_2) = 10$ и $U(s_3) = 30$. Вычислите значение в состоянии $s = (1,2)$, используя локальное приближение по методу k-ближайших соседей ($k = 2$) с метрикой расстояния L_1, с метрикой расстояния L_2 и с метрикой расстояния L_∞.

Решение. Занесем в таблицу расстояния от s до точек $s' \in \mathcal{S}$, как указано ниже:

$s' \in S$	L_1	L_2	L_∞
$s_1 = (4, 5)$	6	$\sqrt{18}$	3
$s_2 = (2, 6)$	5	$\sqrt{17}$	4
$s_3 = (-1, -1)$	5	$\sqrt{13}$	3

Используя норму L_1, мы находим $U(s) = (10 + 30)/2 = 20$. Используя норму L_2, мы находим $U(s) = (10 + 30)/2 = 20$. Используя норму L_∞, находим $U(s) = (2 + 30)/2 = 16$.

Упражнение 8.5. Допустим, мы хотим найти полезность в состоянии s, исходя из полезностей в наборе из двух состояний $\mathcal{S} = \{s_1, s_2\}$. Если мы хотим использовать локальную аппроксимацию итерации по критерию, какие из приведенных ниже весовых функций допустимы? Если они недопустимы, как можно изменить весовые функции, чтобы сделать их допустимыми?

- $\boldsymbol{\beta}(s) = [1, 1]$;
- $\boldsymbol{\beta}(s) = [1 - \lambda, \lambda]$, где $\lambda \in [0, 1]$;
- $\boldsymbol{\beta}(s) = [e^{(s-s_1)^2}, e^{(s-s_2)^2}]$.

Решение. Первый набор весовых функций недопустим, так как он нарушает ограничение $\sum_i \beta_i(s) = 1$. Мы можем изменить весовые функции, нормализовав их по их сумме:

$$\boldsymbol{\beta}(s) = \left[\frac{1}{1+1}, \frac{1}{1+1}\right] = \left[\frac{1}{2}, \frac{1}{2}\right].$$

Второй набор весовых функций допустим. Третий набор весовых функций недопустим, так как он нарушает ограничение $\sum_i \beta_i(s) = 1$. Мы можем изменить весовые функции, нормализовав их по их сумме:

$$\boldsymbol{\beta}(s) = \left[\frac{e^{(s-s_1)^2}}{e^{(s-s_1)^2} + e^{(s-s_2)^2}}, \frac{e^{(s-s_2)^2}}{e^{(s-s_1)^2} + e^{(s-s_2)^2}}\right].$$

Упражнение 8.6. Докажите, что билинейная интерполяция инвариантна относительно (ненулевого) масштабирования линейной сетки.

Решение. Несложно показать, что интерполированное значение инвариантно к линейному масштабированию по одной или обеим осям, например $\tilde{U}_{\boldsymbol{\theta}}(\tilde{s}) = U_{\boldsymbol{\theta}}(s)$. Докажем это, заменяя все значения x и y их масштабированными версиями $\tilde{x} = \beta x$ и $\tilde{y} = \gamma y$ и показывая, что масштабы сетки компенсируются:

$$\begin{aligned}\tilde{U}_{\boldsymbol{\theta}}(\tilde{s}) = {} & \frac{(\tilde{x}_2 - \tilde{x})(\tilde{y}_2 - \tilde{y})}{(\tilde{x}_2 - \tilde{x}_1)(\tilde{y}_2 - \tilde{y}_1)}\theta_1 + \frac{(\tilde{x}_2 - \tilde{x})(\tilde{y} - \tilde{y}_1)}{(\tilde{x}_2 - \tilde{x}_1)(\tilde{y}_2 - \tilde{y}_1)}\theta_2 \\ & + \frac{(\tilde{x} - \tilde{x}_1)(\tilde{y}_2 - \tilde{y})}{(\tilde{x}_2 - \tilde{x}_1)(\tilde{y}_2 - \tilde{y}_1)}\theta_3 + \frac{(\tilde{x} - \tilde{x}_1)(\tilde{y} - \tilde{y}_1)}{(\tilde{x}_2 - \tilde{x}_1)(\tilde{y}_2 - \tilde{y}_1)}\theta_4; \\ \tilde{U}_{\boldsymbol{\theta}}(\tilde{s}) = {} & \frac{\beta(x_2 - x)\beta(y_2 - y)}{\beta(x_2 - x_1)\beta(y_2 - y_1)}\theta_1 + \frac{\beta(x_2 - x)\beta(y - y_1)}{\beta(x_2 - x_1)\beta(y_2 - y_1)}\theta_2 \\ & + \frac{\beta(x - x_1)\beta(y_2 - y)}{\beta(x_2 - x_1)\beta(y_2 - y_1)}\theta_3 + \frac{\beta(x - x_1)\beta(y - y_1)}{\beta(x_2 - x_1)\beta(y_2 - y_1)}\theta_4; \\ \tilde{U}_{\boldsymbol{\theta}}(\tilde{s}) = {} & \frac{(x_2 - x)(y_2 - y)}{(x_2 - x_1)(y_2 - y_1)}\theta_1 + \frac{(x_2 - x)(y - y_1)}{(x_2 - x_1)(y_2 - y_1)}\theta_2 \\ & + \frac{(x - x_1)(y_2 - y)}{(x_2 - x_1)(y_2 - y_1)}\theta_3 + \frac{(x - x_1)(y - y_1)}{(x_2 - x_1)(y_2 - y_1)}\theta_4; \\ \tilde{U}_{\boldsymbol{\theta}}(\tilde{s}) = {} & U_{\boldsymbol{\theta}}(s).\end{aligned}$$

Упражнение 8.7. Пусть даны четыре состояния s_1 – [0.5], s_2 – [0.25], s_3 – [1.5] и s_4 – [1.25] и выборочное состояние s – [0.7, 10]. Сформулируйте уравнение интерполирующей функции $U_{\boldsymbol{\theta}}(s)$ для произвольного $\boldsymbol{\theta}$.

Решение. Общая форма для билинейной интерполяции дана в уравнении (8.12) и ниже повторена для удобства. Чтобы получить уравнение интерполирующей функции, мы подставляем наши значения в уравнение и упрощаем:

$$U_{\boldsymbol{\theta}}(s) = \frac{(x_2 - x)(y_2 - y)}{(x_2 - x_1)(y_2 - y_1)}\theta_1 + \frac{(x_2 - x)(y - y_1)}{(x_2 - x_1)(y_2 - y_1)}\theta_2 + \frac{(x - x_1)(y_2 - y)}{(x_2 - x_1)(y_2 - y_1)}\theta_3 + \frac{(x - x_1)(y - y_1)}{(x_2 - x_1)(y_2 - y_1)}\theta_4;$$

$$U_{\boldsymbol{\theta}}(s) = \frac{(1 - 0.7)(25 - 10)}{(1 - 0)(25 - 5)}\theta_1 + \frac{(1 - 0.7)(10 - 5)}{(1 - 0)(25 - 5)}\theta_2 + \frac{(0.7 - 0)(25 - 10)}{(1 - 0)(25 - 5)}\theta_3 + \frac{(0.7 - 0)(10 - 5)}{(1 - 0)(25 - 5)}\theta_4;$$

$$U_{\boldsymbol{\theta}}(s) = \frac{9}{40}\theta_1 + \frac{3}{40}\theta_2 + \frac{21}{40}\theta_3 + \frac{7}{40}\theta_4.$$

Упражнение 8.8. Вернитесь к примеру 8.1. Какими будут симплексные интерполирующие веса для состояния **s** = [0.4, 0.95, 0.6]?

Решение. Для данного состояния **s** имеем $0 \leqslant x_1 \leqslant x_3 \leqslant x_2 \leqslant 1$, поэтому наш вектор перестановки равен **p** = [1, 3, 2]. Вершины нашего симплекса можно создать, начав с (0, 0, 0) и заменив каждый 0 на 1 в порядке, обратном вектору перестановки. Таким образом, вершинами симплекса являются точки (0, 0, 0), (0, 1, 0), (0, 1, 1) и (1, 1, 1).

Итак, любая точка **s**, принадлежащая симплексу, может быть выражена путем взвешивания вершин:

$$\begin{bmatrix} s_1 \\ s_2 \\ s_3 \end{bmatrix} = w_1 \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + w_2 \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} + w_3 \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} + w_4 \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}.$$

Мы можем определить значения весов в обратном порядке – в конце найдя w_1 – в соответствии с ограничением, согласно которому веса должны в сумме равняться 1. Затем можем вычислить веса для **s** = [0.4, 0.95, 0.6]:

$w_4 = s_1$; $w_3 = s_3 - w_4$; $w_2 = s_2 - w_3 - w_4$; $w_1 = 1 - w_2 - w_3 - w_4$;

$w_4 = 0.4$; $w_3 = 0.2$; $w_2 = 0.35$; $w_1 = 0.05$.

9 Онлайн-планирование

Методы, которые мы рассматривали до сих пор, планируют стратегии в режиме *офлайн* (offline, автономно), прежде чем какие-либо действия будут предприняты на практике. Но даже приближенные офлайн-методы могут быть неразрешимыми во многих многомерных задачах. В этой главе представлены методы *онлайн-планирования* (online planning, планирование в текущем времени), которые находят действия на основе рассуждений о состояниях, достижимых из текущего состояния. В этом случае *доступное пространство состояний* (reachable state space) часто на несколько порядков меньше, чем полное пространство состояний, что позволяет значительно снизить требования к хранилищу и вычислениям по сравнению с офлайн-методами. Мы обсудим различные алгоритмы, направленные на повышение эффективности онлайн-планирования, включая сокращение пространства состояний, выборку и углубленное планирование по траекториям, которые выглядят более многообещающими.

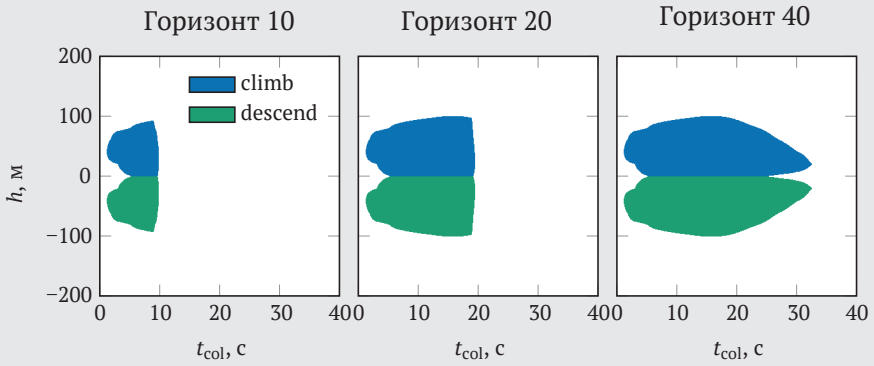
9.1. Планирование с отступающим горизонтом

При *планировании с отступающим горизонтом* (receding horizon planning) мы планируем свои действия от текущего состояния до максимального фиксированного горизонта или глубины d. Затем мы выполняем действие из нашего текущего состояния, переходим в следующее состояние и заново планируем действия. Все методы онлайн-планирования, обсуждаемые в этой главе, построены по схеме планирования с отступающим горизонтом. Они различаются лишь тем, как исследуют различные варианты действий.

Проблема применения планирования с отступающим горизонтом заключается в определении подходящей глубины d. Чем дальше удален горизонт планирования от текущего состояния, тем больше требуется вычислений. Для некоторых задач даже малая глубина оказывается весьма эффективной – тот факт, что мы заново планируем действия на каждом этапе, может компенсировать отсутствие долгосрочного моделирования. Другие задачи могут нуждаться в большой глубине планирования, чтобы планировщик мог успешно двигаться к цели или уходить от небезопасных состояний, как показано в примере 9.1.

Пример 9.1. Планирование с отступающим горизонтом для предотвращения столкновений при разной глубине планирования. В этой задаче есть четыре переменные состояния. На графиках ниже показаны срезы пространства состояний при допущении, что самолет в настоящее время находится в горизонтальном положении и еще не поступило предупреждение. Горизонтальная ось – это время до столкновения t_{col}, а вертикальная ось – наша h относительно другого самолета. В приложении F.6 приведены дополнительные условия этой задачи

Предположим, мы хотим применить планирование с отступающим горизонтом для предотвращения столкновений самолетов. Цель состоит в том, чтобы при необходимости вовремя дать рекомендации по снижению или набору высоты, дабы избежать столкновения. Столкновение происходит, когда наша высота относительно второго самолета h находится в пределах ±50 м, а время до потенциального столкновения t_{col} равно нулю. Здесь требуется довольно глубокое планирование, чтобы иметь возможность предоставить рекомендации достаточно рано. На графиках ниже показаны действия, которые были бы предприняты планировщиками с отступающим горизонтом при различной глубине.



Если глубина $d = 10$, система предоставляет рекомендации только в течение 10 с перед столкновением. Из-за ограничений динамики летательного аппарата и неопределенности поведения других воздушных судов столь поздние рекомендации ставят под угрозу безопасность. При $d = 20$ безопасность возрастает, но возможны случаи, когда хотелось бы получить рекомендации немного раньше, чтобы еще больше снизить риск столкновения. С другой стороны, нет никакого смысла в планировании глубже, чем $d = 40$, потому что нет необходимости рекомендовать какие-либо маневры настолько задолго до потенциального столкновения.

9.2. Стратегия развертывания

В главе 8 было рассмотрено нахождение стратегий, жадных по отношению к аппроксимированной функции полезности U, с помощью метода предпросмотра[1]. Простая рабочая стратегия представляет собой жадные действия по отношению к значениям полезности, найденным с помощью моделирования на глубину d. Чтобы выполнить моделирование с предпросмотром, нам нужна стратегия моделирования. Конечно, мы не знаем оптимальную стратегию, но вместо нее мы можем использовать так называемую *стратегию развертывания* (rollout policy). Стратегии развертывания, как правило, являются стохастическими, при этом действия берутся из распределения $a \sim \pi(s)$. Для создания моделей развертывания мы будем использовать *генеративную модель* $s' \sim T(s, a)$, которая генерирует состояния-преемники s' из распределения $T(s'|s, a)$. Такая генеративная модель может быть реализована с помощью генератора случайных чисел, что обычно проще реализуется на практике по сравнению с явным представлением распределения $T(s'|s, a)$.

Алгоритм 9.1 сочетает в себе метод предпросмотра с получением значений полезности действий, оцениваемых посредством развертывания. Этот подход часто приводит к лучшему поведению, чем исходная стратегия развертывания, но оптимальность не гарантируется. Его можно рассматривать как приблизительную форму улучшения стратегии, используемую в алгоритме итерации по стратегиям (раздел 7.4). Простым вариантом этого алгоритма является использование нескольких развертываний для получения более точной оценки ожидаемого дисконтированного дохода. Если мы запустим m моделей для каждого действия и результирующего состояния, вычислительная сложность по времени составит $O(m \times |\mathcal{A}| \times |\mathcal{S}| \times d)$.

Алгоритм 9.1. Функция, выполняющая развертывание стратегии π в задаче `f` из состояния `s` на глубину `d`. Она возвращает общее дисконтированное вознаграждение. Эту функцию можно использовать с функцией `greedy` (представленной в алгоритме 7.5) для создания действия, которое, вероятно, станет улучшением по сравнению с исходной стратегией развертывания. Мы будем использовать этот алгоритм позже для задач, отличных от MDP, требующих от нас только соответствующей модификации `randstep`

```
struct RolloutLookahead
    𝒫 # задача
    π # стратегия развертывания
    d # глубина
end
```

---

[1] Стратегия предпросмотра была первоначально введена в алгоритме 7.2 как часть нашего обсуждения методов точного решения.

```
randstep(𝒫::MDP, s, a) = 𝒫.TR(s, a)

function rollout(𝒫, s, π, d)
    ret = 0.0
    for t in 1:d
        a = π(s)
        s, r = randstep(𝒫, s, a)
        ret += 𝒫.γ^(t-1) * r
    end
    return ret
end

function (π::RolloutLookahead)(s)
    U(s) = rollout(π.𝒫, s, π.π, π.d)
    return greedy(π.𝒫, U, s).a
end
```

9.3. Прямой поиск

Прямой поиск (forward search, поиск вперед, в прямом направлении) определяет наилучшее действие из начального состояния s, расширяя все возможные переходы до глубины d. Эти расширения образуют *дерево поиска* (search tree)[2]. Такие деревья поиска имеют коэффициент ветвления в худшем случае $|\mathcal{S}|\times|\mathcal{A}|$, что дает вычислительную сложность $O((|\mathcal{S}|\times|\mathcal{A}|)^d)$. На рис. 9.1 показано дерево поиска задачи с тремя состояниями и двумя действиями. На рис. 9.2 показаны состояния, посещенные во время прямого поиска в задаче гексамира.

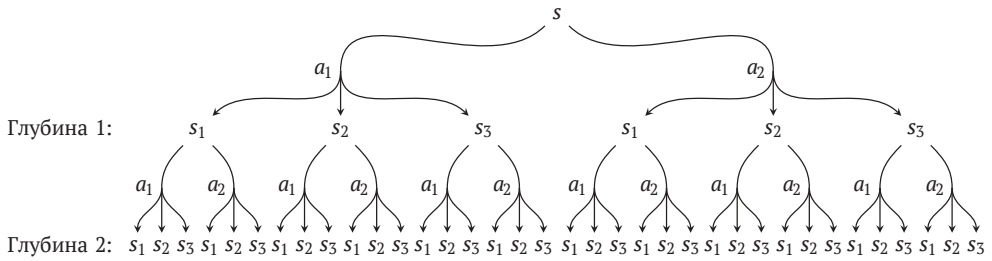


Рис. 9.1. Дерево прямого поиска задачи с тремя состояниями и двумя действиями

Алгоритм 9.2 рекурсивно вызывает себя до указанной глубины. Достигнув заданной глубины, он использует оценку полезности, предоставляемую функцией U. Если мы просто хотим планировать до указанного горизонта, то задаем $U(s) = 0$. Если текущая задача требует планирования за пределами глуби-

[2] Исследование дерева происходит как *поиск в глубину*. В приложении E рассмотрены как поиск в глубину, так и другие стандартные алгоритмы поиска в детерминированном контексте.

ны, ограниченной приемлемой вычислительной нагрузкой, можно применить функцию полезности, получаемую в офлайн-режиме, используя, например, одну из аппроксимаций функции полезности, описанных в предыдущей главе. Такое сочетание онлайн- и офлайн-методов иногда называют *гибридным планированием*.

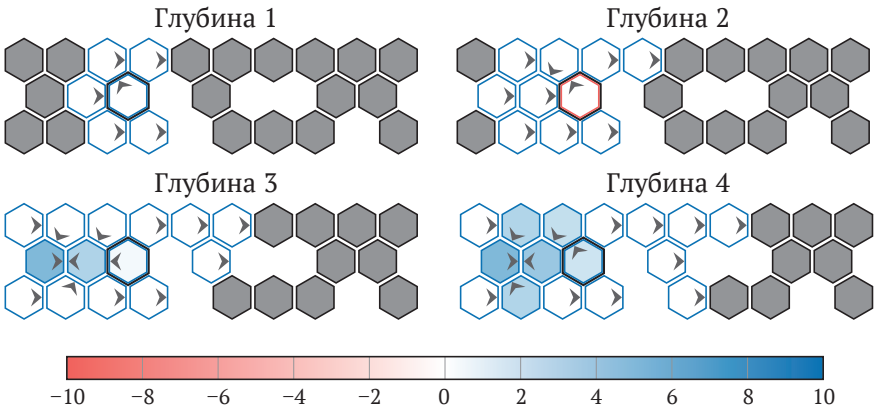


Рис. 9.2. Прямой поиск применяется к задаче гексамира с максимальной глубиной $d = 4$. Поиск может посетить узел несколько раз. Действия и цвета для посещенных состояний были выбраны в соответствии с наименее глубоким узлом с наибольшим значением в дереве поиска для этого состояния. Начальное состояние обозначено дополнительной черной рамкой

Алгоритм 9.2. Алгоритм прямого поиска для нахождения приблизительно оптимального действия, направленного на решение проблемы $\mathcal{P}$ из текущего состояния `s`. Поиск выполняется на глубину `d`, после чего конечное значение оценивается с помощью аппроксимированной функции полезности `U`. Возвращаемый именованный кортеж состоит из наилучшего действия `a` и его ожидаемой полезности `u` на конечном горизонте. Задача не обязательно должна принадлежать к классу MDP; в разделе 22.2 применяется тот же алгоритм в контексте частично наблюдаемых задач с другой реализацией `lookahead`

```
struct ForwardSearch
    𝒫 # задача
    d # глубина
    U # функция полезности на глубине d
end

function forward_search(𝒫, s, d, U)
    if d ≤ 0
        return (a=nothing, u=U(s))
    end
    best = (a=nothing, u=-Inf)
    U′(s) = forward_search(𝒫, s, d-1, U).u
```

```
    for a in 𝒫.𝒜
        u = lookahead(𝒫, U′, s, a)
        if u > best.u
            best = (a=a, u=u)
        end
    end
    return best
end

(π::ForwardSearch)(s) = forward_search(π.𝒫, s, π.d, π.U).a
```

9.4. Метод ветвей и границ

Метод ветвей и границ (branch and bound, алгоритм 9.3) пытается избежать экспоненциального роста вычислительной сложности прямого поиска. Он отсекает ветви, опираясь на границы функции полезности. Алгоритм требует знания нижней границы функции полезности состояния $\underline{U}(s)$ и верхней границы функции полезности действия $\bar{Q}(s, a)$. Нижняя граница используется для оценки состояний на максимальной глубине. Эта нижняя граница распространяется вверх по дереву посредством оператора Беллмана. Если мы обнаружим, что верхняя граница текущего действия в определенном состоянии ниже, чем нижняя граница ранее изученного действия из этого состояния, то нам не нужно исследовать это действие, что позволяет исключить связанное с ним поддерево из рассмотрения.

Метод ветвей и границ дает тот же результат, что и прямой поиск, но его эффективность зависит от того, сколько ветвей обрезано. Вычислительная сложность метода ветвей и границ в наихудшем случае такая же, как и при прямом поиске. Чтобы упростить исключение, действия просматриваются в порядке убывания по верхней границе. Более узкие границы обычно приводят к большему сокращению дерева, как показано в примере 9.2.

Алгоритм 9.3. Алгоритм ветвей и границ для поиска приблизительно оптимального действия в рабочем режиме для дискретного MDP $\mathcal{P}$ из текущего состояния `s`. Поиск выполняется на глубину `d` с нижней границей функции полезности `Ulo` и верхней границей функции полезности действия `Qhi`. Возвращенный именованный кортеж состоит из наилучшего действия `a` и его ожидаемой полезности `u` на конечном горизонте. Этот алгоритм также используется для POMDP

```
struct BranchAndBound
    𝒫 # задача
    d # глубина
    Ulo # нижняя граница функции полезности на глубине d
    Qhi # вехняя граница функции полезности действия
end
```

```
function branch_and_bound(𝒫, s, d, Ulo, Qhi)
    if d ≤ 0
        return (a=nothing, u=Ulo(s))
    end
    U′(s) = branch_and_bound(𝒫, s, d-1, Ulo, Qhi).u
    best = (a=nothing, u=-Inf)
    for a in sort(𝒫.𝒜, by=a->Qhi(s,a), rev=true)
        if Qhi(s, a) < best.u
            return best # не исключать
        end
        u = lookahead(𝒫, U′, s, a)
        if u > best.u
            best = (a=a, u=u)
        end
    end
    return best
end

(π::BranchAndBound)(s) = branch_and_bound(π.𝒫, s, π.d, π.Ulo, π.Qhi).a
```

Пример 9.2. Метод ветвей и границ применительно к задаче о горной машине (приложение F.4). Этот метод может обеспечить значительное ускорение по сравнению с прямым поиском

Рассмотрим применение метода ветвей и границ к задаче о горной машине. Мы можем использовать функцию полезности эвристической стратегии для нижней границы $\underline{U}(s)$, например эвристической стратегии, которая всегда ускоряется в направлении движения. В качестве верхней границы $\bar{Q}([x, v], a)$ мы можем использовать результат, возвращаемый при ускорении в направлении цели без холма. Метод ветвей и границ посещает примерно на треть меньше состояний, чем прямой поиск.

9.5. Разреженная выборка

Метод, известный как *разреженная выборка* (алгоритм 9.4)[3], пытается уменьшить коэффициент ветвления методов прямого поиска и ветвей и границ. Вместо ветвления по всем возможным следующим состояниям рассматривают только ограниченное число выборок следующего состояния. Хотя разреживание состояний неизбежно приводит к приближенному результату, этот метод может хорошо работать на практике и способен значительно сократить объем вычислений. Если мы извлекаем m выборок следующего состояния для каждого узла действия в дереве поиска, вычислительная сложность составит

[3] M. J. Kearns, Y. Mansour, A. Y. Ng, *A Sparse Sampling Algorithm for Near-Optimal Planning in Large Markov Decision Processes*, Machine Learning, vol. 49, no. 2–3, pp. 193–208, 2002.

$O((m \times |\mathcal{A}|)^d)$. Это значение по-прежнему экспоненциально зависит от глубины, но больше не связано с размером пространства состояний. Пример разреженной выборки показан на рис. 9.3.

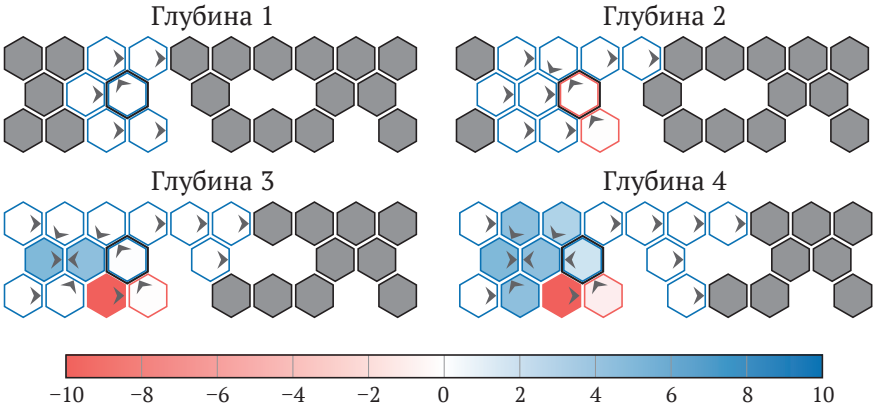


Рис. 9.3. Разреженная выборка с $m = 10$ применительно к задаче гексамира. Посещенные плитки окрашиваются в соответствии с их оценкой стоимости. Плитка с рамкой является начальным состоянием. Сравните с поиском вперед на рис. 9.2

Алгоритм 9.4. Алгоритм разреженной выборки для поиска приблизительно оптимального действия в рабочем режиме для дискретной задачи $\mathcal{P}$ от текущего состояния `s` до глубины `d` с `m` выборками на действие. Возвращенный именованный кортеж состоит из наилучшего действия `a` и его ожидаемой полезности `u` на конечном горизонте

```
struct SparseSampling
    𝒫 # задача
    d # глубина
    m # количество выборок
    U # функция полезности на глубине d
end

function sparse_sampling(𝒫, s, d, m, U)
    if d ≤ 0
        return (a=nothing, u=U(s))
    end
    best = (a=nothing, u=-Inf)
    for a in 𝒫.𝒜
        u = 0.0
        for i in 1:m
            s′, r = randstep(𝒫, s, a)
            a′, u′ = sparse_sampling(𝒫, s′, d-1, m, U)
            u += (r + 𝒫.γ*u′) / m
        end
        if u > best.u
```

```
            best = (a=a, u=u)
        end
    end
    return best
end

(π::SparseSampling)(s) = sparse_sampling(π.𝒫, s, π.d, π.m, π.U).a
```

9.6. Поиск по дереву Монте-Карло

Поиск по дереву Монте-Карло (Monte Carlo tree search, алгоритм 9.5) позволяет избежать экспоненциального роста сложности, запуская m моделей из текущего состояния[4]. Во время этих моделирований алгоритм обновляет расчетные значения функции полезности действия $Q(s, a)$ и значение количества раз, когда была выбрана конкретная пара состояние-действие $N(s, a)$. После выполнения этих m моделирований из текущего состояния s мы просто выбираем действие, которое ведет к максимальной оценке $Q(s, a)$.

Алгоритм 9.5. Стратегия поиска по дереву Монте-Карло для нахождения приблизительно оптимального действия из текущего состояния `s`

```
struct MonteCarloTreeSearch
    𝒫 # задача
    N # счетчик посещений
    Q # расчетные значения полезности действия
    d # глубина
    m # количество моделирований
    c # постоянная поиска
    U # расчетное значение функции полезности
end

function (π::MonteCarloTreeSearch)(s)
    for k in 1:π.m
        simulate!(π, s)
    end
    return argmax(a->π.Q[(s,a)], π.𝒫.𝒜)
end
```

Моделирование начинается с обхода исследуемого пространства состояний, образованного состояниями, для которых у нас есть оценки Q и N (алгоритм 9.6). Мы следуем стратегии исследования, чтобы выбрать действия из

[4] См. обзор C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, «A Survey of Monte Carlo Tree Search Methods», IEEE Transactions on Computational Intelligence and AI in Games, vol. 4, no. 1, pp. 1–43, 2012.

различных состояний. Общий подход заключается в выборе действия, которое максимизирует *эвристику исследования* UCB_1[5]:

$$Q(s,a)+c\sqrt{\frac{\log N(s)}{N(s,a)}}, \tag{9.1}$$

где $N(s) = \sum_a N(s,a)$ – общее количество посещений до s и c – параметр исследования, который масштабирует значение полезности неисследованных действий. Второй член фактически является *вознаграждением за исследование* (exploration bonus). Если $N(s, a) = 0$, вознаграждение становится бесконечным. Поскольку $N(s, a)$ находится в знаменателе, вознаграждение выше для действий, которые предпринимались реже. Алгоритм 9.7 реализует эту стратегию исследования. Мы обсудим многие другие стратегии исследования позже, в главе 15.

Алгоритм 9.6. Метод моделирования поиска по дереву Монте-Карло, начиная с состояния `s` на глубину `d`

```
function simulate!(π::MonteCarloTreeSearch, s, d=π.d)
    if d ≤ 0
        return π.U(s)
    end
    𝒫, N, Q, c = π.𝒫, π.N, π.Q, π.c
    𝒜, TR, γ = 𝒫.𝒜, 𝒫.TR, 𝒫.γ
    if !haskey(N, (s, first(𝒜)))
        for a in 𝒜
            N[(s,a)] = 0
            Q[(s,a)] = 0.0
        end
        return π.U(s)
    end
    a = explore(π, s)
    s′, r = TR(s,a)
    q = r + γ*simulate!(π, s′, d-1)
    N[(s,a)] += 1
    Q[(s,a)] += (q-Q[(s,a)])/N[(s,a)]
    return q
end
```

Когда мы предпринимаем действия, указанные в алгоритме 9.7, мы переходим в новые состояния, выбранные из генеративной модели $T(s, a)$, аналогично методу разреженной выборки. Мы увеличиваем счетчик посещений $N(s, a)$ и обновляем $Q(s, a)$, чтобы сохранить среднее значение.

[5] UCB означает верхнюю доверительную границу (upper confidence border). Это одна из многих стратегий, обсуждаемых в P. Auer, N. Cesa-Bianchi, and P. Fischer, *Finite-Time Analysis of the Multiarmed Bandit Problem*, Machine Learning, vol. 47, no. 2–3, pp. 235–256, 2002. Уравнение получено из границы Чернова–Хёффдинга.

Алгоритм 9.7. Стратегия исследования, используемая при поиске по дереву Монте-Карло при определении узлов, которые необходимо пройти. Стратегия определяется словарем подсчетов посещений состояний-действий `N` и значений `Q`, а также параметром исследования `c`. Когда `N[(s,a)] = 0`, стратегия возвращает бесконечность

```
bonus(Nsa, Ns) = Nsa == 0 ? Inf : sqrt(log(Ns)/Nsa)

function explore(π::MonteCarloTreeSearch, s)
    𝒜, N, Q, c = π.𝒫.𝒜, π.N, π.Q, π.c
    Ns = sum(N[(s,a)] for a in 𝒜)
    return argmax(a->Q[(s,a)] + c*bonus(N[(s,a)], Ns), 𝒜)
end
```

В какой-то момент мы достигнем либо максимальной глубины, либо состояния, которое еще не исследовали. Если мы достигаем неисследованного состояния s, мы инициализируем $N(s, a)$ и $Q(s, a)$ нулевыми значениями для каждого действия a. Мы можем изменить алгоритм 9.6, чтобы инициализировать их другими значениями, основанными на предварительном экспертном знании проблемы. После инициализации N и Q мы возвращаем расчетное значение полезности в состоянии s. Обычно это значение оценивается путем некоторой стратегии развертывания с использованием процесса, описанного в разделе 9.2.

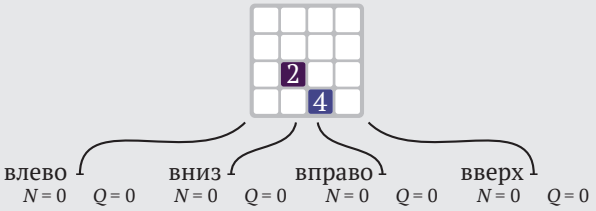
Примеры с 9.3 по 9.7 иллюстрируют поиск по дереву Монте-Карло применительно к игре под названием «2048». На рис. 9.4 показано дерево, созданное путем выполнения поиска по дереву Монте-Карло в игре «2048». В примере 9.8 рассмотрено влияние использования различных стратегий на расчетные значения полезности.

Существуют разновидности базового алгоритма поиска по дереву Монте-Карло, которые могут лучше обрабатывать большие пространства действий и состояний. Вместо развертывания всех действий мы можем использовать *монотонно возрастающее расширение* (progressive widening). Количество действий, рассматриваемых в состоянии s, ограничено $\theta_1 N(s)^{\theta_2}$, где θ_1 и θ_2 – гиперпараметры. Точно так же мы можем ограничить количество состояний, возникающих в результате выполнения действия a из состояния s, используя так называемое *двойное возрастающее расширение* (double progressive widening). Если количество состояний, которые были смоделированы из состояния s после действия a, меньше $\theta_3 N(s,a)^{\theta_4}$, то мы выбираем новое состояние; в противном случае мы выбираем одно из ранее выбранных состояний с вероятностью, пропорциональной количеству посещений. Этот подход можно использовать для обработки как больших областей, так и непрерывных пространств действий и состояний[6].

[6] A. Couëtoux, J.-B. Hoock, N. Sokolovska, O. Teytaud, N. Bonnard, *Continuous Upper Confidence Trees*, in Learning and Intelligent Optimization (LION), 2011.

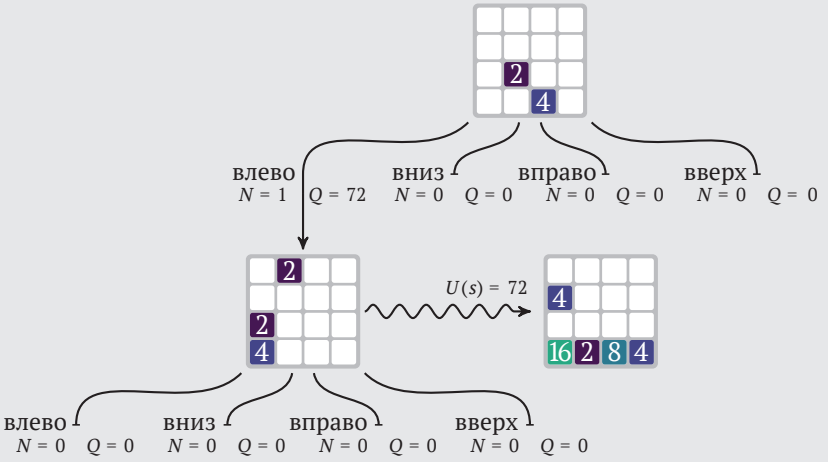
Пример 9.3. Пример поиска решения в игре «2048» с поиском по дереву Монте-Карло

Рассмотрим использование поиска по дереву Монте-Карло для воспроизведения игры 2048 (приложение F.2) с максимальной глубиной $d = 10$, параметром исследования $c = 100$ и 10-шаговым случайным развертыванием для вычисления $U(s)$. Наше первое моделирование расширяет начальное состояние. Счетчик и значение полезности инициализируются для каждого действия из начального состояния:



Пример 9.4. Пример поиска решения в игре «2048» с поиском по дереву Монте-Карло (продолжение)

Второе моделирование начинается с выбора наилучшего действия из начального состояния в соответствии с нашей стратегией исследования в уравнении (9.1). Поскольку все состояния имеют одинаковое значение, мы произвольно выбираем в качестве первого действия `left` (влево). Затем выбираем новое состояние-преемник и обновляем его, инициализируя связанные счетчики и значения полезности. Развертывание запускается из состояния-преемника и используется для обновления полезности состояния `left`:

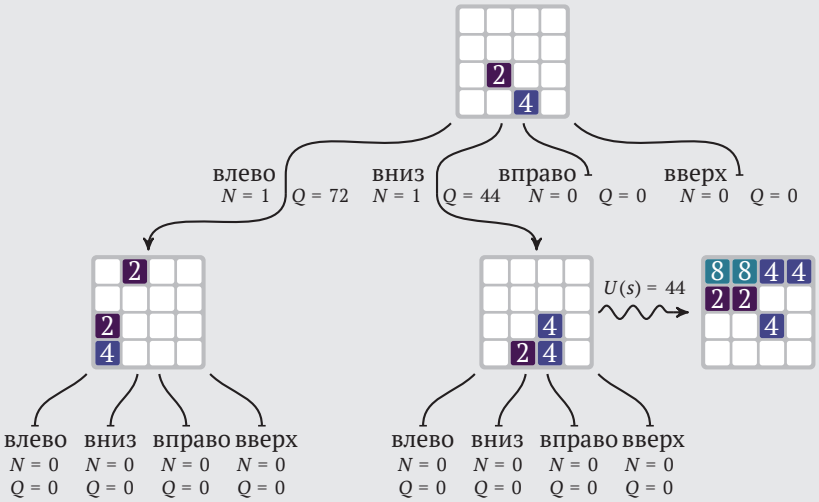


Пример 9.5. Пример поиска решения в игре «2048» с поиском по дереву Монте-Карло (продолжение)

Третье моделирование начинается с выбора второго действия down (вниз), потому что оно имеет бесконечную полезность из-за бонуса, который дается за неисследованные действия. Первое действие имеет конечную полезность:

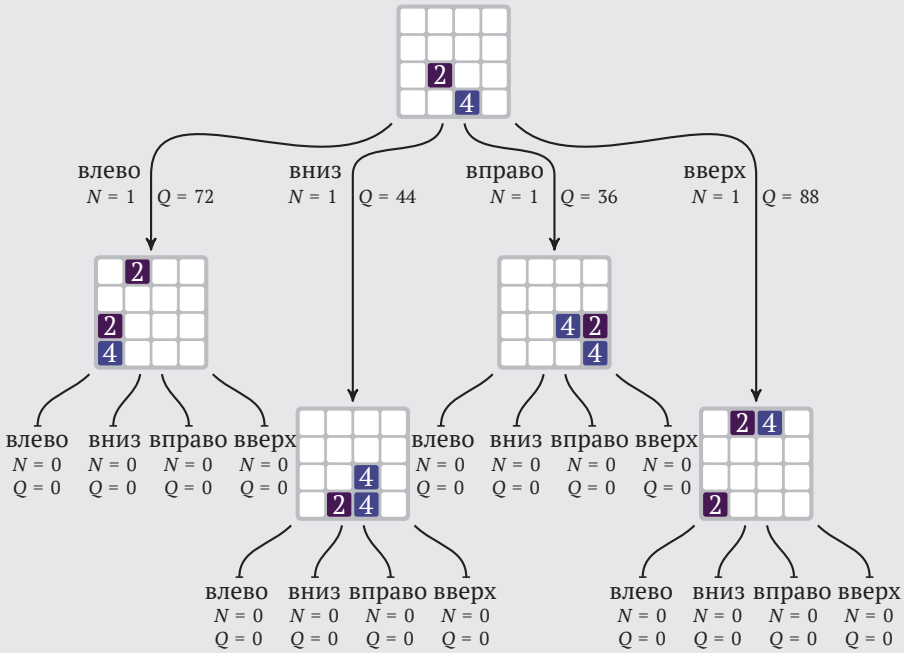
$$Q(s_0, \text{влево}) + c\sqrt{\frac{\log N(s_0)}{N(s_0, \text{влево})}} = 72 + 100\sqrt{\frac{\log 2}{1}} \approx 155.255.$$

Мы выполняем действие down и пробуем обновленное состояние-преемник. Развертывание выполняется из состояния-преемника, и его значение используется для обновления полезности down:



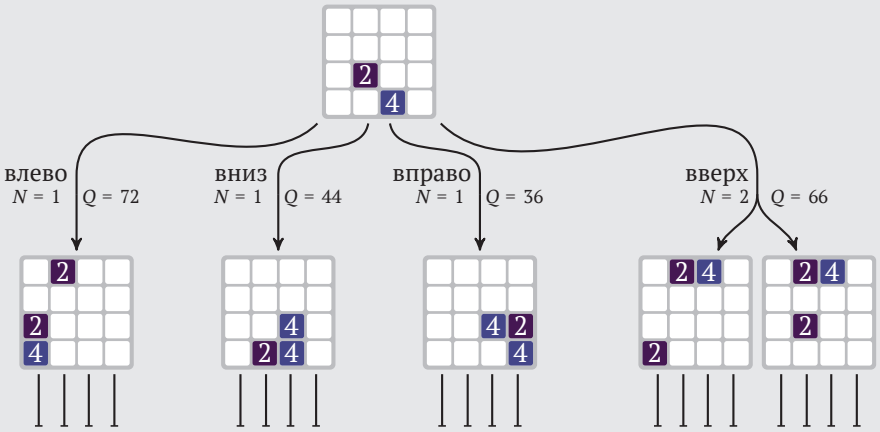
Пример 9.6. Пример поиска решения в игре «2048» с поиском по дереву Монте-Карло (продолжение)

Следующие два моделирования выбирают действия `right` (вправо) и `up` (вверх) соответственно. Это дает нам новое дерево состояний:



Пример 9.7. Пример поиска решения в игре «2048» с поиском по дереву Монте-Карло (продолжение)

В пятом моделировании наибольшее значение полезности дает действие up (вверх). Состояние-преемник после выбора up не обязательно будет таким же, как в предыдущий раз. Мы находим $U(s) = 44$ и обновляем количество посещений до 2, а полезность действия до $Q \leftarrow 88 + (44 - 88)/2 = 66$. Создан новый узел-преемник:



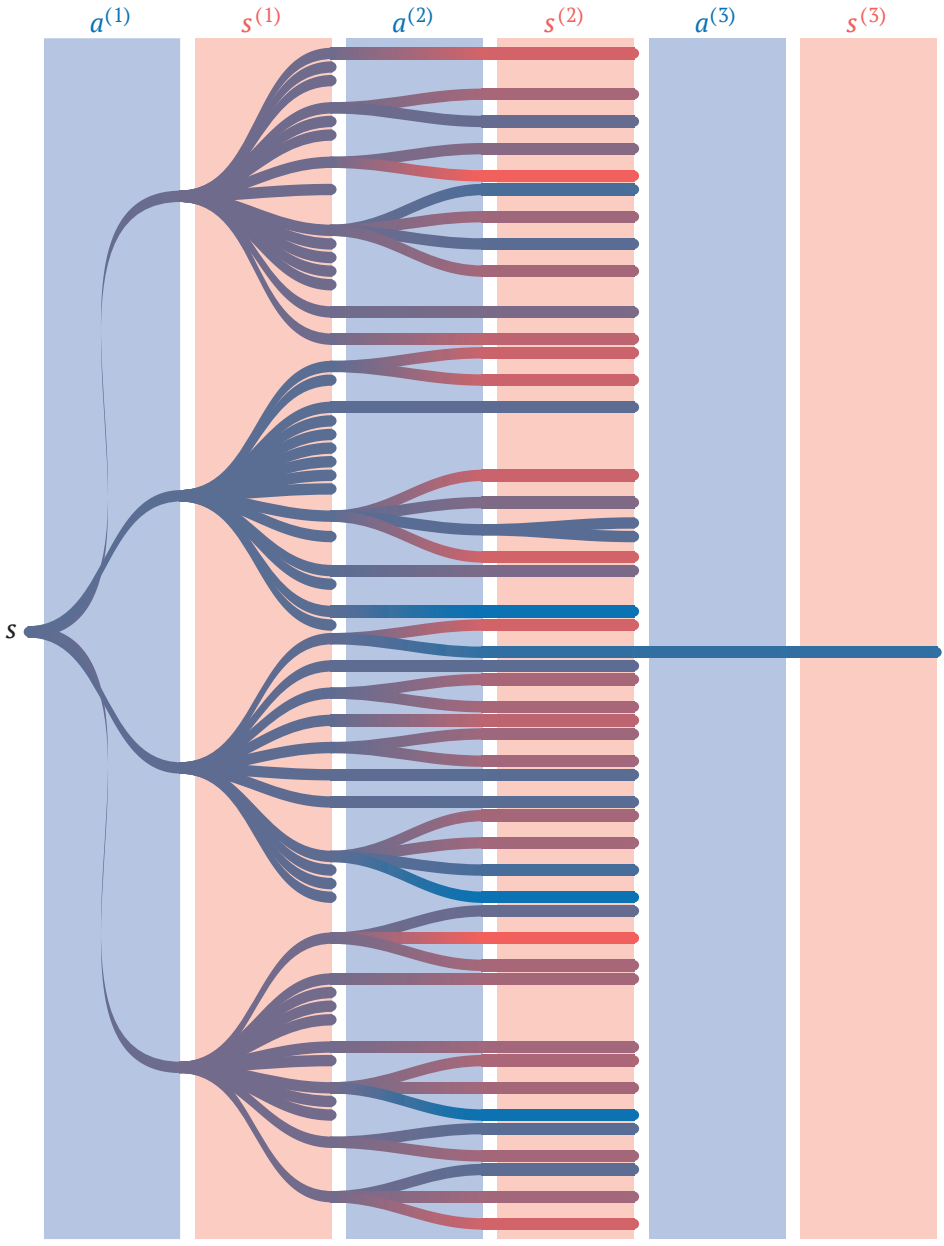


Рис. 9.4. Поиск по дереву Монте-Карло в задаче игры «2048» после 100 моделирований. Как правило, поиск по дереву Монте-Карло для MDP создает граф поиска, потому что может быть несколько способов достичь одного и того же состояния. Цвета в дереве обозначают значения полезности в узлах: высокие значения показаны синим цветом, а низкие – красным. Дерево неглубокое, с довольно высоким коэффициентом ветвления, потому что игра «2048» имеет множество достижимых состояний для каждого действия

Пример 9.8. Производительность поиска по дереву Монте-Карло зависит от количества моделирований и метода оценки игрового поля. Эвристические функции оценки игрового поля, как правило, могут более эффективно направлять поиск, когда количество проходов невелико. Оценки по алгоритму развертывания с пошаговым предпросмотром занимают примерно в 18 раз больше времени, чем эвристические оценки

Развертывание – не единственное средство, с помощью которого мы можем оценить полезность в дереве поиска по дереву Монте-Карло. Для конкретных задач часто удается создать пользовательские функции. Например, мы можем побудить поиск по дереву Монте-Карло упорядочить плитки в игре «2048», используя функции, которые возвращают взвешенную сумму значений плиток:

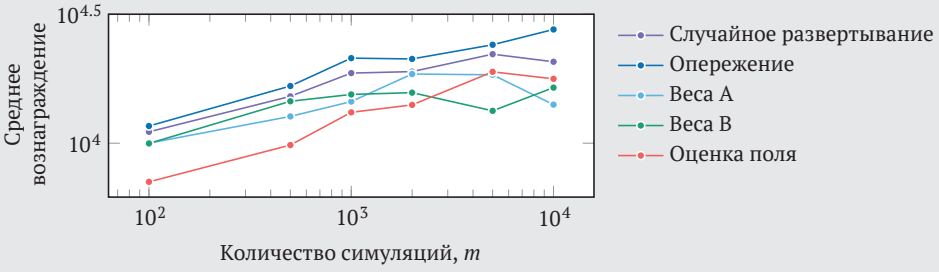
1	2	3	4
2	3	4	5
3	4	5	6
4	5	6	7

Эвристические веса A

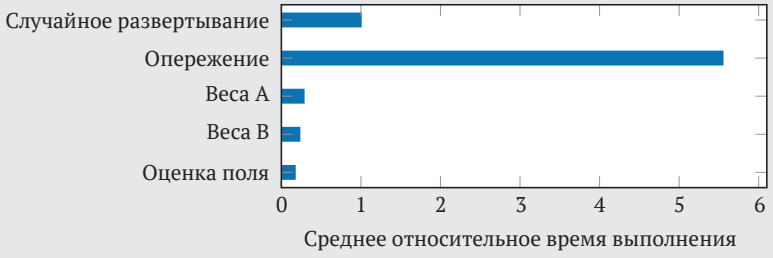
0	1	2	3
7	6	5	4
8	9	10	11
15	14	13	12

Эвристические веса B

На графике ниже представлено сравнение поиска по дереву Монте-Карло в игре «2048» с использованием развертываний с равномерной случайной стратегией, развертываний с пошаговым предпросмотром, двух оценочных функций и с использованием текущей оценки игрового поля:



Развертывания работают хорошо, но требуют больше времени на выполнение. Ниже изображена диаграмма среднего времени выполнения относительно случайных развертываний для $m = 100$ из начального состояния:



9.7. Эвристический поиск

Эвристический поиск (алгоритм 9.8) использует m моделирований жадной стратегии относительно функции полезности U из текущего состояния s[7]. Функция полезности U определена до верхней границы функции полезности $\bar{U}$, которая называется *эвристикой*. Выполняя эти моделирования, мы обновляем нашу оценку U с помощью метода упреждения. После запуска этих моделирований мы просто выбираем жадное действие из s, ориентируясь на U. На рис. 9.5 показано, как U и жадная стратегия меняются в зависимости от количества моделирований.

Алгоритм 9.8. Эвристический поиск запускает `m` моделирований, начиная с начального состояния `s` до глубины `d`. Поиск направляется эвристической функцией начальной полезности `Uhi`, которая сходится к оптимальной в рамках моделирования, если есть верхняя граница оптимальной функции полезности

```
struct HeuristicSearch
    𝒫  # задача
    Uhi # верхняя граница функции полезности
    d   # глубина
    m   # количество моделирований
end

function simulate!(π::HeuristicSearch, U, s)
    𝒫 = π.𝒫
    for d in 1:π.d
        a, u = greedy(𝒫, U, s)
        U[s] = u
        s = rand(𝒫.T(s, a))
    end
end

function (π::HeuristicSearch)(s)
    U = [π.Uhi(s) for s in π.𝒫.𝒮]
    for i in 1:π.m
        simulate!(π, U, s)
    end
    return greedy(π.𝒫, U, s).a
end
```

[7] A. G. Barto, S. J. Bradtke, S. P. Singh, *Learning to Act Using Real-Time Dynamic Programming*, Artificial Intelligence, vol. 72, no. 1–2, pp. 81–138, 1995. Другие формы эвристического поиска рассмотрены в Mausam, A. Kolobov, *Planning with Markov Decision Processes: An AI Perspective*. Morgan & Claypool, 2012.

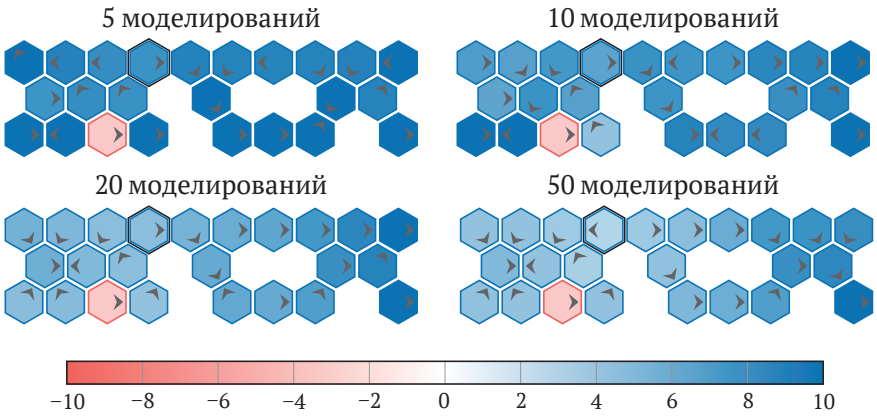


Рис. 9.5. Эвристический поиск выполняет моделирование с операторами Беллмана, чтобы улучшить функцию полезности в задаче гексамира и выстроить стратегию из начального состояния, показанного здесь с дополнительной черной рамкой. Моделирования выполняются до глубины 8 с эвристикой $\bar{U}(s) = 10$. Каждая плитка окрашена в соответствии со значением функции полезности в этой итерации. Мы видим, что алгоритм в итоге находит оптимальную стратегию

Эвристический поиск гарантированно сходится к оптимальной функции полезности, если эвристика $\bar{U}$ действительно является верхней границей функции полезности[8]. Эффективность поиска зависит от строгости верхней границы. К сожалению, на практике трудно получить строгие границы. Хотя эвристика, которая не является истинной верхней границей, может не сойтись к оптимальной стратегии, не исключено, что она сойдется к достаточно хорошо работающей стратегии. Сложность в контексте времени вычислений составляет $O(m \times d \times |\mathcal{S}| \times |\mathcal{A}|)$.

9.8. Эвристический поиск с разметкой

Эвристический поиск с разметкой (алгоритм 9.9) – это разновидность эвристического поиска, которая выполняет моделирование с обновлением значений, при этом состояния помечаются в зависимости от того, решены ли они[9]. Мы говорим, что состояние s *решено* (solved), если его остаток полезности не превышает порог $\delta > 0$:

$$|U_{k+1}(s) - U_k(s)| < \delta. \tag{9.2}$$

[8] Такая эвристика называется *допустимой эвристикой* (admissible heuristic).

[9] B. Bonet, H. Geffner, *Labeled RTDP: Improving the Convergence of Real-Time Dynamic Programming*, in International Conference on Automated Planning and Scheduling (ICAPS), 2003.

Иными словами, мы считаем, что состояние s решено, если дальнейшее совершенствование стратегии не приведет к заметному увеличению полезности.

Мы выполняем моделирование с обновлением полезности до тех пор, пока не будет решено текущее состояние. В отличие от эвристического поиска в предыдущем разделе, который выполняет фиксированное количество итераций, этот процесс фокусирует вычислительные усилия на наиболее важных областях пространства состояний.

Алгоритм 9.9. Эвристический поиск с разметкой, который выполняет моделирование, начиная с текущего состояния до глубины `d`, пока текущее состояние не станет решенным. Поиск управляется эвристической верхней границей функции полезности `Uhi` и хранит растущий набор решенных состояний. Состояния считаются решенными, когда их остатки полезности меньше δ. Алгоритм возвращает стратегию относительно максимума функции полезности

```
struct LabeledHeuristicSearch
    𝒫   # задача
    Uhi # верхняя граница функции полезности
    d   # глубина
    δ   # пороговое значение разрыва
end

function (π::LabeledHeuristicSearch)(s)
    U, solved = [π.Uhi(s) for s in 𝒫.𝒮], Set()
    while s ∉ solved
        simulate!(π, U, solved, s)
    end
    return greedy(π.𝒫, U, s).a
end
```

Моделирование в эвристическом поиске с разметкой (алгоритм 9.10) начинается с прогона до максимальной глубины d, следуя стратегии, которая является жадной по отношению к нашей расчетной функции полезности U, подобно эвристическому поиску в предыдущем разделе. Мы можем остановить моделирование до достижения глубины d, если достигнем состояния, которое было помечено как решенное в предыдущем моделировании.

После каждого моделирования мы перебираем состояния, которые посетили во время этого моделирования, в обратном порядке, выполняя процедуру разметки для каждого состояния и останавливаясь, если обнаруживается состояние, которое не решено. Процедура маркировки (алгоритм 9.11) ищет состояния в *жадной оболочке* (greedy envelope) s, которая определяется как состояния, достижимые из s при жадной стратегии по отношению к U. Состояние s считается нерешенным, если в жадной оболочке есть состояние, остаточная полезность которого больше порога δ. Если такое состояние не найдено, то помечаются решенными само состояние s, а также все состояния в жадной оболочке s, потому что они также должны были сойтись. Если найдено состояние

с достаточно большим остатком полезности, то обновляются полезности всех состояний, пройденных при поиске жадной оболочки.

Алгоритм 9.10. Моделирование выполняется от текущего состояния до максимальной глубины `d`. Мы останавливаем моделирование на глубине `d` или если встречаем состояние, которое находится в наборе `solved`. После моделирования вызываем `label!`, проходя в обратном порядке по состояниям, которые мы посетили

```
function simulate!(π::LabeledHeuristicSearch, U, solved, s)
    visited = []
    for d in 1:π.d
        if s ∈ solved
            break
        end
        push!(visited, s)
        a, u = greedy(π.𝒫, U, s)
        U[s] = u
        s = rand(π.𝒫.T(s, a))
    end
    while !isempty(visited)
        if label!(π, U, solved, pop!(visited))
            break
        end
    end
end
```

Алгоритм 9.11. Функция `label!` пытается найти в жадной оболочке `s` такое состояние, остаток полезности которого превышает пороговое значение `δ`. Функция `expand` вычисляет жадную оболочку s и определяет, имеют ли какие-либо из состояний оболочки остатки полезности выше порога. Если состояние дает остаток, превышающий порог, то мы обновляем полезности состояний в оболочке. В противном случае добавляем эту оболочку в набор решенных состояний

```
function expand(π::LabeledHeuristicSearch, U, solved, s)
    𝒫, δ = π.𝒫, π.δ
    𝒮, 𝒜, T = 𝒫.𝒮, 𝒫.𝒜, 𝒫.T
    found, toexpand, envelope = false, Set(s), []
    while !isempty(toexpand)
        s = pop!(toexpand)
        push!(envelope, s)
        a, u = greedy(𝒫, U, s)
        if abs(U[s] - u) > δ
            found = true
        else
            for s′ in 𝒮
```

```
                if T(s,a,s′) > 0 && s′ ∉ (solved ∪ envelope)
                    push!(toexpand, s′)
                end
            end
        end
    end
    return (found, envelope)
end

function label!(π::LabeledHeuristicSearch, U, solved, s)
    if s ∈ solved
        return false
    end
    found, envelope = expand(π, U, solved, s)
    if found
        for s ∈ reverse(envelope)
            U[s] = greedy(π.𝒫, U, s).u
        end
    else
        union!(solved, envelope)
    end
    return found
end
```

На рис. 9.6 показано несколько различных жадных оболочек. На рис. 9.7 представлены состояния, пройденные за одну итерацию эвристического поиска с разметкой. На рис. 9.8 показана эволюция эвристического поиска в задаче гексамира.

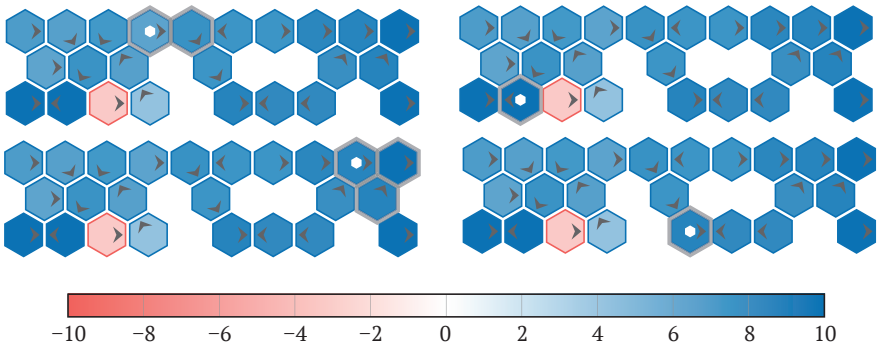


Рис. 9.6. Жадная оболочка при $\delta = 1$ и нескольких состояниях; визуализация для функции полезности в задаче гексамира. Функция полезности была получена путем прогона базового эвристического поиска в течение 10 итераций от начального состояния, показанного плиткой с белым центром, с максимальной глубиной 8. Мы обнаружили, что размер жадной оболочки, обведенной серым, может широко варьироваться в зависимости от состояния

Рис. 9.7. Одиночная итерация эвристического поиска с разметкой выполняет исследовательский прогон (стрелки), за которым следует разметка (шестиугольная рамка). В этой итерации помечены только два состояния: скрытое конечное состояние и состояние с шестиугольной рамкой. И исследовательский прогон, и шаг разметки обновляют функцию полезности

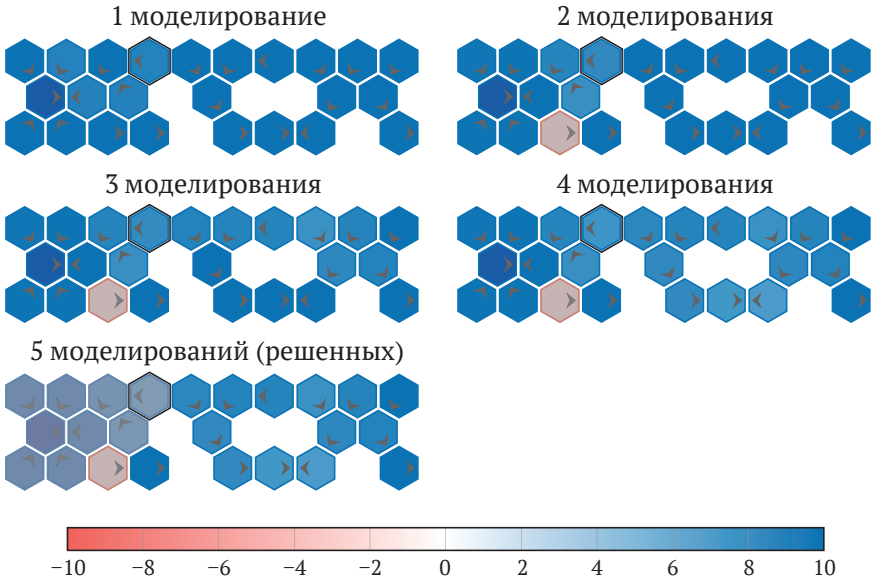


Рис. 9.8. Последовательность эвристического поиска в задаче гексамира с использованием $\delta = 1$ и эвристики $\bar{U}(s) = 10$. Решенные состояния на каждой итерации покрыты серой заливкой. Набор решенных состояний растет по направлению от конечного состояния вознаграждения обратно к начальному состоянию с темной рамкой

9.9. Планирование с открытым контуром

Онлайн-методы поиска, обсуждавшиеся в этой главе, а также офлайн-методы, рассмотренные в предыдущих главах, являются примерами *планирования с замкнутым контуром* (closed-loop planning, или с обратной связью), предусматривающего использование информации о будущем состоянии в процессе планирования[10]. Но зачастую *планирование с разомкнутым контуром* (open-loop planning, или без обратной связи) может обеспечить удовлетворительную аппроксимацию оптимального плана с замкнутым контуром, в то же время значительно повысить эффективность вычислений, избегая необходимости рассуждать о получении предстоящей информации. Иногда этот подход называют *модельным прогнозирующим управлением* (model predictive control)[11]. Как и в случае с управлением по методу отступающего горизонта, модельное прогнозирующее управление решает задачу разомкнутого контура, выполняет действие из текущего состояния, переходит в следующее состояние, а затем строит новый план.

Планирование с разомкнутым контуром можно представить в виде последовательности действий до глубины d. Процесс планирования сводится к задаче оптимизации:

$$\max_{a_{1:d}} U(a_{1:d}), \tag{9.3}$$

где $U(a_{1:d})$ – ожидаемое вознаграждение при выполнении последовательности действий $a_{1:d}$. В зависимости от приложения эта задача оптимизации может быть выпуклой или поддаваться выпуклой аппроксимации, что означает, что ее можно быстро решить с использованием различных алгоритмов[12]. Позже в этом разделе мы обсудим несколько различных формул, которые можно использовать для преобразования уравнения (9.3) в задачу выпуклой оптимизации.

Планирование с разомкнутым контуром часто позволяет разработать эффективные стратегии принятия решений в многомерных пространствах, где планирование с закрытым контуром невыполнимо с вычислительной точки зрения. Вычислительная эффективность этого типа планирования достигается за счет того, что не учитывается будущая информация. В примере 9.9 показано, как планирование с разомкнутым контуром может привести к неправильным решениям, даже если мы учитываем стохастичность.

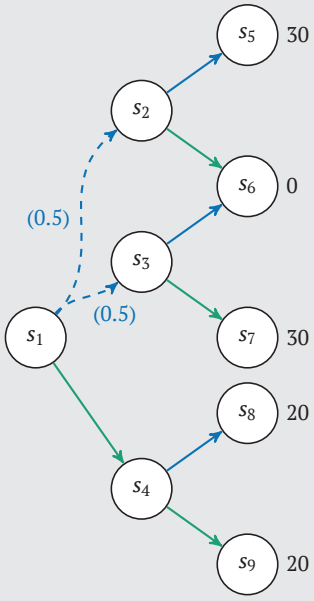
---

[10] В этом контексте мы говорим о цикле «наблюдение-действие», представленном в разделе 1.1.

[11] F. Borrelli, A. Bemporad, M. Morari, *Predictive Control for Linear and Hybrid Systems*. Cambridge University Press, 2019.

[12] В приложении А.6 рассматривается выпуклость. Введение в выпуклую оптимизацию предоставлено в книге S. Boyd, L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.

Пример 9.9. Риск ложной оптимальности в планировании с разомкнутым контуром



Рассмотрим задачу с девятью состояниями, как показано на рисунке справа, с двумя шагами решения, исходя из начального состояния s_1. В наших решениях мы должны выбирать между подъемом (синие стрелки) и спуском (зеленые стрелки). Эффекты этих действий детерминированы, за исключением того, что если мы поднимаемся от s_1, то в половине случаев оказываемся в состоянии s_2, а в другой половине случаев – в состоянии s_3. Мы получаем вознаграждение в размере 30 в состояниях s_5 и s_7 и вознаграждение в размере 20 в состояниях s_8 и s_9, как показано на рисунке.

Существует ровно четыре плана с разомкнутым контуром: (вверх, вверх), (вверх, вниз), (вниз, вверх) и (вниз, вниз). В этом простом примере легко вычислить их ожидаемую полезность:

- U(вверх, вверх) = $0.5 \times 30 + 0.5 \times 0 = 15$;
- U(вверх, вниз) = $0.5 \times 0 + 0.5 \times 30 = 15$;
- U(вниз, вверх) = 20;
- U(вниз, вниз) = 20.

В соответствии с набором планов с разомкнутым контуром лучше всего пойти вниз из s_1, потому что ожидаемое вознаграждение составит 20 вместо 15.

Планирование с закрытым контуром, напротив, учитывает тот факт, что мы можем основывать следующее решение на наблюдаемом результате первого действия. Если мы решим пойти из s_1 вверх, то далее мы можем

пойти вниз или вверх в зависимости от того, окажемся ли мы в s_2 или s_3 (т. е. с учетом будущей информации), тем самым гарантируя вознаграждение в размере 30.

9.9.1. Прогнозирующее управление с детерминированной моделью

Распространенным приближением, позволяющим сделать функцию $U(a_{1:d})$ поддающейся оптимизации, является предположение о детерминированной динамике:

$$\max_{a_{1:d},\, s_{2:d}} \sum_{t=1}^{d} \gamma^t R(s_t, a_t), \tag{9.4}$$

при условии что $s_{t+1} = T(s_t, a_t)$, $t \in 1{:}d - 1$,

где s_1 – текущее состояние, а $T(s, a)$ – детерминированная функция перехода, которая возвращает состояние, возникающее в результате выполнения действия a из состояния s. Распространенной стратегией получения подходящей детерминированной функции перехода из стохастической функции перехода является использование наиболее вероятного перехода. Если динамика в уравнении (9.4) линейна, а функция вознаграждения выпукла, то задача выпукла.

В примере 9.10 представлен пример, иллюстрирующий продвижение к целевому состоянию при обходе препятствия и стремлении к минимизации усилия ускорения. И пространство состояний, и пространство действий непрерывны, и мы можем найти решение менее чем за секунду. Повторное планирование после каждого шага способно скомпенсировать стохастичность или неожиданные события. Например, если препятствие перемещается, мы можем скорректировать наш план, как показано на рис. 9.9.

Пример 9.10. Планирование с открытым контуром в детерминированной среде. Мы пытаемся найти путь вокруг круглого препятствия. Эта реализация использует интерфейс `JuMP.jl` для решателя Ipopt. На основе статьи A. Wächter, L. T. Biegler, *On the Implementation of an Interior-Point Filter Line-Search Algorithm for Large-Scale Nonlinear Programming*, Mathematical Programming, vol. 106, no. 1, pp. 25–57, 2005

В этой задаче состояние **s** представляет двумерное положение агента, связанное с его двумерным вектором скорости, при этом **s** изначально инициализируется значениями [0, 0, 0, 0]. Наше действие **a** представляет собой вектор ускорения, где каждый компонент должен находиться в пределах ±1. На каждом шаге мы используем действие для обновления скорости, а затем скорость для обновления положения агента. Наша цель – достичь целевого состояния $\mathbf{s}_{\text{goal}}$ = [10, 10, 0, 0]. Мы планируем до глубины d = 10 шагов без дисконтирования. С каждым шагом мы накапливаем стоимость $||\mathbf{a}_t||_2^2$, чтобы свести к минимуму усилие ускорения. Мы стремимся на последнем шаге оказаться как можно ближе к целевому состоянию со штрафом $100||\mathbf{s}_d - \mathbf{s}_{\text{goal}}||_2^2$. На пути к цели мы должны избегать круглого препятствия с радиусом 2 и центром в точке [3, 4]. Эту задачу можно сформулировать следующим образом и выделить из плана первое действие:

```
model = Model(Ipopt.Optimizer)
d = 10
current_state = zeros(4)
goal = [10,10,0,0]
obstacle = [3,4]
@variables model begin
    s[1:4, 1:d]
    -1 ≤ a[1:2,1:d] ≤ 1
end
# velocity update
@constraint(model, [i=2:d,j=1:2], s[2+j,i] == s[2+j,i-1] + a[j,i-1])
# position update
@constraint(model, [i=2:d,j=1:2], s[j,i] == s[j,i-1] + s[2+j,i-1])
# initial condition
@constraint(model, s[:,1] .== current_state)
# obstacle
@constraint(model, [i=1:d], sum((s[1:2,i] - obstacle).^2) ≥ 4)
@objective(model, Min, 100*sum((s[:,d] - goal).^2) + sum(a.^2))
optimize!(model)
action = value.(a[:,1])
```

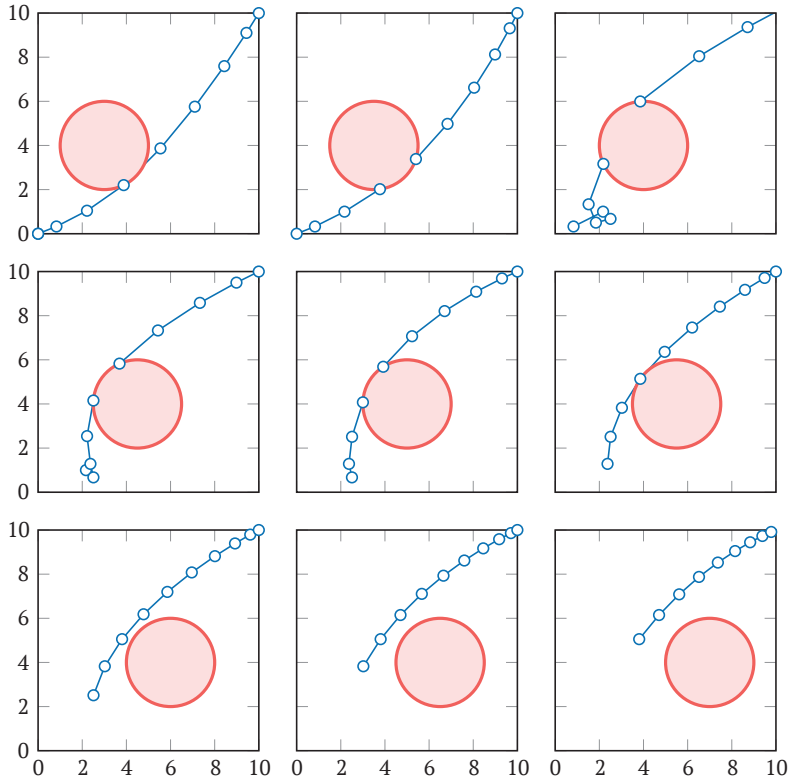


Рис. 9.9. Модель прогнозирующего управления применена к задаче из примера 9.10 с добавлением движущегося препятствия. Последовательность шагов отображена в клетках рисунка слева направо и сверху вниз. Изначально у нас есть план, который предусматривает движение справа от препятствия, но в третьей клетке мы видим, что должны сменить стратегию и пройти слева. Нам нужно немного маневрировать, чтобы соответствующим образом скорректировать вектор скорости с минимальными усилиями. Конечно, мы могли бы создать лучший путь (с точки зрения функции полезности), если бы в процессе планирования было известно, что препятствие движется в определенном направлении

9.9.2. Робастное прогностическое управление

Можно изменить постановку задачи, чтобы обеспечить робастность (устойчивость) к неопределенности вывода. Существует множество способов построения *робастного прогностического управления моделью* (robust model predictive control)[13], но только один из них подразумевает выбор наилучшего плана с ра-

[13] A. Bemporad, M. Morari, *Robust Model Predictive Control: A Survey*, in Robustness in Identification and Control, A. Garulli, A. Tesi, A. Vicino, eds., Springer, 1999, pp. 207–226.

зомкнутым контуром с учетом наихудших переходов состояний. Этот подход определяет $T(s, a)$ как *набор неопределенностей* (uncertainty set), состоящий из всех возможных состояний, которые могут возникнуть в результате выполнения действия a в состоянии s. Другими словами, множество неопределенностей является носителем распределения $T(\cdot|s, a)$. Оптимизация в отношении наихудших переходов состояний требует преобразования задачи оптимизации в уравнении (9.4) в *задачу о минимаксе*:

$$\max_{a_{1:d}} \min_{s_{2:d}} \sum_{t=1}^{d} \gamma^t R(s_t, a_t), \tag{9.5}$$

при условии что $s_{t+1} = T(s_t, a_t)$, $t \in 1{:}d - 1$.

К сожалению, такая постановка задачи может привести к крайне консервативному поведению. Если мы адаптируем пример 9.10 для моделирования неопределенности в движении препятствия, накопление неопределенности может стать довольно большим, даже при планировании с относительно коротким горизонтом. Один из способов уменьшить накопление неопределенности состоит в том, чтобы ограничить вывод набора неопределенностей $T(s, a)$ таким образом, чтобы он охватывал только, скажем, 95 % распределения вероятностей. Другая проблема заключается в том, что задача минимаксной оптимизации часто не является выпуклой и ее трудно решить.

9.9.3. Многовариантное прогностическое управление

Одним из способов решения проблемы избыточных вычислений в рамках минимаксной задачи в уравнении (9.5) является использование m сценариев прогноза, каждый из которых следует своей собственной детерминированной функции перехода[14]. Существуют различные подходы к реализации *многовариантного прогностического управления*, который является разновидностью *ретроспективной оптимизации* (hindsight optimization)[15]. Один из распространенных подходов заключается в том, чтобы сделать детерминированные функции перехода зависимыми от шага k, то есть $T_i(s, a, k)$, что равносильно увеличению размерности пространства состояний за счет добавления глубины. В примере 9.11 показано, как это можно сделать для линейной гауссовой модели.

---

[14] S. Garatti, M. C. Campi, *Modulating Robustness in Control Design: Principles and Algorithms*, IEEE Control Systems Magazine, vol. 33, no. 2, pp. 36–51, 2013.

[15] Этот подход называется *ретроспективной оптимизацией*, поскольку оптимизация решения выполняется с использованием знаний о результатах действий, которые можно получить только задним числом. E. K. P. Chong, R. L. Givan, H. S. Chang, *A Framework for Simulation-Based Network Control via Hindsight Optimization*, in IEEE Conference on Decision and Control (CDC), 2000.

Пример 9.11. Моделирование динамики линейного гауссова перехода методом многовариантного прогностического управления

Предположим, у нас есть задача с линейной гауссовой динамикой:

$T(\mathbf{s}'|\mathbf{s}, \mathbf{a}) = \mathcal{N}(\mathbf{T}_s\mathbf{s} + \mathbf{T}_a\mathbf{a}, \mathbf{\Sigma})$.

Задача на рис. 9.9 является линейной и не содержит неопределенности, но если мы позволим препятствию двигаться в соответствии с гауссовым распределением на каждом шаге, то динамика задачи станет линейной по Гауссу. Мы можем аппроксимировать динамику, используя набор из m сценариев прогноза, каждый из которых состоит из d шагов. Мы можем взять $m \times d$ выборок $\varepsilon_{ik} \sim \mathcal{N}(\mathbf{0}, \mathbf{\Sigma})$ и определить детерминированные функции перехода:

$\mathbf{T}_i(\mathbf{s}, \mathbf{a}, k) = \mathbf{T}_s\mathbf{s} + \mathbf{T}_a\mathbf{a} + \varepsilon_{ik}$.

Мы пытаемся найти наилучшую последовательность действий для худшего из выбранных сценариев:

$$\max_{a_{1:d}} \min_{i, s_{2:d}} \sum_{k=1}^{d} \gamma^k R(s_k, a_k), \tag{9.6}$$

при условии что $s_{k+1} = T(s_k, a_k, k), k \in 1{:}d-1$.

Эта задача решается намного проще, чем исходная робастная задача.

Мы также можем использовать многовариантный подход для оптимизации среднего случая[16]. Подход аналогичен обозначенному в уравнении (9.6), за исключением того, что мы заменяем минимизацию ожиданием и допускаем различные последовательности действий для разных сценариев с ограничением, согласно которому первое действие должно удовлетворять условию:

$$\max_{a_{1:d}^{(1:m)}, s_{2:d}^{(i)}} \frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{d} \gamma^k R(s_k^{(i)}, a_k^{(i)}), \tag{9.7}$$

при условии что $s_{k+1}^{(i)} = T_i(s_k^{(i)}, a_k^{(i)}, k), \quad k \in 1{:}d-1, i \in 1{:}m,$

$$a_1^{(i)} = a_1^{(j)}, \qquad i \in 1{:}m, j \in 1{:}m.$$

Этот подход может обеспечить робастное поведение, не будучи чрезмерно консервативным и сохраняя вычислительную устойчивость. Оба подхода, представленных в уравнениях (9.6) и (9.7), можно сделать более робастными, увеличив количество сценариев прогноза m за счет дополнительных вычислений.

---

[16] Этот подход был применен для оптимизации стратегии потоков мощности в работе N. Moehle, E. Busseti, S. Boyd, M. Wytock, *Dynamic Energy Management*, in Large Scale Optimization in Supply Chains and Smart Manufacturing, Springer, 2019, pp. 69–126.

9.10. Заключение

- Методы, основанные на режиме текущего времени, формируют план исходя из текущего состояния и фокусируя вычисления на достижимых состояниях.
- Планирование с отступающим горизонтом предполагает выработку плана до определенного горизонта, а затем перепланирование на каждом этапе.
- Метод предвидения с развертыванием подразумевает жадную стратегию в отношении функции полезности, найденной с помощью моделирования согласно стратегии развертывания; он более эффективен по сравнению с другими алгоритмами с точки зрения вычислений, но не дает гарантий производительности.
- Прямой поиск рассматривает все переходы состояние-действие до определенной глубины, что приводит к экспоненциальному росту вычислительной сложности как по количеству состояний, так и по количеству действий.
- Метод ветвей и границ использует функции верхней и нижней границ для отсечения частей дерева поиска, которые не приведут к ожидаемому лучшему результату.
- Разреженная выборка позволяет избежать экспоненциального роста количества состояний за счет ограничения количества выборочных переходов из каждого узла поиска.
- Поиск по дереву Монте-Карло направляет процесс к перспективным областям пространства поиска, обеспечивая разумный баланс между исследованием и использованием.
- Эвристический поиск выполняет моделирование стратегии, которая является жадной по отношению к функции полезности, обновляемой по мере продвижения вперед.
- Эвристический поиск с разметкой сокращает объем вычислений за счет пропуска состояний, значения которых сошлись.
- Планирование с разомкнутым контуром направлено на поиск наилучшей возможной последовательности действий и может быть вычислительно эффективным, если задача оптимизации является выпуклой.

9.11. Упражнения

Упражнение 9.1. Почему метод ветвей и границ имеет такую же вычислительную сложность в наихудшем случае, что и прямой поиск перебором?

Решение. В худшем случае никакие ветви дерева не будут отсечены, что приведет к обходу того же дерева, что и при прямом поиске, с той же вычислительной сложностью.

Упражнение 9.2. Если вам даны две допустимые эвристики h_1 и h_2, опишите, как вы могли бы использовать их обе в эвристическом поиске.

Решение. Создайте новую эвристику $h(s) = \min\{h_1(s), h_2(s)\}$ и используйте ее. Эта новая эвристика гарантированно допустима и не может давать худшую границу, чем h_1 или h_2. Из $h_1(s) \geqslant U^*(s)$ и $h_2(s) \geqslant U^*(s)$ следует, что $h(s) \geqslant U^*(s)$.

Упражнение 9.3. Если вам даны две недопустимые эвристики h_1 и h_2, опишите, как вы могли бы использовать их обе в эвристическом поиске.

Решение. Вы можете определить новую эвристику $h_3(s) = \max(h_1(s), h_2(s))$ для получения потенциально допустимой или «менее недопустимой» эвристики. Схождение может происходить медленнее, но возрастет шанс не упустить лучшее решение.

Упражнение 9.4. Предположим, у вас есть дискретный MDP с пространством состояний $\mathcal{S}$ и пространством действий $\mathcal{A}$, и вы хотите выполнить прямой поиск на глубину d. Из-за вычислительных ограничений и требования моделировать до глубины d вы решили сгенерировать новые, меньшие пространства состояний и действий путем повторной дискретизации исходных пространств состояний и действий в более грубом масштабе с $|\mathcal{S}'| < |\mathcal{S}|$ и $|\mathcal{A}'| < |\mathcal{A}|$. Каким должен быть размер новых пространств состояний и действий, чтобы сделать вычислительную сложность прямого поиска приблизительно инвариантной по глубине по отношению к сложности поиска по исходным пространствам состояний и действий $O(|\mathcal{S}||\mathcal{A}|)$?

Решение. Вам нужны новые пространства следующего размера по отношению к исходным:

$$|\mathcal{S}'| = |\mathcal{S}|^{\frac{1}{d}} \quad \text{и} \quad |\mathcal{A}'| = |\mathcal{A}|^{\frac{1}{d}}.$$

Убедимся, что сложность для новых пространств инвариантна по d:

$$O(|\mathcal{S}'|^d|\mathcal{A}'|^d) = O\left(\left(|\mathcal{S}|^{\frac{1}{d}}\right)^d\left(|\mathcal{A}|^{\frac{1}{d}}\right)^d\right) = O(|\mathcal{S}||\mathcal{A}|).$$

Упражнение 9.5. Основываясь на предыдущем упражнении, предположим теперь, что вы хотите сохранить все исходные действия в пространстве действий и повторно дискретизировать только пространство состояний. Каким должен быть размер нового пространства состояний, чтобы сделать вычислительную сложность прямого поиска приблизительно инвариантной по глубине относительно размера наших исходных пространств состояний и действий?

Решение. Вычислительная сложность прямого поиска определяется как $O((|\mathcal{S}||\mathcal{A}|)^d)$, что также можно записать в виде $O(|\mathcal{S}|^d|\mathcal{A}|^d)$. Таким образом, для того чтобы наше более грубое пространство состояний приводило к прямому поиску, который в вычислительном отношении приблизительно инвариантен по глубине относительно нашего исходного пространства состояний и действий, нам нужно соблюсти следующее равенство:

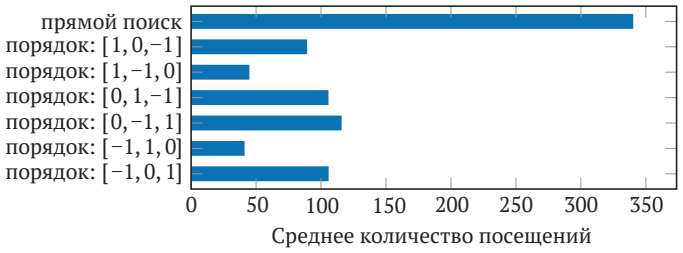
$$|\mathcal{S}'| = \left(\frac{|\mathcal{S}|}{|\mathcal{A}|^{d-1}} \right)^{\frac{1}{d}}.$$

Убедимся в инвариантности по d:

$$O(|\mathcal{S}'|^d |\mathcal{A}'|^d) = O\left(\left\lceil \left(\frac{|\mathcal{S}|}{|\mathcal{A}|^{d-1}} \right)^{\frac{1}{d}} \right\rceil^d |\mathcal{A}|^d \right) = O\left(|\mathcal{S}| \frac{|\mathcal{A}|^d}{|\mathcal{A}|^{d-1}} \right) = O(|\mathcal{S}||\mathcal{A}|).$$

Упражнение 9.6. Приведет ли изменение упорядочивания пространства действий к другим действиям при прямом поиске? Приведет ли изменение упорядочивания пространства действий к другим действиям при использовании метода ветвей и границ? Может ли упорядочивание пространства действий влиять на то, сколько состояний посещает метод ветвей и границ?

Решение. Прямой поиск перебирает все возможные будущие действия. Он может возвращать разные действия, если их ожидаемая полезность совпадает. Метод ветвей и границ обеспечивает ту же гарантию оптимальности на горизонте, что и прямой поиск с сортировкой по верхней границе. Разное упорядочивание пространства действий может повлиять на частоту посещения состояний методом ветвей и границ, когда верхняя граница дает одно и то же ожидаемое значение для двух или более действий. Ниже мы покажем этот эффект на модифицированной задаче о горной машине из примера 9.2. На графике сравнивается количество состояний, посещенных при прямом поиске и методом ветвей и границ для различных порядков действий с глубиной до 6. Метод ветвей и границ постоянно посещает гораздо меньше состояний, чем прямой поиск, но порядок действий все еще может влиять на посещение состояний.



Упражнение 9.7. Эквивалентна ли разреженная выборка с $m = |\mathcal{S}|$ прямому поиску?

Решение. Нет. Хотя вычислительные сложности одинаковы при $O(|\mathcal{S}|^d |\mathcal{A}|^d)$, прямой поиск будет разветвляться на все состояния в пространстве состояний, разреженная выборка будет разветвляться на $|\mathcal{S}|$ случайно выбранных состояний.

Упражнение 9.8. Пусть нам дан MDP с пространством состояний $|\mathcal{S}| = 10$, пространством действий $|\mathcal{A}| = 3$ и равномерным распределением вероятности перехода $T(s'|s, a) = 1/|\mathcal{S}|$ для всех s и a. Какова вероятность того, что метод разреженной выборки с $m = |\mathcal{S}|$ образцов и глубиной $d = 1$ дает дерево поиска, идентичное полученному при прямом поиске с глубиной $d = 1$?

Решение. Как при прямом поиске, так и согласно методу разреженной выборки мы переходим ко всем действиям из узла текущего состояния. Для прямого поиска в каждом из этих узлов действия мы переходим ко всем состояниям, а для разреженной выборки мы переходим к $m = |\mathcal{S}|$ выборочным состояниям. Если эти выборочные состояния в точности равны пространству состояний, эта ветка действия эквивалентна ветвям, созданным при прямом поиске. Таким образом, для одной ветки действия имеем:

вероятность того, что первое состояние уникально: $\frac{10}{10}$;

вероятность того, что второе состояние уникально (не равно первому состоянию): $\frac{9}{10}$;

вероятность того, что третье состояние уникально (не равно первому или второму состоянию): $\frac{8}{10}$;

...и так далее.

Поскольку каждое из выбранных состояний является независимым, из этого следует, что все уникальные состояния в пространстве состояний могут быть выбраны с вероятностью

$$\frac{10 \times 9 \times 8 \times \cdots}{10 \times 10 \times 10 \times \cdots} = \frac{10!}{10^{10}} \approx 0.000363.$$

Поскольку каждое из выбранных состояний в разных ветвях действия является независимым, вероятность того, что все три ветви действия выберут уникальные состояния в пространстве состояний, равна

$$\left(\frac{10!}{10^{10}}\right)^3 \approx (0.000363)^3 \approx 4.78 \times 10^{-11}.$$

Упражнение 9.9. Пусть даны таблицы $Q(s, a)$ и $N(s, a)$, приведенные ниже. Используя верхнюю доверительную границу в уравнении (9.1), найдите действие обходом каждого состояния дерева по методу Монте-Карло с параметром исследования $c_1 = 10$, а затем для $c_2 = 20$.

	$Q(s, a_1)$	$Q(s, a_2)$
s_1	10	−5
s_2	12	10

	$N(s, a_1)$	$N(s, a_2)$
s_1	27	4
s_2	32	18

Решение. Для первого параметра исследования $c_1 = 10$ мы заносим в таблицу верхнюю доверительную границу каждой пары состояние-действие и выбираем действие, максимизирующее границу для каждого состояния:

	$UCB(s, a_1)$	$UCB(s, a_2)$	$\text{argmax}_a UCB(s, a)$
s_1	$10 + 10\sqrt{\frac{\log 31}{27}} \approx 13.566$	$-5 + 10\sqrt{\frac{\log 31}{4}} \approx 4.266$	a_1
s_2	$12 + 10\sqrt{\frac{\log 50}{32}} \approx 15.496$	$10 + 10\sqrt{\frac{\log 50}{18}} \approx 14.662$	a_1

Для $c_2 = 20$ имеем:

	$UCB(s, a_1)$	$UCB(s, a_2)$	$\text{argmax}_a UCB(s, a)$
s_1	$10 + 20\sqrt{\frac{\log 31}{27}} \approx 17.133$	$-5 + 20\sqrt{\frac{\log 31}{4}} \approx 13.531$	a_1
s_2	$12 + 20\sqrt{\frac{\log 50}{32}} \approx 18.993$	$10 + 20\sqrt{\frac{\log 50}{18}} \approx 19.324$	a_2

10 Поиск стратегии

Поиск стратегии (policy search) – это поиск в пространстве стратегий без непосредственного вычисления функции полезности. Пространство стратегий часто имеет меньшую размерность, чем пространство состояний, и в нем нередко удается осуществить более эффективный поиск. Поиск оптимальной стратегии фактически представляет собой оптимизацию параметров в *параметрической стратегии*, направленной на получение максимальной полезности. Параметрическая стратегия может быть представлена в различных формах, таких как нейронные сети, деревья решений и компьютерные программы. Эта глава начинается с обсуждения способа вычисления полезности стратегии при заданном начальном распределении состояний. Затем мы обсудим методы поиска, которые не используют расчет градиента стратегии, оставив методы на основе градиента для следующей главы. Хотя на практике локальный поиск может быть довольно эффективным, мы также обсудим несколько альтернативных подходов к оптимизации, которые помогают избежать локальных оптимумов[1].

10.1. Приблизительная оценка стратегии

Как было показано в разделе 7.2, мы можем вычислить ожидаемый дисконтированный доход при следовании стратегии π из состояния s. Этот ожидаемый дисконтированный доход $U^{\pi}(s)$ может быть вычислен итеративно (алгоритм 7.3) или с помощью матричных операций (алгоритм 7.4), когда пространство состояний является дискретным и относительно небольшим. Мы можем использовать эти подходы для вычисления ожидаемой дисконтированной доходности стратегии π:

$$U(\pi) = \sum_{s} b(s)U^{\pi}(s), \tag{10.1}$$

располагая *начальным распределением состояний* $b(s)$.

Мы будем использовать данное определение $U(\pi)$ на протяжении всей этой главы. Однако часто не удается точно вычислить $U(\pi)$, когда пространство со-

---

[1] Существует много других подходов к оптимизации, как обсуждалось в M. J. Kochenderfer, T. A. Wheeler, *Algorithms for Optimization*. MIT Press, 2019.

стояний большое или непрерывное. В таком случае можно аппроксимировать $U(\pi)$, выбирая *траектории*, состоящие из состояний, действий и вознаграждений при следовании стратегии π. Определение $U(\pi)$ можно переписать так:

$$U(\pi) = \mathbb{E}_{\tau}[R(\tau)] = \int p_{\pi}(\tau) R(\tau) d\tau, \tag{10.2}$$

где $p_{\pi}(\tau)$ – плотность вероятности, связанная с траекторией τ при следовании стратегии π, исходя из начального распределения по состояниям b. *Вознаграждение за траекторию* $R(\tau)$ – это дисконтированный доход, связанный с τ. На рис. 10.1 показано вычисление $U(\pi)$ в контексте траекторий, выбранных из начального распределения по состояниям.

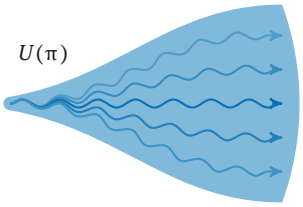


Рис. 10.1. Полезность, связанная со стратегией из начального распределения по состояниям, вычисляется из дохода, связанного со всеми возможными траекториями в рамках данной стратегии, взвешенными в соответствии с их вероятностью

Оценка стратегии методом Монте-Карло (алгоритм 10.1) представляет собой аппроксимацию уравнения (10.2) с помощью m развертываний траектории τ:

$$U(\pi) \approx \frac{1}{m} \sum_{i=1}^{m} R(\tau^{(i)}), \tag{10.3}$$

где $\tau^{(i)}$ – i-я выборка траектории.

Алгоритм 10.1. Оценка стратегии π методом Монте-Карло. Метод выполняет `m` развертываний на глубину `d` в соответствии с динамикой, заданной задачей $\mathcal{P}$. Каждое развертывание запускается из начального состояния, выбранного из распределения состояний `b`. Последняя строка этого кода оценивает стратегию π, параметризованную набором параметров `θ`. Она найдет применение в следующих алгоритмах этой главы, пытающихся найти значение `θ`, которое максимизирует `U`

```
struct MonteCarloPolicyEvaluation
    𝒫 # задача
    b # начальное распределение по состояниям
    d # глубина
    m # количество выборок
end

function (U::MonteCarloPolicyEvaluation)(π)
    R(π) = rollout(U.𝒫, rand(U.b), π, U.d)
```

```
    return mean(R(n) for i = 1:U.m)
end

(U::MonteCarloPolicyEvaluation)(π, θ) = U(s->π(θ, s))
```

Оценка стратегии по методу Монте-Карло является стохастической. Многократные вычисления выражения (10.1) с одной и той же стратегией могут давать разные оценки. Увеличение количества развертываний снижает дисперсию оценки, как показано на рис. 10.2.

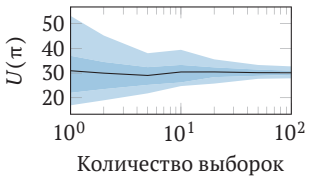


Рис. 10.2. Влияние глубины и количества выборок, заданных для оценки методом Монте-Карло однородной стохастической стратегии в задаче о перевернутом маятнике (приложение F.3). Дисперсия уменьшается по мере увеличения количества выборок. Синие области обозначают эмпирические квантили $U(\pi)$ от 5 % до 95 % и от 25 % до 75 %

Мы будем использовать π_θ для обозначения стратегии, параметризованной набором θ. Для удобства будем использовать $U(\theta)$ как сокращение для $U(\pi_\theta)$ в тех случаях, когда это не вызывает разночтений. Набор параметров θ может быть представлен вектором или в другой более сложной форме. Например, некоторые стратегии можно представить в виде нейронной сети, обладающей определенной структурой. В таком случае θ будет представлять собой набор весовых коэффициентов этой сети. Многие алгоритмы оптимизации предполагают, что θ – это вектор с фиксированным числом компонентов. Другие алгоритмы оптимизации допускают более гибкие представления, включая деревья решений или вычислительные выражения[2].

10.2. Локальный поиск

Распространенным подходом к оптимизации является *локальный поиск*, когда задают начальный набор параметров и постепенно перемещаются от соседа к соседу в пространстве поиска до достижения сходимости. Мы обсуждали эту методику в главе 5 в контексте оптимизации структур байесовской сети с учетом байесовской оценки. В этой главе мы оптимизируем стратегию, параметризованную θ. Мы пытаемся найти значение θ, которое максимизирует $U(\theta)$.

Существует много алгоритмов локального поиска, но в этом разделе основное внимание будет уделено *методу Хука–Дживса* (алгоритм 10.2)[3]. Этот алгоритм предполагает, что наша стратегия параметризована n-мерным век-

[2] Мы не будем обсуждать здесь эти представления, но некоторые из них реализованы в `ExprOptimization.jl`.

[3] R. Hooke, T. A. Jeeves, *Direct Search Solution of Numerical and Statistical Problems*, Journal of the ACM (JACM), vol. 8, no. 2, pp. 212–229, 1961.

тором **θ**. Алгоритм делает шаг размера ±α по каждой из осей координат от текущего **θ**. Эти 2*n* точек образуют окрестность **θ**. Если улучшения стратегии не найдены, то размер шага α уменьшается в соответствии с некоторым коэффициентом. Если улучшение найдено, то начало координат перемещается в лучшую точку. Процесс продолжается до тех пор, пока α не окажется ниже некоторого порогового значения ε > 0. Оптимизация стратегии методом локального поиска продемонстрирована в примере 10.1, а рис. 10.3 иллюстрирует этот процесс.

Алгоритм 10.2. Поиск стратегии с использованием метода Хука–Дживса, который возвращает параметр θ, оптимизированный по отношению к U. Стратегия π принимает в качестве входных данных параметр θ и состояние s. Реализация поиска начинается с определения исходного значения θ. Размер шага α уменьшается в c раз, если ни один из соседей не улучшает стратегию. Итерации выполняются до тех пор, пока размер шага не станет меньше ε

```
struct HookeJeevesPolicySearch
    θ # начальная параметризация
    α # размер шага
    c # коэффициент уменьшения шага
    ε # минимальный размер шага
end

function optimize(M::HookeJeevesPolicySearch, π, U)
    θ, θ′, α, c, ε = copy(M.θ), similar(M.θ), M.α, M.c, M.ε
    u, n = U(π, θ), length(θ)
    while α > ε
        copyto!(θ′, θ)
        best = (i=0, sgn=0, u=u)
        for i in 1:n
            for sgn in (-1,1)
                θ′[i] = θ[i] + sgn*α
                u′ = U(π, θ′)
                if u′ > best.u
                    best = (i=i, sgn=sgn, u=u′)
                end
            end
            θ′[i] = θ[i]
        end
        if best.i != 0
            θ[best.i] += best.sgn*α
            u = best.u
        else
            α *= c
        end
    end
    return θ
end
```

Пример 10.1. Использование алгоритма оптимизации параметров стохастической стратегии

Предположим, мы хотим оптимизировать стратегию для решения простой задачи регулятора, описанной в приложении F.5. Мы определяем стохастическую стратегию π, параметризованную вектором $\boldsymbol{\theta}$, такую, что действие генерируется в соответствии с выражением

$$a \sim \mathcal{N}(\theta_1 s, (|\theta_2| + 10^{-5})^2). \tag{10.4}$$

Следующий код определяет параметрическую стохастическую стратегию π, функцию оценки `U` и метод `M`. Затем он вызывает функцию `optimise(M, π, U)`, которая возвращает оптимизированное значение `θ`. В данном случае мы используем метод Хука–Дживса, но вместо `M` могут быть переданы и другие методы, обсуждаемые в этой главе:

```
function π(θ, s)
    return rand(Normal(θ[1]*s, abs(θ[2]) + 0.00001))
end
b, d, n_rollouts = Normal(0.3,0.1), 10, 3
U = MonteCarloPolicyEvaluation(𝒫, b, d, n_rollouts)
θ, α, c, ε = [0.0,1.0], 0.75, 0.75, 0.01
M = HookeJeevesPolicySearch(θ, α, c, ε)
θ = optimize(M, π, U)
```

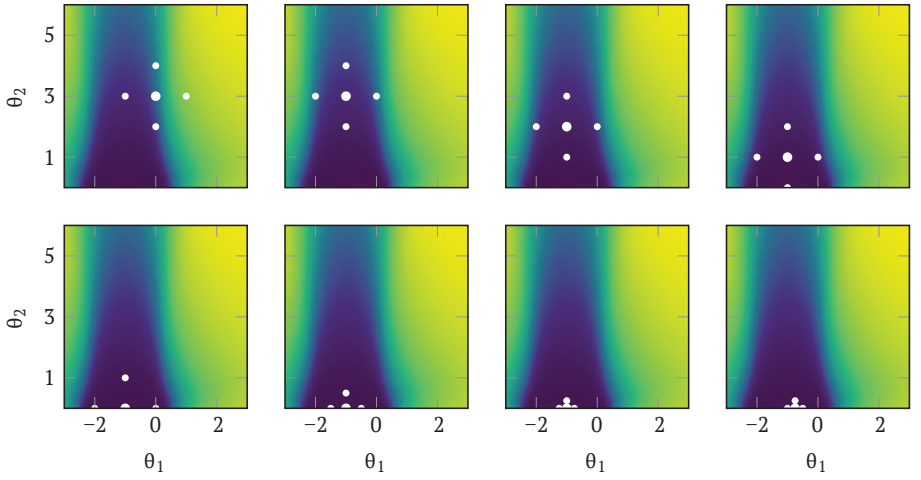


Рис. 10.3. Применение метода Хука–Дживса для оптимизации стратегии в простой задаче регулятора, рассмотренной в примере 10.1. Оценки на каждой итерации показаны белыми точками. Итерации выполняются слева направо и сверху вниз, а фон окрашивается в соответствии с ожидаемой полезностью: желтый цвет указывает на более низкую полезность, а темно-синий – на более высокую полезность

10.3. Генетические алгоритмы

Потенциальная проблема с алгоритмами локального поиска, такими как метод Хука–Дживса, заключается в том, что оптимизация может застрять в локальном оптимуме. Существует множество методов, которые формируют *популяцию*, состоящую из выборок точек в пространстве параметров, выполняют их параллельную оценку близости к цели, а затем рекомбинируют их каким-либо образом, чтобы привести популяцию к глобальному оптимуму. *Генетический алгоритм*[4] – один из таких подходов, вдохновленный биологической эволюцией. Это очень обобщенный метод оптимизации, но он оказался успешным в контексте оптимизации стратегий. Например, этот подход использовался для оптимизации стратегий в видеоиграх Atari, где параметры стратегии соответствуют весам в нейронной сети[5].

Простая версия этого подхода (алгоритм 10.3) начинается с популяции из m случайных параметризаций, $\boldsymbol{\theta}^{(1)}, \ldots, \boldsymbol{\theta}^{(m)}$. Мы вычисляем $U(\boldsymbol{\theta}^{(i)})$ для каждого экземпляра i в популяции. Поскольку эти оценки потенциально подразумевают множество моделирований развертывания и, следовательно, требуют значительных вычислительных ресурсов, они часто выполняются параллельно. Эти оценки помогают нам выделить *элитные экземпляры*, которые являются наилучшими образцами m_{elite} по критерию U.

Популяция на следующей итерации генерируется путем создания $m - 1$ новых параметризаций многократным отбором случайного элитного экземпляра $\boldsymbol{\theta}$ и возмущения его изотропным гауссовым шумом $\boldsymbol{\theta} + \sigma\boldsymbol{\varepsilon}$, где $\varepsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. Наилучшую невозмущенную параметризацию добавляют в качестве m-й выборки. Поскольку оценки основаны на стохастических развертываниях, вариант этого алгоритма может включать прогон дополнительных развертываний, чтобы помочь определить, какой из элитных экземпляров действительно лучший. На рис. 10.4 показано несколько итераций (или *поколений*) генетического алгоритма.

Алгоритм 10.3. Генетический метод поиска стратегии для итеративного обновления совокупности параметров стратегии `θs`, который принимает функцию оценки стратегии `U`, стратегию `π(θ, s)`, стандартное отклонение возмущения `σ`, количество элитных экземпляров `m_elite` и количество итераций `k_max`. Лучшие экземпляры `m_elite` из каждой итерации используются в качестве исходных экземпляров для последующей итерации

```
struct GeneticPolicySearch
    θs      # начальная популяция
```

---

[4] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[5] F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, J. Clune, *Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning*, 2017. arXiv: 171 2.06567v3. Реализация в этом разделе основана на относительно простой формуле. Эта формула не включает кроссовер, который обычно используется для смешивания параметризации в популяции.

```
    σ       # начальное стандартное отклонение
    m_elite # количество элитных экземпляров
    k_max   # количество итераций
end

function optimize(M::GeneticPolicySearch, p, U)
    θs, σ = M.θs, M.σ
    n, m = length(first(θs)), length(θs)
    for k in 1:M.k_max
        us = [U(p, θ) for θ in θs]
        sp = sortperm(us, rev=true)
        θ_best = θs[sp[1]]
        rand_elite() = θs[sp[rand(1:M.m_elite)]]
        θs = [rand_elite() + σ.*randn(n) for i in 1:(m-1)]
        push!(θs, θ_best)
    end
    return last(θs)
end
```

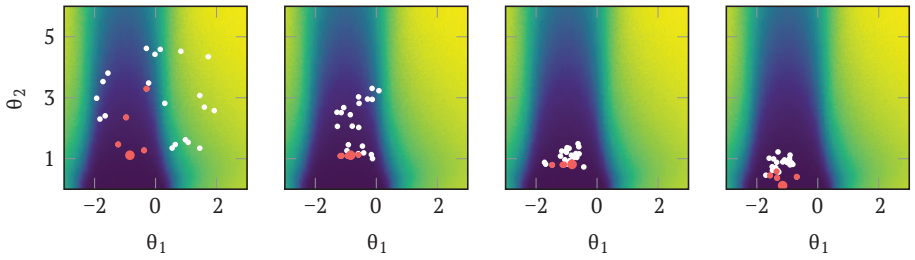


Рис. 10.4. Генетический поиск стратегии с σ = 0.25 применяется к простой задаче регулятора с использованием 25 экземпляров на итерацию. Пять элитных экземпляров в каждом поколении показаны точками красного цвета, а лучший экземпляр отмечен более крупной точкой

10.4. Метод перекрестной энтропии

Метод перекрестной энтропии (алгоритм 10.4) основан на обновлении *поискового распределения* (search distribution) по параметризованному пространству стратегий на каждой итерации[6]. Мы параметризуем это поисковое распределение $p(\theta \mid \psi)$ с помощью ψ[7]. Распределение может принадлежать к любому семейству, но обычно выбирают распределение Гаусса, где ψ представляет среднее значение и ковариацию распределения. Цель состоит в том, чтобы

[6] S. Mannor, R. Y. Rubinstein, Y. Gat, *The Cross Entropy Method for Fast Policy Search*, in International Conference on Machine Learning (ICML), 2003.

[7] Часто θ и ψ являются векторами, но поскольку это не обязательное условие для данного метода, мы не будем выделять их жирным шрифтом в этом разделе.

найти значение ψ^*, которое максимизирует математическое ожидание $U(\theta)$, где θ извлекается из поискового распределения:

$$\psi^* = \arg\max_{\psi} \mathop{\mathbb{E}}_{\theta \sim p(\cdot|\psi)}[U(\theta)] = \arg\max_{\psi} \int U(\theta) p(\theta|\psi) d\theta. \tag{10.5}$$

Прямая максимизация уравнения (10.5) обычно невыполнима с вычислительной точки зрения. Метод перекрестной энтропии заключается в том, чтобы начать поиск со стартового значения ψ, обычно выбираемого из распределения, отражающего соответствующее пространство параметров. На каждой итерации мы берем m экземпляров из такого распределения, а затем обновляем ψ, чтобы точнее подогнать распределение по лучшим выборкам. Для подгонки мы обычно используем оценку максимального правдоподобия (раздел 4.1)[8]. Поиск останавливают после заданного количества итераций или продолжают до тех пор, пока поисковое распределение не станет сильно сфокусированным на оптимуме. На рис. 10.5 показан алгоритм решения простой задачи.

Алгоритм 10.4. Поиск стратегии методом перекрестной энтропии, который итеративно улучшает распределение поиска, изначально заданное как `p`. Этот алгоритм принимает в качестве входных данных параметризованную стратегию `π(θ, s)` и функцию оценки стратегии `U`. На каждой итерации извлекается `m` выборок, и для повторной подгонки распределения используются лучшие экземпляры `m_elite`. Алгоритм завершается после `k_max` итераций. Распределение `p` можно определить с помощью пакета `Distributions.jl`. Например, мы можем задать следующие параметры

```
μ = [0,0,1,0]
Σ = [1,0 0,0; 0,0 1,0]
p = MvNormal (p, X)
```

```
struct CrossEntropyPolicySearch
    p       # начальное распределение
    m       # количество выборок
    m_elite # количество элитных экземпляров
    k_max   # количество итераций
end

function optimize_dist(M::CrossEntropyPolicySearch, π, U)
    p, m, m_elite, k_max = M.p, M.m, M.m_elite, M.k_max
    for k in 1:k_max
        θs = rand(p, m)
        us = [U(π, θs[:,i]) for i in 1:m]
        θ_elite = θs[:,sortperm(us)[(m-m_elite+1):m]]
        p = Distributions.fit(typeof(p), θ_elite)
```

---

[8] Оценка максимального правдоподобия соответствует выбору ψ, который минимизирует перекрестную энтропию (приложение А.9) между поисковым распределением и элитными выборками.

```
    end
    return p
end

function optimize(M, π, U)
    return Distributions.mode(optimize_dist(M, π, U))
end
```

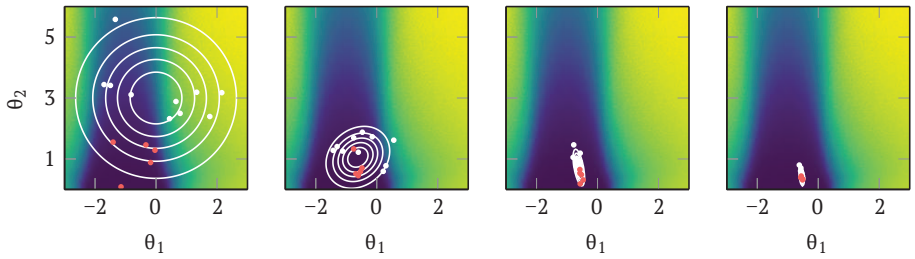


Рис. 10.5. Метод перекрестной энтропии применяется к простой задаче регулятора с использованием многомерного поискового гауссова распределения. Пять элитных образцов в каждой итерации показаны красным цветом. Начальное распределение задано как $\mathcal{N}([0.3], 2\mathbf{I})$

10.5. Эволюционные стратегии

Эволюционные стратегии[9] обновляют поисковое распределение, параметризованное вектором $\boldsymbol{\psi}$, на каждой итерации. Однако, вместо того чтобы подгонять распределение к набору элитных экземпляров, они обновляют распределение, делая шаг в направлении градиента[10]. Градиент цели в уравнении (10.5) можно вычислить следующим образом[11]:

$$\nabla_{\boldsymbol{\psi}} \mathop{\mathbb{E}}_{\theta \sim p(\cdot|\boldsymbol{\psi})}[U(\theta)] = \nabla_{\boldsymbol{\psi}} \int U(\theta) p(\theta|\boldsymbol{\psi}) d\theta \tag{10.6}$$

$$= \int U(\theta) \nabla_{\boldsymbol{\psi}} p(\theta|\boldsymbol{\psi}) d\theta \tag{10.7}$$

$$= \int U(\theta) \nabla_{\boldsymbol{\psi}} p(\theta|\boldsymbol{\psi}) \frac{p(\theta|\boldsymbol{\psi})}{p(\theta|\boldsymbol{\psi})} d\theta \tag{10.8}$$

[9] D. Wierstra, T. Schaul, T. Glasmachers, Y. Sun, J. Peters, J. Schmidhuber, *Natural Evolution Strategies*, Journal of Machine Learning Research, vol. 15, pp. 949–980, 2014.

[10] Мы применяем градиентное восхождение, которое рассматривается в приложении A.11.

[11] Параметр стратегии θ здесь не выделен жирным шрифтом, поскольку он не обязательно должен быть вектором. Однако $\boldsymbol{\psi}$ выделен жирным шрифтом, потому что необходимо, чтобы он был вектором, когда мы работаем с градиентом цели.

$$= \int \left(U(\theta)\nabla_{\boldsymbol{\psi}} \log p(\theta|\boldsymbol{\psi})\right) p(\theta|\boldsymbol{\psi}) d\theta \tag{10.9}$$

$$= \mathop{\mathbb{E}}_{\theta \sim p(\cdot|\boldsymbol{\psi})} [U(\theta)\nabla_{\boldsymbol{\psi}} \log p(\theta|\boldsymbol{\psi})]. \tag{10.10}$$

Логарифм в (10.9) появляется вследствие так называемого *приема с логарифмической производной* (log derivative trick), который основан на наблюдении, что $\nabla_{\boldsymbol{\psi}} \log p(\theta|\boldsymbol{\psi}) = \nabla_{\boldsymbol{\psi}} p(\theta|\boldsymbol{\psi}) / p(\theta|\boldsymbol{\psi})$. Эти вычисления требуют знания $\nabla_{\boldsymbol{\psi}} \log p(\theta|\boldsymbol{\psi})$, но мы часто можем получить результат аналитически, как показано в примере 10.2.

Градиент поиска можно найти из m выборок $\theta^{(1)}, \ldots, \theta^{(m)} \sim p(\cdot|\boldsymbol{\psi})$:

$$\nabla_{\boldsymbol{\psi}} \mathop{\mathbb{E}}_{\theta \sim p(\cdot|\boldsymbol{\psi})} [U(\theta)] \approx \frac{1}{m} \sum_{i=1}^{m} U(\theta^{(i)}) \nabla_{\boldsymbol{\psi}} \log p(\theta^{(i)}|\boldsymbol{\psi}). \tag{10.11}$$

Результат зависит от вычисляемой ожидаемой полезности, которая сама по себе может широко варьироваться. Мы можем сделать нашу оценку градиента более устойчивой с помощью *приема ранжирования* (rank shaping), который заменяет значения полезности весами, основанными на относительной точности каждого образца по сравнению с другими образцами в его итерации. Далее, в соответствии с приемом, m выборок ранжируют в порядке убывания ожидаемой полезности. Вес $w^{(i)}$ присваивается выборке i в соответствии со схемой взвешивания $w^{(1)} \geqslant \cdots \geqslant w^{(m)}$. Градиент поиска приобретает следующий вид:

$$\nabla_{\boldsymbol{\psi}} \mathop{\mathbb{E}}_{\theta \sim p(\cdot|\boldsymbol{\psi})} [U(\theta)] \approx \sum_{i=1}^{m} w^{(i)} \nabla_{\boldsymbol{\psi}} \log p(\theta^{(i)}|\boldsymbol{\psi}). \tag{10.12}$$

Общая схема взвешивания[12]:

$$w^{(i)} = \frac{\max\left(0, \log\left(\frac{m}{2}+1\right) - \log(i)\right)}{\sum_{j=1}^{m} \max\left(0, \log\left(\frac{m}{2}+1\right) - \log(j)\right)} - \frac{1}{m}. \tag{10.13}$$

Эти веса, показанные на рис. 10.6, благоприятствуют нескольким лучшим образцам и дают большинству образцов небольшой отрицательный вес. Использование приема ранжирования уменьшает влияние выбросов.

Алгоритм 10.5 обеспечивает реализацию метода эволюционных стратегий. На рис. 10.7 показан пример последовательности поиска.

---

[12] N. Hansen, A. Ostermeier, *Adapting Arbitrary Normal Mutation Distributions in Evolution Strategies: The Covariance Matrix Adaptation*, in IEEE International Conference on Evolutionary Computation, 1996.

Пример 10.2. Вывод уравнений градиента логарифмического правдоподобия для многомерного гауссова распределения. Исходный вывод и несколько более сложных решений для работы с положительно определенной ковариационной матрицей см. в D. Wierstra, T. Schaul, T. Glasmachers, Y. Sun, J. Peters, J. Schmidhuber, *Natural Evolution Strategies*, Journal of Machine Learning Research, vol. 15, pp. 949–980, 2014

Многомерное нормальное распределение $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ со средним значением $\boldsymbol{\mu}$ и ковариацией $\boldsymbol{\Sigma}$ образует часто применяемое семейство распределений. Вероятность в d измерениях принимает форму

$$p(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = (2\pi)^{-\frac{d}{2}} |\boldsymbol{\Sigma}|^{-\frac{1}{2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}) \right),$$

где $|\boldsymbol{\Sigma}|$ является определителем $\boldsymbol{\Sigma}$. Логарифмическое правдоподобие:

$$\log p(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = -\frac{d}{2}\log(2\pi) - \frac{1}{2}\log(|\boldsymbol{\Sigma}|) - \frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}).$$

Параметры могут быть обновлены с использованием их градиентов логарифмического правдоподобия:

$$\nabla_{\boldsymbol{\mu}} \log p(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu});$$

$$\nabla_{\boldsymbol{\Sigma}} \log p(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{2}\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})(\mathbf{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} - \frac{1}{2}\boldsymbol{\Sigma}^{-1}.$$

Член $\nabla_{\boldsymbol{\Sigma}}$ содержит частную производную каждой записи $\boldsymbol{\Sigma}$ по логарифмическому правдоподобию.

Непосредственное обновление $\boldsymbol{\Sigma}$ может не привести к положительно определенной матрице, как это требуется для ковариационных матриц. Одно из решений состоит в том, чтобы представить $\boldsymbol{\Sigma}$ как произведение $\mathbf{A}^\top\mathbf{A}$, что гарантирует сохранение положительной полуопределенности $\boldsymbol{\Sigma}$, а затем вместо этого обновить $\mathbf{A}$. Замена $\boldsymbol{\Sigma}$ на $\mathbf{A}^\top\mathbf{A}$ и взятие градиента по $\mathbf{A}$ дает

$$\nabla_{(\mathbf{A})} \log p(\mathbf{x}|\boldsymbol{\mu}, \mathbf{A}) = \mathbf{A}[\nabla_{\boldsymbol{\Sigma}} \log p(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) + \nabla_{\boldsymbol{\Sigma}} \log p(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma})^\top.$$

Алгоритм 10.5. Метод эволюционных стратегий для обновления поискового распределения `D(ψ)` по параметризации стратегии `π(θ, s)`. Этот алгоритм также использует параметризацию начального поискового распределения `ψ`, градиент логарифмического правдоподобия `∇logp(ψ, θ)`, функцию оценки стратегии `U` и счетчик итераций `k_max`. На каждой итерации извлекается `m` выборок параметризации, которые используются для вычисления градиента поиска. Этот градиент затем применяется с коэффициентом шага α. Для определения `D(ψ)` можно использовать `Distributions.jl`. Например, если мы хотим определить `D` для построения гауссова распределения с заданным средним значением `ψ` и фиксированной ковариацией `Σ`, это можно сделать так: `D(ψ) = MvNormal(ψ, Σ)`

```
struct EvolutionStrategies
    D     # конструктор распределения
    ψ     # параметризация начального распределения
    ∇logp # градиент логарифмического правдоподобия
    m     # количество выборок
    α     # коэффициент шага
    k_max # количество итераций
end

function evolution_strategy_weights(m)
    ws = [max(0, log(m/2+1) - log(i)) for i in 1:m]
    ws ./= sum(ws)
    ws .-= 1/m
    return ws
end

function optimize_dist(M::EvolutionStrategies, π, U)
    D, ψ, m, ∇logp, α = M.D, M.ψ, M.m, M.∇logp, M.α
    ws = evolution_strategy_weights(m)
    for k in 1:M.k_max
        θs = rand(D(ψ), m)
        us = [U(π, θs[:,i]) for i in 1:m]
        sp = sortperm(us, rev=true)
        ∇ = sum(w.*∇logp(ψ, θs[:,i]) for (w,i) in zip(ws,sp))
        ψ += α.*∇
    end
    return D(ψ)
end
```

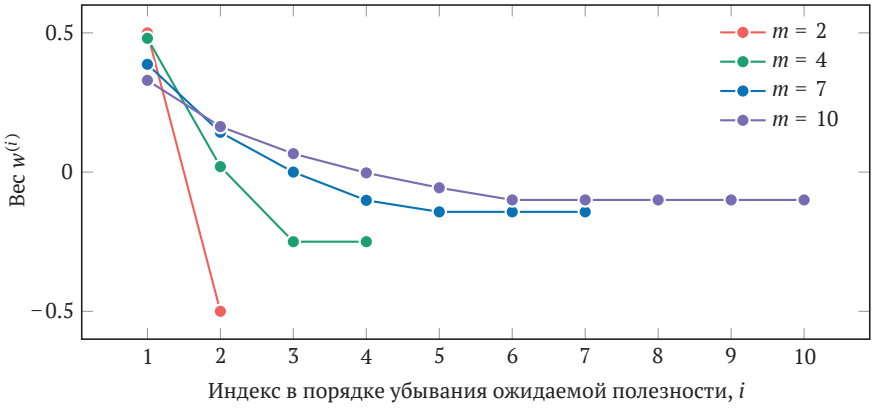


Рис. 10.6. Несколько взвешиваний, построенных с использованием уравнения (10.13)

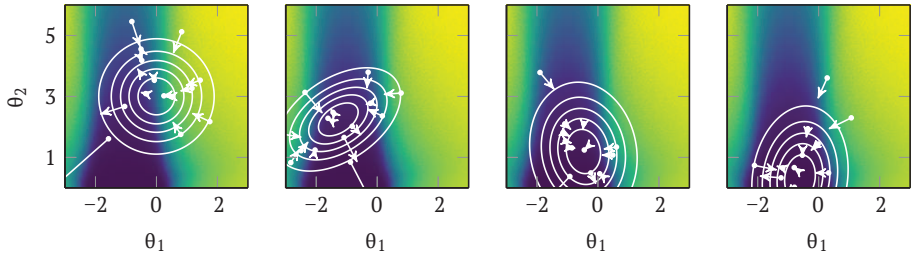


Рис. 10.7. Стратегии эволюции (алгоритм 10.5), применяемые к задаче простого регулятора с использованием многомерного поискового гауссова распределения. Образцы показаны белым цветом вместе с их вкладом в поисковый градиент, $w\nabla \log p$

10.6. Изотропные эволюционные стратегии

В предыдущем разделе были представлены эволюционные стратегии, которые могут работать с обобщенными поисковыми распределениями. В этом разделе будет сделано более конкретное предположение, что поисковое распределение является сферическим (то есть сферически изотропным) гауссовым распределением, где ковариационная матрица принимает вид $\sigma^2\mathbf{I}$[13]. При таком допущении ожидаемая полезность распределения, представленная уравнением (10.5), упрощается до[14]

$$\mathop{\mathbb{E}}_{\boldsymbol{\theta}\sim\mathcal{N}(\boldsymbol{\psi},\sigma^2\mathbf{I})}[U(\theta)] = \mathop{\mathbb{E}}_{\boldsymbol{\varepsilon}\sim\mathcal{N}(\mathbf{0},\mathbf{I})}[U(\boldsymbol{\psi}+\sigma\boldsymbol{\varepsilon})]. \tag{10.14}$$

---

[13] Пример такого подхода применительно к поиску стратегии исследуют T. Salimans, J. Ho, X. Chen, S. Sidor, I. Sutskever, *Evolution Strategies as a Scalable Alternative to Reinforcement Learning*, 2017. arXiv: 1703.03864v2.

[14] В общем случае если $\mathbf{A}^\top\mathbf{A} = \boldsymbol{\Sigma}$, то $\boldsymbol{\theta} = \boldsymbol{\mu} + \mathbf{A}^\top\boldsymbol{\varepsilon}$ преобразует $\varepsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ в выборку $\boldsymbol{\theta} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$.

Поисковый градиент сводится к следующему выражению:

$$\nabla_{\boldsymbol{\psi}} \mathop{\mathbb{E}}_{\boldsymbol{\theta}\sim\mathcal{N}(\boldsymbol{\psi},\sigma^2\mathbf{I})}[U(\boldsymbol{\theta})] = \mathop{\mathbb{E}}_{\boldsymbol{\theta}\sim\mathcal{N}(\boldsymbol{\psi},\sigma^2\mathbf{I})}[U(\boldsymbol{\theta})\nabla_{\boldsymbol{\psi}} \log p(\boldsymbol{\theta}|\boldsymbol{\psi},\sigma^2\mathbf{I})] \tag{10.15}$$

$$= \mathop{\mathbb{E}}_{\boldsymbol{\theta}\sim\mathcal{N}(\boldsymbol{\psi},\sigma^2\mathbf{I})}\left[U(\boldsymbol{\theta})\frac{1}{\sigma^2}(\boldsymbol{\theta}-\boldsymbol{\psi})\right] \tag{10.16}$$

$$= \mathop{\mathbb{E}}_{\boldsymbol{\varepsilon}\sim\mathcal{N}(\mathbf{0},\mathbf{I})}\left[U(\boldsymbol{\psi}+\sigma\boldsymbol{\varepsilon})\frac{1}{\sigma^2}(\sigma\boldsymbol{\varepsilon})\right] \tag{10.17}$$

$$= \frac{1}{\sigma}\mathop{\mathbb{E}}_{\boldsymbol{\varepsilon}\sim\mathcal{N}(\mathbf{0},\mathbf{I})}\left[U(\boldsymbol{\psi}+\sigma\boldsymbol{\varepsilon})\boldsymbol{\varepsilon}\right]. \tag{10.18}$$

Этот подход реализован в алгоритме 10.6, использующем *зеркальную выборку* (mirrored sampling)[15]. Мы выбираем $m/2$ образцов из поискового распределения, а затем генерируем еще $m/2$ выборок, отражая их относительно среднего значения. Отраженные образцы уменьшают дисперсию оценки градиента[16]. Преимущество использования этого метода показано на рис. 10.8.

Алгоритм 10.6. Метод эволюционных стратегий для обновления изотропного многомерного поискового гауссова распределения со средним ψ и ковариацией $\sigma^2\mathbf{I}$ по параметризации для стратегии `π(θ, s)`. Эта реализация также использует функцию оценки стратегии `U`, коэффициент шага α и количество итераций `k_max`. На каждой итерации извлекается и зеркалируется `m/2` выборок параметризации, которые затем используются для вычисления градиента поиска

```
struct IsotropicEvolutionStrategies
    ψ     # начальное среднее
    σ     # начальное стандартное отклонение
    m     # количество выборок
    α     # коэффициент шага
    k_max # количество итераций
end

function optimize_dist(M::IsotropicEvolutionStrategies, π, U)
    ψ, σ, m, α, k_max = M.ψ, M.σ, M.m, M.α, M.k_max
    n = length(ψ)
    ws = evolution_strategy_weights(2*div(m,2))
    for k in 1:k_max
        εs = [randn(n) for i in 1:div(m,2)]
        append!(εs, -εs) # weight mirroring
```

---

[15] D. Brockhoff, A. Auger, N. Hansen, D. Arnold, T. Hohm, *Mirrored Sampling and Sequential Selection for Evolution Strategies*, in International Conference on Parallel Problem Solving from Nature, 2010.

[16] Этот метод был реализован в T. Salimans, J. Ho, X. Chen, S. Sidor, I. Sutskever, *Evolution Strategies as a Scalable Alternative to Reinforcement Learning*, 2017. arXiv: 1703 . 03864v2. Они рассматривали и другие методы, в том числе снижение веса.

```
        us = [U(π, ψ + σ.*ε) for ε in εs]
        sp = sortperm(us, rev=true)
        ∇ = sum(w.*εs[i] for (w,i) in zip(ws,sp)) / σ
        ψ += α.*∇
    end
    return MvNormal(ψ, σ)
end
```

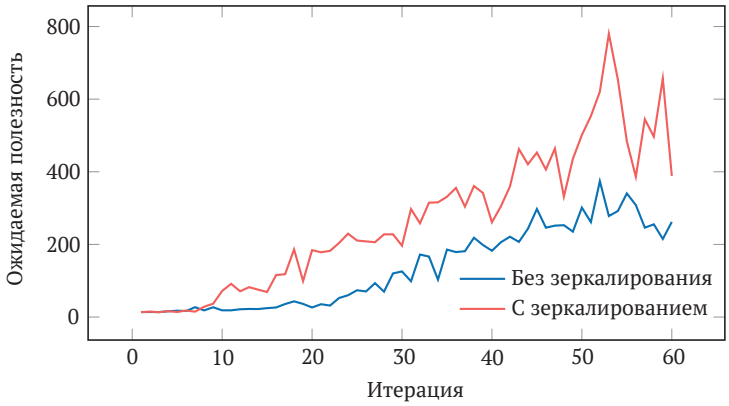


Рис. 10.8. Демонстрация влияния зеркалирования выборки на изотропные эволюционные стратегии. Двухуровневая нейронная сеть была обучена стратегии решения задачи о перевернутом маятнике (приложение F.3) с использованием `m = 10` и `σ = 0.25` с шестью развертываниями на оценку. Зеркальная выборка значительно ускоряет и стабилизирует обучение

10.7. Заключение

- Оценка стратегии по методу Монте-Карло включает в себя вычисление ожидаемой полезности, которую приносит стратегия, с использованием большого количества развертываний из состояний, выбранных из начального распределения по состояниям.
- Методы локального поиска, такие как метод Хука–Дживса, улучшают стратегию, основанную на небольших локальных изменениях.
- Генетические алгоритмы обрабатывают совокупность точек в пространстве параметров, рекомбинируя их различными способами в попытке привести совокупность к глобальному оптимуму.
- Метод перекрестной энтропии итеративно улучшает распределение поиска по параметрам стратегии, подгоняя распределение к элитным выборкам на каждой итерации.
- Эволюционные стратегии пытаются улучшить поисковое распределение, используя информацию о градиенте из выборок из этого распределения.

- Изотропные эволюционные стратегии предполагают, что поисковое распределение является изотропным гауссовым распределением.

10.8. Упражнения

Упражнение 10.1. Как количество выборок влияет на дисперсию оценки полезности при оценке стратегии методом Монте-Карло?

Решение. Дисперсия оценки при использовании метода Монте-Карло – это дисперсия среднего значения m выборок. Предполагается, что эти выборки независимы, поэтому дисперсия среднего представляет собой дисперсию одной оценки развертывания, деленную на количество выборок:

$$\mathrm{Var}[\hat{U}(\pi)] = \mathrm{Var}\left[\sum_{i=1}^{m} R(\tau^{(i)})\right] = \frac{1}{m}\mathrm{Var}_{\tau}[R(\tau)],$$

где $\hat{U}(\pi)$ – полезность, полученная в результате оценки стратегии по методу Монте-Карло, а $R(\tau)$ – траекторное вознаграждение за выбранную траекторию τ. Следовательно, выборочная дисперсия уменьшается пропорционально $1/m$.

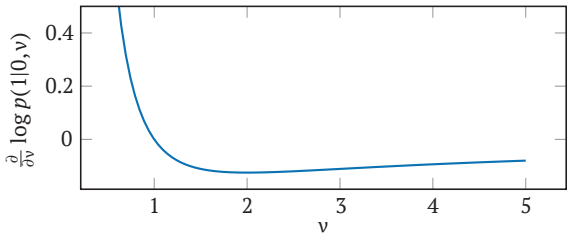
Упражнение 10.2. Какое влияние оказывает изменение количества выборок m и количества элитных экземпляров m_{elite} на поиск стратегии методом кросс-энтропии?

Решение. Вычислительные затраты на итерацию линейно зависят от количества выборок. Чем больше выборок, тем лучше будет охват пространства поиска, что повысит вероятность выявления лучших элитных экземпляров для совершенствования стратегии. Имеет значение и количество элитных экземпляров. Если все выборки будут элитными, вы не получите обратной связи по процессу улучшения. Наличие слишком малого количества элитных экземпляров может вызвать преждевременную сходимость к неоптимальному решению.

Упражнение 10.3. Рассмотрим использование эволюционных стратегий с одномерным гауссовым распределением $\theta \sim \mathcal{N}(\mu, \nu)$. Как выглядит градиент поиска по отношению к дисперсии ν? Какая проблема возникает, когда дисперсия становится малой?

Решение. Градиент поиска – это градиент логарифмического правдоподобия:

$$\begin{aligned}\frac{\partial}{\partial \nu}\log p(x|\mu,\nu) &= \frac{\partial}{\partial \nu}\log\frac{1}{\sqrt{2\pi\nu}}\exp\left(-\frac{(x-\mu)^2}{2\nu}\right)\\ &= \frac{\partial}{\partial \nu}\left(-\frac{1}{2}\log(2\pi)-\frac{1}{2}\log(\nu)-\frac{(x-\mu)^2}{2\nu}\right)\\ &= -\frac{1}{2\nu}+\frac{(x-\mu)^2}{2\nu^2}.\end{aligned}$$



Очевидно, что градиент стремится к бесконечности, когда дисперсия приближается к нулю. Это проблема, потому что для сходимости поиска по распределению дисперсия должна быть небольшой, но слишком большие градиенты могут привести к тому, что простые методы восхождения по градиенту «перескочат» оптимум.

Упражнение 10.4. Уравнение (10.14) определяет цель оптимизации в контексте поискового распределения $\boldsymbol{\theta} \sim \mathcal{N}(\boldsymbol{\psi}, \boldsymbol{\Sigma})$. Какое преимущество имеет эта цель по сравнению с прямой оптимизацией $\boldsymbol{\theta}$, направленной на достижение ожидаемой полезности в уравнении (10.1)?

Решение. Добавление гауссова шума вокруг параметров стратегии может сгладить разрывы в исходной цели, что может сделать оптимизацию более надежной.

Упражнение 10.5. Какой из методов, рассмотренных в этой главе, лучше всего подходит в ситуации, когда для решения конкретной задачи могут одинаково хорошо работать несколько стратегий разного типа?

Решение. Метод Хука–Дживса улучшает параметризацию одной стратегии, поэтому он не может поддерживать несколько стратегий. И метод кросс-энтропии, и эволюционный поиск стратегии используют поисковые распределения. Чтобы успешно представить несколько типов стратегий, необходимо использовать мультимодальное распределение. Одним из распространенных мультимодальных распределений является смесь гауссовых распределений. Такая смесь не поддается аналитическому выводу, но ее можно надежно подогнать с помощью метода максимизации ожидания (expectation maximization, EM), как показано в примере 4.4. Генетические алгоритмы могут поддерживать несколько стратегий, если размер популяции достаточно велик.

Упражнение 10.6. Предположим, у нас есть параметризованная стратегия π_θ, которую мы хотели бы оптимизировать с помощью метода Хука–Дживса. Если мы инициализируем параметр значением $\theta = 0$, а функция полезности равна $U(\theta) = -3\theta^2 + 4\theta + 1$, каков наибольший размер шага α, который все еще гарантирует улучшение стратегии в первой итерации метода Хука–Дживса?

Решение. Метод Хука–Дживса оценивает целевую функцию в центральной точке $\pm\alpha$ по каждому направлению осей координат. Чтобы гарантировать улучше-

ние на первой итерации поиска Хука–Дживса, хотя бы одно из значений целевой функции в новых точках должно улучшить значение целевой функции. Для нашей задачи оптимизации стратегии это означает, что мы ищем наибольший размер шага α такой, что либо $U(\theta + \alpha)$, либо $U(\theta - \alpha)$ больше, чем $U(\theta)$.

Поскольку базисная функция полезности является параболической и вогнутой, максимальный размер шага, который все еще может привести к улучшению, немного меньше ширины параболы в текущей точке. Следовательно, нам нужно найти такую точку θ' на параболе, противоположную текущей точке, в которой $U(\theta') = U(\theta)$:

$$U(\theta) = -3\theta^2 + 4\theta + 1 = -3(0)^2 + 4(0) + 1 = 1;$$

$$U(\theta) = U(\theta');$$

$$1 = -3\theta'^2 + 4\theta' + 1;$$

$$0 = -3\theta'^2 + 4\theta' + 0;$$

$$\theta' = \frac{-4 \pm \sqrt{4^2 - 4(-3)(0)}}{2(-3)} = \frac{-4 \pm 4}{-6} = \frac{2 \pm 2}{3} = \left\{0, \tfrac{4}{3}\right\}.$$

Таким образом, точка на параболе, противоположная текущей точке, – это $\theta' = \frac{4}{3}$. Расстояние между θ и θ' равно $\frac{4}{3} - 0 = \frac{4}{3}$. Таким образом, максимальный размер шага, который мы можем использовать, и при этом гарантировать улучшение на первой итерации, чуть меньше $\frac{4}{3}$.

Упражнение 10.7. Предположим, у нас есть стратегия, параметризованная одним параметром θ. Мы применяем подход эволюционных стратегий с поисковым распределением, которое сответствует распределению Бернулли $p(\theta|\psi) = \psi^\theta(1 - \psi)^{1-\theta}$. Найдите градиент логарифмического правдоподобия $\nabla_\psi \log p(\theta|\psi)$.

Решение. Градиент логарифмического правдоподобия можно вычислить следующим образом:

$$p(\theta|\psi) = \psi^\theta(1 - \psi)^{1-\theta};$$

$$\log p(\theta|\psi) = \log(\psi^\theta(1 - \psi)^{1-\theta});$$

$$\log p(\theta|\psi) = \theta \log \psi + (1 - \theta)\log(1 - \psi);$$

$$\nabla_\psi \log p(\theta|\psi) = \frac{d}{d\psi}[\theta \log \psi + (1-\theta)\log(1-\psi)];$$

$$\nabla_\psi \log p(\theta|\psi) = \frac{\theta}{\psi} - \frac{1-\theta}{1-\psi}.$$

Упражнение 10.8. Вычислите веса выборок для оценки градиента поиска с ранжированием, если дано количество выборок $m = 3$.

Решение. Сначала найдем числитель первого члена уравнения (10.13) для всех i:

$$i = 1 \quad \max\left(0, \log\left(\frac{2}{2}+1\right) - \log 1\right) = \log\frac{5}{2};$$

$$i = 2 \quad \max\left(0, \log\left(\frac{2}{2}+1\right) - \log 2\right) = \log\frac{5}{4};$$

$$i = 3 \quad \max\left(0, \log\left(\frac{2}{2}+1\right) - \log 3\right) = 0.$$

Теперь вычислим веса:

$$w^{(1)} = \frac{\log\frac{5}{2}}{\log\frac{5}{2} + \log\frac{5}{4} + 0} - \frac{1}{3} = 0.47;$$

$$w^{(2)} = \frac{\log\frac{5}{4}}{\log\frac{5}{2} + \log\frac{5}{4} + 0} - \frac{1}{3} = -0.14;$$

$$w^{(3)} = \frac{0}{\log\frac{5}{2} + \log\frac{5}{4} + 0} - \frac{1}{3} = -0.33.$$

11 Нахождение градиента стратегии

В предыдущей главе обсуждалось несколько способов непосредственной оптимизации параметров стратегии для максимизации ожидаемой полезности. Во многих случаях при подборе параметров стратегии бывает удобно использовать градиент полезности в качестве направления процесса оптимизации. В этой главе обсуждается несколько подходов к нахождению этого градиента через развертывания траекторий[1]. Основной проблемой данного подхода выступает дисперсия результата из-за стохастичности траекторий, возникающей как в окружающей среде, так и в наших исследованиях. В следующей главе мы рассмотрим, как использовать эти алгоритмы для вычисления градиентов с целью оптимизации стратегии.

11.1. Конечная разность

Метод *конечных разностей* (finite difference) находит градиент функции по небольшим изменениям в ее оценке. Напомним, что производная функции одной переменной f равна

$$\frac{\mathrm{d}f}{\mathrm{d}x}(x) = \lim_{\delta \to 0} \frac{f(x+\delta) - f(x)}{\delta}. \tag{11.1}$$

Производная в точке x может быть аппроксимирована достаточно малым шагом $\delta > 0$:

$$\frac{\mathrm{d}f}{\mathrm{d}x}(x) \approx \frac{f(x+\delta) - f(x)}{\delta}. \tag{11.2}$$

Это приближение показано на рис. 11.1.

---

[1] Дополнительная информация по этой теме в M. C. Fu, *Gradient Estimation*, in Simulation, S. G. Henderson, B. L. Nelson, eds., Elsevier, 2006, pp. 575–616.

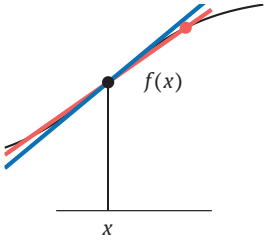


Рис. 11.1. Метод конечных разностей аппроксимирует производную $f(x)$ путем нахождения точки вблизи x. Приближение методом конечной разности, выделенное красным, не совсем соответствует истинной производной, выделенной синим цветом

Градиент многомерной функции f с входом длины n равен

$$\nabla f(\mathbf{x}) = \left[\frac{\partial f}{\partial x_1}(\mathbf{x}), \ldots, \frac{\partial f}{\partial x_n}(\mathbf{x})\right]. \tag{11.3}$$

Для нахождения градиента конечные разности могут быть применены к каждому измерению.

В контексте оптимизации стратегии нам необходимо найти градиент полезности, ожидаемый от соблюдения стратегии, параметризованной значением $\boldsymbol{\theta}$:

$$\nabla U(\boldsymbol{\theta}) = \left[\frac{\partial U}{\partial \theta_1}(\boldsymbol{\theta}), \ldots, \frac{\partial U}{\partial \theta_n}(\boldsymbol{\theta})\right] \tag{11.4}$$

$$\approx \left[\frac{U(\boldsymbol{\theta} + \delta\mathbf{e}^{(1)}) - U(\boldsymbol{\theta})}{\delta}, \ldots, \frac{U(\boldsymbol{\theta} + \delta\mathbf{e}^{(n)}) - U(\boldsymbol{\theta})}{\delta}\right], \tag{11.5}$$

где $\mathbf{e}^{(i)}$ – i-й *стандартный базисный вектор* (standard basis vector), состоящий из нулей, за исключением i-го компонента, который равен 1.

Как было сказано в разделе 10.1, нам необходимо смоделировать развертывание стратегий для нахождения $U(\boldsymbol{\theta})$. Мы можем использовать алгоритм 11.1 для генерации траекторий. Затем мы можем вычислить их доходность и найти совокупную полезность, которую обеспечивает стратегия. Алгоритм 11.2 реализует нахождение градиента в соответствии с уравнением (11.5) путем моделирования m развертываний для каждого компонента и усреднения результатов.

Алгоритм 11.1. Метод генерации траектории, связанной с задачей $\mathcal{P}$, начинающий с состояния `s` и выполняющий стратегию `π` до глубины `d`. Он создает вектор `τ`, содержащий кортежи состояние-действие-вознаграждение

```
function simulate(𝒫::MDP, s, π, d)
    τ = []
    for i = 1:d
        a = π(s)
        s′, r = 𝒫.TR(s,a)
```

```
        push!(τ, (s,a,r))
        s = s′
    end
    return τ
end
```

Алгоритм 11.2. Метод оценки градиента стратегии с использованием конечных разностей для задачи $\mathcal{P}$, параметризованной стратегии `π(θ, s)` и вектора параметризации стратегии `θ`. Оценки полезности получаются из `m` развертываний до глубины `d`. Размер шага равен `δ`

```
struct FiniteDifferenceGradient
    𝒫 # задача
    b # начальное распределение по состояниям
    d # глубина
    m # количество выборок
    δ # размер шага
end

function gradient(M::FiniteDifferenceGradient, π, θ)
    𝒫, b, d, m, δ, γ, n = M.𝒫, M.b, M.d, M.m, M.δ, M.𝒫.γ, length(θ)
    Δθ(i) = [i == k ? δ : 0.0 for k in 1:n]
    R(τ) = sum(r*γ^(k-1) for (k, (s,a,r)) in enumerate(τ))
    U(θ) = mean(R(simulate(𝒫, rand(b), s->π(θ, s), d)) for i in 1:m)
    ΔU = [U(θ + Δθ(i)) - U(θ) for i in 1:n]
    return ΔU ./ δ
end
```

Основной проблемой при получении точных значений градиента стратегии является тот факт, что дисперсия вознаграждения траектории может быть довольно высокой. Один из подходов к уменьшению результирующей дисперсии в значении градиента состоит в том, чтобы в каждом развертывании использовались одни и те же начальные числа генератора случайных чисел[2]. Такой подход может быть полезен, например, в тех случаях, когда одно развертывание в самом начале приводит к маловероятному переходу. Другие развертывания будут иметь ту же тенденцию из-за общего генератора случайных чисел, и их вознаграждения будут иметь такую же предвзятость.

Представления стратегии оказывают значительное влияние на ее градиент. Пример 11.1 демонстрирует чувствительность градиента стратегии к ее параметризации. Метод конечных разностей плохо работает при оптимизации стратегии, если параметры различаются по масштабу.

[2] Общие случайные начальные значения применяются в алгоритме PEGASUS. A. Y. Ng, M. Jordan, *A Policy Search Method for Large MDPs and POMDPs*, in Conference on Uncertainty in Artificial Intelligence (UAI), 2000.

Пример 11.1. Пример того, как параметризация стратегии оказывает существенное влияние на ее градиент

Рассмотрим состоящий из одного состояния одношаговый MDP с одномерным непрерывным пространством действий и функцией вознаграждения $R(s, a) = a$. В этом случае более крупные действия приносят более высокое вознаграждение.

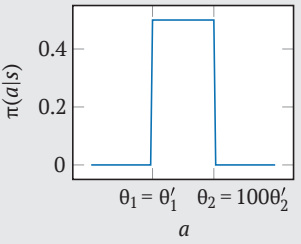
Предположим, у нас есть стохастическая стратегия $\pi_{\boldsymbol{\theta}}$, которая производит выборку своего действия в соответствии с равномерным распределением между θ_1 и θ_2, где $\theta_2 > \theta_1$. Ожидаемое значение полезности равно

$$U(\boldsymbol{\theta}) = \mathbb{E}[a] = \int_{\theta_1}^{\theta_2} a \frac{1}{\theta_2 - \theta_1} \mathrm{d}a = \frac{\theta_1 + \theta_2}{2}.$$

Градиент стратегии будет следующим:

$$\nabla U(\boldsymbol{\theta}) = [1/2, 1/2].$$

Стратегию можно параметризовать заново, чтобы получать действия из равномерного распределения между θ'_1 и $100\theta'_2$, где $100\theta'_2 > \theta'_1$. Теперь ожидаемое вознаграждение равно $(\theta'_1 + 100\theta'_2)/2$, а градиент стратегии равен [1/2, 50].



Эти две параметризации могут представлять одни и те же стратегии, но они имеют очень разные градиенты. Найти подходящий скаляр возмущения для второй стратегии гораздо сложнее, потому что параметры сильно различаются по масштабу.

11.2. Градиент регрессии

Вместо оценки градиента по $\boldsymbol{\theta}$ путем фиксированного шага вдоль каждой оси координат, как это было сделано в предыдущем разделе, мы можем использовать *линейную регрессию*[3] для оценки градиента по результатам случайных возмущений $\boldsymbol{\theta}$. Эти возмущения сохраняются в матрице следующим образом[4]:

$$\Delta\Theta = \begin{bmatrix} (\Delta\boldsymbol{\theta}^{(1)})^\top \\ \vdots \\ (\Delta\boldsymbol{\theta}^{(m)})^\top \end{bmatrix}. \tag{11.6}$$

[3] Линейную регрессию мы рассматривали в разделе 8.6.

[4] Этот общий подход иногда называют стохастической аппроксимацией с одновременным возмущением в J. C. Spall, *Introduction to Stochastic Search and Optimization*. Wiley, 2003. Общая связь с линейной регрессией представлена в J. Peters, S. Schaal, *Reinforcement Learning of Motor Skills with Policy Gradients, Neural Networks*, vol. 21, no. 4, pp. 682–697, 2008.

Чем больше возмущений параметров стратегии, тем точнее оценки градиента[5].

Для каждого из этих возмущений мы выполняем развертывание и находим изменение полезности[6]:

$$\Delta\mathbf{U} = [U(\boldsymbol{\theta} + \Delta\boldsymbol{\theta}^{(1)}) - U(\boldsymbol{\theta}), \ldots, (\boldsymbol{\theta} + \Delta\boldsymbol{\theta}^{(m)}) - U(\boldsymbol{\theta})]. \tag{11.7}$$

Тогда оценка градиента стратегии с использованием линейной регрессии[7] будет следующей:

$$\nabla U(\boldsymbol{\theta}) \approx \Delta\boldsymbol{\Theta}^{+}\Delta\mathbf{U}. \tag{11.8}$$

В алгоритме 11.3 представлена реализация этого подхода, где возмущения равномерно извлекаются из гиперсферы с радиусом δ. Пример 11.2 демонстрирует этот подход с помощью простой функции.

Алгоритм 11.3. Метод оценки градиента стратегии с использованием конечных разностей для задачи MDP $\mathcal{P}$, стохастической параметрической стратегии `π(θ, s)` и вектора параметризации стратегии `θ`. Векторы вариаций стратегии генерируются путем нормализации нормально распределенных выборок и масштабирования с помощью скаляра возмущения `δ`. Генерируется `m` возмущений параметров, и каждое из них оценивается при развертывании на глубину `d` из начального состояния, извлеченного из распределения `b`, и сравнивается с исходной параметризацией стратегии

```
struct RegressionGradient
    𝒫 # задача
    b # начальное распределение по состояниям
    d # глубина
    m # количество выборок
    δ # размер шага
end

function gradient(M::RegressionGradient, π, θ)
    𝒫, b, d, m, δ, γ = M.𝒫, M.b, M.d, M.m, M.δ, M.𝒫.γ
    Δθ = [δ.*normalize(randn(length(θ)), 2) for i = 1:m]
    R(τ) = sum(r*γ^(k-1) for (k, (s,a,r)) in enumerate(τ))
    U(θ) = R(simulate(𝒫, rand(b), s->π(θ,s), d))
    ΔU = [U(θ + Δθ) - U(θ) for Δθ in Δθ]
    return pinv(reduce(hcat, Δθ)') * ΔU
end
```

---

[5] Рекомендуемое эмпирическое правило состоит в том, чтобы использовать примерно в два раза больше возмущений, чем количество параметров.

[6] Это уравнение показывает *разность с интерполяцией вперед* (forward difference). Также могут быть использованы другие формулировки конечной разности, такие как центральная разность.

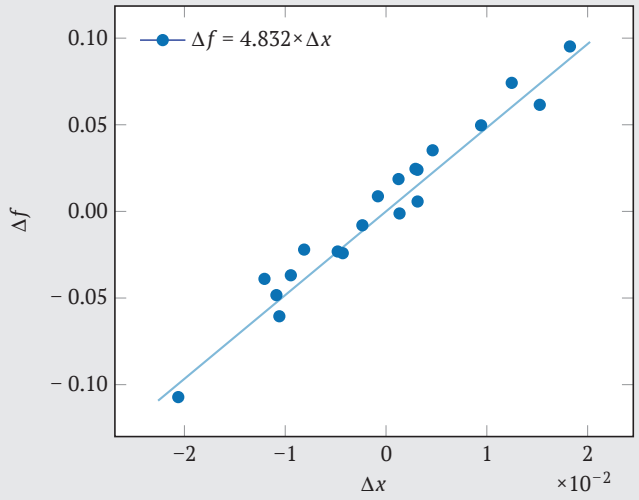
[7] Как было сказано в разделе 8.6, $\mathbf{X}^{+}$ обозначает псевдоинверсию $\mathbf{X}$.

Пример 11.2. Использование метода градиента регрессии для одномерной функции

Допустим, мы хотим применить метод градиента регрессии для нахождения градиента простой одномерной функции $f(x) = x^2$, оцененной при $x_0 = 2$ по $m = 20$ выборок. Чтобы имитировать стохастичность, присущую оценке стратегии, мы добавляем шум к значению функции. Мы генерируем набор возмущений ΔX, выбранных из $\mathcal{N}(0, \delta^2)$, и вычисляем $f(x_0 + \Delta x) - f(x_0)$ для каждого возмущения Δx в ΔX. Затем получаем одномерный градиент (или производную) $\Delta X^+ \Delta F$ с помощью следующего кода:

```
f(x) = x^2 + 1e-2*randn()
m = 20
δ = 1e-2
ΔX = [δ.*randn() for i = 1:m]
x0 = 2.0
ΔF = [f(x0 + Δx) - f(x0) for Δx in ΔX]
pinv(ΔX) * Δ
```

Выборки и линейная регрессия показаны на графике ниже. Наклон линии регрессии близок к точному решению 4:



11.3. Отношение правдоподобия

Метод *отношения правдоподобия* (likelihood ratio)[8] использует аналитическую форму $\nabla\pi_{\boldsymbol{\theta}}$ для улучшения нашего расчета градиента $\nabla U(\boldsymbol{\theta})$. Вспомним, что из уравнения (10.2) следует

[8] P. W. Glynn, *Likelihood Ratio Gradient Estimation for Stochastic Systems*, Communications of the ACM, vol. 33, no. 10, pp. 75–84, 1990.

$$U(\boldsymbol{\theta}) = \int p_{\boldsymbol{\theta}}(\tau) R(\tau) \mathrm{d}\tau. \tag{11.9}$$

Следовательно,

$$\nabla U(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} \int p_{\boldsymbol{\theta}}(\tau) R(\tau) \mathrm{d}\tau \tag{11.10}$$

$$= \int \nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(\tau) R(\tau) \mathrm{d}\tau \tag{11.11}$$

$$= \int p_{\boldsymbol{\theta}}(\tau) \frac{\nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(\tau)}{p_{\boldsymbol{\theta}}(\tau)} R(\tau) \mathrm{d}\tau \tag{11.12}$$

$$= \mathbb{E}_{\tau}\left[\frac{\nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(\tau)}{p_{\boldsymbol{\theta}}(\tau)} R(\tau) \right]. \tag{11.13}$$

Название этого метода происходит от отношения правдоподобия траектории. Это отношение правдоподобия можно рассматривать как вес в выборке, взвешенной по правдоподобию (раздел 3.7), по отношению к вознаграждениям за траекторию.

Применяя прием с логарифмической производной[9], мы имеем

$$\nabla U(\boldsymbol{\theta}) = \mathbb{E}_{\tau}[\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau) R(\tau)]. \tag{11.14}$$

Мы можем найти ожидаемое значение, используя развертывание траектории. Для каждой траектории τ нам нужно вычислить произведение $\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau)$ $R(\tau)$. Напомним, что $R(\tau)$ – это доход, связанный с траекторией τ. Если мы имеем дело со стохастической стратегией[10], градиент $\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau)$ равен

$$\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau) = \sum_{k=1}^{d} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a^{(k)} | s^{(k)}), \tag{11.15}$$

поскольку $p_{\boldsymbol{\theta}}(\tau)$ принимает вид

$$p_{\boldsymbol{\theta}}(\tau) = p(s^{(1)}) \prod_{k=1}^{d} T(s^{(k+1)} | s^{(k)}, a^{(k)}) \pi_{\boldsymbol{\theta}}(a^{(k)} | s^{(k)}), \tag{11.16}$$

где $s^{(k)}$ и $a^{(k)}$ – k-е состояние и действие соответственно на траектории τ. Алгоритм 11.4 обеспечивает реализацию, в которой выбирается m траекторий для получения оценки градиента. Пример 11.3 иллюстрирует этот процесс.

---

[9] Прием с логарифмической производной был представлен в разделе 10.5. Он использует следующее равенство: $\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau) = \nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(\tau) / p_{\boldsymbol{\theta}}(\tau)$.

[10] Мы используем $\pi_{\boldsymbol{\theta}}(a|s)$ для представления вероятности (точнее, ее плотности или массы), которую стратегия $\pi_{\boldsymbol{\theta}}$ присваивает выполнению действия a из состояния s.

Алгоритм 11.4. Метод нахождения градиента для стратегии π(s) задачи MDP $\mathcal{P}$ с распределением начального состояния b с использованием приема отношения правдоподобия. Градиент относительно вектора параметризации θ оценивается путем m развертываний до глубины d с использованием логарифмических градиентов стратегии ∇logπ

```
struct LikelihoodRatioGradient
    𝒫      # задача
    b      # начальное распределение по состояниям
    d      # глубина
    m      # количество выборок
    ∇logπ  # градиент логарифмического правдоподобия
end

function gradient(M::LikelihoodRatioGradient, π, θ)
    𝒫, b, d, m, ∇logπ, γ = M.𝒫, M.b, M.d, M.m, M.∇logπ, M.𝒫.γ
    πθ(s) = π(θ, s)
    R(τ) = sum(r*γ^(k-1) for (k, (s,a,r)) in enumerate(τ))
    ∇U(τ) = sum(∇logπ(θ, a, s) for (s,a) in τ)*R(τ)
    return mean(∇U(simulate(𝒫, rand(b), πθ, d)) for i in 1:m)
end
```

Пример 11.3. Применение приема с отношением правдоподобия для оценки градиента стратегии в простой задаче

Рассмотрим одношаговую задачу с одним состоянием из примера 11.1. Предположим, у нас есть стохастическая стратегия $\pi_{\boldsymbol{\theta}}$, которая производит выборку своего действия в соответствии с гауссовым распределением $\mathcal{N}(\theta_1, \theta_2^2)$, где θ_2^2 – дисперсия.

$$\log \pi_{\boldsymbol{\theta}}(a|s) = \log\left(\frac{1}{\sqrt{2\pi\theta_2^2}}\exp\left(-\frac{(a-\theta_1)^2}{2\theta_2^2}\right)\right)$$

$$= -\frac{(a-\theta_1)^2}{2\theta_2^2} - \frac{1}{2}\log(2\pi\theta_2^2).$$

Градиент логарифмической функции правдоподобия стратегии:

$$\frac{\partial}{\partial\theta_1}\log \pi_{\boldsymbol{\theta}}(a|s) = \frac{a-\theta_1}{\theta_2^2};$$

$$\frac{\partial}{\partial\theta_2}\log \pi_{\boldsymbol{\theta}}(a|s) = -\frac{(a-\theta_1)^2-\theta_2^2}{\theta_2^3}.$$

Предположим, мы запускаем три развертывания с $\boldsymbol{\theta} = [0,1]$, предпринимая действия {0.5, −1, 0.7} и получая одинаковые вознаграждения ($R(s, a) = a$). Ожидаемый градиент стратегии будет следующим:

$$\begin{aligned}\nabla U(\boldsymbol{\theta}) &\approx \frac{1}{m}\sum_{i=1}^{m}\nabla_{\boldsymbol{\theta}}\log p_{\boldsymbol{\theta}}(\tau^{(i)})R(\tau^{(i)}) \\ &= \frac{1}{3}\left(\begin{bmatrix}0.5\\-0.75\end{bmatrix}0.5+\begin{bmatrix}-1.0\\0.0\end{bmatrix}(-1)+\begin{bmatrix}0.7\\-0.51\end{bmatrix}0.7\right) \\ &= [0.58,-0.244].\end{aligned}$$

В случае детерминированной стратегии нахождение градиента основано на следующих вычислениях[11]:

$$\nabla_{\boldsymbol{\theta}}\log p_{\boldsymbol{\theta}}(\tau) = \nabla_{\boldsymbol{\theta}}\log\left[p(s^{(1)})\prod_{k=1}^{d}T(s^{(k+1)}|s^{(k)},\pi_{\boldsymbol{\theta}}(s^{(k)}))\right] \tag{11.17}$$

$$= \sum_{k=1}^{d}\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(s^{(k)})\frac{\partial}{\partial a^{(k)}}\log T(s^{(k+1)}|s^{(k)},a^{(k)}). \tag{11.18}$$

Уравнения (11.17) и (11.18) требуют знания правдоподобия перехода, что отличается от уравнения (11.15) для стохастических стратегий.

11.4. Предстоящее вознаграждение

Метод нахождения градиента через отношения правдоподобия не вносит систематическую ошибку (смещение), но имеет высокую дисперсию. В примере 11.4 рассматривается наличие систематической ошибки и дисперсии. Дисперсия обычно значительно увеличивается с глубиной развертывания из-за корреляции между действиями, состояниями и вознаграждениями на разных временных этапах. Метод *предстоящего вознаграждения* (reward-to-go) пытается уменьшить дисперсию получаемых значений.

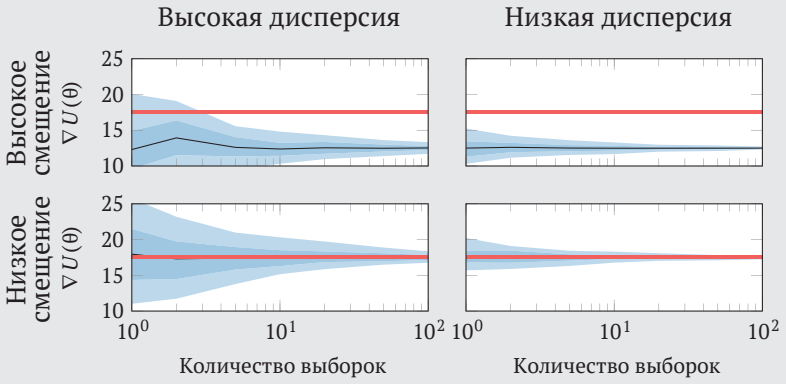
Пример 11.4. Эмпирическая демонстрация систематической ошибки и дисперсии при нахождении $\nabla U(\boldsymbol{\theta})$

При вычислении какого-либо значения обычно стараются использовать схему, которая обеспечивает как низкую систематическую ошибку, так и низкую дисперсию. В этой главе мы стремимся найти $\nabla U(\boldsymbol{\theta})$. Как правило, с увеличением количества смоделированных выборок удается получить более качественную оценку. Некоторые методы могут привести к систематиче-

[11] Многие задачи имеют векторнозначные действия $a \in \mathbb{R}^n$. В этом случае $\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(s^{(k)})$ заменяется матрицей Якоби, j-й столбец которой представляет собой градиент относительно j-й компоненты действия, а $\frac{\partial}{\partial a^{(k)}}\log T(s^{(k+1)}|s^{(k)},a^{(k)})$ заменяется градиентом действия.

ской ошибке даже при бесконечном количестве выборок, что не позволяет получить точную оценку. Тем не менее методы с ненулевым смещением все еще могут быть привлекательными, если они имеют низкую дисперсию, а это означает, что для их сходимости требуется меньше выборок.

Ниже представлены графики дисперсии оценок, полученных четырьмя условными методами вычисления $\nabla U(\boldsymbol{\theta})$. Истинное значение показано красными линиями и составляет 17,5. Мы выполнили 100 прогонов моделирования по 100 раз для каждого метода. Дисперсия уменьшается по мере увеличения количества выборок. Синие области обозначают эмпирические квантили оценок от 5 % до 95 % и от 25 % до 75 %.



Чтобы получить формальное представление этого метода, мы начнем с расширения уравнения (11.14):

$$\nabla U(\boldsymbol{\theta}) = \mathbb{E}_{\tau}\left[\left(\sum_{k=1}^{d} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a^{(k)}|s^{(k)})\right)\left(\sum_{k=1}^{d} r^{(k)}\gamma^{k-1}\right)\right]. \tag{11.19}$$

Заменим $\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a^{(k)}|s^{(k)})$ на $f(k)$ для удобства. Затем уравнение в следующем виде:

$$\nabla U(\boldsymbol{\theta}) = \mathbb{E}_{\tau}\left[\left(\sum_{k=1}^{d} f^{(k)}\right)\left(\sum_{k=1}^{d} r^{(k)}\gamma^{k-1}\right)\right] \tag{11.20}$$

$$= \mathbb{E}_{\tau}\left[(f^{(1)} + f^{(2)} + f^{(3)} + \cdots + f^{(d)})(r^{(1)} + r^{(2)}\gamma + r^{(3)}\gamma^2 + \cdots + r^{(d)}\gamma^{d-1}\right] \tag{11.21}$$

$$= \mathbb{E}_{\tau}\begin{bmatrix} f^{(1)}r^{(1)} + f^{(1)}r^{(2)}\gamma + f^{(1)}r^{(3)}\gamma^2 + \cdots + f^{(1)}r^{(d)}\gamma^{d-1} \\ + f^{(2)}r^{(1)} + f^{(2)}r^{(2)}\gamma + f^{(2)}r^{(3)}\gamma^2 + \cdots + f^{(2)}r^{(d)}\gamma^{d-1} \\ + f^{(3)}r^{(1)} + f^{(3)}r^{(2)}\gamma + f^{(3)}r^{(3)}\gamma^2 + \cdots + f^{(3)}r^{(d)}\gamma^{d-1} \\ \vdots \\ + f^{(d)}r^{(1)} + f^{(d)}r^{(2)}\gamma + f^{(d)}r^{(3)}\gamma^2 + \cdots + f^{(d)}r^{(d)}\gamma^{d-1} \end{bmatrix}. \tag{11.22}$$

На первое вознаграждение $r^{(1)}$ влияет только первое действие. Следовательно, его вклад в градиент не может зависеть от последующих временных шагов. Мы можем удалить остальные члены уравнения, нарушающие причинность, следующим образом[12]:

$$\nabla U(\boldsymbol{\theta}) = \mathbb{E}_\tau\left[\begin{array}{r} f^{(1)}r^{(1)} + f^{(1)}r^{(2)}\gamma + f^{(1)}r^{(3)}\gamma^2 + \cdots + f^{(1)}r^{(d)}\gamma^{d-1} \\ + f^{(2)}r^{(2)}\gamma + f^{(2)}r^{(3)}\gamma^2 + \cdots + f^{(2)}r^{(d)}\gamma^{d-1} \\ + f^{(3)}r^{(3)}\gamma^2 + \cdots + f^{(3)}r^{(d)}\gamma^{d-1} \\ \vdots \\ + f^{(d)}r^{(d)}\gamma^{d-1} \end{array}\right] \tag{11.23}$$

$$= \mathbb{E}_\tau\left[\sum_{k=1}^{d} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a^{(k)}|s^{(k)})\left(\sum_{\ell=k}^{d} r^{(\ell)}\gamma^{\ell-1}\right)\right] \tag{11.24}$$

$$= \mathbb{E}_\tau\left[\sum_{k=1}^{d} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a^{(k)}|s^{(k)})\left(\gamma^{k-1}\sum_{\ell=k}^{d} r^{(\ell)}\gamma^{\ell-1}\right)\right] \tag{11.25}$$

$$= \mathbb{E}_\tau\left[\sum_{k=1}^{d} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a^{(k)}|s^{(k)})\gamma^{k-1} r^{(k)}_{\text{to-go}}\right]. \tag{11.26}$$

Реализация этого метода представлена в алгоритме 11.5.

Обратите внимание, что предстоящее вознаграждение для пары состояние-действие (s, a) при стратегии, параметризованной значением $\boldsymbol{\theta}$, на самом деле является аппроксимацией полезности состояния-действия из этого состояния, т. е. $Q_{\boldsymbol{\theta}}(s, a)$. Функцию полезности действия, если она известна, можно использовать для получения градиента стратегии:

$$\nabla U(\boldsymbol{\theta}) = \mathbb{E}_\tau\left[\sum_{k=1}^{d} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a^{(k)}|s^{(k)})\gamma^{k-1} Q_{\boldsymbol{\theta}}(s^{(k)}, a^{(k)})\right]. \tag{11.27}$$

Алгоритм 11.5. Нахождение градиента стратегии `π(s)` для MDP $\mathcal{P}$ с начальным распределением состояний `b` по методу будущего вознаграждения. Градиент относительно вектора параметризации `θ` оценивается от `m` развертываний до глубины `d` с использованием градиента логарифмической стратегии `∇logπ`

```
struct RewardToGoGradient
    𝒫       # задача
    b       # начальное распределение по состояниям
    d       # глубина
    m       # количество выборок
```

---

[12] Член $\sum_{\ell=1}^{d} r^{(\ell)}\gamma^{\ell-1}$ часто называют *предстоящим вознаграждением* (reward-to-go) на шаге k.

```
    ∇logπ  # градиент логарифмического правдоподобия
end

function gradient(M::RewardToGoGradient, π, θ)
    𝒫, b, d, m, ∇logπ, γ = M.𝒫, M.b, M.d, M.m, M.∇logπ, M.𝒫.γ
    πθ(s) = π(θ, s)
    R(τ, j) = sum(r*γ^(k-1) for (k,(s,a,r)) in zip(j:d, τ[j:end]))
    ∇U(τ) = sum(∇logπ(θ, a, s)*R(τ,j) for (j, (s,a,r)) in enumerate(τ))
    return mean(∇U(simulate(𝒫, rand(b), πθ, d)) for i in 1:m)
end
```

11.5. Вычитание базисного значения

Мы можем развить подход, представленный в предыдущем разделе, вычитая *базисное значение* (baseline value) из будущего вознаграждения[13], чтобы уменьшить дисперсию искомого градиента. Это вычитание не смещает градиент.

Итак, мы вычитаем базисное значение $r_{\text{base}}(s^{(k)})$:

$$\nabla U(\boldsymbol{\theta}) = \mathbb{E}_{\tau}\left[\sum_{k=1}^{d} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a^{(k)}|s^{(k)})\gamma^{k-1}\left(r_{\text{to-go}}^{(k)} - r_{\text{base}}(s^{(k)})\right)\right]. \tag{11.28}$$

Чтобы показать, что вычитание базисного значения не смещает градиент, мы сначала развернем выражение:

$$\nabla U(\boldsymbol{\theta}) = \mathbb{E}_{\tau}\left[\sum_{k=1}^{d} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a^{(k)}|s^{(k)})\gamma^{k-1} r_{\text{to-go}}^{(k)} - \sum_{k=1}^{d} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a^{(k)}|s^{(k)})\gamma^{k-1} r_{\text{base}}(s^{(k)})\right]. \tag{11.29}$$

Линейность математического ожидания (linearity of expectation) означает, что $\mathbb{E}[a + b] = \mathbb{E}[a] + \mathbb{E}[b]$, поэтому достаточно доказать, что уравнение (11.29) эквивалентно уравнению (11.26), если для каждого шага k ожидаемый соответствующий базисный член равен $\mathbf{0}$:

$$\mathbb{E}_{\tau}[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a^{(k)}|s^{(k)})\gamma^{k-1} r_{\text{base}}(s^{(k)})] = 0. \tag{11.30}$$

Начнем с преобразования ожидания во вложенные ожидания, как показано на рис. 11.2:

$$\mathbb{E}_{\tau}\left[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a^{(k)}|s^{(k)})\gamma^{k-1} r_{\text{base}}(s^{(k)})\right]$$
$$= \mathbb{E}_{\tau_{1:k}}\left[\mathbb{E}_{\tau_{k+1:d}}\left[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a^{(k)}|s^{(k)})\gamma^{k-1} r_{\text{base}}(s^{(k)})\right]\right]. \tag{11.31}$$

[13] Мы также можем вычесть базисный уровень из значения состояния-действия.

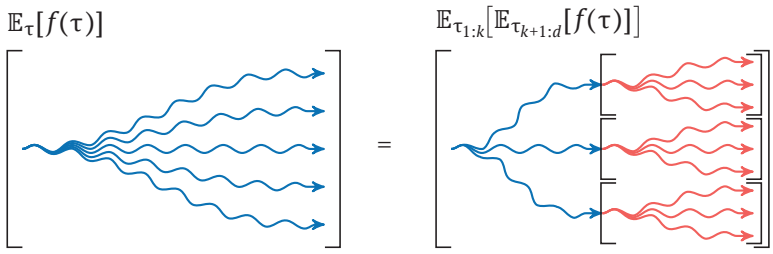


Рис. 11.2. Ожидание функции траекторий, выбранных из стратегии, можно рассматривать как ожидание относительно вложенных подтраекторий. Математический вывод приводится в упражнении 11.4

Мы продолжаем вывод, используя уже знакомый прием с логарифмической производной из раздела 11.3:

$$\mathbb{E}_{\tau_{1:k}}\left[\mathbb{E}_{\tau_{k+1:d}}\left[\nabla_{\boldsymbol{\theta}}\log\pi_{\boldsymbol{\theta}}(a^{(k)}|s^{(k)})\gamma^{k-1}r_{\text{base}}(s^{(k)})\right]\right]$$

$$= \mathbb{E}_{\tau_{1:k}}\left[\gamma^{k-1}r_{\text{base}}(s^{(k)})\mathbb{E}_{\tau_{k+1:d}}\left[\nabla_{\boldsymbol{\theta}}\log\pi_{\boldsymbol{\theta}}(a^{(k)}|s^{(k)})\right]\right] \tag{11.32}$$

$$= \mathbb{E}_{\tau_{1:k}}\left[\gamma^{k-1}r_{\text{base}}(s^{(k)})\mathbb{E}_{a^{(k)}}\left[\nabla_{\boldsymbol{\theta}}\log\pi_{\boldsymbol{\theta}}(a^{(k)}|s^{(k)})\right]\right] \tag{11.33}$$

$$= \mathbb{E}_{\tau_{1:k}}\left[\gamma^{k-1}r_{\text{base}}(s^{(k)})\int\nabla_{\boldsymbol{\theta}}\log\pi_{\boldsymbol{\theta}}(a^{(k)}|s^{(k)})\pi_{\boldsymbol{\theta}}(a^{(k)}|s^{(k)})\mathrm{d}a^{(k)}\right] \tag{11.34}$$

$$= \mathbb{E}_{\tau_{1:k}}\left[\gamma^{k-1}r_{\text{base}}(s^{(k)})\int\frac{\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(a^{(k)}|s^{(k)})}{\pi_{\boldsymbol{\theta}}(a^{(k)}|s^{(k)})}\pi_{\boldsymbol{\theta}}(a^{(k)}|s^{(k)})\mathrm{d}a^{(k)}\right] \tag{11.35}$$

$$= \mathbb{E}_{\tau_{1:k}}\left[\gamma^{k-1}r_{\text{base}}(s^{(k)})\nabla_{\boldsymbol{\theta}}\int\pi_{\boldsymbol{\theta}}(a^{(k)}|s^{(k)})\mathrm{d}a^{(k)}\right] \tag{11.36}$$

$$= \mathbb{E}_{\tau_{1:k}}\left[\gamma^{k-1}r_{\text{base}}(s^{(k)})\nabla_{\boldsymbol{\theta}}1\right] \tag{11.37}$$

$$= \mathbb{E}_{\tau_{1:k}}\left[\gamma^{k-1}r_{\text{base}}(s^{(k)})\mathbf{0}\right]. \tag{11.38}$$

Следовательно, вычитание члена $r_{\text{base}}(s^{(k)})$ не приводит к смещению. Данный вывод предполагал непрерывные пространства состояний и действий. Те же самые рассуждения применимы к дискретным пространствам.

Мы можем выбрать разные $r_{\text{base}}(s)$ для каждого компонента градиента, и выберем их таким образом, чтобы минимизировать дисперсию. Для простоты опустим зависимость от s и будем считать каждую базисную компоненту постоянной[14]. Для компактности записи уравнений в нашем выводе мы установим следующее обозначение:

[14] Некоторые методы аппроксимируют базисный уровень, зависящий от состояния, используя $r_{\text{base}}(s^{(k)}) = \boldsymbol{\varphi}(s^{(k)})^{\mathsf{T}}\mathbf{w}$. Выбрать подходящие базисные функции, как правило, довольно трудно. J. Peters, S. Schaal, *Reinforcement Learning of Motor Skills with Policy Gradients*, Neural Networks, vol. 21, no. 4, pp. 682–697, 2008.

$$\ell_i(a,s,k) = \gamma^{k-1}\frac{\partial}{\partial\theta_i}\log\pi_{\boldsymbol{\theta}}(a|s). \tag{11.39}$$

Дисперсия i-го компонента нашей оценки градиента в уравнении (11.28) равна

$$\mathop{\mathbb{E}}_{a,s,r_{\text{to-go}},k}\left[\left(\ell_i(a,s,k)(r_{\text{to-go}}-r_{\text{base},i})\right)^2\right] - \mathop{\mathbb{E}}_{a,s,r_{\text{to-go}},k}\left[\ell_i(a,s,k)(r_{\text{to-go}}-r_{\text{base},i})\right]^2, \tag{11.40}$$

где математическое ожидание берется по кортежам $(a, s, r_{\text{to-go}})$ в наших выборках траекторий, а k – глубина каждого кортежа.

Мы только что показали, что второй член равен нулю. Следовательно, мы можем сосредоточиться на выборе $r_{\text{base},i}$ для минимизации первого члена, взяв производную по базису и приравняв ее к нулю:

$$\begin{aligned}
&\frac{\partial}{\partial r_{\text{base},i}}\mathop{\mathbb{E}}_{a,s,r_{\text{to-go}},k}\left[\left(\ell_i(a,s,k)(r_{\text{to-go}}-r_{\text{base},i})\right)^2\right] \\
&\quad= \frac{\partial}{\partial r_{\text{base},i}}\Bigg(\mathop{\mathbb{E}}_{a,s,r_{\text{to-go}},k}\left[\ell_i(a,s,k)^2 r_{\text{to-go}}^2\right] \\
&\qquad - 2\mathop{\mathbb{E}}_{a,s,r_{\text{to-go}},k}\left[\ell_i(a,s,k)^2 r_{\text{to-go}} r_{\text{base},i}\right] + r_{\text{base},i}^2\mathop{\mathbb{E}}_{a,s,k}\left[\ell_i(a,s,k)^2\right]\Bigg) \qquad (11.41)\\
&\quad= -2\mathop{\mathbb{E}}_{a,s,r_{\text{to-go}},k}\left[\ell_i(a,s,k)^2 r_{\text{to-go}}\right] + 2r_{\text{base},i}\mathop{\mathbb{E}}_{a,s,k}\left[\ell_i(a,s,k)^2\right] = 0. \qquad (11.42)
\end{aligned}$$

Решение этого уравнения по $r_{\text{base},i}$ дает базисную компоненту, которая минимизирует дисперсию:

$$r_{\text{base},i} = \frac{\mathbb{E}_{a,s,r_{\text{to-go}},k}\left[\ell_i(a,s,k)^2 r_{\text{to-go}}\right]}{\mathbb{E}_{a,s,k}\left[\ell_i(a,s,k)^2\right]}. \tag{11.43}$$

Обычно на практике используют оценку градиента стратегии по методу отношения правдоподобия с вычитанием базисного значения (алгоритм 11.6)[15]. На рис. 11.3 сравниваются обсуждаемые здесь методы.

Алгоритм 11.6. Нахождение градиента методом отношения правдоподобия с будущим вознаграждением и вычитанием базисного значения для задачи MDP $\mathcal{P}$, стратегии π и начального распределения по состояниям `b`. Градиент относительно вектора параметризации θ оценивается путем `m` развертываний до глубины `d` с использованием логарифмических градиентов стратегии $\nabla\log\pi$

```
struct BaselineSubtractionGradient
    𝒫      # задача
```

---

[15] Эта комбинация используется в классе алгоритмов под названием REINFORCE, представленном в R. J. Williams, *Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning*, Machine Learning, vol. 8, pp. 229–256, 1992.

```
    b      # начальное распределение по состояниям
    d      # глубина
    m      # количество выборок
    ∇logπ  # градиент логарифмического правдоподобия
end

function gradient(M::BaselineSubtractionGradient, π, θ)
    𝒫, b, d, m, ∇logπ, γ = M.𝒫, M.b, M.d, M.m, M.∇logπ, M.𝒫.γ
    πθ(s) = π(θ, s)
    ℓ(a, s, k) = ∇logπ(θ, a, s)*γ^(k-1)
    R(τ, k) = sum(r*γ^(j-1) for (j,(s,a,r)) in enumerate(τ[k:end]))
    numer(τ) = sum(ℓ(a,s,k).^2*R(τ,k) for (k,(s,a,r)) in enumerate(τ))
    denom(τ) = sum(ℓ(a,s,k).^2 for (k,(s,a)) in enumerate(τ))
    base(τ) = numer(τ) ./ denom(τ)
    trajs = [simulate(𝒫, rand(b), πθ, d) for i in 1:m]
    rbase = mean(base(τ) for τ in trajs)
    ∇U(τ) = sum(ℓ(a,s,k).*(R(τ,k).-rbase) for (k,(s,a,r)) in enumerate(τ))
    return mean(∇U(τ) for τ in trajs)
end
```

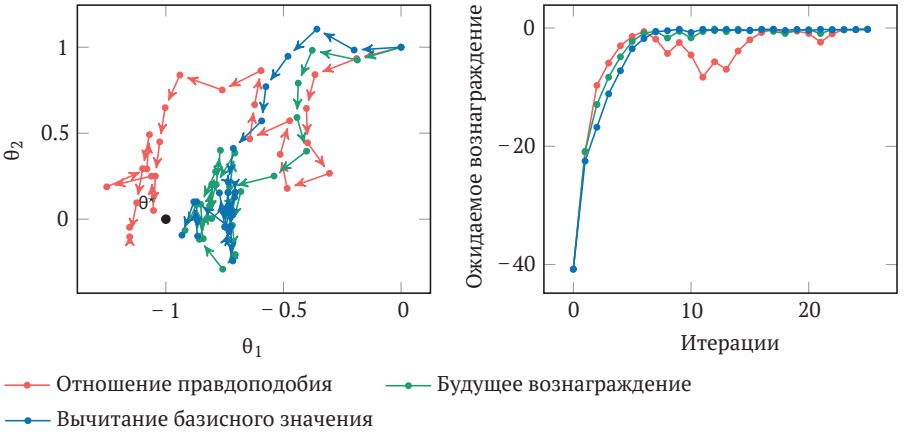


Рис. 11.3. Несколько методов нахождения градиента стратегий, используемых для оптимизации стратегий при решении простой задачи регулятора на основе одной и той же исходной параметризации. Каждая оценка градиента запускала шесть развертываний до глубины 10. Величина градиента была ограничена 1, а пошаговые обновления применялись с размером шага 0.2. Оптимальная параметризация стратегии показана черным цветом

В качественном отношении при рассмотрении вклада пар состояние-действие в градиент нас на самом деле интересует относительная полезность одного действия по сравнению с другим. Если все действия в определенном состоянии обладают одной и той же высокой полезностью, в градиенте отсутствует реальный сигнал обратной связи, и вычитание базисного значения может обнулить его. Нам нужно выявить действия, которые приносят более высокую полезность, чем другие, независимо от средней полезности действий.

Альтернативой полезности действия является *преимущество* (advantage) $A(s, a) = Q(s, a) - U(s)$. Использование функции полезности состояния при вычитании базисного значения дает преимущество. Градиент стратегии, использующий это преимущество, является несмещенным и обычно имеет гораздо меньшую дисперсию. Вычисление градиента принимает следующий вид:

$$\nabla U(\boldsymbol{\theta}) = \mathbb{E}_{\tau}\left[\sum_{k=1}^{d} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a^{(k)}|s^{(k)})\gamma^{k-1} A_{\boldsymbol{\theta}}(s^{(k)}, a^{(k)})\right]. \tag{11.44}$$

Как и в случае с функциями полезности состояния и действия, *функция преимущества* (advantage function) обычно неизвестна. Для ее приближенного вычисления необходимы другие методы, описанные в главе 13.

11.6. Заключение

- Градиент можно оценить с помощью метода конечных разностей.
- Для получения более надежных оценок градиента стратегии также можно использовать линейную регрессию.
- Метод отношения правдоподобия можно использовать для получения формы градиента стратегии, которая не зависит от модели перехода в случае стохастической стратегии.
- Дисперсия градиента стратегии может быть значительно уменьшена с помощью метода будущего вознаграждения и вычитания базисного значения.

11.7. Упражнения

Упражнение 11.1. Если мы вычисляем ожидаемую дисконтированную доходность данной параметрической стратегии $\pi_{\boldsymbol{\theta}}$, определенной n-мерным вектором параметров $\boldsymbol{\theta}$, используя m развертываний, сколько всего развертываний нам нужно выполнить, чтобы вычислить градиент стратегии с использованием метода конечных разностей?

Решение. Чтобы вычислить градиент стратегии с использованием метода конечных разностей, нам необходимо найти полезность стратегии при текущем векторе параметров $U(\boldsymbol{\theta})$, а также при всех n вариациях текущего вектора параметров $U(\boldsymbol{\theta} + \delta\mathbf{e}^{(i)})$, где $i = 1{:}n$. Поскольку мы оцениваем каждую вариацию с помощью m развертываний, нам нужно выполнить в общей сложности $m(n + 1)$ развертываний.

Упражнение 11.2. Предположим, у нас есть роботизированная рука, с помощью которой мы можем проводить эксперименты, манипулируя самыми разными объектами. Мы собираемся использовать градиент стратегии по методу отношения правдоподобия или одно из его дополнений для обучения страте-

гии, которая эффективна при захвате и перемещении этих объектов. Какую стратегию проще использовать – детерминированную или стохастическую? Почему?

Решение. Вычисление градиента стратегии по методу отношения правдоподобия требует явного представления вероятности перехода при использовании с детерминированными стратегиями. Указать точную явную модель перехода для реальной задачи управления роботом-манипулятором было бы непросто. Вычисление градиента для стохастической стратегии не требует явного представления вероятности перехода, что делает использование стохастической стратегии более простым.

Упражнение 11.3. Рассмотрим градиенты стратегии в виде:

$$\nabla_{\boldsymbol{\theta}} U(\boldsymbol{\theta}) = \mathbb{E}_{\tau}\left[\sum_{k=1}^{d} \gamma^{k-1} y \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a^{(k)} | s^{(k)})\right].$$

Какое из следующих значений y дает действительный градиент стратегии? Обоснуйте свой ответ.

a) $\sum_{\ell=1}^{\infty} r^{(\ell)}$;
b) $\sum_{\ell=1}^{\infty} r^{(\ell)} \gamma^{\ell-1}$;
c) $\left(\sum_{\ell=1}^{\infty} r^{(\ell)} \gamma^{\ell-1}\right) - r_{\text{base}}(s^{(k)})$;
d) $U(s^{(k)})$;
e) $Q(s^{(k)}, a^{(k)})$;
f) $A(s^{(k)}, a^{(k)})$;
g) $r^{(k)} + \gamma U(s^{(k+1)}) - U(s^{(k)})$.

Решение.

a) $\sum_{\ell=1}^{\infty} r^{(\ell)}$; приводит к следующему совокупному дисконтированному вознаграждению:

$$\gamma^{k-1} \gamma^{1-k} \sum_{\ell=1}^{\infty} r^{(\ell)} \gamma^{\ell-1} = \sum_{\ell=1}^{\infty} r^{(\ell)} \gamma^{\ell-1}$$

– и дает действительный градиент стратегии, как указано в уравнении (11.19).

b) $\sum_{\ell=1}^{\infty} r^{(\ell)} \gamma^{\ell-1}$; представляет собой метод будущего вознаграждения и дает действительный градиент стратегии, как указано в уравнении (11.26).

c) $\left(\sum_{\ell=1}^{\infty} r^{(\ell)} \gamma^{\ell-1}\right) - r_{\text{base}}(s^{(k)})$; представляет собой предстоящее вознаграждение за вычетом базисного значения и дает действительный градиент стратегии, как указано в уравнении (11.28).

d) $U(s^{(k)})$ представляет собой функцию полезности состояния и не дает действительного градиента стратегии.

e) $Q(s^{(k)}, a^{(k)})$ представляет собой функцию полезности действия состояния и дает действительный градиент стратегии, как указано в уравнении (11.27).

f) $A(s^{(k)}, a^{(k)})$ является функцией преимущества и дает действительный градиент стратегии, как указано в уравнении (11.44).
g) $r^{(k)} + \gamma U(s^{(k+1)}) - U(s^{(k)})$ – остаточная временная разница (будет обсуждаться далее в главе 13), которая дает допустимый градиент стратегии, поскольку является несмещенной аппроксимацией функции преимущества.

Упражнение 11.4. Докажите, что $\mathbb{E}_{\tau\sim\pi}[f(\tau)] = \mathbb{E}_{\tau_{1:k}\sim\pi}\left[\mathbb{E}_{\tau_{k:d}\sim\pi}[f(\tau)]\right]$ для шага k.

Решение. Вложенность ожиданий можно доказать, записав ожидание в интегральной форме и затем преобразовав обратно:

$$
\begin{aligned}
&\mathbb{E}_{\tau\sim\pi}[f(\tau)] = \\
&\quad = \int p(\tau) f(\tau) \mathrm{d}\tau \\
&\quad = \int \left(p(s^{(1)}) \prod_{k=1}^{d} p(s^{(k+1)}|s^{(k)}, a^{(k)}) \pi(a^{(k)}|s^{(k)}) \right) f(\tau) \mathrm{d}s^{(d)} \cdots \mathrm{d}a^{(2)} \mathrm{d}s^{(2)} \mathrm{d}a^{(1)} \mathrm{d}s^{(1)} \\
&\quad = \mathop{\mathbb{E}}_{\tau_{1:k}\sim\pi} \left[\int\int\int\int \cdots \int \left(\prod_{q=k}^{d} p(s^{(q+1)}|s^{(q)}, a^{(q)}) \pi(a^{(q)}|s^{(q)}) \right) f(\tau) \mathrm{d}s^{(d)} \cdots \mathrm{d}a^{(k+1)} \mathrm{d}s^{(k+1)} \mathrm{d}a^{(k)} \mathrm{d}s^{(k)} \right] \\
&\quad = \mathop{\mathbb{E}}_{\tau_{1:k}\sim\pi} \left[\mathop{\mathbb{E}}_{\tau_{k:d}\sim\pi} [f(\tau)] \right].
\end{aligned}
$$

Упражнение 11.5. Наша реализация нахождения градиента методом регрессии (алгоритм 11.3) подгоняет линейное отображение возмущений на разницу в доходах $U(\boldsymbol{\theta} + \Delta\boldsymbol{\theta}^{(i)}) - U(\boldsymbol{\theta})$. Мы находим $U(\boldsymbol{\theta} + \Delta\boldsymbol{\theta}^{(i)})$ и $U(\boldsymbol{\theta})$ для каждого из m возмущений, тем самым оценивая $U(\boldsymbol{\theta})$ в общей сложности m раз. Как мы можем перераспределить выборки более эффективным образом?

Решение. Один из подходов состоит в том, чтобы оценить $U(\boldsymbol{\theta})$ один раз и использовать одно и то же значение для каждого возмущения, тем самым выполняя только $m + 1$ оценку. Наличие точного значения $U(\boldsymbol{\theta})$ особенно важно для точной оценки градиента методом регрессии. В качестве альтернативы можно по-прежнему вычислять $U(\boldsymbol{\theta})$ один раз, но использовать m развертываний, сохраняя таким образом общее количество развертываний на итерацию. В этом подходе используется тот же объем вычислений, что и в алгоритме 11.3, но он может дать более надежную оценку градиента.

12 Оптимизация методом градиентного спуска по стратегиям

Для поиска оптимальной стратегии в пространстве параметров можно использовать *градиент по стратегиям*. В предыдущей главе были описаны методы нахождения этого градиента. В этой главе объясняется, как использовать полученные значения градиента для выполнения оптимизации различными способами. Мы начнем с градиентного подъема, который просто делает шаги в направлении градиента на каждой итерации. Выбор подходящего размера шага является серьезной проблемой. Большие шаги позволяют ускорить продвижение к оптимуму, но есть риск проскочить его. Метод натурального градиента изменяет текущее направление градиента, чтобы лучше справляться с различными уровнями зависимости от компонентов параметров. Главу завершает описание метода доверительной области, который, как и метод натурального градиента, начинается с получения стратегии-кандидата. Затем он ищет сегмент линии в пространстве стратегий, соединяющий исходную стратегию с этим кандидатом, чтобы найти лучшую стратегию.

12.1. Обновление стратегии методом градиентного подъема

Метод *градиентного подъема* (gradient ascent, подробнее рассмотрен в приложении А.11) применяется для поиска такой стратегии, параметризованной набором $\boldsymbol{\theta}$, которая максимизирует ожидаемую полезность $U(\boldsymbol{\theta})$. Градиентный подъем – это вариант метода *итеративного подъема* (iterated ascent), основанного на выполнении определенных шагов в пространстве параметров на каждой итерации в стремлении улучшить качество текущей стратегии. Все методы, обсуждаемые в этой главе, являются итеративными методами подъема, но различаются способом выполнения шагов. Метод градиентного подъема,

о котором пойдет речь в этом разделе, делает шаги в направлении градиента $\nabla U(\boldsymbol{\theta})$, который можно найти с помощью одного из методов, обсуждавшихся в предыдущей главе. Обновление $\boldsymbol{\theta}$ выполняется следующим образом:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \nabla U(\boldsymbol{\theta}), \tag{12.1}$$

где величина шага равна произведению коэффициента шага $\alpha > 0$ и величины градиента.

Алгоритм 12.1 реализует метод, который итеративно выполняет шаг градиентного подъема. Этот метод можно вызывать либо для фиксированного числа итераций, либо до тех пор, пока не сойдется $\boldsymbol{\theta}$ или $U(\boldsymbol{\theta})$. Градиентный подъем, как и другие алгоритмы, обсуждаемые в этой главе, не гарантирует сходимость к оптимальной стратегии. Однако существуют методы поощрения сходимости к *локально оптимальной* стратегии, относительно которой бесконечно малый шаг в пространстве параметров не может привести к лучшей стратегии. Один из подходов состоит в уменьшении коэффициента α с каждым шагом[1].

Алгоритм 12.1. Оптимизация стратегии методом градиентного подъема. Здесь выполняется шаг от точки `θ` в направлении градиента `∇U` с коэффициентом шага `α`. Для вычисления `∇U` можно использовать один из методов, описанных в предыдущей главе

```
struct PolicyGradientUpdate
    ∇U # расчетный градиент стратегии
    α  # коэффициент шага
end

function update(M::PolicyGradientUpdate, θ)
    return θ + M.α * M.∇U(θ)
end
```

Очень большие градиенты, как правило, проскакивают оптимум и могут возникать по разным причинам. Вознаграждения в некоторых задачах, например в игре «2048» (приложение F.2), могут различаться на порядки. Один из подходов к решению этой проблемы заключается в использовании *масштабирования градиента* (gradient scaling), которое ограничивает величину градиента перед его использованием для обновления параметров стратегии. Градиенты обычно ограничиваются нормой L_2, равной 1. Другим подходом является *отсечение градиента* (gradient clipping), которое выполняет поэлементное ограничение градиента перед его использованием для обновления стратегии. Отсечение обычно ограничивает значения в пределах ±1. Оба метода реализованы в алгоритме 12.2.

---

[1] Этот подход, как и многие другие, подробно описан в M. J. Kochenderfer, T. A. Wheeler, *Algorithms for Optimization*. MIT Press, 2019.

Алгоритм 12.2. Методы масштабирования и отсечения градиента. Масштабирование ограничивает величину предоставленного вектора градиента ∇ значением `L2_max`. Отсечение градиента обеспечивает поэлементное отсечение вектора градиента ∇ в диапазон между `a` и `b`

```
scale_gradient(∇, L2_max) = min(L2_max/norm(∇), 1)*∇
clip_gradient(∇, a, b) = clamp.(∇, a, b)
```

Масштабирование и отсечение различаются механизмом влияния на конечное направление градиента, как показано на рис. 12.1. Масштабирование оставляет направление неизменным, в то время как отсечение влияет на каждый компонент отдельно. Выбор одного из этих двух приемов зависит от задачи. Например, если в векторе градиента существенно преобладает один компонент, масштабирование обнуляет другие компоненты, что не всегда приемлемо.

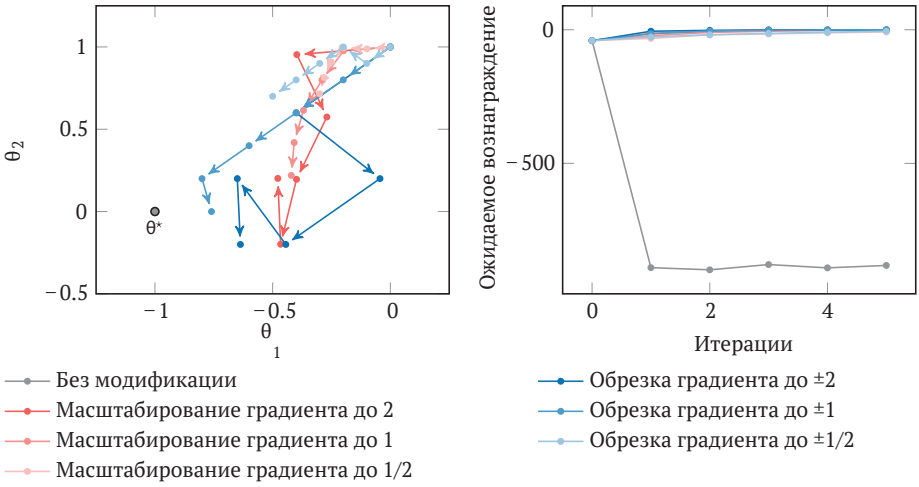


Рис. 12.1. Влияние масштабирования и отсечения градиента применительно к задаче простого регулятора. Для каждой оценки градиента выполнялось 10 развертываний до глубины 10. Пошаговые обновления применялись с размером шага 0,2. Оптимальная параметризация стратегии показана черным цветом

12.2. Ограниченное обновление градиента

Остальные алгоритмы в этой главе пытаются оптимизировать аппроксимацию целевой функции $U(\boldsymbol{\theta})$, соблюдая ограничение, согласно которому параметры $\boldsymbol{\theta}'$ стратегии на следующем шаге не слишком далеки от $\boldsymbol{\theta}$ на текущем шаге. Ограничение имеет вид $g(\boldsymbol{\theta}, \boldsymbol{\theta}') \leqslant \varepsilon$, где $\varepsilon > 0$ – свободный параметр алгоритма. Методы различаются аппроксимацией $U(\boldsymbol{\theta})$ и формой g. В этом разделе описывается простой метод *ограниченного шага* (restricted step).

Воспользуемся *аппроксимацией Тейлора* первого порядка (приложение А.12), полученной из нашей оценки градиента в точке $\boldsymbol{\theta}$, для вычисления приближенного значения U:

$$U(\boldsymbol{\theta}') \approx U(\boldsymbol{\theta}) + \nabla U(\boldsymbol{\theta})^\top(\boldsymbol{\theta}' - \boldsymbol{\theta}). \tag{12.2}$$

Установим следующее ограничение:

$$g(\boldsymbol{\theta}, \boldsymbol{\theta}') = \tfrac{1}{2}(\boldsymbol{\theta}' - \boldsymbol{\theta})^\top \mathbf{I}(\boldsymbol{\theta}' - \boldsymbol{\theta}) = \tfrac{1}{2}\|\boldsymbol{\theta}' - \boldsymbol{\theta}\|_2^2. \tag{12.3}$$

Мы можем рассматривать выражение (12.3) как ограничение длины шага значением, не превышающим $\sqrt{2\varepsilon}$. Другими словами, возможная область нашей оптимизации – это шар радиуса $\sqrt{2\varepsilon}$ с центром в $\boldsymbol{\theta}$.

Таким образом, цель оптимизации заключается в том, чтобы максимизировать

$$\max_{\boldsymbol{\theta}'} U(\boldsymbol{\theta}) + \nabla U(\boldsymbol{\theta})^\top(\boldsymbol{\theta}' - \boldsymbol{\theta}), \tag{12.4}$$

при условии что $\frac{1}{2}(\boldsymbol{\theta}' - \boldsymbol{\theta})^\top \mathbf{I}(\boldsymbol{\theta}' - \boldsymbol{\theta}) \leqslant \varepsilon$.

Мы можем исключить член $U(\boldsymbol{\theta})$ из уравнения цели, поскольку он не зависит от $\boldsymbol{\theta}'$. Кроме того, можем заменить в ограничении неравенство на равенство, поскольку линейная цель заставляет оптимальное решение находиться на границе допустимой области значений. Эти изменения приводят к эквивалентной задаче оптимизации:

$$\max_{\boldsymbol{\theta}'} \nabla U(\boldsymbol{\theta})^\top(\boldsymbol{\theta}' - \boldsymbol{\theta}), \tag{12.5}$$

при условии что $\frac{1}{2}(\boldsymbol{\theta}' - \boldsymbol{\theta})^\top \mathbf{I}(\boldsymbol{\theta}' - \boldsymbol{\theta}) = \varepsilon$.

Это уравнение оптимизации можно решить аналитически:

$$\boldsymbol{\theta}' = \boldsymbol{\theta} + \mathbf{u}\sqrt{\frac{2\varepsilon}{\mathbf{u}^\top\mathbf{u}}} = \boldsymbol{\theta} + \sqrt{2\varepsilon}\,\frac{\mathbf{u}}{\|\mathbf{u}\|}, \tag{12.6}$$

где ненормированное направление поиска $\mathbf{u}$ – это просто $\nabla U(\boldsymbol{\theta})$. Конечно, мы не знаем точное значение $\nabla U(\boldsymbol{\theta})$, но можем использовать любой из методов, описанных в предыдущей главе, для его вычисления. Реализация этого подхода представлена в алгоритме 12.3.

Алгоритм 12.3. Функция обновления для метода ограниченного градиента стратегии в точке `θ` для задачи $\mathcal{P}$ с распределением начального состояния `b`. Градиент оценивается от распределения начального состояния `b` до глубины `d` с помощью `m` симуляций параметрической стратегии `π(θ, s)` с логарифмическим градиентом стратегии `∇logπ`

```
struct RestrictedPolicyUpdate
    𝒫      # задача
    b      # начальное распределение по состояниям
    d      # глубина
    m      # количество выборок
    ∇logπ  # градиент логарифмического правдоподобия
    π      # стратегия
    ε      # граница дивергенции
end

function update(M::RestrictedPolicyUpdate, θ)
    𝒫, b, d, m, ∇logπ, π, γ = M.𝒫, M.b, M.d, M.m, M.∇logπ, M.π, M.𝒫.γ
    πθ(s) = π(θ, s)
    R(τ) = sum(r*γ^(k-1) for (k, (s,a,r)) in enumerate(τ))
    τs = [simulate(𝒫, rand(b), πθ, d) for i in 1:m]
    ∇log(τ) = sum(∇logπ(θ, a, s) for (s,a) in τ)
    ∇U(τ) = ∇log(τ)*R(τ)
    u = mean(∇U(τ) for τ in τs)
    return θ + u*sqrt(2*M.ε/dot(u,u))
end
```

12.3. Метод натурального градиента

Метод *натурального градиента* (natural gradient)[2] представляет собой разновидность метода ограниченного шага из предыдущего раздела, предназначенную для лучшей обработки ситуаций, когда стратегия более чувствительна к отдельным параметрам. *Чувствительность* (sensitivity) в данном контексте показывает, насколько полезность стратегии зависит от небольших изменений одного из параметров. Чувствительность градиентных методов во многом определяется выбором способа масштабирования параметров стратегии. Метод натурального градиента стратегии делает направление поиска $\mathbf{u}$ инвариантным к масштабированию параметров. На рис. 12.2 показаны различия между истинным и натуральным градиентами.

[2] S. Amari, *Natural Gradient Works Efficiently in Learning*, Neural Computation, vol. 10, no. 2, pp. 251–276, 1998.

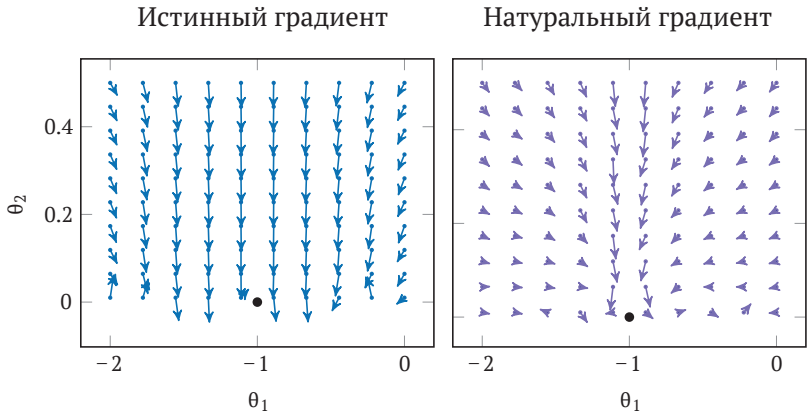


Рис. 12.2. Сравнение истинного и натурального градиентов в задаче простого регулятора (приложение F.5). Истинный градиент обычно строго указывает в отрицательном направлении θ_2, тогда как натуральный градиент обычно указывает на оптимум (черная точка) с координатами $[-1, 0]$. Аналогичный рисунок представлен в J. Peters, S. Schaal, *Reinforcement Learning of Motor Skills with Policy Gradients*, Neural Networks, vol. 21, no. 4, pp. 682–697, 2008

В методе натурального градиента используется то же приближение цели первого порядка, что и в предыдущем разделе. Ограничение, однако, другое. Здесь мы стремимся ограничить изменения $\boldsymbol{\theta}$, которые приводят к большим изменениям в распределении траекторий. Одним из способов измерения разницы между распределениями является вычисление *расхождения Кульбака–Лейблера* (Kullback-Leibler divergence, KL) (приложение A.10). Мы могли бы установить ограничение

$$g(\boldsymbol{\theta}, \boldsymbol{\theta}') = D_{\mathrm{KL}}(p(\cdot|\boldsymbol{\theta})||p(\cdot|\boldsymbol{\theta}')) \leqslant \varepsilon, \tag{12.7}$$

но вместо этого будем использовать приближение Тейлора второго порядка:

$$g(\boldsymbol{\theta}, \boldsymbol{\theta}') = \tfrac{1}{2}(\boldsymbol{\theta}' - \boldsymbol{\theta})^{\mathsf{T}}\mathbf{F}_{\boldsymbol{\theta}}(\boldsymbol{\theta}' - \boldsymbol{\theta}) \leqslant \varepsilon, \tag{12.8}$$

где *информационная матрица Фишера* выглядит так:

$$\mathbf{F}_{\boldsymbol{\theta}} = \int p(\tau|\boldsymbol{\theta})\nabla \log p(\tau|\boldsymbol{\theta})\nabla \log p(\tau|\boldsymbol{\theta})^{\mathsf{T}} \mathrm{d}\tau \tag{12.9}$$

$$= \mathbb{E}_{\tau}[\nabla \log p(\tau|\boldsymbol{\theta})\nabla \log p(\tau|\boldsymbol{\theta})^{\mathsf{T}}]. \tag{12.10}$$

В итоге цель оптимизации приобретает следующий вид:

$$\max_{\boldsymbol{\theta}'} \nabla U(\boldsymbol{\theta})^{\mathsf{T}}(\boldsymbol{\theta}' - \boldsymbol{\theta}), \tag{12.11}$$

при условии что $\tfrac{1}{2}(\boldsymbol{\theta}' - \boldsymbol{\theta})^{\mathsf{T}}\mathbf{F}_{\boldsymbol{\theta}}(\boldsymbol{\theta}' - \boldsymbol{\theta}) = \varepsilon$,

который идентичен уравнению (12.5), за исключением того, что вместо единичной матрицы $\mathbf{I}$ у нас есть матрица Фишера $\mathbf{F}_\theta$. Это дает нам допустимое множество параметров в форме эллипсоида. На рис. 12.3 показан пример в двумерном пространстве.

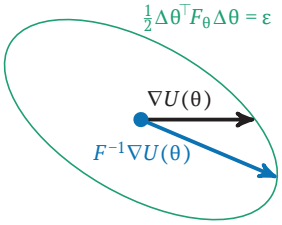


Рис. 12.3. Метод натурального градиента накладывает ограничение на аппроксимированное расхождение Кульбака–Лейблера. Это ограничение принимает форму эллипсоида. Поскольку эллипсоид может быть вытянут в определенных направлениях, это позволяет делать более крупные шаги при вращении градиента

Это уравнение оптимизации тоже можно решить аналитически по аналогии с уравнением в предыдущем разделе:

$$\boldsymbol{\theta}' = \boldsymbol{\theta} + \mathbf{u}\sqrt{\frac{2\varepsilon}{\nabla U(\boldsymbol{\theta})^\top \mathbf{u}}}, \tag{12.12}$$

за исключением того, что теперь мы используем[3]

$$\mathbf{u} = \mathbf{F}_\theta^{-1}\nabla U(\boldsymbol{\theta}). \tag{12.13}$$

Для оценки $\mathbf{F}_\theta$ и $\nabla U(\boldsymbol{\theta})$ можно использовать выборочные траектории. Реализация представлена в алгоритме 12.4.

Алгоритм 12.4. Функция обновления на основе метода натурального градиента при заданной стратегии `π(θ, s)` для задачи MDP $\mathcal{P}$ с распределением начального состояния `b`. Натуральный градиент по отношению к вектору параметров `θ` оценивается за `m` развертываний на глубину `d` с использованием логарифмических градиентов стратегии `∇logπ`. Вспомогательный метод `natural_update` выполняет обновление стратегии по списку траекторий в соответствии с уравнением (12.12) с заданным целевым градиентом `∇f(t)` и матрицей Фишера `F(t)`

```
struct NaturalPolicyUpdate
    𝒫     # задача
    b     # начальное распределение по состояниям
    d     # глубина
    m     # количество выборок
    ∇logπ # градиент логарифмического правдоподобия
```

---

[3] Это вычисление можно выполнить, используя сопряженный градиентный спуск, который сокращает объем вычислений, когда размерность $\boldsymbol{\theta}$ велика. S. M. Kakade, *A Natural Policy Gradient*, in Advances in Neural Information Processing Systems (NIPS), 2001.

```
    π      # стратегия
    ε      # граница дивергенции
end

function natural_update(θ, ∇f, F, ε, τs)
    ∇fθ = mean(∇f(τ) for τ in τs)
    u = mean(F(τ) for τ in τs) \ ∇fθ
    return θ + u*sqrt(2ε/dot(∇fθ,u))
end

function update(M::NaturalPolicyUpdate, θ)
    𝒫, b, d, m, ∇logπ, π, γ = M.𝒫, M.b, M.d, M.m, M.∇logπ, M.π, M.𝒫.γ
    πθ(s) = π(θ, s)
    R(τ) = sum(r*γ^(k-1) for (k, (s,a,r)) in enumerate(τ))
    ∇log(τ) = sum(∇logπ(θ, a, s) for (s,a) in τ)
    ∇U(τ) = ∇log(τ)*R(τ)
    F(τ) = ∇log(τ)*∇log(τ)'
    τs = [simulate(𝒫, rand(b), πθ, d) for i in 1:m]
    return natural_update(θ, ∇U, F, M.ε, τs)
end
```

12.4. Метод поиска в доверительной области

В этом разделе обсуждается метод поиска в *доверительной области* (trust region), ограниченной эллиптической допустимой областью из предыдущего раздела. Если точнее, этот подход называют *оптимизацией стратегии методом доверительной области* (trust region policy optimization, TRPO)[4]. Он работает, вычисляя следующую оценочную точку $\boldsymbol{\theta}'$, определяемую натуральным градиентом стратегии, а затем проводя *линейный поиск* вдоль сегмента линии, соединяющего $\boldsymbol{\theta}$ с $\boldsymbol{\theta}'$. Ключевая особенность фазы линейного поиска заключается в том, что оценка приблизительной цели и ограничение не требуют дополнительных прогонов моделирования.

На этапе линейного поиска мы больше не используем приближение первого порядка. Вместо этого мы используем приближение, полученное из равенства, включающего функцию преимущества[5]:

$$U(\boldsymbol{\theta}') = U(\boldsymbol{\theta}) + \mathop{\mathbb{E}}_{\tau \sim \pi_{\boldsymbol{\theta}'}}\left[\sum_{k=1}^{d} A_{\boldsymbol{\theta}}(s^{(k)}, a^{(k)})\right]. \tag{12.14}$$

[4] J. Schulman, S. Levine, P. Moritz, M. Jordan, P. Abbeel, *Trust Region Policy Optimization*, in International Conference on Machine Learning (ICML), 2015.

[5] Вариант этого равенства доказан в лемме 6.1 в докладе S. M. Kakade, J. Langford, *Approximately Optimal Approximate Reinforcement Learning*, International Conference on Machine Learning (ICML), 2002.

Другой способ записать этот шаг – использовать член $b_{\gamma,\theta}$, который представляет собой *дисконтированное распределение посещений* состояний s в соответствии со стратегией π_θ, где

$$b_{\gamma,\theta}(s) \propto P(s^{(1)} = s) + \gamma P(s^{(2)} = s) + \gamma^2 P(s^{(3)} = s) + \cdots. \tag{12.15}$$

При использовании дисконтированного распределения посещений уравнение цели приобретает следующий вид:

$$U(\boldsymbol{\theta}') = U(\boldsymbol{\theta}) + \mathop{\mathbb{E}}_{s\sim b_{\gamma,\boldsymbol{\theta}'}}\left[\mathop{\mathbb{E}}_{a\sim\pi_{\boldsymbol{\theta}'}(\cdot|s)}[A_{\boldsymbol{\theta}}(s,a)]\right]. \tag{12.16}$$

Разумеется, нам хотелось бы получить выборки из нашей стратегии, параметризованной $\boldsymbol{\theta}$ вместо $\boldsymbol{\theta}'$, чтобы не приходилось запускать больше прогонов моделирования во время линейного поиска. Выборки, связанные с внутренним ожиданием, можно заменить выборками из нашей первоначальной стратегии, если соответствующим образом взвесить преимущество[6]:

$$U(\boldsymbol{\theta}') = U(\boldsymbol{\theta}) + \mathop{\mathbb{E}}_{s\sim b_{\gamma,\boldsymbol{\theta}'}}\left[\mathop{\mathbb{E}}_{a\sim\pi_{\boldsymbol{\theta}'}(\cdot|s)}\left[\frac{\pi_{\boldsymbol{\theta}'}(a|s)}{\pi_{\boldsymbol{\theta}}(a|s)} A_{\boldsymbol{\theta}}(s,a)\right]\right]. \tag{12.17}$$

На следующем шаге мы заменяем распределение состояния на $b_{\gamma,\theta}$. Качество аппроксимации ухудшается по мере удаления $\boldsymbol{\theta}'$ от $\boldsymbol{\theta}$, но предполагается, что оно остается приемлемым в пределах доверительной области. Поскольку член $U(\boldsymbol{\theta})$ не зависит от $\boldsymbol{\theta}'$, мы можем исключить его из уравнения. Мы также можем исключить функцию полезности состояния из совокупной функции преимущества, оставив в ней лишь функцию полезности действия. Получившийся остаток называется *замещенной целью* (surrogate objective):

$$f(\boldsymbol{\theta}, \boldsymbol{\theta}') = \mathop{\mathbb{E}}_{s\sim b_{\gamma,\boldsymbol{\theta}}}\left[\mathop{\mathbb{E}}_{a\sim\pi_{\boldsymbol{\theta}}(\cdot|s)}\left[\frac{\pi_{\boldsymbol{\theta}'}(a|s)}{\pi_{\boldsymbol{\theta}}(a|s)} Q_{\boldsymbol{\theta}}(s,a)\right]\right]. \tag{12.18}$$

Это выражение может быть вычислено с использованием того же набора траекторий, который использовался для вычисления обновления натурального градиента. Мы можем найти $Q_{\boldsymbol{\theta}}(s, a)$, применяя метод предстоящего вознаграждения к выбранным траекториям[7].

---

[6] Это взвешивание происходит на основе выборки по важности, которая рассматривается в приложении А.14.

[7] Алгоритм 12.5 вместо этого использует $\sum_{\ell=k} r^{(\ell)}\gamma^{\ell-1}$, что эффективно снижает предстоящее вознаграждение на γ^{k-1}. Этот дисконт необходим для взвешивания вклада каждой выборки в соответствии с дисконтированным распределением посещений. Аналогичным образом дисконтируется замещенное ограничение.

Замещенное ограничение (surrogate constraint) в линейном поиске задается выражением

$$g(\boldsymbol{\theta}, \boldsymbol{\theta}') = \mathop{\mathbb{E}}_{s \sim b_{\gamma,\boldsymbol{\theta}}} \left[D_{KL}(\pi_{\boldsymbol{\theta}}(\cdot|s) || \pi_{\boldsymbol{\theta}'}(\cdot|s)) \right] \leq \varepsilon. \tag{12.19}$$

Линейный поиск подразумевает итеративную оценку нашей замещенной цели f и замещенного ограничения g для разных точек в пространстве стратегий. Начнем с точки $\boldsymbol{\theta}'$, полученной в результате того же процесса, что и обновление натурального градиента. Затем итеративно применяем действие

$$\boldsymbol{\theta}' \leftarrow \boldsymbol{\theta} + \alpha(\boldsymbol{\theta}' - \boldsymbol{\theta}) \tag{12.20}$$

до тех пор, пока не улучшим нашу цель до состояния $f(\boldsymbol{\theta}, \boldsymbol{\theta}') > f(\boldsymbol{\theta}, \boldsymbol{\theta})$ и не будет выполнено ограничительное условие $g(\boldsymbol{\theta}, \boldsymbol{\theta}') \leqslant \varepsilon$. Коэффициент шага $0 < \alpha < 1$ уменьшает расстояние между $\boldsymbol{\theta}$ и $\boldsymbol{\theta}'$ с каждой итерацией, при этом α обычно принимают равным 0.5.

В алгоритме 12.5 представлена реализация этого подхода. На рис. 12.4 показана взаимосвязь между областями допустимых решений, связанными с натуральным градиентом, и линейным поиском. На рис. 12.5 показано применение метода к задаче регулятора, а в примере 12.1 представлен пример простой задачи принятия решений.

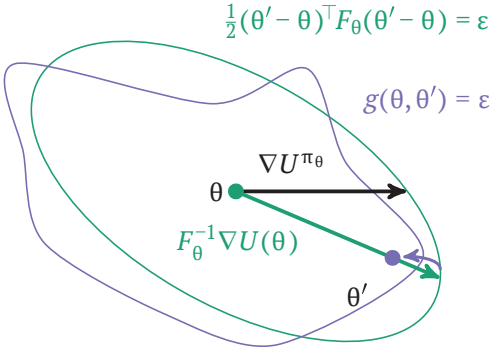


Рис. 12.4. Поиск стратегии оптимизации методом доверительной области в пределах эллиптического ограничения, созданного аппроксимацией второго порядка расхождения Кульбака–Лейблера. После вычисления направления подъема натурального градиента задается направление линейного поиска, чтобы убедиться, что обновленная стратегия улучшает вознаграждение и придерживается ограничения расхождения. Линейный поиск начинается с заданного максимального размера шага и уменьшает шаг в направлении подъема до тех пор, пока не будет найдена удовлетворительная точка

Алгоритм 12.5. Процедура обновления для оптимизации стратегии методом доверительной области, который представляет собой сочетание метода натурального градиента и линейного поиска. Этот метод генерирует `m` траекторий, используя стратегию `π` в задаче $\mathcal{P}$ с распределением начального состояния `b` и глубиной `d`. Чтобы получить начальную точку линейного поиска, нам нужен градиент логарифмической вероятности стратегии, генерирующей конкретное действие из текущего состояния, которое мы обозначим как `∇logπ`. Для замещенной цели нам нужна функция вероятности `p`, которая возвращает вероятность того, что наша стратегия порождает конкретное действие из текущего состояния. Для замещенного ограничения нам нужно расхождение между распределениями действий, порожденными π_θ и $\pi_{\theta'}$. На каждом шаге линейного поиска мы уменьшаем расстояние между рассматриваемыми точками `θ′` и `θ`, сохраняя направление поиска

```
struct TrustRegionUpdate
    𝒫     # задача
    b     # начальное распределение по состояниям
    d     # глубина
    m     # количество выборок
    π     # стратегия π(s)
    p     # правдоподобие стратегии p(θ, a, s)
    ∇logπ # логарифмический градиент правдоподобия
    KL    # дивергенция Кульбака-Лейблера KL(θ, θ′, s)
    ε     # граница дивергенции
    α     # коэффициент уменьшения расстояния (например, 0.5)
end

function surrogate_objective(M::TrustRegionUpdate, θ, θ′, τs)
    d, p, γ = M.d, M.p, M.𝒫.γ
    R(τ, j) = sum(r*γ^(k-1) for (k,(s,a,r)) in zip(j:d, τ[j:end]))
    w(a,s) = p(θ′,a,s) / p(θ,a,s)
    f(τ) = mean(w(a,s)*R(τ,k) for (k,(s,a,r)) in enumerate(τ))
    return mean(f(τ) for τ in τs)
end

function surrogate_constraint(M::TrustRegionUpdate, θ, θ′, τs)
    γ = M.𝒫.γ
    KL(τ) = mean(M.KL(θ, θ′, s)*γ^(k-1) for (k,(s,a,r)) in enumerate(τ))
    return mean(KL(τ) for τ in τs)
end

function linesearch(M::TrustRegionUpdate, f, g, θ, θ′)
    fθ = f(θ)
    while g(θ′) > M.ε || f(θ′) ≤ fθ
        θ′ = θ + M.α*(θ′ - θ)
    end
    return θ′
end
```

```
function update(M::TrustRegionUpdate, θ)
    𝒫, b, d, m, ∇logπ, π, γ = M.𝒫, M.b, M.d, M.m, M.∇logπ, M.π, M.𝒫.γ
    πθ(s) = π(θ, s)
    R(τ) = sum(r*γ^(k-1) for (k, (s,a,r)) in enumerate(τ))
    ∇log(τ) = sum(∇logπ(θ, a, s) for (s,a) in τ)
    ∇U(τ) = ∇log(τ)*R(τ)
    F(τ) = ∇log(τ)*∇log(τ)'
    τs = [simulate(𝒫, rand(b), πθ, d) for i in 1:m]
    θ′ = natural_update(θ, ∇U, F, M.ε, τs)
    f(θ′) = surrogate_objective(M, θ, θ′, τs)
    g(θ′) = surrogate_constraint(M, θ, θ′, τs)
    return linesearch(M, f, g, θ, θ′)
end
```

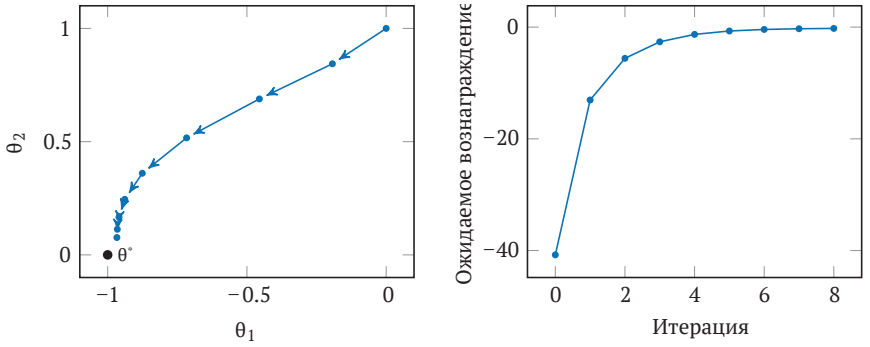


Рис. 12.5. Оптимизация стратегии по методу доверительной области применительно к задаче простого регулятора с развертыванием до глубины 10, $\varepsilon = 1$ и $c = 2$. Оптимальная параметризация стратегии показана черным цветом

Пример 12.1. Пример одной итерации оптимизации стратегии методом доверительной области

Рассмотрим применение TRPO к гауссовой стратегии $\mathcal{N}(\theta_1, \theta_2^2)$ из примера 11.3 и MDP с одним состоянием из примера 11.1 с $\gamma = 1$. Напомним, что градиент логарифмической функции правдоподобия равен

$$\frac{\partial}{\partial\theta_1}\log\pi_{\boldsymbol{\theta}}(a|s) = \frac{a-\theta_1}{\theta_2^2};$$

$$\frac{\partial}{\partial\theta_2}\log\pi_{\boldsymbol{\theta}}(a|s) = \frac{(a-\theta_1)^2-\theta_2^2}{\theta_2^3}.$$

Предположим, что мы выполняем два развертывания с $\boldsymbol{\theta} = [0,1]$ (эта задача имеет только одно состояние):

$\tau_1 = \{(a = r = -0.532), (a = r = \ \ 0.597), (a = r = 1.947)\}$;

$\tau_2 = \{(a = r = -0.263), (a = r = -2.212), (a = r = 2.364)\}$.

Информационная матрица Фишера вычисляется следующим образом:

$$\mathbf{F}_{\boldsymbol{\theta}} = \frac{1}{2}\left(\nabla \log p(\tau^{(1)}) \nabla \log p(\tau^{(1)})^{\top} + \nabla \log p(\tau^{(2)}) \nabla \log p(\tau^{(2)})^{\top}\right)$$
$$= \frac{1}{2}\left(\begin{bmatrix} 4.048 & 2.878 \\ 2.878 & 2.046 \end{bmatrix} + \begin{bmatrix} 0.012 & -0.838 \\ -0.838 & 57.012 \end{bmatrix}\right) = \begin{bmatrix} 2.030 & 1.020 \\ 1.019 & 29.529 \end{bmatrix}.$$

Градиент целевой функции равен [2.030, 1.020]. Результирующее направление спуска $\mathbf{u}$ равно [1, 0]. Задав $\varepsilon = 0.1$, мы вычисляем наш обновленный вектор параметризации и получаем $\boldsymbol{\theta}' = [0.314, 1]$.

Значение замещенной целевой функции в $\boldsymbol{\theta}$ равно 1.485. Линейный поиск начинается с $\boldsymbol{\theta}'$, где значение замещенной целевой функции равно 2.110, а ограничение равно 0.049. Это соответствует нашим требованиям (поскольку $0.049 < \varepsilon$), поэтому мы возвращаем новый набор параметров.

12.5. Зажатие замещенной цели

Мы можем избежать неблагоприятных обновлений стратегии из-за чрезмерно оптимистичных оценок замещенной цели по методу доверительной области путем процедуры *зажатия* (clamping)[8].

$$\mathop{\mathbb{E}}_{s\sim b_{\gamma,\boldsymbol{\theta}}}\left[\mathop{\mathbb{E}}_{a\sim\pi_{\boldsymbol{\theta}}(\cdot|s)}\left[\frac{\pi_{\boldsymbol{\theta}'}(a|s)}{\pi_{\boldsymbol{\theta}}(a|s)} A_{\boldsymbol{\theta}}(s,a)\right]\right]. \tag{12.12}$$

Отношение вероятностей $\pi_{\boldsymbol{\theta}'}(a|s)/\pi_{\boldsymbol{\theta}}(a|s)$ может быть чрезмерно оптимистичным. Пессимистичная нижняя граница цели может значительно улучшить точность:

$$\mathop{\mathbb{E}}_{s\sim b_{\gamma,\boldsymbol{\theta}}}\left[\mathop{\mathbb{E}}_{a\sim\pi_{\boldsymbol{\theta}}(\cdot|s)}\left[\min\left(\frac{\pi_{\boldsymbol{\theta}'}(a|s)}{\pi_{\boldsymbol{\theta}}(a|s)} A_{\boldsymbol{\theta}}(s,a), \operatorname{clamp}\left(\frac{\pi_{\boldsymbol{\theta}'}(a|s)}{\pi_{\boldsymbol{\theta}}(a|s)}, 1-\varepsilon, 1+\varepsilon\right) A_{\boldsymbol{\theta}}(s,a)\right)\right]\right], \tag{12.22}$$

где ε – маленькое положительное значение[9], а $\operatorname{clamp}(x, a, b)$ заставляет x оставаться между a и b. По умолчанию $\operatorname{clamp}(x, a, b) = \min\{\max\{x, a\}, b\}$.

Само по себе зажатие отношения вероятностей не дает нижней границы; надо взять минимум от зажатых и исходных значений цели. Нижняя граница

---

[8] Ограничение является ключевой идеей так называемой *оптимизации проксимальной стратегии* (proximal policy optimization, PPO), как обсуждалось в J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov, *Proximal Policy Optimization Algorithms*, 2017. arXiv: 1707.06347v2.

[9] Хотя здесь ε не действует напрямую как порог расхождения, как это было в предыдущих алгоритмах, его роль аналогична. Типичное значение равно 0.2.

показана на рис. 12.6 вместе с исходными и зажатыми целями. Конечным результатом наличия нижней границы является то, что изменение отношения вероятностей игнорируется, когда оно может привести к значительному улучшению цели. Таким образом, использование нижней границы предотвращает чрезмерные, часто вредные обновления стратегии и устраняет необходимость в уравнении ограничения замещенной доверительной области (12.19). Без этого ограничения мы также можем отказаться от линейного поиска и использовать стандартные методы градиентного подъема.

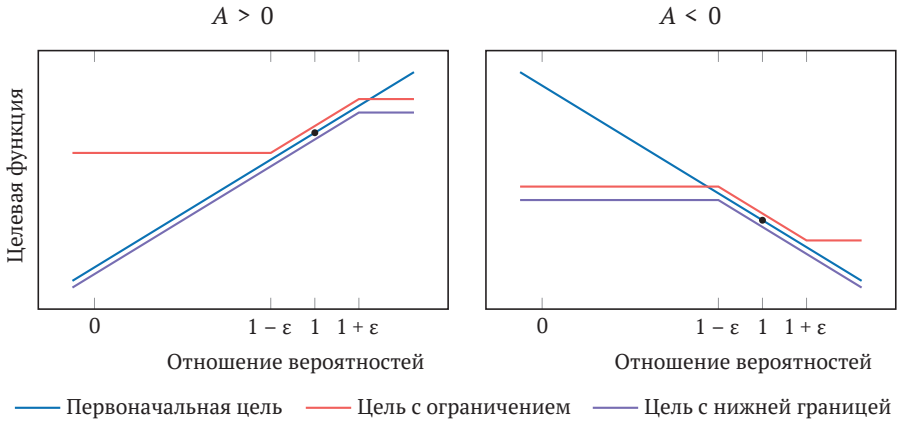


Рис. 12.6. Визуализация ограниченной снизу цели для положительных и отрицательных преимуществ по сравнению с исходной и зажатой целями. Черная точка показывает базисный уровень, вокруг которого выполняется оптимизация, $\pi_{\theta'}(a|s)/\pi_{\theta}(a|s) = 1$. Три линейных графика на каждой оси разнесены по вертикали для ясности

Градиент из уравнения цели без зажатия (12.21) с полезностями действия равен

$$\nabla_{\theta'} f(\theta, \theta') = \mathop{\mathbb{E}}_{s\sim b_{\gamma,\theta}}\left[\mathop{\mathbb{E}}_{a\sim\pi_{\theta}(\cdot|s)}\left[\frac{\nabla_{\theta'}\pi_{\theta'}(a|s)}{\pi_{\theta}(a|s)} Q_{\theta}(s,a)\right]\right], \tag{12.23}$$

где $Q_{\theta}(s, a)$ можно оценить по методу предстоящего вознаграждения. Градиент из уравнения цели с нижней границей (12.22) (с зажатием) такой же, за исключением того, что нет вклада от кортежей опыта, относительно которых и происходит активное зажатие. То есть вклад градиента равен нулю в случаях, когда предстоящее вознаграждение положительно и отношение вероятностей больше 1 + ε или предстоящее вознаграждение за выполнение отрицательно и отношение вероятностей меньше 1 – ε.

Как и в случае TRPO, градиент для параметризации $\boldsymbol{\theta}'$ можно вычислить на основе опыта, полученного из $\boldsymbol{\theta}$. Таким образом, несколько обновлений градиента могут быть выполнены подряд с использованием одного и того же набора выбранных траекторий. Реализация этого подхода представлена в алгоритме 12.6.

Зажатая замещенная цель сравнивается с несколькими другими замещенными целями на рис. 12.7, который содержит также линейный график эффективной цели по методу TRPO:

$$\mathop{\mathbb{E}}_{\substack{s\sim b_{\gamma,\theta}\\ a\sim\pi_\theta(\cdot|s)}}\left[\frac{\pi_{\theta'}(a|s)}{\pi_\theta(a|s)}A_\theta(s,a)-\beta D_{\mathrm{KL}}(\pi_\theta(\cdot|s)||\pi_{\theta'}(\cdot|s))\right], \tag{12.24}$$

что фактически является целью поиска стратегии по методу доверительной области, где ограничение реализуется как штраф в виде некоторого коэффициента β. TRPO обычно использует жесткое ограничение, а не штраф, потому что трудно подобрать значение β, которое хорошо работает в рамках даже одной задачи, не говоря уже о множестве задач.

Алгоритм 12.6. Реализация оптимизации стратегии с зажатой замещенной целью, которая возвращает новый набор параметров для стратегии `π(s)` в задаче MDP $\mathcal{P}$ с распределением начального состояния `b`. Данный алгоритм отбирает `m` траекторий на глубину `d`, а затем использует их для оценки градиента стратегии в `k_max` последующих обновлениях. Градиент стратегии с использованием зажатой цели вычисляется с использованием градиентов стратегии `∇p` с параметром зажатия ε

```
struct ClampedSurrogateUpdate
    𝒫     # задача
    b     # начальное распределение по состояниям
    d     # глубина
    m     # количество траекторий
    π     # стратегия
    p     # правдоподобие стратегии
    ∇π    # градиент правдоподобия стратегии
    ε     # граница дивергенции
    α     # размер шага
    k_max # количество итераций на обновление
end

function clamped_gradient(M::ClampedSurrogateUpdate, θ, θ′, τs)
    d, p, ∇π, ε, γ = M.d, M.p, M.∇π, M.ε, M.𝒫.γ
    R(τ, j) = sum(r*γ^(k-1) for (k,(s,a,r)) in zip(j:d, τ[j:end]))
    ∇f(a,s,r_togo) = begin
        P = p(θ, a,s)
        w = p(θ′,a,s) / P
        if (r_togo > 0 && w > 1+ε) || (r_togo < 0 && w < 1-ε)
            return zeros(length(θ))
        end
        return ∇π(θ′, a, s) * r_togo / P
    end
    ∇f(τ) = mean(∇f(a,s,R(τ,k)) for (k,(s,a,r)) in enumerate(τ))
    return mean(∇f(τ) for τ in τs)
```

```
end

function update(M::ClampedSurrogateUpdate, θ)
    𝒫, b, d, m, π, α, k_max= M.𝒫, M.b, M.d, M.m, M.π, M.α, M.k_max
    πθ(s) = π(θ, s)
    τs = [simulate(𝒫, rand(b), πθ, d) for i in 1:m]
    θ′ = copy(θ)
    for k in 1:k_max
        θ′ += α*clamped_gradient(M, θ, θ′, τs)
    end
    return θ′
end
```

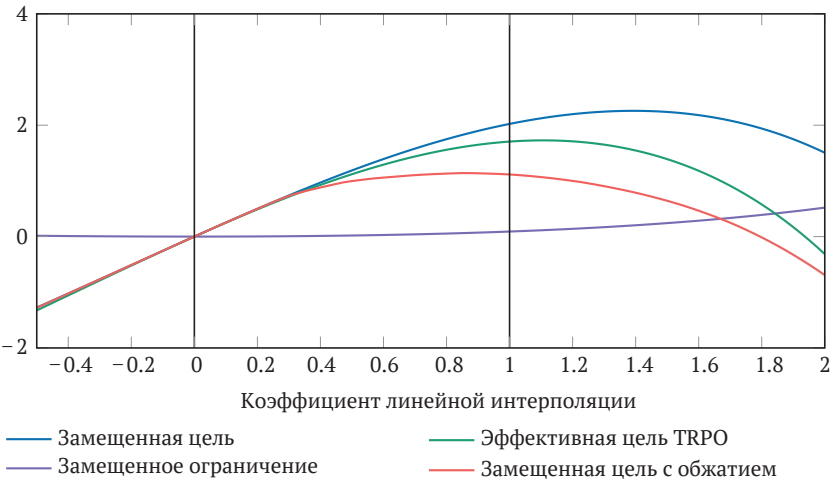


Рис. 12.7. Сравнение оптимизации стратегий с замещенной целью и замещенным ограничением в задаче линейно-квадратичного регулятора. На оси *x* показаны замещенные цели, когда мы перемещаемся от $\boldsymbol{\theta}$ в точке 0 к $\boldsymbol{\theta}'$, учитывая естественное обновление стратегии в 1. Замещающие цели были сведены к 0 путем вычитания значения замещенной целевой функции для $\boldsymbol{\theta}$. Мы видим, что замещенная цель с зажатием ведет себя очень похоже на эффективную цель TRPO без зажатия. Обратите внимание, что ε и β могут быть скорректированы для обоих алгоритмов, что повлияет на то, где находится максимум в каждом случае

12.6. Заключение

- Алгоритм градиентного подъема может использовать значения градиента, полученные с помощью методов, рассмотренных в предыдущей главе, для итеративного улучшения стратегии.
- Градиентный подъем можно сделать более устойчивым путем масштабирования, отсечения или переменного шага.

- В методе натурального градиента используется приближение первого порядка целевой функции и применяется аппроксимация с использованием информационной матрицы Фишера.
- Оптимизация стратегий методом доверительной области предусматривает дополнение метода натурального градиента линейным поиском для дальнейшего улучшения стратегии без моделирования траектории.
- Мы можем использовать пессимистическую нижнюю границу цели TRPO, чтобы получить замещенную цель с зажатием, которая работает аналогично без необходимости линейного поиска.

12.7. Упражнения

Упражнение 12.1. TRPO начинает линейный поиск с новой параметризации, заданной обновлением натурального градиента стратегии. Однако TRPO проводит линейный поиск, используя цель, отличную от той, на которую указывает натуральный градиент. Покажите, что градиент из уравнения замещенной цели (12.18), используемой в TRPO, на самом деле такой же, как и в уравнении по методу предстоящего вознаграждения (11.26).

Решение. Градиент замещенной цели TRPO равен

$$\nabla_{\boldsymbol{\theta}'} U_{\text{TRPO}} = \mathop{\mathbb{E}}_{s\sim b_{\gamma,\boldsymbol{\theta}}}\left[\mathop{\mathbb{E}}_{a\sim\pi_{\boldsymbol{\theta}}(\cdot|s)}\left[\frac{\nabla_{\boldsymbol{\theta}'}\pi_{\boldsymbol{\theta}'}(a|s)}{\pi_{\boldsymbol{\theta}}(a|s)} Q_{\boldsymbol{\theta}}(s,a)\right]\right].$$

При начальном обновлении натурального градиента направление поиска оценивается как $\boldsymbol{\theta}' = \boldsymbol{\theta}$. Кроме того, полезность действия аппроксимируется по методу предстоящего вознаграждения:

$$\nabla_{\boldsymbol{\theta}'} U_{\text{TRPO}} = \mathop{\mathbb{E}}_{s\sim b_{\gamma,\boldsymbol{\theta}}}\left[\mathop{\mathbb{E}}_{a\sim\pi_{\boldsymbol{\theta}}(\cdot|s)}\left[\frac{\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(a|s)}{\pi_{\boldsymbol{\theta}}(a|s)} r_{\text{to-go}}\right]\right].$$

Напомним, что производная от $\log f(x)$ равна $f'(x)/f(x)$. Отсюда следует уравнение

$$\nabla_{\boldsymbol{\theta}'} U_{\text{TRPO}} = \mathop{\mathbb{E}}_{s\sim b_{\gamma,\boldsymbol{\theta}}}\left[\mathop{\mathbb{E}}_{a\sim\pi_{\boldsymbol{\theta}}(\cdot|s)}\left[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a|s) r_{\text{to-go}}\right]\right],$$

которое принимает ту же форму, что и уравнение градиента стратегии при использовании метода предстоящего вознаграждения (11.26).

Упражнение 12.2. Выполните расчеты из примера 12.1. Сначала найдите обратную информационную матрицу Фишера $\mathbf{F}_{\boldsymbol{\theta}}^{-1}$, затем вычислите $\mathbf{u}$ и обновленные параметры $\boldsymbol{\theta}'$.

Решение. Начнем с вычисления обратной информационной матрицы Фишера:

$$\mathbf{F}_{\boldsymbol{\theta}}^{-1} \approx \frac{1}{0.341(29.529)-0.332(0.332)}\begin{bmatrix} 29.529 & -0.332 \\ -0.332 & 0.341 \end{bmatrix} \approx \begin{bmatrix} 0.501 & -0.017 \\ -0.017 & 0.034 \end{bmatrix}.$$

Теперь обновим $\mathbf{u}$ следующим образом:

$$\mathbf{u} = \mathbf{F}_{\boldsymbol{\theta}}^{-1}\nabla U(\theta) \approx \begin{bmatrix} 0.501 & -0.017 \\ -0.017 & 0.034 \end{bmatrix}\begin{bmatrix} 2.030 \\ 1.020 \end{bmatrix} \approx \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

Наконец, оценим обновленные параметры $\boldsymbol{\theta}$:

$$\begin{aligned} \boldsymbol{\theta}' &= \boldsymbol{\theta} + \mathbf{u}\sqrt{\frac{2\varepsilon}{\nabla U(\boldsymbol{\theta})^{\mathsf{T}}\mathbf{u}}} \\ &\approx \begin{bmatrix} 0 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix}\sqrt{\frac{2(0.1)}{\begin{bmatrix} 2.030 & 1.020 \end{bmatrix}\begin{bmatrix} 1 \\ 0 \end{bmatrix}}} \\ &\approx \begin{bmatrix} 0 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix}\sqrt{\frac{0.2}{2.030}} \\ &\approx \begin{bmatrix} 0.314 \\ 1 \end{bmatrix}. \end{aligned}$$

Упражнение 12.3. Предположим, нам даны параметрические стратегии $\pi_{\boldsymbol{\theta}}$ и $\pi_{\boldsymbol{\theta}'}$, указанные в следующей таблице:

	a_1	a_2	a_3	a_4
$\pi_{\boldsymbol{\theta}}(a\|s_1)$	0.1	0.2	0.3	0.4
$\pi_{\boldsymbol{\theta}'}(a\|s_1)$	0.4	0.3	0.2	0.1
$\pi_{\boldsymbol{\theta}}(a\|s_2)$	0.1	0.1	0.6	0.2
$\pi_{\boldsymbol{\theta}'}(a\|s_2)$	0.1	0.1	0.5	0.3

Полагая, что выбраны следующие пять состояний: s_1, s_2, s_1, s_1, s_2, – аппроксимируйте $\mathbb{E}_s[D_{\mathrm{KL}}(\pi_{\boldsymbol{\theta}}(\cdot|s)||\pi_{\boldsymbol{\theta}'}(\cdot|s))]$, используя определение

$$D_{\mathrm{KL}}(P||Q) = \sum_x P(x)\log\frac{P(x)}{Q(x)}.$$

Решение. Сначала вычислим расхождение KL для выборки состояния s_1:

$$D_{\mathrm{KL}}(\pi_{\boldsymbol{\theta}}(\cdot|s_1)\|\pi_{\boldsymbol{\theta}'}(\cdot|s_1) = 0.1\log\left(\frac{0.1}{0.4}\right)+0.2\log\left(\frac{0.2}{0.3}\right) \\ +0.3\log\left(\frac{0.3}{0.3}\right)+0.4\log\left(\frac{0.4}{0.1}\right)\approx 0.456.$$

Затем вычислим расхождение KL для выборки состояния s_2:

$$D_{\mathrm{KL}}(\pi_{\boldsymbol{\theta}}(\cdot|s_2)\|\pi_{\boldsymbol{\theta}'}(\cdot|s_2) = 0.1\log\left(\frac{0.1}{0.1}\right)+0.1\log\left(\frac{0.1}{0.1}\right) \\ +0.6\log\left(\frac{0.6}{0.5}\right)+0.2\log\left(\frac{0.2}{0.3}\right)\approx 0.0283.$$

Наконец, вычислим аппроксимацию ожидания, которая представляет собой среднее расхождение KL параметрических стратегий по n выборкам состояний:

$$\mathbb{E}_s\left[D_{\mathrm{KL}}(\pi_{\boldsymbol{\theta}}(\cdot|s)\|\pi_{\boldsymbol{\theta}'}(\cdot|s)\right] \approx \frac{1}{n}\sum_{i=1}^{n} D_{\mathrm{KL}}(\pi_{\boldsymbol{\theta}}(\cdot|s^{(i)})\|\pi_{\boldsymbol{\theta}'}(\cdot|s^{(i)}) \\ \approx \frac{1}{5}(0.456+0.283+0.456+0.456+0.283) \\ \approx 0.285.$$

13 Методы «актор–критик»

В предыдущей главе обсуждались способы улучшения параметрической стратегии с помощью информации о градиенте, полученной из развертываний. В этой главе представлены *методы «актор–критик»* (actor-critic method), которые для выбора направления оптимизации используют значение функции полезности. В данном контексте актором является стратегия, а критиком – функция полезности. Оба они обучаются параллельно. Мы обсудим несколько методов, которые различаются тем, какую часть аппроксимируют: функцию полезности, функцию преимущества или функцию полезности действия. Большинство методов нацелены на стохастические стратегии, но мы также обсудим один метод, который работает с детерминированными стратегиями, определяющими непрерывные действия. Наконец, мы обсудим способ выбора действий в текущем времени для создания более информативных траекторий обучения актора и критика.

13.1. Определение актора и критика

В методах актора–критика у нас есть *актор*, представленный стратегией π_θ с набором параметров $\boldsymbol{\theta}$, который опирается на помощь *критика*, предоставляющего значение функции полезности $U_{\boldsymbol{\varphi}}(s)$, $Q_{\boldsymbol{\varphi}}(s, a)$ или $A_{\boldsymbol{\varphi}}(s, a)$ с параметрами $\boldsymbol{\varphi}$. Мы начнем эту главу с простого метода «актор–критик», в котором оптимизация стратегии π_θ выполняется посредством градиентного подъема, при этом градиент нашей цели такой же, как в уравнении (11.44):

$$\nabla U(\boldsymbol{\theta}) = \mathbb{E}_\tau\left[\sum_{k=1}^{d} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a^{(k)}|s^{(k)})\gamma^{k-1} A_{\boldsymbol{\theta}}(s^{(k)}, a^{(k)})\right]. \tag{13.1}$$

Преимущество следования стратегии с параметрами $\boldsymbol{\theta}$ можно оценить, используя набор наблюдаемых переходов от s к s' с вознаграждением r:

$$A_{\boldsymbol{\theta}}(s,a) = \mathbb{E}_{r,s'}\left[r + \gamma U^{\pi_{\boldsymbol{\theta}}}(s') - U^{\pi_{\boldsymbol{\theta}}}(s)\right]. \tag{13.2}$$

Выражение $r + \gamma U^{\pi_{\boldsymbol{\theta}}}(s') - U^{\pi_{\boldsymbol{\theta}}}(s)$ называется *времяразностным остатком* (temporal difference residual).

Критик позволяет нам найти истинную функцию полезности $U^{\pi_{\boldsymbol{\theta}}}$ при следовании стратегии $\pi_{\boldsymbol{\theta}}$, что приводит к следующему градиенту для актора:

$$\nabla U(\boldsymbol{\theta}) \approx \mathbb{E}_{\tau}\left[\sum_{k=1}^{d} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a^{(k)}|s^{(k)}) \gamma^{k-1}\left(r^{(k)} + \gamma U_{\boldsymbol{\varphi}}(s^{(k+1)}) - U_{\boldsymbol{\varphi}}(s^{(k)})\right)\right]. \tag{13.3}$$

Это ожидание можно найти с помощью траекторий развертывания, как это сделано в главе 11.

Критик тоже обновляется путем градиентной оптимизации. Нам необходимо найти набор $\boldsymbol{\varphi}$, который минимизирует функцию потерь:

$$\ell(\boldsymbol{\varphi}) = \frac{1}{2}\mathbb{E}_{s}\left[\left(U_{\boldsymbol{\varphi}}(s) - U^{\pi_{\boldsymbol{\theta}}}(s)\right)^2\right]. \tag{13.4}$$

Чтобы свести к минимуму значение этой функции, предпримем шаги в направлении, противоположном градиенту:

$$\nabla \ell(\boldsymbol{\varphi}) = \mathbb{E}_{s}\left[\left(U_{\boldsymbol{\varphi}}(s) - U^{\pi_{\boldsymbol{\theta}}}(s)\right)\nabla_{\boldsymbol{\varphi}} U_{\boldsymbol{\varphi}}(s)\right]. \tag{13.5}$$

Конечно, мы не знаем точное значение $U^{\pi_{\boldsymbol{\theta}}}$, но его можно найти, используя метод предстоящего вознаграждения, в результате чего получаем

$$\nabla \ell(\boldsymbol{\varphi}) = \mathbb{E}_{\tau}\left[\sum_{k=1}^{d}\left(U_{\boldsymbol{\varphi}}(s^{(k)}) - r_{\text{to-go}}^{(k)}\right)\nabla_{\boldsymbol{\varphi}} U_{\boldsymbol{\varphi}}(s^{(k)})\right], \tag{13.6}$$

где $r_{\text{to-go}}^{(k)}$ – предстоящее вознаграждение на шаге k на конкретной траектории τ.

Алгоритм 13.1 показывает, как найти $\nabla U(\boldsymbol{\theta})$ и $\nabla \ell(\boldsymbol{\varphi})$ из развертываний. На каждой итерации мы делаем шаг по $\boldsymbol{\theta}$ в направлении $\nabla U(\boldsymbol{\theta})$, чтобы максимизировать полезность, и делаем шаг $\boldsymbol{\varphi}$ в направлении, противоположном $\nabla \ell(\boldsymbol{\varphi})$, чтобы минимизировать потери. Этот подход иногда бывает нестабильным из-за зависимости между оценками по $\boldsymbol{\theta}$ и $\boldsymbol{\varphi}$, но хорошо работает для многих задач. Общепринятой практикой повышения стабильности является более частое обновление стратегии по сравнению с функцией полезности. Алгоритмы в этой главе можно легко адаптировать для извлечения обновлений функции полезности только из небольшого подмножества итераций, в которых обновляется стратегия.

Алгоритм 13.1. Базовый метод «актор–критик» для вычисления как градиента стратегии, так и градиента функции полезности в задаче MDP $\mathcal{P}$ с начальным распределением состояний `b`. Стратегия π параметризована набором θ и имеет логарифмический градиент $\nabla\log\pi$. Функция полезности `U` параметризована ϕ, а градиент ее целевой функции равен ∇U. Этот метод выполняет `m` развертываний на глубину `d`. Результаты используются для обновления θ и φ. Набор параметров стратегии обновляется в направлении $\nabla\theta$, чтобы максимизировать ожидаемую полезность, тогда как параметры функции полезности обновляются в отрицательном направлении $\nabla\varphi$, чтобы минимизировать потерю полезности

```
struct ActorCritic
    𝒫     # задача
    b     # начальное распределение по состояниям
    d     # глубина
    m     # количество выборок
    ∇logπ # градиент логарифмического правдоподобия ∇logπ(θ,a,s)
    U     # параметрическая функция полезности U(φ, s)
    ∇U    # градиент функции полезности ∇U(φ,s)
end

function gradient(M::ActorCritic, π, θ, φ)
    𝒫, b, d, m, ∇logπ = M.𝒫, M.b, M.d, M.m, M.∇logπ
    U, ∇U, γ = M.U, M.∇U, M.𝒫.γ
    πθ(s) = π(θ, s)
    R(τ,j) = sum(r*γ^(k-1) for (k,(s,a,r)) in enumerate(τ[j:end]))
    A(τ,j) = τ[j][3] + γ*U(φ,τ[j+1][1]) - U(φ,τ[j][1])
    ∇Uθ(τ) = sum(∇logπ(θ,a,s)*A(τ,j)*γ^(j-1) for (j, (s,a,r))
                 in enumerate(τ[1:end-1]))
    ∇ℓφ(τ) = sum((U(φ,s) - R(τ,j))*∇U(φ,s) for (j, (s,a,r))
                 in enumerate(τ))
    trajs = [simulate(𝒫, rand(b), πθ, d) for i in 1:m]
    return mean(∇Uθ(τ) for τ in trajs), mean(∇ℓφ(τ) for τ in trajs)
end
```

13.2. Обобщенная оценка преимуществ

Обобщенная оценка преимуществ (generalized advantage estimation), реализованная в алгоритме 13.2, – это метод типа «актор–критик», который использует более общую версию оценки преимуществ, показанную в уравнении (13.2), что позволяет нам находить баланс между систематической ошибкой и дисперсией[1]. Аппроксимация с времяразностным остатком имеет низкую дисперсию,

[1] J. Schulman, P. Moritz, S. Levine, M. Jordan, P. Abbeel, *High-Dimensional Continuous Control Using Generalized Advantage Estimation*, in International Conference on Learning Representations (ICLR), 2016. arXiv: 1506.02438v6.

но она вносит систематическую ошибку из-за потенциальной неточности U_{φ}, используемой для аппроксимации U^{π_θ}. Альтернативой является замена $r + \gamma U^{\pi_\theta}(s')$ последовательностью вознаграждений за развертывание $r_1, \ldots, r_d$:

$$A_{\boldsymbol{\theta}}(s,a) = \mathbb{E}_{r_1,\ldots,r_d}\left[r_1 + \gamma r_2 + \gamma^2 r_3 + \cdots + \gamma^{d-1} r_d - U^{\pi_\theta}(s)\right] \tag{13.7}$$

$$= \mathbb{E}_{r_1,\ldots,r_d}\left[-U^{\pi_\theta}(s) + \sum_{\ell=1}^{d} \gamma^{\ell-1} r_\ell\right]. \tag{13.8}$$

Мы можем получить объективное значение этого ожидания с помощью траекторий развертывания, как это делается в методах нахождения градиента стратегии (глава 11). Однако такой способ дает высокую дисперсию, а это означает, что нам нужно много выборок, чтобы получить точную оценку.

Подход, используемый при обобщенной оценке преимуществ, заключается в том, чтобы балансировать между двумя крайностями в виде использования времяразностных остатков и полного развертывания. Обозначим через $\hat{A}^{(k)}$ оценку преимущества, полученную из k шагов развертывания и полезности, связанной с результирующим состоянием s':

$$\hat{A}^{(k)}(s,a) = \mathbb{E}_{r_1,\ldots,r_k,s'}\left[r_1 + \gamma r_2 + \cdots + \gamma^{k-1} r_k + \gamma^k U^{\pi_\theta}(s') - U^{\pi_\theta}(s)\right] \tag{13.9}$$

$$= \mathbb{E}_{r_1,\ldots,r_k,s'}\left[-U^{\pi_\theta}(s) + \gamma^k U^{\pi_\theta}(s') + \sum_{\ell=1}^{k} \gamma^{\ell-1} r_\ell\right]. \tag{13.10}$$

В качестве альтернативы можно записать $\hat{A}^{(k)}$ как ожидание по времяразностным остаткам. Мы можем определить

$$\delta_t = r_t + \gamma U(s_{t+1}) - U(s_t), \tag{13.11}$$

где s_t, r_t и s_{t+1} – это состояние, вознаграждение и последующее состояние на выбранной траектории, а U – наша оценка функции полезности. Отсюда

$$\hat{A}^{(k)}(s,a) = \mathbb{E}\left[\sum_{\ell=1}^{k} \gamma^{\ell-1} \delta_\ell\right]. \tag{13.12}$$

Вместо фиксированного конкретного значения k метод обобщенной оценки преимущества вводит параметр $\lambda \in [0, 1]$, который обеспечивает *экспоненциально взвешенное среднее* (exponentially weighted average) $\hat{A}^{(k)}$ для k в диапазоне от 1 до d[2]:

$$\hat{A}^{\mathrm{GAE}}(s,a)\big|_{d=1} = \hat{A}^{(1)}; \tag{13.13}$$

---

[2] Экспоненциально взвешенное среднее последовательности $x_1, x_2, \ldots$ имеет вид $(1 - \lambda)(x_1 + \lambda x_2 + \lambda 2 x_3 + \cdots)$.

$$\hat{A}^{\text{GAE}}(s,a)\big|_{d=2} = (1-\lambda)\hat{A}^{(1)} + \lambda\hat{A}^{(2)}; \tag{13.14}$$

$$\hat{A}^{\text{GAE}}(s,a)\big|_{d=3} = (1-\lambda)\hat{A}^{(1)} + \lambda\left((1-\lambda)\hat{A}^{(2)} + \lambda\hat{A}^{(3)}\right) \tag{13.15}$$

$$= (1-\lambda)\hat{A}^{(1)} + \lambda(1-\lambda)\hat{A}^{(2)} + \lambda^2\hat{A}^{(3)}; \tag{13.16}$$

$$\hat{A}^{\text{GAE}}(s,a) = (1-\lambda)\left(\hat{A}^{(1)} + \lambda\hat{A}^{(2)} + \lambda^2\hat{A}^{(3)} + \cdots + \lambda^{d-2}\hat{A}^{(d-1)}\right) + \lambda^{d-1}\hat{A}^{(d)}. \tag{13.17}$$

Для бесконечного горизонта обобщенная оценка преимущества упрощается до

$$\hat{A}^{\text{GAE}}(s,a) = (1-\lambda)\left(\hat{A}^{(1)} + \lambda\hat{A}^{(2)} + \lambda^2\hat{A}^{(3)} + \cdots\right) \tag{13.18}$$

$$= (1-\lambda)\left(\delta_1(1+\lambda+\lambda^2+\cdots) + \gamma\delta_2(\lambda+\lambda^2+\cdots) + \gamma^2\delta_3(\lambda^2+\cdots) + \cdots\right) \tag{13.19}$$

$$= (1-\lambda)\left(\delta_1\frac{1}{1-\lambda} + \gamma\delta_2\frac{\lambda}{1-\lambda} + \gamma^2\delta_3\frac{\lambda^2}{1-\lambda} + \cdots\right) \tag{13.20}$$

$$= \mathbb{E}\left[\sum_{k=1}^{\infty}(\gamma\lambda)^{k-1}\delta_k\right]. \tag{13.21}$$

Мы можем подобрать параметр λ таким образом, чтобы найти компромисс между систематической ошибкой и дисперсией. Если $\lambda = 0$, то получается оценка с высокой систематической ошибкой и низкой дисперсией по методу временной разностного остатка из предыдущего раздела. Если $\lambda = 1$, мы имеем несмещенную оценку по методу полного развертывания, но с повышенной дисперсией. На рис. 13.1 показан результат работы алгоритма с различными значениями λ.

Алгоритм 13.2. Обобщенная оценка преимущества для вычисления как градиента стратегии, так и градиента функции полезности для задачи MDP $\mathcal{P}$ с начальным распределением состояний `b`. Стратегия параметризована набором `θ` и имеет логарифмический градиент `∇logπ`. Функция полезности `U` параметризована набором `φ` и имеет градиент `∇U`. Этот метод выполняет `m` развертываний на глубину `d`. Обобщенное преимущество вычисляется по уравнению (13.21) с экспоненциальным взвешиванием λ с конечным горизонтом. Реализация здесь представляет собой упрощенную версию алгоритма, представленного в исходной статье и включающего аспекты выбора доверительных областей при выполнении шагов

```
struct GeneralizedAdvantageEstimation
    𝒫     # задача
    b     # начальное распределение по состояниям
    d     # глубина
    m     # количество выборок
    ∇logn # градиент логарифмического правдоподобия ∇logn(θ,a,s)
```

```
    U     # параметрическая функция полезности U(φ, s)
    ∇U    # градиент функции полезности ∇U(φ,s)
    λ     # вес ∈ [0,1]
end

function gradient(M::GeneralizedAdvantageEstimation, π, θ, φ)
    𝒫, b, d, m, ∇logπ = M.𝒫, M.b, M.d, M.m, M.∇logπ
    U, ∇U, γ, λ = M.U, M.∇U, M.𝒫.γ, M.λ
    πθ(s) = π(θ, s)
    R(τ,j) = sum(r*γ^(k-1) for (k,(s,a,r)) in enumerate(τ[j:end]))
    δ(τ,j) = τ[j][3] + γ*U(φ,τ[j+1][1]) - U(φ,τ[j][1])
    A(τ,j) = sum((γ*λ)^(ℓ-1)*δ(τ, j+ℓ-1) for ℓ in 1:d-j)
    ∇Uθ(τ) = sum(∇logπ(θ,a,s)*A(τ,j)*γ^(j-1)
    for (j, (s,a,r)) in enumerate(τ[1:end-1]))
    ∇ℓφ(τ) = sum((U(φ,s) - R(τ,j))*∇U(φ,s)
    for (j, (s,a,r)) in enumerate(τ))
    trajs = [simulate(𝒫, rand(b), πθ, d) for i in 1:m]
    return mean(∇Uθ(τ) for τ in trajs), mean(∇ℓφ(τ) for τ in trajs)
end
```

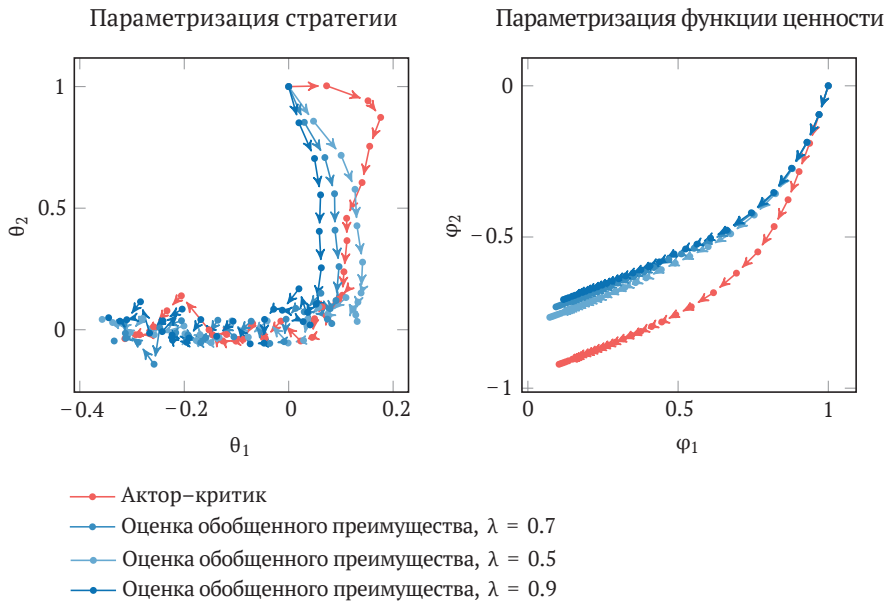


Рис. 13.1. Сравнение базовой оценки «актор–критик» с обобщенной оценкой преимущества в задаче простого регулятора с $\gamma = 0.9$, гауссовой стратегией $\pi_\theta(s) = \mathcal{N}(\theta_1, s, \theta_2^2)$ и приближенной функцией полезности $U_\varphi(s) = \varphi_1 s + \varphi_2 s^2$. Мы обнаружили, что обобщенная оценка преимущества лучше подходит для эффективной параметризации стратегии и функции полезности. (Напомним, что оптимальная параметризация стратегии равна $[-1, 0]$, а оптимальная параметризация функции полезности близка к $[0, -0.7]$)

13.3. Градиент детерминированной стратегии

Подход, основанный на *градиенте детерминированной стратегии*[3], предусматривает оптимизацию детерминированной стратегии $\pi_{\boldsymbol{\theta}}(s)$, которая производит непрерывные действия с поддержкой со стороны критика в виде параметрической функции полезности действия $Q_{\boldsymbol{\varphi}}(s, a)$. Как и в обсуждавшихся до сих пор методах типа «актор–критик», мы определяем функцию потерь относительно параметризации $\boldsymbol{\varphi}$:

$$\ell(\boldsymbol{\varphi}) = \frac{1}{2} \mathop{\mathbb{E}}_{s,a,r,s'} \left[\left(r + \gamma Q_{\boldsymbol{\varphi}}(s', \pi_{\boldsymbol{\theta}}(s')) - Q_{\boldsymbol{\varphi}}(s,a) \right)^2 \right], \tag{13.22}$$

где ожидание основано на кортежах опыта, сгенерированных развертываниями $\pi_{\boldsymbol{\theta}}$. Эта функция потерь пытается минимизировать остаток $Q_{\boldsymbol{\varphi}}$, подобно тому, как метод «актор–критик» в разделе 13.1 пытался минимизировать остаток $U_{\boldsymbol{\varphi}}$.

Как и в других методах, мы обновляем $\boldsymbol{\varphi}$, делая шаг в направлении, противоположном градиенту:

$$\nabla \ell(\boldsymbol{\varphi}) = \mathop{\mathbb{E}}_{s,a,r,s'} \left[\left(r + \gamma Q_{\boldsymbol{\varphi}}(s', \pi_{\boldsymbol{\theta}}(s')) - Q_{\boldsymbol{\varphi}}(s,a) \right) \left(\gamma \nabla_{\boldsymbol{\varphi}} Q_{\boldsymbol{\varphi}}(s', \pi_{\boldsymbol{\theta}}(s')) - \nabla_{\boldsymbol{\varphi}} Q_{\boldsymbol{\varphi}}(s,a) \right) \right]. \tag{13.23}$$

Следовательно, нам нужна дифференцируемая параметрическая функция полезности действия, из которой мы можем вычислить $\nabla_{\boldsymbol{\varphi}} Q_{\boldsymbol{\varphi}}(s, a)$. В роли такой функции может выступать нейронная сеть.

Для актора мы стремимся найти значение $\boldsymbol{\theta}$, которое максимизирует значение выражения

$$U(\boldsymbol{\theta}) = \mathop{\mathbb{E}}_{s \sim b_{\gamma,\boldsymbol{\theta}}} \left[Q_{\boldsymbol{\varphi}}(s, \pi_{\boldsymbol{\theta}}(s)) \right], \tag{13.24}$$

где ожидание берется по состояниям при следовании стратегии $\pi_{\boldsymbol{\theta}}$. Мы снова можем использовать градиентный подъем для оптимизации $\boldsymbol{\theta}$ с градиентом, заданным уравнением

$$\nabla U(\boldsymbol{\theta}) = \mathbb{E}_S \left[\nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\varphi}}(s, \pi_{\boldsymbol{\theta}}(s)) \right] \tag{13.25}$$

$$= \mathbb{E}_S \left[\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(s) \nabla_a Q_{\boldsymbol{\varphi}}(s,a) \big|_{a = \pi_{\boldsymbol{\theta}}(s)} \right]. \tag{13.26}$$

Здесь $\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(s)$ – матрица Якоби, i-й столбец которой представляет собой градиент относительно i-го измерения действия стратегии при параметризации $\boldsymbol{\theta}$

[3] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, *Deterministic Policy Gradient Algorithms*, in International Conference on Machine Learning (ICML), 2014.

(пример 13.1). Градиент $\nabla_a Q_{\boldsymbol{\varphi}}(s,a)\big|_{a=\pi_{\boldsymbol{\theta}}(s)}$ – это вектор, показывающий, насколько изменяется ожидаемая полезность действия, по мере того как мы возмущаем действие, заданное стратегией в состоянии s. Следовательно, чтобы получить градиент детерминированной стратегии, в дополнение к якобиану мы должны знать градиент $\nabla_a Q_{\boldsymbol{\varphi}}(s,a)\big|_{a=\pi_{\boldsymbol{\theta}}(s)}$.

Пример 13.1. Пример использования якобиана при нахождении градиента детерминированной стратегии

Рассмотрим следующую детерминированную стратегию для двумерного пространства действий и одномерного пространства состояний:

$$\pi_{\boldsymbol{\theta}}(s) = \begin{bmatrix} \theta_1 + \theta_2 s + \theta_3 s^2 \\ \theta_1 + \sin(\theta_4 s) + \cos(\theta_5 s) \end{bmatrix}.$$

Тогда матрица $\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(s)$ принимает следующий вид:

$$\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(s) = \begin{bmatrix} \nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(s)\big|_{a_1} & \nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(s)\big|_{a_2} \end{bmatrix} \begin{bmatrix} 1 & 1 \\ s & 0 \\ s^2 & 0 \\ 0 & \cos(\theta_4 s)s \\ 0 & -\sin(\theta_5 s)s \end{bmatrix}.$$

Как и в других методах «актор–критик», мы выполняем градиентный спуск по $\ell(\boldsymbol{\varphi})$ и градиентный подъем по $U(\boldsymbol{\theta})$. Чтобы этот метод работал на практике, необходимо использовать несколько дополнительных приемов. Один из них – извлекать опыт из стохастической стратегии, чтобы улучшить охват исследования. Как правило, достаточно добавить гауссов шум с нулевым средним к действиям, генерируемым нашей детерминированной стратегией $\pi_{\boldsymbol{\theta}}$, как это делается в алгоритме 13.3. Чтобы обеспечить стабильность модели при изучении $\boldsymbol{\theta}$ и $\boldsymbol{\varphi}$, можно использовать *воспроизведение опыта* (experience replay)[4].

Использование этого метода и влияние параметра σ продемонстрированы в примере 13.2.

[4] Мы обсудим воспроизведение опыта в разделе 17.7 в контексте обучения с подкреплением. Другие методы стабилизации обучения включают использование целевых параметризаций, описанных в контексте нейронных репрезентаций в работе T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, D. Wierstra, *Continuous Control with Deep Reinforcement Learning*, in International Conference on Learning Representations (ICLR), 2016. arXiv: 1509.02971v6.

Алгоритм 13.3. Реализация метода градиента детерминированной стратегии для вычисления градиента $\nabla\theta$ детерминированной стратегии π и градиента функции полезности $\nabla\phi$ для задачи MDP $\mathcal{P}$ с непрерывным действием и начальным распределением состояний `b`. Стратегия параметризована набором θ и имеет градиент $\nabla\pi$, который дает матрицу, где каждый столбец является градиентом по отношению к компоненту непрерывного действия. Функция полезности `Q` параметризуется набором ϕ и имеет градиент ∇Q_ϕ относительно параметризации и градиент ∇Q_a относительно действия. Этот метод запускает `m` развертываний на глубину `d` и выполняет исследование с использованием гауссова шума с нулевым средним значением и стандартным отклонением σ

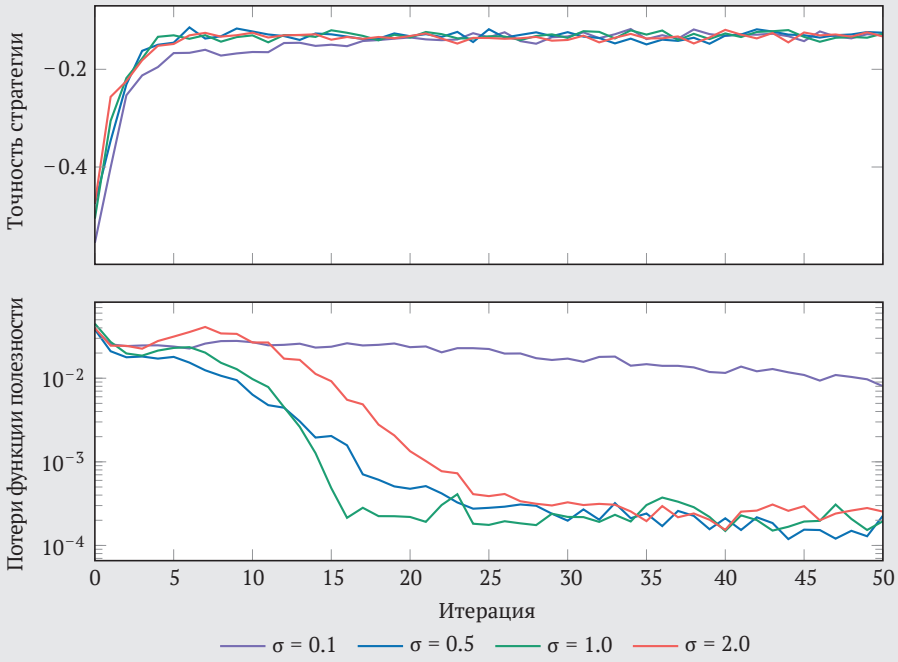
```
struct DeterministicPolicyGradient
    𝒫  # задача
    b  # начальное распределение по состояниям
    d  # глубина
    m  # количество выборок
    ∇π  # градиент детерминированной стратегии π(φ, s)
    Q  # параметрическая функция полезности Q(φ,s,a)
    ∇Qφ # градиент функции полезности по отношению к φ
    ∇Qa # градиент функции полезности по отношению к a
    σ  # шум стратегии
end

function gradient(M::DeterministicPolicyGradient, π, θ, φ)
    𝒫, b, d, m, ∇π = M.𝒫, M.b, M.d, M.m, M.∇π
    Q, ∇Qφ, ∇Qa, σ, γ = M.Q, M.∇Qφ, M.∇Qa, M.σ, M.𝒫.γ
    π_rand(s) = π(θ, s) + σ*randn()*I
    ∇Uθ(τ) = sum(∇π(θ,s)*∇Qa(φ,s,π(θ,s))*γ^(j-1) for (j,(s,a,r))
                 in enumerate(τ))
    ∇ℓφ(τ,j) = begin
        s, a, r = τ[j]
        s′ = τ[j+1][1]
        a′ = π(θ,s′)
        δ = r + γ*Q(φ,s′,a′) - Q(φ,s,a)
        return δ*(γ*∇Qφ(φ,s′,a′) - ∇Qφ(φ,s,a))
    end
    ∇ℓφ(τ) = sum(∇ℓφ(τ,j) for j in 1:length(τ)-1)
    trajs = [simulate(𝒫, rand(b), π_rand, d) for i in 1:m]
    return mean(∇Uθ(τ) for τ in trajs), mean(∇ℓφ(τ) for τ in trajs)
end
```

Пример 13.2. Применение метода градиента детерминированной стратегии к задаче простого регулятора и исследование влияния параметра стохастичности стратегии σ

Рассмотрим применение алгоритма градиента детерминированной стратегии к задаче простого регулятора. Предположим, мы используем простую параметрическую детерминированную стратегию $\pi_\theta(s) = \theta_1$ и параметрическую функцию полезности состояния-действия:

$Q_{\varphi}(s, a) = \varphi_1 + \varphi_2 s + \varphi_3 s^2 + \varphi_4 (s + a)^2$.

На рисунке ниже показаны результаты работы алгоритма, начиная с $\boldsymbol{\theta} = [0]$ и $\boldsymbol{\varphi} = [0, 1, 0, -1]$ для различных значений σ. Каждая итерация выполнялась с пятью развертываниями до глубины 10 с $\gamma = 0.9$.



Для этой простой задачи стратегия быстро сходится к оптимальности почти независимо от σ. Однако если значение σ слишком мало или слишком велико, для улучшения функции полезности требуется больше времени. В случае очень малых значений σ стратегия выполняет недостаточно обширное исследование пространства для эффективного обучения функции полезности. При больших значениях σ алгоритм исследует более обширное пространство, но склонен чаще делать неправильный выбор следующего действия.

13.4. Метод «актор–критик» с поиском по дереву Монте-Карло

Мы можем развить идею онлайн-планирования (глава 9) до подхода «актор–критик», в котором мы улучшаем параметрическую стратегию $\pi_{\theta}(a|s)$ и пара-

метрическую функцию полезности[5]. В этом разделе обсуждается применение поиска по дереву Монте-Карло (раздел 9.6) для обучения стохастической стратегии в дискретном пространстве действий. Мы применим параметрическую стратегию и функцию полезности для поиска по дереву Монте-Карло, а затем воспользуемся результатами этого поиска для уточнения параметров стратегии и функции полезности. Как и в случае с другими методами, относящимися к типу «актор–критик», мы применяем градиентную оптимизацию $\boldsymbol{\theta}$ и $\boldsymbol{\varphi}$[6].

Выполняя поиск по дереву Монте-Карло, мы должны нацелить наше исследование в определенном направлении с помощью параметрической стратегии $\pi_{\boldsymbol{\theta}}(a|s)$. Один из подходов заключается в использовании действия, которое максимизирует *вероятностную верхнюю доверительную границу*:

$$a = \arg\max_a Q(s,a) + c\pi_{\boldsymbol{\theta}}(a|s)\frac{\sqrt{N(s)}}{1+N(s,a)}, \tag{13.27}$$

где $Q(s, a)$ – полезность действия, найденного посредством поиска по дереву, $N(s, a)$ – количество посещений узла, которое обсуждалось в разделе 9.6, и $N(s) = \sum_a N(s, a)$[7].

Выполнив поиск по дереву, мы можем использовать собранную статистику для получения $\pi_{\text{MCTS}}(a|s)$. Одним из способов является использование подсчетов[8]:

$$\pi_{\text{MCTS}}(a|s) \propto N(s, a)^{\eta}, \tag{13.28}$$

где $\eta \geqslant 0$ – гиперпараметр, определяющий жадность стратегии. Если $\eta = 0$, то π_{MCTS} будет генерировать действия случайным образом. При $\eta \to \infty$ метод предпочтет действие, которое было наиболее популярно для этого состояния.

Для оптимизации $\boldsymbol{\theta}$ нам нужно, чтобы модель $\pi_{\boldsymbol{\theta}}$ соответствовала результатам, полученным с помощью поиска по дереву Монте-Карло. Одна из функций

---

[5] Градиент детерминированной стратегии использует $Q_{\boldsymbol{\varphi}}$, но в этом подходе применяется $U_{\boldsymbol{\varphi}}$, как и в других методах «актор–критик», обсуждаемых в этой главе.

[6] Этот общий подход был предложен в работе D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al., *Mastering the Game of Go Without Human Knowledge*, Nature, vol. 550, pp. 354–359, 2017. Обсуждение здесь в общих чертах соответствует их алгоритму AlphaGo Zero, но, вместо того чтобы пытаться решить задачу игры в го, мы пытаемся решить общую задачу MDP. Поскольку Alpha Zero играет за обоих игроков в го и, как правило, в этой игре есть победитель и проигравший, это позволяет исходному методу подкреплять выигрышное поведение и наказывать за проигрышное поведение. Обобщенный подход к MDP будет склонен страдать от скудного вознаграждения при применении к аналогичным задачам.

[7] Имеются некоторые заметные отличия от верхней доверительной границы, представленной в уравнении (9.1); например, в уравнении (13.27) нет логарифма, и мы добавляем 1 к знаменателю, чтобы соответствовать представлению, используемому AlphaGo Zero.

[8] В алгоритме 9.5 мы выбираем жадное действие по отношению к Q. C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, S. Colton, *A Survey of Monte Carlo Tree Search Methods*, IEEE Transactions on Computational Intelligence and AI in Games, vol. 4, no. 1, pp. 1–43, 2012. Предлагаемый здесь подход соответствует алгоритму AlphaGo Zero.

потерь, которую мы можем определить, – это ожидаемая перекрестная энтропия $\pi_{\boldsymbol{\theta}}(\cdot|s)$ относительно $\pi_{\text{MCTS}}(\cdot|s)$:

$$\ell(\boldsymbol{\theta}) = -\mathbb{E}_s\left[\sum_a \pi_{\text{MCTS}}(a|s)\log\pi_{\boldsymbol{\theta}}(a|s)\right], \tag{13.29}$$

где ожидание вычисляется для состояний, испытанных во время изучения дерева. Градиент выглядит следующим образом:

$$\nabla\ell(\boldsymbol{\theta}) = -\mathbb{E}_s\left[\sum_a \frac{\pi_{\text{MCTS}}(a|s)}{\pi_{\boldsymbol{\theta}}(a|s)}\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(a|s)\right]. \tag{13.30}$$

Чтобы обучить $\boldsymbol{\varphi}$, мы определяем функцию потерь в отношении функции полезности, сгенерированной во время исследования дерева

$$U_{MCTS}(s) = \max_a Q(s,a), \tag{13.31}$$

которая определена, по крайней мере, в состояниях, которые мы исследуем во время поиска по дереву. Функция потерь стремится сделать так, чтобы $U_{\boldsymbol{\varphi}}$ максимально совпадала с оценками поиска по дереву:

$$\ell(\boldsymbol{\varphi}) = \frac{1}{2}\mathbb{E}_s\left[(U_{\boldsymbol{\varphi}}(s) - U_{\text{MCTS}}(s))^2\right]. \tag{13.32}$$

Наконец, получаем градиент:

$$\nabla\ell(\boldsymbol{\varphi}) = \mathbb{E}_s\left[(U_{\boldsymbol{\varphi}}(s) - U_{\text{MCTS}}(s))\nabla_{\boldsymbol{\varphi}}U_{\boldsymbol{\varphi}}(s)\right]. \tag{13.33}$$

Как и в случае с методом «актор–критик» в первом разделе, нам нужно уметь вычислять градиент параметрической функции полезности.

Выполнив некоторое количество проходов поиска по дереву Монте-Карло, мы обновляем θ, делая шаги в направлении, противоположном $\nabla\ell(\boldsymbol{\theta})$, и $\boldsymbol{\varphi}$, двигаясь в направлении, противоположном $\nabla\ell(\boldsymbol{\varphi})$[9].

13.5. Заключение

- В методах «актор–критик» актор пытается оптимизировать набор параметров стратегии с помощью критика, который предоставляет оценку параметров функции полезности.

[9] Реализация AlphaGo Zero использует единую нейронную сеть для представления как функции полезности, так и стратегии вместо независимых параметризаций, как обсуждалось в этом разделе. Градиент, используемый для обновления параметров сети, представляет собой смесь уравнений (13.30) и (13.33). Это усовершенствование значительно сокращает время оценки и время изучения признаков.

- Как правило, в методах «актор–критик» оптимизация на основе градиента используется для изучения параметров как стратегии, так и приближенной функции полезности.
- Базовый метод «актор–критик» использует градиент стратегии для актора и минимизирует квадратичную временную разность для критика.
- Метод обобщенного преимущества пытается уменьшить дисперсию градиента стратегии ценой увеличения систематической ошибки за счет накопления остатков временной разности на нескольких временных шагах.
- Градиент детерминированной стратегии может быть применен к задачам с непрерывным пространством действий и использует актора детерминированной стратегии и критика полезности действия.
- Методы текущего времени, такие как поиск по дереву Монте-Карло, могут использоваться для управления оптимизацией стратегии и оценки функции полезности.

13.6. Упражнения

Упражнение 13.1. Будет ли метод «актор–критик» с поиском по дереву Монте-Карло, представленный в разделе 13.4, удачным выбором для решения задачи о перевернутом маятнике (приложение F.3)?

Решение. Поиск по дереву Монте-Карло расширяет дерево на основе посещенных состояний. Задача перевернутого маятника имеет непрерывное пространство состояний, что приводит к дереву поиска с бесконечным коэффициентом ветвления. Использование этого алгоритма потребует корректировки условия задачи, например дискретизации пространства состояний.

Упражнение 13.2. Ниже даны уравнения функций преимуществ. Определите, какие из них верны, и объясните, к чему они относятся:

a) $\mathbb{E}_{r,s'}\left[r + \gamma U^{\pi_\theta}(s) - U^{\pi_\theta}(s')\right]$;

b) $\mathbb{E}_{r,s'}\left[r + \gamma U^{\pi_\theta}(s') - U^{\pi_\theta}(s)\right]$;

c) $\mathbb{E}_{r_{1:d},s'}\left[-U^{\pi_\theta}(s) + \gamma^k U^{\pi_\theta}(s') + \sum_{\ell=1}^{k} \gamma^{l-1} r_l\right]$;

d) $\mathbb{E}_{r_{1:d},s'}\left[-U^{\pi_\theta}(s) + \gamma U^{\pi_\theta}(s') + \sum_{\ell=1}^{k} \gamma^{l-1} r_l\right]$;

e) $\mathbb{E}\left[-U^{\pi_\theta}(s) + \sum_{\ell=1}^{d} \gamma^{l-1} r_l\right]$;

f) $\mathbb{E}\left[-\gamma U^{\pi_\theta}(s') + \sum_{\ell=1}^{d+1} \gamma^{l-1} r_l\right]$;

g) $\mathbb{E}\left[\sum_{\ell=1}^{k} \gamma^{l-1}\delta_{l-1}\right]$;

h) $\mathbb{E}\left[\sum_{\ell=1}^{k} \gamma^{l-1}\delta_{l}\right]$;

i) $\mathbb{E}\left[\sum_{k=1}^{\infty} (\gamma\lambda)^{k-1}\delta_{k}\right]$;

j) $\mathbb{E}\left[\sum_{k=1}^{\infty} (\lambda)^{k-1}\delta_{k}\right]$.

Решение. Ниже перечислены корректные выражения и методы, к которым они относятся:

b) преимущество с остатком временной разности;
c) оценка преимущества после k-шаговых развертываний;
e) преимущество с последовательностью вознаграждений за развертывание;
h) оценка преимущества с остатками временной разности;
i) обобщенная оценка преимущества.

Упражнение 13.3. Каковы преимущества использования остатка временной разности по сравнению с последовательностью вознаграждений за развертывание, и наоборот?

Решение. Аппроксимация с использованием остатка временной разности более эффективна с вычислительной точки зрения, чем использование последовательности развертываний. Такая аппроксимация имеет низкую дисперсию, но высокую систематическую ошибку из-за использования критиком функции полезности U_{φ} в качестве аппроксиматора функции истинного значения U^{π_θ}. С другой стороны, аппроксимация через развертывания имеет высокую дисперсию, но является несмещенной. Получение точной оценки с использованием аппроксимации остаточной временной разности обычно требует гораздо меньшего количества выборок, чем при использовании аппроксимации развертываниями, но ценой появления систематической ошибки в оценке.

Упражнение 13.4. Рассмотрим функцию полезности действия $Q_{\varphi}(s, a) = \varphi_1 + \varphi_2 s + \varphi_3 s^2 + \varphi_4(s + a)^2$, приведенную в примере 13.2. Вычислите градиенты, необходимые для реализации подхода градиента детерминированной стратегии.

Решение. Нам нужно рассчитать два градиента. Для актора нам нужно вычислить $\nabla_{\varphi} Q(s, a)$, а для критика – $\nabla_a Q(s, a)$:

$\nabla_{\varphi} Q(s, a) = [1, s, s^2, (s + a)^2]$;

$\nabla_a Q(s, a) = 2\varphi_4(s + a)$.

14 Проверка стратегии

Методы, представленные в предыдущих главах, показывают, как найти оптимальное или приблизительно оптимальное решение, используя модели динамики и вознаграждения. Однако перед развертыванием системы принятия решений в реальном мире, как правило, желательно убедиться, что поведение найденной стратегии соответствует нашим ожиданиям. В этой главе обсуждаются различные аналитические инструменты для проверки стратегий принятия решений[1]. Глава начинается с обсуждения способов вычисления показателей качества стратегии. Оценка таких показателей может быть сложной вычислительной задачей, особенно когда они связаны с такими редкими событиями, как отказы. Мы обсудим методы повышения эффективности вычислений. Важно, чтобы наши системы были *робастными* – устойчивыми к различиям между аналитическими моделями и реальным миром. В этой главе рассмотрены методы анализа робастности. В основе многих систем принятия решений лежит компромисс между несколькими целями, и мы обрисуем в общих чертах способы анализа этих компромиссов. Глава завершается рассмотрением состязательного анализа, который можно использовать для поиска наиболее вероятной траектории отказа.

14.1. Оценка показателей качества стратегии

Разработав стратегию принятия решений, мы должны оценить, насколько она хороша на практике. Показатели качества стратегии зависят от конкретной задачи. Например, предположим, что мы построили систему предотвращения столкновений – либо с помощью какой-либо формы оптимизации скалярной функции вознаграждения, либо просто эвристически, как показано в примере 14.1, – и хотим оценить ее безопасность, вычислив вероятность столкновения при следовании нашей стратегии[2]. Или, допустим, мы разработали стратегию построения инвестиционных портфелей, и теперь нам интересно оценить вероятность того, что наша стратегия приведет к огромным убыткам или какова может быть ожидаемая доходность.

---

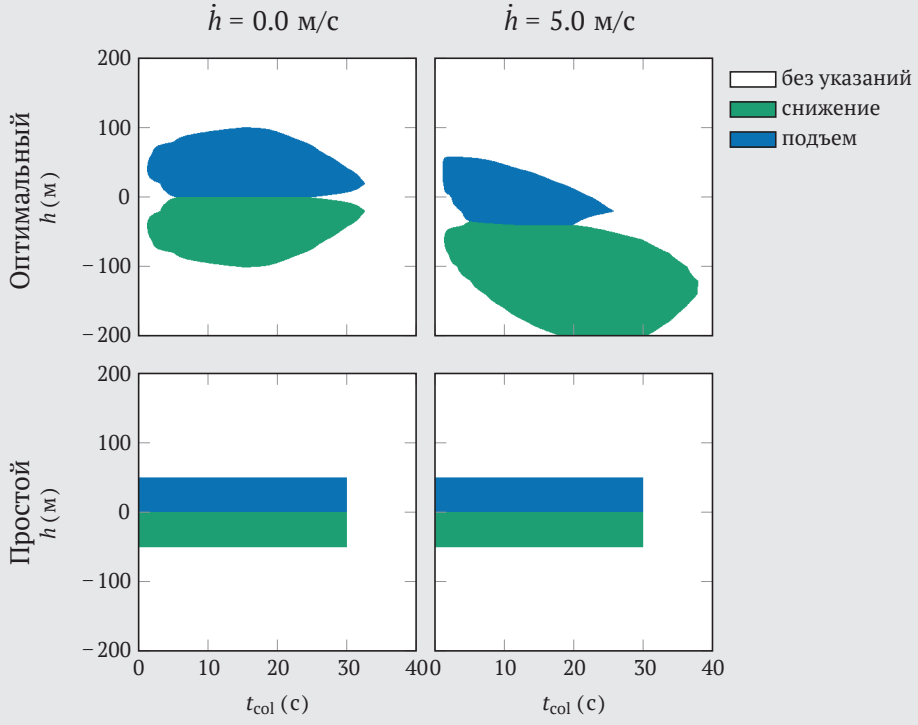
[1] Более подробное обсуждение представлено в статье A. Corso, R. J. Moss, M. Koren, R. Lee, M. J. Kochenderfer, *A Survey of Algorithms for Black-Box Safety Validation*, Journal of Artificial Intelligence Research, vol. 72, pp. 377–428, 2021.

[2] Другие показатели риска для безопасности полетов обсуждаются в статье I. L. Johansen, M. Rausand, *Foundations and Choice of Risk Metrics*, Safety Science, vol. 62, pp. 386–399, 2014.

Пример 14.1. Оптимальная и простая стратегии предотвращения столкновений. Более подробно задача описана в приложении F.6

В задаче предотвращения столкновений самолетов нам нужно решить, когда выдать рекомендации о наборе высоты или снижении для нашего самолета, чтобы избежать столкновения с самолетом на встречном курсе. Встречный самолет приближается к нам лоб в лоб, с постоянной горизонтальной скоростью сближения. Состояние определяется высотой h нашего самолета, измеренной относительно встречного самолета, нашей вертикальной скоростью $\dot{h}$, предыдущим действием a_{prev} и временем до возможного столкновения t_{col}. Штраф 1 за столкновение назначается, если встречный самолет находится в пределах 50 м, когда $t_{\text{col}} = 0$. Кроме того, применяется штраф 0.01, когда $a \neq a_{\text{prev}}$, чтобы воспрепятствовать чрезмерной выдаче рекомендаций.

Для получения оптимальной стратегии можно использовать динамическое программирование с линейной интерполяцией (раздел 8.4). В качестве альтернативы мы можем определить простую эвристическую стратегию, параметризованную пороговыми значениями t_{col} и h, которая работает следующим образом. Если $|h| < h_{\text{thresh}}$ и $t_{\text{col}} < t_{\text{thresh}}$, то генерируется рекомендация. Эта рекомендация заключается в наборе высоты, если $h > 0$, и снижении в противном случае. По умолчанию мы используем $h_{\text{thresh}} = 50$ м и $t_{\text{thresh}} = 30$ с. Ниже приведены графики оптимальных и простых стратегий для двух срезов пространства состояний:



На данный момент мы будем рассматривать один показатель f, оцениваемый для стратегии π. Часто этот показатель определяется как математическое ожидание траекторного показателя f_{traj}, оцениваемого по траекториям $\tau = (s_1, a_1, \ldots)$, полученным в результате следования стратегии:

$$f(\pi) = \mathbb{E}_{\tau}[f_{\text{traj}}(\tau)]. \tag{14.1}$$

Это ожидание некоторого распределения траекторий. Чтобы определить распределение траекторий, соответствующее задаче MDP, нам нужно указать *распределение начального состояния* (initial state distribution) b. Вероятность создания траектории τ равна

$$P(\tau) = P(s_1, a_1, \ldots) = b(s_1)\prod_t T(s_{t+1}|s_t, a_t). \tag{14.2}$$

В контексте предотвращения столкновений f_{traj} может быть равно 1, если траектория привела к столкновению, и 0 в противном случае. Ожидание будет соответствовать вероятности столкновения.

В некоторых случаях нас интересует изучение распределения выхода f_{traj}. На рис. 14.1 показан пример такого распределения. Математическое ожидание в уравнении (14.1) – это лишь один из многих способов преобразовать распределение показателей траектории в единственное значение. В данной главе мы сосредоточимся в первую очередь на этом ожидании, но будет полезно упомянуть, что другими популярными вариантами преобразований распределения в значение являются дисперсия, пятый процентиль и среднее значение выборок ниже пятого процентиля[3].

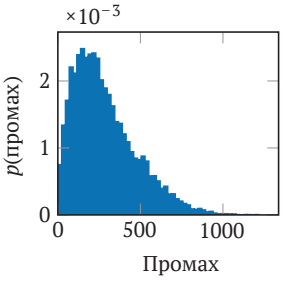


Рис. 14.1. Распределение расстояний промаха, полученное на основе 10^4 моделирований при следовании простой стратегии предотвращения столкновений из начальных состояний с $h \sim U(-10, 10)$, м; $\dot{h} \sim U(-200, 200)$, м/с; $a_{\text{prev}} = 0$ м/с; $t_{\text{col}} = 40$ с

[3] В литературе обсуждались различные меры риска. Обзор некоторых из них, используемых в контексте MDP, предоставлен в работе A. Ruszczyński, *Risk-Averse Dynamic Programming for Markov Decision Processes*, Mathematical Programming, vol. 125, no. 2, pp. 235–261, 2010.

Показатель траектории иногда можно записать в таком виде:

$$f_{\text{traj}}(\tau) = f_{\text{traj}}(s_1, a_1, \ldots) = \sum_t f_{\text{step}}(s_t, a_t), \tag{14.3}$$

где f_{step} – это функция, зависящая от текущего состояния и действия, очень похожая на функцию вознаграждения в MDP. Если $f(\pi)$ определяется как математическое ожидание f_{traj}, цель та же, что и при решении задачи MDP, где f_{step} – это просто функция вознаграждения. Таким образом, мы можем использовать алгоритмы оценки стратегии, представленные в разделе 7.2, для оценки нашей стратегии по отношению к любому параметру в уравнении (14.3).

Оценка стратегии фактически является функцией полезности состояния[4], которая возвращает значение интересующего показателя при старте из этого состояния. В примере 14.2 показаны срезы такой функции полезности для задачи предотвращения столкновений. Совокупное качество стратегии определяется из уравнения

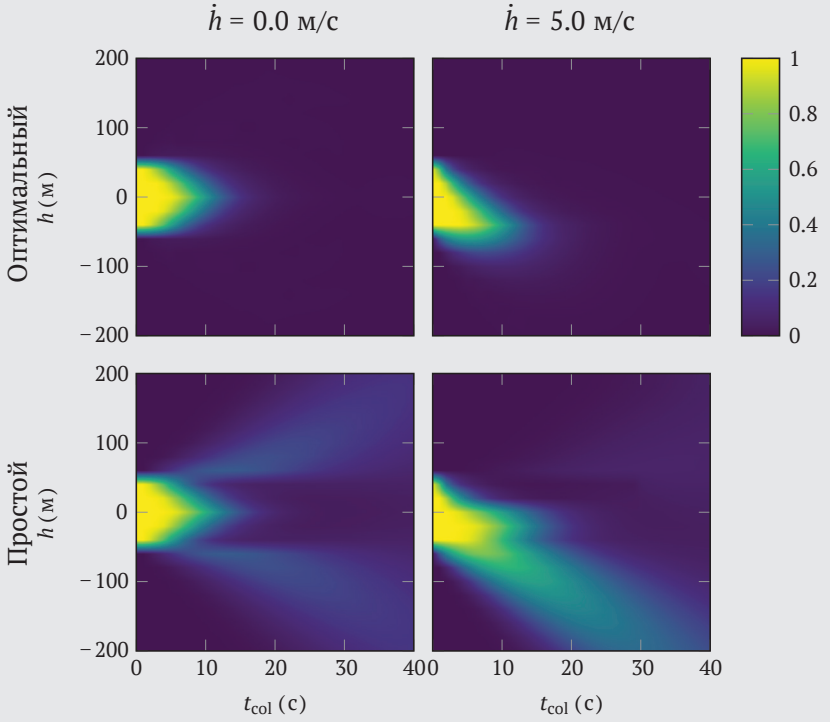
$$f(\pi) = \sum_t f_{\text{state}}(s) b(s), \tag{14.4}$$

где f_{state} – функция полезности, полученная в результате оценки стратегии.

Пример 14.2. Вероятность столкновения при соблюдении оптимальной и простой стратегий предотвращения столкновений

На рисунке ниже показан пример оценивания двух вариантов стратегии – оптимальной и простой, представленной ранее в примере 14.1. Каждая точка на графике соответствует значению показателя, обусловленному стартом из соответствующего состояния. Мы определяем $f_{\text{state}}(s, a) = 1$, если s является столкновением, и 0 в противном случае. Этот график показывает, где в пространстве состояний существует значительный риск столкновения, обозначенный более яркими цветами, при следовании определенной стратегии. Мы видим, что оптимальная стратегия вполне безопасна, особенно если $t_{\text{col}} > 20$ с. При слишком низком значении t_{col} даже оптимальная стратегия не может избежать столкновения из-за физических ограничений ускорения транспортного средства. Простая стратегия имеет гораздо более высокий уровень риска по сравнению с оптимальной стратегией, особенно когда $t_{\text{col}} > 20$ с, $\dot{h} = 5$ м/с, а встречный самолет находится ниже нашего – отчасти потому, что в простой стратегии выбор рекомендации не учитывает $\dot{h}$.

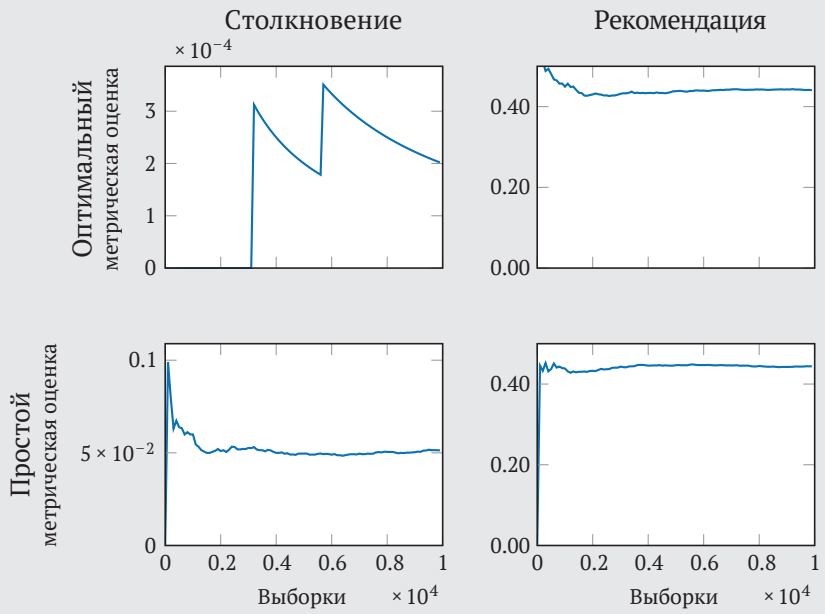
[4] Мы использовали U^{π} в предыдущих главах для обозначения функции полезности, связанной со стратегией π.



Если пространство состояний дискретно, то уравнение (14.4) можно решить аналитически. Однако если пространство состояний большое или непрерывное, иногда приходится вычислять $f(\pi)$ посредством выборки. Мы можем извлечь выборку из начального распределения по состояниям, а затем развернуть стратегию и вычислить показатель траектории. Затем мы можем оценить значение обобщенного показателя по среднему значению показателей траекторий. Качество оценки обычно улучшается с увеличением количества выборок. В примере 14.3 этот процесс продемонстрирован на оценке различных показателей, связанных со стратегией предотвращения столкновений.

Пример 14.3. Вероятность столкновения и правильной рекомендации при соблюдении оптимальной и простой стратегий предотвращения столкновений

Здесь мы вновь рассмотрим варианты двух стратегий – оптимальной и простой, – представленные в примере 14.1. Нам необходимо оценить вероятности столкновения и генерации правильной рекомендации. Чтобы оценить эти показатели, мы используем 10^4 выборок из распределения начального состояния, показанного на рис. 14.1, а затем выполняем развертывание. Графики ниже показывают кривые сходимости:



По графикам можно заключить, что оптимальная стратегия намного безопаснее, чем простая стратегия, при этом частота выдачи рекомендаций примерно одинаковая. Оценка показателя рекомендаций сходится гораздо быстрее, чем оценка столкновений. Причина более быстрой сходимости первого показателя заключается в том, что рекомендации встречаются чаще, чем столкновения. Столкновения при соблюдении оптимальной стратегии случаются настолько редко, что даже 10^4 выборок кажется недостаточно для точной оценки. Кривая очень неровная, с большими пиками в выборках, связанных со столкновениями, за которыми следует спад в оценке вероятности столкновения, поскольку моделируются выборки без столкновений.

Для измерения качества нашей оценки часто используют *стандартную ошибку* (standard error, SE):

$$SE = \hat{\sigma}/\sqrt{n}, \tag{14.5}$$

где $\hat{\sigma}$ – стандартное отклонение выборок, а n – их количество. В примере 14.3 стандартное отклонение показателя столкновений составляет 0.0173, что делает стандартную ошибку показателя вероятности столкновений равной 0.000173.

Мы можем преобразовать стандартную ошибку в *доверительный интервал* (confidence interval). Например, 95-процентный доверительный интервал составит $\hat{\mu} \pm 1.96$ SE, где $\hat{\mu}$ – среднее значение наших выборок. Для нашей задачи предотвращения столкновений этот интервал равен $(-3.94\times10^{-5}, 6.39\times10^{-4})$.

В качестве альтернативы мы можем использовать байесовский подход и представить нашу апостериорную вероятность в виде бета-распределения, как обсуждалось в разделе 4.2.

В случае малых вероятностей, таких как вероятности отказа в относительно безопасной системе, часто находят *относительную стандартную ошибку* (relative standard error), которая определяется выражением

$$\frac{\hat{\sigma}}{\hat{\mu}\sqrt{n}}. \tag{14.6}$$

Это выражение эквивалентно делению стандартной ошибки на среднее значение. В задаче предотвращения столкновений относительная ошибка составляет 0.578. Хотя абсолютная ошибка может быть небольшой, относительная ошибка довольно высока, поскольку мы пытаемся оценить небольшую вероятность.

14.2. Моделирование редких событий

В системе, где очень важны редкие события, нам может понадобиться много выборок для точной оценки, например, такого показателя, как вероятность столкновения. В примере с предотвращением столкновений 10^4 выборок содержали всего три столкновения, о чем свидетельствуют три пика на графике. Когда мы разрабатываем алгоритмы для систем с высокой ценой ошибочного решения (отказа), таких как системы биржевой торговли или управления автомобилями, точная оценка вероятности отказа с помощью прямой выборки и моделирования может быть сложной вычислительной задачей.

Распространенный подход к повышению точности и вычислительной эффективности решений называется *выборкой, взвешенной по значимости* (importance sampling). Он заключается в выборке из альтернативного распределения и взвешивании результатов для получения несмещенной оценки[5]. Мы использовали такой же подход в контексте логического вывода в байесовских сетях под названием *выборки, взвешенной по правдоподобию* (раздел 3.7). Альтернативное выборочное распределение часто называют *вспомогательным распределением* (proposal distribution). Мы будем использовать обозначение $P'(\tau)$ для представления вероятности, которую наше вспомогательное распределение присваивает траектории τ.

Выведем надлежащий способ взвешивания выборок из P'. Если у нас есть $\tau^{(1)}$, …, $\tau^{(n)}$, извлеченных из истинного распределения P, то мы имеем:

$$f(\pi) = \mathbb{E}_{\tau}[f_{\text{traj}}(\tau)] \tag{14.7}$$

---

[5] Более подробное введение в выборку по важности и другие методы моделирования редких событий представлено в книге J. A. Bucklew, *Introduction to Rare Event Simulation*. Springer, 2004.

$$= \sum_{\tau} f_{\text{traj}}(\tau) P(\tau) \tag{14.8}$$

$$\approx \frac{1}{n} \sum_{i} f_{\text{traj}}(\tau^{(i)}), \quad \text{где } \tau^{(i)} \sim P. \tag{14.9}$$

Мы можем умножить уравнение (14.8) на $P'(\tau)/P'(\tau)$ и получить следующее:

$$f(\pi) = \sum_{\tau} f_{\text{traj}}(\tau) P(\tau) \frac{P'(\tau)}{P'(\tau)} \tag{14.10}$$

$$= \sum_{\tau} f_{\text{traj}}(\tau) P'(\tau) \frac{P(\tau)}{P'(\tau)} \tag{14.11}$$

$$\approx \frac{1}{n} \sum_{i} f_{\text{traj}}(\tau^{(i)}) \frac{P(\tau^{(i)})}{P'(\tau^{(i)})}, \quad \text{где } \tau^{(i)} \sim P'. \tag{14.12}$$

Другими словами, нам нужно взвесить результаты выборок из вспомогательного распределения, где вес[6], присвоенный выборке i, равен $P(\tau^{(i)})/P'(\tau^{(i)})$.

Нам необходимо выбрать вспомогательное распределение P', чтобы сосредоточить внимание на выборках, которые являются «важными» в том смысле, что они с большей вероятностью способствуют правильной совокупной оценке. В случае задачи предотвращения столкновений мы стремимся сделать так, чтобы вспомогательное распределение поощряло столкновения, т. е. чтобы у нас было больше, чем просто несколько ситуаций столкновения для оценки риска. С другой стороны, нам не нужно, чтобы все выборки приводили к столкновениям. В общем, если предположить, что пространство исходов дискретно, оптимальное вспомогательное распределение описывается следующим уравнением:

$$P^{*}(\tau) = \frac{|f_{\text{traj}}(\tau)| P(\tau)}{\sum_{\tau'} |f_{\text{traj}}(\tau')| P(\tau')}. \tag{14.13}$$

Если значение f_{traj} неотрицательно, то знаменатель точно такой же, как показатель, который мы пытаемся найти в уравнении (14.1).

Хотя уравнение (14.13), как правило, нецелесообразно вычислять точно (вот почему мы в первую очередь используем выборку по значимости), оно может дать некоторое представление о том, как использовать знание предметной области для построения вспомогательного распределения. Обычно распределение начального состояния или модель перехода слегка смещают в сторону более важных траекторий, таких как столкновение.

Чтобы проиллюстрировать построение распределения с учетом значимости, воспользуемся оптимальной стратегией для задачи предотвращения столкно-

---

[6] Важно отметить, что P' не должен присваивать нулевую вероятность любой траектории, которой P присваивает положительную вероятность.

вений в примере 14.1. Вместо того чтобы начинать с $t_{col} = 40$ с, мы будем начинать моделирование с меньшего расстояния, с $t_{col} = 20$ с, чтобы усложнить задачу предотвращения столкновения. Истинное распределение имеет параметры $h \sim U(-10, 10)$ м и $\dot{h} \sim U(-200, 200)$ м/с. Однако для определенных сочетаний h и $\dot{h}$ сложно найти оптимальную стратегию. Мы использовали динамическое программирование для дискретной версии задачи, чтобы определить вероятность столкновения для различных значений h и $\dot{h}$, а затем нормализовали полученные результаты, чтобы превратить их во вспомогательное распределение, показанное на рис. 14.2.

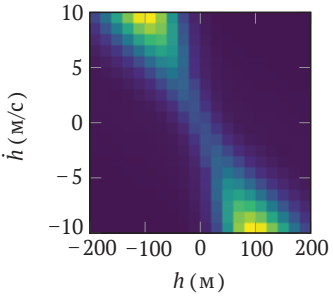


Рис. 14.2. Вспомогательное распределение, сгенерированное из вероятности столкновения при следовании оптимальным стратегиям предотвращения столкновений из разных начальных состояний с $t_{col} = 20$ с и $a_{prev} = 0$ м/с. Желтый цвет указывает на более высокую плотность вероятности

При одинаковом количестве выборок вспомогательное распределение, показанное на рис. 14.2, дает более точные оценки вероятности столкновения, чем начальное распределение. На рис. 14.3 изображены кривые сходимости. При 5×10^4 выборок оба метода сходятся к одной и той же оценке. Однако выборка, взвешенная по значимости, приближается к истинному значению уже в пределах 10^4 выборок. Опираясь на вспомогательное распределение, выборка по значимости сгенерировала 939 столкновений, в то время как выборка из исходного распределения сгенерировала только 246. Можно было бы сгенерировать еще больше столкновений, если бы мы также смещали распределение перехода, а не только распределение начального состояния.

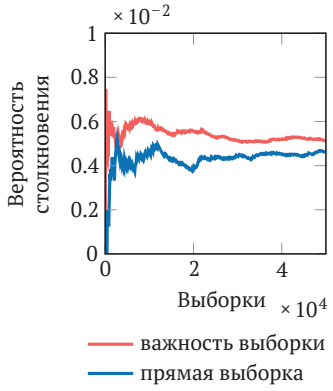


Рис. 14.3. Вероятность столкновения при следовании оптимальной стратегии, оцениваемая на основе выборки по значимости и прямой выборки

14.3. Анализ робастности системы

Перед развертыванием системы на производстве важно изучить ее робастность – устойчивость к отклонению параметров реальной среды от параметров, на которых обучена модель. Мы можем использовать инструменты, упомянутые в предыдущих разделах, такие как оценка стратегии и выборка по значимости, но должны оценивать наши стратегии в средах, которые отклоняются от модели, принятой при оптимизации стратегии. На рис. 14.4 показано, как меняется робастность стратегии, по мере того как реальная модель отклоняется от модели, которую использовали для оптимизации. Мы также можем изучить чувствительность наших показателей к предположениям моделирования в пространстве состояний (пример 14.4). Если качество стратегии по соответствующим показателям сохраняется при возможных возмущениях модели среды, это аргумент в пользу того, что в реальной среде система будет вести себя как запланировано.

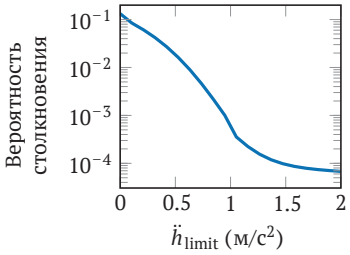


Рис. 14.4. Анализ устойчивости стратегии, оптимизированной для $\ddot{h}_{\text{limit}} = 1$ м/с^2, но оцененной в средах с разными значениями $\ddot{h}_{\text{limit}}$

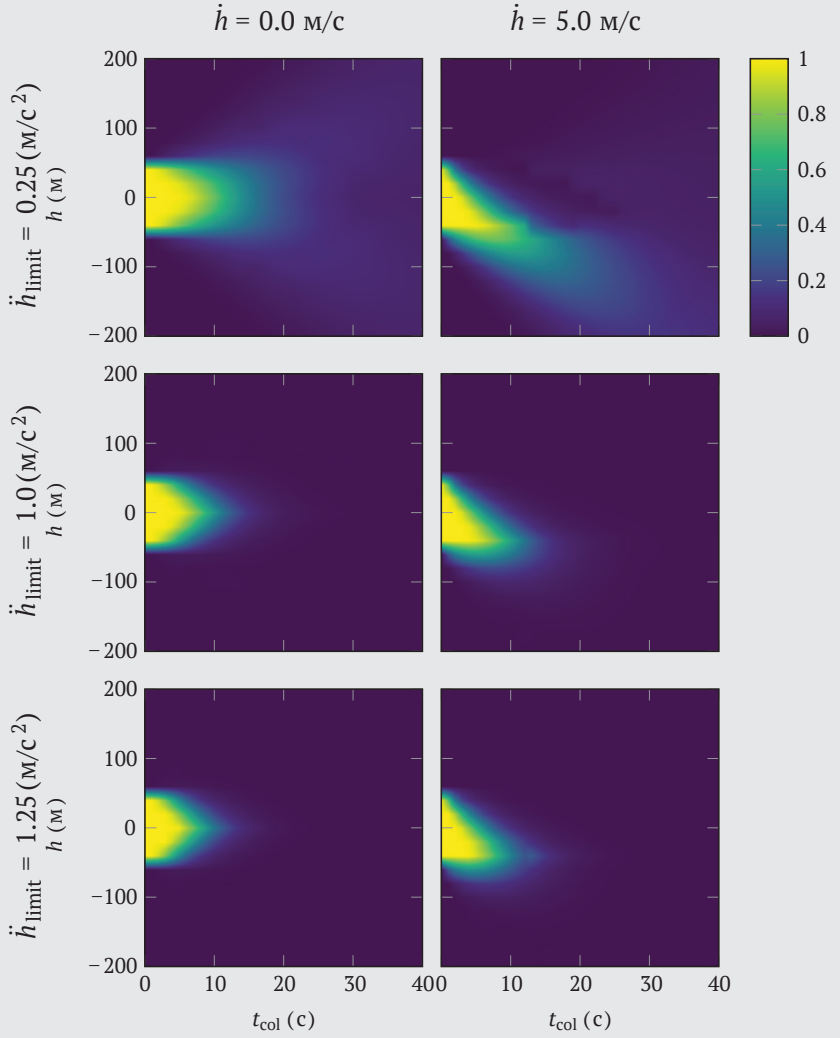
Как правило, мы стремимся к тому, чтобы наша *модель планирования* (planning model), которую мы используем для оптимизации стратегии, была относительно простой и не допускала *переобучения* (overfitting) – чрезмерного приспособления к потенциально ошибочным предположениям моделирования, которые не отражают реальный мир. Дополнительным преимуществом более простых моделей планирования является то, что они могут сделать планирование более эффективным в плане вычислений. В то же время *оценочная модель* может быть настолько сложной, насколько это оправдано или реализуемо. Например, мы можем использовать простую низкоразмерную дискретную модель динамики самолета при разработке стратегии предотвращения столкновений, а затем оценивать эту стратегию в непрерывном высокоточном моделировании. Более простая модель планирования часто более устойчива к возмущениям в оценочной модели.

Процесс проверки стратегий с использованием различных оценочных моделей иногда называют *стрессовым тестированием*, особенно если спектр оценочных моделей охватывает довольно экстремальные сценарии. При тестировании системы предотвращения столкновений к экстремальным сценариям могут относиться ситуации, когда воздушные суда сближаются друг с другом с чрезвычайно большими скоростями набора высоты, которые могут быть физически недостижимы. Понимание того, какие категории сценариев приводят к отказу системы, может оказаться полезным на этапе проектирования, даже

если мы решим не оптимизировать поведение системы для этих сценариев, поскольку они считаются нереалистичными.

Пример 14.4. Вероятность столкновения при соблюдении оптимальных стратегий предотвращения столкновений, когда существует расхождение между моделями, используемыми для планирования и для оценки

Мы можем построить графики вероятности столкновения при прогоне моделирования из разных начальных состояний, как в примере 14.2. Здесь мы используем стратегию, оптимизированную для параметров, указанных в приложении F.6, но изменяем предел $\ddot{h}_{\text{limit}}$ в оценочной модели.



Мы оптимизировали стратегию с $\ddot{h}_{\text{limit}} = 1$ м/с^2. Если на самом деле этот параметр составляет 0.25 м/с^2, то в некоторых состояниях стратегия работала бы плохо, поскольку для достижения заданной вертикальной скорости требуется больше времени. При $\ddot{h}_{\text{limit}} = 1.25$ м/с^2 стратегия становится немного безопаснее.

Если оказалось, что выбранная стратегия слишком чувствительна к сделанным ранее предположениям о моделировании, имеет смысл воспользоваться методом, известным как *робастное динамическое программирование* (robust dynamic programming)[7]. Согласно этому методу, вместо единственной конкретной модели перехода у нас есть набор моделей перехода $T_{1:n}$ и моделей вознаграждения $R_{1:n}$. Мы можем пересмотреть уравнение оператора Беллмана (7.16), чтобы обеспечить робастность различных моделей следующим образом:

$$U_{k+1}(s) = \max_a \min_i \left(R_i(s,a) + \gamma \sum_{s'} T_i(s'|s,a) U_k(s') \right). \tag{14.14}$$

Оператор выбирает действие, которое обеспечивает максимальную ожидаемую полезность при использовании модели с минимальной полезностью.

14.4. Анализ компромиссов

Многие сложные задачи имеют несколько целей, которые противоречат друг другу. Например, в автономных системах часто приходится искать компромисс между безопасностью и вычислительной эффективностью. При разработке системы предотвращения столкновений мы стремимся обеспечить максимальную безопасность, не совершая слишком много ненужных маневров. *Анализ компромиссов* (trade analysis) изучает, как меняются различные показатели качества стратегии при изменении проектных параметров.

Если рассматривать только два показателя качества, можно построить кривую компромисса, подобную той, что обсуждалась в примере 14.5. Варьируя параметры в стратегии, мы получаем разные значения двух показателей. Эти кривые полезны при сравнении различных методик разработки стратегий. Например, кривые в примере 14.5 показывают, что метод динамического программирования способен обеспечить значительные преимущества по сравнению с простыми пороговыми стратегиями – по крайней мере, в том виде, как мы их определили.

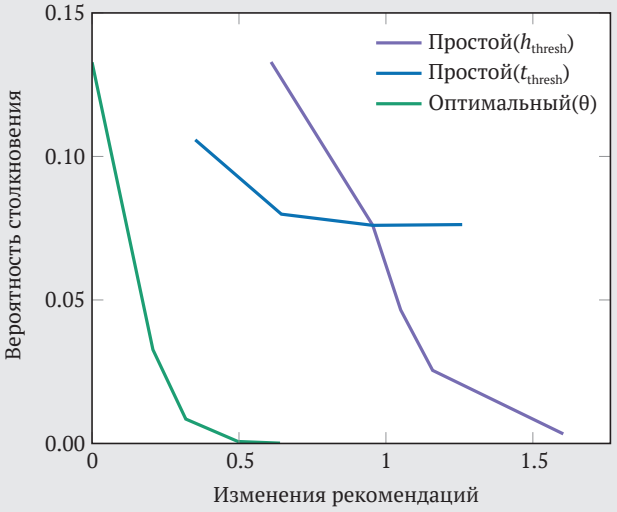
---

[7] G. N. Iyengar, *Robust Dynamic Programming*, Mathematics of Operations Research, vol. 30, no. 2, pp. 257–280, 2005. Этот подход может повысить робастность в контексте предотвращения столкновений. M. J. Kochenderfer, J. P. Chryssanthacopoulos, P. Radecki, *Robustness of Optimized Collision Avoidance Logic to Modeling Errors*, in Digital Avionics Systems Conference (DASC), 2010.

Пример 14.5. Анализ компромисса между безопасностью и эффективностью эксплуатации при изменении параметров различных систем предотвращения столкновений

В нашей задаче предотвращения столкновений самолетов мы должны найти компромисс между такими показателями, как вероятность столкновения и ожидаемое количество рекомендуемых изменений курса. Оба они могут быть реализованы с использованием траекторных показателей, которые аддитивно разлагаются на шаги, как это сделано в уравнении (14.3), что позволяет нам вычислить их, используя точную оценку стратегии.

На графике ниже показаны три кривые, связанные с различными параметрическими версиями простой и оптимальной стратегий. Первая кривая показывает качество простой стратегии по двум показателям при изменении параметра h_{thresh} (определенного в примере 14.1). Вторая кривая показывает качество простой стратегии при изменении t_{thresh}. Третья кривая показывает оптимальную стратегию при изменении параметра θ, где стоимость столкновения равна −θ, а стоимость изменения рекомендации равна −(1 − θ).



Мы видим, что оптимальная стратегия доминирует над кривыми, выработанными простыми параметрическими стратегиями. Когда θ приближается к 1, мы в полной безопасности, но нам приходится мириться с большим количеством рекомендаций по изменению курса. Когда θ приближается к 0, мы в меньшей безопасности, но почти не получаем рекомендаций. Выбрав определенный пороговый уровень безопасности, мы можем создать оптимизированную стратегию, от которой можно ожидать меньше рекомендуемых изменений курса, чем от любой из простых параметрических стратегий.

Для каждой кривой в примере 14.5 мы изменяем только один параметр за раз, но чтобы получить удовлетворительную систему, нам может потребоваться изучить влияние изменения нескольких параметров. Поскольку мы варьируем несколько параметров, в итоге получается пространство возможных стратегий. Некоторые из этих стратегий могут работать хуже по всем показателям качества по сравнению с хотя бы одной другой стратегией в этой области. Часто удается исключить из рассмотрения те стратегии, над которыми явно доминируют другие. Стратегия называется *оптимальной по Парето*[8], или *эффективной по Парето*, если над ней не доминирует никакая другая стратегия в рассматриваемом пространстве. Набор стратегий, оптимальных по Парето, называется *границей Парето*, или (в частном случае двух измерений) *кривой Парето*. На рис. 14.5 показан пример кривой Парето.

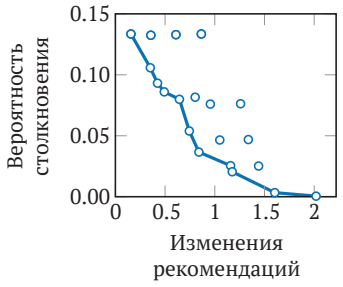


Рис. 14.5. Качество стратегий, сгенерированных изменением параметров простой стратегии из примера 14.1. Приблизительная кривая Парето выделена синим цветом

14.5. Состязательный анализ

Иногда бывает полезно изучить робастность стратегии с точки зрения *состязательного анализа* (adversarial analysis). На каждом временном шаге наш *противник* (adversary) выбирает состояние, возникающее в результате выполнения действия, указанного стратегией, из текущего состояния. У противника есть две цели, между которыми он стремится найти оптимальный баланс: минимизировать наш доход и максимизировать правдоподобие результирующей траектории в соответствии с имеющейся моделью перехода. Исходную задачу можно преобразовать в состязательную. Состязательное пространство состояний такое же, как и в исходной задаче, но состязательное пространство действий – это пространство состояний исходной задачи. Состязательное вознаграждение вычисляется следующим образом:

$$R'(s, a) = -R(s, \pi(s)) + \lambda \log(T(a|s, \pi(s))), \quad (14{,}15)$$

где π – наша стратегия, R – наша исходная функция вознаграждения, T – наша исходная модель перехода, а $\lambda \geqslant 0$ – параметр, определяющий важность макси-

[8] Назван в честь итальянского экономиста Вильфредо Федерико Дамасо Парето (1848–1923).

мизации результирующего правдоподобия траектории. Поскольку противник пытается максимизировать сумму состязательного вознаграждения, он максимизирует ожидаемый отрицательный доход плюс λ, умноженное на логарифмическую вероятность результирующей траектории[9]. Модель состязательного перехода является детерминированной; состояние переходит точно в такое, которое оппонент указывает в качестве своего действия.

В алгоритме 14.1 реализовано преобразование исходной задачи к состязательной. Алгоритм предполагает дискретное пространство состояний и действий, которые затем могут быть выбраны с помощью одного из алгоритмов динамического программирования, описанных в главе 7. Решением является состязательная стратегия, которая отображает состояния в состояния. Исходя из начального состояния, мы можем сгенерировать траекторию, которая минимизирует вознаграждение с заданным уровнем вероятности. Поскольку задача является детерминированной, на самом деле это задача поиска, и можно использовать любой из алгоритмов, упомянутых в приложении Е. Если задача является многомерной или непрерывной, можно применить один из методов приближенного решения, рассмотренных в главах 8 и 9.

Алгоритм 14.1. Преобразование задачи в состязательную для заданной стратегии π. Состязательный агент (противник) пытается изменить результаты наших действий в рамках стратегии, стремясь минимизировать нашу исходную полезность и максимизировать правдоподобие траектории. Параметр λ определяет, насколько важно максимизировать правдоподобие результирующей траектории. Алгоритм возвращает MDP, модели перехода и вознаграждения которого представлены в виде матриц

```
function adversarial(𝒫::MDP, π, λ)
    𝒮, 𝒜, T, R, γ = 𝒫.𝒮, 𝒫.𝒜, 𝒫.T, 𝒫.R, 𝒫.γ
    𝒮′ = 𝒜′ = 𝒮
    R′ = zeros(length(𝒮′), length(𝒜′))
    T′ = zeros(length(𝒮′), length(𝒜′), length(𝒮′))
    for s in 𝒮′
        for a in 𝒜′
            R′[s,a] = -R(s, π(s)) + λ*log(T(s, π(s), a))
            T′[s,a,a] = 1
        end
    end
    return MDP(T′, R′, γ)
end
```

Иногда нам нужно найти *наиболее вероятный отказ* (most likely failure), связанный с выбранной стратегией. В некоторых задачах отказом можно считать переход в определенное состояние. Например, столкновение самолетов можно

[9] Логарифмическая вероятность траектории равна сумме логарифмических вероятностей переходов отдельных состояний.

считать отказом системы в нашей задаче предотвращения столкновений. Для других задач может потребоваться более сложное определение отказа, выходящее за рамки простого входа в подмножество пространства состояний. Например, иногда бывает необходимо определить отказ, используя *темпоральную логику* (temporal logic) – способ представления и рассуждения о предположениях, определяемых с точки зрения времени. Однако во многих случаях мы можем использовать эти определения отказов для создания расширенного пространства состояний, а затем решить его[10].

Определив состояния отказа, мы можем найти наиболее вероятную траекторию отказа, изменив функцию вознаграждения в уравнении (14.15) на следующую:

$$R'(s,a) = \begin{cases} -\infty, & \text{если } s \text{ является конечным и не отказ} \\ 0, & \text{если } s \text{ является конечным и отказ} \\ \log\left(T(a|s,\pi(s))\right) & \text{в остальных случаях} \end{cases}. \tag{14.16}$$

Для поиска наиболее вероятных отказов можно использовать различные методы аппроксимации. В зависимости от метода аппроксимации иногда важно ослабить бесконечный штраф за отказ, чтобы поиск мог приводить к отказам. Если применить поиск по дереву Монте-Карло для предотвращения столкновений, штраф может быть связан с расстоянием промаха[11].

Мы можем воспроизвести наиболее вероятную траекторию отказа и оценить, заслуживает ли эта траектория беспокойства. Если траектория выглядит крайне неправдоподобной, этот факт подкрепляет нашу уверенность в том, что выбранная стратегия безопасна. Однако если траектория отказа действительно заслуживает беспокойства, у нас есть несколько вариантов устранения проблемы:

1) *изменить пространство действий.* Мы можем добавить больше экстремальных маневров в наш набор действий для решения задачи предотвращения столкновений;
2) *изменить функцию вознаграждения.* Мы можем уменьшить затраты на изменение рекомендаций с целью снижения риска столкновения, как показано на компромиссной кривой в примере 14.5;
3) *изменить функцию перехода.* Мы можем увеличить предел вертикального ускорения, чтобы самолет мог быстрее достигать нужных вертикальных скоростей, если это следует из нашей стратегии;
4) *улучшить решатель.* Возможно, мы использовали слишком грубую дискретизацию пространства состояний и пропустили важные особенности

---

[10] M. Bouton, J. Tumova, M. J. Kochenderfer, *Point-Based Methods for Model Checking in Partially Observable Markov Decision Processes*, in AAAI Conference on Artificial Intelligence (AAAI), 2020.

[11] Эта стратегия была представлена в работе R. Lee, M. J. Kochenderfer, O. J. Mengshoel, G. P. Brat, M. P. Owen, *Adaptive Stress Testing of Airborne Collision Avoidance Systems*, in Digital Avionics Systems Conference (DASC), 2015.

оптимальной стратегии. Можно увеличить степень дискретизации ценой увеличения объема вычислений, чтобы получить лучшую стратегию. В качестве альтернативы можно выбрать другой способ аппроксимации;

5) *не использовать систему принятия решений.* Если стратегия небезопасна, возможно, лучше не применять ее в реальном мире.

14.6. Заключение

- Показатели качества стратегий могут быть получены с использованием методов динамического программирования, рассмотренных в предыдущих главах, или с помощью развертывания выборки.
- Мы можем оценить нашу уверенность в показателях качества, используя стандартную ошибку, доверительные интервалы или один из байесовских подходов, рассмотренных ранее.
- Для более точного вычисления вероятности редких событий можно использовать метод, называемый выборкой по значимости.
- Выборка по значимости подразумевает выборку из альтернативного распределения и соответствующее взвешивание результатов.
- Поскольку модель, используемая для оптимизации, может быть неточным представлением реального мира, важно изучить чувствительность найденной стратегии к допущениям моделирования.
- Робастное динамическое программирование может повысить устойчивость стратегии к неопределенности модели за счет оптимизации на основе набора различных моделей перехода и вознаграждения.
- Анализ компромиссов помогает найти баланс между несколькими критериями качества при оптимизации стратегии.
- Состязательный анализ основан на действиях оппонента, выбирающего состояние, в которое мы переходим на каждом шаге, таким образом, чтобы минимизировать нашу цель и максимизировать правдоподобие траектории.

14.7. Упражнения

Упражнение 14.1. Пусть дана следующая траектория τ:

s_1	a_1	s_2	a_2	s_3
6.0	2.2	1.4	0.7	6.0

Динамика является линейной по Гауссу, $T(s'|s, a) = \mathcal{N}(s'|2s + a, 5^2)$, а распределение начального состояния задано как $\mathcal{N}(5, 6^2)$. Каково логарифмическое правдоподобие траектории τ?

Решение. Логарифмическая вероятность траектории равна

$$\log\mathcal{N}(6.0|5, 6^2) + \log\mathcal{N}(1.4|2\cdot 6.0 + 2.2, 5^2) + \log\mathcal{N}(6.0|2\cdot 1.4 + 0.7, 5^2) \approx -11.183.$$

Упражнение 14.2. Допустим, вы провели миллион прогонов модели и обнаружили, что система предотвращения столкновений допустила 10 отказов (столкновений). Какова ваша оценка вероятности столкновения и относительная стандартная ошибка?

Решение. Оценка вероятности столкновения равна

$$\hat{\mu} = 10/10^6 = 10^{-5}.$$

Здесь i-я выборка x_i равна 1, если произошло столкновение, и 0 в противном случае. Стандартное отклонение:

$$\hat{\sigma} = \sqrt{\frac{1}{10^6 - 1}\sum_{i=1}^{n}(x_i - \hat{\mu})^2} = \sqrt{\frac{1}{10^6 - 1}\left(10(1-\hat{\mu})^2 + (10^6 - 10)\hat{\mu}^2\right)} \approx 0.00316.$$

Относительная ошибка:

$$\frac{\hat{\sigma}}{\hat{\mu}\sqrt{n}} \approx \frac{0.00316}{10^{-5}\sqrt{10^6}} = 0.316.$$

Упражнение 14.3. Допустим, вам нужно вычислить математическое ожидание $\mathbb{E}_{x\sim U(0,5)}[f(x)]$, где $f(x)$ равно -1, если $|x| \leqslant 1$, и 0 в противном случае. Каким будет оптимальное вспомогательное распределение?

Решение. Оптимальное вспомогательное распределение имеет вид

$$p^*(x) = \frac{|f(x)|p(x)}{\int |f(x)|p(x)\mathrm{d}x},$$

что эквивалентно $U(0, 1)$, потому что $f(x)$ отлично от нуля только для $x \in [-1, 1]$, функция $U(0, 5)$ определена только для $x \in [0, 5]$, и как $f(x)$, так и $p(x)$ дают постоянные значения, когда они отличны от нуля.

Упражнение 14.4. Предположим, вы взяли выборку 0.3 из вспомогательного распределения в предыдущем упражнении. Каков ее вес? Какова оценка ожидания $\mathbb{E}_{x\sim U(0,5)}[f(x)]$?

Решение. Вес равен $p(x)/p^*(x) = 0.2/1$. Поскольку $f(0.3) = -1$, оценка равна -0.2, что является точным ответом.

Упражнение 14.5. Предположим, у вас есть следующие четыре стратегии, оцененные по трем показателям, которые вы стремитесь максимизировать:

Система	f_1	f_2	f_3
π_1	2.7	1.1	2.8
π_2	1.8	2.8	4.5
π_3	9.0	4.5	2.3
π_4	5.3	6.0	2.8

Какие стратегии находятся на границе Парето?

Решение. Только над π_1 преобладают все остальные стратегии. Следовательно, π_2, π_3 и π_4 находятся на границе Парето.

Часть III

Неопределенность модели

До сих пор в нашем обсуждении задач последовательного принятия решений мы предполагали, что модели перехода и вознаграждения известны. Однако во многих задачах эти модели точно неизвестны и агент должен учиться действовать на собственном опыте. Наблюдая за результатами своих действий в виде переходов состояний и вознаграждений, агент должен выбирать действия, которые максимизируют его долгосрочное накопленное вознаграждение. Решение задач, в которых присутствует неопределенность модели, является предметом *обучения с подкреплением* (reinforcement learning), которому посвящена третья часть книги. Мы обсудим несколько проблем, связанных с устранением неопределенности модели. Во-первых, агент должен тщательно сбалансировать исследование окружающей среды с использованием знаний, полученных в результате опыта. Во-вторых, вознаграждение может быть получено спустя долгое время после того, как важные решения были приняты, так что запоздавшие вознаграждения должны быть правильно отнесены к ранним решениям. В-третьих, агент должен уметь обобщать ограниченный опыт. Мы рассмотрим теорию и некоторые ключевые алгоритмы, направленные на решение этих проблем.

15 Исследование среды и использование знаний

Агенты обучения с подкреплением[1] должны найти оптимальный баланс между *исследованием* (exploration) окружающей среды и *использованием* (exploitation) знаний, полученных в результате взаимодействия с ней[2]. Чистое исследование среды позволит агенту построить исчерпывающую модель, но, скорее всего, ему придется отказаться от получения вознаграждения. Чистое использование знаний заключается в том, что агент постоянно выбирает действие, которое, по его мнению, лучше всего добавляет вознаграждение, но могут существовать и лучшие действия, дающие большее вознаграждение. В этой главе рассматриваются проблемы, связанные с поиском компромисса между исследованием и использованием, на примере задачи с одним состоянием. Главу завершает пример изучения среды в задаче типа MDP с несколькими состояниями.

15.1. Задача однорукого бандита

Ранние попытки найти компромисс между исследованием и использованием были сосредоточены на игровых автоматах, также называемых *однорукими бандитами*[3]. Название происходит от старых игровых автоматов с одной пус-

---

[1] Обзор области обучения с подкреплением представлен в книге M. Wiering, M. van Otterlo, eds., *Reinforcement Learning: State of the Art*. Springer, 2012.

[2] В некоторых приложениях мы стремимся оптимизировать стратегию при фиксированном наборе траекторий. Этот прием известен как *пакетное обучение с подкреплением*. В данной главе предполагается, что мы должны собирать собственные данные посредством взаимодействия, что делает важным выбор соответствующей стратегии исследования.

[3] Эти задачи игровых автоматов были изучены во время Второй мировой войны и оказались исключительно сложными для решения. По словам Питера Уиттла, «усилия по поиску решения [задач игровых автоматов] настолько истощили энергию и умы аналитиков союзников, что даже прозвучало предложение переложить эту проблему на Германию в качестве главного инструмента интеллектуального саботажа». J. C. Gittins, *Bandit Processes and Dynamic Allocation Indices*, Journal of the Royal Statistical Society. Series B (Methodological), vol. 41, no. 2, pp. 148–177, 1979.

ковой рукояткой, а также от того факта, что при достаточно долгой игре автомат забирает у игрока все деньги. Многие сценарии реального мира можно рассматривать как *задачи многорукого бандита* (multiarmed bandit problem)[4]. К ним относятся, например, распределение клинических испытаний и адаптивная сетевая маршрутизация. В литературе существует множество формулировок задачи бандитов, но в этой главе основное внимание будет уделено тому, что называется *бинарным бандитом* (binary bandit), *бандитом Бернулли* (Bernoulli bandit) или *биномиальным бандитом* (binomial bandit). В этих задачах рывок за руку a дает выплату, равную 1, с вероятностью θ_a и 0 в противном случае. Дернуть за руку автомата ничего не стоит, но доступно только h попыток.

Задачу о бандитах можно рассматривать как h-шаговый MDP с одним состоянием, n действиями и неизвестной стохастической функцией вознаграждения $R(s, a)$, как показано на рис. 15.1. Напомним, что $R(s, a)$ – это *ожидаемое* вознаграждение за выполнение действия a в состоянии s, но индивидуальные вознаграждения, реализуемые в среде, могут быть основаны на распределении вероятностей.

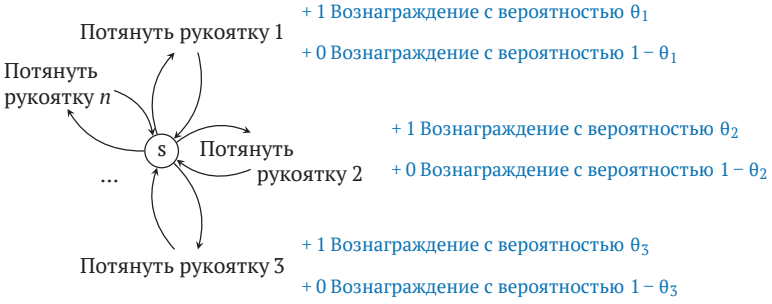


Рис. 15.1. Задача многорукого бандита – это MDP с одним состоянием, где действия могут различаться только вероятностью того, что они принесут вознаграждение

Алгоритм 15.1 определяет рабочий цикл модели для задачи о бандитах. На каждом шаге мы формируем стратегию исследования π на основе текущей модели вероятностей выплат, чтобы совершить действие a. В следующем разделе будет рассмотрен способ моделирования вероятностей выплат, а в оставшейся части главы будет описано несколько стратегий изучения среды. Получив a, мы моделируем рывок за эту руку, возвращая бинарное вознаграждение r. Затем модель обновляется на основе наблюдаемых a и r. Цикл моделирования повторяется до горизонта h.

[4] C. Szepesvári, T. Lattimore, Bandit Algorithms. Cambridge University Press, 2020.

Алгоритм 15.1. Моделирование задачи бандита. Данная задача определяется вектором вероятностей выплат θ, по одной выплате на каждое действие. Мы также определяем функцию R, которая имитирует генерацию стохастического бинарного вознаграждения в ответ на выбор действия. Каждый шаг моделирования включает создание действия a на основе стратегии изучения среды π. Стратегия исследования при выборе действия обычно опирается на модель. Выбор этого действия приводит к случайно сгенерированному вознаграждению, которое затем используется для обновления модели. Моделирование проводится до горизонта h

```
struct BanditProblem
    θ # vector of payoff probabilities
    R # reward sampler
end

function BanditProblem(θ)
    R(a) = rand() < θ[a] ? 1 : 0
    return BanditProblem(θ, R)
end

function simulate(𝒫::BanditProblem, model, π, h)
    for i in 1:h
        a = π(model)
        r = 𝒫.R(a)
        update!(model, a, r)
    end
end
```

15.2. Оценка байесовской модели

Нам необходимо проследить эволюцию нашей уверенности в вероятности выигрыша θ_a для руки a. Для представления этой уверенности часто используется бета-распределение (раздел 4.2). Если предположить наличие равномерного априорного распределения Beta(1, 1), апостериорное распределение вероятностей для θ_a после w_a выигрышей и ℓ_a проигрышей имеет вид Beta(w_a + 1, ℓ_a + 1). Апостериорная вероятность выигрыша равна

$$\rho_a = P(\text{win}_a | w_a, \ell_a) = \int_0^1 \theta \times \text{Beta}(\theta | w_a + 1, \ell_a + 1) \text{d}\theta = \frac{w_a + 1}{w_a + \ell_a + 2}. \quad (15.1)$$

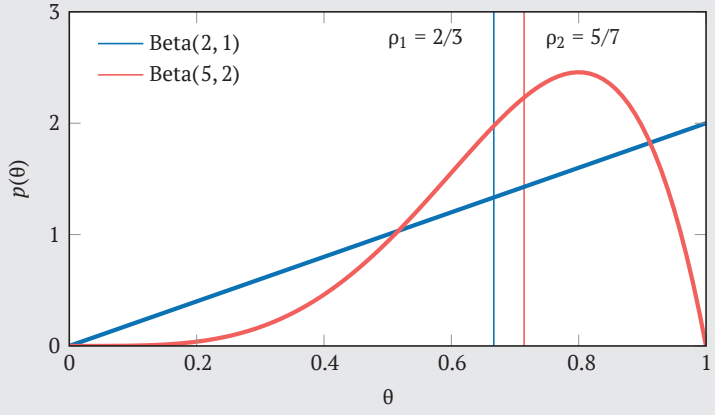
Реализация этого байесовского подхода представлена в алгоритме 15.2. В примере 15.1 показано, как вычислить апостериорные распределения на основе подсчета выигрышей и проигрышей.

Алгоритм 15.2. Байесовская функция обновления модели бандита. Наблюдая за вознаграждением `r` после выполнения действия `a`, мы обновляем бета-распределение, связанное с этим действием, увеличивая соответствующий параметр

```
struct BanditModel
    B # vector of beta distributions
end

function update!(model::BanditModel, a, r)
    α, β = StatsBase.params(model.B[a])
    model.B[a] = Beta(α + r, β + (1-r))
    return model
end
```

Пример 15.1. Апостериорные распределения вероятностей и ожидаемые выплаты для многорукого бандита

Предположим, у нас есть двурукий бандит, с которым мы сыграли шесть раз. Первая рука дала 1 выигрыш и 0 проигрышей, а другая дала 4 выигрыша и 1 проигрыш. Исходя из предположения о равномерном априорном распределении, апостериорное распределение для θ_1 – Beta(2, 1), а апостериорное распределение для θ_2 – Beta(5, 2).



Эти апостериорные распределения присваивают ненулевое правдоподобие вероятности выигрыша между 0 и 1. Плотность в точке 0 равна 0 для обеих рук, потому что они обе принесли хотя бы одну победу. Точно так же плотность в точке 1 для руки 2 равна 0, потому что она принесла по крайней мере один проигрыш. Вероятности выигрыша $\rho_1 = 2/3$ и $\rho_2 = 5/7$ показаны вертикальными линиями. Мы считаем, что вторая рука имеет наилучшие шансы принести выплату.

Жадное действие (greedy action) – это действие, которое максимизирует наше ожидаемое немедленное вознаграждение, или, другими словами, апостериорную вероятность выигрыша в контексте нашей задачи бинарного бандита. Иногда бывает доступно несколько жадных действий. Но это не означает, что мы обязаны всегда выбирать жадное действие, потому что есть риск упустить возможность обнаружить другое действие, которое обеспечивает более высокую награду в отдаленной перспективе. Для изучения нежадных действий можно использовать информацию из бета-распределений, связанную с различными действиями.

15.3. Стратегии ненаправленного исследования

Существует несколько специальных стратегий исследования, которые обычно применяют для достижения компромисса между исследованием и использованием. В этом разделе обсуждается стратегия *ненаправленного исследования* (undirected exploration), в соответствии с которой мы не используем информацию из предыдущих результатов для выбора направления поиска нежадных действий.

Одной из наиболее распространенных ненаправленных стратегий является ε-*жадное исследование* (алгоритм 15.3). Эта стратегия выбирает случайную руку с вероятностью ε. В иных случаях будет выбрана жадная рука $\text{argmax}_a \rho_a$. Здесь ρ_a представляет собой апостериорную вероятность выигрыша с действием *a* с использованием байесовской модели, приведенной в предыдущем разделе. В качестве альтернативы мы можем использовать оценку максимального правдоподобия, но при достаточно большом количестве попыток разница между этими двумя подходами невелика. Большие значения ε приводят к более обширному охвату исследования, что приводит к более быстрому определению лучшей руки, но больше усилий тратится впустую на неоптимальные руки. Пример 15.2 демонстрирует ε-*жадную* стратегию исследования и эволюцию наших убеждений.

Несмотря на большую неопределенность в начале взаимодействия с бандитом, ε-жадный метод сохраняет постоянный объем исследования. Одной из распространенных корректировок метода является затухание ε с течением времени, например по экспоненциальному закону. Обновление параметра ε записывают следующим образом:

$$\varepsilon \leftarrow \alpha\varepsilon, \tag{15.2}$$

где значение $\alpha \in (0, 1)$ обычно близко к 1.

Алгоритм 15.3. Реализация стратегии ε-жадного исследования. С вероятностью ε она вернет случайное действие. В ином случае она вернет жадное действие

```
mutable struct EpsilonGreedyExploration
    ε # probability of random arm
end
```

```
function (π::EpsilonGreedyExploration)(model::BanditModel)
    if rand() < π.ε
        return rand(eachindex(model.B))
    else
        return argmax(mean.(model.B))
    end
end
```

Пример 15.2. Применение стратегии ε-жадного исследования к задаче о двуруком бандите

Применим ε-жадную стратегию исследования к двурукому бандиту. Мы можем построить модель с равномерным априорным распределением и стратегией исследования с ε = 0.3:

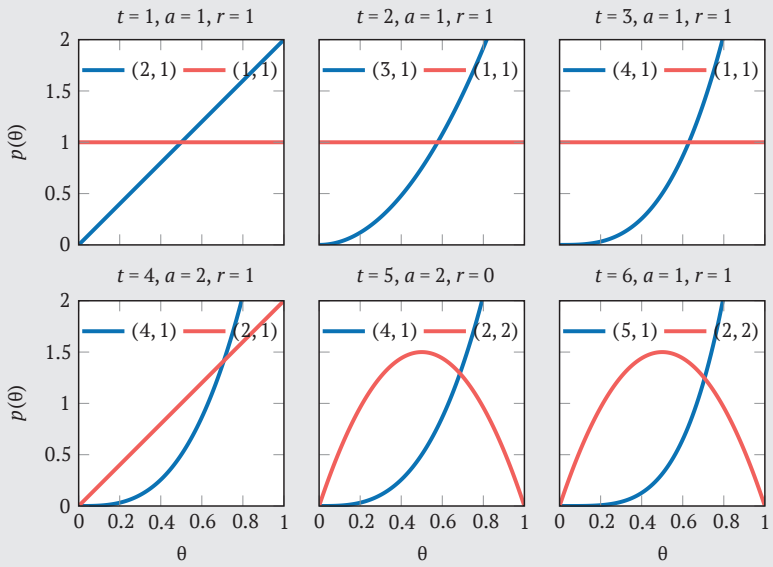
```
model(fill(Beta(),2))
π = EpsilonGreedyExploration(0.3)
```

Чтобы получить наше первое действие, мы вызываем функцию `π(model)`, которая возвращает 1 в зависимости от текущего состояния генератора случайных чисел. Мы наблюдаем проигрыш с $r = 0$, а затем вызываем метод

```
update!(model, 1, 0)
```

который обновляет бета-распределения в модели, чтобы отразить, что мы выполнили действие 1 и получили вознаграждение 0.

Представленные ниже графики показывают эволюцию убеждений о выигрыше после каждого из шести шагов использования нашей стратегии исследования. Синяя кривая относится к первой руке, а красная – ко второй:



Другой стратегией является подход *«исследуй, затем сделай»* (explore-then-commit), реализация которого представлена в алгоритме 15.4, когда мы выбираем действия равномерно случайным образом в течение первых k временных шагов, а дальше предпочитаем жадное действие[5]. Большие значения k снижают риск совершения неоптимального действия, но мы тратим больше времени на исследование потенциально неоптимальных действий.

Алгоритм 15.4. Стратегия «исследуй, затем сделай». Если значение `k` строго положительное, алгоритм вернет случайное действие после уменьшения `k`. В противном случае он вернет жадное действие

```
mutable struct ExploreThenCommitExploration
    k # количество случайных действий на этапе исследования
end

function (π::ExploreThenCommitExploration)(model::BanditModel)
    if π.k > 0
        π.k -= 1
        return rand(eachindex(model.B))
    end
    return argmax(mean.(model.B))
end
```

15.4. Стратегии направленного исследования

Направленное исследование (directed exploration) использует информацию, собранную из результатов предыдущих попыток, чтобы направлять исследование нежадных действий. Например, *стратегия softmax* (алгоритм 15.5) дергает за руку a с вероятностью, пропорциональной $\exp(\lambda \rho_a)$, где *параметр точности* $\lambda \geqslant 0$ определяет объем исследования. Мы получаем равномерный случайный отбор при $\lambda \to 0$ и жадный отбор при $\lambda \to \infty$. По мере накопления большего количества данных целесообразно постепенно увеличивать λ, чтобы сократить охват исследования.

Алгоритм 15.5. Стратегия исследования softmax. Она выбирает действие a с вероятностью, пропорциональной $\exp(\lambda \rho_a)$. Параметр точности λ масштабируется с коэффициентом `α` на каждом шаге

```
mutable struct SoftmaxExploration
    λ # параметр точности
    α # коэффициент точности
end
```

---

[5] A. Garivier, T. Lattimore, E. Kaufmann, *On Explore-Then-Commit Strategies*, in Advances in Neural Information Processing Systems (NIPS), 2016.

```
function (π::SoftmaxExploration)(model::BanditModel)
    weights = exp.(π.λ * mean.(model.B))
    π.λ *= π.α
    return rand(Categorical(normalize(weights, 1)))
end
```

Различные стратегии исследования основаны на идее *оптимизма в условиях неопределенности* (optimism under uncertainty). Если мы с оптимизмом смотрим на результаты наших действий в той мере, в какой это позволяют наши статистические данные, мы будем неявно вынуждены балансировать между исследованием и использованием. Одним из таких подходов является *квантильное исследование* (quantile exploration), реализованное в алгоритме 15.6[6], где мы выбираем руку с самым высоким α-*квантилем* (раздел 2.2.2) для вероятности выигрыша. Значения $\alpha > 0.5$ приводят к оптимизму в условиях неопределенности, стимулируя исследование действий, которые не были опробованы достаточно часто. Чем больше значение α, тем больше исследование. В примере 15.3 показана квантильная оценка, которая сравнивается с другими стратегиями исследования.

Алгоритм 15.6. Квантильное исследование, которое возвращает действие с наивысшим квантилем α

```
mutable struct QuantileExploration
    α # квантиль (т. е. 0.95)
end

function (π::QuantileExploration)(model::BanditModel)
    return argmax([quantile(B, π.α) for B in model.B])
end
```

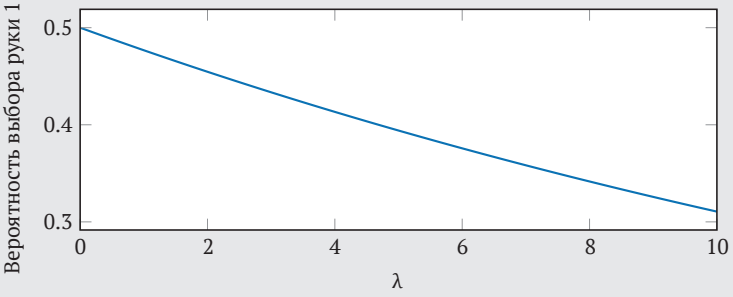
Пример 15.3. Стратегии исследования, использованные в задаче о двуруком бандите из примера 15.1

Рассмотрим использование стратегий исследования с учетом информации, полученной в задаче о двуруком бандите из примера 15.1, где апостериорное распределение для θ_1 – Beta(2, 1), а апостериорное распределение для θ_2 – Beta(5, 2). Вторая рука имеет более высокую вероятность выигрыша.

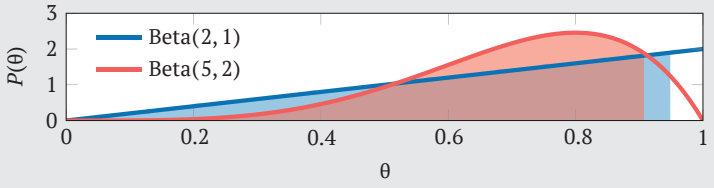
Жадная стратегия с $\varepsilon = 0.2$ обеспечивает шанс 20 % случайного выбора между руками и шанс 80 %, что будет выбрана вторая рука. Следовательно, совокупная вероятность выбора первой руки равна 0.1, а вероятность выбора второй руки равна 0.9.

[6] Эта общая стратегия связана с исследованием верхней доверительной границы, исследованием интервала и оценкой интервала, относящейся к верхней границе доверительного интервала. L. P. Kaelbling, *Learning in Embedded Systems*. MIT Press, 1993. См. также E. Kaufmann, *On Bayesian Index Policies for Sequential Resource Allocation*, Annals of Statistics, vol. 46, no. 2, pp. 842–865, 2018.

Стратегия softmax с $\lambda = 1$ присваивает первой руке вес $\exp(\rho_1) = \exp(2/3) \approx 1.948$, а второй – вес $\exp(\rho_2) = \exp(5/7) \approx 2.043$. Вероятность выбора первой руки равна $1.948/(1.948 + 2.043) \approx 0.488$, а вероятность выбора второй руки – 0.512. На графике ниже показано, как вероятность выбора первой руки зависит от λ:



Квантильное исследование с $\alpha = 0.9$ вычисляет вероятность выигрыша, превышающую 90 % вероятностной меры, связанной с каждым апостериорным распределением. Квантиль 0.9 для θ_1 равен 0.949, а для θ_2 – 0.907, как показано на графике ниже. Первая рука (синяя линия) имеет более высокий квантиль и будет выбрана следующей.



Альтернативой вычислению верхней доверительной границы для нашего апостериорного распределения является использование *исследования* UCB_1 (алгоритм 15.7), первоначально представленного в разделе 9.6 как исследование в поиске по дереву Монте-Карло. Согласно этой стратегии, мы выбираем действие a, дающее наибольшее значение выражения

$$\rho_a + c\sqrt{\frac{\log N}{N(a)}}, \tag{15.3}$$

где $N(a)$ – число раз, когда мы совершили действие a, и $N = \sum_a N(a)$. Параметр $c \geqslant 0$ определяет количество исследований, поощряемых вторым членом уравнения. Большие значения c приводят к увеличению исследования. Эта стратегия часто используется с оценками максимального правдоподобия вероятностей выигрыша, но мы можем адаптировать ее к байесовскому подходу, если $N(a)$ будет суммой параметров бета-распределения, связанных с a.

Алгоритм 15.7. Стратегия исследования UCB_1 с постоянной исследования `c`. Мы вычисляем значение выражения (15.3) для каждого действия из параметров псевдосчетчика в `B`. Затем возвращаем действие, при котором это количество наибольшее

```
mutable struct UCB1Exploration
    c # постоянная исследования
end

function bonus(π::UCB1Exploration, B, a)
    N = sum(b.α + b.β for b in B)
    Na = B[a].α + B[a].β
    return π.c * sqrt(log(N)/Na)
end

function (π::UCB1Exploration)(model::BanditModel)
    B = model.B
    ρ = mean.(B)
    u = ρ .+ [bonus(π, B, a) for a in eachindex(B)]
    return argmax(u)
end
```

Другой общий подход к исследованию заключается в использовании *апостериорной выборки* (posterior sampling), реализованной в алгоритме 15.8, также называемой *рандомизированным сопоставлением вероятностей*, или *выборкой Томпсона*[7]. Этот метод прост в реализации и не требует тщательной настройки параметров. Идея состоит в том, чтобы сделать выборку из апостериорного распределения вознаграждений, связанных с различными действиями. Алгоритм предпочитает действие с наибольшим выборочным значением.

Алгоритм 15.8. Стратегия исследования методом апостериорной выборки. У этого алгоритма нет настраиваемых параметров. Он просто делает выборку из бета-распределений, связанных с каждым действием, а затем возвращает действие, связанное с наибольшей выборкой

```
struct PosteriorSamplingExploration end

(π::PosteriorSamplingExploration)(model::BanditModel) =
    argmax(rand.(model.B))
```

[7] W. R. Thompson, *On the Likelihood That One Unknown Probability Exceeds Another in View of the Evidence of Two Samples*, Biometrika, vol. 25, no. 3/4, pp. 285–294, 1933. Более новый обзор представлен в статье D. Russo, B. V. Roy, A. Kazerouni, I. Osband, Z. Wen, *A Tutorial on Thompson Sampling*, Foundations and Trends in Machine Learning, vol. 11, no. 1, pp. 1–96, 2018.

15.5. Оптимальные стратегии исследования

Бета-распределение, связанное с рукой a, параметризовано подсчетами (w_a, ℓ_a). Вместе эти подсчеты $w_1, \ell_1, \ldots, w_n, \ell_n$ отражают наше доверие текущей модели выигрышей и, таким образом, представляют *доверительное состояние* (belief state). Эти $2n$ чисел могут описывать n непрерывных распределений вероятностей возможных выигрышей.

Мы можем сформулировать задачу в форме MDP, состояния которого представляют собой векторы длины $2n$, отражающие доверие агента в задаче о n-руких бандитах. Для решения этой задачи можно использовать динамическое программирование, чтобы получить оптимальную стратегию π^*, которая указывает, за какую руку дернуть, исходя из имеющихся подсчетов.

Пусть $Q^*(w_1, \ell_1, \ldots, w_n, \ell_n)$ представляет собой ожидаемый выигрыш после рывка за руку a и последующего оптимального действия. Оптимальную функцию полезности и оптимальную стратегию можно выразить через Q^*:

$$U^*(w_1, \ell_1, \ldots, w_n, \ell_n) = \max_a Q^*(w_1, \ell_1, \ldots, w_n, \ell_n, a); \tag{15.4}$$

$$\pi^*(w_1, \ell_1, \ldots, w_n, \ell_n) = \arg\max_a Q^*(w_1, \ell_1, \ldots, w_n, \ell_n, a). \tag{15.5}$$

Разложим Q^* на два члена:

$$\begin{aligned} Q^*(w_1, \ell_1, \ldots, w_n, \ell_n, a) = {} & \frac{w_a + 1}{w_a + \ell_a + 2}(1 + U^*(\ldots, w_a + 1, \ell_a, \ldots)) \\ & + \left(1 + \frac{w_a + 1}{w_a + \ell_a + 2}\right) U^*(\ldots, w_a, \ell_a + 1, \ldots). \end{aligned} \tag{15.6}$$

Первый член соответствует выигрышу с рукой a, а второй – проигрышу. Значение $(w_a + 1)/(w_a + \ell_a + 2)$ представляет собой апостериорную вероятность выигрыша, которая получается из уравнения (15.1)[8]. Аргумент первой U^* в уравнении (15.6) содержит выигрыши, а второй U^* – проигрыши.

Мы можем вычислить Q^* для всего доверительного пространства, поскольку предполагаем конечный горизонт h. Мы начинаем со всех конечных доверительных состояний с $\sum_a(w_a + \ell_a) = h$, где $U^* = 0$. Затем можем перейти к состояниям с $\sum_a(w_a + \ell_a) = h - 1$ и применить уравнение (15.6). Этот процесс повторяется до тех пор, пока мы не достигнем исходного состояния. Такая оптимальная стратегия вычисляется в примере 15.4.

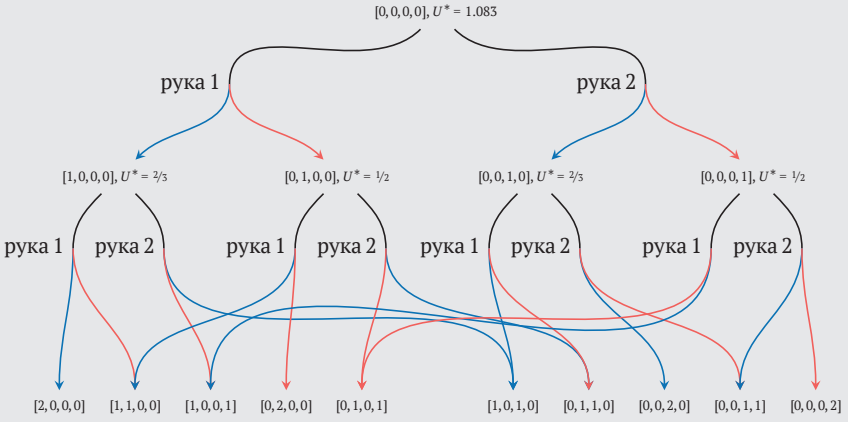
Хотя это решение методом динамического программирования является оптимальным, количество доверительных состояний равно $O(h^{2n})$. Мы можем сформулировать дисконтированную версию задачи с бесконечным горизонтом, которая может быть эффективно решена с использованием *индекса распре-*

[8] Эту вероятность можно уточнить, если известно неравномерное априорное распределение.

деления Гиттинса (Gittins allocation index)[9], представленного в виде переводной таблицы, задающей скалярное значение индекса распределения с учетом количества рывков и количества выигрышей, относящихся к руке[10]. Следующей необходимо дернуть руку, имеющую самый высокий индекс распределения.

Пример 15.4. Вычисление оптимальной стратегии для задачи о двуруких двухшаговых бандитах

Мы построили дерево состояния-действия для задачи о двуруком бандите с горизонтом в два шага. Векторы состояния показаны как $[w_1, \ell_1, w_2, \ell_2]$; синие стрелки обозначают выигрыши, а красные стрелки – проигрыши.



Неудивительно, что стратегия симметрична по отношению к рукам 1 и 2. Мы обнаружили, что не имеет значения, с какой руки начинать, и лучше всего дважды дергать выигравшую руку и не дергать дважды проигравшую.

Оптимальные функции полезности рассчитывались с использованием уравнений

$$Q^*([1,0,0,0],1) = \frac{2}{3}(1+0) + \frac{1}{3}(0) = 2/3;$$

$$Q^*([1,0,0,0],2) = \frac{1}{2}(1+0) + \frac{1}{2}(0) = 1/2;$$

$$Q^*([0,1,0,0],1) = \frac{1}{3}(1+0) + \frac{2}{3}(0) = 1/3;$$

[9] J. C. Gittins, *Bandit Processes and Dynamic Allocation Indices*, Journal of the Royal Statistical Society. Series B (Methodological), vol. 41, no. 2, pp. 148–177, 1979. J. Gittins, K. Glazebrook, R. Weber, *Multi-Armed Bandit Allocation Indices*, 2nd ed. Wiley, 2011.

[10] Обзор алгоритмов для вычисления этой таблицы поиска представлен в J. Chakravorty and A. Mahajan, *Multi-Armed Bandits, Gittins Index, and Its Calculation*, in Methods and Applications of Statistics in Clinical Trials, N. Balakrishnan, ed., vol. 2, Wiley, 2014, pp. 416–435.

$$Q^*([0,1,0,0],1) = \frac{1}{2}(1+0) + \frac{1}{2}(0) = 1/2;$$
$$Q^*([0,0,0,0],1) = \frac{1}{2}(1+2/3) + \frac{1}{2}(1/2) = 1.083.$$

15.6. Исследование с несколькими состояниями

В общем контексте обучения с подкреплением с несколькими состояниями мы должны использовать наблюдения за переходами состояний для обоснования наших решений. Мы можем изменить процесс моделирования в алгоритме 15.1, чтобы учесть переходы между состояниями и соответствующим образом обновить нашу модель. Обновленный вариант показан в алгоритме 15.9. Есть много способов смоделировать задачу и выполнить исследование, о чем мы поговорим в следующих нескольких главах, но структура модели в целом такая же.

Алгоритм 15.9. Цикл моделирования для задач обучения с подкреплением. Стратегия исследования π генерирует следующее действие на основе информации, содержащейся в модели, и текущего состояния `s`. Задача MDP $\mathcal{P}$ рассматривается как эталонная и используется для выборки следующего состояния и вознаграждения. Переход состояния и вознаграждение используются для обновления модели. Моделирование проводится до горизонта `h`

```
function simulate(𝒫::MDP, model, π, h, s)
    for i in 1:h
        a = π(model, s)
        s′, r = 𝒫.TR(s, a)
        update!(model, s, a, r, s′)
        s = s′
    end
end
```

15.7. Заключение

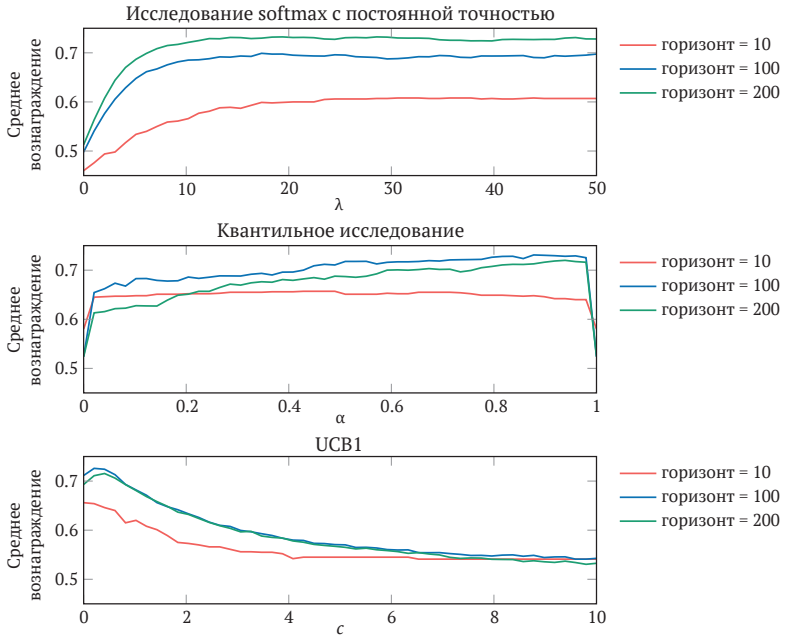
- Компромисс между исследованием и использованием представляет собой оптимальный баланс между исследованием пространства состояний-действий для получения более высоких вознаграждений и использованием уже известных благоприятных действий из текущего состояния.
- Задачи о многоруких бандитах содержат одно состояние, в котором агент получает стохастическое вознаграждение за различные действия.

- Для формирования убеждения о выигрыше в задачах о многоруких бандитах можно использовать бета-распределения.
- Стратегии ненаправленного исследования, в том числе ε-жадная и «исследуй, затем сделай», просты в реализации, но не используют информацию из предыдущих результатов при исследовании нежадных действий.
- Стратегии направленного исследования, в том числе softmax, квантили, UCB_1 и исследование апостериорной выборки, используют информацию о прошлых действиях для лучшего исследования перспективных действий.
- Оптимальные стратегии исследования для конечных горизонтов можно получить с помощью динамического программирования, но эти стратегии могут быть вычислительно дорогими.

15.8. Упражнения

Упражнение 15.1. Пусть у нас есть трехрукий бандит, у которого вероятность выигрыша каждой руки равномерно распределена между 0 и 1. Сравните стратегии исследования softmax, квантилей и UCB_1. В качественном смысле, какие значения λ, α и *c* дают наибольшее ожидаемое вознаграждение в случайно сгенерированных задачах о трехруком бандите?

Решение. Построим графики ожидаемого вознаграждения за шаг для каждой из трех стратегий. Эффективность параметризации зависит от горизонта задачи, поэтому на графике также показано несколько разных глубин.



Стратегия softmax работает лучше всего при больших значениях λ, которые отдают приоритет рукам с более высоким ожидаемым вознаграждением в соответствии с текущим убеждением о выигрыше. Исследование с верхней доверительной границей работает лучше с более отдаленными горизонтами, независимо от параметризации. Размер доверительной границы α не оказывает существенного влияния на производительность, за исключением значений, очень близких к 0 или 1. Стратегия UCB_1 работает лучше всего при небольших положительных значениях скаляра исследования *c*. Ожидаемое вознаграждение уменьшается по мере увеличения *c*. Все три стратегии могут быть настроены для получения одинакового максимального ожидаемого вознаграждения.

Упражнение 15.2. Приведите пример практического применения задачи о многоруком бандите

Решение. На практике можно часто встретить задачи, относящиеся к классу задач о многоруком бандите. Рассмотрим, например, новостную компанию, которая хотела бы максимизировать взаимодействие с читателями (клики по статьям) на своем веб-сайте. У компании может быть несколько статей для отображения на главной странице, но она должна выбрать только одну статью для отображения в любой момент времени. Это типичная задача о многоруком бандите, потому что пользователь либо нажмет на *i*-ю статью с вероятностью θ_i, либо не нажмет на нее с вероятностью $1 - \theta_i$. Исследование заключается в отображении статей на веб-сайте и наблюдении за количеством кликов, а использование – в отображении статьи, которая может привести к наибольшему количеству кликов. Эта задача напрямую связана с *A/B-тестированием*, когда компании тестируют разные версии веб-сайта, чтобы определить, какая версия обеспечивает наибольшее количество взаимодействий.

Упражнение 15.3. Для однорукого бандита с априорным распределением $\theta \sim \text{Beta}(7, 2)$ укажите границы апостериорной вероятности выигрыша после 10 дополнительных попыток.

Решение. Нижнюю границу нашей апостериорной вероятности выигрыша $\underline{\rho}$ можно вычислить, если предположить, что все попытки приводят к проигрышу (например, $\underline{\ell} = 10$ и $\underline{w} = 0$). Аналогичным образом мы можем вычислить верхнюю границу $\bar{\rho}$, предполагая, что все попытки приводят к выигрышу (например, $\bar{w} = 10$ и $\bar{\ell} = 0$). Таким образом, границы будут следующими:

$$\underline{\rho} = \frac{\underline{w}+7}{\underline{w}+\underline{\ell}+9} = \frac{0+7}{0+10+9} = \frac{7}{19};$$
$$\bar{\rho} = \frac{\bar{w}+7}{\bar{w}+\bar{\ell}+9} = \frac{10+7}{10+0+9} = \frac{17}{19}.$$

Упражнение 15.4. Предположим, что у нас есть бандит с руками *a* и *b* и мы используем ε-жадную стратегию исследования с $\varepsilon = 0.3$ и коэффициентом за-

тухания исследования $\alpha = 0.9$. Мы генерируем случайное число x от 0 до 1, от которого зависит, занимаемся ли мы на текущем шаге исследованием ($x < \varepsilon$) или использованием ($x > \varepsilon$). Если известно, что у нас $\rho_a > \rho_b$, какую руку следует выбрать при $x = 0.2914$ на первой итерации? Какую руку следует выбрать, если $x = 0.1773$ на девятой итерации?

Решение. Поскольку $x < \varepsilon_1$ на первой итерации, мы выполняем исследование и выбираем a и b с вероятностью 0.5. На девятой итерации $\varepsilon_9 = \alpha^8\varepsilon_1 \approx 0.129$. Поскольку $x > \varepsilon_9$, мы используем собранную информацию и выбираем a.

Упражнение 15.5. Допустим, у нас есть четырехрукий бандит и мы собираемся использовать стратегию исследования softmax с параметром точности $\lambda = 2$ и априорным убеждением $\theta_a \sim \text{Beta}(2, 2)$ для каждой руки a. Предположим, что мы тянули каждую руку четыре раза, в результате чего руки 1, 2, 3 и 4 принесли выигрыш 1, 2, 3 и 4 раза соответственно. Перечислите апостериорные распределения по θ_a и рассчитайте вероятность того, что мы выберем руку 2.

Решение. Апостериорные распределения для рук 1, 2, 3 и 4 составляют Beta(3, 5), Beta(4, 4), Beta(5, 3) и Beta(6, 2) соответственно. Вероятность выбора руки 2 можно вычислить, выполнив следующие шаги:

$$P(a = i) \propto \exp(\lambda\rho_i);$$

$$P(a = i) = \frac{\exp(\lambda\rho_i)}{\sum_a \exp(\lambda\rho_a)};$$

$$P(a = 2) = \frac{\exp\left(2 \times \frac{4}{8}\right)}{\exp\left(2 \times \frac{3}{8}\right) + \exp\left(2 \times \frac{4}{8}\right) + \exp\left(2 \times \frac{5}{8}\right) + \exp\left(2 \times \frac{6}{8}\right)};$$

$$P(a = 2) \approx 0.2122.$$

Упражнение 15.6. Перепишите уравнение (15.6) для произвольного априорного значения Beta(α, β).

Решение. Мы можем переписать уравнение в более общем виде следующим образом:

$$Q^*(w_1, \ell_1, \ldots, w_n, \ell_n, a) = \frac{w_a + \alpha}{w_a + \ell_a + \alpha + \beta}(1 + U^*(\ldots, w_a + 1, \ell_a, \ldots)) + \left(1 - \frac{w_a + \alpha}{w_a + \ell_a + \alpha + \beta}\right)U^*(\ldots, w_a, \ell_a + 1, \ldots).$$

Упражнение 15.7. Вернемся к примеру 15.4. Вместо одинакового выигрыша величиной 1 для каждой руки давайте предположим, что рука 1 дает выигрыш величиной 1, а рука 2 дает выигрыш величиной 2. Рассчитайте новые значения функции полезности действия для обеих рук.

Решение. Для руки 1 имеем полезность действия:

$$Q^*([1,0,0,0],1) = \frac{2}{3}(1+0) + \frac{1}{3}(0) = 2/3;$$
$$Q^*([1,0,0,0],2) = \frac{1}{2}(2+0) + \frac{1}{2}(0) = 1;$$
$$Q^*([0,1,0,0],1) = \frac{1}{3}(1+0) + \frac{2}{3}(0) = 1/3;$$
$$Q^*([0,1,0,0],2) = \frac{1}{2}(2+0) + \frac{1}{2}(0) = 1;$$
$$Q^*([0,0,0,0],1) = \frac{1}{2}(1+1) + \frac{1}{2}(1) = 1.5.$$

Для руки 2 имеем полезность действия:

$$Q^*([0,0,1,0],1) = \frac{1}{2}(1+0) + \frac{1}{2}(0) = 1/2;$$
$$Q^*([0,0,1,0],2) = \frac{2}{3}(2+0) + \frac{1}{3}(0) = 4/3;$$
$$Q^*([0,0,0,0],1) = \frac{1}{2}(1+0) + \frac{1}{2}(0) = 1/2;$$
$$Q^*([0,0,0,1],2) = \frac{1}{3}(2+0) + \frac{2}{3}(0) = 2/3;$$
$$Q^*([0,0,0,0],2) = \frac{1}{2}(2+4/3) + \frac{1}{2}(2/3) = 2.$$

Упражнение 15.8. Докажите, что количество доверительных состояний в задаче об n-руких бандитах с горизонтом h равно $O(h^{2n})$.

Решение. Начнем с подсчета количества решений $w_1 + \ell_1 + \cdots + w_n + \ell_n = k$, где $0 \leqslant k \leqslant h$. Если $n = 2$ и $k = 6$, одно решение равно 2 + 0 + 3 + 1 = 6. Для представления целых чисел при подсчете мы будем использовать метки. Например, мы можем записать решение в виде 2 + 0 + 3 + 1 = ||++|||+| = 6. В общем случае для значений n и k у нас было бы k меток и $2n - 1$ знаков плюса. Мы можем представить решение в виде строки из $k + 2n - 1$ символов, где символ – это | или + и где k из этих символов равно |. Чтобы узнать количество решений, подсчитаем количество способов, которыми мы можем выбрать k позиций для | из множества $k + 2n - 1$ позиций, в результате чего получается

$$\frac{(k+2n-1)!}{(2n-1)!\,k!} = O(h^{2n-1})$$

решений. Количество доверительных состояний – это сумма значений выражения выше для k от 0 до h, что равно $O(h \times h^{2n-1}) = O(h^{2n})$.

16 Методы на основе моделей

В этой главе обсуждаются как методы максимального правдоподобия, так и байесовские подходы к изучению динамики и вознаграждения за счет взаимодействия с окружающей средой. Методы максимального правдоподобия основаны на подсчете переходов между состояниями и записи суммы полученного вознаграждения для оценки параметров модели. Мы обсудим несколько подходов к планированию с использованием непрерывно обновляющихся моделей. В процессе поиска обычно приходится полагаться на эвристические стратегии исследования, даже если удается найти точное решение задачи. Байесовские методы основаны на вычислении апостериорного распределения по параметрам модели. Поиск оптимальной стратегии исследования, как правило, затруднен, но мы часто можем получить разумное приближение с помощью апостериорной выборки.

16.1. Модели максимального правдоподобия

В разделе 15.6 было сказано о том, что при обучении с подкреплением для принятия обоснованных решений применяется информация о предыдущих вознаграждениях и переходах между состояниями. Этот подход реализован в алгоритме 15.9. Здесь мы рассмотрим, как получить *оценку по методу максимального правдоподобия* (maximum likelihood estimate) для основной задачи. Затем из этой оценки можно найти значение функции ценности, которое вместе со стратегией исследования применяется для создания действий.

Запишем количество переходов $N(s, a, s')$, указывающее, сколько раз наблюдался переход от s к s' при выполнении действия a. Приближенное уравнение для оценки функции перехода методом максимального правдоподобия с учетом переходов выглядит так:

$$T(s'|s, a) \approx N(s, a, s')/N(s, a), \tag{16.1}$$

где $N(s, a) = \sum_{s'} N(s, a, s')$. Если $N(s, a) = 0$, то расчетная вероятность перехода равна 0.

Аналогичным образом оценивают функцию вознаграждения. По мере получения вознаграждения обновляют сумму всех вознаграждений $\rho(s, a)$, полученных при выполнении действия a в состоянии s. Оценка функции вознаграждения методом максимального правдоподобия – это среднее вознаграждение:

$$R(s, a) \approx \rho(s, a)/N(s, a). \tag{16.2}$$

Если $N(s, a) = 0$, то оценка $R(s, a)$ равна 0. Если же известны априорные вероятности перехода или вознаграждения, то имеет смысл инициализировать $N(s, a, s')$ и $\rho(s, a)$ значениями, отличными от 0.

Алгоритм 16.1 обновляет N и ρ после наблюдения за переходом от s к s' вследствие выполнения действия a и получения вознаграждения r. Алгоритм 16.2 преобразует модель максимального правдоподобия в представление MDP. Пример 16.1 иллюстрирует этот процесс. Модель максимального правдоподобия применяется для выбора действий в ходе взаимодействия с окружающей средой с одновременным улучшением самой модели.

Алгоритм 16.1. Реализация метода обновления модели переходов и вознаграждений путем обучения с подкреплением по методу максимального правдоподобия в дискретных пространствах состояний и действий. Мы увеличиваем `N[s, a, s′]`, наблюдая за переходом от `s` к `s′` после выполнения действия `a`, и прибавляем `r` к `ρ[s,a]`. Модель также содержит оценку функции ценности `U` и планировщик. Этот блок кода еще включает методы для выполнения оператора Беллмана и предсказания следующего шага по отношению к этой модели

```
mutable struct MaximumLikelihoodMDP
    𝒮  # пространство состояний (пусть 1:nstates)
    𝒜  # пространство действий (пусть 1:nactions)
    N  # счетчик переходов N(s,a,s′)
    ρ  # суммарное вознаграждение ρ(s, a)
    γ  # дисконт
    U  # функция ценности
    planner
end

function lookahead(model::MaximumLikelihoodMDP, s, a)
    𝒮, U, γ = model.𝒮, model.U, model.γ
    n = sum(model.N[s,a,:])
    if n == 0
        return 0.0
    end
    r = model.ρ[s, a] / n
    T(s,a,s′) = model.N[s,a,s′] / n
    return r + γ * sum(T(s,a,s′)*U[s′] for s′ in 𝒮)
end

function backup(model::MaximumLikelihoodMDP, U, s)
    return maximum(lookahead(model, s, a) for a in model.𝒜)
end

function update!(model::MaximumLikelihoodMDP, s, a, r, s′)
    model.N[s,a,s′] += 1
    model.ρ[s,a] += r
```

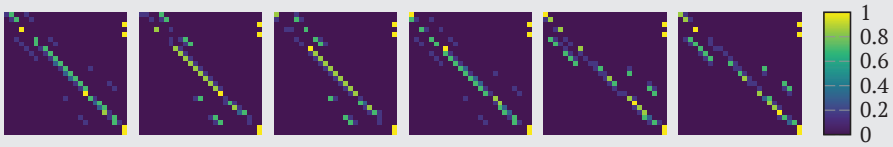
```
    update!(model.planner, model, s, a, r, s′)
    return model
end
```

Алгоритм 16.2. Метод преобразования модели максимального правдоподобия в задачу MDP

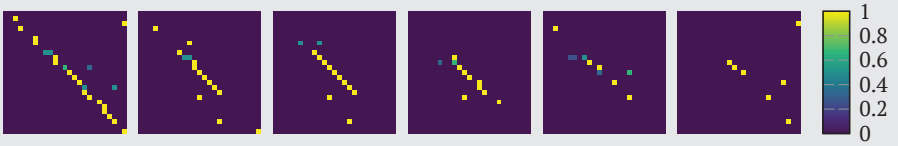
```
function MDP(model::MaximumLikelihoodMDP)
    N, ρ, 𝒮, 𝒜, γ = model.N, model.ρ, model.𝒮, model.𝒜, model.γ
    T, R = similar(N), similar(ρ)
    for s in 𝒮
        for a in 𝒜
            n = sum(N[s,a,:])
            if n == 0
                T[s,a,:] .= 0.0
                R[s,a] = 0.0
            else
                T[s,a,:] = N[s,a,:] / n
                R[s,a] = ρ[s,a] / n
            end
        end
    end
    return MDP(T, R, γ)
end
```

Пример 16.1. Применение оценки максимального правдоподобия к задаче гексамира

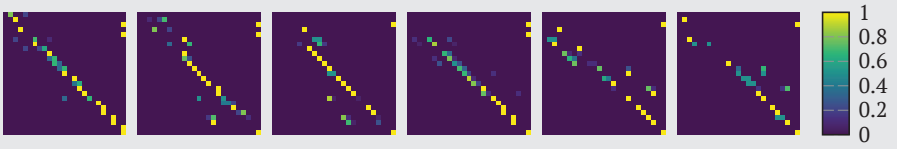
Применим оценку модели по методу максимального правдоподобия в задаче гексамира. Истинные матрицы перехода выглядят так:



Имеется шесть матриц переходов, по одной для каждого действия. Строки соответствуют текущему состоянию, а столбцы – следующему состоянию. Есть 26 состояний. Температура цвета отражает вероятность совершения соответствующего перехода. В контексте обучения с подкреплением мы не знаем эти вероятности перехода заранее. Однако мы можем взаимодействовать с окружающей средой и записывать переходы, которые наблюдаем. После 10 прогонов модели по 10 шагов в каждом из случайных начальных состояний оценка по методу максимального правдоподобия дает нам следующие матрицы:



После 1000 прогонов оценка приобретает вид:



16.2. Схемы обновления модели

Обновляя оценку максимального правдоподобия модели, нам также необходимо обновлять план. В этом разделе рассматривается несколько схем обновления, применяемых к постоянно меняющейся модели. Ключевым критерием является вычислительная эффективность, потому что в процессе взаимодействия со средой эти обновления приходится выполнять довольно часто.

16.2.1. Полное обновление

Алгоритм 16.3 обновляет модель по методу максимального правдоподобия, используя формулу линейного программирования из раздела 7.7, хотя можно также использовать итерацию значений или какой-либо другой алгоритм. После каждого шага мы получаем новую оценку и заново строим модель.

Алгоритм 16.3. Метод, который выполняет полное обновление функции ценности `U`, используя формулу линейного программирования из раздела 7.7

```
struct FullUpdate end

function update!(planner::FullUpdate, model, s, a, r, s′)
    𝒫 = MDP(model)
    U = solve(𝒫).U
    copy!(model.U, U)
    return planner
end
```

16.2.2. Рандомизированное обновление

Повторное вычисление оптимальной стратегии после каждого перехода состояния обычно потребляет значительные вычислительные ресурсы. Альтернативой является выполнение обновления Беллмана для модели в ранее посещенном состоянии, а также в нескольких случайно выбранных состояниях[1]. Этот прием реализован в алгоритме 16.4.

Алгоритм 16.4. Обучение с подкреплением на основе модели максимального правдоподобия с обновлениями в случайно выбранных состояниях. Этот алгоритм выполняет обновление Беллмана в ранее посещенном состоянии, а также в `m` дополнительных состояниях, выбранных случайным образом

```
struct RandomizedUpdate
    m # количество обновлений
end

function update!(planner::RandomizedUpdate, model, s, a, r, s′)
    U = model.U
    U[s] = backup(model, U, s)
    for i in 1:planner.m
        s = rand(model.𝒮)
        U[s] = backup(model, U, s)
    end
    return planner
end
```

16.2.3. Приоритетный механизм обновления

Подход, называемый *приоритетным выметанием* (prioritized sweeping)[2] и реализованный в алгоритме 16.5, использует приоритетную очередь для определения того, какие состояния больше всего нуждаются в обновлении. За переходом от s к s' следует обновление $U(s)$ на основе наших обновленных моделей перехода и вознаграждения. Затем алгоритм перебирает все пары состояние-действие (s^-, a^-), которые могут сразу перейти в s. Приоритет любого такого состояния s^- увеличивается до $T(s \mid s^-, a^-) \times |U(s) - u|$, где u было значением функции $U(s)$ до обновления. Следовательно, чем больше изменение $U(s)$ и чем более вероятен переход к s, тем выше приоритет состояний, ведущих к s. Процесс

---

[1] Этот подход связан с архитектурой Dyna, предложенной в статье R. S. Sutton, *Dyna, an Integrated Architecture for Learning, Planning, and Reacting*, SIGART Bulletin, vol. 2, no. 4, pp. 160–163, 1991.

[2] A. W. Moore, C. G. Atkeson, *Prioritized Sweeping: Reinforcement Learning with Less Data and Less Time*, Machine Learning, vol. 13, no. 1, pp. 103–130, 1993.

обновления состояния с наивысшим приоритетом в очереди продолжается в течение фиксированного числа итераций или до тех пор, пока очередь не опустеет.

Алгоритм 16.5. Алгоритм приоритетного выметания использует приоритетную очередь состояний `pq`, которая определяет, какие из них должны быть обновлены. С каждым обновлением мы назначаем предыдущему состоянию бесконечный приоритет. Затем мы выполняем `m` обновлений Беллмана функции ценности `U` для состояний с наивысшим приоритетом

```
struct PrioritizedUpdate
    m  # количество обновлений
    pq # очередь приоритетов
end

function update!(planner::PrioritizedUpdate, model, s)
    N, U, pq = model.N, model.U, planner.pq
    𝒮, 𝒜 = model.𝒮, model.𝒜
    u = U[s]
    U[s] = backup(model, U, s)
    for s⁻ in 𝒮
        for a⁻ in 𝒜
            n_sa = sum(N[s⁻,a⁻,s′] for s′ in 𝒮)
            if n_sa > 0
                T = N[s⁻,a⁻,s] / n_sa
                priority = T * abs(U[s] - u)
                if priority > 0
                    pq[s⁻] = max(get(pq, s⁻, 0.0), priority)
                end
            end
        end
    end
    return planner
end

function update!(planner::PrioritizedUpdate, model, s, a, r, s′)
    planner.pq[s] = Inf
    for i in 1:planner.m
        if isempty(planner.pq)
            break
        end
        update!(planner, model, dequeue!(planner.pq))
    end
    return planner
end
```

16.3. Исследование

Независимо от схемы обновления, обычно необходимо придерживаться определенной стратегии исследования, чтобы избежать ловушек «чистого использования», упомянутых в предыдущей главе. Мы можем адаптировать алгоритмы исследования, представленные в этой главе, для использования в задачах с несколькими состояниями. Алгоритм 16.6 реализует стратегию ε-жадного исследования.

Алгоритм 16.6. Стратегия ε-жадного исследования для оценивания модели по методу максимального правдоподобия. Она выбирает случайное действие с вероятностью ε; в противном случае она использует модель для получения жадного действия

```
function (π::EpsilonGreedyExploration)(model, s)
    𝒜, ε = model.𝒜, π.ε
    if rand() < ε
        return rand(𝒜)
    end
    Q(s,a) = lookahead(model, s, a)
    return argmax(a->Q(s,a), 𝒜)
end
```

Ограничение стратегий исследования, рассмотренных в главе 15, состоит в том, что они не рассуждают об исследовании действий из других состояний, кроме текущего. Например, нам может потребоваться предпринять действия, которые приведут нас в неисследованную область пространства состояний. Для решения этой проблемы было предложено несколько алгоритмов, которые также обеспечивают вероятностные оценки качества результирующей стратегии после конечного числа взаимодействий[3].

Один из таких алгоритмов известен под названием *R-MAX* (алгоритм 16.7)[4]. Его название отражает присвоение максимального вознаграждения малоизученным парам состояние-действие. Малоизученными считаются пары состояние-действие с числом посещений менее m. Вместо оценки вознаграждения по методу максимального правдоподобия (уравнение 16.2) используют следующее правило:

---

[3] M. Kearns, S. Singh, *Near-Optimal Reinforcement Learning in Polynomial Time*, Machine Learning, vol. 49, no. 2/3, pp. 209–232, 2002.

[4] R. I. Brafman, M. Tennenholtz, *R-MAX–A General Polynomial Time Algorithm for Near-Optimal Reinforcement Learning*, Journal of Machine Learning Research, vol. 3, pp. 213–231, 2002.

$$R(s,a) = \begin{cases} r_{\max}, & \text{если } N(s,a) < m \\ \rho(s,a)/N(s,a) & \text{в ином случае} \end{cases}, \tag{16.3}$$

где $r_{\max}$ – максимально достижимое вознаграждение.

Модель перехода в R-MAX изменена таким образом, что малоизученные пары состояние-действие остаются в прежнем состоянии:

$$T(s'|s,a) = \begin{cases} (s' = s), & \text{если } N(s,a) < m \\ N(s,a,s')/N(s,a) & \text{в ином случае} \end{cases}. \tag{16.4}$$

Следовательно, малоизученные состояния имеют ценность $r_{\max}/(1 - \gamma)$, что служит стимулом для их исследования. Этот стимул к исследованию избавляет нас от необходимости в отдельном механизме исследования. Мы просто жадно выбираем наши действия в соответствии с функцией ценности, полученной из наших оценок перехода и вознаграждения. В примере 16.2 сопоставлены ε-жадное исследование и исследование R-MAX.

Алгоритм 16.7. Стратегия исследования R-MAX изменяет модель перехода и вознаграждения, исходя из оценки по методу максимального правдоподобия. Она присваивает максимальное вознаграждение `rmax` любой неисследованной паре состояние-действие, то есть такой, которая была опробована менее `m` раз. Кроме того, все неисследованные пары состояние-действие моделируются как переходы в то же самое состояние. Стратегию `RmaxMDP` можно использовать в качестве замены `MaximumLikelihoodMDP`, реализованной в алгоритме 16.1

```
mutable struct RmaxMDP
    𝒮     # пространство состояний (пусть 1:nstates)
    𝒜     # пространство действий (пусть 1:nactions)
    N     # счетчик переходов N(s,a,s′)
    ρ     # суммарное вознаграждение ρ(s, a)
    γ     # дисконт
    U     # функция ценности
    planner
    m     # порог счетчика
    rmax  # максимальное вознаграждение
end

function lookahead(model::RmaxMDP, s, a)
    𝒮, U, γ = model.𝒮, model.U, model.γ
    n = sum(model.N[s,a,:])
    if n < model.m
        return model.rmax / (1-γ)
    end
    r = model.ρ[s, a] / n
    T(s,a,s′) = model.N[s,a,s′] / n
    return r + γ * sum(T(s,a,s′)*U[s′] for s′ in 𝒮)
end
```

```
function backup(model::RmaxMDP, U, s)
    return maximum(lookahead(model, s, a) for a in model.𝒜)
end

function update!(model::RmaxMDP, s, a, r, s′)
    model.N[s,a,s′] += 1
    model.ρ[s,a] += r
    update!(model.planner, model, s, a, r, s′)
    return model
end

function MDP(model::RmaxMDP)
    N, ρ, 𝒮, 𝒜, γ = model.N, model.ρ, model.𝒮, model.𝒜, model.γ
    T, R, m, rmax = similar(N), similar(ρ), model.m, model.rmax
    for s in 𝒮
        for a in 𝒜
            n = sum(N[s,a,:])
            if n < m
                T[s,a,:] .= 0.0
                T[s,a,s] = 1.0
                R[s,a] = rmax
            else
                T[s,a,:] = N[s,a,:] / n
                R[s,a] = ρ[s,a] / n
            end
        end
    end
    return MDP(T, R, γ)
end
```

Пример 16.2. Демонстрация стратегий ε-жадного исследования и R-MAX

Сначала применим стратегию ε-жадного исследования к оценкам модели по методу максимального правдоподобия, построенным во время взаимодействия с окружающей средой. Следующий код инициализирует нулевыми значениями подсчеты, вознаграждения и полезности. Он использует полные обновления функции ценности на каждом шаге. Для исследования мы выбираем случайное действие с вероятностью 0.1. Последняя строка запускает моделирование (алгоритм 15.9) задачи $\mathcal{P}$ на 100 шагов, начиная со случайного начального состояния:

```
N = zeros(length(𝒮), length(𝒜), length(𝒮))
ρ = zeros(length(𝒮), length(𝒜))
U = zeros(length(𝒮))
planner = FullUpdate()
model = MaximumLikelihoodMDP(𝒮, 𝒜, N, ρ, γ, U, planner)
π = EpsilonGreedyExploration(0.1)
simulate(𝒫, model, π, 100, rand(𝒮))
```

В качестве альтернативы воспользуемся алгоритмом R-MAX с порогом исследования $m = 3$. В случае оценки модели по алгоритму R-MAX тоже можно применить жадное исследование:

```
rmax = maximum(𝒫.R(s,a) for s in 𝒮, a in 𝒜)
m = 3
model = RmaxMDP(𝒮, 𝒜, N, ρ, γ, U, planner, m, rmax)
π = EpsilonGreedyExploration(0)
simulate(𝒫, model, π, 100, rand(𝒮))
```

16.4. Байесовские методы

В отличие от рассмотренных выше методов максимального правдоподобия, байесовские методы находят компромисс между исследованием и использованием, не полагаясь на эвристические стратегии исследования. Далее мы перейдем к обобщению байесовских методов, рассмотренных в разделе 15.5. В *байесовском обучении с подкреплением* (Bayesian reinforcement learning) указывают априорное распределение по всем параметрам модели θ[5]. В их число могут входить параметры, определяющие распределение вероятностей *непосредственных подкреплений* (immediate reward), но в этом разделе основное внимание уделяется параметрам, определяющим вероятности перехода состояний.

Структура задачи может быть представлена с помощью динамической сети принятия решений, показанной на рис. 16.1, в которой параметры модели

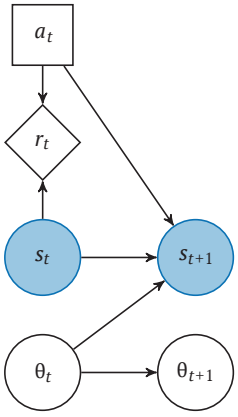


Рис. 16.1. Динамическая сеть принятия решений для задачи MDP с неопределенностью модели

[5] Обзор по этой теме предоставлен в статье M. Ghavamzadeh, S. Mannor, J. Pineau, A. Tamar, *Bayesian Reinforcement Learning: A Survey*, Foundations and Trends in Machine Learning, vol. 8, no. 5–6, pp. 359–483, 2015. В ней рассматриваются методы включения априорных распределений в функции вознаграждения, которые здесь не обсуждаются.

сделаны явными. Закрашенные узлы указывают на то, что состояния наблюдаются, а параметры модели – нет. Обычно мы предполагаем, что параметры модели не зависят от времени: $\theta_{t+1} = \theta_t$. Однако наше первоначальное убеждение относительно θ меняется со временем, когда система переходит в новые состояния.

Начальное убеждение о вероятности перехода (belief of probability transition) может быть представлено с помощью набора распределений Дирихле, по одному для каждого исходного состояния и действия. Каждое распределение Дирихле представляет собой распределение по s' для заданных s и a. Если $\boldsymbol{\theta}_{(s,a)}$ является $|\mathcal{S}|$-элементным вектором, представляющим распределение в пределах следующего состояния, то априорное распределение задается выражением

$$\text{Dir}(\boldsymbol{\theta}_{(s,a)}|\mathbf{N}(s, a)), \tag{16.5}$$

где $\mathbf{N}(s, a)$ – вектор подсчетов, связанных с переходами, начинающимися в состоянии s и выполняющими действие a. Обычно используют равномерное априорное распределение, где всем компонентам присвоено значение 1, но подсчеты могут быть инициализированы иначе, исходя из априорного знания динамики перехода. В примере 16.3 показано, как эти подсчеты применяются в распределении Дирихле для представления распределения по возможным вероятностям перехода.

Распределение по θ получается в результате перемножения распределений Дирихле:

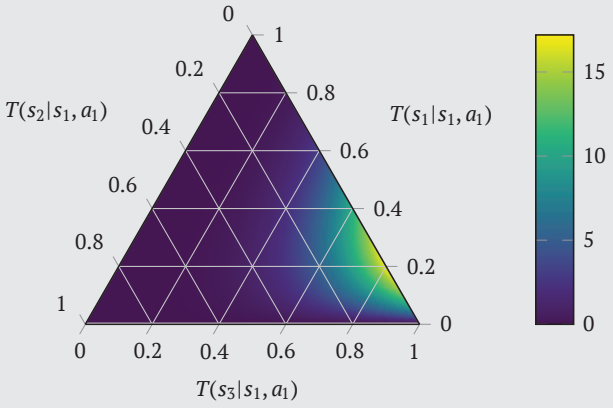
$$b(\pi) = \prod_s \prod_a \text{Dir}(\boldsymbol{\theta}_{(s,a)}|\mathbf{N}(s,a)). \tag{16.6}$$

В алгоритме 16.8 представлена реализация байесовского обновления для этого типа апостериорной модели. Для задач с большими или непрерывными пространствами можно использовать другие апостериорные представления.

Пример 16.3. Апостериорное распределение Дирихле по вероятностям перехода из определенного состояния при выполнении определенного действия. Агент, изучающий функцию перехода в неизвестном MDP, может сохранить такое распределение для каждой пары состояние-действие

Предположим, наш агент случайным образом исследует среду с тремя состояниями. Агент выполняет действие a_1 из состояния s_1 пять раз. Он переходит в s_3 четыре раза и один раз остается в s_1. Подсчеты, связанные с s_1 и a_1, равны $\mathbf{N}(s_1, a_1) = [1, 0, 4]$. Если мы хотим рассматривать равномерное априорное распределение по результирующим состояниям, то должны увеличить подсчеты на 1, чтобы получить $\mathbf{N}(s_1, a_1) = [2, 1, 5]$. Функция перехода от s_1 вследствие действия a_1 является категориальным распределением трех значений, поскольку существует три возможных последующих состояния. Каждое последующее состояние имеет неизвестную вероятность перехода.

Пространство возможных вероятностей перехода представляет собой набор трехэлементных векторов, сумма которых равна 1. Распределение Дирихле представляет собой распределение по этим возможным вероятностям перехода. На рисунке ниже показано, как выглядит график плотности распределения вероятности:



Алгоритм 16.8. Байесовский метод обновления, согласно которому апостериорное распределение по моделям переходов представляется как произведение распределений Дирихле. В этой реализации алгоритма предполагается, что модель вознаграждения `R` известна, хотя можно также использовать байесовские методы для оценки ожидаемого вознаграждения на основе опыта. Матрица `D` связывает распределения Дирихле с каждой парой состояние-действие для моделирования неопределенности при переходе к их последующим состояниям

```
mutable struct BayesianMDP
    𝒮  # пространство состояний (пусть 1:nstates)
    𝒜  # пространство действий (пусть 1:nactions)
    D  # распределение Дирихле D[s,a]
    R  # матричная функция вознаграждения (не вычислена)
    γ  # дисконт
    U  # функция ценности
    planner
end

function lookahead(model::BayesianMDP, s, a)
    𝒮, U, γ = model.𝒮, model.U, model.γ
    n = sum(model.D[s,a].alpha)
    if n == 0
        return 0.0
    end
    r = model.R(s,a)
```

```
    T(s,a,s′) = model.D[s,a].alpha[s′] / n
    return r + γ * sum(T(s,a,s′)*U[s′] for s′ in 𝒮)
end

function update!(model::BayesianMDP, s, a, r, s′)
    α = model.D[s,a].alpha
    α[s′] += 1
    model.D[s,a] = Dirichlet(α)
    update!(model.planner, model, s, a, r, s′)
    return model
end
```

16.5. Адаптивные по Байесу марковские процессы принятия решений

Задачу нахождения оптимального действия в MDP с неизвестной моделью можно рассматривать как многомерный MDP с известной моделью. Такой MDP известен как *адаптивный по Байесу марковский процесс принятия решений* (Bayes-adaptive Markov decision process, адаптивный байесов MDP), который связан с частично наблюдаемым марковским процессом принятия решений, рассматриваемым в части IV.

Пространство состояний в адаптивном байесовом MDP представляет собой декартово произведение $\mathcal{S}\times\mathcal{B}$, где B – пространство возможных начальных убеждений относительно параметров модели θ. Несмотря на дискретный характер $\mathcal{S}$, B часто является многомерным непрерывным пространством состояний[6]. Состояние в адаптивном байесовом MDP – это пара (s, b), состоящая из текущего состояния s в базовом MDP и доверительного состояния b. Пространство действия и функция вознаграждения такие же, как и в базовом MDP.

Функция перехода в адаптивном байесовом MDP $T(s', b'|s, b, a)$ представляет собой вероятность перехода в новое состояние s' с доверительным состоянием b', при условии что агент начинает в s с начальным убеждением b и предпринимает действие a. Новое доверительное состояние b' может быть детерминированно вычислено в соответствии с правилом Байеса. Если мы обозначим эту детерминированную функцию как τ, такую, что $b' = \tau(s, b, a, s')$, то мы можем разложить адаптивную по Байесу функцию перехода MDP следующим образом:

$$T(s', b'|s, b, a) = \delta_{\tau(s,b,a,s')}(b')P(s'|s, b, a), \tag{16.7}$$

где $\delta_x(y)$ – *дельта-функция Кронекера*[7], такая, что $\delta_x(y) = 1$, если $x = y$, и 0 в противном случае.

---

[6] Оно непрерывно в случае распределений Дирихле в пределах вероятностей переходов, как показано в примере 16.3.

[7] Эта функция названа в честь немецкого математика Леопольда Кронекера (1823–1891).

Второй член можно вычислить путем интегрирования:

$$P(s'|s,b,a) = \int_\theta b(\theta) P(s'|s,b,a) \mathrm{d}\theta = \int_\theta b(\theta) \theta_{(s,a,s')} \mathrm{d}\theta. \tag{16.8}$$

Это уравнение имеет аналитическое решение аналогично уравнению (15.1). В случае когда наше начальное убеждение b представлено разложением распределения Дирихле в уравнении (16.6), мы имеем

$$P(s'|s,b,a) = N(s,a,s') / \sum_{s''} N(s,a,s''). \tag{16.9}$$

Уравнение оптимальности Беллмана (7.16) для MDP с известной моделью можно обобщить на случай, когда модель неизвестна:

$$U^*(s,b) = \max_a \left(R(s,a) + \gamma \sum_{s'} P(s'|s,b,a) U^*(s', \tau(s,b,a,s')) \right). \tag{16.10}$$

К сожалению, мы не можем напрямую использовать итерацию по стратегиям или по значениям, поскольку b является непрерывным. Однако мы можем прибегнуть к приближенным методам из главы 8 или методам текущего времени из главы 9. В данной главе представлены методы, которые лучше используют структуру адаптивного по Байесу MDP.

16.6. Апостериорная выборка

Альтернативой нахождению оптимальной функции ценности в доверительном пространстве является использование *апостериорной выборки* (posterior sampling)[8], которая первоначально была введена в контексте исследования задач о бандите в разделе 15.4[9]. Мы извлекаем выборку θ из текущего доверительного пространства b, а затем ищем наилучшее действие, предполагая, что θ описывает истинную модель. Затем мы обновляем наше убеждение о пространстве параметров, берем новую выборку и решаем соответствующую задачу MDP. Иллюстрация этого процесса приведена в примере 16.4.

Преимущество метода апостериорной выборки состоит в том, что нам не нужно выбирать эвристические параметры исследования. Однако решение MDP на каждом этапе может быть дорогостоящим в вычислительном плане. Метод выборки дискретного MDP из апостериорного распределения реализован в алгоритме 16.9.

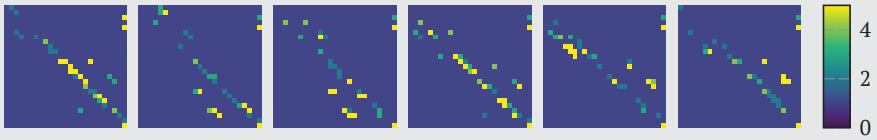
---

[8] M. J. A. Strens, *A Bayesian Framework for Reinforcement Learning*, in International Conference on Machine Learning (ICML), 2000.

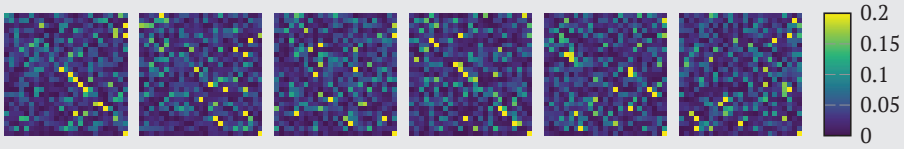
[9] В этом разделе мы сделали выборку из апостериорного распределения вероятности выигрыша, а затем предположили, что выбранные вероятности были правильными при выборе действия.

Пример 16.4. Применение оценки байесовской модели и апостериорной выборки к задаче гексамира

Применим оценку байесовской модели к задаче гексамира. Начнем с сопоставления однородных априорных распределений Дирихле с каждой парой состояние-действие. После 100 прогонов модели длиной 10 и добавления подсчетов переходов к псевдоподсчетам в априорном распределении параметры наших апостериорных распределений по последующим состояниям выглядят следующим образом:



Мы можем сделать выборку из этого распределения, чтобы создать модель, показанную здесь. Обратите внимание, что она имеет гораздо больше отличных от нуля вероятностей перехода, чем модели максимального правдоподобия, показанные в примере 16.1.



Алгоритм 16.9. Метод обновления с апостериорной выборкой. После обновления параметров байесовского апостериорного распределения мы выбираем задачу MDP из этого апостериорного распределения. Эта реализация алгоритма предполагает дискретное пространство состояний и действий с моделью Дирихле, отражающей неопределенность в вероятностях перехода от каждой пары состояние-действие. Чтобы сгенерировать модель перехода, мы перебираем каждое состояние и действие и выборку из связанного распределения Дирихле. Получив выборочную задачу $\mathcal{P}$, мы решаем ее, используя формулу линейного программирования, и сохраняем результирующую функцию ценности U

```
struct PosteriorSamplingUpdate end

function Base.rand(model::BayesianMDP)
    𝒮, 𝒜 = model.𝒮, model.𝒜
    T = zeros(length(𝒮), length(𝒜), length(𝒮))
    for s in 𝒮
        for a in 𝒜
            T[s,a,:] = rand(model.D[s,a])
        end
```

```
    end
    return MDP(T, model.R, model.γ)
end

function update!(planner::PosteriorSamplingUpdate, model, s, a, r, s′)
    𝒫 = rand(model)
    U = solve(𝒫).U
    copy!(model.U, U)
end
```

16.7. Заключение

- Методы на основе моделей изучают модели перехода и вознаграждения посредством взаимодействия с окружающей средой.
- Модели максимального правдоподобия используют подсчеты переходов для вычисления оценки вероятностей перехода в последующие состояния и для отслеживания среднего вознаграждения, связанного с парами состояние-действие.
- Модели максимального правдоподобия должны сочетаться со стратегией исследования, подобной той, что была представлена в предыдущей главе применительно к задаче о бандитах.
- Хотя мы можем заново спланировать каждый шаг на основе опыта, прямая реализация этого подхода является вычислительно затратной.
- Метод приоритетного выметания может сократить объем вычислительных затрат на перепланирование, обновляя ценность только тех состояний, которые, по-видимому, нуждаются в этом больше всего в развивающейся модели среды.
- Методы, основанные на байесовской модели, сохраняют распределение вероятностей возможных задач, позволяя получать принципиальное обоснование исследований.
- В адаптивных байесовых MDP их состояния дополняют исходный MDP распределением вероятностей по возможным моделям MDP.
- Апостериорная выборка снижает высокую вычислительную сложность решения адаптивного байесова MDP за счет решения MDP, выбранного из апостериорного распределения параметров, вместо рассуждений обо всех возможных MDP.

16.8. Упражнения

Упражнение 16.1. Предположим, у нас есть агент, действующий в среде с тремя состояниями и двумя действиями с неизвестными моделями перехода и вознаграждения. Выполним одну последовательность прямого взаимодействия со средой. В табл. 16.1 приведены состояние, действие, вознаграждение и результирующее состояние. Используйте метод максимального правдоподобия, чтобы оценить функции перехода и вознаграждения по этим данным.

s	a	r	s'
s_2	a_1	2	s_1
s_1	a_2	1	s_2
s_2	a_2	1	s_1
s_1	a_2	1	s_2
s_2	a_2	1	s_3
s_3	a_2	2	s_2
s_2	a_2	1	s_3
s_3	a_2	2	s_3
s_3	a_1	2	s_2
s_2	a_1	2	s_3

Таблица 16.1. Данные о переходах состояний

Решение. Сначала мы заносим в таблицу количество переходов из каждого состояния и действия $N(s, a)$, полученные вознаграждения $\rho(s, a)$ и оценку по методу максимального правдоподобия функции вознаграждения $\hat{R}(s, a) = \rho(s, a)/N(s, a)$ следующим образом:

s	a	$N(s, a)$	$\rho(s, a)$	$\hat{R}(s,a) = \frac{\rho(s,a)}{N(s,a)}$
s_1	a_1	0	0	0
s_1	a_2	2	2	1
s_2	a_1	2	4	2
s_2	a_2	3	3	1
s_3	a_1	1	2	2
s_3	a_2	2	4	2

В следующем наборе таблиц мы вычисляем количество наблюдаемых переходов $N(s, a, s')$ и оценку максимального правдоподобия модели перехода $\hat{T}(s'|s, a) = N(s, a, s')/N(s, a)$. При $N(s, a) = 0$ используется равномерное распределение по полученным состояниям.

s	a	s'	$N(s, a, s')$	$\hat{T}(s'\|s,a) = \frac{N(s,a,s')}{N(s,a)}$
s_1	a_1	s_1	0	1/3
s_1	a_1	s_2	0	1/3
s_1	a_1	s_3	0	1/3
s_1	a_2	s_1	0	0
s_1	a_2	s_2	2	1
s_1	a_2	s_3	0	0
s_2	a_1	s_1	1	1/2
s_2	a_1	s_2	0	0
s_2	a_1	s_3	1	1/2
s_2	a_2	s_1	1	1/3
s_2	a_2	s_2	0	0
s_2	a_2	s_3	2	2/3
s_3	a_1	s_1	0	0
s_3	a_1	s_2	1	1
s_3	a_1	s_3	0	0
s_3	a_2	s_1	0	0
s_3	a_2	s_2	1	1/2
s_3	a_2	s_3	1	1/2

Упражнение 16.2. Укажите нижнюю и верхнюю границы количества обновлений, которые могут быть выполнены во время итерации приоритетного выметания.

Решение. Нижняя граница количества обновлений, выполненных в итерации приоритетного выметания, равна 1. Она достигается во время нашей первой итерации с использованием модели максимального правдоподобия, где единственным ненулевым элементом в нашей модели перехода является $T(s'|s, a)$. Поскольку никакие пары состояние=действие (s^-, a^-) не переходят в s, наша очередь приоритетов будет пустой и, таким образом, будет выполнено единственное обновление $U(s)$.

Верхняя граница количества обновлений, выполненных в итерации приоритетного выметания, равна $|\mathcal{S}|$. Предположим, что мы только что выполнили переходы к s' и $\hat{T}(s'|s, a) > 0$ для всех s и a. Если не задано другое максимальное количество, мы выполним $|\mathcal{S}|$ обновлений. Если указано максимальное количество обновлений $m < |\mathcal{S}|$, верхняя граница снижается до m.

Упражнение 16.3. Допустим, мы выполняем байесовское обучение с подкреплением параметров модели перехода для дискретной MDP с пространством состояний $\mathcal{S}$ и пространством действий $\mathcal{A}$. Сколько независимых параметров

существует при использовании распределений Дирихле для представления неопределенности в модели перехода?

Решение. Для каждого состояния и действия мы задаем распределение Дирихле по параметрам вероятности перехода, поэтому у нас будет $|\mathcal{S}||\mathcal{A}|$ распределений Дирихле. Каждое распределение Дирихле определяется с помощью $|\mathcal{S}|$ независимых параметров. Итого имеем $|\mathcal{S}|^2|\mathcal{A}|$ независимых параметров.

Упражнение 16.4. Вернемся к условию задачи в упражнении 16.1, но на этот раз будем использовать байесовское обучение с подкреплением с априорным распределением Дирихле. Если нам дано равномерное априорное распределение и мы находимся в состоянии s_2 и предпринимаем действие a_1, то каким будет апостериорное распределение для следующего состояния?

Решение. $\mathrm{Dir}\left(\boldsymbol{\theta}_{(s_2, s_1)} | [2,1,2]\right)$.

17 Свободные методы обучения с подкреплением

В отличие от методов, основанных на моделях перехода и вознаграждения, *обучение с подкреплением без использования моделей* (model-free reinforcement learning) не требует наличия явных представлений таких моделей[1]. Для краткости далее мы будем называть эти методы *свободными от моделей*, или просто *свободными*. Свободные методы, обсуждаемые в этой главе, напрямую моделируют функцию полезности действия. В отсутствие явных представлений моделей есть определенное удобство, особенно когда задача представлена в многомерном пространстве. Эта глава начинается с понятия инкрементной оценки среднего значения распределения. Затем мы обсудим некоторые распространенные свободные алгоритмы и методы, предназначенные для более эффективной оценки и использования отложенного вознаграждения. Наконец, мы обсудим, как использовать аппроксимацию функций для обобщения опыта[2].

17.1. Инкрементное вычисление среднего значения распределения

Многие свободные методы оценивают функцию полезности действия $Q(s, a)$ по выборкам путем инкрементного вычисления. На данный момент предположим, что нас интересует только ожидание одной переменной X, вычисляемое по m выборкам:

---

[1] Многие темы этой главы более подробно освещены в книге R. S. Sutton, A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. MIT Press, 2018. См. также D. P. Bertsekas, *Reinforcement Learning and Optimal Control*. Athena Scientific, 2019.

[2] Хотя эта часть книги посвящена задачам, в которых модели среды неизвестны, обучение с подкреплением часто используется для решения задач с известными моделями. Методы без использования моделей, обсуждаемые в этой главе, могут быть особенно полезны в сложных средах как форма приближенного динамического программирования. Их можно использовать для формирования стратегий в офлайн-режиме или в качестве инструмента для выбора следующего действия в онлайн-режиме.

$$\hat{x}_m = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}, \tag{17.1}$$

где $x^{(1)}$, ..., $x^{(m)}$ – выборки. Мы можем получить инкрементное обновление следующим образом:

$$\hat{x}_m = \frac{1}{m}\left(x^{(m)} + \sum_{i=1}^{m-1} x^{(i)} \right) \tag{17.2}$$

$$= \frac{1}{m}\left(x^{(m)} + (m-1)\hat{x}_{m-1} \right) \tag{17.3}$$

$$= \hat{x}_{m-1} + \frac{1}{m}\left(x^{(m)} - \hat{x}_{m-1} \right). \tag{17.4}$$

Перепишем это уравнение, введя функцию *скорости обучения* $\alpha(m)$:

$$\hat{x}_m = \hat{x}_{m-1} + \alpha(m)\left(x^{(m)} - \hat{x}_{m-1} \right). \tag{17.5}$$

Скорость обучения может быть функцией, отличной от $1/m$. Чтобы гарантировать сходимость, мы обычно выбираем $\alpha(m)$ так, чтобы выполнялись условия $\sum_{m=1}^{\infty} \alpha(m) = \infty$ и $\sum_{m=1}^{\infty} \alpha^2(m) < \infty$. Первое условие гарантирует, что шаги будут достаточно большими, а второе – что шаги будут достаточно малыми[3].

Если скорость обучения постоянна, что характерно для обучения с подкреплением, то веса более старых выборок затухают по степенному закону со скоростью $(1 - \alpha)$. При постоянной скорости обучения мы можем обновить нашу оценку после наблюдения за x, используя следующее правило:

$$\hat{x} \leftarrow \hat{x} + \alpha(x - \hat{x}). \tag{17.6}$$

Реализация инкрементного вычисления представлена в алгоритме 17.1. Использование нескольких скоростей обучения показано в примере 17.1.

Обсуждаемое здесь правило обновления встретится снова в последующих разделах и связано со стохастическим градиентным спуском. Величина обновления пропорциональна разнице между выборкой и предыдущей оценкой. Разница между текущей выборкой и предыдущими оценками называется *ошибкой временно́й разницы* (temporal difference error).

---

[3] Для обсуждения сходимости и ее применения к некоторым другим алгоритмам, упомянутым в этой главе, см. статью T. Jaakkola, M. I. Jordan, S. P. Singh, *On the Convergence of Stochastic Iterative Dynamic Programming Algorithms*, Neural Computation, vol. 6, no. 6, pp. 1185–1201, 1994.

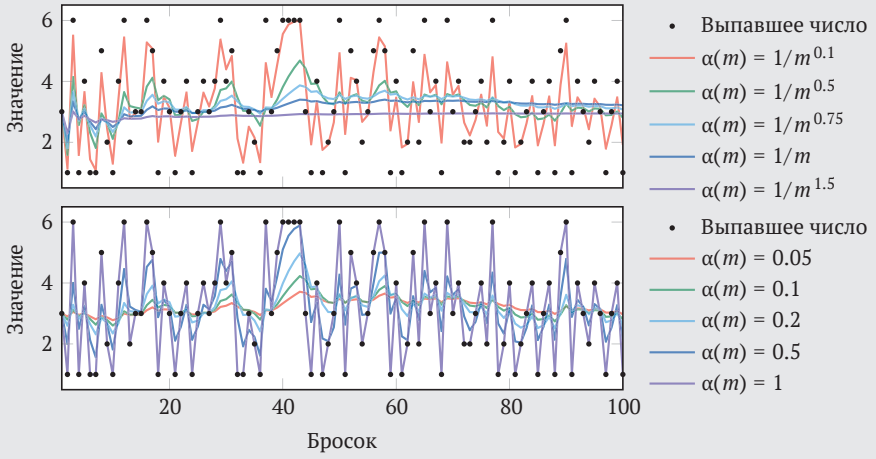
Алгоритм 17.1. Определение типа для инкрементного вычисления оценки среднего значения случайной величины. Ассоциированный тип обрабатывает текущее среднее значение `μ`, функцию скорости обучения `α` и количество итераций `m`. Вызов функции `update!` с новым значением `x` обновляет оценку

```
mutable struct IncrementalEstimate
    μ # средняя оценка
    α # функция скорости обучения
    m # количество обновлений
end

function update!(model::IncrementalEstimate, x)
    model.m += 1
    model.μ += model.α(model.m) * (x - model.μ)
    return model
end
```

Пример 17.1. Эффект затухания скорости обучения с различными функциями $\alpha(m)$

Рассмотрим вычисление ожидаемого значения, полученного при броске идеального шестигранного кубика. На рисунке ниже изображены *кривые обучения* (learning curve), которые показывают инкрементные оценки более 100 бросков для различных функций скорости обучения. Как мы видим, если $\alpha(m)$ затухает слишком быстро, сходимость не гарантируется, а если $\alpha(m)$ затухает недостаточно быстро, то сходимость достигается медленно.

При постоянных значениях $\alpha \in (0, 1]$ оценка среднего будет продолжать колебаться. Большие значения постоянной α приводят к сильным колебаниям, тогда как маленькие значения замедляют сходимость.



17.2. Q-обучение

Метод *Q-обучения* (алгоритм 17.2) основан на инкрементной оценке функции полезности действия $Q(s, a)$[4]. Выведем обновление функции из формулы полезности действия уравнения ожидания Беллмана:

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} T(s'|s,a) U(s') \tag{17.7}$$

$$= R(s,a) + \gamma \sum_{s'} T(s'|s,a) \max_{a'} Q(s',a'). \tag{17.8}$$

Вместо использования T и R мы можем переписать приведенное выше уравнение в форме ожидания по выборкам для вознаграждения r и следующего состояния s':

$$Q(s,a) = \mathbb{E}_{r,s'}\left[r + \gamma \max_{a'} Q(s',a') \right]. \tag{17.9}$$

Воспользуемся уравнением (17.6), чтобы сформулировать правило инкрементного обновления функции полезности действия[5]:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]. \tag{17.10}$$

Наш выбор действий влияет на то, в каком состоянии мы оказываемся, и, следовательно, на нашу способность точно оценивать $Q(s, a)$. Чтобы гарантировать сходимость функции полезности действия, необходимо применять определенную стратегию исследования, такую как ε-жадную или softmax, точно так же, как мы это сделали для алгоритмов на основе моделей в предыдущей главе. В примере 17.2 показано, как выполняется моделирование с использованием Q-обучения и стратегии исследования. На рис. 17.1 изображена иллюстрация этого процесса при решении задачи о гексамире.

---

[4] C. J. C. H. Watkins, *Learning from Delayed Rewards*, Ph.D. dissertation, University of Cambridge, 1989.

[5] Максимизация в этом уравнении может внести систематическую ошибку. Такие алгоритмы, как *двойное Q-обучение*, пытаются исправить ее и могут способствовать повышению точности. H. van Hasselt, *Double Q-Learning*, in Advances in Neural Information Processing Systems (NIPS), 2010.

Алгоритм 17.2. Реализация метода Q-обучения, который можно применять в задачах с неизвестными функциями перехода и вознаграждения. Обновление изменяет матрицу значений состояние-действие `Q`. Данный метод можно использовать вместе с ε-жадной стратегией исследования в функции моделирования из алгоритма 15.9. Упомянутая функция вызывает функцию коррекции Q для состояния `s′`, но данная реализация Q-обучения не использует ее

```
mutable struct QLearning
    𝒮  # пространство состояний (пусть 1:nstates)
    𝒜  # пространство действий (пусть 1:nactions)
    γ  # дисконт
    Q  # функция полезности действия
    α  # скорость обучения
end

lookahead(model::QLearning, s, a) = model.Q[s,a]

function update!(model::QLearning, s, a, r, s′)
    γ, Q, α = model.γ, model.Q, model.α
    Q[s,a] += α*(r + γ*maximum(Q[s′,:]) - Q[s,a])
    return model
end
```

Пример 17.2. Совместное использование стратегии исследования и Q-обучения. Величины параметров являются условными

Предположим, мы хотим применить Q-обучение к задаче MDP $\mathcal{P}$. Мы можем сформировать определенную стратегию исследования, например ε-жадную, реализованную в алгоритме 16.6 из предыдущей главы. Модель Q-обучения получена из алгоритма 17.2, а функция моделирования реализована в алгоритме 15.9.

```
Q = zeros(length(𝒫.𝒮), length(𝒫.𝒜))
α = 0.2 # скорость обучения
model = QLearning(𝒫.𝒮, 𝒫.𝒜, 𝒫.γ, Q, α)
ε = 0.1 # вероятность случайного действия
π = EpsilonGreedyExploration(ε)
k = 20 # количество шагов моделирования
s = 1  # начальное состояние
simulate(𝒫, model, π, k, s)
```

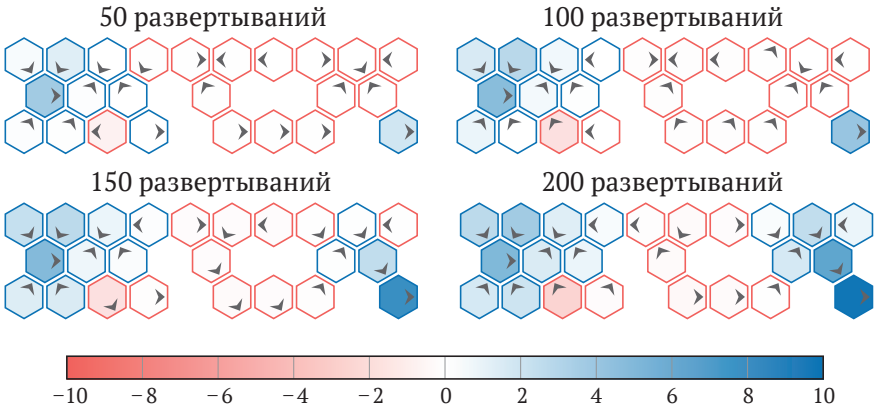


Рис. 17.1. *Q*-обучение, используемое для итеративного обучения функции полезности действия в задаче гексамира. Каждое состояние окрашено в соответствии с ожидаемым значением наилучшего действия в этом состоянии относительно *Q*. Аналогичным образом выполняемые действия являются наилучшими ожидаемыми действиями. *Q*-обучение проводилось с $\alpha = 0.1$ и 10 шагами на развертывание

17.3. Алгоритм SARSA

Метод *SARSA* (алгоритм 17.3) – это альтернатива *Q*-обучению[6]. Метод получил такое название, потому что использует (s, a, r, s', a') для обновления функции Q на каждом шаге. Он использует для обновления Q фактическое следующее действие a' вместо максимизации по всем возможным действиям:

$$Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma Q(s', a') - Q(s, a)). \tag{17.11}$$

При подходящей стратегии исследования a' будет сходиться к $\arg\max_{a'} Q(s', a')$, что возвращает нас к обновлению методом *Q*-обучения.

Алгоритм SARSA относится к группе методов обучения с подкреплением *с прямым использованием стратегии* (on-policy), потому что он пытается напрямую оценить полезность стратегии исследования, просто следуя ей. Напротив, *Q*-обучение – это метод *косвенного использования стратегии* (off-policy), поскольку он пытается найти полезность оптимальной стратегии через использование стратегии исследования. Хотя *Q*-обучение и SARSA сходятся к оптимальной стратегии, скорость сходимости зависит от приложения. Использование алгоритма SARSA на примере задачи гексамира показано на рис. 17.2.

[6] Этот подход был предложен под другим названием в G. A. Rummery, M. Niranjan, *On-Line Q-Learning Using Connectionist Systems*, Cambridge University, Tech. Rep. CUED/F-INFENG/TR 166, 1994.

Алгоритм 17.3. Обновление по методу SARSA для свободного обучения с подкреплением. Мы обновляем матрицу `Q`, содержащую значения состояния-действия, `α` – это постоянная скорость обучения, а `ℓ` – кортеж самого последнего опыта. Как и в случае реализации *Q*-обучения, функция коррекции может использоваться для моделирования в алгоритме 15.9

```
mutable struct Sarsa
    𝒮  # пространство состояний (пусть 1:nstates)
    𝒜  # пространство действий (пусть 1:nactions)
    γ  # дисконт
    Q  # функция полезности действия
    α  # скорость обучения
    ℓ  # кортеж самого последнего опыта (s,a,r)
end

lookahead(model::SARSA, s, a) = model.Q[s,a]

function update!(model::SARSA, s, a, r, s′)
    if model.ℓ != nothing
        γ, Q, α, ℓ = model.γ, model.Q, model.α, model.ℓ
        model.Q[ℓ.s,ℓ.a] += α*(ℓ.r + γ*Q[s,a] - Q[ℓ.s,ℓ.a])
    end
    model.ℓ = (s=s, a=a, r=r)
    return model
end
```

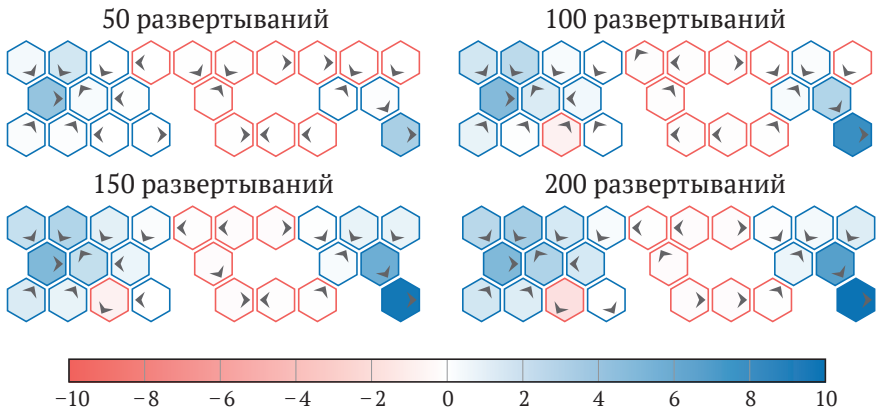


Рис. 17.2. Применение SARSA для итеративного обучения функции полезности действия в задаче гексамира способом, в остальном идентичным показанному на рис. 17.1. Мы обнаружили, что SARSA медленнее сходится к истинной функции полезности действия

17.4. Следы приемлемости

Одним из недостатков алгоритмов *Q*-обучения и SARSA является то, что обучение может быть очень медленным, особенно при *разреженном вознаграждении* (sparse reward). Например, предположим, что среда имеет единственное целевое состояние, обеспечивающее большое вознаграждение, а во всех остальных состояниях вознаграждение равно нулю. После нескольких случайных исследований в окружающей среде мы достигаем целевого состояния. Независимо от того, используем ли мы *Q*-обучение или SARSA, полезность действия обновится только для состояния, непосредственно предшествующего целевому состоянию. Полезности действий во всех других состояниях, ведущих к цели, остаются равными нулю. Требуется большой объем исследования, чтобы постепенно распространить ненулевые значения на оставшуюся часть пространства состояний.

Q-обучение и SARSA можно модифицировать таким образом, чтобы вознаграждение распространялось обратно к состояниям и действиям, ведущим к источнику вознаграждения, в соответствии с методом *следов приемлемости* (eligibility traces)[7]. Согласно этому методу величина вознаграждения убывает экспоненциально, поэтому состояниям, расположенным ближе к вознаграждению, присваиваются более высокие значения полезности. Обычно в качестве параметра экспоненциального затухания используется коэффициент $0 < \lambda < 1$. Версии методов *Q*-обучения и SARSA с использованием следов приемлемости часто называют $Q(\lambda)$ и SARSA(λ)[8].

Вариант SARSA(λ) реализован в алгоритме 17.4, который формирует экспоненциально убывающее количество посещений $N(s, a)$ для всех пар состояние-действие. Когда действие a выполняется в состоянии s, $N(s, a)$ увеличивается на 1. Затем к каждой паре состояние-действие частично применяется обновление временной разницы SARSA в соответствии с этим затухающим счетчиком посещений.

Пусть δ обозначает обновление временной разницы SARSA:

$$\delta = r + \gamma Q(s', a') - Q(s, a). \tag{17.12}$$

Каждый элемент функции полезности действия обновляется в соответствии с выражением

$$Q(s, a) \leftarrow Q(s, a) + \alpha\delta N(s, a). \tag{17.13}$$

---

[7] Следы приемлемости были предложены в контексте изучения временных различий в работе R. Sutton, *Learning to Predict by the Methods of Temporal Differences*, Machine Learning, vol. 3, no. 1, pp. 9–44, 1988.

[8] Эти алгоритмы были предложены в диссертации C. J. C. H. Watkins, *Learning from Delayed Rewards*, University of Cambridge, 1989, и в статье J. Peng, R. J. Williams, *Incremental Multi-Step Q-Learning*, Machine Learning, vol. 22, no. 1–3, pp. 283–290, 1996.

Затем количество посещений затухает с использованием как коэффициента дисконтирования, так и параметра экспоненциального затухания:

$$N(s, a) \leftarrow \gamma\lambda N(s, a). \tag{17.14}$$

Хотя влияние следов приемлемости особенно заметно в средах с разреженным вознаграждением, алгоритм может ускорить обучение и в обычных средах, где вознаграждение более широко распределено по пространству состояний.

Алгоритм 17.4. Обновление по методу SARSA(λ), которое использует следы приемлемости для распространения вознаграждений в прошлое, чтобы ускорить изучение редких вознаграждений. Матрица `Q` содержит значения состояния-действия, матрица `N` содержит экспоненциально затухающие счетчики посещений состояния-действия, `α` – постоянная скорость обучения, `λ` – параметр экспоненциального затухания, а `ℓ` – кортеж самого последнего опыта

```
mutable struct SarsaLambda
    𝒮  # пространство состояний (пусть 1:nstates)
    𝒜  # пространство действий (пусть 1:nactions)
    γ  # дисконт
    Q  # функция полезности действия
    N  # след
    α  # скорость обучения
    λ  # скорость затухания следа
    ℓ  # кортеж самого последнего опыта (s,a,r)
end

lookahead(model::SarsaLambda, s, a) = model.Q[s,a]

function update!(model::SarsaLambda, s, a, r, s′)
    if model.ℓ != nothing
        γ, λ, Q, α, ℓ = model.γ, model.λ, model.Q, model.α, model.ℓ
        model.N[ℓ.s,ℓ.a] += 1
        δ = ℓ.r + γ*Q[s,a] - Q[ℓ.s,ℓ.a]
        for s in model.𝒮
            for a in model.𝒜
                model.Q[s,a] += α*δ*model.N[s,a]
                model.N[s,a] *= γ*λ
            end
        end
    else
        model.N[:,:] .= 0.0
    end
    model.ℓ = (s=s, a=a, r=r)
    return model
end
```

Следует проявлять особую осторожность при использовании следов приемлемости в алгоритме косвенной оценки стратегии, таком как Q-обучение,

который пытается найти оптимальную стратегию *принятия решений*[9]. Алгоритмы с использованием следов приемлемости передают обратно вознаграждения, полученные из стратегии *исследования*. Это несоответствие между типами стратегии может привести к нестабильности обучения.

17.5. Формирование вознаграждения

Расширение горизонта функции вознаграждения также может улучшить обучение, особенно при проблемах с редкими вознаграждениями. Например, если мы пытаемся достичь единственного целевого состояния, мы могли бы дополнить функцию вознаграждения величиной, обратно пропорциональной расстоянию до цели. В качестве альтернативы мы могли бы добавить штраф, пропорциональный нашему расстоянию до цели. Например, во время игры в шахматы к функции вознаграждения можно добавить штраф за потерю фигуры, даже если мы заботимся только о победе в конце игры, а не о сохранности отдельных фигур.

Уточнение функции вознаграждения во время обучения путем добавления знаний предметной области для ускорения обучения называется *доводкой вознаграждения* (reward shaping). Предположим, что вознаграждения в нашей задаче генерируются при помощи функции $R(s, a, s')$, благодаря которой они зависят от результирующего состояния. Мы добавим в этот механизм *функцию доводки* (shaping function) $F(s, a, s')$. Во время обучения вместо $R(s, a, s')$ для оценки вознаграждения мы используем $R(s, a, s') + F(s, a, s')$.

Разумеется, добавление слагаемого $F(s, a, s')$ может изменить оптимальную стратегию. Часто бывает необходимо сохранить оптимальную стратегию при доводке вознаграждения. Оказывается, что стратегия сохраняет оптимальность, только если

$$F(s, a, s') = \gamma\beta(s') - \beta(s) \tag{17.15}$$

для некоторой гармонической функции $\beta(s)$[10].

17.6. Аппроксимация функции полезности действия

Алгоритмы, рассмотренные до сих пор в этой главе, предполагали дискретные пространства состояний и действий, в которых функция полезности действия

---

[9] Для обзора этой задачи и потенциального решения см. A. Harutyunyan, M. G. Bellemare, T. Stepleton, R. Munos, *Q*(λ) *with Off-Policy Corrections*, in International Conference on Algorithmic Learning Theory (ALT), 2016.

[10] A. Y. Ng, D. Harada, S. Russell, *Policy Invariance Under Reward Transformations: Theory and Application to Reward Shaping*, in International Conference on Machine Learning (ICML), 1999.

может храниться в табличной форме. Мы можем доработать наши алгоритмы для использования аппроксимации функции полезности, что позволит нам применять их к задачам с большими или непрерывными пространствами и обобщать ограниченный опыт. Подобно методу, рассмотренному в главе 8 для случая известной модели, мы будем использовать $Q_{\boldsymbol{\theta}}(s, a)$ для представления параметрической аппроксимации нашей функции полезности действия, когда модель неизвестна[11].

Чтобы проиллюстрировать эту идею, получим версию Q-обучения, в которой используется параметрическая аппроксимация. Нам нужно минимизировать потерю между нашим приближением и оптимальной функцией полезности действия $Q^*(s, a)$, которую мы определяем как[12]

$$\ell(\boldsymbol{\theta}) = \frac{1}{2} \mathop{\mathbb{E}}_{(s,a)\sim\pi^*} \left[(Q^*(s,a) - Q_{\boldsymbol{\theta}}(s,a))^2 \right]. \tag{17.16}$$

Ожидание вычисляется по парам состояние-действие, которые возникают при следовании оптимальной стратегии π^*.

Обычный подход к минимизации этой потери заключается в использовании какой-либо формы градиентного спуска. Градиент потери вычисляется из уравнения

$$\nabla\ell(\boldsymbol{\theta}) = -\mathop{\mathbb{E}}_{(s,a)\sim\pi^*} \left[(Q^*(s,a) - Q_{\boldsymbol{\theta}}(s,a)) \nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\theta}}(s,a) \right]. \tag{17.17}$$

Обычно выбирают параметрические представления функции полезности действия, которые являются дифференцируемыми и где $\nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\theta}}(s, a)$ легко вычислить; например, это могут быть линейные представления или нейронная сеть. Если мы применим градиентный спуск[13], правило обновления параметров будет следующим:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \mathop{\mathbb{E}}_{(s,a)\sim\pi^*} \left[(Q^*(s,a) - Q_{\boldsymbol{\theta}}(s,a)) \nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\theta}}(s,a) \right], \tag{17.18}$$

где α – коэффициент шага или скорость обучения. Мы можем аппроксимировать правило обновления (17.18), используя образцы пар состояние-действие (s, a) по мере их получения:

$$(Q^*(s,a) - Q_{\boldsymbol{\theta}}(s,a)) \nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\theta}}(s,a). \tag{17.19}$$

---

[11] В последние годы основное внимание уделялось глубокому обучению с подкреплением, где для параметрического приближения используются глубокие нейронные сети. Обсуждение практических реализаций предоставлено в книге L. Graesser, W. L. Keng, *Foundations of Deep Reinforcement Learning*. Addison Wesley, 2020.

[12] Дробь 1/2 помещена впереди для удобства, потому что позже мы будем вычислять производную этого квадратичного уравнения.

[13] Мы должны спускаться, а не подниматься по градиенту, потому что пытаемся минимизировать потери.

Конечно, мы не можем вычислить уравнение (17.19) напрямую, потому что это потребовало бы знания оптимальной стратегии, которую мы пытаемся найти. Вместо этого попытаемся найти приближенное решение из наблюдаемого перехода и аппроксимации полезности действия:

$$Q^*(s,a) \approx r + \gamma \max_{a'} Q_{\boldsymbol{\theta}}(s',a'), \tag{17.20}$$

что приводит к следующему правилу обновления:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\left(r + \gamma \max_{a'} Q_{\boldsymbol{\theta}}(s',a') - Q_{\boldsymbol{\theta}}(s,a)\right)\nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\theta}}(s,a). \tag{17.21}$$

Это обновление реализовано в алгоритме 17.5 с добавлением масштабированного шага градиента (алгоритм 12.2), который часто необходим, для того чтобы шаги градиента не становились слишком большими. В примере 17.3 показано использование правила обновления с линейной аппроксимацией полезности действия. На рис. 17.3 продемонстрировано применение этого алгоритма в задаче о горной машине.

Алгоритм 17.5. Обновление параметров методом Q-обучения с аппроксимацией функции полезности действия. С каждым новым кортежем опыта `s`, `a`, `r`, `s′` мы обновляем вектор параметров `θ` с постоянной скоростью обучения `α`. Параметрическая функция полезности действия задана как `Q(θ,s,a)`, а ее градиент равен `∇Q(θ,s,a)`

```
struct GradientQLearning
    𝒜  # пространство действий (пусть 1:nactions)
    γ  # дисконт
    Q  # параметрическая функция полезности действия Q(θ,s,a)
    ∇Q # градиент функции полезности действия
    θ  # вектор параметров функции полезности действия
    α  # скорость обучения
end

function lookahead(model::GradientQLearning, s, a)
    return model.Q(model.θ, s,a)
end

function update!(model::GradientQLearning, s, a, r, s′)
    𝒜, γ, Q, θ, α = model.𝒜, model.γ, model.Q, model.θ, model.α
    u = maximum(Q(θ,s′,a′) for a′ in 𝒜)
    Δ = (r + γ*u - Q(θ,s,a))*model.∇Q(θ,s,a)
    θ[:] += α*scale_gradient(Δ, 1)
    return model
end
```

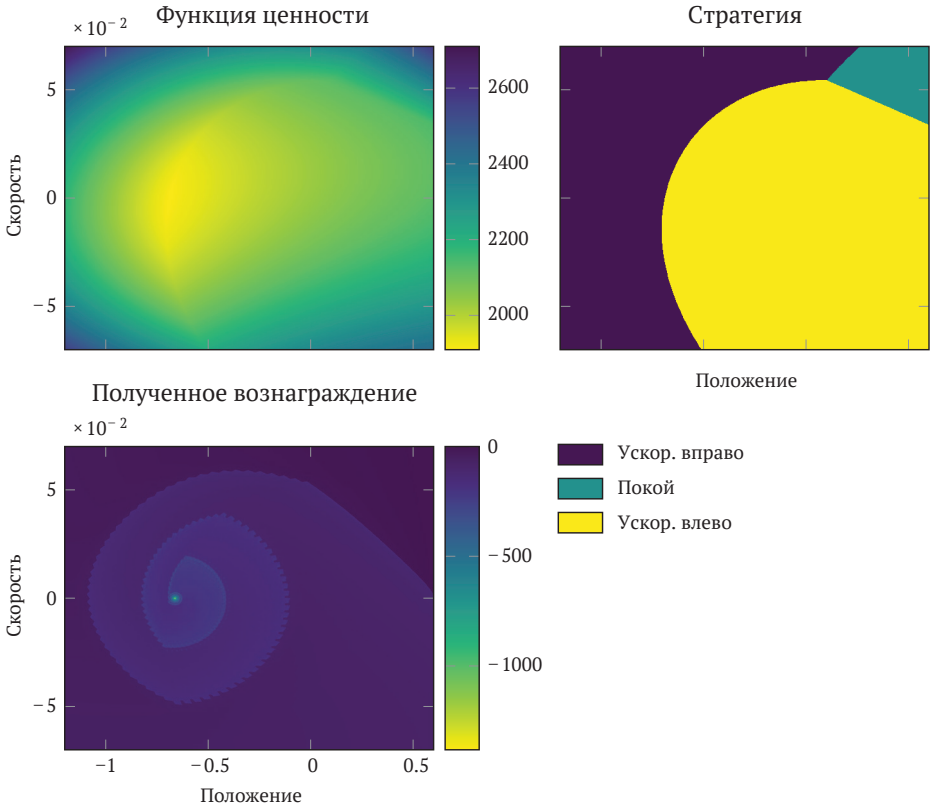


Рис. 17.3. Функция полезности и стратегия, полученные путем использования *Q*-обучения с линейной аппроксимацией, применительно к задаче о горной машине (приложение F.4). Базисные функции представляют собой полиномы от положения и скорости не выше восьмой степени, и каждая из них повторяется трижды для трех действий. «Полученное вознаграждение» означает вознаграждение, полученное агентом при прогоне с использованием жадной стратегии с аппроксимированной функцией полезности

Пример 17.3. Использование стратегии исследования с *Q*-обучением совместно с аппроксимацией функции полезности действия. Значения параметров являются условными

Допустим, мы собираемся применить *Q*-обучение с линейной аппроксимацией функции полезности действия к задаче простого регулятора с $\gamma = 1$. Приближенная функция полезности действия имеет вид $Q_{\boldsymbol{\theta}}(s, a) = \boldsymbol{\theta}^{\top}\boldsymbol{\beta}(s, a)$, где базисная функция

$$\boldsymbol{\beta}(s, a) = [s, s^2, a, a^2, 1].$$

В случае линейной модели

$$\nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\theta}}(s, a) = \boldsymbol{\beta}(s, a).$$

Этот подход к решению задачи $\mathcal{P}$ реализован в следующем алгоритме:

```
β(s,a) = [s,s^2,a,a^2,1]
Q(θ,s,a) = dot(θ,β(s,a))
∇Q(θ,s,a) = β(s,a)
θ = [0.1,0.2,0.3,0.4,0.5] # вектор начальных параметров
α = 0.5 # скорость обучения
model = GradientQLearning(𝒫.𝒜, 𝒫.γ, Q, ∇Q, θ, α)
ε = 0.1 # вероятность случайного действия
π = EpsilonGreedyExploration(ε)
k = 20  # количество шагов моделирования
s = 0.0 # начальное состояние
simulate(𝒫, model, π, k, s)
```

17.7. Воспроизведение опыта

Основной проблемой использования аппроксимации глобальной функции в обучении с подкреплением является *катастрофическое забывание* (catastrophic forgetting). Например, агент обнаруживает, что конкретная стратегия приводит его в область пространства состояний с низким вознаграждением. Затем он уточняет стратегию, чтобы избежать этой области. Однако по прошествии некоторого времени он может забыть, почему было важно избегать этой области пространства состояний, и рискует вернуться к неэффективной стратегии.

Катастрофическое забывание можно смягчить с помощью *воспроизведения опыта* (experience replay)[14], когда фиксированное количество самых последних кортежей опыта сохраняется на протяжении итераций обучения. *Пакет* кортежей равномерно выбирают из *глобальной памяти* (replay memory) в качестве напоминания о том, что следует избегать стратегий, которые, как мы уже обнаружили, являются плохими[15]. Уравнение обновления (17.21) принимает следующий вид:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \frac{1}{m_{\text{grad}}} \sum_i \left(r^{(i)} + \gamma \max_{a'} Q_{\boldsymbol{\theta}}(s'^{(i)}, a') - Q_{\boldsymbol{\theta}}(s^{(i)}, a^{(i)}) \right) \nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\theta}}(s^{(i)}, a^{(i)}), \tag{17.22}$$

где $s^{(i)}$, $a^{(i)}$, $r^{(i)}$ и $s'^{(i)}$ – i-й кортеж опыта в случайном пакете размером m_{grad}.

---

[14] Воспроизведение опыта сыграло важную роль в работе V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller, *Playing Atari with Deep Reinforcement Learning*, 2013. arXiv: 1312. 5602v1. Эта идея была предложена ранее в L.-J. Lin, *Reinforcement Learning for Robots Using Neural Networks*, Ph.D. dissertation, Carnegie Mellon University, 1993.

[15] Варианты этого подхода предусматривают назначение приоритетов опыта. T. Schaul, J. Quan, I. Antonoglou, D. Silver, *Prioritized Experience Replay*, in International Conference on Learning Representations (ICLR), 2016.

Воспроизведение опыта многократно улучшает процесс обучения, тем самым повышая эффективность данных. Кроме того, равномерная случайная выборка из глобальной памяти разбивает на части полученные из развертываний последовательности. Это разбиение уменьшает дисперсию оценки градиента, поскольку в противном случае последовательности коррелируют. Воспроизведение опыта стабилизирует процесс обучения, сохраняя информацию из предыдущих наборов параметров стратегий.

В алгоритме 17.6 реализовано добавление воспроизведения опыта в Q-обучение с аппроксимацией функции полезности действия. В примере 17.4 показано, как применить этот подход к задаче простого регулятора.

Алгоритм 17.6. Q-обучение с аппроксимацией функции полезности и воспроизведением опыта. Обновление зависит от параметрической стратегии `Q(θ,s,a)` и градиента `∇Q(θ,s,a)`. Здесь выполняется обновление вектора параметров `θ` и кольцевого буфера памяти, предоставленного в `DataStructures.jl`. Алгоритм обновляет `θ` каждые `m` шагов, используя градиент, найденный по образцам `m_grad` из кольцевого буфера

```
struct ReplayGradientQLearning
    𝒜      # пространство действий (пусть 1:nactions)
    γ      # дисконт
    Q      # параметрическая функция полезности действия Q(θ,s,a)
    ∇Q     # градиент функции полезности действия
    θ      # параметр функции полезности действия
    α      # скорость обучения
    buffer # кольцевой буфер памяти
    m      # количество шагов между обновлениями градиента
    m_grad # размер пакета
end

function lookahead(model::ReplayGradientQLearning, s, a)
    return model.Q(model.θ, s,a)
end

function update!(model::ReplayGradientQLearning, s, a, r, s′)
    𝒜, γ, Q, θ, α = model.𝒜, model.γ, model.Q, model.θ, model.α
    buffer, m, m_grad = model.buffer, model.m, model.m_grad
    if isfull(buffer)
        U(s) = maximum(Q(θ,s,a) for a in 𝒜)
        ∇Q(s,a,r,s′) = (r + γ*U(s′) - Q(θ,s,a))*model.∇Q(θ,s,a)
        Δ = mean(∇Q(s,a,r,s′) for (s,a,r,s′) in rand(buffer, m_grad))
        θ[:] += α*scale_gradient(Δ, 1)
        for i in 1:m # discard oldest experiences
            popfirst!(buffer)
        end
    else
```

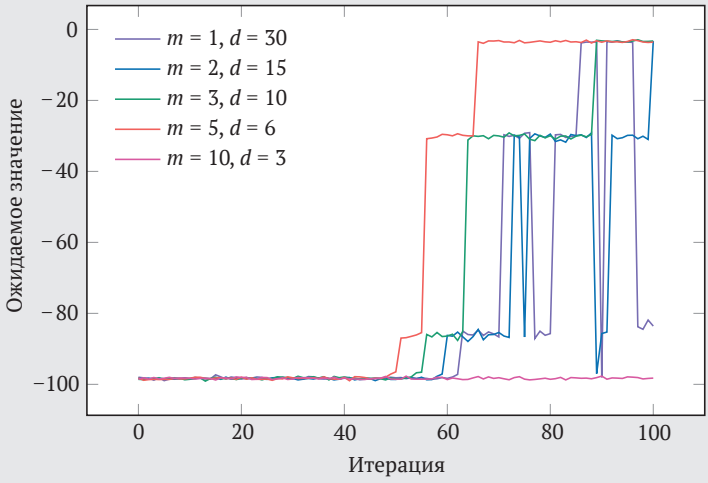
```
            push!(buffer, (s,a,r,s′))
        end
        return model
end
```

Пример 17.4. Применение метода воспроизведения опыта к задаче простого регулятора с *Q*-обучением и аппроксимацией функции полезности действия

Предположим, мы хотим добавить воспроизведение опыта в пример 17.3. При построении модели нам необходимо обеспечить буфер воспроизведения с желаемой емкостью:

```
capacity = 100 # максимальный размер буфера воспроизведения
ExperienceTuple = Tuple{Float64,Float64,Float64,Float64}
M = CircularBuffer{ExperienceTuple}(capacity) # буфер воспроизведения
m_grad = 20 # размер пакета
model = ReplayGradientQLearning(𝒫.𝒜, 𝒫.γ, Q, ∇Q, θ, α, M, m, m_grad)
```

Мы можем выбирать количество шагов между обновлениями градиента m и глубину каждого прохода моделирования d. На показанном ниже графике мы ограничиваем все обучающие прогоны $md = 30$ кортежами опыта на каждой итерации. Это говорит о том, что для успешного обучения необходимо развертывание на достаточную глубину. Кроме того, в большинстве случаев развертывания на чрезмерную глубину работают, как минимум, не хуже, чем умеренное количество развертываний на умеренную глубину, поэтому можно не бояться переусердствовать (в пределах разумных вычислительных затрат, разумеется).



17.8. Заключение

- Методы без использования моделей направлены на прямое обучение функции полезности действия, а не моделей перехода и вознаграждения.
- Для постепенного изучения среднего значения из последовательных обновлений можно использовать простые методы.
- Алгоритм *Q*-обучения пошагово обучает функцию полезности действия, используя аппроксимацию уравнения Беллмана.
- В отличие от *Q*-обучения, SARSA при обновлении использует действие, предпринятое стратегией исследования, а не ищет максимум среди последующих действий.
- Метод следов приемлемости может ускорить обучение, распространяя разреженные вознаграждения по пространству состояний-действий.
- *Q*-обучение можно применять для аппроксимации функций полезности с помощью стохастического градиентного спуска.
- Катастрофическое забывание, с которым сталкиваются *Q*-обучение и SARSA, можно смягчить с помощью метода воспроизведения опыта, основанного на использовании кортежей прошлого опыта.

17.9. Упражнения

Упражнение 17.1. Используя данный ниже набор выборок, дважды выполните инкрементную оценку среднего значения: один раз, используя скорость обучения $\alpha = 0.1$, и один раз, используя скорость обучения $\alpha = 0.5$. В обоих случаях применяйте начальное среднее значение, равное первой выборке:

$x^{(1:5)} = \{1.0, 1.8, 2.0, 1.6, 2.2\}$.

Решение. Зададим среднее значение на первой итерации равным первой выборке и перейдем к пошаговой оценке среднего значения, используя уравнение (17.6):

$\hat{x}_1 = 1.0;$
$\hat{x}_2 = 1.0 + 0.1(1.8 - 1.0) = 1.08;$
$\hat{x}_3 = 1.08 + 0.1(2.0 - 1.08) = 1.172;$
$\hat{x}_4 = 1.172 + 0.1(1.6 - 1.172) \approx 1.215;$
$\hat{x}_5 = 1.215 + 0.1(2.2 - 1.215) \approx 1.313;$

$\hat{x}_1 = 1.0;$
$\hat{x}_2 = 1.0 + 0.5(1.8 - 1.0) = 1.4;$
$\hat{x}_3 = 1.4 + 0.5(2.0 - 1.4) = 1.7;$
$\hat{x}_4 = 1.7 + 0.5(1.6 - 1.7) = 1.65;$
$\hat{x}_5 = 1.65 + 0.5(2.2 - 1.65) = 1.925.$

Упражнение 17.2. Продолжим предыдущее упражнение. Предположим, что после того, как вы оценили среднее значение по пяти выборкам для обоих методов, вам дана дополнительная выборка $x^{(6)}$, которую вы будете использовать

в качестве окончательной выборки при оценке среднего значения. Какой из двух вариантов инкрементной оценки (т. е. $\alpha = 0.1$ или $\alpha = 0.5$) предпочтительнее?

Решение. Хотя мы не знаем, какой будет выборка или каково лежащее в основе среднее значение процесса, вероятно, было бы разумно предпочесть второй вариант пошаговой оценки среднего, который использует $\alpha = 0.5$. Поскольку у нас осталась только одна выборка, первая скорость обучения слишком мала, чтобы значительно изменить среднее значение, в то время как вторая скорость обучения достаточно велика, чтобы отреагировать, не игнорируя прошлые выборки. Рассмотрим два случая.

1. Если предположить, что следующая выборка приблизительно равна *инкрементному* среднему всех предыдущих выборок, то мы имеем $x^{(6)} \approx \hat{x}_5$. Таким образом, выполнение инкрементного обновления среднего значения не приводит к изменению нашей оценки. У нас получится $\hat{x}_6 \approx 1.313$ для скорости обучения 0.1 и $\hat{x}_6 \approx 1.925$ для скорости обучения 0.5.
2. Если предположить, что следующая выборка приблизительно равна *точному* среднему значению всех предыдущих выборок, то мы имеем $x^{(6)} \approx 1.72$. У нас получится $\hat{x}_6 \approx 1.354$ для скорости обучения 0.1 и $\hat{x}_6 \approx 1.823$ для скорости обучения 0.5.

В обоих этих случаях, если предположить, что следующая выборка равна среднему значению всех предыдущих выборок, то оценка с использованием скорости обучения 0.5 является более точной.

Упражнение 17.3. Рассмотрим применение Q-обучения с аппроксимацией функции к задаче с непрерывным пространством действий путем дискретизации этого пространства. Предположим, что пространство непрерывного действия представлено в $\mathbb{R}^n$ – например, это может быть робот, у которого n исполнительных механизмов и каждое измерение разбито на m интервалов. Сколько действий находится в полученном дискретном пространстве действий? Подходит ли Q-обучение с аппроксимацией функций для непрерывных многомерных задач?

Решение. Пространство действий с n измерениями и m интервалами в каждом измерении дает нам m^n дискретных действий. Количество дискретных действий экспоненциально возрастает с увеличением n. Даже при маленьком m большие значения n могут быстро привести к огромному количеству действий. Следовательно, Q-обучение с аппроксимацией функций плохо подходит для использования в непрерывных задачах с многомерным пространством действий.

Упражнение 17.4. Какова вычислительная сложность алгоритма Q-обучения, если мы взаимодействуем со средой в течение d временных шагов? Какова вычислительная сложность алгоритма SARSA, если мы взаимодействуем со средой в течение d временных шагов?

Решение. Для метода Q-обучения наше правило обновления имеет следующий вид:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right).$$

На каждом временном шаге мы должны выполнять поиск максимума по действиям, поэтому для d временных шагов сложность Q-обучения составляет $O(d|\mathcal{A}|)$. Для SARSA правило обновления выглядит так:

$$Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma Q(s', a') - Q(s, a)). \tag{17.23}$$

В данном случае на каждом временном шаге, в отличие от Q-обучения, нам не нужно выполнять поиск максимума по действиям, поэтому для d временных шагов сложность SARSA составляет просто $O(d)$.

Упражнение 17.5. Какова вычислительная сложность алгоритма SARSA в пересчете на кортеж опыта (s_t, a_t, r_t, s_{t+1}) по сравнению с алгоритмом SARSA(λ)?

Решение. Для SARSA наше правило обновления выглядит так:

$$Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma Q(s', a') - Q(s, a)). \tag{17.24}$$

Следовательно, для каждого кортежа опыта получается сложность $O(1)$. Для SARSA(λ) правила обновления таковы:

$$\delta \leftarrow r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t);$$

$$N(s_t, a_t) \leftarrow N(s_t, a_t) + 1;$$

$$Q(s, a) \leftarrow Q(s, a) + \alpha\delta N(s, a) \quad \text{для всех } s, a;$$

$$N(s, a) \leftarrow \gamma\lambda N(s, a) \quad \text{для всех } s, a.$$

Для каждого кортежа опыта нам нужно вычислить δ и увеличить счетчик посещений (s_t, a_t); оба действия имеют вычислительную сложность $O(1)$. Однако нам нужно обновить как функцию полезности действия, так и количество посещений для всех состояний и действий. Оба этих обновления имеют вычислительную сложность $O(|\mathcal{S}||\mathcal{A}|)$. Таким образом, вычислительная сложность на кортеж опыта больше у алгоритма SARSA(λ). Однако SARSA(λ) часто обходится меньшим количеством кортежей опыта, поэтому выбор по критерию вычислительной сложности не всегда однозначен.

Упражнение 17.6. Как ведет себя $Q(\lambda)$ в пределе при $\lambda \to 0$? Как ведет себя $Q(\lambda)$ в пределе при $\lambda \to 1$?

Решение. Для $Q(\lambda)$ мы применяем следующие правила обновления:

$$\delta \leftarrow r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t);$$

$$N(s_t, a_t) \leftarrow N(s_t, a_t) + 1;$$

$$Q(s, a) \leftarrow Q(s, a) + \alpha\delta N(s, a) \quad \text{для всех } s, a;$$

$$N(s, a) \leftarrow \gamma\lambda N(s, a) \quad \text{для всех } s, a.$$

В пределе при $\lambda \to 0$ для нашей первой итерации мы вычисляем ошибку временной разницы δ и увеличиваем количество посещений $N(s_t, a_t)$. В обновлении функции полезности действия единственный ненулевой член $N(s, a)$ – это $N(s_t, a_t)$, поэтому мы выполняем обновление $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\delta N(s_t, a_t)$. Наконец, мы обнуляем все счетчики посещений. Из этого следует, что в пределе при $\lambda \to 0$ у нас нет следов приемлемости, и мы выполняем простое обновление по методу Q-обучения.

В пределе при $\lambda \to 1$ счетчики посещений будут накапливаться, и мы получим полные следы приемлемости, распределяющие вознаграждение по всем ранее посещенным парам состояние-действие.

Упражнение 17.7. Вычислите $Q(s, a)$ с помощью алгоритма SARSA(λ) после следования по траектории

$(s_1, a_R, 0, s_2, a_R, 0, s_3, a_L, 10, s_2, a_R, 4, s_1, a_R)$.

Будем использовать параметры $\alpha = 0.5$, $\lambda = 1$, $\gamma = 0.9$, при этом начальные значения функции полезности действия и количество посещений везде равны нулю. Предположим, что $\mathcal{S} = \{s_1, s_2, s_3, s_4\}$ и $\mathcal{A} = \{a_L, a_R\}$.

Решение. Для SARSA(λ) правила обновления таковы:

$\delta \leftarrow r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$;

$N(s_t, a_t) \leftarrow N(s_t, a_t) + 1$;

$Q(s, a) \leftarrow Q(s, a) + \alpha\delta N(s, a)$ для всех s, a;

$N(s, a) \leftarrow \gamma\lambda N(s, a)$ для всех s, a.

Для первого кортежа опыта мы имеем $\delta = 0 + 0.9 \times 0 - 0 = 0$, увеличиваем количество посещений $N(s_1, a_R)$, функцию полезности действия не меняем, так как $\delta = 0$, и обновляем подсчеты. После этого получаем следующее:

$Q(s, a)$	s_1	s_2	s_3	s_4
a_L	0	0	0	0
a_R	0	0	0	0

$N(s, a)$	s_1	s_2	s_3	s_4
a_L	0	0	0	0
a_R	0.9	0	0	0

Для второго кортежа опыта мы имеем $\delta = 0$, функцию полезности действия не меняем, так как $\delta = 0$, и обновляем подсчеты. После этого получаем следующее:

$Q(s, a)$	s_1	s_2	s_3	s_4
a_L	0	0	0	0
a_R	0	0	0	0

$N(s, a)$	s_1	s_2	s_3	s_4
a_L	0	0	0	0
a_R	0.81	0.9	0	0

Для третьего кортежа опыта мы имеем δ = 10, увеличиваем количество посещений на $N(s_3, a_L)$, обновляем функцию полезности действия и подсчеты. После этого получаем следующее:

$Q(s, a)$	s_1	s_2	s_3	s_4
a_L	0	0	0	0
a_R	4.05	4.5	0	0

$N(s, a)$	s_1	s_2	s_3	s_4
a_L	0	0	0.9	0
a_R	0.729	0.81	0	0

Для четвертого кортежа опыта мы имеем $\delta = 4 + 0.9 \times 4.05 - 4.5 = 3.145$, увеличиваем количество посещений на $N(s_2, a_R) = 0.81 + 1 = 1.81$, обновляем функцию полезности действия и подсчеты. После этого окончательно получаем:

$Q(s, a)$	s_1	s_2	s_3	s_4
a_L	0	0	6.415	0
a_R	5.196	7.346	0	0

$N(s, a)$	s_1	s_2	s_3	s_4
a_L	0	0	0.81	0
a_R	0.656	1.629	0	0

18. Имитационное обучение

В предыдущих главах предполагалось, что функция вознаграждения известна или ее модель обучается при взаимодействии с окружающей средой. В некоторых случаях бывает проще продемонстрировать желаемое поведение на примере эксперта, чем строить функцию вознаграждения. В этой главе обсуждаются алгоритмы *имитационного обучения*, когда желаемое поведение изучается путем наблюдения за экспертом и повторения его действий. Мы рассмотрим различные методы, начиная от очень простых методов максимизации правдоподобия и заканчивая более сложными итеративными методами, включающими обучение с подкреплением[1].

18.1. Поведенческое копирование

Простая форма имитационного обучения состоит в том, чтобы рассматривать его как задачу обучения с учителем. Этот метод, получивший название *поведенческого копирования* (behavioral cloning)[2], обучает стохастическую стратегию $\pi_{\boldsymbol{\theta}}$, параметризованную вектором $\boldsymbol{\theta}$, стремясь максимизировать правдоподобие действий из набора данных $\mathcal{D}$ о состояниях-действиях, полученного от эксперта:

$$\max_{\boldsymbol{\theta}} \prod_{(s,a)\in\mathcal{D}} \pi_{\boldsymbol{\theta}}(a|s). \tag{18.1}$$

Как и в предыдущих главах, мы можем преобразовать максимизацию произведения по $\pi_{\boldsymbol{\theta}}(a|s)$ в сумму по $\log \pi_{\boldsymbol{\theta}}(a|s)$.

Оценка максимального правдоподобия $\boldsymbol{\theta}$ вычисляется аналитически в зависимости от того, как представлено условное распределение $\pi_{\boldsymbol{\theta}}(a|s)$. Например, если мы используем дискретную условную модель (раздел 2.4), $\boldsymbol{\theta}$ будет состоять из подсчетов $N(s, a)$ для набора $\mathcal{D}$ и $\pi_{\boldsymbol{\theta}}(a|s) = N(s, a)/\sum_a N(s, a)$. В примере 18.1 дискретная условная модель применяется к данным из задачи о горной машине.

---

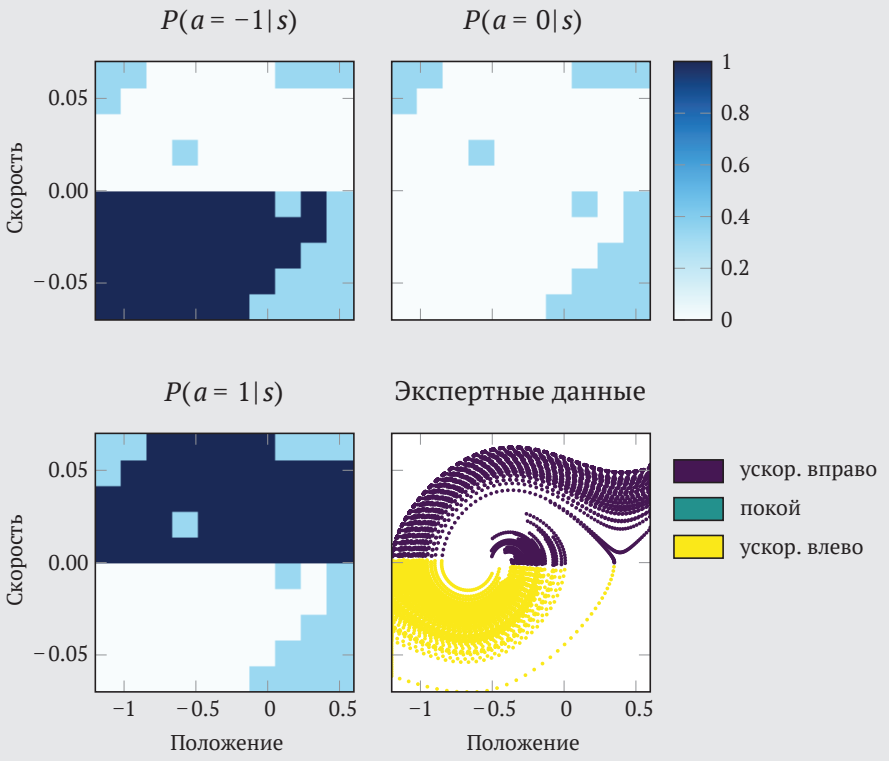
[1] Дополнительные методы и приложения рассмотрены в статье A. Hussein, M. M. Gaber, E. Elyan, C. Jayne, *Imitation Learning: A Survey of Learning Methods*, ACM Computing Surveys, vol. 50, no. 2, pp. 1–35, 2017.

[2] D. A. Pomerleau, *Efficient Training of Artificial Neural Networks for Autonomous Navigation*, Neural Computation, vol. 3, no. 1, pp. 88–97, 1991.

При наличии факторизованного представления стратегии можно использовать байесовскую сеть для отображения совместного распределения по переменным состояния и действия. На рис. 18.1 показан пример такого представления. Мы можем узнать из данных $\mathcal{D}$ как структуру (глава 5), так и параметры (глава 4). Затем можно вывести распределение по действиям с учетом текущего состояния, используя один из алгоритмов вывода, рассмотренных ранее (глава 3).

Пример 18.1. Демонстрация поведенческого копирования применительно к задаче о горной машине. Области голубого цвета не содержат обучающие данные. Наличие подобных областей снижает качество стратегии, когда агент сталкивается с такими состояниями

Рассмотрим использование поведенческого копирования на основе экспертных данных для задачи о горной машине (приложение F.4). У нас есть 10 развертываний из экспертной стратегии. Наша задача – подогнать условное распределение к экспертным данным и построить визуализацию результатов. Непрерывные траектории были разбиты на 10 отрезков по положению и по скорости.



Пространство состояний не полностью покрыто экспертными данными, что характерно для задач имитационного обучения. Полученная стратегия может хорошо работать при использовании в областях с покрытием, но она назначает равномерное распределение вероятности действий в областях без покрытия. Даже стартовав в области с покрытием, мы можем довольно быстро оказаться в области без покрытия из-за стохастичности среды.

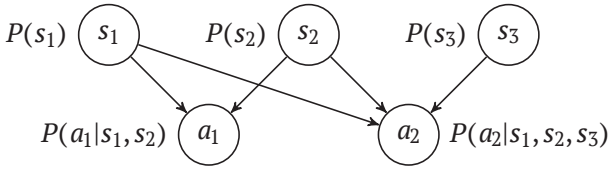


Рис. 18.1. Совместное распределение по переменным состояния и действия, представленное в виде байесовской сети. Мы можем применить алгоритм вывода для создания распределения по действиям, исходя из текущих значений переменных состояния

Существуют и другие представления π_θ. Например, можно использовать нейронную сеть, где вход соответствует значениям переменных состояния, а выход – параметрам распределения по пространству действий. Если наше представление дифференцируемо, как в случае нейронных сетей, можно попытаться оптимизировать уравнение (18.1), используя градиентное восхождение. Этот подход реализован в алгоритме 18.1.

Алгоритм 18.1. Метод обучения параметрической стохастической стратегии на экспертных примерах в виде набора кортежей состояния-действия `D`. Вектор параметров стратегии `θ` итеративно улучшается путем максимизации логарифмического правдоподобия действий при заданных состояниях. Для поведенческого копирования необходимо указать размер шага `α`, количество итераций `k_max` и градиент логарифмического правдоподобия `∇logπ`

```
struct BehavioralCloning
    α      # размер шага
    k_max  # количество итераций
    ∇logπ  # логарифмический градиент правдоподобия
end

function optimize(M::BehavioralCloning, D, θ)
    α, k_max, ∇logπ = M.α, M.k_max, M.∇logπ
    for k in 1:k_max
        ∇ = mean(∇logπ(θ, a, s) for (s,a) in D)
        θ += α*∇
    end
    return θ
end
```

Чем ближе обучающие примеры экспертов к оптимальной стратегии, тем лучше будет работать стратегия, полученная методом поведенческого копирования[3]. Однако поведенческое копирование страдает от *кумулятивных ошибок* (cascading error). Как показано в примере 18.2, небольшие неточности накапливаются во время развертывания и в конечном итоге приводят к состояниям, которые плохо представлены в обучающих данных, что приводит к худшим решениям и в конечном итоге к недействительным или незнакомым ситуациям. Хотя поведенческое копирование привлекательно благодаря своей простоте, кумулятивные ошибки приводят к тому, что метод плохо работает для многих задач, особенно когда стратегии должны работать в течение длительного периода времени.

Пример 18.2. Краткий пример проблемы обобщения, присущей методу поведенческого копирования

Допустим, метод поведенческого копирования применяется для обучения системы вождения автономного гоночного автомобиля. Гонщик-человек проводит экспертные демонстрации заездов по трассе. Будучи экспертом, водитель никогда не выезжает на газон и не проезжает слишком близко к ограждению. Модель, обученная с помощью поведенческого копирования, не будет иметь никакой информации о том, как себя вести, оказавшись на газоне или соприкоснувшись с ограждением. Как следствие она не сможет вернуть гоночный автомобиль на трассу.

18.2. Агрегация наборов данных

Одним из способов решения проблемы кумулятивных ошибок является исправление полученной стратегии при помощи дополнительных экспертных данных. В методах *последовательной интерактивной демонстрации* (sequential interactive demonstration) применяется чередование между запросом решений эксперта в ситуациях, сгенерированных обученной стратегией и использованием полученных данных для улучшения стратегии.

Один из методов последовательной интерактивной демонстрации называется *агрегацией набора данных* (data set aggregation, DAgger) (алгоритм 18.2)[4]. Он начинается с обучения стохастической стратегии по методу поведенческого копирования. Затем на основе полученной стратегии выполняют несколько развертываний из начального распределения состояний b, которые потом передают эксперту. Эксперт возвращает правильные действия для каждого со-

[3] U. Syed and R. E. Schapire, *A Reduction from Apprenticeship Learning to Classification*, in Advances in Neural Information Processing Systems (NIPS), 2010.

[4] S. Ross, G. J. Gordon, J. A. Bagnell, *A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning*, in International Conference on Artificial Intelligence and Statistics (AISTATS), vol. 15, 2011.

стояния. Новые данные объединяются с предыдущим набором данных, и на основе дополненного набора обучается новая стратегия. Этот процесс проиллюстрирован примером 18.3.

Алгоритм 18.2. Реализация метода агрегации набора данных DAgger для изучения стохастической параметрической стратегии на примерах экспертов. Этот метод использует начальный набор данных кортежей состояния-действия `D`, стохастическую параметрическую стратегию `πθ(θ, s)`, MDP $\mathcal{P}$, который определяет функцию перехода и начальное распределение по состояниям `b`. Для улучшения стратегии в каждой итерации используется поведенческое копирование (алгоритм 18.1).

Экспертная стратегия `πE` помечает траектории, выбранные из последней изученной стратегии, для дополнения набора данных. В исходной научной публикации траектории генерировались путем стохастического смешивания экспертной стратегии. Фактически данная реализация метода является исходным алгоритмом DAgger с нулевым значением параметра смешивания.

На практике экспертной стратегии может не быть, и в таком случае обращения к ней заменяют запросами к эксперту-человеку

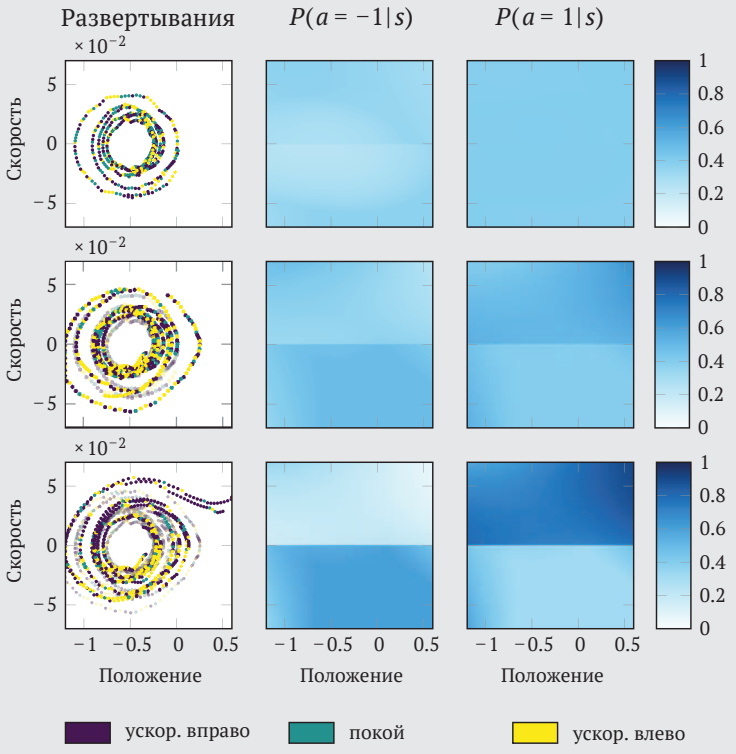
```
struct DataSetAggregation
    𝒫     # задача с неизвестной функцией вознаграждения
    bc    # структура поведенческого копирования
    k_max # количество итераций
    m     # количество развертываний на итерацию
    d     # глубина развертывания
    b     # начальное распределение по состояниям
    πE    # эксперт
    πθ    # параметрическая стратегия
end

function optimize(M::DataSetAggregation, D, θ)
    𝒫, bc, k_max, m = M.𝒫, M.bc, M.k_max, M.m
    d, b, πE, πθ = M.d, M.b, M.πE, M.πθ
    θ = optimize(bc, D, θ)
    for k in 2:k_max
        for i in 1:m
            s = rand(b)
            for j in 1:d
                push!(D, (s, πE(s)))
                a = rand(πθ(θ, s))
                s = rand(𝒫.T(s, a))
            end
        end
        θ = optimize(bc, D, θ)
    end
    return θ
end
```

Пример 18.3. Применение алгоритма DAgger к задаче о горной машине. Итерации изображены по порядку сверху вниз. Траектории накапливаются в наборе данных с течением времени. Поведение агента улучшается с каждой итерацией

Рассмотрим использование алгоритма DAgger для обучения стратегии решения задачи о горной машине, когда вознаграждение не наблюдается. Мы используем экспертную стратегию, которая предписывает ускоряться в направлении движения. В этом примере обучаем стратегию, используя следующие признаки:

$$\mathbf{f}(s) = [\mathbf{1}[v > 0], \mathbf{1}[v < 0], x, x^2, v, v^2, xv],$$

где x и v – положение и скорость автомобиля.



Траектории окрашены в соответствии с действием. На первой итерации агент ведет себя хаотично, демонстрируя неспособность продвинуться к цели ($x \geqslant 0.6$). В ходе дополнительных итераций агент учится имитировать экспертную стратегию ускорения в направлении движения. Это поведение проявляется в новых траекториях, направленных по спирали наружу, и в стратегии, которая приписывает высокое правдоподобие $a = 1$, когда $v > 0$, и $a = -1$, когда $v < 0$.

Интерактивные примеры экспертных действий пошагово создают набор данных, более полно охватывающий области пространства состояний, с которыми агент может столкнуться. С каждой итерацией вновь добавляемые экспертные примеры составляют все меньшую часть набора данных, что приводит к меньшим изменениям стратегии. Хотя последовательное интерактивное обучение может хорошо работать на практике, сходимость не гарантируется. Можно показать, что перемешивание с экспертной стратегией обеспечивает сходимость. Этому вопросу посвящен следующий раздел.

18.3. Итеративное обучение путем стохастического смешивания

Последовательные интерактивные методы могут итеративно формировать стратегию путем стохастического перемешивания вновь обученных стратегий. Одним из таких методов является *итеративное обучение путем стохастического перемешивания* (stochastic mixing iterative learning, SMILe) (алгоритм 18.3)[5]. Этот метод использует поведенческое копирование в каждой итерации, но смешивает новую обученную стратегию с предыдущими.

Алгоритм 18.3. Реализация метода SMILe для обучения стохастической параметрической стратегии на основе экспертных примеров для MDP $\mathcal{P}$. Алгоритм последовательно подмешивает новые компонентные стратегии с убывающими весами, одновременно снижая вероятность действий в соответствии с экспертной стратегией. Метод возвращает вероятности `Ps` и параметризации `θs` для компонентных стратегий

```
struct SMILe
    𝒫     # задача с неизвестной функцией вознаграждения
    bc    # структура поведенческого клонирования
    k_max # количество итераций
    m     # количество развертываний на итерацию
    d     # глубина развертывания
    b     # начальное распределение по состояниям
    β     # скалярный параметр смешивания (e.g., d^-3)
    πE    # экспертная стратегия
    πθ    # параметрическая стратегия
end

function optimize(M::SMILe, θ)
    𝒫, bc, k_max, m = M.𝒫, M.bc, M.k_max, M.m
    d, b, β, πE, πθ = M.d, M.b, M.β, M.πE, M.πθ
    𝒜, T = 𝒫.𝒜, 𝒫.T
    θs = []
    π = s -> πE(s)
```

---

[5] S. Ross, J. A. Bagnell, *Efficient Reductions for Imitation Learning*, in International Conference on Artificial Intelligence and Statistics (AISTATS), 2010.

```
    for k in 1:k_max
        # выполняем последнюю стратегию π для формирования нового набора D
        D = []
        for i in 1:m
            s = rand(b)
            for j in 1:d
                push!(D, (s, πE(s)))
                a = π(s)
                s = rand(T(s, a))
            end
        end
        # обучаем новый классификатор стратегии
        θ = optimize(bc, D, θ)
        push!(θs, θ)
        # вычисляем новую смесь стратегий
        Pπ = Categorical(normalize([(1-β)^(i-1) for i in 1:k],1))
        π = s -> begin
            if rand() < (1-β)^(k-1)
                return πE(s)
            else
                return rand(Categorical(πθ(θs[rand(Pπ)], s)))
            end
        end
    end
    Ps = normalize([(1-β)^(i-1) for i in 1:k_max],1)
    return Ps, θs
end
```

Начнем с экспертной стратегии $\pi^{(1)} = \pi_E$[6]. На каждой итерации мы выполняем последнюю стратегию $\pi^{(k)}$ и создаем новый набор данных, запрашивая правильные действия у эксперта. Поведенческое копирование применяется только к этому новому набору данных для обучения новой *компонентной стратегии* (component policy) $\hat{\pi}^{(k)}$. Текущая компонентная стратегия смешивается с компонентными стратегиями из предыдущих итераций для создания новой стратегии $\pi^{(k+1)}$.

Процесс смешивания компонентных стратегий для генерации $\pi^{(k+1)}$ регулируется скалярным параметром смешивания $\beta \in (0, 1)$. Вероятность действия в соответствии с экспертной стратегией равна $(1 - \beta)^k$, а вероятность действия в соответствии с $\pi^{(i)}$ равна $\beta(1 - \beta)^{i-1}$. Эта схема придает больший вес более старым стратегиям в соответствии с гипотезой о том, что старые компонентные стратегии были обучены состояниям, которые могут встретиться с наибольшей вероятностью[7]. С каждой итерацией вероятность действовать в соответствии

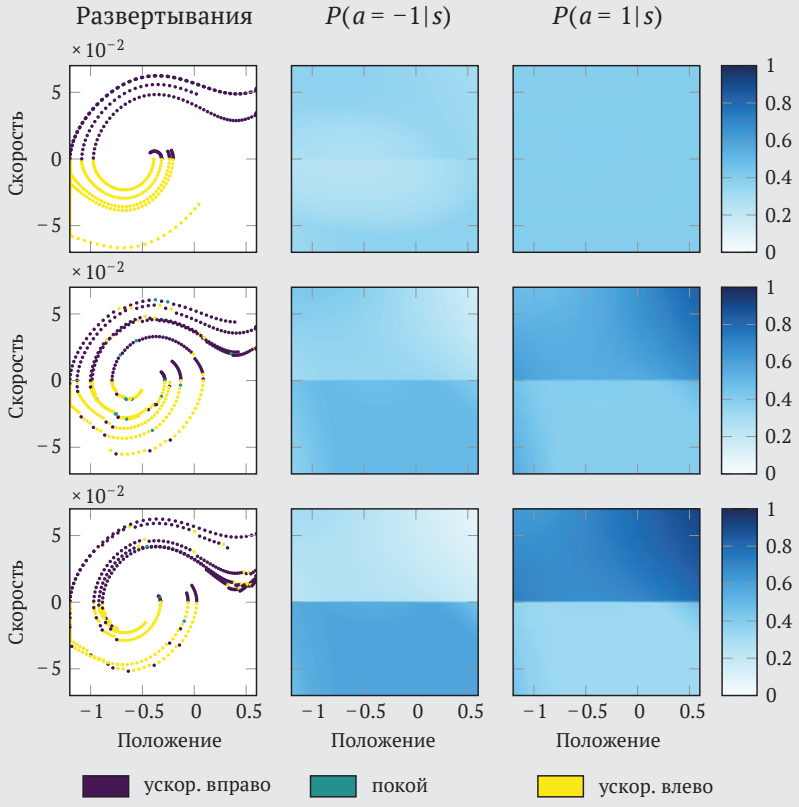
[6] У нас нет явного представления π_E. Оценка π_E требует интерактивного опроса эксперта, как это было сделано в предыдущем разделе.

[7] В SMILe мы действуем в соответствии с нашей последней обученной стратегией. Мы ожидаем, что эта стратегия будет достаточно хорошо совпадать с видением эксперта и давать неверные прогнозы в основном тогда, когда мы отклоняемся от экспертной стратегии. Изученные компонентные стратегии, как правило, должны вносить все меньший и меньший вклад с каждой итерацией, чтобы компенсировать разницу в том, что еще не было изучено.

с исходной экспертной стратегией постепенно снижается до нуля. Обычно используют небольшие значения скалярного параметра смешивания, чтобы агент не отказывался от экспертной стратегии слишком быстро. Пример 18.4 демонстрирует этот подход на примере задачи о горной машине.

Пример 18.4. Использование SMILe для обучения стратегии решения задачи о горной машине. В отличие от DAgger в примере 18.3, SMILe подмешивает экспертные решения к стратегии во время развертывания. Этот экспертный компонент, влияние которого ослабевает с каждой итерацией, позволяет начальным развертываниям лучше продвигаться к цели

Рассмотрим использование SMILe для обучения стратегии решения задачи о горной машине, когда вознаграждение не наблюдается. Мы используем те же признаки, что и для DAgger в примере 18.3. И DAgger, и SMILe с каждой итерацией получают новый набор данных, помеченный экспертом. Вместо того чтобы накапливать больший набор данных, помеченных экспертом, SMILe обучает новый компонент стратегии только на самых свежих данных, смешивая новый компонент стратегии с предыдущими компонентами.



18.4. Обратное обучение с подкреплением с максимальной разницей

На практике часто складывается ситуация, когда у нас нет эксперта, к которому можно было бы обратиться в интерактивном режиме, но вместо этого есть набор обучающих экспертных траекторий. Будем считать, что обучающие экспертные данные $\mathcal{D}$ состоят из m траекторий. Каждая траектория τ в $\mathcal{D}$ содержит развертывание на глубину d. В *обратном обучении с подкреплением* (inverse reinforcement learning, IRL) агент предполагает, что эксперт оптимизирует неизвестную функцию вознаграждения. Агент пытается вывести эту функцию вознаграждения из $\mathcal{D}$. Затем, используя функцию вознаграждения, агент может использовать методы, рассмотренные в предыдущих главах, для получения оптимальной стратегии.

Существуют различные методы обратного обучения с подкреплением. Чаще всего требуется определить параметризацию функции вознаграждения. Обычно предполагают, что эта параметризация является линейной с $R_{\boldsymbol{\varphi}}(s, a) = \boldsymbol{\varphi}^\top \boldsymbol{\beta}(s, a)$, где $\boldsymbol{\beta}(s, a)$ – вектор признаков, а $\boldsymbol{\varphi}$ – вектор весов. В этом разделе мы сосредоточимся на методе, известном как *обратное обучение с подкреплением с максимальной разницей* (maximum margin inverse reinforcement learning)[8], где предполагается, что признаки являются бинарными. Поскольку оптимальные стратегии остаются оптимальными при положительном масштабировании функции вознаграждения, этот метод дополнительно ограничивает весовой вектор так, что $\|\boldsymbol{\varphi}\|_2 \leqslant 1$. Экспертные данные используют бинарные признаки с разной частотой, предпочитая одни и избегая других. Данный метод пытается изучить принцип использования признаков и обучает агента имитировать экспертную частотность использования.

Важной частью этого алгоритма являются рассуждения об ожидаемом доходе при следовании стратегии π с весами $\boldsymbol{\varphi}$ и начальным распределением по состояниям b:

$$\mathop{\mathbb{E}}_{s\sim b}[U(s)] = \mathbb{E}_\tau\left[\sum_{k=1}^{d} \gamma^{k-1} R_{\boldsymbol{\varphi}}(s^{(k)}, a^{(k)})\right] \tag{18.2}$$

$$= \mathbb{E}_\tau\left[\sum_{k=1}^{d} \gamma^{k-1} \boldsymbol{\varphi}^\top \boldsymbol{\beta}(s^{(k)}, a^{(k)})\right] \tag{18.3}$$

$$= \boldsymbol{\varphi}^\top\left(\mathbb{E}_\tau\left[\sum_{k=1}^{d} \gamma^{k-1} \boldsymbol{\beta}(s^{(k)}, a^{(k)})\right]\right) \tag{18.4}$$

$$= \boldsymbol{\varphi}^\top \mu_\pi. \tag{18.5}$$

[8] P. Abbeel, A. Y. Ng, *Apprenticeship Learning via Inverse Reinforcement Learning*, in International Conference on Machine Learning (ICML), 2004.

где τ обозначает траектории, порожденные стратегией π до глубины d. Здесь вводится понятие вектора *ожиданий признаков* (feature expectations) $\boldsymbol{\mu}_\pi$, который представляет собой ожидаемые дисконтированные накопленные значения признаков. Эти ожидания признаков можно оценить по m развертываниям, как реализовано в алгоритме 18.4.

Алгоритм 18.4. Структура обучения с обратным подкреплением и метод оценки вектора ожиданий признаков по развертываниям

```
struct InverseReinforcementLearning
    𝒫  # задача
    b  # начальное распределение по состояниям
    d  # глубина
    m  # количество выборок
    π  # параметрическая стратегия
    β  # отображение бинарных признаков
    μE # экспертные ожидания признаков
    RL # метод обучения с подкреплением
    ε  # устойчивость
end

function feature_expectations(M::InverseReinforcementLearning, π)
    𝒫, b, m, d, β, γ = M.𝒫, M.b, M.m, M.d, M.β, M.𝒫.γ
    μ(τ) = sum(γ^(k-1)*β(s, a) for (k,(s,a)) in enumerate(τ))
    τs = [simulate(𝒫, rand(b), π, d) for i in 1:m]
    return mean(μ(τ) for τ in τs)
end
```

Сначала используют экспертные обучающие примеры для оценки экспертных ожиданий признаков $\boldsymbol{\mu}_E$, а затем находят стратегию, которая максимально точно соответствует этим ожиданиям признаков. На первой итерации мы начинаем с рандомизированной стратегии $\pi^{(1)}$ и оцениваем ее ожидания признаков, обозначаемые как $\boldsymbol{\mu}^{(1)}$. На итерации k мы находим новый вектор $\boldsymbol{\varphi}^{(k)}$, соответствующий функции вознаграждения $R_{\boldsymbol{\varphi}^{(k)}}(s,a) = \boldsymbol{\varphi}^{(k)\top}\boldsymbol{\beta}(s,a)$, такой, что эксперт превосходит все ранее найденные стратегии по наибольшей разнице t:

$$\begin{aligned} &\text{максимизировать } t \\ &\quad t,\boldsymbol{\varphi} \\ &\text{при условии } \boldsymbol{\varphi}^\top\boldsymbol{\mu}_E \geqslant \boldsymbol{\varphi}^\top\boldsymbol{\mu}^{(i)} + t \text{ для } i = 1, \ldots, k-1 \\ &\qquad\qquad ||\boldsymbol{\varphi}||_2 \leqslant 1. \end{aligned} \tag{18.6}$$

Уравнение (18.6) легко решается программно. Затем мы находим новую стратегию $\pi^{(k)}$, используя функцию вознаграждения $R(s, a) = \boldsymbol{\varphi}^{(k)\top}\boldsymbol{\beta}(s, a)$, и создаем новый вектор ожиданий признаков. На рис. 18.2 показан этот процесс максимизации разницы.

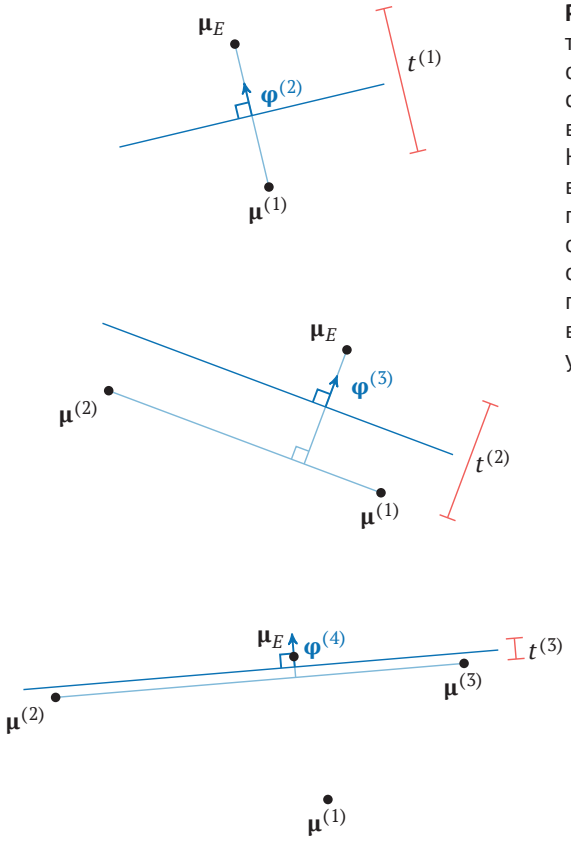


Рис. 18.2. Геометрическая визуализация трех примеров итераций алгоритма обратного обучения с подкреплением с максимальной разницей; порядок выполнения показан сверху вниз. На каждой итерации новый вектор весов указывает в направлении, перпендикулярном гиперплоскости, отделяющей экспертный вектор ожиданий признаков от вектора предыдущей стратегии с максимально возможной разницей. Разница уменьшается с каждой итерацией

Мы повторяем итерации до тех пор, пока разница не станет достаточно малой, то есть $t < \varepsilon$. При достижении сходимости мы можем найти смешанную стратегию, которая пытается обладать ожиданиями признаков, расположенными как можно ближе к таковым в экспертной стратегии:

$$\begin{aligned} &\underset{\boldsymbol{\lambda}}{\text{минимизировать}} \; \|\boldsymbol{\mu}_E - \boldsymbol{\mu}_{\boldsymbol{\lambda}}\|_2 \\ &\text{при условии } \boldsymbol{\lambda} \geqslant 0 \\ &\qquad\qquad \|\boldsymbol{\lambda}\|_1 = 1, \end{aligned} \tag{18.7}$$

где $\boldsymbol{\mu}_{\boldsymbol{\lambda}} = \sum_i \boldsymbol{\mu}^{(i)}$. Веса смеси $\boldsymbol{\lambda}$ определяют комбинацию стратегий, найденных на каждой итерации. С вероятностью λ_i мы следуем стратегии $\pi^{(i)}$. Обратное обучение с подкреплением с максимальной разницей реализовано в алгоритме 18.5.

Алгоритм 18.5. Обратное обучение с подкреплением с максимальной разницей, вычисляющее смешанную стратегию, ожидания признаков которой соответствуют экспертным обучающим данным. Мы используем `JuMP.jl` для решения задач оптимизации с ограничениями. Эта реализация алгоритма требует, чтобы применяемая структура обучения с подкреплением имела весовой вектор `φ`, который можно обновить другими значениями. Метод возвращает стохастические веса `λ` и параметризации `θs` для компонентных стратегий

```
function calc_weighting(M::InverseReinforcementLearning, μs)
    μE = M.μE
    k = length(μE)
    model = Model(Ipopt.Optimizer)
    @variable(model, t)
    @variable(model, φ[1:k] ≥ 0)
    @objective(model, Max, t)
    for μ in μs
        @constraint(model, φ⋅μE ≥ φ⋅μ + t)
    end
    @constraint(model, φ⋅φ ≤ 1)
    optimize!(model)
    return (value(t), value.(φ))
end

function calc_policy_mixture(M::InverseReinforcementLearning, μs)
    μE = M.μE
    k = length(μs)
    model = Model(Ipopt.Optimizer)
    @variable(model, λ[1:k] ≥ 0)
    @objective(model, Min, (μE - sum(λ[i]*μs[i] for i in 1:k))⋅
     (μE - sum(λ[i]*μs[i] for i in 1:k)))
    @constraint(model, sum(λ) == 1)
    optimize!(model)
    return value.(λ)
end

function optimize(M::InverseReinforcementLearning, θ)
    π, ε, RL = M.π, M.ε, M.RL
    θs = [θ]
    μs = [feature_expectations(M, s->π(θ,s))]
    while true
        t, φ = calc_weighting(M, μs)
        if t ≤ ε
            break
        end
        copyto!(RL.φ, φ) # R(s,a) = φ⋅β(s,a)
        θ = optimize(RL, π, θ)
        push!(θs, θ)
        push!(μs, feature_expectations(M, s->π(θ,s)))
    end
    λ = calc_policy_mixture(M, μs)
    return λ, θs
end
```

18.5. Обратное обучение с подкреплением с максимальной энтропией

Метод обратного обучения с подкреплением из предыдущего раздела недоопределен, а это означает, что часто существует несколько стратегий, которые могут давать те же ожидания признаков, что и примеры экспертов. В этом разделе представлено *обратное обучение с подкреплением с максимальной энтропией*, позволяющее избежать подобной неоднозначности, отдавая предпочтение стратегии, которая приводит к распределению по траекториям с максимальной энтропией (приложение А.8)[9]. Эта задача может быть преобразована в поиск наилучших параметров функции вознаграждения $\boldsymbol{\varphi}$ в задаче оценки максимального правдоподобия по имеющимся экспертным данным $\mathcal{D}$.

Любая стратегия π порождает распределение вероятностей по траекториям $P_\pi(\tau)$[10]. Разные стратегии создают разные распределения вероятностей. Мы можем выбрать любое из распределений по траекториям, соответствующих ожиданиям экспертов, но в соответствии с *принципом максимальной энтропии* следует выбирать наименее информативное распределение, что означает распределение с максимальной энтропией[11]. Можно показать, что наименее информативное распределение по траекториям имеет следующий вид:

$$P_{\boldsymbol{\varphi}}(\tau) = \frac{1}{z(\boldsymbol{\varphi})} \exp\left(R_{\boldsymbol{\varphi}}(\tau)\right), \tag{18.8}$$

где $P_{\boldsymbol{\varphi}}(\tau)$ – вероятность траектории τ при заданном параметре вознаграждения $\boldsymbol{\varphi}$ и

$$R_{\boldsymbol{\varphi}}(\tau) = \sum_{k=1}^{d} \gamma^{k-1} R_{\boldsymbol{\varphi}}(s^{(k)}, a^{(k)}) \tag{18.9}$$

является дисконтированным вознаграждением за траекторию. Мы не делаем никаких предположений о параметризации $R_{\boldsymbol{\varphi}}(s^{(k)}, a^{(k)})$, кроме того что она дифференцируема, что позволяет использовать такие представления, как нейронные сети. Скаляр нормализации $Z(\boldsymbol{\varphi})$ гарантирует, что сумма вероятностей равна 1:

---

[9] B. D. Ziebart, A. Maas, J. A. Bagnell, A. K. Dey, *Maximum Entropy Inverse Reinforcement Learning*, in AAAI Conference on Artificial Intelligence (AAAI), 2008.

[10] Для простоты в этом разделе предполагается, что горизонт конечен и что пространства состояний и действий дискретны, что делает $P_{\boldsymbol{\varphi}}(\tau)$ вероятностной мерой. Чтобы распространить обучение с обратным подкреплением с максимальной энтропией как на задачи с непрерывным состоянием, так и на пространства действий, где динамика может быть неизвестна, попробуйте применить *обучение с управляемыми затратами* (guided cost learning). C. Finn, S. Levine, P. Abbeel, *Guided Cost Learning: Deep Inverse Optimal Control via Policy Optimization*, in International Conference on Machine Learning (ICML), 2016.

[11] Введение в этот принцип см. в E. T. Jaynes, *Information Theory and Statistical Mechanics*, Physical Review, vol. 106, no. 4, pp. 620–630, 1957.

$$\mathcal{Z}(\boldsymbol{\varphi}) = \sum_{\tau} \exp\left(R_{\boldsymbol{\varphi}}(\tau)\right). \tag{18.10}$$

Суммирование ведется по всем возможным траекториям.

Итак, допустим, мы выбрали для нашей стратегии определенный класс распределений по траекториям. Теперь подгоняем распределение к нашим траекториям, используя метод максимального правдоподобия, чтобы получить параметры, которые лучше всего описывают имеющиеся данные:

$$\max_{\boldsymbol{\varphi}} f(\boldsymbol{\varphi}) = \max_{\boldsymbol{\varphi}} \sum_{\tau \in \mathcal{D}} \log P_{\boldsymbol{\varphi}}(\tau). \tag{18.11}$$

Мы можем переписать целевую функцию $f(\boldsymbol{\varphi})$ из уравнения (18.11) следующим образом:

$$f(\boldsymbol{\varphi}) = \sum_{\tau \in \mathcal{D}} \log \frac{1}{\mathcal{Z}(\boldsymbol{\varphi})} \exp\left(R_{\boldsymbol{\varphi}}(\tau)\right) \tag{18.12}$$

$$= \left(\sum_{\tau \in \mathcal{D}} R_{\boldsymbol{\varphi}}(\tau)\right) - |\mathcal{D}| \log \mathcal{Z}(\boldsymbol{\varphi}) \tag{18.13}$$

$$= \left(\sum_{\tau \in \mathcal{D}} R_{\boldsymbol{\varphi}}(\tau)\right) - |\mathcal{D}| \log \sum_{\tau} \exp\left(R_{\boldsymbol{\varphi}}(\tau)\right). \tag{18.14}$$

Далее оптимизируем эту целевую функцию с помощью градиентного подъема. Градиент f равен

$$\nabla_{\boldsymbol{\varphi}} f = \left(\sum_{\tau \in \mathcal{D}} \nabla_{\boldsymbol{\varphi}} R_{\boldsymbol{\varphi}}(\tau)\right) - \frac{|\mathcal{D}|}{\sum_{\tau} \exp\left(R_{\boldsymbol{\varphi}}(\tau)\right)} \sum_{\tau} \exp\left(R_{\boldsymbol{\varphi}}(\tau)\right) \nabla_{\boldsymbol{\varphi}} R_{\boldsymbol{\varphi}}(\tau) \tag{18.15}$$

$$= \left(\sum_{\tau \in \mathcal{D}} \nabla_{\boldsymbol{\varphi}} R_{\boldsymbol{\varphi}}(\tau)\right) - |\mathcal{D}| \sum_{\tau} P_{\boldsymbol{\varphi}}(\tau) \nabla_{\boldsymbol{\varphi}} R_{\boldsymbol{\varphi}}(\tau) \tag{18.16}$$

$$= \left(\sum_{\tau \in \mathcal{D}} \nabla_{\boldsymbol{\varphi}} R_{\boldsymbol{\varphi}}(\tau)\right) - |\mathcal{D}| \sum_{\tau} b_{\gamma,\boldsymbol{\varphi}}(s) \sum_{a} \pi_{\boldsymbol{\varphi}}(a|s) \nabla_{\boldsymbol{\varphi}} R_{\boldsymbol{\varphi}}(s,a). \tag{18.17}$$

Если функция вознаграждения является линейной с $R_{\boldsymbol{\varphi}}(s, a) = \boldsymbol{\varphi}^{\top} R_{\boldsymbol{\varphi}}(s, a)$, как в предыдущем разделе, то градиент $\nabla_{\boldsymbol{\varphi}} R_{\boldsymbol{\varphi}}(s, a)$ равен просто $\boldsymbol{\beta}(s, a)$.

Таким образом, для обновления вектора параметров $\boldsymbol{\varphi}$ требуется как дисконтированная частота посещения состояния $b_{\gamma,\boldsymbol{\varphi}}$, так и оптимальная стратегия в соответствии с текущим вектором параметров $\pi_{\boldsymbol{\varphi}}(a|s)$. Оптимальную стратегию можно получить, выполнив обучение с подкреплением. Для вычисления дисконтированной частоты посещений состояния можно применить развертывание или метод динамического программирования.

Воспользуемся методом динамического программирования. Возьмем начальное распределение вероятностей по состояниям $b^{(1)}_{\gamma\boldsymbol{\varphi}} = b(s)$ и будем итеративно двигаться вперед во времени:

$$b^{(k+1)}_{\gamma,\boldsymbol{\varphi}}(s) = \gamma \sum_{a} \sum_{s'} b^{(k)}_{\gamma,\boldsymbol{\varphi}}(s) \pi(a|s) T(s'|s,a). \tag{18.18}$$

Эта версия обучения с обратным подкреплением с максимальной энтропией реализована в алгоритме 18.6.

Алгоритм 18.6. Обратное обучение с подкреплением с максимальной энтропией, которое находит стохастическую стратегию, максимизирующую правдоподобие данных эксперта при распределении по траекториям с максимальной энтропией. Даннный код вычисляет ожидаемые посещения, используя динамическое программирование для всех состояний. Из этого следует, что задача должна быть дискретной

```
struct MaximumEntropyIRL
    𝒫     # задача
    b     # начальное распределение по состояниям
    d     # глубина
    π     # параметрическая стратегия π(θ,s)
    Pπ    # правдоподобие параметрической стратегии π(θ, a, s)
    ∇R    # градиент функции вознаграждения
    RL    # метод обучения с подкреплением
    α     # размер шага
    k_max # количество итераций
end

function discounted_state_visitations(M::MaximumEntropyIRL, θ)
    𝒫, b, d, Pπ = M.𝒫, M.b, M.d, M.Pπ
    𝒮, 𝒜, T, γ = 𝒫.𝒮, 𝒫.𝒜, 𝒫.T, 𝒫.γ
    b_sk = zeros(length(𝒫.𝒮), d)
    b_sk[:,1] = [pdf(b, s) for s in 𝒮]
    for k in 2:d
        for (si′, s′) in enumerate(𝒮)
            b_sk[si′,k] = γ*sum(sum(b_sk[si,k-1]*Pπ(θ, a, s)*T(s, a, s′)
                    for (si,s) in enumerate(𝒮))
                for a in 𝒜)
        end
    end
    return normalize!(vec(mean(b_sk, dims=2)),1)
end

function optimize(M::MaximumEntropyIRL, D, φ, θ)
    𝒫, π, Pπ, ∇R, RL, α, k_max = M.𝒫, M.π, M.Pπ, M.∇R, M.RL, M.α, M.k_max
    𝒮, 𝒜, γ, nD = 𝒫.𝒮, 𝒫.𝒜, 𝒫.γ, length(D)
    for k in 1:k_max
        copyto!(RL.φ, φ) # update parameters
        θ = optimize(RL, π, θ)
        b = discounted_state_visitations(M, θ)
        ∇Rτ = τ -> sum(γ^(i-1)*∇R(φ,s,a) for (i,(s,a)) in enumerate(τ))
        ∇f = sum(∇Rτ(τ) for τ in D) - nD*sum(b[si]*sum(Pπ(θ,a,s)*∇R(φ,s,a)
                    for (ai,a) in enumerate(𝒜))
                for (si, s) in enumerate(𝒮))
        φ += α*∇f
    end
    return φ, θ
end
```

18.6. Генеративно-состязательное имитационное обучение

В *генеративно-состязательном имитационном обучении* (generative adversarial imitation learning, GAIL)[12] мы оптимизируем дифференцируемую параметрическую стратегию π_θ, часто представленную нейронной сетью. Вместо функции вознаграждения мы используем *состязательное обучение* (приложение D.7). Мы обучаем *дискриминатор* $C_\varphi(s, a)$ (обычно это тоже нейронная сеть) возвращать вероятность, которую он присваивает паре состояние-действие, исходя из обученной стратегии. Процесс генеративно-состязательного имитационного обучения включает в себя чередование обучения дискриминатора, чтобы он лучше различал смоделированные и экспертные пары состояние-действие, и обучения стратегии, чтобы ее действия выглядели неотличимыми от действий эксперта. Процесс изображен на рис. 18.3.

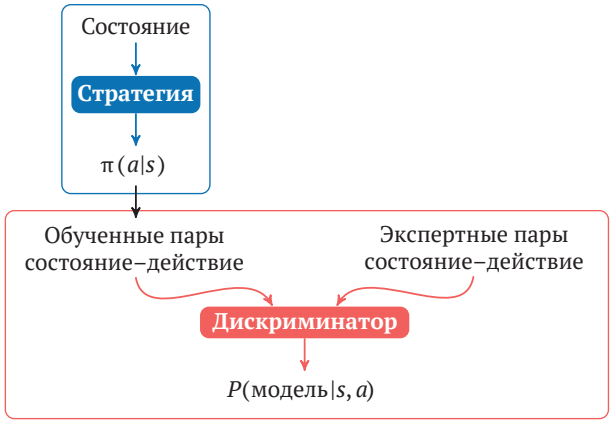


Рис. 18.3. Вместо того чтобы выводить функцию вознаграждения, генеративно-состязательное имитационное обучение оптимизирует дискриминатор, чтобы он лучше различал смоделированные и экспертные пары состояние-действие, и оптимизирует модельную стратегию, чтобы для дискриминатора она была неотличимой от экспертной. Цель состоит в том, чтобы в конечном итоге создать стратегию, максимально похожую на экспертную

Дискриминатор и стратегия преследуют противоположные цели. GAIL пытается найти седловую точку ($\boldsymbol{\theta}$, $\boldsymbol{\varphi}$) отрицательной логарифмической потери для задачи бинарной классификации[13]:

---

[12] J. Ho, S. Ermon, *Generative Adversarial Imitation Learning*, in Advances in Neural Information Processing Systems (NIPS), 2016.

[13] В оригинальной статье также используется следующий член энтропии: $-\lambda\mathbb{E}_{(s,a)\sim\mathcal{D}}[-\log\pi_\theta(a|s)]$.

$$\max_{\boldsymbol{\varphi}} \min_{\boldsymbol{\theta}} \mathbb{E}_{(s,a)\sim\pi_{\boldsymbol{\theta}}}\left[\log(C_{\boldsymbol{\varphi}}(s,a))\right] + \mathbb{E}_{(s,a)\sim\mathcal{D}}\left[\log(1-C_{\boldsymbol{\varphi}}(s,a))\right], \tag{18.19}$$

где $(s, a) \sim \mathcal{D}$ используется для представления выборок из распределения, представленного набором экспертных данных $\mathcal{D}$. Мы можем чередовать градиентное восхождение по $\boldsymbol{\varphi}$, чтобы увеличить значение целевой функции, и оптимизацию стратегии методом доверительной области (раздел 12.4) по $\boldsymbol{\theta}$, чтобы уменьшить значение целевой функции, генерируя необходимые выборки траектории из стратегии для выполнения каждого из этих шагов. Дискриминатор вырабатывает обучающий сигнал для стратегии подобно тому, как вырабатывался бы сигнал вознаграждения, если бы функция вознаграждения была известна.

18.7. Заключение

- Имитационное обучение – это обучение желаемому поведению на примере поведения эксперта без использования функции вознаграждения.
- Одним из видов имитационного обучения является поведенческое копирование, которое создает стохастическую стратегию, максимизирующую условное правдоподобие действий в наборе данных.
- На многократных обращениях к эксперту основаны итеративные методы обучения, такие как агрегация наборов данных и стохастическое перемешивание.
- Обратное обучение с подкреплением основано на выводе функции вознаграждения из экспертных данных и последующем использовании традиционных методов для поиска оптимальной стратегии.
- Обратное обучение с подкреплением с максимальной разницей пытается найти стратегию, которая соответствует частотности бинарных признаков, обнаруженных в наборе экспертных данных.
- Обратное обучение с подкреплением с максимальной энтропией пытается найти наилучший параметр вознаграждения в виде задачи оценки максимального правдоподобия, которая решается с помощью градиентного восхождения.
- Генеративно-состязательное имитационное обучение итеративно оптимизирует дискриминатор и стратегию; дискриминатор пытается различить решения стратегии и эксперта, а стратегия пытается обмануть дискриминатор.

18.8. Упражнения

Упражнение 18.1. Рассмотрим применение поведенческого копирования к отдельной задаче, для которой имеются экспертные данные. Сначала мы определили функцию признаков $\boldsymbol{\beta}(s)$ и представили стратегию с распределением softmax:

$\pi(a|s) \propto \exp(\boldsymbol{\theta}_a{}^\top \boldsymbol{\beta}(s))$.

Затем узнали параметры $\boldsymbol{\theta}_a$ для каждого действия из экспертных данных. Как вы думаете, почем мы отдали предпочтение именно этому методу, а не тому, где напрямую оценивают дискретное распределение для каждого состояния с одним параметром на каждую пару состояние-действие?

Решение. При имитационном обучении мы обычно ограничиваемся относительно небольшим набором экспертных примеров. Распределение $P(a|s)$ имеет $(|\mathcal{A}| - 1)|\mathcal{S}|$ независимых обучаемых параметров, что часто бывает непозволительно много. Экспертные примеры обычно охватывают лишь небольшую часть пространства состояний. Даже если $P(a|s)$ надежно обучить для состояний, охватываемых предоставленным набором данных, результирующая стратегия не будет обучена для других состояний. Использование функции признаков позволяет обобщать незнакомые состояния.

Упражнение 18.2. В разделе 18.1 предлагается использовать метод максимального правдоподобия для обучения стратегии на основе экспертных данных. Этот подход пытается найти параметры стратегии, которые максимизируют правдоподобие, назначенное обучающим примерам. Однако из некоторых задач мы знаем, что присвоение высокой вероятности одному неправильному действию может быть менее плохо, чем присвоение высокой вероятности другому неправильному действию. Например, если в задаче о горной машине эксперт задает ускорение 1, то предсказание ускорения –1 будет худшей ошибкой модели по сравнению с предсказанием нулевого ускорения. Как можно изменить поведенческое копирование, чтобы налагать разные штрафы за разные ошибочные классификации?

Решение. Применим функцию стоимости $C(s, a_{\text{true}}, a_{\text{pred}})$, которая определяет стоимость прогнозирования действия a_{pred} для состояния s, когда эксперт совершает действие a_{true}. Например, в задаче о горной машине мы могли бы использовать функцию

$C(s, a_{\text{true}}, a_{\text{pred}}) = -|a_{\text{true}} - a_{\text{pred}}|$, которая наказывает большие отклонения от экспертного решения сильнее, чем меньшие. Стоимость, связанная с действием эксперта, обычно равна нулю.

Если у нас есть стохастическая стратегия $\pi(a \mid s)$, мы стремимся минимизировать функцию стоимости по всему набору данных:

$$\text{минимизировать}_{\boldsymbol{\theta}} \sum_{(s, a_{\text{true}}) \in \mathcal{D}} \sum_{a_{\text{pred}}} C(s, a_{\text{true}}, a_{\text{pred}}) \pi(a_{\text{pred}}|s).$$

Этот метод называется *классификацией с учетом цены* (cost-sensitive classification)[14]. Одним из преимуществ такой классификации является возможность использовать для обучения стратегии широкий спектр готовых моделей клас-

[14] C. Elkan, *The Foundations of Cost-Sensitive Learning*, in International Joint Conference on Artificial Intelligence (IJCAI), 2001.

сификации, таких как k-ближайшие соседи, метод опорных векторов или деревья решений.

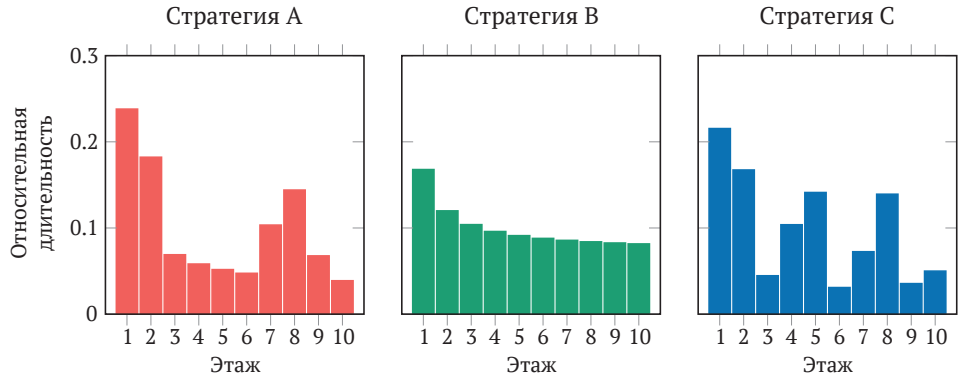
Упражнение 18.3. Приведите пример, когда обратное обучение с подкреплением с максимальной разницей однозначно не определяет оптимальную стратегию.

Решение. Обратное обучение с подкреплением с максимальной разницей извлекает бинарные признаки из экспертных данных и ищет функцию вознаграждения, оптимальная стратегия которой создает траектории с теми же частотами этих бинарных признаков. Нет гарантии, что несколько стратегий не приведут к одинаковым ожиданиям признаков. Например, автономный автомобиль, который меняет только левую полосу движения, может иметь те же частоты смены полосы движения, что и автономный автомобиль, который меняет только правую полосу движения.

Упражнение 18.4. Обратное обучение с подкреплением с максимальной разницей измеряет, насколько стратегия похожа на обучающие примеры экспертов, используя для этого ожидания признаков. Как влияет на эту меру сходства использование небинарных признаков?

Решение. Если мы используем небинарные признаки, то некоторые из них могут стать больше, чем другие, стимулируя агента сопоставлять эти признаки, а не те, которые склонны быть меньше. Но масштаб признаков – не единственная проблема. Даже если все признаки ограничены пределами [0,1], то стратегия, которая последовательно дает $\varphi(s, a)_1 = 0.5$, будет иметь те же ожидания признаков, что и стратегия, которая в одной половине случаев дает $\varphi(s, a)_1 = 0$, а в другой половине случаев $\varphi(s, a)_1 = 1$. В зависимости от того, что кодирует признак, это может привести к очень разным стратегиям. Любой набор непрерывных признаков можно дискретизировать и, таким образом, аппроксимировать набором бинарных признаков.

Упражнение 18.5. Предположим, мы строим систему управления лифтами в высотном здании, которая должна выбирать, на какой этаж отправить лифт. Мы обучили несколько стратегий, чтобы они соответствовали ожиданиям признаков экспертных примеров, например как долго жильцы должны ждать прибытия лифта или как долго они должны ждать, пока доедут до нужного этажа. Мы запускаем несколько развертываний для каждой стратегии и строим график относительной длительности ожидания лифта на каждом этаже. Какую стратегию мы должны предпочесть в соответствии с принципом максимальной энтропии, предполагая, что каждая стратегия в равной степени соответствует ожиданиям признаков?



Решение. В случае задачи о лифте распределения по относительной длительности аналогичны распределениям по траекториям. Применяя принцип максимальной энтропии, мы должны выбрать распределение с наибольшей энтропией. Следовательно, мы выбираем стратегию B, которая, будучи наиболее однородной, имеет наибольшую энтропию.

Упражнение 18.6. Возьмите шаг оптимизации стратегии в генеративно-состязательном имитационном обучении. Перепишите цель в виде функции вознаграждения, чтобы можно было применять традиционные методы обучения с подкреплением.

Решение. Перепишем уравнение (18.19), отбросив члены, зависящие от набора экспертных данных, и изменим знак, чтобы перейти от минимизации по $\boldsymbol{\theta}$ к максимизации по $\boldsymbol{\theta}$ вознаграждения, создав замещающую функцию вознаграждения:

$$\tilde{R}_{\boldsymbol{\varphi}}(s, a) = -\log C_{\boldsymbol{\varphi}}(s, a).$$

Хотя $\tilde{R}_{\boldsymbol{\varphi}}(s, a)$ может сильно отличаться от неизвестной истинной функции вознаграждения, ее можно использовать для направления изученной стратегии в области пространства состояний-действий, подобных тем, которые охвачены экспертом.

Упражнение 18.7. Поясните, как можно изменить генеративно-состязательное имитационное обучение, чтобы дискриминатор принимал траектории, а не пары состояние-действие. Почему это может быть полезно?

Решение. Изменить генеративно-состязательное имитационное обучение таким образом, чтобы дискриминатор выбирал траектории, не составит труда, особенно если траектории имеют фиксированную длину. Набор экспертных данных разбивается на траектории, и обученная стратегия используется для

создания траекторий, как это было раньше. Вместо того чтобы работать с парами состояние-действие, дискриминатор принимает траектории, используя такое представление, как рекуррентная нейронная сеть (приложение D.5), и выдает вероятность классификации. Целевая функция остается практически неизменной:

$$\max_{\boldsymbol{\varphi}} \min_{\boldsymbol{\theta}} \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}}\Big[\log(C_{\boldsymbol{\varphi}}(\tau))\Big] + \mathbb{E}_{\tau \sim \mathcal{D}}\Big[\log(1 - C_{\boldsymbol{\varphi}}(\tau))\Big].$$

Преимущество прогона дискриминатора по всем траекториям заключается в том, что это помогает дискриминатору выявить признаки, которые не очевидны из отдельных пар состояний-действий, и в конечном итоге улучшить стратегии. Например, оперируя отдельными ускорениями и скоростями поворотов для стратегии автономного вождения, дискриминатор получает очень мало полезных обучающих данных. Дискриминатор, обученный на более длинных траекториях, может лучше видеть поведение автомобиля, например агрессивность ускорения и плавность смены полосы движения. Полученная стратегия будет лучше повторять примеры профессионального вождения[15].

[15] Этот подход применяется в статье A. Kuefler, J. Morton, T. A. Wheeler, M. J. Kochenderfer, *Imitating Driver Behavior with Generative Adversarial Networks*, in IEEE Intelligent Vehicles Symposium (IV), 2017.

Часть IV

Неопределенность состояния

В предыдущих главах мы рассматривали неопределенность, отраженную в функции перехода, в контексте результирующего состояния и модели. В этой части мы добавим *неопределенность состояния*. Вместо точного наблюдения за состоянием мы получаем наблюдения, имеющие лишь вероятностную связь с истинным состоянием. Такие задачи можно смоделировать как *частично наблюдаемый марковский процесс принятия решений* (partially observed MDP, POMDP). Общий подход к решению задачи POMDP заключается в выводе распределения вероятностей по убеждениям агента о среде на текущем временном шаге и последующем применении стратегии, которая сопоставляет убеждения с действиями. Мы покажем, как обновить распределение по убеждениям, учитывая прошлую последовательность наблюдений и действий. Это позволяет нам разрабатывать точные методы решения для оптимизации стратегий, основанных на убеждениях агента о среде. К сожалению, задачи POMDP не поддаются прямому решению, за исключением разве что самых мелких. Мы рассмотрим различные офлайн-методы аппроксимации, которые, как правило, гораздо лучше масштабируются, чем точные методы, при решении более крупных задач. Мы также покажем, как расширить некоторые онлайн-методы аппроксимации, рассмотренные ранее в этой книге, на случаи частичной наблюдаемости. Наконец, опишем применение контроллеров с конечным состоянием в качестве альтернативного представления стратегии и обсудим методы их оптимизации для решения POMDP.

19 Убеждения

В данной главе мы рассматриваем POMDP как марковский процесс принятия решений в условиях неопределенности состояния. Агент получает потенциально неточное *наблюдение* за текущим состоянием, а не истинное состояние. Из всей предыдущей последовательности наблюдений и действий агент развивает свое *убеждение* (belief) о среде. В этой главе мы рассмотрим различные алгоритмы для обновления убеждений на основе наблюдений и действий, предпринятых агентом[1]. Мы можем выполнить точное обновление убеждения, если пространство состояний является дискретным или если выполняются определенные линейные допущения по Гауссу. В случаях, когда эти допущения не выполняются, мы можем использовать аппроксимации, основанные на линеаризации или выборке.

19.1. Начальные убеждения

Существуют разные способы представления убеждений агента. В этой главе мы обсудим *параметрическую* форму, в которой *распределение по убеждениям* (belief distribution) задано набором параметров фиксированного семейства распределений, таких как категориальное или многомерное нормальное распределение. Мы также обсудим *непараметрическую* форму, в которой распределение задано точками, выбранными из пространства состояний. От формы представления зависит процедура обновления убеждений, основанных на действиях и наблюдениях агента.

Прежде чем агент предпримет какие-либо действия или сделает какие-либо наблюдения, необходимо сформировать начальное распределение вероятностей по убеждениям. Если у нас есть некоторая предварительная информация о том, где агент может находиться в пространстве состояний, мы можем закодировать ее в *начальном убеждении* (initial belief). Обычно при отсутствии такой информации стараются использовать нечеткие начальные убеждения, чтобы избежать чрезмерной уверенности в том, что агент находится в той области пространства состояний, где его на самом деле может и не быть. Сильное начальное убеждение, сосредоточенное на состояниях, далеких от истины, может привести к плохим оценкам состояния даже после многих наблюдений.

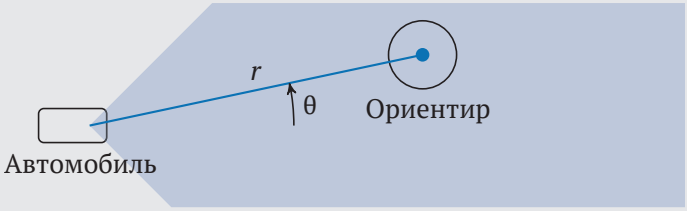
---

[1] Различные методы обновления убеждений обсуждаются в контексте роботизированных приложений в S. Thrun, W. Burgard, D. Fox, *Probabilistic Robotics*. MIT Press, 2006.

Размытое начальное убеждение может вызвать трудности, особенно в непараметрической форме, когда пространство состояний бывает очень сильно разрежено. В некоторых случаях полезно подождать с инициализацией убеждения, пока не будет сделано информативное наблюдение. Например, в задачах навигации роботов мы должны дождаться, пока датчики обнаружат известный ориентир, и затем соответствующим образом инициализировать убеждение. Ориентир формирует связанную область пространства состояний, чтобы мы могли сфокусировать нашу выборку пространства в области, соответствующей наблюдению ориентира. Этот подход продемонстрирован в примере 19.1.

Пример 19.1. Генерация начального непараметрического убеждения на основе наблюдения ориентира. В данном случае автономный автомобиль может находиться в любом месте кольца вокруг ориентира

Допустим, у нас есть автономный автомобиль, оснащенный системой позиционирования, которая использует данные камеры, радара и лидара для отслеживания своего положения. Автомобиль может распознавать уникальный ориентир на расстоянии r с азимутом θ из его текущего положения:



Измерения дальности и азимута содержат гауссов шум с нулевым средним значением с дисперсией ν_r и ν_θ соответственно, и известно, что ориентир находится в точке (x, y). Используя измерения r и θ, мы можем получить распределение по положению автомобиля $(\hat{x}, \hat{y})$ и ориентации $\hat{\psi}$:

$$\hat{r} \sim \mathcal{N}(r, \nu_r) \qquad \hat{\theta} \sim \mathcal{N}(\theta, \nu_\theta) \qquad \hat{\varphi} \sim \mathcal{U}(0, 2\pi)$$

$$\hat{x} \leftarrow x + \hat{r}\cos\hat{\varphi} \qquad \hat{y} \leftarrow y + \hat{r}\sin\hat{\varphi} \qquad \hat{\psi} \leftarrow \hat{\varphi} - \hat{\theta} - \pi,$$

где $\hat{\varphi}$ – угол автомобиля относительно ориентира в глобальной системе отсчета.

19.2. Фильтр дискретных состояний

В POMDP агент не наблюдает напрямую за состоянием среды. Вместо этого агент на каждом временном шаге получает наблюдение, принадлежащее некоторому *пространству наблюдений* (observation space) $\mathcal{O}$. Вероятность наблю-

дения o, при условии что агент предпринял действие a и перешел в состояние s', определяется выражением $O(o|a, s')$. Если $\mathcal{O}$ непрерывно, то $O(o|a, s')$ представляет собой плотность вероятности. На рис. 19.1 показана динамическая сеть принятия решений, относящаяся к POMDP. Алгоритм 19.1 обеспечивает реализацию структуры данных POMDP.

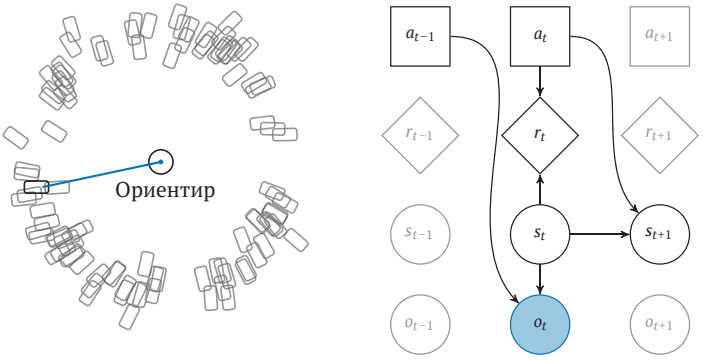


Рис. 19.1. Динамическая сеть принятия решений для задачи POMDP. Как и на рис. 7.1, информационные ребра в узлах действия не показаны

Алгоритм 19.1. Структура данных для POMDP. Мы будем использовать поле `TRO` для выборки следующего состояния, вознаграждения и наблюдения с учетом текущего состояния и действия: `s', r, o = TRO(s, a)`. Комплексный пакет для определения и решения POMDP предложен в работе M. Egorov, Z. N. Sunberg, E. Balaban, T. A. Wheeler, J. K. Gupta, M. J. Kochenderfer, *POMDPs.jl: A Framework for Sequential Decision Making Under Uncertainty*, Journal of Machine Learning Research, vol. 18, no. 26, pp. 1–5, 2017. В математической записи POMDP иногда определяют в виде кортежа, состоящего из различных компонентов MDP и записанного как $(\mathcal{S}, \mathcal{A}, \mathcal{O}, T, R, O, \gamma)$

```
struct POMDP
    γ   # коэффициент дисконтирования
    𝒮   # пространство состояний
    𝒜   # пространство действий
    𝒪   # пространство наблюдений
    T   # функция перехода
    R   # функция вознаграждения
    O   # функция наблюдения
    TRO # выборка перехода, вознаграждения и наблюдения
end
```

Для обновления распределения убеждений агента по текущему состоянию с учетом самого последнего действия и наблюдения применяется разновидность вывода, известная как *рекурсивная байесовская оценка* (recursive Bayesian estimation). Мы будем использовать $b(s)$ для обозначения вероятности (или плотности вероятности для непрерывных пространств состояний), присвоенной состоянию s. Конкретное убеждение b принадлежит пространству убеждений $\mathcal{B}$, которое содержит все возможные убеждения.

Когда пространства состояний и наблюдений конечны, для точного выполнения этого вывода можно использовать *фильтр дискретных состояний* (discrete state filter). Убеждения для задач с дискретными пространствами состояний записывают с помощью категориальных распределений, где каждому состоянию присваивается вероятностная мера. Это категориальное распределение может быть представлено как вектор длины $|\mathcal{S}|$, который часто называют *вектором убеждений* (belief vector). В случаях, когда b можно рассматривать как вектор, мы будем использовать обозначение $\mathbf{b}$. В этом случае $\mathcal{B} \subset \mathbb{R}^{|\mathcal{S}|}$. Иногда $\mathcal{B}$ называют *симплексом вероятности*, или *симплексом представлений*.

Поскольку вектор убеждений представляет собой распределение вероятностей, его элементы должны быть строго неотрицательными и в сумме давать 1:

$$\text{b(s)} \geqslant 0 \text{ для всех } \text{s} \in \mathcal{S} \qquad \sum_s b(s) = 1. \tag{19.1}$$

В векторной форме условия выглядят так:

$$\mathbf{b} \geqslant 0 \qquad \mathbf{1}^\top \mathbf{b} = 1. \tag{19.2}$$

Пространство убеждений для POMDP с тремя состояниями показано на рис. 19.2. Дискретная задача POMDP рассмотрена в примере 19.2.

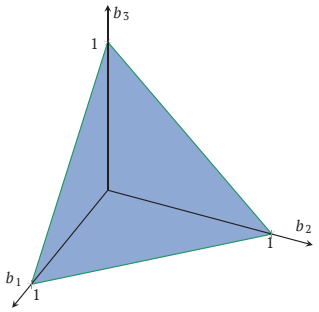
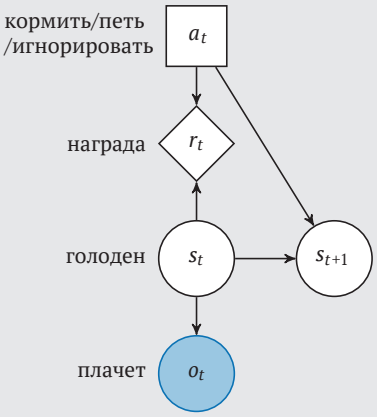


Рис. 19.2. Набор допустимых векторов убеждений для задач с тремя состояниями. Хотя пространство состояний дискретно, пространство убеждений непрерывно

Пример 19.2. Задача о плачущем ребенке – это простая POMDP, которую часто используют для демонстрации принятия решений в условиях неопределенности состояния



Задача о плачущем ребенке – это простая POMDP с двумя состояниями, тремя действиями и двумя наблюдениями. Наша цель – заботиться о ребенке, и мы делаем это, выбирая на каждом временном шаге одно из трех действий: накормить его, спеть ему колыбельную или проигнорировать.

Ребенок со временем становится голодным. Мы не можем напрямую наблюдать, голоден ли ребенок, а вместо этого наблюдаем, плачет ли ребенок. Голодный ребенок плачет 80 % времени, а сытый ребенок плачет 10 % времени. Пение колыбельной возвращает идеальное наблюдение. Пространства состояния, действия и наблюдения таковы:

$\mathcal{S}$ = {сыт, голоден}

$\mathcal{A}$ = {кормить, петь, игнорировать}

$\mathcal{O}$ = {плачет, тихий}

Динамика перехода такова:

T(сыт | голоден, кормить) = 100 %

T(голоден | голоден, петь) = 100 %

T(голоден | голоден, игнорировать) = 100 %

T(сыт | сыт, кормить) = 100 %

T(голоден | сыт, петь) = 10 %

T(голоден | сыт, игнорировать) = 10 %

Функция вознаграждения возвращает –10, если ребенок голоден, и дополнительное вознаграждение в –5 за кормление ребенка из-за затраченных усилий. Таким образом, кормление голодного ребенка дает награду –15. Пение колыбельной требует усилий и влечет за собой дополнительное вознаг-

раждение –0.5. В этой задаче мы ищем оптимальную стратегию бесконечного горизонта с коэффициентом дисконтирования $\gamma = 0.9$.

Если агент с убеждением b выполняет действие a и получает наблюдение o, новое убеждение b' может быть рассчитано следующим образом из-за предположений о независимости на рис. 19.1:

$$b'(s') = P(s'|b,a,o) \tag{19.3}$$

$$\propto P(o|b,a,s')P(s'|b,a) \tag{19.4}$$

$$= O(o|a,s')P(s'|b,a) \tag{19.5}$$

$$= O(o|a,s')\sum_s P(s'|a,b,s)P(s|b,a) \tag{19.6}$$

$$= O(o|a,s')\sum_s T(s'|s,a)b(s). \tag{19.7}$$

Обновление дискретных убеждений проиллюстрировано в примере 19.3 и реализовано в алгоритме 19.2. Успех обновления убеждений зависит от наличия точных моделей наблюдения и перехода. В тех случаях, когда эти модели вызывают сомнения, обычно рекомендуется использовать упрощенные модели с более размытыми распределениями, чтобы предотвратить чрезмерную уверенность, которая приводит к ненадежности оценок состояния.

Пример 19.3. Обновление дискретных убеждений в задаче о плачущем ребенке

Задача о плачущем ребенке (пример 19.2) предполагает равномерное начальное убеждение: [b(сыт), b(голоден)] = [0.5, 0.5].

Предположим, мы игнорируем ребенка, и ребенок плачет. Обновим наше убеждение согласно уравнению (19.7) следующим образом:

$$b'(\text{сыт}) \propto O(\text{плачет}|\text{игнорировать, сыт})\sum_s T(\text{сыт}|s, \text{игнорировать})b(s)$$

$$\propto 0.1(0.0 \cdot 0.5 + 0.1 \cdot 0.5)$$
$$\propto 0.045;$$

$$b'(\text{голоден}) \propto O(\text{плачет}|\text{игнорировать, голоден})\sum_s T(\text{голоден}|s, \text{игнорировать})b(s)$$

$$\propto 0.8(1.0 \cdot 0.5 + 0.1 \cdot 0.5)$$
$$\propto 0.440.$$

После нормализации наше новое убеждение приблизительно равно [0.0928, 0.9072]. Плачущий ребенок, скорее всего, голоден.

Предположим, мы кормим ребенка, и плач прекращается. Кормление детерминированно вызывает насыщение ребенка, поэтому новое убеждение равно [1, 0].

Наконец, мы поем ребенку колыбельную, и он затихает. Снова воспользуемся уравнением (19.7) для обновления убеждения, в результате чего получается [0.9890, 0.0110]. Сытый ребенок начинает плакать только в 10 % случаев, и этот процент в убеждении еще больше снижается, если не наблюдается плача.

Алгоритм 19.2. Метод, обновляющий дискретное убеждение на основе уравнения (19.7), где `b` – вектор, а $\mathcal{P}$ – модель POMDP. Если определенное наблюдение имеет нулевую вероятность, возвращается равномерное распределение

```
function update(b::Vector{Float64}, 𝒫, a, o)
    𝒮, T, O = 𝒫.𝒮, 𝒫.T, 𝒫.O
    b′ = similar(b)
    for (i′, s′) in enumerate(𝒮)
        po = O(a, s′, o)
        b′[i′] = po * sum(T(s, a, s′) * b[i] for (i, s) in enumerate(𝒮))
    end
    if sum(b′) ≈ 0.0
        fill!(b′, 1)
    end
    return normalize!(b′, 1)
end
```

19.3. Фильтр Калмана

Мы можем адаптировать уравнение (19.7) для работы с непрерывными пространствами состояний следующим образом:

$$b'(s') \propto O(o|a,s')\int T(s'|s,a)b(s)\mathrm{d}s. \tag{19.8}$$

Интегрирование в (19.8) может вызывать затруднения, если не сделать некоторых предположений о форме T, O и b. Особая разновидность фильтра, известного как *фильтр Калмана*[2] (алгоритм 19.3), обеспечивает точное обновление в предположении, что T и O линейны по Гауссу, а b представляет собой гауссово распределение[3]:

[2] Назван в честь венгерско-американского инженера-электрика Рудольфа Э. Калмана (1930–2016), который участвовал в начале разработки этого фильтра.

[3] R. E. Kálmán, *A New Approach to Linear Filtering and Prediction Problems*, ASME Journal of Basic Engineering, vol. 82, pp. 35–45, 1960. Подробный обзор фильтра Калмана и его вариантов представлен в работе Y. Bar-Shalom, X. R. Li, T. Kirubarajan, *Estimation with Applications to Tracking and Navigation*. Wiley, 2001.

$$T(\mathbf{s}'|\mathbf{s}, \mathbf{a}) = \mathcal{N}(\mathbf{s}'|\mathbf{T}_s\mathbf{s} + \mathbf{T}_a\mathbf{a}, \mathbf{\Sigma}_s); \tag{19.9}$$

$$O(\mathbf{o}|\mathbf{s}') = \mathcal{N}(\mathbf{o}|\mathbf{O}_s\mathbf{s}', \mathbf{\Sigma}_o); \tag{19.10}$$

$$b(\mathbf{s}) = \mathcal{N}(\mathbf{s}|\boldsymbol{\mu}_b, \mathbf{\Sigma}_b). \tag{19.11}$$

Фильтр Калмана начинается с *этапа прогнозирования* (predict step), который использует динамику перехода, чтобы получить предсказанное распределение со следующим средним значением и ковариацией:

$$\boldsymbol{\mu}_p \leftarrow \mathbf{T}_s\boldsymbol{\mu}_b + \mathbf{T}_a\mathbf{a}; \tag{19.12}$$

$$\mathbf{\Sigma}_p \leftarrow \mathbf{T}_s\mathbf{\Sigma}_b\mathbf{T}_s^\top + \mathbf{\Sigma}_s. \tag{19.13}$$

На *этапе обновления* (update step) мы используем это предсказанное распределение с текущим наблюдением, чтобы обновить убеждение:

$$\mathbf{K} \leftarrow \mathbf{\Sigma}_p\mathbf{O}_s^\top(\mathbf{O}_s\mathbf{\Sigma}_p\mathbf{O}_s^\top + \mathbf{\Sigma}_o)^{-1}; \tag{19.14}$$

$$\boldsymbol{\mu}_b \leftarrow \boldsymbol{\mu}_p + \mathbf{K}(\mathbf{o} - \mathbf{O}_s\boldsymbol{\mu}_p); \tag{19.15}$$

$$\mathbf{\Sigma}_b \leftarrow (\mathbf{I} - \mathbf{K}\mathbf{O}_s)\mathbf{\Sigma}_p, \tag{19.16}$$

где $\mathbf{K}$ называется *коэффициентом усиления фильтра Калмана* (Kalman gain).

Алгоритм 19.3. Фильтр Калмана, который обновляет убеждения в виде гауссовых распределений. Текущее представление выражено через μb и Σb, а $\mathcal{P}$ содержит матрицы, которые определяют линейную по Гауссу динамику и модель наблюдения. $\mathcal{P}$ можно определить с использованием составного типа или именованного кортежа

```
struct KalmanFilter
    μb # вектор матожидания
    Σb # матрица ковариации
end

function update(b::KalmanFilter, 𝒫, a, o)
    μb, Σb = b.μb, b.Σb
    Ts, Ta, Os = 𝒫.Ts, 𝒫.Ta, 𝒫.Os
    Σs, Σo = 𝒫.Σs, 𝒫.Σo
    # прогноз
    μp = Ts*μb + Ta*a
    Σp = Ts*Σb*Ts' + Σs
    # обновление
    Σpo = Σp*Os'
    K = Σpo/(Os*Σp*Os' + Σo)
    μb′ = μp + K*(o - Os*μp)
    Σb′ = (I - K*Os)*Σp
    return KalmanFilter(μb′, Σb′)
end
```

Фильтры Калмана часто применяются к системам, которые на самом деле не имеют линейной по Гауссу динамики и наблюдений. Было предложено множество модификаций базового фильтра Калмана, чтобы лучше приспособить его к таким системам[4].

19.4. Расширенный фильтр Калмана

Расширенный фильтр Калмана (extended Kalman filter, EKF) представляет собой простое расширение фильтра Калмана для задач, динамика которых нелинейна с гауссовым шумом:

$$T(\mathbf{s}'|\mathbf{s}, \mathbf{a}) = \mathcal{N}(\mathbf{s}'|\mathbf{f}_T(\mathbf{s}, \mathbf{a}), \boldsymbol{\Sigma}_s); \tag{19.17}$$

$$O(\mathbf{o}|\mathbf{s}') = \mathcal{N}(\mathbf{o}|\mathbf{f}_O(\mathbf{s}'), \boldsymbol{\Sigma}_o), \tag{19.18}$$

где $\mathbf{f}_T(\mathbf{s}, \mathbf{a})$ и $\mathbf{f}_O(\mathbf{s}')$ – дифференцируемые функции.

Точные обновления убеждений посредством нелинейной динамики не гарантируют создания новых гауссовых убеждений, как показано на рис. 19.3. Расширенный фильтр Калмана использует локальную линейную аппроксимацию нелинейной динамики, тем самым создавая новое гауссово убеждение, которое аппроксимирует истинное обновленное убеждение. Мы можем использовать аналогичные уравнения обновления в качестве фильтра Калмана, но должны вычислять матрицы $\mathbf{T}_s$ и $\mathbf{O}_s$ на каждой итерации, основываясь на текущем убеждении.

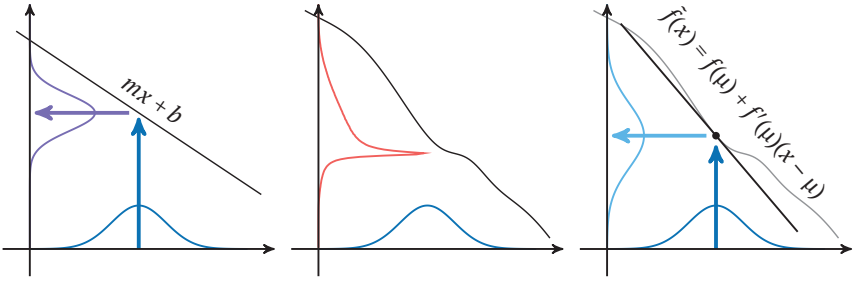


Рис. 19.3. Обновление гауссова убеждения с помощью линейного преобразования (слева) дает другое гауссово распределение. Обновление с помощью нелинейного преобразования (в центре) обычно не дает гауссова распределения. Расширенный фильтр Калмана использует линейную аппроксимацию преобразования (справа), тем самым создавая другое гауссово распределение, которое аппроксимирует апостериорное распределение

[4] S. Thrun, W. Burgard, D. Fox, *Probabilistic Robotics*. MIT Press, 2006.

Локальное линейное приближение к динамике, или *линеаризация*, задается разложениями Тейлора первого порядка в виде якобианов[5]. Для матрицы состояния разложение Тейлора выполняется при $\boldsymbol{\mu}_b$ и текущем действии, тогда как для матрицы наблюдения оно вычисляется при прогнозируемом среднем $\boldsymbol{\mu}_p = \mathbf{f}_T(\boldsymbol{\mu}_b)$.

Расширенный фильтр Калмана реализован в алгоритме 19.4. Это приближение быстрое и хорошо работает на множестве реальных задач. EKF обычно не сохраняет истинное среднее значение и дисперсию апостериорного распределения и не моделирует мультимодальные апостериорные распределения.

Алгоритм 19.4. Расширение фильтра Калмана для задач с нелинейной динамикой по Гауссу. Текущее убеждение представлено средним значением μb и ковариацией Σb. Задача $\mathcal{P}$ задает нелинейную динамику с использованием функции перехода fT и функции наблюдения fO. Якобианы получаются с помощью пакета ForwardDiff.jl

```
struct ExtendedKalmanFilter
    μb # вектор матожидания
    Σb # матрица ковариации
end

import ForwardDiff: jacobian
function update(b::ExtendedKalmanFilter, 𝒫, a, o)
    μb, Σb = b.μb, b.Σb
    fT, fO = 𝒫.fT, 𝒫.fO
    Σs, Σo = 𝒫.Σs, 𝒫.Σo
    # прогноз
    μp = fT(μb, a)
    Ts = jacobian(s->fT(s, a), μb)
    Os = jacobian(fO, μp)
    Σp = Ts*Σb*Ts' + Σs
    # обновление
    Σpo = Σp*Os'
    K = Σpo/(Os*Σp*Os' + Σo)
    μb′ = μp + K*(o - fO(μp))
    Σb′ = (I - K*Os)*Σp
    return ExtendedKalmanFilter(μb′, Σb′)
end
```

19.5. Сигма-точечный фильтр Калмана

Сигма-точечный фильтр Калмана, который известен также под названием «фильтр Калмана без запаха» (unscented Kalman filter, UKF)[6], является еще

---

[5] Якобиан многомерной функции $\mathbf{f}$ с n входами и m выходами представляет собой матрицу размера $m \times n$, где (i, j)-й элемент равен $\partial f_i / \partial x_j$.

[6] S. J. Julier, J. K. Uhlmann, *Unscented Filtering and Nonlinear Estimation*, Proceedings of the IEEE, vol. 92, no. 3, pp. 401–422, 2004.

одним расширением фильтра Калмана для нелинейных задач с гауссовым шумом[7]. В отличие от расширенного фильтра Калмана, сигма-точечный фильтр Калмана не использует производные и опирается на детерминированную стратегию выборки для аппроксимации эффекта распределения, претерпевающего преобразование (обычно нелинейное).

Сигма-точечный фильтр Калмана был разработан для оценки эффекта преобразования распределения по $\mathbf{x}$ с помощью нелинейной функции $\mathbf{f}(\mathbf{x})$, создающей распределение по $\mathbf{x}'$. Нам необходимо найти среднее $\boldsymbol{\mu}'$ и ковариацию $\boldsymbol{\Sigma}'$ распределения по $\mathbf{x}'$. Сигма-точечное преобразование позволяет использовать больше информации о $p(\mathbf{x})$, чем среднее значение $\boldsymbol{\mu}$ и ковариация $\boldsymbol{\Sigma}$ распределения по $\mathbf{x}$[8].

Сигма-точечное преобразование (unscented transform) пропускает набор *сигма-точек* S через $\mathbf{f}$ и использует преобразованные точки для аппроксимации среднего $\boldsymbol{\mu}'$ и ковариации $\boldsymbol{\Sigma}'$. Исходное среднее и ковариация строятся с использованием сигма-точек и вектора весов $\mathbf{w}$:

$$\boldsymbol{\mu} = \sum_i w_i \mathbf{s}_i; \tag{19.19}$$

$$\boldsymbol{\Sigma} = \sum_i w_i (\mathbf{s}_i - \boldsymbol{\mu})(\mathbf{s}_i - \boldsymbol{\mu})^\top, \tag{19.20}$$

где i-я сигма-точка $\mathbf{s}_i$ имеет вес w_i. Эти веса должны в сумме равняться 1, чтобы обеспечить несмещенную оценку, но не обязательно все должны быть положительными.

Таким образом, уравнения обновленного среднего значения и ковариационной матрицы, полученные сигма-точечным преобразованием $\mathbf{f}$, имеют вид:

$$\boldsymbol{\mu}' = \sum_i w_i \mathbf{f}(\mathbf{s}_i); \tag{19.21}$$

$$\boldsymbol{\Sigma}' = \sum_i w_i (\mathbf{f}(\mathbf{s}_i) - \boldsymbol{\mu}')(\mathbf{f}(\mathbf{s}_i) - \boldsymbol{\mu}')^\top. \tag{19.22}$$

Обобщенный набор сигма-точек включает среднее $\boldsymbol{\mu} \in \mathbb{R}^n$ и дополнительные $2n$ точек, образованных возмущениями $\boldsymbol{\mu}$ в направлениях, определяемых ковариационной матрицей $\boldsymbol{\Sigma}$[9]:

---

[7] По словам Джеффри К. Ульмана, термин «без запаха» происходит от этикетки на упаковке дезодоранта, которую он увидел на чьем-то столе. Он использовал этот термин из скромности, чтобы не применять название «фильтр Ульмана». IEEE History Center Staff, *Proceedings of the IEEE Through 100 Years: 2000–2009*, Proceedings of the IEEE, vol. 100, no. 11, pp. 3131–3145, 2012.

[8] Нам не обязательно предполагать, что априорное распределение является гауссовым.

[9] Квадратный корень матрицы $\mathbf{A}$ – это матрица $\mathbf{B}$ такая, что $\mathbf{B}\mathbf{B}^\top = \mathbf{A}$. В языке Julia метод `sqrt` создает матрицу $\mathbf{C}$ такую, что $\mathbf{C}\mathbf{C} = \mathbf{A}$, что не одно и то же. Одна общая матрица квадратного корня может быть получена из разложения Холецкого.

$$\mathbf{s}_1 = \boldsymbol{\mu}; \tag{19.23}$$

$$\mathbf{s}_{2i} = \boldsymbol{\mu} + \left(\sqrt{(n+\lambda)\boldsymbol{\Sigma}}\right)_i \quad \text{для } i \text{ в интервале } 1{:}n; \tag{19.24}$$

$$\mathbf{s}_{2i+1} = \boldsymbol{\mu} - \left(\sqrt{(n+\lambda)\boldsymbol{\Sigma}}\right)_i \quad \text{для } i \text{ в интервале } 1{:}n. \tag{19.25}$$

С этими сигма-точками связаны следующие веса:

$$w_i = \begin{cases} \frac{\lambda}{n+\lambda} & \text{для } i = 1 \\ \frac{1}{2(n+\lambda)} & \text{в остальных случаях} \end{cases}. \tag{19.26}$$

Скалярный *показатель рассеивания* (spread parameter) λ определяет, насколько далеко сигма-точки разбросаны от среднего значения[10]. Несколько наборов сигма-точек для различных значений λ показаны на рис. 19.4.

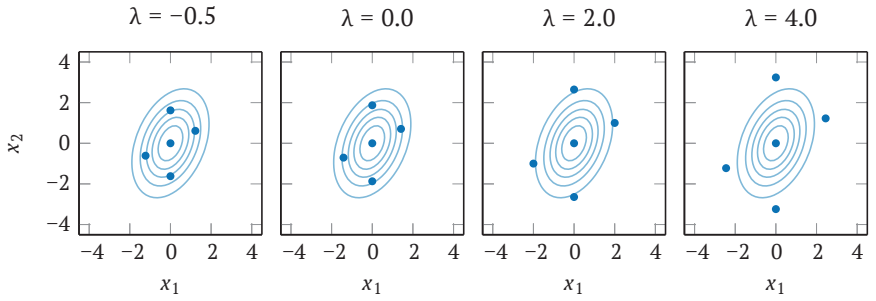


Рис. 19.4. Влияние показателя λ на сигма-точки из уравнения (19.23), полученные для гауссова распределения с нулевым средним значением и ковариацией `Σ = [1 1/2; 1/2 2]`

Сигма-точечный фильтр Калмана выполняет два преобразования: одно для шага прогнозирования и одно для обновления. Реализация этого метода показана в алгоритме 19.5.

Алгоритм 19.5. Сигма-точечный фильтр Калмана – расширение фильтра Калмана для задач с нелинейной по Гауссу динамикой. Текущее убеждение представлено средним значением `μb` и ковариацией `Σb`. Задача $\mathcal{P}$ определяет нелинейную динамику с использованием динамической функции среднего перехода `fT` и динамической функции среднего наблюдения `fO`. Сигма-точки, используемые в сигма-точечных преобразованиях, зависят от показателя рассеивания λ

```
struct UnscentedKalmanFilter
    μb # вектор среднего
```

[10] Обычно используется $\lambda = 2$, что оптимально для согласования четвертого момента гауссовых распределений. Мотивы выбора наборов сигма-точек такой формы приведены в упражнениях 19.13 и 19.14.

```
    Σb # ковариационная матрица
    λ  # показатель рассеивания
end

function unscented_transform(μ, Σ, f, λ, ws)
    n = length(μ)
    Δ = cholesky((n + λ) * Σ).L
    S = [μ]
    for i in 1:n
        push!(S, μ + Δ[:,i])
        push!(S, μ - Δ[:,i])
    end
    S′ = f.(S)
    μ′ = sum(w*s for (w,s) in zip(ws, S′))
    Σ′ = sum(w*(s - μ′)*(s - μ′)' for (w,s) in zip(ws, S′))
    return (μ′, Σ′, S, S′)
end

function update(b::UnscentedKalmanFilter, 𝒫, a, o)
    μb, Σb, λ = b.μb, b.Σb, b.λ
    fT, fO = 𝒫.fT, 𝒫.fO
    n = length(μb)
    ws = [λ / (n + λ); fill(1/(2(n + λ)), 2n)]
    # прогнозирование
    μp, Σp, Sp, Sp′ = unscented_transform(μb, Σb, s->fT(s,a), λ, ws)
    Σp += 𝒫.Σs
    # обновление
    μo, Σo, So, So′ = unscented_transform(μp, Σp, fO, λ, ws)
    Σo += 𝒫.Σo
    Σpo = sum(w*(s - μp)*(s′ - μo)' for (w,s,s′) in zip(ws, So, So′))
    K = Σpo / Σo
    μb′ = μp + K*(o - μo)
    Σb′ = Σp - K*Σo*K'
    return UnscentedKalmanFilter(μb′, Σb′, λ)
end
```

19.6. Парциальный фильтр

При решении дискретных задач с большими пространствами состояний или непрерывных задач с динамикой, которая плохо аппроксимируется линейным по Гауссу предположением о фильтре Калмана, часто приходится прибегать к методам аппроксимации убеждений и их приближенного обновления. Одним из распространенных подходов является использование *парциального фильтра* (particle filter), который рассматривает состояние в убеждении агента как набор состояний[11]. Каждое состояние в приближенном убеждении называется *частицей* (particle).

[11] Учебное пособие по парциальным фильтрам представлено в M. S. Arulampalam, S. Maskell, N. Gordon, T. Clapp, *A Tutorial on Particle Filters for Online Nonlinear / Non-Gaussian Bayesian Tracking*, IEEE Transactions on Signal Processing, vol. 50, no. 2, pp. 174–188, 2002.

Парциальный фильтр инициализируют путем прямого назначения или случайной выборки набора частиц, которые определяют исходное убеждение агента. Обновление убеждения по методу парциального фильтра с m частицами начинается с распространения каждого состояния s_i путем выборки из распределения переходов для получения нового состояния s_i' с вероятностью $T(s_i'|s_i, a)$. Новое убеждение строится путем извлечения m частиц из распространенных состояний, взвешенных в соответствии с функцией наблюдения $w_i = O(o|a, s')$. Эта процедура представлена в алгоритме 19.6. Пример 19.4 иллюстрирует применение парциального фильтра.

Алгоритм 19.6. Механизм обновления убеждения методом парциального фильтра, который обновляет вектор состояний, определяющий убеждение, на основе действия `a` и наблюдения `o`. В приложение G.5 показана реализация функции `SetCategorical` для определения распределений по дискретным множествам

```
struct ParticleFilter
    states # вектор выборок состояния
end

function update(b::ParticleFilter, 𝒫, a, o)
    T, O = 𝒫.T, 𝒫.O
    states = [rand(T(s, a)) for s in b.states]
    weights = [O(a, s′, o) for s′ in states]
    D = SetCategorical(states, weights)
    return ParticleFilter(rand(D, length(states)))
end
```

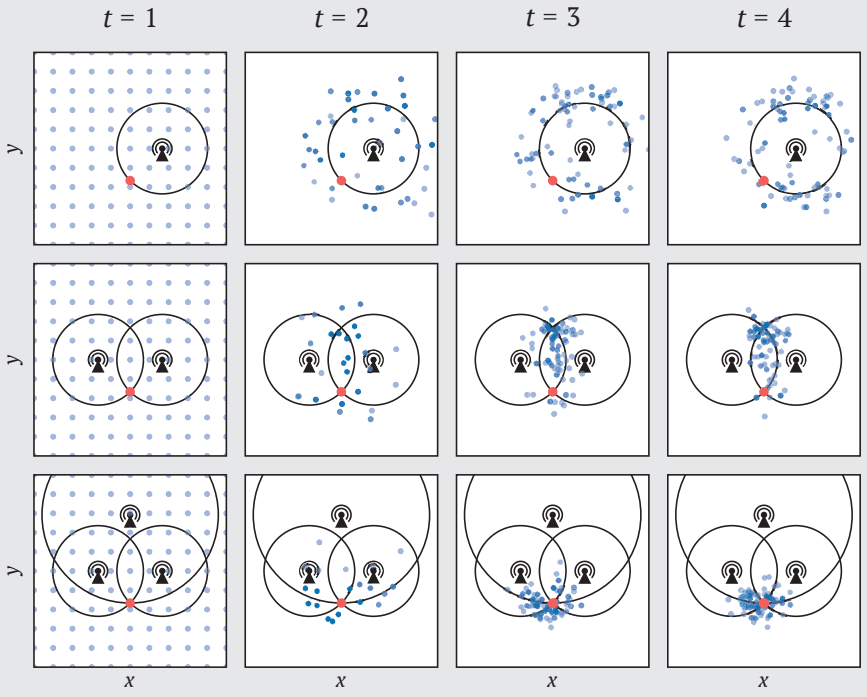
Пример 19.4. Применение парциального фильтра к различным конфигурациям задачи о маяке

Предположим, что мы хотим определить наше местоположение на основе неточных измерений расстояния до радиомаяков, местоположение которых известно. Мы остаемся приблизительно неподвижными в течение нескольких шагов, чтобы собрать независимые измерения. Состояния парциального фильтра – это наши потенциальные местоположения. Мы можем сравнить дальности, которые ожидаем измерить для каждой частицы, с наблюдаемыми дальностями.

Мы предполагаем, что наблюдения за расстоянием до каждого маяка содержат гауссов шум с нулевым средним. Наша функция перехода частиц содержит слагаемое гауссова шума с нулевым средним, поскольку мы остаемся неподвижными лишь приблизительно.

На рисунках ниже показана эволюция парциального фильтра. Ряды соответствуют разным количествам маяков. Красные точки обозначают наше истинное местоположение, а синие точки – частицы. Окружностями показаны

положения, соответствующие незашумленным измерениям расстояния от каждого датчика до маяка.



Для точного определения нашего местоположения необходимы три маяка. Сильной стороной парциального фильтра является его способность обрабатывать мультимодальные распределения, что особенно актуально, когда имеется только один или два маяка.

В задачах с дискретными наблюдениями мы также можем выполнять обновления частичных убеждений с *исключением* (rejection). Мы повторяем описанный ниже процесс m раз, чтобы сгенерировать набор последующих выборок состояния. Сначала случайным образом выбирают некоторое состояние s_i в фильтре, а затем назначают следующее состояние s_i' в соответствии с нашей моделью перехода. Потом генерируют случайное наблюдение в соответствии с нашей моделью наблюдения. Если o_i не соответствует истинному наблюдению o, оно исключается, и процесс генерации новых s_i и o_i повторяют до тех пор, пока наблюдения не совпадут. Этот *парциальный фильтр с исключением* (particle filter with rejection) реализован в алгоритме 19.7.

Алгоритм 19.7. Обновление по методу парциального фильтра с исключением, которое заставляет выборочные состояния соответствовать входному наблюдению o

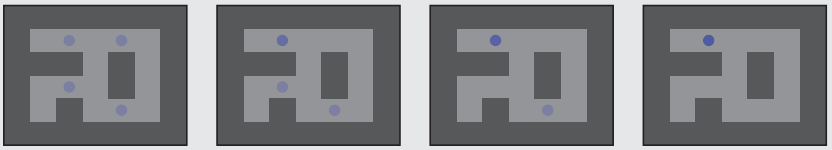
```
struct RejectionParticleFilter
    states # вектор выборок состояния
end

function update(b::RejectionParticleFilter, 𝒫, a, o)
    T, O = 𝒫.T, 𝒫.O
    states = similar(b.states)
    i = 1
    while i ≤ length(states)
        s = rand(b.states)
        s′ = rand(T(s,a))
        if rand(O(a,s′)) == o
            states[i] = s′
            i += 1
        end
    end
    return RejectionParticleFilter(states)
end
```

По мере увеличения количества частиц в фильтре распределение, представленное частицами, приближается к истинному апостериорному распределению. К сожалению, на практике парциальные фильтры не всегда работают. Низкое покрытие пространства состояний частицами и стохастический характер процедуры повторной выборки могут привести к тому, что частицы окажутся далеки от истинного состояния. Эта проблема *нехватки частиц* может быть смягчена несколькими стратегиями. Один из подходов продемонстрирован в примере 19.5.

Пример 19.5. Работа парциального фильтра в течение достаточно длительного времени может привести к нехватке частиц в соответствующих областях пространства состояний из-за стохастического характера передискретизации. Проблема более выражена, когда частиц меньше или когда частицы разбросаны по большому пространству состояний

Спелеолог Джо заблудился в лабиринте, построенном на основе квадратной сетки. Он потерял свой фонарь, поэтому может наблюдать окружающую среду только на ощупь. В любой момент Джо может сказать, есть ли стены в его местоположении в лабиринте со всех четырех сторон вокруг него. Джо достаточно уверен в своей способности чувствовать стены, поэтому он предполагает, что его наблюдения совершенны.

Джо использует парциальный фильтр, чтобы отслеживать свои представления о лабиринте с течением времени. В какой-то момент он останавливается, чтобы отдохнуть. Он продолжает использовать парциальный фильтр, чтобы обновить свои убеждения. На рисунках ниже показана эволюция его убеждений о своем положении в лабиринте, а точки указывают на частицы убеждений в его парциальном фильтре, соответствующие возможным местам в лабиринте.



Исходное убеждение имеет по одной частице в каждом месте сетки, которое соответствует его текущему наблюдению о наличии стенок на севере и юге. Спелеолог Джо не двигается и не получает новой информации, поэтому его убеждение не должно меняться со временем. Из-за стохастического характера повторной выборки последующие убеждения могут не содержать всех начальных состояний. Со временем его убеждение будет продолжать терять состояния, пока не останется только одно состояние. Может случиться так, что это состояние не будет соответствовать тому, где находится спелеолог Джо на самом деле.

19.7. Внесение частиц

Метод *внесения частиц* (particle injection) основан на добавлении случайных частиц для смягчения проблемы исключения. Алгоритм 19.8 вносит фиксированное количество частиц из более широкого распределения, такого как равномерное распределение по пространству состояний[12]. Хотя внесение частиц помогает компенсировать исключение, оно также снижает точность апостериорного представления, сформированного парциальным фильтром.

Алгоритм 19.8. Обновление убеждения методом парциального фильтра с внесением, при котором частицы `m_inject` отбираются из распределения внесения `D_inject` для снижения последствий исключения частиц

```
struct InjectionParticleFilter
    states   # вектор выборок состояния
```

[12] Для задач локализации роботов обычной практикой является введение частиц из равномерного распределения по всем возможным позам робота, взвешенных по текущему наблюдению.

```
    m_inject # количество выборок для внесения
    D_inject # распределение внесения
end

function update(b::InjectionParticleFilter, 𝒫, a, o)
    T, O, m_inject, D_inject = 𝒫.T, 𝒫.O, b.m_inject, b.D_inject
    states = [rand(T(s, a)) for s in b.states]
    weights = [O(a, s′, o) for s′ in states]
    D = SetCategorical(states, weights)
    m = length(states)
    states = vcat(rand(D, m - m_inject), rand(D_inject, m_inject))
    return InjectionParticleFilter(states, m_inject, D_inject)
end
```

Вместо использования фиксированного количества вносимых частиц при каждом обновлении мы можем применить более адаптивный подход. Когда всем частицам присваивается очень низкий вес, возникает естественное желание внести больше частиц. Может показаться заманчивым выбрать количество вносимых частиц исключительно на основе среднего веса текущего набора частиц. Однако это может сделать успех фильтра зависимым от естественных низких вероятностей наблюдения в ранние периоды, когда фильтр все еще сходится, или в моменты высокого шума сенсора[13].

Алгоритм 19.9 представляет собой реализацию метода *адаптивного внесения частиц*, который отслеживает два экспоненциальных скользящих средневзвешенных значения среднего веса частиц и основывает количество вносимых частиц на их соотношении[14]. Если w_{mean} – текущий средний вес частиц, два скользящих значения обновляются в соответствии с выражениями

$$w_{\text{fast}} \leftarrow w_{\text{fast}} + \alpha_{\text{fast}}(w_{\text{mean}} - w_{\text{fast}}); \tag{19.27}$$

$$w_{\text{slow}} \leftarrow w_{\text{slow}} + \alpha_{\text{slow}}(w_{\text{mean}} - w_{\text{slow}}), \tag{19.28}$$

где $0 \leqslant \alpha_{\text{slow}} < \alpha_{\text{fast}} \leqslant 1$.

Количество внесенных выборок в определенной итерации получается путем сравнения «быстрых» и «медленных» средних весов частиц[15]:

$$m_{\text{inject}} = \left\lfloor m \max\left(0, 1 - \nu \frac{w_{\text{fast}}}{w_{\text{slow}}}\right)\right\rceil. \tag{19.29}$$

Скаляр $\nu \geqslant 1$ позволяет нам управлять скоростью внесения. Использование метода адаптивного внесения частиц показано в примере 19.6.

---

[13] S. Thrun, W. Burgard, D. Fox, *Probabilistic Robotics*. MIT Press, 2006.

[14] D. E. Goldberg, J. Richardson, *An Experimental Comparison of Localization Methods*, in International Conference on Genetic Algorithms, 1987.

[15] Напомним, что $[x]$ обозначает ближайшее к x целое число.

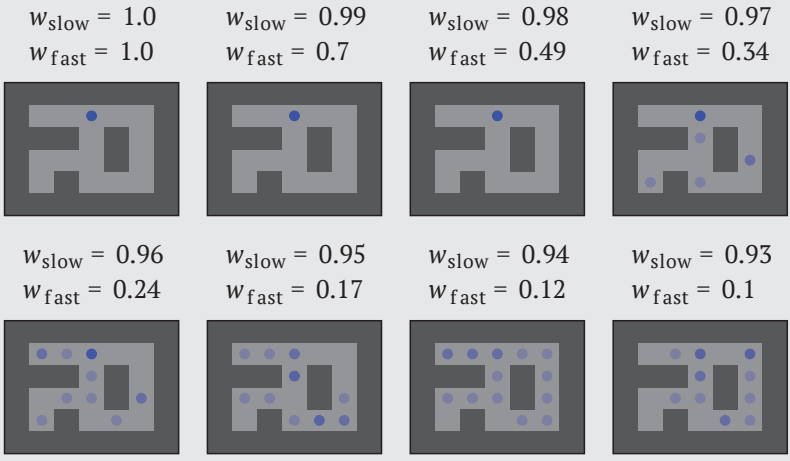
Алгоритм 19.9. Реализация парциального фильтра с адаптивным внесением, использующая быстрые и медленные экспоненциальные скользящие средневзвешенные значения `w_fast` и `w_slow` среднего веса частиц с коэффициентами гладкости `α_fast` и `α_slow` соответственно. Частицы вносят только в том случае, если быстрое скользящее среднее меньше $1/\nu$ медленного скользящего среднего. Рекомендуемые значения из оригинальной статьи: `α_fast = 0.1`, `α_slow = 0.001` и `ν = 2`

```
mutable struct AdaptiveInjectionParticleFilter
    states   # вектор выборок состояния
    w_slow   # медленное скользящее среднее
    w_fast   # быстрое скользящее среднее
    α_slow   # коэффициент медленного скользящего среднего
    α_fast   # коэффициент быстрого скользящего среднего
    ν        # коэффициент внесения
    D_inject # распределение внесения
end

function update(b::AdaptiveInjectionParticleFilter, 𝒫, a, o)
    T, O = 𝒫.T, 𝒫.O
    w_slow, w_fast, α_slow, α_fast, ν, D_inject =
    b.w_slow, b.w_fast, b.α_slow, b.α_fast, b.ν, b.D_inject
    states = [rand(T(s, a)) for s in b.states]
    weights = [O(a, s′, o) for s′ in states]
    w_mean = mean(weights)
    w_slow += α_slow*(w_mean - w_slow)
    w_fast += α_fast*(w_mean - w_fast)
    m = length(states)
    m_inject = round(Int, m * max(0, 1.0 - ν*w_fast / w_slow))
    D = SetCategorical(states, weights)
    states = vcat(rand(D, m - m_inject), rand(D_inject, m_inject))
    b.w_slow, b.w_fast = w_slow, w_fast
    return AdaptiveInjectionParticleFilter(states,
    w_slow, w_fast, α_slow, α_fast, ν, D_inject)
end
```

Пример 19.6. Парциальный фильтр с адаптивным внесением `α_slow = 0.01`, `α_fast = 0.3` и `ν = 2.0`, начиная с прореженного состояния с 16 одинаковыми частицами. Скользящие средние инициализируются значениями 1, чтобы отразить длительный период наблюдений, идеально совпадающих с каждой частицей в фильтре. На следующих итерациях эти скользящие средние изменяются с разной скоростью в зависимости от количества частиц, совпадающих с наблюдениями. Итерации изображены на рисунке слева направо и сверху вниз

Спелеолог Джо из примера 19.5 теперь перемещается на одну клетку на восток и также перемещает все частицы в своем парциальном фильтре на

одну клетку на восток. Теперь он чувствует стены только на севере и востоке, и, к сожалению, это наблюдение не согласуется ни с одной из обновленных частиц в его фильтре. Следовательно, эти частицы необходимо исключить. Джо решает использовать адаптивное внесение, чтобы решить проблему исключения частиц. На рисунках ниже мы видим, как его фильтр вносит частицы из равномерного случайного распределения, а также значения для быстрого и медленного фильтров:



Итерации выполняются слева направо и сверху вниз. Каждая синяя точка представляет собой частицу в парциальном фильтре, соответствующую частичному убеждению о том, что она находится в этом месте сетки.

19.8. Заключение

- Частично наблюдаемые марковские процессы принятия решений (POMDP) расширяют MDP за счет добавления неопределенности состояния.
- Неопределенность заставляет агентов в POMDP обладать убеждением о текущем состоянии.
- Убеждения агентов о POMDP с дискретными пространствами состояний могут быть выражены с помощью категориальных распределений и обновлены аналитическим путем.
- Убеждения агентов о линейных по Гауссу POMDP могут быть представлены с использованием гауссовых распределений и тоже могут быть обновлены аналитически.
- Убеждения о нелинейных непрерывных POMDP также могут быть представлены с помощью гауссовых распределений, но обычно они не могут быть

обновлены аналитически. В этом случае применяют расширенный фильтр Калмана и сигма-точечный фильтр Калмана.

- Непрерывные задачи иногда можно моделировать в предположении, что они линейны по Гауссу.
- Парциальные фильтры аппроксимируют убеждение большим набором частиц состояний.

19.9. Упражнения

Упражнение 19.1. Можно ли представить каждый MDP как POMDP?

Решение. Да. Формулировка POMDP расширяет формулировку MDP, вводя неопределенность состояния в виде распределения вероятности по наблюдениям. Любой MDP можно представить как POMDP с $\mathcal{O} = \mathcal{S}$ и $O(o \mid a, s') = (o = s')$.

Упражнение 19.2. Как выглядит обновление убеждений для дискретного POMDP без наблюдения? Как выглядит обновление убеждений для POMDP с линейной по Гауссу динамикой без наблюдения?

Решение. Если агент в POMDP без наблюдения с убеждением b выполняет действие a, новое убеждение b' можно рассчитать следующим образом:

$$b'(s') = P(s'|b,a) = \sum_s P(s'|a,b,s)P(s|b,a) = \sum_s T(s'|s,a)b(s).$$

Это обновление убеждения эквивалентно равномерному распределению по наблюдениям. В случае POMDP с линейной по Гауссу динамикой, которая не имеет наблюдения, агент обновит убеждение, используя только шаг предсказания фильтра Калмана в уравнении (19.12).

Упражнение 19.3. Автономный автомобиль формирует убеждение о своем местоположении с помощью многомерного нормального распределения. Автомобиль останавливается на светофоре, но модуль обновления убеждения продолжает работать, пока он стоит. Со временем убеждение крепнет и автомобиль приобретает чрезвычайно сильную уверенность в конкретном местоположении. Почему это может быть проблемой? Как можно избежать чрезмерной уверенности?

Решение. Чрезмерная уверенность в своем убеждении о среде может стать проблемой агента, когда модели или обновления убеждений не полностью отражают реальность. Слишком самонадеянное убеждение могло сойтись на состоянии, которое не соответствует истинному состоянию. Как только транспортное средство снова начнет движение, новые наблюдения могут резко разойтись с убеждением и привести к неверным оценкам. Чтобы решить эту проблему, можно потребовать, чтобы значения диагональных элементов матрицы ковариации были выше определенного порога.

Упражнение 19.4. Допустим, мы отслеживаем убеждение о частоте брака гаджетов, произведенных на фабрике. Мы используем распределение Пуассона, чтобы смоделировать вероятность того, что за один день работы фабрики будет произведено k бракованных изделий, при условии что процентная доля бракованных изделий на фабрике равна λ:

$$P(k|\lambda) = \frac{1}{k!}\lambda^k e^{-\lambda}.$$

Предположим, что наше первоначальное убеждение о частоте случаев брака следует гамма-распределению:

$$p(\lambda|\alpha,\beta) = \frac{\beta^\alpha}{\Gamma(\alpha)}\lambda^{\alpha-1}e^{-\beta\lambda},$$

где $\lambda \in (0, \infty)$, а убеждение параметризовано формой $\alpha > 0$ и коэффициентом $\beta > 0$. В конце рабочего дня завода мы наблюдаем, что было произведено $d \geqslant 0$ бракованных изделий. Покажите, что обновленное убеждение о частоте случаев брака также является гамма-распределением[16].

Решение. Мы ищем апостериорное распределение $p(\lambda|d, \alpha, \beta)$, которое можем получить с помощью правила Байеса:

$$\begin{aligned} p(\lambda|d,\alpha,\beta) &\propto p(d|\lambda)p(\lambda|\alpha,\beta) \\ &\propto \frac{1}{d!}\lambda^d e^{-\lambda}\frac{\beta^\alpha}{\Gamma(\alpha)}\lambda^{\alpha-1}e^{-\beta\lambda} \\ &\propto \lambda^{\alpha+d-1}e^{-(\beta+1)\lambda}. \end{aligned}$$

Это гамма-распределение:

$$\begin{aligned} p(\lambda|\alpha+d,\beta+1) &\propto \frac{(\beta+1)^{\alpha+d}}{\Gamma(\alpha+d)}\lambda^{\alpha+d-1}e^{-(\beta+1)\lambda} \\ &\propto \lambda^{\alpha+d-1}e^{-(\beta+1)\lambda}. \end{aligned}$$

Упражнение 19.5. Почему фильтры с исключением частиц не используют для обновления убеждений в POMDP с непрерывными наблюдениями?

Решение. Процедура исключения частиц требует многократной выборки функций перехода и наблюдения до тех пор, пока выборочное наблюдение не совпадет с истинным наблюдением. Вероятность выбора любого конкретного значения в непрерывном распределении вероятностей равна нулю, поэтому

[16] Гамма-распределение является априорно сопряженным по отношению к распределению Пуассона. *Сопряженное априорное распределение* (conjugate prior) – это семейство распределений вероятности, которые остаются в пределах своего семейства при обновлении с помощью наблюдения. Сопряженные априорные распределения полезны для моделирования убеждений, поскольку их форма остается постоянной.

выборки в процедуре исключения будут выполняться вечно. На практике мы бы использовали конечное представление для непрерывных значений, такое как 64-битные числа с плавающей запятой, но и в этом случае процесс выборки может выполняться очень долго для каждой частицы.

Упражнение 19.6. Объясните, почему спелеолог Джо ничего не выиграет от перехода на парциальный фильтр с адаптивным внесением с $\nu \geqslant 1$ в примере 19.5.

Решение. Согласно методу адаптивного внесения, мы вносим новые частицы, когда $\nu w_{\text{fast}}/w_{\text{slow}} < 1$. Спелеолог Джо предполагает идеальные наблюдения и абсолютно уверен в частицах, которые соответствуют его текущему наблюдению. Таким образом, вес каждой частицы равен 1, и w_{fast} и w_{slow} тоже равны 1. Отсюда следует, что отношение $w_{\text{fast}}/w_{\text{slow}}$ всегда равно 1, что не приводит к появлению новых частиц.

Упражнение 19.7. Почему скаляру скорости внесения ν в фильтре с адаптивным внесением обычно не присваивают значение меньше 1?

Решение. Метод внесения частиц был разработан для добавления частиц, когда правдоподобие текущих наблюдений ниже, чем историческая тенденция, следующая из правдоподобия всех наблюдений. Таким образом, внесение обычно происходит только тогда, когда краткосрочная оценка среднего веса частиц w_{fast} меньше долгосрочной оценки среднего веса частиц w_{slow}. Если $\nu < 1$, то частицы продолжают генерироваться, даже если $w_{\text{fast}} > w_{\text{slow}}$, хотя из этого неравенства следует, что текущие наблюдения имеют более высокое правдоподобие, чем в среднем в прошлом.

Упражнение 19.8. Предположим, что мы попали в прямоугольный лес в начальном месте, выбранном случайным образом. Мы не знаем, в каком направлении смотрим. К счастью, мы знаем размеры леса (у него ширина w и длина $\ell \gg w$)[17]. Мы можем двигаться по непрерывному пути, постоянно наблюдая, находимся ли мы все еще в лесу. Как мы можем применить обновление представлений к этой задаче? На рисунке ниже показаны три возможные стратегии, каждая из которых определяет свой путь. Какая из этих стратегий гарантированно выведет из леса? Какая стратегия лучше?

Прямой путь длиной $2w$

Два перпендикулярных сегмента, каждый длиной $\sqrt{2w}$

Две стороны равностороннего треугольника, каждая длиной $\frac{2\sqrt{3}}{3}w$

[17] В основе этого задания лежит «задача о заблудившихся в лесу» Ричарда Беллмана, в которой мы начинаем со случайного места и направления в лесу с известной геометрией и должны найти стратегию, которая минимизирует среднее (или максимальное) время выхода из леса. R. Bellman, *Minimization Problem*, Bulletin of the American Mathematical Society, vol. 62, no. 3, p. 270, 1956.

Решение. Наше первоначальное убеждение состоит в равномерном распределении вероятности по всем двумерным местоположениям и ориентациям (состояниям) в лесу. Мы можем составить обновленное убеждение, используя путь, который прошли до сих пор. Если мы все еще находимся в лесу, наше убеждение состоит из всех состояний, которые можно достичь из состояния внутри леса, следуя по нашему пути и оставаясь полностью в лесу. Как только мы выходим из леса, наше убеждение состоит из всех состояний, которые позволяют достичь края леса, следуя нашему пути и оставаясь полностью в лесу.

Из заданных стратегий только две последние гарантированно выведут из леса. Путь, образованный двумя перпендикулярными сегментами и двумя сторонами равностороннего треугольника, всегда будет пересекаться с границей леса. Прямой отрезок, однако, может не выйти за пределы леса. Мы предпочитаем более короткую из двух стратегий спасения, а именно равносторонний треугольник.

Упражнение 19.9. Алгоритм 19.2 проверяет, является ли обновленное убеждение нулевым вектором. Когда обновление убеждения может привести к нулевому вектору? Почему это может произойти в реальных ситуациях?

Решение. Нулевой вектор убеждения может быть результатом наблюдения o, которое считается невозможным. Эта ситуация может возникнуть после выполнения действия a из представления b, когда $O(o|a, s') = 0$ для всех возможных следующих состояний s' в соответствии с убеждением b и нашей моделью перехода. Алгоритм 19.2 обрабатывает этот случай, возвращая равномерное распределение по убеждениям. В практических сценариях модель может не соответствовать реальному миру. Как правило, при построении модели стараются не присваивать наблюдениям нулевую вероятность на тот случай, если наши убеждения, модели перехода или наблюдения неверны.

Упражнение 19.10. Предположим, мы осуществляем мониторинг состояния самолета в полете. Самолет находится либо в исправном состоянии s^0, либо в состоянии неисправности s^1. Мы получаем наблюдения через отсутствие аварийного предупреждения w^0 или наличие такого предупреждения w^1. Мы можем разрешить самолету продолжать полет m^0 или отправить самолет на техническое обслуживание m^1. У нас есть следующие динамики переходов и наблюдений, где мы предполагаем, что при заданном состоянии самолета предупреждения не зависят от действий:

$T(s^0|s^0, m^0) = 0.95$; $O(w^0|s^0) = 0.99$;

$T(s^0|s^0, m^1) = 1$; $O(w^1|s^1) = 0.7$;

$T(s^1|s^1, m^0) = 1$;

$T(s^0|s^1, m^1) = 0.98$.

Исходя из начального убеждения **b** = [0.95, 0.05], вычислите обновленное убеждение **b**′, при условии что мы позволяем самолету продолжать полет и наблюдаем аварийное предупреждение.

Решение. Используя уравнение (19.7), мы обновляем убеждение для s^0:

$$b'(s^0) \propto O(w^1|s^0)\sum_s T(s^0|s,m^0)b(s);$$
$$b'(s^0) \propto O(w^1|s^0)\left(T(s^0|s^0,m^0)b(s^0)+T(s^0|s^1,m^0)b(s^1)\right);$$
$$b'(s^0) \propto (1-0.99)(0.95\times 0.95+(1-1)\times 0.05)=0.009025.$$

Затем повторяем обновление для s^1:

$$b'(s^1) \propto O(w^1|s^1)\sum_s T(s^1|s,m^0)b(s);$$
$$b'(s^1) \propto O(w^1|s^1)\left(T(s^1|s^0,m^0)b(s^0)+T(s^1|s^1,m^0)b(s^1)\right);$$
$$b'(s^1) \propto 0.7((1-0.95)\times 0.95+1\times 0.05)=0.06825.$$

После нормализации получаем следующее обновленное убеждение:

$$b'(s^0)=\frac{b'(s^0)}{b'(s^0)+b'(s^1)}\approx 0.117;$$
$$b'(s^1)=\frac{b'(s^1)}{b'(s^0)+b'(s^1)}\approx 0.883;$$
$$b'\approx[0.117, 0.883].$$

Упражнение 19.11. Допустим, у нас есть робот, движущийся по прямой линии с координатой x, скоростью v и ускорением a. На каждом временном шаге мы напрямую контролируем ускорение и наблюдаем за скоростью. Уравнения движения робота имеют вид:

$$x'=x+v\Delta t+\frac{1}{2}a\Delta t^2;$$
$$v'=v+a\Delta t,$$

где Δt – продолжительность каждого шага. Предположим, мы хотим использовать фильтр Калмана, чтобы обновить наше убеждение. Вектор состояния **s** = [x, v]. Определите $\mathbf{T}_s$, $\mathbf{T}_a$ и $\mathbf{O}_s$.

Решение. Динамику перехода и наблюдения можно записать в линейной форме следующим образом:

$$\begin{bmatrix}x'\\v'\end{bmatrix}=\begin{bmatrix}1 & \Delta t\\0 & 1\end{bmatrix}\begin{bmatrix}x\\v\end{bmatrix}+\begin{bmatrix}\frac{1}{2}\Delta t^2\\\Delta t\end{bmatrix}a;$$

$$o = \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} x' \\ v' \end{bmatrix}.$$

С помощью этих уравнений мы можем определить $\mathbf{T}_s$, $\mathbf{T}_a$ и $\mathbf{O}_s$:

$$\mathbf{T}_s = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}; \quad \mathbf{T}_a = \begin{bmatrix} \frac{1}{2}\Delta t^2 \\ \Delta t \end{bmatrix}; \quad \mathbf{O}_s = \begin{bmatrix} 0 & 1 \end{bmatrix}.$$

Упражнение 19.12. Допустим, у нас есть робот с дифференциальным приводом, движущийся в двух измерениях с постоянной скоростью v. Состоянием робота является его положение (x, y) и его курс θ. На каждом временном шаге мы контролируем угловую скорость поворота робота w. Уравнения движения робота следующие:

$x' = x + v\cos(\theta)\Delta t$;

$y' = y + v\sin(\theta)\Delta t$;

$\theta' = \theta + w\Delta t$.

Эта функция перехода нелинейна. Какова ее линеаризация $\mathbf{T}_s$ для состояния $\mathbf{s} = [x, y, \theta]$?

Решение. Линеаризация может быть записана в виде якобиана следующим образом:

$$\mathbf{T_s} = \begin{bmatrix} \frac{\partial x'}{\partial x} & \frac{\partial x'}{\partial y} & \frac{\partial x'}{\partial \theta} \\ \frac{\partial y'}{\partial x} & \frac{\partial y'}{\partial y} & \frac{\partial y'}{\partial \theta} \\ \frac{\partial \theta'}{\partial x} & \frac{\partial \theta'}{\partial y} & \frac{\partial \theta'}{\partial \theta} \end{bmatrix} = \begin{bmatrix} 1 & 0 & -v\sin(\theta)\Delta t \\ 0 & 1 & v\cos(\theta)\Delta t \\ 0 & 0 & 1 \end{bmatrix}.$$

Эту линеаризацию можно использовать в расширенном фильтре Калмана для работы с представлениями.

Упражнение 19.13. Предположим, мы выбрали следующие $2n$ сигма-точек для n-мерного распределения:

$\mathbf{s}_{2i} = \boldsymbol{\mu} + \sqrt{n\boldsymbol{\Sigma}_i}$ для i в интервале $1{:}n$;

$\mathbf{s}_{2i-1} = \boldsymbol{\mu} - \sqrt{n\boldsymbol{\Sigma}_i}$ для i в интервале $1{:}n$.

Покажите, что мы можем восстановить среднее значение и ковариацию по этим сигма-точкам, используя веса $w_i = 1/(2n)$.

Решение. Если мы воспользуемся весами $w_i = 1/(2n)$, восстановленное среднее будет равно

$$\sum_i w_i \mathbf{s}_i = \sum_{i=1}^{n} \frac{1}{2n}\left(\mu + \sqrt{n\Sigma_i}\right) + \frac{1}{2n}\left(\mu - \sqrt{n\Sigma_i}\right) = \sum_{i=1}^{n} \frac{1}{n}\mu = \mu,$$

а восстановленная ковариация:

$$\begin{aligned}\sum_i w_i(\mathbf{s}_i - \boldsymbol{\mu}')(\mathbf{s}_i - \boldsymbol{\mu}')^\top &= 2\sum_{i=1}^{n} \frac{1}{2n}\left(\sqrt{n\boldsymbol{\Sigma}_i}\right)\left(\sqrt{n\boldsymbol{\Sigma}_i}\right)^\top \\ &= \frac{1}{n}\sum_{i=1}^{n}\left(\sqrt{n\boldsymbol{\Sigma}_i}\right)\left(\sqrt{n\boldsymbol{\Sigma}_i}\right)^\top \\ &= \sqrt{\boldsymbol{\Sigma}}\sqrt{\boldsymbol{\Sigma}}^\top \\ &= \boldsymbol{\Sigma}.\end{aligned}$$

Упражнение 19.14. Возьмем $2n$ сигма-точек и весов из предыдущей задачи, обладающих средним $\boldsymbol{\mu}$ и ковариацией $\boldsymbol{\Sigma}$. Мы должны параметризовать сигма-точки и веса, чтобы контролировать концентрацию точек вокруг среднего значения. Покажите, что мы можем построить новый набор сигма-точек, равномерно понизив веса исходных сигма-точек, а затем включив среднее $\boldsymbol{\mu}$ в качестве дополнительной сигма-точки. Покажите, что этот новый набор из $2n + 1$ сигма-точек соответствует форме в уравнении (19.23).

Решение. Мы можем включить среднее $\boldsymbol{\mu}$ в набор сигма-точек из упражнения 19.13, чтобы получить новый набор из $2n + 1$ сигма-точек:

$\mathbf{s}_1 = \boldsymbol{\mu}$;

$\mathbf{s}_{2i} = \boldsymbol{\mu} + \left(\sqrt{\frac{n}{1-w_1}\boldsymbol{\Sigma}}\right)_i$ для i в интервале $1{:}n$;

$\mathbf{s}_{2i+1} = \boldsymbol{\mu} - \left(\sqrt{\frac{n}{1-w_1}\boldsymbol{\Sigma}}\right)_i$ для i в интервале $1{:}n$,

где w_1 – вес первой сигма-точки. Веса оставшихся сигма-точек равномерно уменьшаются от $1/(2n)$ до $(1 - w_1)/(2n)$. Восстановленное среднее по-прежнему равно $\boldsymbol{\mu}$, а восстановленная ковариация по-прежнему равна $\boldsymbol{\Sigma}$.

Мы можем менять значение w_1 для получения различных наборов сигма-точек. Использование $w_1 > 0$ приводит к тому, что сигма-точки расходятся от среднего значения; в случае $w_1 < 0$ сигма-точки перемещаются ближе к среднему значению. Это дает нам масштабированное множество сигма-точек с разными моментами более высокого порядка, но сохраняет неизменными среднее значение и ковариацию.

Мы можем получить уравнение (19.23), подставив $w_1 = \lambda/(n + \lambda)$. Отсюда следует, что $(1 - w_1)/ 2n = 1/(2(n + \lambda))$ и $n/(1 - w_1) = n + \lambda$.

Упражнение 19.15. Найдите набор сигма-точек и весов с $\lambda = 2$ для многомерного распределения Гаусса с

$$\boldsymbol{\mu} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}; \quad \boldsymbol{\Sigma} = \begin{bmatrix} 4 & 0 \\ 0 & 2.25 \end{bmatrix}.$$

Решение. Поскольку нам дано двумерное распределение Гаусса и $\lambda = 2$, мы должны найти $2n + 1 = 5$ сигма-точек. Нам нужно вычислить матрицу квадратного корня $\mathbf{B} = \sqrt{(n+\lambda)\boldsymbol{\Sigma}}$ такую, что $\mathbf{B}\mathbf{B}^\mathsf{T} = (n+\lambda)\boldsymbol{\Sigma}$. Поскольку масштабированная ковариационная матрица является диагональной, матрица квадратного корня представляет собой просто поэлементный квадратный корень из $(n+\lambda)\boldsymbol{\Sigma}$:

$$\sqrt{(n+\lambda)\boldsymbol{\Sigma}} = \sqrt{(2+2)\begin{bmatrix} 4 & 0 \\ 0 & 2.25 \end{bmatrix}} = \begin{bmatrix} 4 & 0 \\ 0 & 3 \end{bmatrix}.$$

Теперь мы можем вычислить сигма-точки и веса:

$$\mathbf{s}_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}; \qquad w_1 = \frac{2}{2+2} = \frac{1}{2};$$

$$\mathbf{s}_2 = \begin{bmatrix} 1 \\ 2 \end{bmatrix} + \begin{bmatrix} 4 \\ 0 \end{bmatrix} = \begin{bmatrix} 5 \\ 2 \end{bmatrix}; \qquad w_2 = \frac{1}{2(2+2)} = \frac{1}{8};$$

$$\mathbf{s}_3 = \begin{bmatrix} 1 \\ 2 \end{bmatrix} - \begin{bmatrix} 4 \\ 0 \end{bmatrix} = \begin{bmatrix} -3 \\ 2 \end{bmatrix}; \qquad w_3 = \frac{1}{2(2+2)} = \frac{1}{8};$$

$$\mathbf{s}_4 = \begin{bmatrix} 1 \\ 2 \end{bmatrix} + \begin{bmatrix} 0 \\ 3 \end{bmatrix} = \begin{bmatrix} 1 \\ 5 \end{bmatrix}; \qquad w_4 = \frac{1}{2(2+2)} = \frac{1}{8};$$

$$\mathbf{s}_5 = \begin{bmatrix} 1 \\ 2 \end{bmatrix} - \begin{bmatrix} 0 \\ 3 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}; \qquad w_5 = \frac{1}{2(2+2)} = \frac{1}{8}.$$

Упражнение 19.16. Используя сигма-точки и веса из предыдущего упражнения, вычислите обновленное среднее значение и ковариацию, полученные сигма-точечным преобразованием как $\mathbf{f}(\mathbf{x}) = [2x_1, x_1x_2]$.

Решение. Преобразованные сигма-точки равны

$$\mathbf{f}(\mathbf{s}_1) = \begin{bmatrix} 2 \\ 2 \end{bmatrix}; \quad \mathbf{f}(\mathbf{s}_2) = \begin{bmatrix} 10 \\ 10 \end{bmatrix}; \quad \mathbf{f}(\mathbf{s}_3) = \begin{bmatrix} -6 \\ -6 \end{bmatrix}; \quad \mathbf{f}(\mathbf{s}_4) = \begin{bmatrix} 2 \\ 5 \end{bmatrix}; \quad \mathbf{f}(\mathbf{s}_5) = \begin{bmatrix} 2 \\ -1 \end{bmatrix}.$$

Мы можем восстановить среднее значение как взвешенную сумму преобразованных сигма-точек:

$$\boldsymbol{\mu}' = \sum_i w_i \mathbf{f}(\mathbf{s}_i);$$

$$\boldsymbol{\mu}' = \frac{1}{2}\begin{bmatrix}2\\2\end{bmatrix} + \frac{1}{8}\begin{bmatrix}10\\10\end{bmatrix} + \frac{1}{8}\begin{bmatrix}-6\\-6\end{bmatrix} + \frac{1}{8}\begin{bmatrix}2\\5\end{bmatrix} + \frac{1}{8}\begin{bmatrix}2\\-1\end{bmatrix} = \begin{bmatrix}2\\2\end{bmatrix}.$$

Ковариационная матрица может быть восстановлена из взвешенной суммы поточечных ковариационных матриц:

$$\boldsymbol{\Sigma}' = \sum_i w_i \big(\mathbf{f}(\mathbf{s}_i) - \boldsymbol{\mu}'\big)\big(\mathbf{f}(\mathbf{s}_i) - \boldsymbol{\mu}'\big)^\top;$$

$$\boldsymbol{\Sigma}' = \frac{1}{2}\begin{bmatrix}0 & 0\\0 & 0\end{bmatrix} + \frac{1}{8}\begin{bmatrix}64 & 64\\64 & 64\end{bmatrix} + \frac{1}{8}\begin{bmatrix}64 & 64\\64 & 64\end{bmatrix} + \frac{1}{8}\begin{bmatrix}0 & 0\\0 & 9\end{bmatrix} + \frac{1}{8}\begin{bmatrix}0 & 0\\0 & 9\end{bmatrix} = \begin{bmatrix}16 & 16\\16 & 18.25\end{bmatrix}.$$

Упражнение 19.17. И фильтр Калмана, и расширенный фильтр Калмана вычисляют матрицу взаимной ковариации $\boldsymbol{\Sigma}_{po}$, используя ковариацию наблюдения $\mathbf{O}_s$. Сигма-точечный фильтр Калмана вычисляет напрямую не эту матрицу наблюдений, а матрицу $\boldsymbol{\Sigma}_{po}$. Покажите, что обновление ковариации для сигма-точечного фильтра Калмана $\boldsymbol{\Sigma}_{b'} \leftarrow \boldsymbol{\Sigma}_p - \mathbf{K}\boldsymbol{\Sigma}_o\mathbf{K}^\top$ совпадает с обновлением ковариации для фильтра Калмана и расширенного фильтра Калмана $\boldsymbol{\Sigma}_{b'} \leftarrow (\mathbf{I} - \mathbf{K}\mathbf{O}_s)\boldsymbol{\Sigma}_p$.

Решение. Мы можем использовать отношения $\mathbf{K} = \boldsymbol{\Sigma}_{po}\boldsymbol{\Sigma}_o^{-1}$ и $\boldsymbol{\Sigma}_{po} = \boldsymbol{\Sigma}_p\mathbf{O}_s^\top$, чтобы показать, что эти два обновления эквивалентны. Обратите также внимание, что симметричная матрица является транспонированной по отношению к себе, а ковариационные матрицы симметричны.

$$\begin{aligned}
\boldsymbol{\Sigma}_{b'} &= \boldsymbol{\Sigma}_p - \mathbf{K}\boldsymbol{\Sigma}_o\mathbf{K}^\top;\\
&= \boldsymbol{\Sigma}_p - \mathbf{K}\boldsymbol{\Sigma}_o(\boldsymbol{\Sigma}_{po}\boldsymbol{\Sigma}_o^{-1})^\top;\\
&= \boldsymbol{\Sigma}_p - \mathbf{K}\boldsymbol{\Sigma}_o(\boldsymbol{\Sigma}_o^{-1})^\top\boldsymbol{\Sigma}_{po}^\top;\\
&= \boldsymbol{\Sigma}_p - \mathbf{K}\boldsymbol{\Sigma}_{po}^\top;\\
&= \boldsymbol{\Sigma}_p - \mathbf{K}(\boldsymbol{\Sigma}_p\mathbf{O}_s^\top)^\top;\\
&= \boldsymbol{\Sigma}_p - \mathbf{K}\mathbf{O}_s\boldsymbol{\Sigma}_p^\top;\\
&= \boldsymbol{\Sigma}_p - \mathbf{K}\mathbf{O}_s\boldsymbol{\Sigma}_p;\\
&= (\mathbf{I} - \mathbf{K}\mathbf{O}_s)\boldsymbol{\Sigma}_p.
\end{aligned}$$

Упражнение 19.18. Каковы преимущества и недостатки использования парциального фильтра вместо фильтра Калмана?

Решение. Фильтр Калмана может обеспечить точное обновление убеждений, когда система линейна по Гауссу. Парциальные фильтры могут работать лучше, когда система является нелинейной, а неопределенность – мультимодальной. Но при этом парциальные фильтры, как правило, более затратны в вычислительном отношении и могут страдать от нехватки частиц.

Упражнение 19.19. Допустим, мы используем парциальный фильтр, чтобы обновлять убеждение в задаче с очень надежными наблюдениями, причем наблюдения имеют либо высокую, либо низкую вероятность. Например, в задаче о спелеологе Джо мы можем надежно определить, какие из четырех стен присутствуют рядом, что позволяет нам немедленно игнорировать любые состояния, которые не соответствуют наблюдению. Почему парциальный фильтр с исключением лучше подходит для таких проблем, чем обычный?

Решение. Обычный парциальный фильтр создает набор частиц и присваивает им веса в соответствии с вероятностью их наблюдения. В задачах, подобных задаче о спелеологе Джо, многие частицы могут получить практически нулевой вес. Наличие большого количества частиц с малым весом делает убеждение уязвимым к утрате частиц. Фильтр частиц с исключением гарантирует, что состояние преемника каждой частицы совместимо с наблюдением, тем самым смягчая проблему утраты частиц.

20 Точное планирование с использованием убеждений-состояний

Решение задачи POMDP состоит в том, чтобы выбрать действия, которые максимизируют совокупное вознаграждение при взаимодействии агента со средой. В отличие от MDP, здесь состояния нельзя наблюдать напрямую, и агенту необходимо использовать свою прошлую историю действий и наблюдений, чтобы сформировать убеждение. В предыдущей главе говорилось о том, что убеждения можно представить как распределения вероятностей по состояниям. Существуют разные подходы к вычислению оптимальной стратегии, которая отображает убеждения в действия в соответствии с имеющимися моделями переходов, наблюдений и вознаграждений[1]. Один из подходов заключается в преобразовании POMDP в MDP и применении динамического программирования. К другим подходам относятся представления стратегий в виде условных планов или кусочно-линейных функций значений в пространстве убеждений. Глава завершается алгоритмом вычисления оптимальной стратегии, который аналогичен алгоритму итерации по полезности для MDP.

20.1. MDP убеждений-состояний

Любой POMDP можно рассматривать как MDP, использующий убеждения в качестве состояний. Такие алгоритмы часто встречаются в литературе под названием *MDP убеждений-состояний* (belief-state MDP, BS-MDP)[2]. Пространство состояний такого MDP – это набор всех убеждений B. Пространство действий идентично пространству POMDP.

Функция вознаграждения BS-MDP зависит от убеждения и выполненного действия. Это просто ожидаемое значение вознаграждения. Для дискретного

---

[1] Обсуждение методов точного решения представлено в работе L. P. Kaelbling, M. L. Littman, A. R. Cassandra, *Planning and Acting in Partially Observable Stochastic Domains*, Artificial Intelligence, vol. 101, no. 1–2, pp. 99–134, 1998.

[2] K. J. Åström, *Optimal Control of Markov Processes with Incomplete State Information*, Journal of Mathematical Analysis and Applications, vol. 10, no. 1, pp. 174–205, 1965.

пространства состояний функция вознаграждения определяется следующим образом:

$$R(b,a) = \sum_s R(s,a)b(s). \tag{20.1}$$

Если пространства состояний и наблюдений дискретны, функция перехода состояния-убеждения для BS-MDP задается уравнением

$$\begin{aligned} T(b'|b,a) &= P(b'|b,a) && (20.2)\\ &= \sum_o P(b'|b,a,o)P(o|b,a) && (20.3)\\ &= \sum_o P(b'|b,a,o)\sum_s P(o|b,a,s)P(s|b,a) && (20.4)\\ &= \sum_o P(b'|b,a,o)\sum_s P(o|b,a,s)b(s) && (20.5)\\ &= \sum_o P(b'|b,a,o)\sum_{s'}\sum_s P(o|b,a,s,s')P(s'|b,s,a)b(s) && (20.6)\\ &= \sum_o \left(b' = \text{Update}(b,a,o)\right)\sum_{s'} O(o|a,s')\sum_s T(s'|s,a)b(s). && (20.7) \end{aligned}$$

В уравнении (20.7) оператор Update(b, a, o) возвращает обновленное убеждение, используя детерминированный процесс, обсуждавшийся в предыдущей главе[3]. Для непрерывных задач суммирование заменяется интегрированием.

Решение задач BS-MDP затруднительно, если пространство состояний является непрерывным. Мы можем использовать приемы приближенного динамического программирования, представленные в предыдущих главах, но часто удается добиться большего успеха, используя структуру BS-MDP, о чем пойдет речь в оставшейся части этой главы.

20.2. Условные планы

Существует несколько способов представления стратегий для POMDP. Один из подходов заключается в использовании *условного плана* (conditional plan), представленного в виде дерева. На рис. 20.1 показан пример трехшагового условного плана с бинарными пространствами действия и наблюдения. Узлы соответствуют состояниям-убеждениям. Ребра обозначают наблюдения, а узлы – действия. Если у нас есть план π, действие, ассоциированное с корнем, обозначается как $\pi()$, а подплан, ассоциированный с наблюдением o, обозначается как $\pi(o)$. Реализация этого метода приведена в алгоритме 20.1.

[3] Напоминаем, что мы используем соглашение, согласно которому логическое утверждение в круглых скобках обрабатывается численно как 1, если оно истинно, и 0, если ложно.

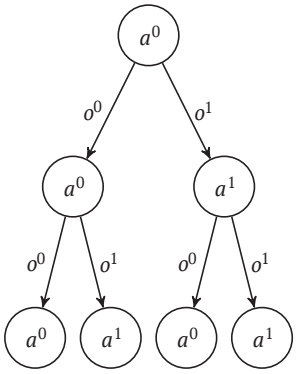


Рис. 20.1. Условный план из трех шагов

Алгоритм 20.1. Структура данных условного плана, состоящая из действия и отображения из наблюдений в подпланы. Поле `subplans` – это `Dict`, словарь отображения из наблюдений в условные планы. Для удобства мы создали специальный конструктор планов, состоящих из одного узла

```
struct ConditionalPlan
    a        # действие, совершаемое из корня
    subplans # словарь отображения действий в подпланы
end

ConditionalPlan(a) = ConditionalPlan(a, Dict())

(π::ConditionalPlan)() = π.a
(π::ConditionalPlan)(o) = π.subplans[o]
```

Условный план говорит нам, что делать в ответ на наблюдения вплоть до горизонта, представленного деревом. Чтобы выполнить условный план, мы начинаем с корневого узла и выполняем связанное с ним действие. Мы идем вниз по дереву в соответствии с нашими наблюдениями, выполняя действия, связанные с узлами, через которые мы проходим.

Предположим, у нас есть условный план π и мы хотим вычислить его ожидаемую полезность при запуске из состояния s. Это вычисление может быть выполнено рекурсивно:

$$U^{\pi}(s) = R(s,\pi()) + \gamma\left[\sum_{s'} T(s'|s,\pi()) \sum_{o} O(o|\pi(),s') U^{\pi(o)}(s')\right]. \tag{20.8}$$

Реализация этой процедуры дана в алгоритме 20.2.

Алгоритм 20.2. Метод оценки условного плана π для MDP $\mathcal{P}$ начиная с состояния s. Планы представлены в виде кортежей, состоящих из действия и словаря, отображающего наблюдения в подпланы

```
function lookahead(𝒫::POMDP, U, s, a)
    𝒮, 𝒪, T, O, R, γ = 𝒫.𝒮, 𝒫.𝒪, 𝒫.T, 𝒫.O, 𝒫.R, 𝒫.γ
    u′ = sum(T(s,a,s′)*sum(O(a,s′,o)*U(o,s′) for o in 𝒪) for s′ in 𝒮)
    return R(s,a) + γ*u′
end

function evaluate_plan(𝒫::POMDP, π::ConditionalPlan, s)
    U(o,s′) = evaluate_plan(𝒫, π(o), s′)
    return isempty(π.subplans) ? 𝒫.R(s,π()) : lookahead(𝒫, U, s, π())
end
```

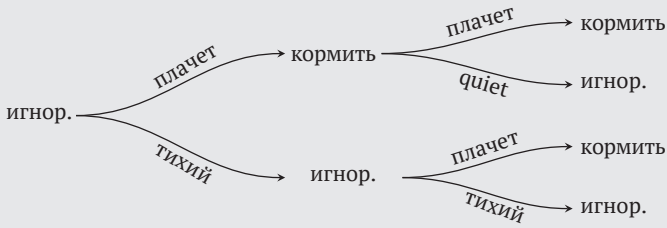
Мы можем вычислить полезность нашего убеждения b следующим образом:

$$\mathcal{U}^{\pi}(b) = \sum_{s} b(s)\mathcal{U}^{\pi}(s). \tag{20.9}$$

В примере 20.1 показано вычисление полезности, связанной с трехшаговым условным планом.

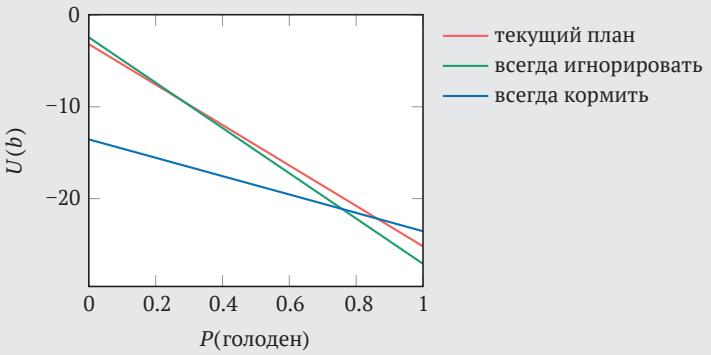
Пример 20.1. Условный план для трехшаговой задачи о плачущем ребенке (приложение F.7), оцениваемый в сравнении с двумя более простыми условными планами

Рассмотрим следующий трехшаговый условный план для задачи о плачущем ребенке:



В этом плане мы начинаем с игнорирования ребенка. Если мы наблюдаем плач, мы кормим ребенка. Если мы не наблюдаем плач, мы игнорируем ребенка. Нашим третьим условным действием снова будет кормление, если ребенок плачет.

На рисунке ниже ожидаемая полезность этого плана в пространстве убеждений изображена рядом с трехшаговым планом, согласно которому мы всегда кормим ребенка, и планом, согласно которому всегда игнорируем его.



Из графика следует, что наш текущий план не всегда лучше, чем постоянное игнорирование или постоянное кормление ребенка.

Теперь, когда у нас есть способ оценить условные планы до горизонта h, мы можем вычислить оптимальную h-шаговую функцию полезности:

$$\mathcal{U}^{*}(b) = \max_{\pi} \mathcal{U}^{\pi}(b). \tag{20.10}$$

Оптимальное действие может быть сгенерировано из действия, связанного с корнем π, дающего максимальную полезность.

Как показано на рис. 20.2, решение h-шаговой задачи POMDP путем прямого перебора всех условных h-шаговых планов обычно трудновыполнимо с точки зрения вычислений. В h-шаговом плане имеется $(|\mathcal{O}|^h - 1)/(|\mathcal{O}| - 1)$ узлов. В общем случае любое действие может быть вставлено в любой узел, что дает $|\mathcal{A}|^{(|\mathcal{O}|^h-1)/(|\mathcal{O}|-1)}$ возможных h-шаговых планов. Этот экспоненциальный взрыв означает, что перебор всех планов невозможен даже при скромных значениях h. Как будет показано далее в этой главе, существуют альтернативы прямому перебору всех возможных планов.

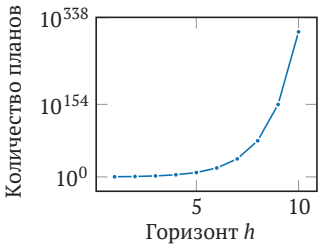


Рис. 20.2. Даже для небольших POMDP с двумя действиями и двумя наблюдениями количество возможных планов чрезвычайно быстро растет с увеличением горизонта планирования. Часто мы можем ощутимо сократить набор альфа-векторов на каждой итерации и рассматривать гораздо меньшее количество планов

20.3. Альфа-векторы

Перепишем уравнение (20.9) в векторной форме:

$$\mathcal{U}^{\pi}(b) = \sum_{s} b(s)\mathcal{U}^{\pi}(s) = \boldsymbol{\alpha}_{\pi}^{\top}\mathbf{b}. \tag{20.11}$$

Вектор $\boldsymbol{\alpha}_{\pi}$, называемый *альфа-вектором*, содержит ожидаемую полезность при следовании плану π для каждого состояния. Как и в случае векторов убеждений, альфа-векторы имеют размерность $|\mathcal{S}|$. Но, в отличие от убеждений, компоненты альфа-векторов представляют полезности, а не вероятностные меры. В алгоритме 20.3 реализовано вычисление альфа-вектора.

Алгоритм 20.3. Мы можем сгенерировать альфа-вектор из условного плана, вызвав `evaluate_plan` из всех возможных начальных состояний

```
function alphavector(𝒫::POMDP, π::ConditionalPlan)
    return [evaluate_plan(𝒫, π, s) for s in 𝒫.𝒮]
end
```

Каждый альфа-вектор определяет гиперплоскость в пространстве убеждений. Оптимальная функция полезности, заданная в уравнении (20.11), является максимальной на этих гиперплоскостях:

$$\mathcal{U}^{*}(\mathbf{b}) = \max_{\pi} \boldsymbol{\alpha}_{\pi}^{\top}\mathbf{b}, \tag{20.12}$$

что делает функцию полезности кусочно-линейной и выпуклой[4].

Альтернативой использованию условного плана для представления стратегии является использование набора альфа-векторов Γ, каждый из которых связан с действием. Хотя это и нецелесообразно, один из способов сгенерировать набор Γ заключается в том, чтобы составить набор h-шаговых условных планов и затем вычислить их альфа-векторы. Действие, связанное с альфа-вектором, является действием в корне ассоциированного условного плана. Мы следуем стратегии, представленной Γ, обновляя состояние-убеждение и выполняя действие, связанное с доминирующим альфа-вектором в новом убеждении $\mathbf{b}$. *Доминирующий альфа-вектор* $\boldsymbol{\alpha}$ в $\mathbf{b}$ – это вектор, который обеспечивает наибольшее значение $\boldsymbol{\alpha}^{\top}\mathbf{b}$. Эту стратегию можно использовать для выбора действий за горизонтом исходных условных планов. Реализация представлена в алгоритме 20.4.

[4] Оптимальная функция полезности для POMDP с непрерывным состоянием также является выпуклой, как можно увидеть, аппроксимируя POMDP посредством дискретизации пространства состояний и рассматривая предел, когда количество дискретных состояний приближается к бесконечности.

Алгоритм 20.4. Стратегия по методу альфа-вектора определяется набором альфа-векторов Γ и массивом связанных действий a. При заданном убеждении b алгоритм найдет альфа-вектор, который дает наибольшее значение в этой точке убеждения, и вернет связанное действие

```
struct AlphaVectorPolicy
    𝒫 # задача POMDP
    Γ # альфа-векторы
    a # действия, связанные с альфа-векторами
end

function utility(π::AlphaVectorPolicy, b)
    return maximum(α⋅b for α in π.Γ)
end

function (π::AlphaVectorPolicy)(b)
    i = argmax([α⋅b for α in π.Γ])
    return π.a[i]
end
```

Если мы используем *пошаговый предпросмотр* (one-step lookahead), нам не нужно отслеживать действия, связанные с альфа-векторами в Γ. Предварительно просматриваемое действие из убеждения b с использованием функции полезности, представленной Γ и обозначенной как $\mathcal{U}^{\Gamma}$:

$$\pi^{\Gamma}(b) = \arg\max_a \left[R(b,a) + \gamma \sum_o P(o|b,a)\mathcal{U}^{\Gamma}(\text{Update}(b,a,o)) \right], \tag{20.13}$$

где

$$P(o|b,a) = \sum_s P(o|s,a)b(s); \tag{20.14}$$

$$P(o|s,a) = \sum_{s'} T(s'|s,a)O(o|s',a). \tag{20.15}$$

Реализация показана в алгоритме 20.5. Пример 20.2 демонстрирует использование алгоритма пошагового предпросмотра в задаче о плачущем ребенке.

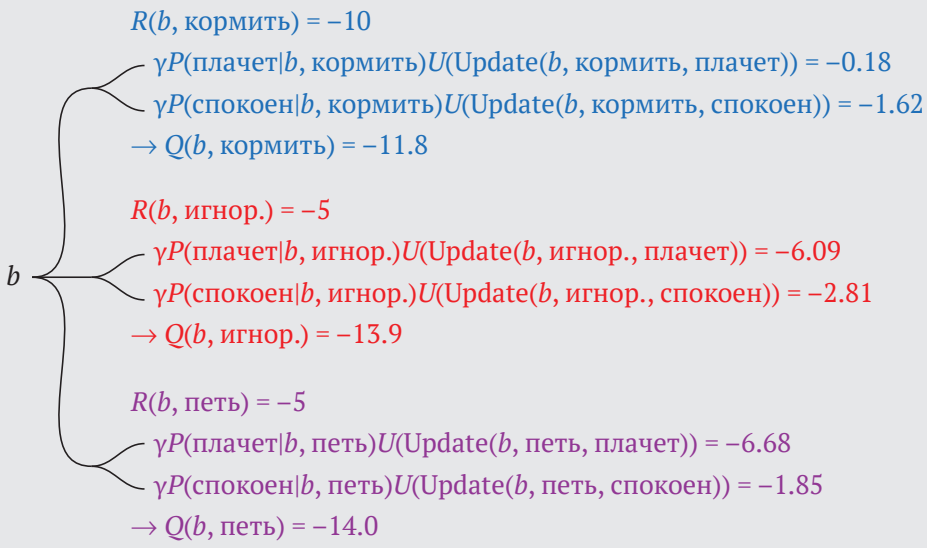
Алгоритм 20.5. Стратегия, представленная набором альфа-векторов Γ. Она использует пошаговый предпросмотр для получения оптимального действия и связанной с ним полезности. Уравнение (20.13) используется для вычисления предпросмотра

```
function lookahead(𝒫::POMDP, U, b::Vector, a)
    𝒮, 𝒪, T, O, R, γ = 𝒫.𝒮, 𝒫.𝒪, 𝒫.T, 𝒫.O, 𝒫.R, 𝒫.γ
    r = sum(R(s,a)*b[i] for (i,s) in enumerate(𝒮))
    Posa(o,s,a) = sum(O(a,s′,o)*T(s,a,s′) for s′ in 𝒮)
    Poba(o,b,a) = sum(b[i]*Posa(o,s,a) for (i,s) in enumerate(𝒮))
```

```
    return r + γ*sum(Poba(o,b,a)*U(update(b, 𝒫, a, o)) for o in 𝒪)
end

function greedy(𝒫::POMDP, U, b::Vector)
    u, a = findmax(a->lookahead(𝒫, U, b, a), 𝒫.𝒜)
    return (a=a, u=u)
end

struct LookaheadAlphaVectorPolicy
    𝒫 # задача POMDP
    Γ # альфа-векторы
end

function utility(π::LookaheadAlphaVectorPolicy, b)
    return maximum(α⋅b for α in π.Γ)
end

function greedy(π, b)
    U(b) = utility(π, b)
    return greedy(π.𝒫, U, b)
end

(π::LookaheadAlphaVectorPolicy)(b) = greedy(π, b).a
```

Пример 20.2. Применение стратегии предпросмотра к задаче о плачущем ребенке

Рассмотрим использование пошагового предпросмотра в задаче о плачущем ребенке с функцией полезности, заданной альфа-векторами [–3.7, –15] и [–2, –21]. Предположим, что наше текущее убеждение равно b = [0.5, 0.5]. Это означает, что согласно нашему убеждению вероятность того, что ребенок голоден, равна вероятности того, что он не голоден. Применим уравнение (20.13):



Здесь мы используем $Q(b, a)$ для обозначения функции ценности действия из состояния-убеждения. Стратегия предсказывает, что кормление ребенка приведет к наибольшей ожидаемой полезности, поэтому она указывает выполнить действие.

20.4. Сокращение

Если у нас есть набор альфа-векторов Γ, может возникнуть потребность в *сокращении* (pruning) альфа-векторов, которые не вносят вклад в представление функции полезности, или планов, которые не являются оптимальными для какого-либо убеждения. Удаление таких альфа-векторов или планов может повысить эффективность вычислений. Мы можем проверить, доминируют ли над альфа-вектором $\boldsymbol{\alpha}$ альфа-векторы в наборе Γ, выполнив линейную программу для максимизации разрыва полезности δ, которого достигает этот вектор по сравнению со всеми другими векторами[5]:

$$\begin{aligned} &\underset{\delta, \mathbf{b}}{\text{максимизировать}}\ \delta, \\ &\text{при условии что } \mathbf{b} \geqslant 0 \\ &\qquad \mathbf{1}^\top \mathbf{b} = 1; \\ &\qquad \boldsymbol{\alpha}^\top \geqslant \boldsymbol{\alpha}'^\top \mathbf{b} + \delta, \quad \boldsymbol{\alpha}' \in \Gamma. \end{aligned} \tag{20.16}$$

Первые два ограничения гарантируют, что $\mathbf{b}$ является категориальным распределением, а последний набор ограничений гарантирует, что мы найдем вектор убеждения, для которого $\boldsymbol{\alpha}$ имеет более высокое ожидаемое вознаграждение, чем все альфа-векторы в Γ. Если после выполнения линейной программы разрыв полезности δ отрицателен, то альфа-векторы в наборе Γ доминируют над $\boldsymbol{\alpha}$. Если значение δ положительно, то это не так, и $\mathbf{b}$ является убеждением, при котором альфа-векторы в наборе Γ не доминируют над $\boldsymbol{\alpha}$. Алгоритм 20.6 представляет реализацию решения уравнения (20.16) для определения убеждения (если оно существует), при котором δ имеет наибольшее положительное значение.

Алгоритм 20.6. Метод поиска вектора убеждения `b`, для которого альфа-вектор `α` улучшается больше всего по сравнению с набором альфа-векторов `Γ`. Если такого убеждения не существует, метод ничего не возвращает. Пакеты `JuMP.jl` и `GLPK.jl` предоставляют библиотеку математической оптимизации и решатель для линейных программ соответственно

```
function find_maximal_belief(α, Γ)
    m = length(α)
```

[5] Ограничения вида $\mathbf{a} \geqslant \mathbf{b}$ являются поэлементными. Это означает, что $a_i \geqslant b_i$ для всех i.

```
    if isempty(Γ)
        return fill(1/m, m) # произвольное убеждение
    end
    model = Model(GLPK.Optimizer)
    @variable(model, δ)
    @variable(model, b[i=1:m] ≥ 0)
    @constraint(model, sum(b) == 1.0)
    for a in Γ
        @constraint(model, (α-a)⋅b ≥ δ)
    end
    @objective(model, Max, δ)
    optimize!(model)
    return value(δ) > 0 ? value.(b) : nothing
end
```

В алгоритме 20.7 представлена процедура, использующая алгоритм 20.6 для нахождения доминирующих альфа-векторов в наборе Γ. Изначально все альфа-векторы являются кандидатами на доминирование. Затем мы выбираем одного из этих кандидатов и находим убеждение b, при котором кандидат приводит к наибольшему улучшению полезности по сравнению со всеми другими альфа-векторами в доминирующем наборе. Если кандидат не приносит улучшения, мы удаляем его из набора. Если он приносит улучшение, мы перемещаем альфа-вектор из набора кандидатов, который приносит наибольшее улучшение в точке b, в доминирующий набор. Процесс продолжается до тех пор, пока не останется кандидатов. Мы можем сократить любые альфа-векторы и связанные с ними условные планы, которые не преобладают ни в одной точке убеждения. Пример 20.3 демонстрирует сокращение задачи о плачущем ребенке.

Алгоритм 20.7. Метод нахождения доминирующих и доминируемых альфа-векторов и связанных с ними планов. Функция `find_dominating` идентифицирует все доминирующие альфа-векторы в наборе `Γ`. Она использует бинарные векторы `candidates` и `dominating`, чтобы отслеживать, какие альфа-векторы являются кандидатами на включение в доминирующий набор, а какие в настоящее время уже находятся в доминирующем наборе соответственно

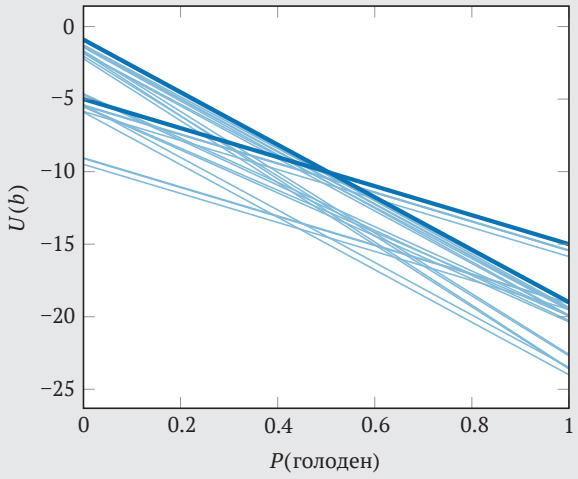
```
function find_dominating(Γ)
    n = length(Γ)
    candidates, dominating = trues(n), falses(n)
    while any(candidates)
        i = findfirst(candidates)
        b = find_maximal_belief(Γ[i], Γ[dominating])
        if b === nothing
            candidates[i] = false
        else
            k = argmax([candidates[j] ? b⋅Γ[j] : -Inf for j in 1:n])
            candidates[k], dominating[k] = false, true
```

```
        end
    end
    return dominating
end

function prune(plans, Γ)
    d = find_dominating(Γ)
    return (plans[d], Γ[d])
end
```

Пример 20.3. Ожидаемая полезность в пространстве убеждений для всех двухэтапных планов решения задачи о плачущем ребенке (приложение F.7). Толстые линии показывают оптимумы для некоторых убеждений, тогда как тонкие линии обозначают доминируемые планы

Мы можем построить все двухшаговые планы для задачи о плачущем ребенке. Таких планов $3^3 = 27$.

Ожидаемая полезность каждого плана в пространстве убеждений представлена на графике ниже. Мы видим, что два плана доминируют над всеми остальными. Эти доминирующие планы являются единственными, которые необходимо рассматривать как подпланы для оптимальных трехэтапных планов.



20.5. Итерация по полезности

Алгоритм итерации по полезности для MDP может быть адаптирован к POMDP[6]. *Итерация по полезности* в задаче POMDP (алгоритм 20.8) начинается с построения всех одношаговых планов. Мы отбрасываем планы, которые никогда не бывают оптимальными для любого начального убеждения. Затем расширяем все комбинации одношаговых планов, чтобы получить двухшаговые планы. И снова исключаем любые неоптимальные планы из рассмотрения. Эта процедура чередования расширения и сокращения повторяется до тех пор, пока не будет достигнут желаемый горизонт. На рис. 20.3 показана итерация полезности в задаче о плачущем ребенке.

Алгоритм 20.8. Итерация по полезности для POMDP, которая находит доминирующие h-шаговые планы для POMDP с конечным горизонтом `k_max` путем итеративного построения оптимальных подпланов. Структура `ValueIteration` такая же, как определена в алгоритме 7.8 в контексте MDP

```
function value_iteration(𝒫::POMDP, k_max)
    𝒮, 𝒜, R = 𝒫.𝒮, 𝒫.𝒜, 𝒫.R
    plans = [ConditionalPlan(a) for a in 𝒜]
    Γ = [[R(s,a) for s in 𝒮] for a in 𝒜]
    plans, Γ = prune(plans, Γ)
    for k in 2:k_max
        plans, Γ = expand(plans, Γ, 𝒫)
        plans, Γ = prune(plans, Γ)
    end
    return (plans, Γ)
end

function solve(M::ValueIteration, 𝒫::POMDP)
    plans, Γ = value_iteration(𝒫, M.k_max)
    return LookaheadAlphaVectorPolicy(𝒫, Γ)
end
```

---

[6] В этом разделе описывается версия итерации по полезности с точки зрения условных планов и альфа-векторов. Для версии, которая использует только альфа-векторы, см. A. R. Cassandra, M. L. Littman, N. L. Zhang, *Incremental Pruning: A Simple, Fast, Exact Method for Partially Observable Markov Decision Processes*, in Conference on Uncertainty in Artificial Intelligence (UAI), 1997.

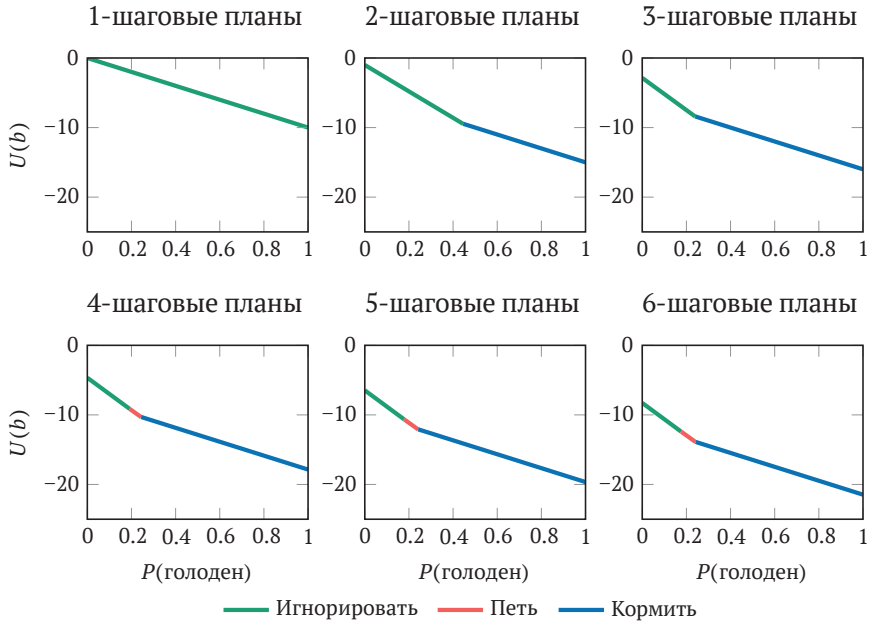


Рис. 20.3. Итерация по полезности в POMDP, используемая для поиска оптимальной функции полезности в задаче о плачущем ребенке с различными горизонтами

Шаг расширения (алгоритм 20.9) в этом процессе строит все возможные $(k + 1)$-шаговые планы из набора k-шаговых планов. Новые планы могут быть построены с использованием нового первого действия и всех возможных комбинаций k-шаговых планов в качестве подпланов, как показано на рис. 20.4. Хотя планы также могут быть расширены путем добавления действий в концы подпланов, расширение верхнего уровня позволяет использовать альфа-векторы, созданные для k-шаговых планов, для эффективного построения альфа-векторов для $(k + 1)$-шаговых планов.

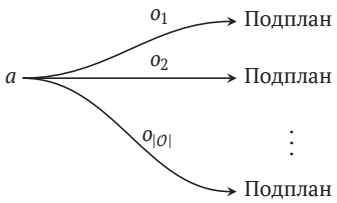


Рис. 20.4. План из $(k + 1)$ шагов можно построить с использованием нового начального действия, ведущего к любой комбинации подпланов из k шагов

Вычисление альфа-вектора, ассоциированного с планом π, из набора альфа-векторов, ассоциированных с его подпланами, можно выполнить следующим образом. Воспользуемся обозначением $\boldsymbol{\alpha}_o$ для представления альфа-вектора, ассоциированного с подпланом $\pi(o)$. Отсюда альфа-вектор, ассоциированный с π, равен

$$\alpha(s) = R(s, \pi()) + \gamma \sum_{s'} T(s'|s, \pi()) \sum_{o} O(o|\pi(), s') \alpha_o(s'). \tag{20.17}$$

Даже для относительно простых задач с небольшой глубиной вычисление альфа-векторов из подпланов таким способом намного эффективнее, чем их вычисление с нуля, как в алгоритме 20.2.

Алгоритм 20.9. Шаг расширения в итерации по полезности, который строит все (k + 1)-шаговые условные планы и связанные альфа-векторы из набора k-шаговых условных планов и альфа-векторов. Способ, которым мы объединяем альфа-векторы подпланов, основан на уравнении (20.17)

```
function ConditionalPlan(𝒫::POMDP, a, plans)
    subplans = Dict(o=>π for (o, π) in zip(𝒫.𝒪, plans))
    return ConditionalPlan(a, subplans)
end

function combine_lookahead(𝒫::POMDP, s, a, Γo)
    𝒮, 𝒪, T, O, R, γ = 𝒫.𝒮, 𝒫.𝒪, 𝒫.T, 𝒫.O, 𝒫.R, 𝒫.γ
    U′(s′,i) = sum(O(a,s′,o)*α[i] for (o,α) in zip(𝒪,Γo))
    return R(s,a) + γ*sum(T(s,a,s′)*U′(s′,i) for (i,s′) in enumerate(𝒮))
end

function combine_alphavector(𝒫::POMDP, a, Γo)
    return [combine_lookahead(𝒫, s, a, Γo) for s in 𝒫.𝒮]
end

function expand(plans, Γ, 𝒫)
    𝒮, 𝒜, 𝒪, T, O, R = 𝒫.𝒮, 𝒫.𝒜, 𝒫.𝒪, 𝒫.T, 𝒫.O, 𝒫.R
    plans′, Γ′ = [], []
    for a in 𝒜
        # итерации по всем возможным отображениям из наблюдений в планы
        for inds in product([eachindex(plans) for o in 𝒪]...)
            πo = plans[[inds...]]
            Γo = Γ[[inds...]]
            π = ConditionalPlan(𝒫, a, πo)
            α = combine_alphavector(𝒫, a, Γo)
            push!(plans′, π)
            push!(Γ′, α)
        end
    end
    return (plans′, Γ′)
end
```

20.6. Линейные стратегии

Как было сказано в разделе 19.3, состояние-убеждение в задаче с линейной по Гауссу динамикой может быть представлено гауссовым распределением

$\mathcal{N}(\boldsymbol{\mu}_b, \boldsymbol{\Sigma}_b)$. Если функция вознаграждения является квадратичной, то можно показать, что оптимальная стратегия может быть точно вычислена в офлайн-режиме с использованием процесса, который часто называют *линейно-квадратичным гауссовым управлением* (linear quadratic Gaussian control, LQG control). Оптимальное действие получается таким же образом, как и в разделе 7.8, но $\boldsymbol{\mu}_b$, вычисленное с использованием линейного гауссова фильтра, рассматривается как истинное состояние[7]. С каждым наблюдением мы просто используем фильтр для обновления нашего $\boldsymbol{\mu}_b$ и получаем оптимальное действие, умножая $\boldsymbol{\mu}_b$ на матрицу стратегии из алгоритма 7.11. Пример 20.4 демонстрирует этот процесс.

Пример 20.4. Оптимальная стратегия, используемая для POMDP с линейной по Гауссу динамикой и квадратичным вознаграждением

Рассмотрим спутник, перемещающийся в двух измерениях, пренебрегая гравитацией, сопротивлением и другими внешними силами. Спутник может использовать свои двигатели для ускорения в любом направлении с линейной динамикой:

$$\begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \end{bmatrix} \leftarrow \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \end{bmatrix} + \begin{bmatrix} \frac{1}{2}\Delta t^2 & 0 \\ 0 & \frac{1}{2}\Delta t^2 \\ \Delta t & 0 \\ 0 & \Delta t \end{bmatrix} \begin{bmatrix} \ddot{x} \\ \ddot{y} \end{bmatrix} + \boldsymbol{\varepsilon},$$

где Δt – длительность шага времени, а ε – гауссов шум с нулевым средним и ковариацией $\Delta t/20\mathbf{I}$.

Мы стремимся разместить спутник на его орбитальной позиции, которая служит началом координат, минимизировав при этом расход топлива. Наша квадратичная функция вознаграждения:

$$R(\mathbf{s}, \mathbf{a}) = -\mathbf{s}^\top \begin{bmatrix} \mathbf{I}_{2\times 2} & \mathbf{0}_{2\times 2} \\ \mathbf{0}_{2\times 2} & \mathbf{0}_{2\times 2} \end{bmatrix} \mathbf{s} - 2\mathbf{a}^\top \mathbf{a}.$$

Датчики спутника измеряют его положение в соответствии с уравнением:

$$\mathbf{o} = \begin{bmatrix} \mathbf{I}_{2\times 2} & \mathbf{0}_{2\times 2} \end{bmatrix} \mathbf{s} + \boldsymbol{\varepsilon},$$

где $\boldsymbol{\varepsilon}$ – гауссов шум с нулевым средним и ковариацией $\Delta t/10\mathbf{I}$.

Ниже показаны 50 траекторий, сформированных из 10-шаговых развертываний с использованием оптимальной стратегии для $\Delta t = 1$ и фильтра Калмана для отслеживания убеждений. В каждом случае спутник запускался из состояния $\mathbf{s} = \boldsymbol{\mu}_b = [-5, 2, 0, 1]$ с $\boldsymbol{\Sigma}_b = [\mathbf{I}\ 0;\ \ \mathbf{0}\ 0.25\mathbf{I}]$.

[7] Возможность просто использовать среднее значение распределения – еще один пример принципа *эквивалентности достоверности*, впервые представленного в разделе 7.8.

20.7. Заключение

- Точные решения задачи POMDP обычно могут быть получены только для дискретных POMDP с конечным горизонтом.
- Стратегии для этих задач могут быть представлены в виде условных древовидных планов, описывающих действия, которые необходимо предпринять на основе наблюдений.
- Альфа-векторы содержат ожидаемую полезность при запуске из разных состояний и следовании определенному условному плану.
- Альфа-векторы также могут служить альтернативным представлением стратегии POMDP.
- Итерация по полезности POMDP позволяет избежать вычислительной нагрузки, связанной с перебором всех условных планов, путем итеративного вычисления подпланов и сокращения тех из них, которые неоптимальны.
- Линейные по Гауссу задачи с квадратичным вознаграждением могут быть точно решены с использованием методов, очень похожих на те, что получены для полностью наблюдаемого случая.

20.8. Упражнения

Упражнение 20.1. Можно ли каждый POMDP оформить как MDP?

Решение. Да. Любой POMDP можно эквивалентно рассматривать как MDP типа «состояние-убеждение», пространство состояний которого является пространством убеждений в POMDP, чье пространство действий такое же, как у POMDP, и чья функция перехода задается уравнением (20.2).

Упражнение 20.2. Каковы альфа-векторы для одношаговой задачи о плачущем ребенке (приложение F.7)? Все ли доступные действия доминантны?

Решение. Есть три одношаговых условных плана, по одному для каждого действия, что дает три альфа-вектора. Оптимальная одношаговая стратегия должна выбирать между этими действиями, исходя из текущего убеждения. Одношаговые альфа-векторы для POMDP могут быть получены из оптимальной одношаговой функции полезности убеждения:

$$\mathcal{U}^*(b) = \max_a \sum_s b(s)R(s,a).$$

Ожидаемое вознаграждение за кормление ребенка:

$$\begin{aligned} &R(\text{голодный, кормить})P(\text{голодный}) + R(\text{сытый, кормить})P(\text{сытый}) \\ &\quad = -15P(\text{голодный}) - 5(1 - P(\text{голодный})) \\ &\quad = -10P(\text{голодный}) - 5. \end{aligned}$$

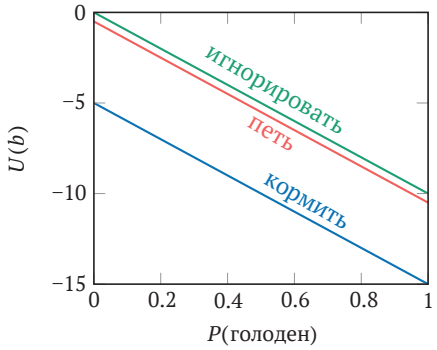
Ожидаемое вознаграждение за пение колыбельной:

$$R(\text{голодный, петь})P(\text{голодный}) + R(\text{сытый, петь})P(\text{сытый})$$
$$= -10.5P(\text{голодный}) - 0.5(1 - P(\text{голодный}))$$
$$= -10P(\text{голодный}) - 0.5.$$

Ожидаемое вознаграждение за игнорирование ребенка:

$$R(\text{голодный, игнорировать})P(\text{голодный}) + R(\text{сытый, игнорировать})P(\text{сытый})$$
$$= -10P(\text{голодный}).$$

Ожидаемое вознаграждение за каждое действие отображается в пространстве убеждений следующим образом:



Из графика следует, что при горизонте в один шаг кормление или пение никогда не бывает оптимальным. Доминирующим действием является игнорирование.

Упражнение 20.3. Почему реализация итерации по полезности в алгоритме 20.8 вызывает метод `expand` в алгоритме 20.9, вместо того чтобы выполнить оценку плана по алгоритму 20.2 для получения альфа-векторов для каждого нового условного плана?

Решение. Метод оценки плана рекурсивно применяет уравнение (20.8) для оценки ожидаемой полезности условного плана. По мере увеличения горизонта условные планы становятся очень большими. Итерация по полезности POMDP помогает сэкономить вычисления, используя альфа-векторы для подпланов из предыдущей итерации:

$$\mathcal{U}^{\pi}(s) = R(s, \pi()) + \gamma \left[\sum_{s'} T(s'|s, \pi()) \sum_{o} O(o|\pi(), s') \alpha_{s'}^{\pi(o)} \right].$$

Упражнение 20.4. От чего быстрее увеличивается количество условных планов – от роста количества действий или наблюдений?

Решение. Напомним, что существует $|\mathcal{A}|^{(|\mathcal{O}|^h-1)/(|\mathcal{O}|-1)}$ возможных h-шаговых планов. Экспоненциальный рост (n^x) происходит быстрее, чем полиномиальный рост (x^n), и мы имеем суперэкспоненциальный рост по $|\mathcal{O}|$ и полиномиальный рост по $|\mathcal{A}|$. Таким образом, количество планов намного сильнее зависит от количества наблюдений. Чтобы продемонстрировать этот эффект, давайте возьмем значения $|\mathcal{A}| = 3$, $|\mathcal{O}| = 3$ и $h = 3$ в качестве исходных условий, согласно которым получается 1 594 323 плана. Увеличение количества действий дает 67 108 864 плана, тогда как увеличение количества наблюдений дает 10 460 353 203 плана.

Упражнение 20.5. Предположим, что у нас есть пациент и мы не уверены в наличии у него конкретного заболевания. У нас также есть три диагностических теста, каждый с разной вероятностью достоверного выявления болезни. Пока пациент находится в нашем кабинете, у нас есть возможность последовательно провести несколько диагностических тестов. Мы немедленно наблюдаем результат каждого теста. Кроме того, любой диагностический тест можно повторять многократно, при этом результаты всех тестов условно независимы друг от друга и зависят только от наличия или отсутствия заболевания. Выполнив анализ, мы должны принять решение, следует ли лечить пациента. Какими, на ваш взгляд, будут компоненты задачи POMDP?

Решение. У нас есть три состояния:

1) $s_{\text{no-disease}}$: у пациента нет заболевания;
2) s_{disease}: у пациента есть заболевание;
3) s_{terminal}: взаимодействие завершено (конечное состояние).

У нас есть пять действий:

1) a_1: провести тест 1;
2) a_2: провести тест 2;
3) a_3: провести тест 3;
4) a_{treat}: назначить лечение и отправить пациента домой;
5) a_{stop}: отправить пациента домой без лечения.

У нас есть три наблюдения:

1) $o_{\text{no-disease}}$: результат теста (если он проводится) указывает на то, что у пациента нет заболевания;
2) o_{disease}: результат теста (если он проводится) указывает на то, что у пациента есть заболевание;
3) $o_{\text{терминал}}$: тест не проводился.

Модель перехода будет детерминированной, со следующими условиями:

$$T(s'|s,a) = \begin{cases} 1, & \text{если } a \in \{a_{\text{treat}}, a_{\text{stop}}\} \wedge s' = s_{\text{terminal}} \\ 1, & \text{если } s = s' \\ 0 & \text{в остальных случаях} \end{cases}.$$

Функция вознаграждения представляет собой функцию от стоимости проведения лечения и стоимости каждого теста, а также стоимости отказа от лечения болезни, если пациент действительно болен. Вознаграждение, доступное из состояния s_{terminal}, равно 0. Модель наблюдения присваивает вероятности правильным и неправильным наблюдениям за наличием болезни в результате диагностического теста, выполняемого из состояния, которое не является конечным. Первоначальное убеждение будет содержать априорную вероятность наличия болезни, с нулевой вероятностью, назначенной конечному состоянию.

Упражнение 20.6. Почему может возникнуть потребность несколько раз выполнить один и тот же тест в предыдущем упражнении?

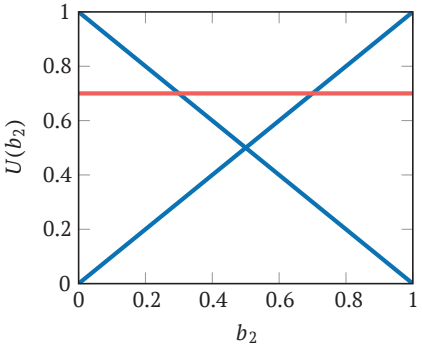
Решение. В зависимости от вероятности получения неверных результатов мы можем провести один и тот же тест несколько раз, чтобы повысить нашу уверенность в том, есть ли у пациента заболевание. Результаты тестов не зависят от состояния s_{disease}.

Упражнение 20.7. Предположим, у нас есть три альфа-вектора [1, 0], [0, 1] и [θ, θ] для константы θ. Каким условиям должна соответствовать θ, чтобы мы могли обрезать альфа-векторы?

Решение. Мы можем обрезать альфа-векторы, если $\theta < 0.5$ или $\theta > 1$. В первом случае над [θ, θ] доминируют два других альфа-вектора. Если $\theta > 1$, то [θ, θ] доминирует над двумя другими альфа-векторами.

Упражнение 20.8. Пусть дано Γ = {[1, 0], [0, 1]} и $\boldsymbol{\alpha} = [0.7, 0.7]$. Какое убеждение $\mathbf{b}$ максимизирует разрыв полезности δ, определяемый линейной программой в уравнении (20.16)?

Решение. На рисунке ниже альфа-векторы в Γ показаны синим цветом, а альфа-вектор $\boldsymbol{\alpha}$ показан красным. Нас интересует только область $0.3 \leqslant b_2 \leqslant 0.7$, где $\boldsymbol{\alpha}$ доминирует над альфа-векторами в Γ – когда красная линия находится над синими линиями. Максимальное расстояние между красной линией и двумя синими наблюдается в точке $b_2 = 0.5$, с разрывом $\delta = 0.2$. Следовательно, разрыв максимален при $\mathbf{b} = [0.5, 0.5]$.



21 Офлайн-планирование с использованием убеждений-состояний

В худшем случае точное решение обобщенной задачи POMDP с конечным горизонтом является PSPACE-полным. Это класс сложности, который охватывает NP-полные задачи и, как предполагается, включает в себя еще более сложные задачи[1]. Было показано, что обобщенные POMDP с бесконечным горизонтом невычислимы[2]. По этой причине в последнее время было проведено огромное количество исследований, направленных на поиск методов приближенного решения. В этой главе мы рассмотрим различные офлайн-методы решения POMDP, которые подразумевают выполнение всех или большей части вычислений до выполнения действий. Мы сосредоточимся на методах, которые представляют функцию полезности в виде альфа-векторов, и на различных формах интерполяции.

21.1. Аппроксимация полностью наблюдаемой полезности

Одним из самых простых методов офлайн-аппроксимации является QMDP, название которого происходит от функции полезности действия, связанной с полностью наблюдаемым MDP[3]. Этот подход, как и несколько других, обсуждаемых в данной главе, предусматривает итеративное обновление набора Γ альфа-векторов, как показано в алгоритме 21.1. Результирующий набор Γ

---

1 C. Papadimitriou, J. Tsitsiklis, *The Complexity of Markov Decision Processes*, Mathematics of Operation Research, vol. 12, no. 3, pp. 441–450, 1987.

2 O. Madani, S. Hanks, A. Condon, *On the Undecidability of Probabilistic Planning and Related Stochastic Optimization Problems*, Artificial Intelligence, vol. 147, no. 1–2, pp. 5–34, 2003.

3 M. L. Littman, A. R. Cassandra, L. P. Kaelbling, *Learning Policies for Partially Observable Environments: Scaling Up*, in International Conference on Machine Learning (ICML), 1995. Доказательство того, что QMDP дает верхнюю границу оптимальной функции полезности, приведено в публикации M. Hauskrecht, *Value-Function Approximations for Partially Observable Markov Decision Processes*, Journal of Artificial Intelligence Research, vol. 13, pp. 33–94, 2000.

определяет функцию полезности и стратегию, которые можно использовать напрямую или с пошаговым предпросмотром, как обсуждалось в предыдущей главе, хотя результирующая стратегия будет лишь приближением к оптимальному решению.

Алгоритм 21.1. Итеративная структура для обновления набора альфа-векторов `Γ`, используемая несколькими методами в этой главе. Различные методы, включая QMDP, отличаются реализацией `update`. После `k_max` итераций эта функция возвращает стратегию, представленную альфа-векторами в `Γ`

```
function alphavector_iteration(𝒫::POMDP, M, Γ)
    for k in 1:M.k_max
        Γ = update(𝒫, M, Γ)
    end
    return Γ
end
```

QMDP (алгоритм 21.2) строит один альфа-вектор $\boldsymbol{\alpha}_a$ для каждого действия a, используя итерацию по полезности. Каждый альфа-вектор инициализируется нулевым значением, а затем мы выполняем следующие итерации:

$$\alpha_a^{(k+1)}(s) = R(s,a) + \gamma \sum_{s'} T(s'|s,a) \max_{a'} \alpha_{a'}^{(k)}(s'). \tag{21.1}$$

Для каждой итерации требуется $\mathcal{O}(|\mathcal{A}|^2|\mathcal{S}|^2)$ операций. На рис. 21.1 показана иллюстрация этого процесса.

Алгоритм 21.2. Реализация метода QMDP, который находит приблизительно оптимальную стратегию для POMDP с бесконечным горизонтом с дискретным пространством состояний и действий, где `k_max` – количество итераций. QMDP предполагает идеальную наблюдаемость

```
struct QMDP
    k_max # максимальное количество итераций
end

function update(𝒫::POMDP, M::QMDP, Γ)
    𝒮, 𝒜, R, T, γ = 𝒫.𝒮, 𝒫.𝒜, 𝒫.R, 𝒫.T, 𝒫.γ
    Γ′ = [[R(s,a) + γ*sum(T(s,a,s′)*maximum(α′[j] for α′ in Γ)
        for (j,s′) in enumerate(𝒮)) for s in 𝒮] for a in 𝒜]
    return Γ′
end

function solve(M::QMDP, 𝒫::POMDP)
    Γ = [zeros(length(𝒫.𝒮)) for a in 𝒫.𝒜]
    Γ = alphavector_iteration(𝒫, M, Γ)
    return AlphaVectorPolicy(𝒫, Γ, 𝒫.𝒜)
end
```

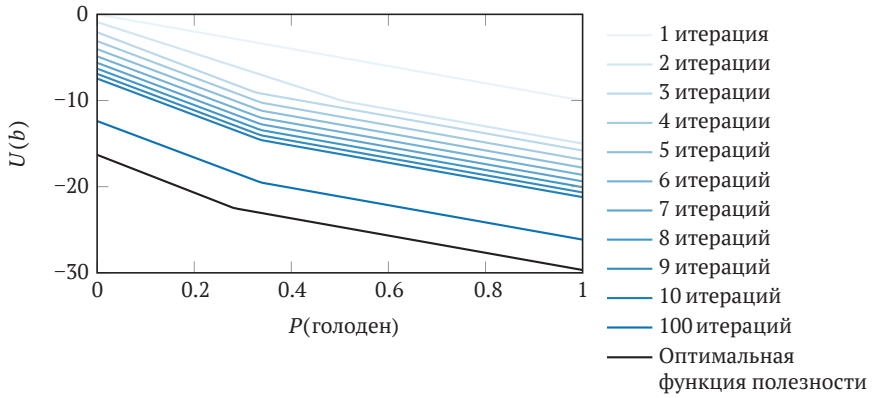


Рис. 21.1. Функции полезности, полученные для задачи о плачущем ребенке (приложение F.7) с использованием QMDP. В первой итерации доминирует один альфа-вектор. В последующих итерациях доминируют два альфа-вектора

Когда QMDP выполняется до горизонта в задачах с конечным горизонтом или до сходимости для задач с бесконечным горизонтом, результирующая стратегия эквивалентна предположению, что после первого шага будет полная наблюдаемость. Поскольку мы можем добиться истинно оптимального результата, только если у нас есть полная наблюдаемость, QMDP будет представлять собой *оценку сверху* (верхнюю границу) истинной оптимальной функции полезности $U^*(\mathbf{b})$. Другими словами, $\max_a \boldsymbol{\alpha}_a^\top \mathbf{b} \geq U^*(\mathbf{b})$ для всех $\mathbf{b}$[4].

Если QMDP не сходится в задаче с бесконечным горизонтом, он может не дать достоверную оценку сверху. Один из способов гарантировать, что QMDP даст искомый результат после конечного числа итераций, состоит в том, чтобы изначально инициализировать функцию полезности некоторой верхней границей. Один из возможных вариантов начальной верхней границы – это оценка сверху по *наилучшему действию и наилучшему состоянию* (best-action best-state), которая представляет собой полезность, полученную от бесконечного выполнения наилучшего действия из наилучшего состояния:

$$\bar{U}(b) = \max_{s,a} \frac{R(s,a)}{1-\gamma}. \tag{21.2}$$

Предположение о полной наблюдаемости после первого шага может привести к тому, что QMDP будет плохо аппроксимировать полезность *действий по сбору информации* (information-gathering action), которые значительно уменьшают неопределенность в текущем состоянии. Например, когда вы оглядываетесь через плечо, паркуя автомобиль задним ходом, – это действие по сбору информации. QMDP хорошо работает в тех случаях, когда оптимальная стратегия не предусматривает дорогостоящий сбор информации.

[4] Хотя функция полезности, представленная альфа-векторами QMDP, ограничивает верхнюю границу оптимальной функции, полезность, реализуемая стратегией QMDP, конечно, не будет превышать ожидаемой полезности оптимальной стратегии.

Метод QMDP поддается обобщению на задачи, которые могут не иметь маленького дискретного пространства состояний. В таких задачах итерация согласно уравнению (21.1) не всегда выполнима, но мы можем использовать один из многих методов, рассмотренных в предыдущих главах, для получения аппроксимированной функции полезности действия $Q(s, a)$. Эта функция полезности может быть определена в многомерном непрерывном пространстве состояний с использованием, например, представления в форме нейронной сети. Тогда функция полезности, вычисляемая в *точке убеждения* b (belief point), равна

$$U(b) = \max_a \int Q(s,a)b(s)\mathrm{d}s. \tag{21.3}$$

Этот интеграл может быть аппроксимирован с помощью дискретного представления.

21.2. Метод быстрой инфограницы

Как и в случае с QMDP, метод *быстрой инфограницы* (fast informed bound) вычисляет один альфа-вектор для каждого действия. Однако данный метод в некоторой степени учитывает модель наблюдения[5]. Уравнение итерации имеет вид

$$\alpha_a^{(k+1)}(s) = R(s,a) + \gamma \sum_o \max_{a'} \sum_{s'} O(o|a,s')T(s'|s,a)\alpha_{a'}^{(k)}(s') \tag{21.4}$$

и дает вычислительную нагрузку $\mathcal{O}(|\mathcal{A}|^2|\mathcal{S}|^2|\mathcal{O}|)$ операций на итерацию.

Метод быстрой инфограницы тоже дает оценку сверху для оптимальной функции полезности. Эта оценочная граница гарантированно не слабее, чем та, которую дает QMDP, скорее, даже более строгая. Данный метод реализован в алгоритме 21.3, а его применение для вычисления оптимальной функции полезности показано на рис. 21.2.

Алгоритм 21.3. Реализация метода быстрой инфограницы, который находит приблизительно оптимальную стратегию для задачи POMDP с бесконечным горизонтом с дискретными пространствами состояний, действий и наблюдений, где `k_max` – количество итераций

```
struct FastInformedBound
    k_max # максимальное количество итераций
end
```

---

[5] Взаимосвязь между методами QMDP и быстрой инфограницы вместе с эмпирическими результатами обсуждается в работе M. Hauskrecht, *Value-Function Approximations for Partially Observable Markov Decision Processes*, Journal of Artificial Intelligence Research, vol. 13, pp. 33–94, 2000.

```
function update(𝒫::POMDP, M::FastInformedBound, Γ)
    𝒮, 𝒜, 𝒪, R, T, O, γ = 𝒫.𝒮, 𝒫.𝒜, 𝒫.𝒪, 𝒫.R, 𝒫.T, 𝒫.O, 𝒫.γ
    Γ′ = [[R(s, a) + γ*sum(maximum(sum(O(a,s′,o)*T(s,a,s′)*α′[j]
        for (j,s′) in enumerate(𝒮)) for α′ in Γ) for o in 𝒪)
        for s in 𝒮] for a in 𝒜]
    return Γ′
end

function solve(M::FastInformedBound, 𝒫::POMDP)
    Γ = [zeros(length(𝒫.𝒮)) for a in 𝒫.𝒜]
    Γ = alphavector_iteration(𝒫, M, Γ)
    return AlphaVectorPolicy(𝒫, Γ, 𝒫.𝒜)
end
```

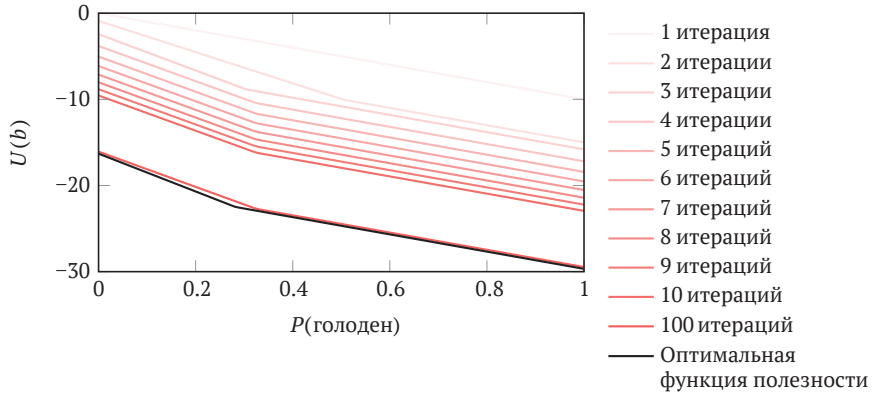


Рис. 21.2. Функции полезности, полученные для задачи о плачущем ребенке с использованием метода быстрой инфограницы. Функция полезности после 10 итераций заметно ниже, чем у алгоритма QMDP

21.3. Методы быстрой оценки снизу

В предыдущих двух разделах были рассмотрены методы, которые можно использовать для оценки сверху функции полезности, представленной в виде альфа-векторов. В этом разделе мы рассмотрим два метода для быстрой оценки снизу функций полезности, представленных в виде альфа-векторов, без какого-либо планирования в пространстве убеждений. В то время как методы оценки сверху часто напрямую используют для создания разумной стратегии, методы *оценки снизу*, обсуждаемые далее, обычно применяются только в качестве отправной точки для других алгоритмов планирования. На рис. 21.3 показаны два метода оценки снизу, упомянутых в этом разделе.

Распространенным методом оценки снизу является *нижняя граница наилучшего действия в наихудшем состоянии* (best-action worst-state, BAWS) (алго-

ритм 21.4). Это дисконтированное вознаграждение, всегда получаемое за лучшее действие в худшем состоянии:

$$r_{\text{baws}} = \max_a \sum_{k=1}^{\infty} \gamma^{k-1} \min_s R(s,a) = \frac{1}{1-\gamma} \max_a \min_s R(s,a). \tag{21.5}$$

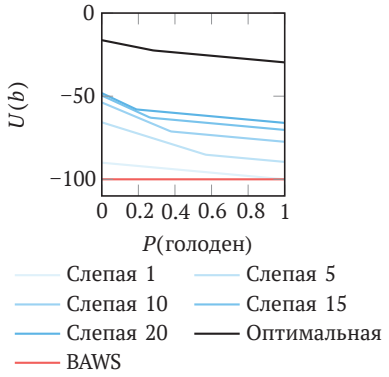


Рис. 21.3. Слепая оценка снизу с разным числом итераций и оценка BAWS применительно к задаче о плачущем ребенке

Эта нижняя граница представлена одним альфа-вектором. Обычно она очень нечеткая, но ее можно использовать как отправную точку для других алгоритмов, которые дают более строгую оценку снизу. Мы расскажем об этом немного позже.

Алгоритм 21.4. Реализация метода BAWS из уравнения (21.5), представленного в виде альфа-вектора

```
function baws_lowerbound(𝒫::POMDP)
    𝒮, 𝒜, R, γ = 𝒫.𝒮, 𝒫.𝒜, 𝒫.R, 𝒫.γ
    r = maximum(minimum(R(s, a) for s in 𝒮) for a in 𝒜) / (1-γ)
    α = fill(r, length(𝒮))
    return α
end
```

Слепая оценка снизу (blind lower bound), реализованная в алгоритме 21.5, представляет собой нижнюю границу с одним альфа-вектором на действие. Этот подход основан на предположении, что мы вынуждены всегда совершать одно действие, слепое к будущим наблюдениям. Чтобы вычислить упомянутые альфа-векторы, мы начинаем с другой оценки снизу (обычно это BAWS), а затем выполняем ряд итераций:

$$\alpha_a^{(k+1)}(s) = R(s,a) + \gamma \sum_{s'} T(s'|s,a)\alpha_a^{(k)}(s'). \tag{21.6}$$

Итерация согласно уравнению (21.6) аналогична обновлению QMDP в уравнении (21.1), за исключением того, что она не содержит максимизации по альфа-векторам в правой части.

Алгоритм 21.5. Реализация слепой оценки снизу, представленной в виде набора альфа-векторов

```
function blind_lowerbound(𝒫, k_max)
    𝒮, 𝒜, T, R, γ = 𝒫.𝒮, 𝒫.𝒜, 𝒫.T, 𝒫.R, 𝒫.γ
    Q(s,a,α) = R(s,a) + γ*sum(T(s,a,s′)*α[j] for (j,s′) in enumerate(𝒮))
    Γ = [baws_lowerbound(𝒫) for a in 𝒜]
    for k in 1:k_max
        Γ = [[Q(s,a,α) for s in 𝒮] for (α,a) in zip(Γ, 𝒜)]
    end
    return Γ
end
```

21.4. Точечная итерация по полезности

Методы QMDP и быстрой инфограницы генерируют по одному альфа-вектору для каждого действия, но оптимальная функция полезности часто лучше аппроксимируется гораздо большим количеством альфа-векторов. Алгоритм *точечной итерации по полезности* (point-based value iteration)[6] вычисляет m различных альфа-векторов $\Gamma = \{\boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_m\}$, каждый из которых ассоциирован с разными точками из набора точек в пространстве убеждения $\mathcal{B} = \{\mathbf{b}_1, \ldots, \mathbf{b}_m\}$. Методы выбора этих убеждений мы рассмотрим в разделе 21.7. Как и прежде, эти альфа-векторы определяют приблизительно оптимальную функцию полезности:

$$U^{\Gamma}(\mathbf{b}) = \max_{\boldsymbol{\alpha} \in \Gamma} \boldsymbol{\alpha}^{\top} \mathbf{b}. \tag{21.7}$$

Алгоритм использует нижнюю границу оптимальной функции полезности $U^{\Gamma}(\mathbf{b}) \leqslant U^{*}(\mathbf{b})$ для всех убеждений $\mathbf{b}$. Альфа-векторы инициализируют так, чтобы они начинались с нижней границы, а затем выполняют *дублирование* (backup) для обновления альфа-векторов в каждой точке $\mathcal{B}$. Операция дублирования (алгоритм 21.6) получает убеждение $\mathbf{b}$ и набор альфа-векторов Γ и строит новый альфа-вектор. Алгоритм итеративно проходит по всем возможным действиям a и наблюдениям o и извлекает из Γ альфа-вектор, который является максимальным в результирующем состоянии этого убеждения:

[6] Обзор точечных методов итерации по полезности предоставлен G. Shani, J. Pineau, R. Kaplow, *A Survey of Point-Based POMDP Solvers*, Autonomous Agents and Multi-Agent Systems, vol. 27, pp. 1–51, 2012. В этой публикации предложен несколько иной способ дублирования, хотя результат тот же.

$$\boldsymbol{\alpha}_{a,o} = \arg\max_{\boldsymbol{\alpha}\in\Gamma} \boldsymbol{\alpha}^\top \text{Update}(\mathbf{b}, a, o). \tag{21.8}$$

Затем на основе этих векторов $\boldsymbol{\alpha}_{a,o}$ для каждого доступного действия a создают новый альфа-вектор:

$$\alpha_a(s) = R(s,a) + \gamma \sum_{s',o} O(o|a,s')T(s'|s,a)\alpha_{a,o}(s'). \tag{21.9}$$

Альфа-вектор, который в конечном итоге создается операцией дублирования, равен

$$\boldsymbol{\alpha} = \arg\max_{\boldsymbol{\alpha}_a} \boldsymbol{\alpha}^\top \mathbf{b}. \tag{21.10}$$

Если Γ является нижней границей, операция дублирования создаст только альфа-векторы, которые также являются нижней границей.

Алгоритм 21.6. Операция дублирования для задачи POMDP с дискретными пространствами состояний и действий, где `Γ` – вектор альфа-векторов, а `b` – вектор убеждений, к которому применяется дублирование. Метод `update` определен в алгоритме 19.2

```
function backup(𝒫::POMDP, Γ, b)
    𝒮, 𝒜, 𝒪, γ = 𝒫.𝒮, 𝒫.𝒜, 𝒫.𝒪, 𝒫.γ
    R, T, O = 𝒫.R, 𝒫.T, 𝒫.O
    Γa = []
    for a in 𝒜
        Γao = []
        for o in 𝒪
            b′ = update(b, 𝒫, a, o)
            push!(Γao, argmax(α->α⋅b′, Γ))
        end
        α = [R(s, a) + γ*sum(sum(T(s, a, s′)*O(a, s′, o)*Γao[i][j]
            for (j,s′) in enumerate(𝒮)) for (i,o) in enumerate(𝒪))
            for s in 𝒮]
        push!(Γa, α)
    end
    return argmax(α->α⋅b, Γa)
end
```

Повторное применение операции дублирования к убеждениям в $\mathcal{B}$ постепенно увеличивает оценку снизу функции полезности, представленной альфа-векторами, до схождения. После схождения функция полезности не обязательно будет оптимальной, потому что $\mathcal{B}$ обычно не содержит всех убеждений,

достижимых из исходного убеждения. Однако если убеждения в $\mathcal{B}$ хорошо распределены по достижимому пространству убеждений, получается приемлемая аппроксимация. В любом случае результирующая функция полезности гарантированно обеспечивает нижнюю границу, которую можно использовать с другими алгоритмами, включая онлайн-методы, для дальнейшего улучшения стратегии.

Метод точечной итерации по полезности реализован в алгоритме 21.7. На рис. 21.4 показано несколько итераций демонстрационной задачи.

Алгоритм 21.7. Точечная итерация по полезности, которая находит приблизительно оптимальную стратегию для задачи POMDP с бесконечным горизонтом с дискретными пространствами состояний, действий и наблюдений, где `B` – вектор убеждений, а `k_max` – количество итераций

```
struct PointBasedValueIteration
    B     # набор точек в пространстве убеждений
    k_max # максимальное количество итераций
end

function update(𝒫::POMDP, M::PointBasedValueIteration, Γ)
    return [backup(𝒫, Γ, b) for b in M.B]
end

function solve(M::PointBasedValueIteration, 𝒫)
    Γ = fill(baws_lowerbound(𝒫), length(𝒫.𝒜))
    Γ = alphavector_iteration(𝒫, M, Γ)
    return LookaheadAlphaVectorPolicy(𝒫, Γ)
end
```

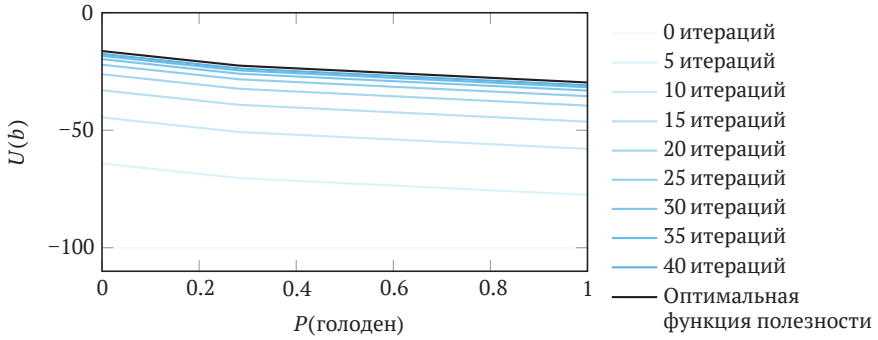


Рис. 21.4. Приближенные функции полезности, полученные методом точечной итерации по полезности для задачи о плачущем ребенке с векторами убеждений [1/4, 3/4] и [3/4, 1/4]. В отличие от методов QMDP и быстрой инфограницы, этот метод всегда дает оценку снизу истинной функции полезности

21.5. Рандомизированная точечная итерация по полезности

Рандомизированная точечная итерация по полезности (алгоритм 21.8) представляет собой разновидность метода точечной итерации по полезности из предыдущего раздела[7]. Основное отличие состоит в том, что мы не обязаны находить альфа-вектор для каждого убеждения в $\mathcal{B}$. Мы инициализируем алгоритм одним альфа-вектором в Γ, а затем обновляем Γ на каждой итерации, увеличивая или уменьшая количество альфа-векторов в Γ по мере необходимости. Эта модификация шага обновления может повысить эффективность.

Алгоритм 21.8. Рандомизированная точечная итерация по полезности, которая находит приблизительно оптимальную политику для задачи POMDP с бесконечным горизонтом с дискретными пространствами состояний, действий и наблюдений, где `B` – вектор убеждений, а `k_max` – количество итераций

```
struct RandomizedPointBasedValueIteration
    B # набор точек в пространстве убеждений
    k_max # максимальное количество итераций
end

function update(𝒫::POMDP, M::RandomizedPointBasedValueIteration, Γ)
    Γ′, B′ = [], copy(M.B)
    while !isempty(B′)
        b = rand(B′)
        α = argmax(α->α⋅b, Γ)
        α′ = backup(𝒫, Γ, b)
        if α′⋅b ≥ α⋅b
            push!(Γ′, α′)
        else
            push!(Γ′, α)
        end
        filter!(b->maximum(α⋅b for α in Γ′) <
            maximum(α⋅b for α in Γ), B′)
    end
    return Γ′
end

function solve(M::RandomizedPointBasedValueIteration, 𝒫)
    Γ = [baws_lowerbound(𝒫)]
    Γ = alphavector_iteration(𝒫, M, Γ)
    return LookaheadAlphaVectorPolicy(𝒫, Γ)
end
```

---

[7] M. T. J. Spaan, N. A. Vlassis, *Perseus: Randomized Point-Based Value Iteration for POMDPs*, Journal of Artificial Intelligence Research, vol. 24, pp. 195–220, 2005.

Каждое обновление получает в качестве входных данных набор альфа-векторов Γ и выводит набор альфа-векторов Γ′, которые улучшают функцию полезности, представленную Γ для убеждений в $\mathcal{B}$. Другими словами, процедура обновления выводит Γ′ такое, что $U^{\Gamma'}(\mathbf{b}) \geqslant U^{\Gamma}(\mathbf{b})$ для всех $\mathbf{b} \in B$. Мы начинаем с инициализации Γ′ пустым множеством и инициализации $\mathcal{B}'$ содержимым $\mathcal{B}$. Затем случайным образом удаляем точку $\mathbf{b}$ из $\mathcal{B}'$ и выполняем операцию дублирования убеждений (алгоритм 21.6) на $\mathbf{b}$, используя Γ для получения нового альфа-вектора $\boldsymbol{\alpha}$. Потом мы находим в $\Gamma \cup \{\boldsymbol{\alpha}\}$ альфа-вектор, который доминирует в точке $\mathbf{b}$, и добавляем его к Γ′. Все точки убеждений в $\mathcal{B}'$, значение которых улучшается с помощью этого альфа-вектора, затем удаляются из $\mathcal{B}'$. По мере работы алгоритма $\mathcal{B}'$ становится меньше и содержит набор точек, которые не были улучшены с помощью Γ′. Обновление завершается, когда $\mathcal{B}'$ становится пустым. На рис. 21.5 этот процесс показан на примере задачи о плачущем ребенке.

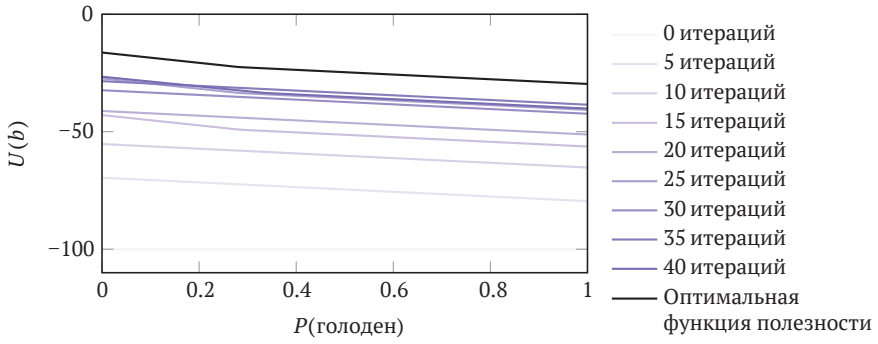


Рис. 21.5. Аппроксимированные функции полезности, полученные с использованием рандомизированной точечной итерации по полезности для задачи о плачущем ребенке с точками убеждений в [1/4, 3/4] и [3/4, 1/4]

21.6. Пилообразная оценка сверху

Пилообразная оценка сверху (sawtooth upper bound) – это альтернативный способ представления функции полезности. Вместо хранения набора альфа-векторов Γ мы храним набор пар «убеждение-полезность»:

$$V = \{(b_1, U(b_1)), \ldots, (b_m, U(b_m))\} \tag{21.11}$$

– с требованием, чтобы V содержал все стандартные базисные убеждения:

$$E = \{e_1 = [1, 0, \ldots, 0], \ldots, e_n = [0, 0, \ldots, 1]\}, \tag{21.12}$$

так что

$$\{(e_1, U(e_1)), \ldots, (e_n, U(e_n))\} \subseteq V. \tag{21.13}$$

Если эти полезности являются оценками функции полезности сверху (например, полученными по методу быстрой инфограницы), то способ, которым мы используем V для оценки $U(b)$ при произвольных убеждениях b, даст нам верхнюю границу[8].

Название «пилообразная» происходит от способа, которым мы оцениваем $U(b)$ путем интерполяции точек в V. Для каждой пары «убеждение-полезность» $(b, U(b))$ в V мы формируем одиночный заостренный «зуб». Если пространство убеждений n-мерное, каждый зубец представляет собой перевернутую n-мерную пирамиду. При наличии нескольких пар получается «пилообразная» структура. Основания пирамид образованы стандартными базисными убеждениями $(e_i, U(e_i))$. Вершина каждого зубца соответствует паре «убеждение-полезность» $(b, U(b)) \in V$. Так как в общем случае это пирамиды, каждый зуб имеет стенки, эквивалентно определяемые n-гиперплоскостями с ограниченными областями. Эти гиперплоскости также можно интерпретировать как альфа-векторы, которые действуют на ограниченную область пространства убеждений, а не на все пространство убеждений, как обычные альфа-векторы. Комбинация нескольких пирамид образует n-мерную пилообразную форму.

Рассмотрим пилообразное представление в POMDP с двумя состояниями, например в задаче о плачущем ребенке, как показано на рис. 21.6. Углы каждого зубца представляют собой значения $U(e_1)$ и $U(e_2)$ для каждого стандартного базисного убеждения e_i. Острая нижняя точка каждого зубца представляет собой значение $U(b)$, так как каждый зубец является парой точка–множество $(b, U(b))$. Линейная интерполяция от $U(e_1)$ до $U(b)$ и далее от $U(b)$ до $U(e_2)$ образует зубец. Чтобы объединить несколько зубцов и сформировать верхнюю границу, мы берем минимальное интерполированное значение при каждом убеждении, формируя характерную пилообразную форму.

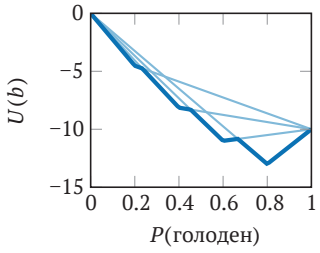


Рис. 21.6. Пилообразное представление верхней границы применимо к задаче о плачущем ребенке

Чтобы вычислить значение пилообразного представления для любого убеждения b, мы перебираем каждую пару убеждение-полезность $(b', U(b'))$ в V. Ключевая идея состоит в том, чтобы вычислить полезность $U'(b)$ для этой гиперпирамиды, сначала найдя самую дальнюю базисную точку, затем использовать ее для определения соответствующей гиперплоскости из гиперпирамиды и, наконец, вычислить полезность с использованием масштабированной

[8] Взаимосвязь между пилообразными и другими границами обсуждается в работе M. Hauskrecht, *Value-Function Approximations for Partially Observable Markov Decision Processes*, Journal of Artificial Intelligence Research, vol. 13, pp. 33–94, 2000.

версии гиперплоскости. Самое дальнее базисное убеждение e_i вычисляется с использованием расстояний L_1 от b и b':

$$i \leftarrow \arg\max_j \|b - e_j\|_1 - \|b' - e_j\|_1. \tag{21.14}$$

Это e_i однозначно выделяет конкретную гиперплоскость среди гиперплоскостей, образующих гиперпирамиду для $U(b')$. В частности, эта гиперплоскость определяется всеми углами $e_j \neq e_i$ и с использованием b' вместо e_i. На данный момент мы знаем, что это гиперплоскость для области полезности, в которой содержится b. Полезности гиперплоскости равны $U(e_j)$ при $e_j \neq e_i$ и $U(b')$ вместо $U(e_i)$. Однако мы не можем напрямую вычислить желаемую полезность $U'(b)$ с помощью скалярного произведения, потому что это не стандартный симплекс. Вместо этого мы вычисляем вес w точки b, исходя из взвешенного расстояния от углов гиперплоскости, $e_j \neq e_i$ и b'. Это позволяет нам вычислять $U'(b)$, по существу создавая симплекс, допускающий скалярное произведение $U(e_j)$ и $U(b')$:

$$U'(b) = w_i U(b') + \sum_{j \neq i} w_j U(e_j). \tag{21.15}$$

Весь этот процесс выполняется среди всех $(b', U(b'))$, в результате чего

$$U(b) = \min_{(b', U(b')) \in V} U'(b). \tag{21.16}$$

В алгоритме 21.9 показана реализация данного процесса. Мы также можем вывести стратегию, используя жадный пошаговый предпросмотр.

Алгоритм 21.9. Пилообразное представление верхней границы функций полезности и стратегий. Оно определяется с помощью словаря `V`, который сопоставляет векторы убеждений с верхними границами их полезности, полученными, например, из алгоритма быстрой инфограницы. Обязательным условием этого представления является то, что `V` хранит пары «убеждение-полезность» в стандартных базисных убеждениях, которые могут быть получены из функции `basis`. Мы можем использовать пошаговый предпросмотр, чтобы получить жадным образом пары «действие-полезность» из произвольных убеждений `b`

```
struct SawtoothPolicy
    𝒫 # задача POMDP
    V # словарь перевода убеждений в полезности
end

function basis(𝒫)
    n = length(𝒫.𝒮)
    e(i) = [j == i ? 1.0 : 0.0 for j in 1:n]
    return [e(i) for i in 1:n]
end
```

```
function utility(π::SawtoothPolicy, b)
    𝒫, V = π.𝒫, π.V
    if haskey(V, b)
        return V[b]
    end
    n = length(𝒫.𝒮)
    E = basis(𝒫)
    u = sum(V[E[i]] * b[i] for i in 1:n)
    for (b′, u′) in V
        if b′ ∉ E
            i = argmax([norm(b-e, 1) - norm(b′-e, 1) for e in E])
            w = [norm(b - e, 1) for e in E]
            w[i] = norm(b - b′, 1)
            w /= sum(w)
            w = [1 - wi for wi in w]
            α = [V[e] for e in E]
            α[i] = u′
            u = min(u, w⋅α)
        end
    end
    return u
end

(π::SawtoothPolicy)(b) = greedy(π, b).a
```

Мы можем итеративно применить жадный пошаговый предпросмотр к набору убеждений $\mathcal{B}$, чтобы сделать наши оценки верхней границы более строгими. Убеждения в наборе $\mathcal{B}$ могут быть надмножеством убеждений в V. Соответствующая реализация представлена в алгоритме 21.10. В примере 21.1 показано влияние нескольких итераций пилообразного приближения в задаче о плачущем ребенке.

Алгоритм 21.10. Метод пилообразной итерации применяет пошаговый предпросмотр в точках пространства убеждений `B`, чтобы улучшить оценки полезности в точках `V`. Убеждения в `B` являются надмножеством убеждений, содержащихся в `V`. Чтобы сохранить верхнюю границу на каждой итерации, обновления не применяются к стандартным базисным убеждениям, хранящимся в `E`. Мы запускаем `k_max` итераций

```
struct SawtoothIteration
    V     # начальное отображение убеждений в полезности
    B     # убеждения для вычисления полезностей, включая те, что в отображении V
    k_max # максимальное количество итераций
end

function solve(M::SawtoothIteration, 𝒫::POMDP)
    E = basis(𝒫)
    π = SawtoothPolicy(𝒫, M.V)
    for k in 1:M.k_max
```

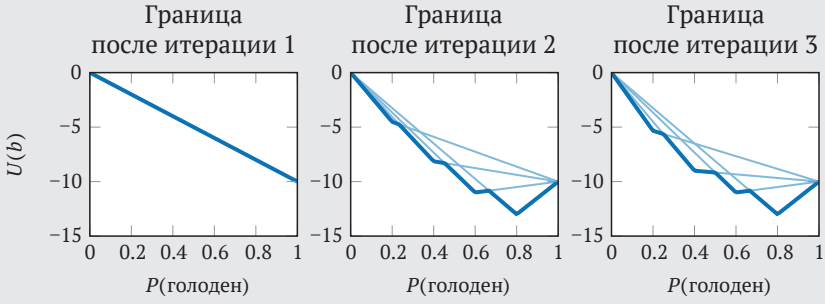
```
        V = Dict(b => (b ∈ E ? M.V[b] : greedy(π, b).u) for b in M.B)
        π = SawtoothPolicy(𝒫, V)
    end
    return π
end
```

Пример 21.1. Иллюстрация способности метода пилообразной итерации находить верхнюю границу регулярно расположенных убеждений для задачи о плачущем ребенке

Предположим, что мы хотим сохранить верхнюю границу значения для задачи о плачущем ребенке с регулярно расположенными точками убеждений с размером шага 0.2. Чтобы получить начальную верхнюю границу, мы используем метод быстрой инфограницы. Затем мы можем выполнить трехшаговую пилообразную итерацию следующим образом:

```
π = length(𝒫.𝒮)
πfib = solve(FastInformedBound(1), 𝒫)
V = Dict(e => utility(πfib, e) for e in basis(𝒫))
B = [[p, 1 - p] for p in 0.0:0.2:1.0]
π = solve(SawtoothIteration(V, B, 2), 𝒫)
```

Пилообразная верхняя граница улучшается следующим образом:



21.7. Выбор точек в наборе убеждений

Такие алгоритмы, как точечная итерация по полезности и пилообразная итерация, нуждаются в наборе убеждений $\mathcal{B}$. Необходимо выбрать такой $\mathcal{B}$, чтобы точки располагались в подходящих областях пространства убеждений – нет смысла тратить вычислительные ресурсы на убеждения, которые вряд будут достигнуты при приблизительно оптимальной стратегии. Один из способов исследовать потенциально достижимое пространство – выполнить шаги в пространстве убеждений (алгоритм 21.11). Результат шага будет случайным, потому что наблюдение генерируется в соответствии с нашей вероятностной моделью.

Алгоритм 21.11. Функция для случайной выборки следующего убеждения `b′` и вознаграждения `r` с учетом текущего убеждения `b` и действия `a` в задаче $\mathcal{P}$

```
function randstep(𝒫::POMDP, b, a)
    s = rand(SetCategorical(𝒫.𝒮, b))
    s′, r, o = 𝒫.TRO(s, a)
    b′ = update(b, 𝒫, a, o)
    return b′, r
end
```

Мы можем сформировать $\mathcal{B}$ из состояний-убеждений, достижимых из некоторого исходного убеждения при следовании случайной стратегии. Эта процедура *случайного расширения убеждений* (random belief expansion), реализованная в алгоритме 21.12, может исследовать гораздо больше пространства убеждений, чем это необходимо; пространство убеждений, охватываемое случайной стратегией, может оказаться намного больше, чем пространство, доступное оптимальной стратегии. Разумеется, вычисление пространства убеждений, достижимого с помощью оптимальной стратегии, обычно требует знания этой стратегии, а ведь именно ее мы и пытаемся найти. Один из подходов, который можно использовать, заключается в использовании последовательных приближений оптимальной стратегии для итеративного формирования B[9].

Алгоритм 21.12. Случайное расширение конечного набора убеждений `B`, используемых в точечной итерации по полезности на основе достижимых убеждений

```
function random_belief_expansion(𝒫, B)
    B′ = copy(B)
    for b in B
        a = rand(𝒫.𝒜)
        b′, r = randstep(𝒫, b, a)
        push!(B′, b′)
    end
    return unique!(B′)
end
```

Помимо условия, чтобы точки были сосредоточены на достижимом пространстве убеждений, необходимо, чтобы эти точки были равномерно распределены для лучшей аппроксимации функции полезности. Качество аппроксимации, обеспечиваемой альфа-векторами, ассоциированными с точками $\mathcal{B}$, ухудшается по мере того, как мы оцениваем точки, расположенные дальше от $\mathcal{B}$. Существует метод *исследовательского расширения убеждений* (exploratory belief expansion, алгоритм 21.13), согласно которому мы пробуем каждое дей-

[9] Это идея, лежащая в основе алгоритма, известного как *последовательные приближения достижимого пространства при оптимальных стратегиях* (SARSOP). H. Kurniawati, D. Hsu, W. S. Lee, «SARSOP: Efficient Point-Based POMDP Planning by Approximating Optimally Reachable Belief Spaces», in Robotics: Science and Systems, 2008.

ствие для каждого убеждения в $\mathcal{B}$ и добавляем в него убеждения результирующих состояний, которые наиболее далеки от убеждений, уже находящихся в наборе. Расстояние в пространстве убеждений можно измерять по-разному. Данный алгоритм использует L_1-норму[10]. На рис. 21.7 показан пример точек в пространстве убеждений, добавленных к B с использованием этого метода.

Алгоритм 21.13. Расширение конечного набора убеждений `B`, используемых в точечной итерации по полезности, путем изучения достижимых убеждений и добавления тех, которые наиболее далеки от текущих убеждений

```
function exploratory_belief_expansion(𝒫, B)
    B′ = copy(B)
    for b in B
        best = (b=copy(b), d=0.0)
        for a in 𝒫.𝒜
            b′, r = randstep(𝒫, b, a)
            d = minimum(norm(b - b′, 1) for b in B′)
            if d > best.d
                best = (b=b′, d=d)
            end
        end
        push!(B′, best.b)
    end
    return unique!(B′)
end
```

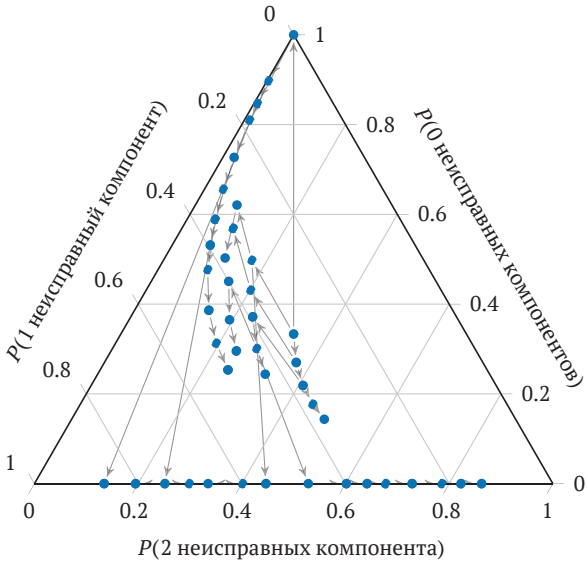


Рис. 21.7. Исследовательское расширение пространства убеждений, выполняемое в задаче о неисправной машине с тремя состояниями, начиная с исходного равномерного убеждения **b** = [1/3, 1/3, 1/3]. Новые убеждения добавляются, если расстояние до любого предыдущего убеждения составляет не менее 0.05

[10] Расстояние L_1 между b и b' равно $\sum_s |b(s) - b'(s)|$ и обозначается как $||\mathbf{b} - \mathbf{b}'||_1$. Подробнее об этом – в примечании А.4.

21.8. Пилообразный эвристический поиск

В главе 9 мы рассмотрели эвристический поиск наилучшего действия как онлайн-метод, работающий в полностью наблюдаемом контексте. В этом разделе обсуждается *пилообразный эвристический поиск* (sawtooth heuristic search, алгоритм 21.14) – офлайн-метод, создающий набор альфа-векторов, которые можно использовать для представления офлайн-стратегии. Однако здесь, как и в онлайн-методах POMDP, обсуждаемых в следующей главе, вычислительные усилия сосредоточены на убеждениях, которые достижимы из некоторого заданного исходного убеждения. Эвристика, управляющая исследованием достижимого пространства убеждений, – это разрыв между верхней и нижней границами функции полезности[11].

Алгоритм инициализируется верхней границей функции ценности, представленной набором пилообразных пар «убеждение-полезность» V, и нижней границей функции полезности, представленной набором альфа-векторов Γ. Пары «убеждение-полезность», определяющие пилообразную верхнюю границу, можно получить из алгоритма быстрой инфограницы. Нижняя граница может быть получена методом наилучшего действия и наихудшего состояния, как показано в алгоритме 21.14, или каким-либо другим методом, например точечной итерацией по полезности.

Алгоритм 21.14. Стратегия пилообразного эвристического поиска. Решатель начинает с убеждения `b` и исследует глубину `d` не более чем за `k_max` итераций. Он использует верхнюю границу, полученную с помощью метода быстрой инфограницы, вычисленной с помощью итераций `k_fib`. Нижняя граница получена по методу наилучшего действия и наихудшего состояния. Порог разрыва между границами равен δ

```
struct SawtoothHeuristicSearch
    b # начальное убеждение
    δ # пороговое значение разрыва
    d # глубина
    k_max # максимальное количество итераций
    k_fib # количество итераций для метода быстрой инфограницы
end

function explore!(M::SawtoothHeuristicSearch, 𝒫, πhi, πlo, b, d=0)
    𝒮, 𝒜, 𝒪, γ = 𝒫.𝒮, 𝒫.𝒜, 𝒫.𝒪, 𝒫.γ
    ε(b′) = utility(πhi, b′) - utility(πlo, b′)
```

---

[11] В алгоритме *эвристического поиска путем итерации по полезности* (HSVI) отражена идея использования эвристики действия на основе границы пилообразной формы и эвристики наблюдения на основе промежутка. T. Smith, R. G. Simmons, *Heuristic Search Value Iteration for POMDPs*, in Conference on Uncertainty in Artificial Intelligence (UAI), 2004. Алгоритм SARSOP основан на этой работе. H. Kurniawati, D. Hsu, W. S. Lee, *SARSOP: Efficient Point-Based POMDP Planning by Approximating Optimally Reachable Belief Spaces*, in Robotics: Science and Systems, 2008.

```
    if d ≥ M.d || ε(b) ≤ M.δ / γ^d
        return
    end
    a = πhi(b)
    o = argmax(o -> ε(update(b, 𝒫, a, o)), 𝒪)
    b′ = update(b, 𝒫, a, o)
    explore!(M, 𝒫, πhi, πlo, b′, d+1)
    if b′ ∉ basis(𝒫)
        πhi.V[b′] = greedy(πhi, b′).u
    end
    push!(πlo.Γ, backup(𝒫, πlo.Γ, b′))
end

function solve(M::SawtoothHeuristicSearch, 𝒫::POMDP)
    πfib = solve(FastInformedBound(M.k_fib), 𝒫)
    Vhi = Dict(e => utility(πfib, e) for e in basis(𝒫))
    πhi = SawtoothPolicy(𝒫, Vhi)
    πlo = LookaheadAlphaVectorPolicy(𝒫, [baws_lowerbound(𝒫)])
    for i in 1:M.k_max
        explore!(M, 𝒫, πhi, πlo, M.b)
        if utility(πhi, M.b) - utility(πlo, M.b) < M.δ
            break
        end
    end
    return πlo
end
```

На каждой итерации мы исследуем убеждения, которые достижимы от нашего первоначального убеждения до максимальной глубины. По мере исследования мы обновляем набор пар «убеждение-действие», формирующих пилообразную верхнюю границу, и набор альфа-векторов, формирующих нижнюю границу. Мы прекращаем исследование после определенного количества итераций или когда разрыв в начальном состоянии не станет ниже порога $\delta > 0$.

Встречая убеждение b на пути от начального узла во время исследования, мы проверяем, находится ли разрыв в точке b ниже порогового значения δ/γ^d, где d – текущая глубина. Если разрыв ниже этого порога, то исследование текущей ветки можно прекратить. Нам нужно, чтобы порог увеличивался по мере увеличения d, потому что разрыв в точке b после обновления не более чем в γ раз превышает средневзвешенное значение разрыва в точках убеждений, которые сразу достижимы.

Если разрыв в точке b выше порога и мы не достигли максимальной глубины, то можно исследовать следующее убеждение b'. Сначала мы определяем действие, рекомендуемое нашей пилообразной стратегией поиска. Затем мы выбираем наблюдение o, которое максимизирует разрыв в результирующем убеждении[12]. Таким образом, мы рекурсивно исследуем дерево. Изучив по-

[12] Некоторые варианты просто выбирают следующие наблюдения. Другие выбирают наблюдение, которое максимизирует разрыв, взвешенный по его вероятности.

томков b', мы добавляем к V пару (b', u), где u – полезность состояния b' по методу пошагового предпросмотра. Далее мы добавляем к Γ альфа-вектор, полученный в результате операции дублирования в точке b'. На рис. 21.8 показан процесс итеративного повышения строгости границ.

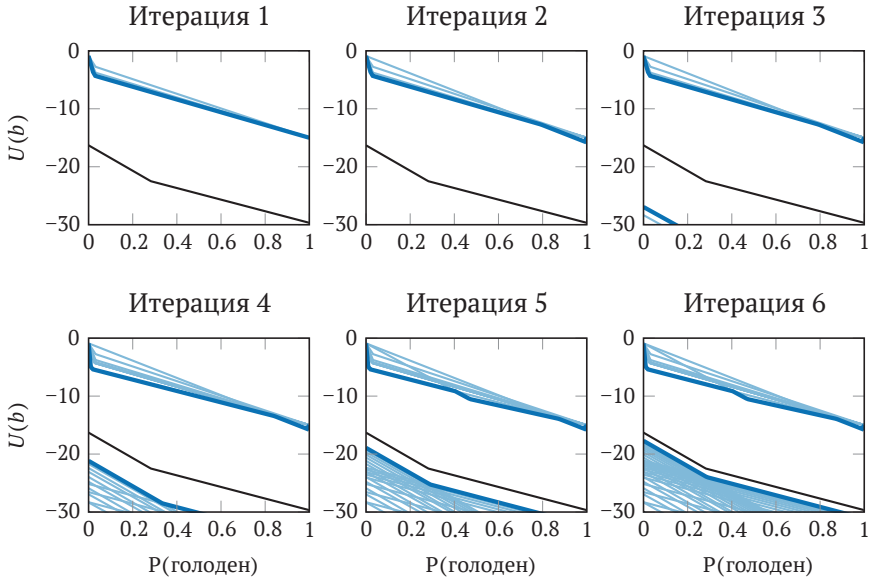


Рис. 21.8. Эволюция верхней границы, представленной пилообразными парами, и нижней границы, представленной альфа-векторами для задачи о плачущем ребенке. Оптимальная функция полезности показана черным цветом

21.9. Триангулированные функции полезности

Как было сказано в разделе 20.1, POMDP можно преобразовать в BS-MDP. Пространство состояний в задаче BS-MDP является непрерывным и соответствует пространству возможных убеждений в исходном POMDP. Мы можем аппроксимировать функцию полезности способом, подобным тому, который был описан в главе 8, а затем применить к результату аппроксимации алгоритм динамического программирования, такой как итерация по полезности. В этом разделе обсуждается особый вид аппроксимации локальной функции полезности, который основан на *триангуляции Фрейденталя* (Freudenthal triangulation)[13] по дискретному набору точек убеждения $\mathcal{B}$. Эта триангуляция позволяет нам интерполировать функцию полезности в произвольных точках пространства

[13] H. Freudenthal, *Simplizialzerlegungen von Beschränkter Flachheit*, Annals of Mathematics, vol. 43, pp. 580–582, 1942. Этот метод триангуляции был применен к POMDP в исследовании W. S. Lovejoy, *Computationally Feasible Bounds for Partially Observed Markov Decision Processes*, Operations Research, vol. 39, no. 1, pp. 162–175, 1991.

убеждений. Как и в случае пилообразного представления, для представления функции полезности мы используем набор пар «убеждение-полезность» $V = \{(b, U(b)) | b \in \mathcal{B}\}$. Этот подход можно использовать для оценки сверху границы функции полезности.

Интерполяция Фрейденталя в пространстве убеждений подразумевает равномерное распределение точек убеждений в $\mathcal{B}$ по всему пространству, как показано на рис. 21.9. Количество убеждений в $\mathcal{B}$ зависит от размерности n и детализации m триангуляции Фрейденталя[14]:

$$|\mathcal{B}| = \frac{(m+n-1)!}{m!(n-1)!}. \tag{21.17}$$

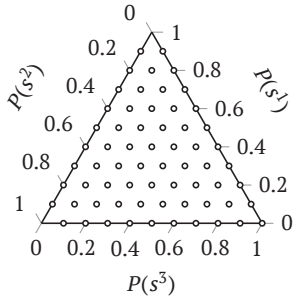


Рис. 21.9. Дискретизация с использованием триангуляции Фрейденталя в трехмерном пространстве убеждений ($n = 3$) с детализацией $m = 10$

Мы можем оценить $U(\mathbf{b})$ в произвольной точке b, интерполируя значения в дискретных точках $\mathcal{B}$. Подобно симплексной интерполяции, представленной в разделе 8.5, мы находим множество точек убеждений в $\mathcal{B}$, образующих симплекс, который охватывает b, и взвешиваем их вместе. В n-мерных пространствах убеждений существует до $n + 1$ вершин, полезности которых необходимо взвесить вместе. Если $b^{(1)}, \ldots, b^{(n+1)}$ – окружающие точки и $\lambda_1, \ldots, \lambda_{n+1}$ – их веса, то оценка полезности в точке b равна

$$U(b) = \sum_i \lambda_i U(b^{(i)}). \tag{21.18}$$

Алгоритм 21.15 извлекает эту функцию полезности и стратегию из пар в V.

Алгоритм 21.16 применяет разновидность приближенной итерации по полезности (представленной в алгоритме 8.1) к триангулированному представлению стратегии. Мы просто итеративно применяем операцию дублирования убеждений в $\mathcal{B}$, используя метод пошагового предпросмотра с интерполяцией функции полезности. Если U инициализируется начальной верхней границей, итерация по полезности приведет к верхней границе даже после конечного числа итераций. Это свойство наблюдается, потому что функции полезности являются выпуклыми, а линейная интерполяция между вершинами функции

[14] Реализация выработки этих представлений дана в `FreudenthalTriangulations.jl`.

полезности должна лежать на базисной выпуклой функции или над ней[15]. На рис. 21.10 показан пример стратегии и функции полезности.

Алгоритм 21.15. Представление стратегии с использованием триангуляции Фрейденталя с детализацией `m`. Как и в случае с пилообразным методом, мы используем словарь, который сопоставляет векторы убеждений с полезностями. Данный код инициализирует полезности нулевыми значениями, но если необходимо обозначить начальную верхнюю границу, следует соответствующим образом инициализировать эти полезности. Мы находим функцию для оценки полезности заданного убеждения с помощью интерполяции. Далее мы можем извлечь стратегию, используя жадный предпросмотр. Структуре триангуляции Фрейденталя при построении передаются размерность и детализация. В пакете `FreudenthalTriangulations.jl` есть функция `trust_vertices`, которая возвращает `B` при заданной триангуляции. Кроме того, в нем предусмотрена функция `wake_simplex`, которая возвращает набор ближайших точек и весов для текущего убеждения

```
struct TriangulatedPolicy
    𝒫  # задача POMDP
    V  # словарь отображения убеждений в полезности
    B  # убеждения
    T  # триангуляция Фрейденталя
end

function TriangulatedPolicy(𝒫::POMDP, m)
    T = FreudenthalTriangulation(length(𝒫.𝒮), m)
    B = belief_vertices(T)
    V = Dict(b => 0.0 for b in B)
    return TriangulatedPolicy(𝒫, V, B, T)
end

function utility(π::TriangulatedPolicy, b)
    B, λ = belief_simplex(π.T, b)
    return sum(λi*π.V[b] for (λi, b) in zip(λ, B))
end

(π::TriangulatedPolicy)(b) = greedy(π, b).a
```

Алгоритм 21.16. Приблизительная итерация по полезности с `k_max` итераций и использованием триангулированной стратегии с детализацией `m`. На каждой итерации мы обновляем полезности, связанные с убеждениями в `B`, используя жадный предпросмотр с триангулированными полезностями

```
struct TriangulatedIteration
    m     # гранулярность
```

---

[15] См. лемму 4 в W. S. Lovejoy, *Computationally Feasible Bounds for Partially Observed Markov Decision Processes*, Operations Research, vol. 39, no. 1, pp. 162–175. 1991.

```
    k_max # максимальное количество итераций
end

function solve(M::TriangulatedIteration, 𝒫)
    π = TriangulatedPolicy(𝒫, M.m)
    U(b) = utility(π, b)
    for k in 1:M.k_max
        U′ = [greedy(𝒫, U, b).u for b in π.B]
        for (b, u′) in zip(π.B, U′)
            π.V[b] = u′
        end
    end
    return π
end
```

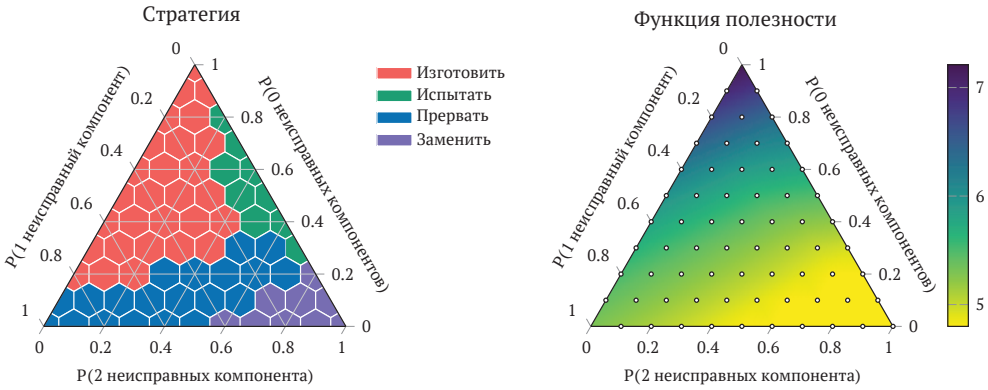


Рис. 21.10. Стратегия и функция полезности для задачи о ремонте с детализацией $m = 10$ после 11 итераций. Визуализация функции полезности показывает дискретные точки убеждений в виде белых кружков. Эта стратегия приблизительно соответствует стратегии, показанной в приложении F.8

21.10. Заключение

- Алгоритм QMDP предполагает идеальную наблюдаемость после первого шага, что дает оценку верхней границы истинной функции полезности.
- Метод быстрой инфограницы обеспечивает более точную верхнюю границу функции полезности, чем QMDP, потому что учитывает модель наблюдения.
- Метод точечной итерации по полезности дает нижнюю границу функции полезности с использованием альфа-векторов при конечном наборе допущений.
- Рандомизированная точечная итерация по полезности выполняет обновления в случайно выбранных точках в наборе убеждений до тех пор, пока полезности во всех точках в наборе не улучшатся.

- Метод пилообразной верхней границы позволяет итеративно улучшать оценку быстрой инфограницы, используя эффективное представление набора точек.
- Тщательный выбор точек убеждений для использования в точечной итерации по полезности помогает улучшить качество результирующих стратегий.
- Пилообразный эвристический поиск пытается сузить верхнюю и нижнюю границы функции полезности, представленные пилообразной аппроксимацией и альфа-векторами соответственно.
- Один из подходов к приблизительному решению задачи POMDP заключается в дискретизации пространства убеждений и последующем применении динамического программирования для нахождения верхней границы функции полезности и стратегии.

21.11. Упражнения

Упражнение 21.1. Предположим, что нам дан прямолинейный вариант задачи о гексамире (приложение F.1), состоящий из четырех клеток, соответствующих состояниям $s_{1:4}$. Доступны два действия: двигаться влево (ℓ) и двигаться вправо (r). Последствия этих действий детерминированы. Движение влево в s_1 или вправо в s_4 дает вознаграждение 100 и заканчивает игру. Используя коэффициент дисконтирования 0.9, вычислите альфа-векторы с помощью метода QMDP. Затем, используя альфа-векторы, найдите приблизительно оптимальное действие, исходя из начального убеждения $\mathbf{b} = [0.3, 0.1, 0.5, 0.1]$.

Решение. Обозначим альфа-вектор, связанный с движением влево, за α_ℓ, а альфа-вектор, связанный с движением вправо, за α_r. Инициализируем эти альфа-векторы нулями:

$$\boldsymbol{\alpha}_\ell^{(1)} = [R(s_1, \ell), R(s_2, \ell), R(s_3, \ell), R(s_4, \ell)] = [0, 0, 0, 0];$$

$$\boldsymbol{\alpha}_r^{(1)} = [R(s_1, r), R(s_2, r), R(s_3, r), R(s_4, r)] = [0, 0, 0, 0].$$

В первой итерации, поскольку все элементы альфа-векторов равны нулю, только член вознаграждения вносит вклад в обновление QMDP (уравнение 21.1):

$$\boldsymbol{\alpha}_\ell^{(2)} = [100, 0, 0, 0];$$

$$\boldsymbol{\alpha}_r^{(2)} = [0, 0, 0, 100].$$

На следующей итерации мы применяем обновление, которое дает новые значения s_2 для левого альфа-вектора и s_3 для правого альфа-вектора. Обновления для левого альфа-вектора следующие (обновления правого альфа-вектора симметричны):

$\boldsymbol{\alpha}_{\ell}^{(3)}(s_1) = 100$ (конечное состояние);

$\boldsymbol{\alpha}_{\ell}^{(3)}(s_2) = 0 + 0.9 \times \max(\boldsymbol{\alpha}_{\ell}^{(2)}(s_1), \boldsymbol{\alpha}_{r}^{(2)}(s_1)) = 90$;

$\boldsymbol{\alpha}_{\ell}^{(3)}(s_3) = 0 + 0.9 \times \max(\boldsymbol{\alpha}_{\ell}^{(2)}(s_2), \boldsymbol{\alpha}_{r}^{(2)}(s_2)) = 0$;

$\boldsymbol{\alpha}_{\ell}^{(3)}(s_4) = 0 + 0.9 \times \max(\boldsymbol{\alpha}_{\ell}^{(2)}(s_3), \boldsymbol{\alpha}_{r}^{(2)}(s_3)) = 0$.

Это дает нам следующие векторы:

$\boldsymbol{\alpha}_{\ell}^{(3)} = [100, 90, 0, 0]$;

$\boldsymbol{\alpha}_{r}^{(3)} = [0, 0, 90, 100]$.

В третьей итерации обновления для левого альфа-вектора выглядят так:

$\boldsymbol{\alpha}_{\ell}^{(4)}(s_1) = 100$ (конечное состояние);

$\boldsymbol{\alpha}_{\ell}^{(4)}(s_2) = 0 + 0.9 \times \max(\boldsymbol{\alpha}_{\ell}^{(3)}(s_1), \boldsymbol{\alpha}_{r}^{(3)}(s_1)) = 90$;

$\boldsymbol{\alpha}_{\ell}^{(4)}(s_3) = 0 + 0.9 \times \max(\boldsymbol{\alpha}_{\ell}^{(3)}(s_2), \boldsymbol{\alpha}_{r}^{(3)}(s_2)) = 81$;

$\boldsymbol{\alpha}_{\ell}^{(4)}(s_4) = 0 + 0.9 \times \max(\boldsymbol{\alpha}_{\ell}^{(3)}(s_3), \boldsymbol{\alpha}_{r}^{(3)}(s_3)) = 81$.

Тогда наши альфа-векторы равны

$\boldsymbol{\alpha}_{\ell}^{(4)} = [100, 90, 81, 81]$;

$\boldsymbol{\alpha}_{r}^{(4)} = [81, 81, 90, 100]$.

На данный момент наши оценки альфа-векторов сошлись. Теперь определим оптимальное действие, максимизируя полезность, связанную с нашим убеждением, по всем действиям:

$\boldsymbol{\alpha}_{\ell}^{\top}\mathbf{b} = 100 \times 0.3 + 90 \times 0.1 + 81 \times 0.5 + 81 \times 0.1 = 87.6$;

$\boldsymbol{\alpha}_{r}^{\top}\mathbf{b} = 81 \times 0.3 + 81 \times 0.1 + 90 \times 0.5 + 100 \times 0.1 = 87.4$.

Таким образом, мы обнаруживаем, что движение влево является оптимальным действием для текущих убеждений, несмотря на более высокую вероятность оказаться в правой половине сетки. Это связано с относительно высокой вероятностью, которую мы приписываем нахождению в состоянии s_1, где мы получили бы большее немедленное вознаграждение, двигаясь влево.

Упражнение 21.2. Вернемся к условиям задачи из упражнения 21.1. Вычислите альфа-векторы для каждого действия, используя слепой метод нижней границы. Затем, используя альфа-векторы, вычислите полезность в точке $\mathbf{b} = [0.3, 0.1, 0.5, 0.1]$.

Решение. Оценка границы снизу, найденная слепым методом и показанная в уравнении (21.6), аналогична обновлению QMDP, но в ней отсутствует мак-

симизация. Мы инициализируем компоненты альфа-векторов нулями и движемся к сходимости следующим образом:

$\boldsymbol{\alpha}_\ell^{(2)} = [100, 0, 0, 0]$;
$\boldsymbol{\alpha}_r^{(2)} = [0, 0, 0, 100]$.

$\boldsymbol{\alpha}_\ell^{(3)} = [100, 90, 0, 0]$;
$\boldsymbol{\alpha}_r^{(3)} = [0, 0, 90, 100]$.

$\boldsymbol{\alpha}_\ell^{(4)} = [100, 90, 81, 81]$;
$\boldsymbol{\alpha}_r^{(4)} = [81, 81, 90, 100]$.

$\boldsymbol{\alpha}_\ell^{(5)} = [100, 90, 81, 72.9]$;
$\boldsymbol{\alpha}_r^{(5)} = [72.9, 81, 90, 100]$.

На данный момент наши оценки альфа-векторов сошлись. Теперь определим границу, максимизируя полезность, связанную с нашим убеждением, по всем действиям:

$\boldsymbol{\alpha}_\ell^\top \mathbf{b} = 100 \times 0.3 + 90 \times 0.1 + 81 \times 0.5 + 72.9 \times 0.1 = 86.79$;
$\boldsymbol{\alpha}_r^\top \mathbf{b} = 72.9 \times 0.3 + 81 \times 0.1 + 90 \times 0.5 + 100 \times 0.1 = 84.97$.

Таким образом, нижняя граница в точке $\mathbf{b}$ равна 86.79.

Упражнение 21.3. Какова вычислительная сложность операции дублирования в одной точке пространства убеждений при точечной итерации полезности, предполагая, что $|\Gamma| > |\mathcal{S}|$?

Решение. В процессе дублирования мы вычисляем $\boldsymbol{\alpha}_{a,o}$ для каждого действия a и наблюдения o. Вычисление $\boldsymbol{\alpha}_{a,o}$ в уравнении (21.8) требует нахождения такого альфа-вектора $\boldsymbol{\alpha}$ в Γ, который максимизирует $\boldsymbol{\alpha}_a^\top \text{Update}(\mathbf{b}, a, o)$. Вычислительная нагрузка обновления убеждения, как следует из уравнения (19.7), равна $O(|\mathcal{S}|^2)$, потому что обновление повторяется по всем начальным и последующим состояниям. Следовательно, для вычисления $\boldsymbol{\alpha}_{a,o}$ требуется $O(|\Gamma||\mathcal{S}| + |\mathcal{S}|^2) = O(|\Gamma||\mathcal{S}|)$ операций для конкретных a и o, что в сумме дает $O(|\Gamma||\mathcal{S}||\mathcal{A}||\mathcal{O}|)$ операций. Затем мы вычисляем $\boldsymbol{\alpha}_a$ в уравнении (21.9) для каждого действия a и используем эти значения для нахождения $\boldsymbol{\alpha}_{a,o}$, что требует в сумме $O(|\mathcal{S}|^2|\mathcal{A}||\mathcal{O}|)$ операций. Нахождение альфа-вектора $\boldsymbol{\alpha}_a$, максимизирующего $\boldsymbol{\alpha}_a^\top \mathbf{b}$, требует еще $O(|\mathcal{S}||\mathcal{A}|)$ операций после нахождения всех $\boldsymbol{\alpha}_a$. В совокупности получается $O(|\Gamma||\mathcal{S}||\mathcal{A}||\mathcal{O}|)$ операций для дублирования убеждения $\mathbf{b}$.

Упражнение 21.4. Рассмотрим набор пар «убеждение-полезность», заданный следующим образом:

$V = \{([1, 0], 0), ([0, 1], -10), ([0.8, 0.2], -4), ([0.4, 0.6], -6)\}$.

Используя веса $w_i = 0.5$ для всех i, определите полезность для убеждения $\mathbf{b} = [0.5, 0.5]$ с помощью метода пилообразной верхней границы.

Решение. Выполним интерполяцию с помощью пар «убеждение-полезность». Для каждого небазисного убеждения мы начинаем с поиска самого дальнего базисного убеждения $\mathbf{e}_i$. Начиная с $\mathbf{b}_3$, выполним следующие вычисления:

$$i_3 = \arg\max_j \|\mathbf{b} - \mathbf{e}_j\|_1 - \|\mathbf{b}_3 - \mathbf{e}_j\|_1;$$

$$\begin{aligned}\|\mathbf{b} - \mathbf{e}_1\|_1 - \|\mathbf{b}_3 - \mathbf{e}_1\|_1 &= \left\|\begin{bmatrix}0.5\\0.5\end{bmatrix} - \begin{bmatrix}1\\0\end{bmatrix}\right\|_1 - \left\|\begin{bmatrix}0.8\\0.2\end{bmatrix} - \begin{bmatrix}1\\0\end{bmatrix}\right\|_1 \\ &= \left\|\begin{bmatrix}-0.5\\0.5\end{bmatrix}\right\|_1 - \left\|\begin{bmatrix}-0.2\\0.2\end{bmatrix}\right\|_1 \\ &= 0.6;\end{aligned}$$

$$\begin{aligned}\|\mathbf{b} - \mathbf{e}_2\|_1 - \|\mathbf{b}_3 - \mathbf{e}_2\|_1 &= \left\|\begin{bmatrix}0.5\\0.5\end{bmatrix} - \begin{bmatrix}0\\1\end{bmatrix}\right\|_1 - \left\|\begin{bmatrix}0.8\\0.2\end{bmatrix} - \begin{bmatrix}0\\1\end{bmatrix}\right\|_1 \\ &= \left\|\begin{bmatrix}0.5\\-0.5\end{bmatrix}\right\|_1 - \left\|\begin{bmatrix}0.8\\-0.8\end{bmatrix}\right\|_1 \\ &= -0.6;\end{aligned}$$

$$i_3 = 1.$$

Следовательно, $\mathbf{e}_1$ – самое дальнее базисное убеждение от $\mathbf{b}_3$. Для $\mathbf{b}_4$ выполним следующие вычисления:

$$i_4 = \arg\max_j \|\mathbf{b} - \mathbf{e}_j\|_1 - \|\mathbf{b}_4 - \mathbf{e}_j\|_1;$$

$$\begin{aligned}\|\mathbf{b} - \mathbf{e}_1\|_1 - \|\mathbf{b}_3 - \mathbf{e}_1\|_1 &= \left\|\begin{bmatrix}0.5\\0.5\end{bmatrix} - \begin{bmatrix}1\\0\end{bmatrix}\right\|_1 - \left\|\begin{bmatrix}0.4\\0.6\end{bmatrix} - \begin{bmatrix}1\\0\end{bmatrix}\right\|_1 \\ &= \left\|\begin{bmatrix}-0.5\\0.5\end{bmatrix}\right\|_1 - \left\|\begin{bmatrix}-0.6\\0.6\end{bmatrix}\right\|_1 \\ &= -0.2;\end{aligned}$$

$$\begin{aligned}\|\mathbf{b} - \mathbf{e}_2\|_1 - \|\mathbf{b}_3 - \mathbf{e}_2\|_1 &= \left\|\begin{bmatrix}0.5\\0.5\end{bmatrix} - \begin{bmatrix}0\\1\end{bmatrix}\right\|_1 - \left\|\begin{bmatrix}0.4\\0.6\end{bmatrix} - \begin{bmatrix}0\\1\end{bmatrix}\right\|_1 \\ &= \left\|\begin{bmatrix}0.5\\-0.5\end{bmatrix}\right\|_1 - \left\|\begin{bmatrix}0.4\\-0.4\end{bmatrix}\right\|_1 \\ &= -0.2;\end{aligned}$$

$$i_4 = 2.$$

Следовательно, $\mathbf{e}_2$ является самым дальним базисным убеждением от $\mathbf{b}_4$.

Теперь вычислим $U(\mathbf{b})$, используя веса вместе с соответствующими парами $(\mathbf{e}_2, \mathbf{b}_3)$ и $(\mathbf{e}_1, \mathbf{b}_4)$:

$$U_3(\mathbf{b}) = 0.5 \times -4 + 0.5 \times (-10) = -7;$$

$$U_4(\mathbf{b}) = 0.5 \times -6 + 0.5 \times 0 = -3.$$

Наконец, мы вычисляем $U(b)$, выбрав минимальное значение из $U_3(\mathbf{b})$ и $U_4(\mathbf{b})$. Таким образом, $U(\mathbf{b}) = -7$.

Упражнение 21.5. Предположим, что у нас есть допустимая оценка границы снизу, представленная в виде набора альфа-векторов Г. Возможно ли, чтобы операция дублирования в убеждении b давала альфа-вектор $\boldsymbol{\alpha}'$, такой что $\boldsymbol{\alpha}'^{\top}\mathbf{b}$ меньше, чем функция полезности, представленная Г? Другими словами, может ли дублирование в точке убеждения b дать альфа-вектор, который присваивает b более низкую полезность, чем функция полезности, представленная Г?

Решение. Это возможно. Предположим, что у нас есть только одно действие, наблюдения совершенны, дисконтирования нет, а пространство состояний определено как $\{s^0, s^1\}$. Вознаграждение равно $R(s^i) = i$ для всех i, и состояния детерминированно переходят в s^0. Начнем с допустимой нижней границы Г = $\{[-1, +1]\}$, как показано красной линией на рис. 21.11. В качестве точки, в которой будем выполнять дублирование, выбираем $\mathbf{b} = [0.5, 0.5]$. Используя уравнение (21.9), получаем:

$$\alpha(s^0) = R(s^0) + U^{\Gamma}(s^0) = 0 + (-1) = -1;$$

$$\alpha(s^1) = R(s^1) + U^{\Gamma}(s^0) = 1 + (-1) = 0.$$

Следовательно, альфа-вектор, который мы получаем после операции дублирования, равен [–1,0], как показано синим цветом на рис. 21.11. Полезность в точке $\mathbf{b}$ с этим альфа-вектором составляет –0.5. Однако $U^{\Gamma}(\mathbf{b}) = 0$, следовательно, дублирование убеждения может дать альфа-вектор, который представляет более низкую полезность этого убеждения. Данный факт объясняет использование оператора `if` в рандомизированной точечной итерации по полезности (алгоритм 21.8). Благодаря оператору `if` алгоритм будет использовать либо альфа-вектор после дублирования, либо доминирующий альфа-вектор в Г при убеждении b, в зависимости от того, что дает наибольшую оценку полезности.

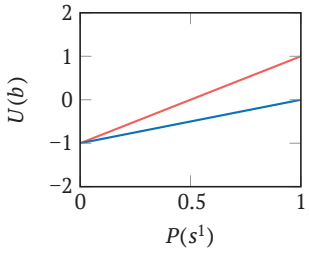


Рис. 21.11. Пример того, как операция дублирования убеждения может дать альфа-вектор, который сам по себе снижает полезность этого убеждения по сравнению с исходной функцией полезности. Убеждение b, в точке которого мы делаем обновление, соответствует $P(s_1) = 0.5$. Исходная функция полезности, представленная через Г, показана красным цветом. Альфа-вектор, полученный в результате дублирования в точке $\mathbf{b}$, показан синим цветом

22 Онлайн-планирование с использованием убеждений-состояний

Онлайн-методы определяют оптимальную стратегию путем планирования из текущего убеждения-состояния. Пространство убеждений, доступное из текущего состояния, обычно невелико по сравнению с полным пространством убеждений. Как мы уже говорили ранее, многие онлайн-методы используют варианты поиска по дереву до некоторого горизонта[1]. Существуют различные приемы, помогающие избежать экспоненциального роста вычислительной нагрузки с ростом глубины дерева. Хотя онлайн-методы требуют больше вычислений на шаг решения, чем офлайн-методы, иногда они лучше подходят для решения многомерных задач.

22.1. Предпросмотр с развертываниями

В алгоритме 9.1 была представлена реализация предпросмотра с развертыванием в качестве онлайн-метода для решения полностью наблюдаемых задач. Но этот алгоритм можно применить и в случае частично наблюдаемых задач. Он использует функцию для случайной выборки следующего состояния, аналогично убеждению-состоянию в случае частичной наблюдаемости. Эта функция уже была задействована в алгоритме 21.11. Благодаря использованию генеративной, а не явной модели для переходов, вознаграждений и наблюдений мы можем справиться с задачами, в которых присутствуют многомерные пространства состояний и наблюдений.

22.2. Прямой поиск

Мы можем в неизменном виде применить стратегию прямого поиска из алгоритма 9.2 к частично наблюдаемым задачам. Разница между MDP и POMDP в данном случае купируется пошаговым предпросмотром, который разветвляется на действия и наблюдения, как показано на рис. 22.1. Полезность действия a из убеждения b может быть определена рекурсивно до глубины d:

---

[1] Обзор опубликован в S. Ross, J. Pineau, S. Paquet, B. Chaib-draa, *Online Planning Algorithms for POMDPs*, Journal of Artificial Intelligence Research, vol. 32, pp. 663–704, 2008.

$$Q_d(b,a) = \begin{cases} R(b,a) + \gamma \sum_o P(o|b,a) U_{d-1}(\text{Update}((b,a,o)), & \text{если } d > 0 \\ U(b) & \text{в ином случае} \end{cases}, \quad (22.1)$$

где $U_d(b) = \max_a Q_d(b,a)$. Когда $d = 0$, это означает, что мы достигли максимальной глубины и возвращаем полезность, используя аппроксимированную функцию полезности $U(b)$, которую можно получить одним из методов, обсуждавшихся в предыдущей главе, эвристическим выбором или оценкой из одного или нескольких развертываний. Когда $d > 0$, мы продолжаем поиск глубже, рекурсивно переходя на следующий уровень. Пример 22.1 демонстрирует применение QMDP с предпросмотром для решения задачи о ремонте машины. Пример 22.2 демонстрирует применение прямого поиска в задаче о плачущем ребенке.

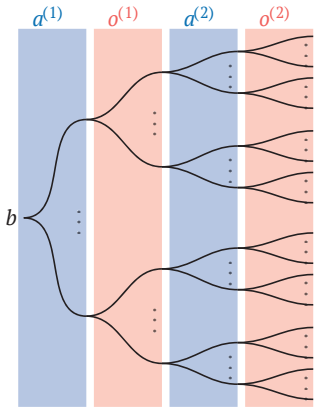


Рис. 22.1. Прямой поиск просматривает граф действие–наблюдение–убеждение до произвольной конечной глубины, чтобы выбрать действие, которое дает наивысшее ожидаемое вознаграждение. Здесь показан поиск до глубины 2

Пример 22.1. Применение прямого поиска к задаче о ремонте машины (приложение F.8)

Рассмотрим применение прямого поиска к задаче о ремонте машины. Сначала получим аппроксимированную функцию полезности с помощью QMDP (алгоритм 21.2). Затем создадим объект `ForwardSearch`, который изначально был определен в алгоритме 9.2. Вызов `lookahead` внутри этой функции ранее был определен для POMDP в алгоритме 20.5. Следующий код применяет метод прямого поиска к задаче $\mathcal{P}$ начиная с убеждения-состояния [0.5, 0.2, 0.3] на глубину 5, используя оценку полезности, полученную из QMDP на листовых узлах:

```
k_max = 10 # максимальное количество итераций QMDP
πQMDP = solve(QMDP(k_max), 𝒫)
d = 5 # depth
U(b) = utility(πQMDP, b)
π = ForwardSearch(𝒫, d, U)
π([0.5,0.2,0.3])
```

Пример 22.2. Применение прямого поиска к задаче о плачущем ребенке (приложение F.7)

Рассмотрим применение прямого поиска к задаче о плачущем ребенке с аппроксимированной функцией полезности, заданной альфа-векторами [−3.7, −15] и [−2, −21]. Поиск вперед на глубину 2 от начального убеждения b = [0.5, 0.5] происходит следующим образом:

$$\begin{aligned} Q_2(b, a_{\text{кормить}}) &= R(b, a_{\text{кормить}}) + \gamma(P(\text{плачет}|b, \text{кормить})U_1([1.0, 0.0]) \\ &\quad + P(\text{спокоен}|b, \text{кормить})U_1([1.0, 0.0])) \\ &= -10 + 0.9(0.1 \times -3.2157 + 0.9 \times -3.2157) \\ &= -12.894 \end{aligned}$$

$$\begin{aligned} Q_2(b, a_{\text{игнорировать}}) &= R(b, a_{\text{игнорировать}}) + \gamma(P(\text{плачет}|b, \text{игнорировать})U_1([0.093, 0.907]) \\ &\quad + P(\text{спокоен}|b, \text{игнорировать})U_1([0.786, 0.214])) \\ &= -5 + 0.9(0.485 \times -15.872 + 0.515 \times -7.779) \\ &= -15.534 \end{aligned}$$

$$\begin{aligned} Q_2(b, a_{\text{петь}}) &= R(b, a_{\text{петь}}) + \gamma(P(\text{плачет}|b, \text{петь})U_1([0.0, 1.0]) \\ &\quad + P(\text{спокоен}|b, \text{петь})U_1([0.891, 0.109])) \\ &= -5.5 + 0.9(0.495 \times -16.8 + 0.505 \times -5.543) \\ &= -15.503 \end{aligned}$$

Напомним, что кормление ребенка всегда приводит к тому, что он насыщается (b = [1, 0]), а пение колыбельной гарантирует, что он плачет, только если голоден (b = [0, 1]). Каждая полезность U_1 оценивается путем рекурсии на один уровень глубже в уравнении (22.1) с использованием функции $U_d(b) = \max_a Q_d(b, a)$. На максимальной глубине мы используем аппроксимированную функцию полезности, заданную альфа-векторами, $Q_0(b, \text{а}) = \max(b^\top[-3.7, -15], b^\top[-2, -21])$.

Стратегия предсказывает, что кормление ребенка приведет к наивысшей ожидаемой полезности, поэтому она рекомендует это действие.

Вычислительная нагрузка, связанная с рекурсией в уравнении (22.1), экспоненциально растет с глубиной: $O(|\mathcal{A}|^d|\mathcal{O}|^d)$. По этой причине прямой поиск обычно ограничивается относительно небольшой глубиной. Чтобы достичь большей глубины, обычно ограничивают ветвление по действиям или наблюдениям. Например, если у нас есть некоторые знания предметной области, мы можем априорно ограничить выбор действий либо в корне, либо ниже по дереву. Кроме того, мы можем ограничить свое внимание небольшим набором вероятных наблюдений или даже единственным наиболее вероятным наблюдением[2]. Ветвления можно полностью избежать, применив методы оптимизации без обратной связи, описанные в разделе 9.9.3, с выборкой состояний из текущего убеждения.

[2] R. Platt Jr., R. Tedrake, L. P. Kaelbling, T. Lozano-Pérez, *Belief Space Planning Assuming Maximum Likelihood Observations*, in Robotics: Science and Systems, 2010.

22.3. Метод ветвей и границ

Метод ветвей и границ, первоначально предложенный для задач MDP, применим и для POMDP. Алгоритм из раздела 9.4 можно использовать без изменений (пример 22.3), снабдив его соответствующей реализацией предпросмотра для обновления убеждений и учета наблюдений. Эффективность алгоритма попрежнему зависит от качества верхней и нижней границ обрезки.

Хотя для определения верхней и нижней границ можно использовать эвристику, специфичную для предметной области, как мы это делали в случае полностью наблюдаемой задачи, здесь также подойдет один из методов, представленных в предыдущей главе для дискретных пространств состояний. Например, мы можем использовать метод быстрой инфограницы для оценки границы сверху и точечную итерацию по полезности для оценки границы снизу. Пока нижняя граница $\underline{U}$ и верхняя граница $\overline{Q}$ являются истинными нижней и верхней границами, результат применения алгоритма ветвей и границ будет таким же, как у алгоритма прямого поиска с $\underline{U}$ в качестве аппроксимированной функции полезности.

Пример 22.3. Применение метода ветвей и границ к задаче о плачущем ребенке

Применим метод ветвей и границ к задаче о плачущем ребенке с глубиной 5. Оценку границы сверху найдем по методу быстрой инфограницы, а оценку границы снизу получим путем точечной итерации по полезности. Вычислим действие из убеждения [0.4, 0.6] следующим образом:

```
k_max = 10 # максимальное количество итераций для оценки границ
πFIB = solve(FastInformedBound(k_max), 𝒫)
d = 5 # глубина
Uhi(b) = utility(πFIB, b)
Qhi(b,a) = lookahead(𝒫, Uhi, b, a)
B = [[p, 1 - p] for p in 0.0:0.2:1.0]
πPBVI = solve(PointBasedValueIteration(B, k_max), 𝒫)
Ulo(b) = utility(πPBVI, b)
π = BranchAndBound(𝒫, d, Ulo, Qhi)
π([0.4,0.6])
```

22.4. Разреженная выборка

Прямой поиск выполняет суммирование по всем возможным наблюдениям, что приводит к экспоненциальному времени выполнения $|\mathcal{O}|$. Как было показано в разделе 9.5, чтобы избежать взрывного роста вычислительной нагрузки, можно использовать ограниченную выборку. Мы можем сгенерировать m наблюдений для каждого действия, а затем вычислить

$$Q_d(b,a) = \begin{cases} \frac{1}{m}\sum_{i=1}^{m}\left(r_a^{(i)} + \gamma U_{d-1}\left(\text{Update}(b,a,o_a^{(i)})\right)\right), & \text{если } d > 0 \\ U(b) & \text{в ином случае} \end{cases}, \tag{22.2}$$

где $r_a^{(i)}$ и $o_a^{(i)}$ – наблюдение и вознаграждение в i-й выборке, связанные с действием a из убеждения b, а $U(b)$ – оценка функции ценности на максимальной глубине. Мы можем использовать алгоритм 9.4 без изменений. Итоговая сложность равна $O(|\mathcal{O}|^d m^d)$.

22.5. Поиск по дереву Монте-Карло

Метод поиска по дереву Монте-Карло для MDP можно распространить на POMDP, хотя в данном случае не получится обойтись без изменений в реализации[3]. Входными данными для алгоритма являются убеждение-состояние b, глубина d, коэффициент исследования c и стратегия развертывания π[4]. Основное отличие алгоритма POMDP (алгоритм 22.1) от алгоритма MDP заключается в том, что подсчеты и значения связаны с историями, а не с состояниями. *История* – это последовательность прошлых действий и наблюдений. Например, если у нас есть два действия a_1 и a_2 и два наблюдения o_1 и o_2, то возможной историей может быть последовательность $h = a_1o_2a_2o_2a_1o_1$. Во время выполнения алгоритма мы обновляем оценки полезности $Q(h, a)$ и подсчеты $N(h, a)$ для набора пар история-действие[5].

Алгоритм 22.1. Поиск по дереву Монте-Карло для задачи POMDP из убеждения `b`. Начальная история `h` не является обязательной. Этот код аналогичен реализации алгоритма 9.5

```
struct HistoryMonteCarloTreeSearch
    𝒫 # задача
    N # счетчик посещений
    Q # оценки полезности действия
```

---

[3] Сильвер и Венесс предложили алгоритм поиска по дереву Монте-Карло для POMDP под названием «Планирование Монте-Карло в условиях частичного наблюдения» (Partially Observable Monte Carlo Planning, POMCP) и продемонстрировали его сходимость. D. Silver, J. Veness, *Monte-Carlo Planning in Large POMDPs*, in Advances in Neural Information Processing Systems (NIPS), 2010.

[4] Поиск по дереву Монте-Карло может быть реализован с помощью стратегии развертывания POMDP, которая работает с убеждениями, или с помощью стратегии развертывания MDP, которая работает с состояниями. Обычно используются случайные стратегии.

[5] Существует множество вариаций базового алгоритма, в том числе использующие прием двойного прогрессивного расширения, обсуждавшийся в разделе 9.6. Z. N. Sunberg, M. J. Kochenderfer, *Online Algorithms for POMDPs with Continuous State, Action, and Observation Spaces*, in International Conference on Automated Planning and Scheduling (ICAPS), 2018.

```
    d # глубина
    m # количество прогонов модели
    c # постоянная исследования
    U # оценка функции полезности
end

function explore(π::HistoryMonteCarloTreeSearch, h)
    𝒜, N, Q, c = π.𝒫.𝒜, π.N, π.Q, π.c
    Nh = sum(get(N, (h,a), 0) for a in 𝒜)
    return argmax(a->Q[(h,a)] + c*bonus(N[(h,a)], Nh), 𝒜)
end

function simulate(π::HistoryMonteCarloTreeSearch, s, h, d)
    if d ≤ 0
        return π.U(s)
    end
    𝒫, N, Q, c = π.𝒫, π.N, π.Q, π.c
    𝒮, 𝒜, TRO, γ = 𝒫.𝒮, 𝒫.𝒜, 𝒫.TRO, 𝒫.γ
    if !haskey(N, (h, first(𝒜)))
        for a in 𝒜
            N[(h,a)] = 0
            Q[(h,a)] = 0.0
        end
        return π.U(s)
    end
    a = explore(π, h)
    s′, r, o = TRO(s,a)
    q = r + γ*simulate(π, s′, vcat(h, (a,o)), d-1)
    N[(h,a)] += 1
    Q[(h,a)] += (q-Q[(h,a)])/N[(h,a)]
    return q
end

function (π::HistoryMonteCarloTreeSearch)(b, h=[])
    for i in 1:π.m
        s = rand(SetCategorical(π.𝒫.𝒮, b))
        simulate(π, s, h, π.d)
    end
    return argmax(a->π.Q[(h,a)], π.𝒫.𝒜)
end
```

Истории, ассоциированные с Q и N, могут быть организованы в виде дерева, подобного изображенному на рис. 22.2. Корневой узел представляет собой пустую историю, начинающуюся с начального убеждения-состояния b. В ходе выполнения алгоритма древовидная структура расширяется. Слои дерева образованы чередованием между узлами действия и узлами наблюдения. С каждым узлом действия связаны значения полезности $Q(h, a)$ и подсчетов $N(h, a)$, а история представляет собой путь от корневого узла. Как и в версии MDP, при поиске вниз по дереву алгоритм выполняет действие, которое максимизирует полезность

$$Q(h,a) + c\sqrt{\frac{\log N(h)}{N(h,a)}}, \tag{22.3}$$

где $N(h) = \sum_a N(h, a)$ – общее количество посещений для истории h, а c – параметр исследования. Важно отметить, что c увеличивает ценность неисследованных и недостаточно исследованных действий, тем самым создавая некий компромисс между исследованием и использованием.

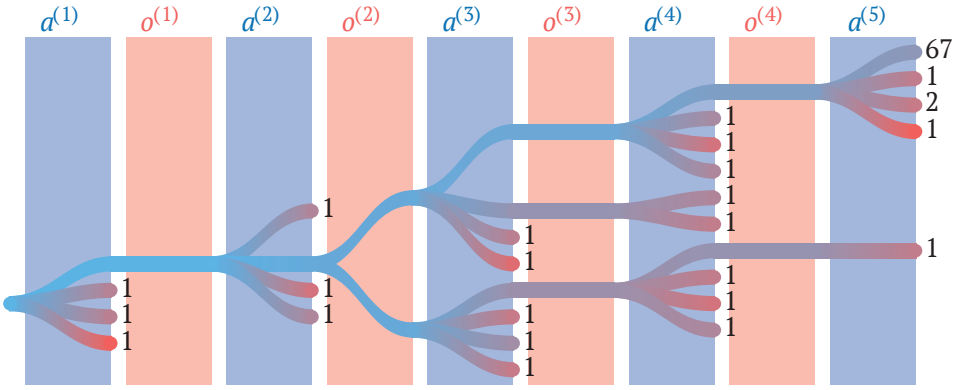


Рис. 22.2. Дерево поиска, содержащее все истории, охваченные при выполнении поиска по дереву Монте-Карло со 100 выборками в задаче о ремонте машины. Подсчеты посещений указаны под каждым узлом действия, а цвет указывает полезность узла: высокие значения обозначены синим цветом, а низкие – красным. Расширенные узлы с нулевыми посещениями не показаны. В этом поиске использовалась постоянная исследования c = 0.5, максимальная глубина d = 5 и общая стратегия случайного развертывания. Первоначальное убеждение – это уверенность в полностью исправной машине. Поиск по дереву Монте-Карло позволяет избежать определенных действий и вместо этого сосредоточить выборки на более перспективных путях

Как и в версии MDP, алгоритм поиска по дереву Монте-Карло не зависит от времени выполнения. Цикл в алгоритме 22.1 можно прервать в любой момент, и будет возвращено лучшее решение, найденное к этому моменту. При достаточном количестве итераций алгоритм сходится к оптимальному действию.

Априорные знания можно добавить в поиск по дереву Монте-Карло в виде начальных значений N и Q. В нашей реализации применяются нулевые значения, но возможны и другие варианты, в том числе инициализация полезности действия как функции от истории. Оценки полезности в данном случае тоже могут быть получены путем моделирования стратегии развертывания.

Алгоритм не нужно повторно инициализировать при каждом решении. Достаточно хранить между вызовами дерево истории и связанные с ним подсчеты и оценки полезности. Узел наблюдения, ассоциированный с выбранным действием и фактическим наблюдением, становится корневым узлом на следующем шаге.

22.6. Поиск по детерминированному разреженному дереву

Поиск по детерминированному разреженному дереву (determinized sparse tree search) направлен на уменьшение общего объема выборки как при использовании метода разреженной выборки, так и при поиске по дереву Монте-Карло за счет наблюдения, полученного в результате выполнения детерминированного действия[6]. Он делает это путем построения *детерминированного дерева убеждений* (determinized belief tree) из специального парциального представления убеждений, чтобы сформировать разреженное приближение к истинному дереву убеждений. Каждая *частица* (раздел 19.6) в разреженном представлении относится к одному из m *сценариев*, глубина каждого из которых равна d. Сценарий представляет собой фиксированную историю, которой будет следовать любая заданная последовательность действий $a^{(1)}, a^{(2)}, \ldots, a^{(d)}$. Каждая отдельная последовательность действий создает отдельную историю в рамках определенного сценария[7]. Эта детерминация уменьшает размер дерева поиска до $O(|\mathcal{A}|^d m)$. Историю иллюстрирует пример 22.4. Детерминированное дерево показано на рис. 22.3.

Пример 22.4. История и сценарий в контексте поиска по детерминированному разреженному дереву

Предположим, у нас есть два состояния s_1 и s_2, два действия a_1 и a_2 и два наблюдения o_1 и o_2. Возможная история с глубиной $d = 2$ для частицы с начальным состоянием s_2 представляет собой последовательность $h = s_2 a_1 o_2 s_1 a_2 o_1$. Если история используется в качестве сценария, то она каждый раз возвращает дерево убеждений, пройденное из начального состояния с последовательностью действий $a^{(1)} = a_1$ и $a^{(2)} = a_2$.

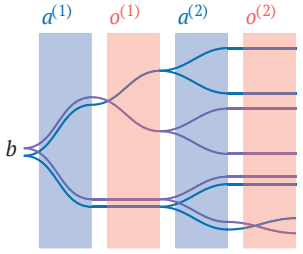


Рис. 22.3. Детерминированное разреженное дерево поиска с двумя сценариями, изображенными синим и фиолетовым цветами. Линии показывают возможные пути для каждого сценария при различных последовательностях действий

[6] Йе, Сомани, Хсу и Ли предложили алгоритм поиска в детерминированном разреженном дереве для POMDP, называемый *детерминированным разреженным частично наблюдаемым деревом* (Determinized Sparse Partially Observable Tree, DESPOT). N. Ye, A. Somani, D. Hsu, W. S. Lee, *DESPOT: Online POMDP Planning with Regularization*, Journal of Artificial Intelligence Research, vol. 58, pp. 231–266, 2017. Этот алгоритм также использует методы ветвей и границ, эвристического поиска и регуляризации.

[7] Похожая идея обсуждалась в разделе 9.9.3 и связана с алгоритмом PEGASUS, упомянутым в разделе 11.1.

Дерево поиска с m сценариями до глубины d может быть полностью задано компактной *детерминирующей матрицей* (determinizing matrix) **Ф** размерностью $m \times d$, содержащей вероятностные меры. Элемент Φ_{ij} содержит информацию, необходимую частице, следующей по i-му сценарию на глубине j, чтобы идентифицировать ее последующее состояние и наблюдение. В частности, Φ_{ij} – это случайное число из равномерного распределения, которое может генерировать последующую пару (s', o) из пары состояние-действие (s, a), следуя распределению $P(s', o \mid s, a) = T(s' \mid s, a)O(o \mid a, s')$. Можно сгенерировать детерминирующую матрицу, заполнив ее значениями, равномерно выбранными между 0 и 1.

Убеждения представлены в виде векторов частиц убеждений. Каждая частица убеждения φ содержит состояние s и индексы i и j в детерминирующей матрице **Ф**, соответствующие сценарию i и текущей глубине j. Φ_{ij} используется для детерминированного перехода к последующему состоянию s' и наблюдению o при данном заданном действии a. Последующая частица $\varphi' = (s', i, j + 1)$ получает s' в качестве своего состояния и увеличивает j на 1. Процесс обхода дерева продемонстрирован в примере 22.5. Представление убеждений в виде частиц реализовано в алгоритме 22.2 и используется в прямом поиске (алгоритм 22.3).

Пример 22.5. Поиск по детерминированному разреженному дереву использует матрицу, чтобы сделать обход дерева детерминированным для данной частицы

Допустим, мы генерируем детерминирующую матрицу **Ф** для задачи с четырьмя историями до глубины 3:

$$\mathbf{\Phi} = \begin{bmatrix} 0.393 & 0.056 & 0.369 \\ 0.313 & 0.749 & 0.273 \\ 0.078 & 0.262 & 0.009 \\ 0.969 & 0.598 & 0.095 \end{bmatrix}.$$

Предположим, что мы предпринимаем действие a_3 в состоянии s_2, находясь на глубине 2, следуя истории 3. Соответствующая частица убеждения равна $\varphi = (2, 3, 2)$, а определяющее значение в **Ф** равно $\Phi_{3,2} = 0.262$.

Детерминированное последующее действие и наблюдение задаются путем итеративного перебора всех последующих пар состояний-наблюдений и накопления их вероятностей перехода. Начнем с $p = 0$ и вычислим $s' = s_1$, $o = o_1$. Предположим, что мы получили $T(s_1|s_2, a_3)O(o_1|a_3, s_1) = 0.1$. Увеличиваем p до 0.1, что меньше, чем $\Phi_{3,2}$, поэтому продолжаем поиск.

Далее находим $s' = s_1$, $o = o_2$. Предположим, что мы получили $T(s_1|s_2, a_3)$ $O(o_2|a_3, s_2) = 0.17$. Увеличиваем p до 0.27, что превышает $\Phi_{3,2}$. Следовательно, мы детерминистически переходим к $s' = s_1$, $o = o_2$ в качестве последующего состояния, в результате чего получается новая частица $\varphi' = (1, 3, 3)$.

Алгоритм 22.2. Детерминированное обновление частичного убеждения, используемое в поиске по детерминированному разреженному дереву для задачи POMDP $\mathcal{P}$. Каждое убеждение `b` состоит из частиц `φ`, каждая из которых кодирует определенный сценарий и глубину сценария. Траектория их сценария детерминирована при помощи матрицы `Φ`, содержащей случайные значения в диапазоне [0, 1]. Каждая частица `φ` представляет конкретный сценарий `i` на определенной глубине `j`, относящийся к `i`-й строке и `j`-му столбцу `Φ`

```
struct DeterminizedParticle
    s # состояние
    i # индекс сценария
    j # индекс глубины
end

function successor(𝒫, Φ, φ, a)
    𝒮, 𝒪, T, O = 𝒫.𝒮, 𝒫.𝒪, 𝒫.T, 𝒫.O
    p = 0.0
    for (s′, o) in product(𝒮, 𝒪)
        p += T(φ.s, a, s′) * O(a, s′, o)
        if p ≥ Φ[φ.i, φ.j]
            return (s′, o)
        end
    end
    return last(𝒮), last(𝒪)
end

function possible_observations(𝒫, Φ, b, a)
    𝒪 = []
    for φ in b
        s′, o = successor(𝒫, Φ, φ, a)
        push!(𝒪, o)
    end
    return unique(𝒪)
end

function update(b, Φ, 𝒫, a, o)
    b′ = []
    for φ in b
        s′, o′ = successor(𝒫, Φ, φ, a)
        if o == o′
            push!(b′, DeterminizedParticle(s′, φ.i, φ.j + 1))
        end
    end
    return b′
end
```

Алгоритм 22.3. Реализация поиска по детерминированному разреженному дереву (модификация прямого поиска для POMDP). Стратегия получает убеждение b в виде вектора вероятностей, который аппроксимируется вектором детерминированных частиц с помощью `determinized_belief`

```
struct DeterminizedSparseTreeSearch
    𝒫 # задача
    d # глубина
    ϕ # детерминирующая матрица m×d
    U # функция полезности для использования в листовых узлах
end

function determinized_sparse_tree_search(𝒫, b, d, ϕ, U)
    𝒮, 𝒜, 𝒪, T, R, O, γ = 𝒫.𝒮, 𝒫.𝒜, 𝒫.𝒪, 𝒫.T, 𝒫.R, 𝒫.O, 𝒫.γ
    if d == 0
        return (a=nothing, u=U(b))
    end
    best = (a=nothing, u=-Inf)
    for a in 𝒜
        u = sum(R(φ.s, a) for φ in b) / length(b)
        for o in possible_observations(𝒫, ϕ, b, a)
            Poba = sum(sum(O(a,s′,o)*T(φ.s,a,s′) for s′ in 𝒮)
                       for φ in b) / length(b)
            b′ = update(b, ϕ, 𝒫, a, o)
            u′ = determinized_sparse_tree_search(𝒫,b′,d-1,ϕ,U).u
            u += γ*Poba*u′
        end
        if u > best.u
            best = (a=a, u=u)
        end
    end
    return best
end

function determinized_belief(b, 𝒫, m)
    particles = []
    for i in 1:m
        s = rand(SetCategorical(𝒫.𝒮, b))
        push!(particles, DeterminizedParticle(s, i, 1))
    end
    return particles
end

function (π::DeterminizedSparseTreeSearch)(b)
    particles = determinized_belief(b, π.𝒫, size(π.ϕ,1))
    return determinized_sparse_tree_search(π.𝒫,particles,π.d,π.ϕ,π.U).a
end
```

22.7. Эвристический поиск на основе разности границ

Подобно эвристическому поиску в офлайн-режиме, представленному в разделе 21.8, метод *эвристического поиска на основе разности границ* (gap heuristic search) использует разрыв между верхней и нижней границами, чтобы направить наш поиск к убеждениям, имеющим неопределенность в связанной с ними полезности, и как указание на то, когда мы можем прекратить исследование. Разрыв при убеждении b – это разность между верхним и нижним граничными значениями функции полезности: $\overline{U}(b) - \underline{U}(b)$. Алгоритмы эвристического поиска на основе разности границ выбирают наблюдение, которое максимизирует разрыв, потому что они с большей вероятностью выигрывают от операции дублирования убеждений. Действия часто выбирают методом предпросмотра с использованием аппроксимированной функции полезности. Реализация метода представлена в алгоритме 22.4[8].

Начальные значения нижней и верхней границ, используемые в эвристическом поиске, имеют большое значение для качества работы алгоритма. В примере 22.6 применяется стратегия случайного развертывания для нижней границы $\underline{U}(b)$. Развертывание, конечно, не гарантирует получение достоверной нижней границы, потому что оно основано на единственном испытании до фиксированной глубины. По мере увеличения количества выборок оценка будет сходиться к истинной нижней границе. В этом примере используется верхняя граница по методу наилучшего состояния – наилучшего действия из уравнения (21.2). Существует множество других способов оценки верхних и нижних границ, обеспечивающих более быструю сходимость, но за счет увеличения времени выполнения и сложности реализации. Например, оценка по методу быстрой инфограницы (алгоритм 21.3) для верхней границы помогает улучшить исследование и сократить разрыв. Выполняя оценку границы снизу, мы можем применить стратегию развертывания для конкретной задачи, чтобы лучше направлять поиск.

Алгоритм 22.4. Реализация эвристического поиска, в которой используются границы, критерий разрыва и начальные значения нижней и верхней границ функции полезности. Мы обновляем словари `Ulo` и `Uhi`, чтобы представить нижнюю и верхнюю границы функции полезности как конкретные убеждения. При убеждении `b` разрыв равен `Uhi[b] - Ulo[b]`. Исследование прекращается, когда разрыв становится меньше порога δ или достигается максимальная глубина `d_max`. На поиск отводится максимальное количество итераций `k_max`

```
struct GapHeuristicSearch
    𝒫 # задача
```

---

[8] Существует множество различных алгоритмов эвристического поиска для POMDP, которые пытаются минимизировать разрыв. Например, см. S. Ross and B. Chaib-draa, *AEMS: An Anytime Online Search Algorithm for Approximate Policy Refinement in Large POMDPs*, in International Joint Conference on Artificial Intelligence (IJCAI), 2007. Эта реализация аналогична той, которая использует DESPOT, упомянутый в предыдущем разделе.

```
    Ulo # граница снизу функции полезности
    Uhi # граница сверху функции полезности
    δ # порог разрыва между границами
    k_max # максимальное количество прогонов модели
    d_max # максимальная глубина
end

function heuristic_search(π::GapHeuristicSearch, Ulo, Uhi, b, d)
    𝒫, δ = π.𝒫, π.δ
    𝒮, 𝒜, 𝒪, R, γ = 𝒫.𝒮, 𝒫.𝒜, 𝒫.𝒪, 𝒫.R, 𝒫.γ
    B = Dict((a,o)=>update(b,𝒫,a,o) for (a,o) in product(𝒜,𝒪))
    B = merge(B, Dict(()=>copy(b)))
    for (ao, b′) in B
        if !haskey(Uhi, b′)
            Ulo[b′], Uhi[b′] = π.Ulo(b′), π.Uhi(b′)
        end
    end
    if d == 0 || Uhi[b] - Ulo[b] ≤ δ
        return
    end
    a = argmax(a -> lookahead(𝒫,b′->Uhi[b′],b,a), 𝒜)
    o = argmax(o -> Uhi[B[(a, o)]] - Ulo[B[(a, o)]], 𝒪)
    b′ = update(b,𝒫,a,o)
    heuristic_search(π,Ulo,Uhi,b′,d-1)
    Ulo[b] = maximum(lookahead(𝒫,b′->Ulo[b′],b,a) for a in 𝒜)
    Uhi[b] = maximum(lookahead(𝒫,b′->Uhi[b′],b,a) for a in 𝒜)
end

function (π::GapHeuristicSearch)(b)
    𝒫, k_max, d_max, δ = π.𝒫, π.k_max, π.d_max, π.δ
    Ulo = Dict{Vector{Float64}, Float64}()
    Uhi = Dict{Vector{Float64}, Float64}()
    for i in 1:k_max
        heuristic_search(π, Ulo, Uhi, b, d_max)
        if Uhi[b] - Ulo[b] < δ
            break
        end
    end
    return argmax(a -> lookahead(𝒫,b′->Ulo[b′],b,a), 𝒫.𝒜)
end
```

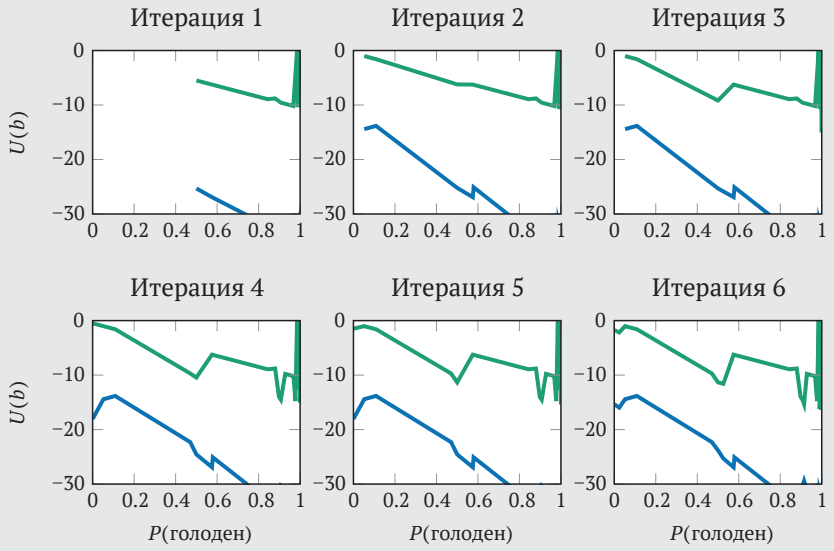
Пример 22.6. Использование нижней и верхней границ эвристического поиска для решения задачи о плачущем ребенке путем итераций эвристического поиска

В следующем коде показано, как применить эвристический поиск на основе разницы между границами к задаче о плачущем ребенке.

```
δ = 0.001  # порог разрыва
k_max = 5  # максимальное количество итераций
d_max = 10 # maximum depth
```

```
πrollout(b) = rand(𝒜) # стратегия случайного развертывания
Ulo(b) = rollout(𝒫, b, πrollout, d_max) # начальная нижняя граница
Rmax = maximum(R(s,a) for (s,a) in product(𝒮,𝒜)) # максимальное вознаграждение
Uhi(b) = Rmax / (1.0 - 𝒫.γ) # верхняя граница по методу «лучшее действие –
                                лучшее состояние»
π = GapHeuristicSearch(𝒫, Ulo, Uhi, δ, k_max, d_max)
π([0.5, 0.5]) # оценка в точке начального убеждения
```

Ниже показаны шесть итераций эвристического поиска с начальным убеждением b, равным [0.5, 0.5]. На каждой итерации верхняя граница отображается зеленым цветом, а нижняя – синим.



Границы зубчатой формы возникают из-за того, что некоторые убеждения не исследуются повторно. В нижнем ряду видно, что поиск исследовал многие убеждения по одному разу, но границы все еще расплывчаты. Эвристический поиск направлен на уменьшение максимального разрыва.

22.8. Заключение

- Простая онлайн-стратегия заключается в выполнении пошагового предпросмотра, который рассматривает каждое действие, предпринятое на основе текущего убеждения, и оценивает его ожидаемую полезность с использованием аппроксимированной функции полезности.
- Прямой поиск – это обобщение предпросмотра на произвольные горизонты, что может привести к улучшению стратегий, но его вычислительная сложность растет экспоненциально с увеличением горизонта.

- Метод ветвей и границ – это более эффективная версия прямого поиска, позволяющая избежать поиска по определенным путям за счет ограничения функции полезности сверху и снизу.
- Разреженная выборка – это метод аппроксимации, который снижает вычислительную нагрузку от итераций по пространству всех возможных наблюдений.
- Поиск по дереву Монте-Карло можно адаптировать к POMDP, оперируя историями, а не состояниями.
- Поиск по детерминированному разреженному дереву использует особую форму убеждения, которая гарантирует детерминированность наблюдений, что значительно сокращает дерево поиска.
- Эвристический поиск интеллектуально выбирает пары действие–наблюдение для изучения областей с большим разрывом между верхней и нижней границами функции полезности.

22.9. Упражнения

Упражнение 22.1. Предположим, нам даны $\mathcal{A} = \{a^1, a^2\}$ и убеждение $\mathbf{b} = [0.5, 0.5]$. Награда всегда равна 1. Функция наблюдения определяется как $P(o^1|a^1) = 0.8$ и $P(o^1|a^2) = 0.4$. У нас есть аппроксимированная функция полезности, представленная альфа-вектором $\boldsymbol{\alpha} = [-3, 4]$. Используйте прямой поиск на глубину 1 для вычисления $U(\mathbf{b})$ при $\gamma = 0.9$. Используйте в расчетах следующие обновленные убеждения:

a	o	Update($\mathbf{b}, a, o$)
a^1	o^1	[0.3, 0.7]
a^2	o^1	[0.2, 0.8]
a^1	o^2	[0.5, 0.5]
a^2	o^2	[0.8, 0.2]

Решение. Нам нужно рассчитать функцию полезности действия на глубине 1 в соответствии с уравнением (22.1):

$$Q_d(\mathbf{b}, a) = R(\mathbf{b}, a) + \gamma \sum_o P(o|b, a) U_{d-1}(\text{Update}(\mathbf{b}, a, o)).$$

Сначала вычислим полезность для обновленных убеждений:

$U_0(\text{Update}(\mathbf{b}, a^1, o^1)) = \boldsymbol{\alpha}^\top \mathbf{b}' = 0.3 \times -3 + 0.7 \times 4 = 1.9;$

$U_0(\text{Update}(\mathbf{b}, a^2, o^1)) = 0.2 \times -3 + 0.8 \times 4 = 2.6;$

$U_0(\text{Update}(\mathbf{b}, a^1, o^2)) = 0.5 \times -3 + 0.5 \times 4 = 0.5;$

$U_0(\text{Update}(\mathbf{b}, a^2, o^2)) = 0.8 \times -3 + 0.2 \times 4 = -1.6.$

Затем вычислим функцию полезности действия для обоих действий:

$$Q_1(\mathbf{b}, a^1) = 1 + 0.9((P(o_1|\mathbf{b}, a^1)U_0(\text{Update}(\mathbf{b}, a^1, o^1)) + (P(o^2|\mathbf{b}, a^1)U_0(\text{Update}(\mathbf{b}, a^1, o^2)))$$
$$= 1 + 0.9(0.8 \times 1.9 + 0.2 \times 0.5) = 2.458;$$

$$Q_1(\mathbf{b}, a^2) = 1 + 0.9((P(o^1|\mathbf{b}, a^2)U_0(\text{Update}(\mathbf{b}, a^2, o^1)) + (P(o^2|\mathbf{b}, a^2)U_0(\text{Update}(\mathbf{b}, a^2, o^2)))$$
$$= 1 + 0.9(0.4 \times 2.6 + 0.6 \times -1.6) = 1.072.$$

Наконец, получаем $U_1(\mathbf{b}) = \max_a Q_1(\mathbf{b}, a) = 2.458$.

Упражнение 22.2. Используя следующие выборки траекторий, вычислите функцию полезности действия для убеждения **b** и действий a^1 и a^2 на основе разреженной выборки до глубины 1. Используйте следующие обновленные убеждения, коэффициент дисконтирования $\gamma = 0.9$ и аппроксимированную функцию полезности, представленную альфа-вектором $\boldsymbol{\alpha} = [10, 1]$.

a	o	r	Update($\mathbf{b}$, a, o)
1	1	0	[0.47, 0.53]
2	1	1	[0.22, 0.78]
1	2	1	[0.49, 0.51]
2	1	1	[0.22, 0.78]
2	2	1	[0.32, 0.68]
1	2	1	[0.49, 0.51]

Решение. Сначала найдем полезность обновленных убеждений:

a	o	r	Update ($\mathbf{b}$, a, o_a)	U_0(Update($\mathbf{b}$, a, o))
1	1	0	[0.47, 0.53]	5.23
2	1	1	[0.22, 0.78]	2.98
1	2	1	[0.49, 0.51]	5.41
2	1	1	[0.22, 0.78]	2.98
2	2	1	[0.32, 0.68]	3.88
1	2	1	[0.49, 0.51]	5.41

Затем вычислим функцию полезности действия для всех действий, используя уравнение (22.2):

$$Q_1(\mathbf{b}, a^1) = 1/3(0 + 1 + 1 + 0.9(5.23 + 5.41 + 5.41)) = 5.48;$$

$$Q_1(\mathbf{b}, a^2) = 1/3(1 + 1 + 1 + 0.9(2.98 + 2.98 + 3.88)) = 3.95.$$

Упражнение 22.3. Вспомним пример 22.5. Предположим, у нас есть следующие функции перехода:

$$T(s_2|s_1, a_3) = 0.4; \qquad O(o_1|s_1, a_3) = 0.6;$$

$$T(s_3|s_1, a_3) = 0.45; \qquad O(o_2|s_1, a_3) = 0.5.$$

Какой путь пройдет частица, ассоциированная с $\varphi = (1, 4, 2)$, если мы предпримем действие a_3?

Решение. Из детерминирующей матрицы следует, что наше детерминирующее значение равно $\Phi_{4,2} = 0.598$ и мы находимся в состоянии s_1. Затем мы вычисляем p следующим образом:

$$p \leftarrow T(s_2|s_1, a_3)O(o_1|s_1, a_3) = 0.4 \times 0.6 = 0.24;$$

$$p \leftarrow p + T(s_2|s_1, a_3)O(o_2|s_1, a_3) = 0.24 + 0.4 \times 0.5 = 0.44;$$

$$p \leftarrow p + T(s_3|s_1, a_3)O(o_1|s_1, a_3) = 0.44 + 0.45 \times 0.6 = 0.71.$$

Мы останавливаем итерацию, потому что $p > 0.598$. Таким образом, от нашей последней итерации мы переходим к (s_3, o_1).

Упражнение 22.4. Обобщите приемы, описанные в этой главе, направленные на уменьшение ветвлений по действиям.

Решение. Метод ветвей и границ может уменьшить количество ветвлений действия, используя оценку сверху функции полезности. Он пропускает действия, которые не могут улучшить полезность, полученную от действий, исследованных ранее. Эвристический поиск на основе разности границ и поиск по дереву Монте-Карло используют аппроксимированные функции полезности действий, чтобы направлять выбор действий во время исследования.

Упражнение 22.5. Обобщите приемы, описанные в этой главе, направленные на уменьшение ветвлений по наблюдениям.

Решение. Метод разреженной выборки уменьшает ветвление наблюдений за счет выборки только небольшого числа наблюдений. Наблюдения выбираются из $P(o|b, a)$, то есть, скорее всего, будут выбраны наблюдения с большей вероятностью. Поиск по детерминированному разреженному дереву использует аналогичный подход, но выборка выполняется один раз, а затем фиксируется. Ветвление по наблюдениям также может быть уменьшено с использованием предпросмотра полезности $U(b')$. Эвристический поиск на основе разности границ оценивает разрыв между границами и избегает ветвления по наблюдениям, для которых у нас уже есть высокая уверенность в функции полезности.

23 Понятие контроллера

В этой главе мы рассмотрим представления контроллера стратегий POMDP, которые позволяют стратегиям сохранять собственное внутреннее состояние. Эти представления дают возможность улучшить масштабируемость по сравнению с предыдущими методами, которые перебирают точки в пространстве убеждений. Здесь представлены алгоритмы, которые создают контроллеры, используя итерацию по стратегиям, нелинейное программирование и градиентное восхождение.

23.1. Контроллеры

Контроллер – это представление стратегии, которое способно хранить собственное внутреннее состояние. Контроллер представляют в виде графа, состоящего из конечного набора узлов X[1]. Активный узел изменяется по мере выполнения действий и новых наблюдений. Наличие конечного набора узлов значительно снижает вычислительную нагрузку по сравнению с методами, которые должны охватывать все достижимое пространство убеждений.

Действия выбираются в соответствии с *распределением вероятностей действий* (action distribution) $\psi(a|x)$, которое зависит от текущего узла. При выборе действия, кроме перехода к ненаблюдаемому состоянию s' и получению наблюдения o, *состояние управления* (control state) также продвигается в соответствии с *распределением вероятностей последующих элементов* (successor distribution) $\eta(x'|x, a, o)$. На рис. 23.1 показано, как эти распределения применяются при следовании стратегии контроллера. Реализация представлена в алгоритме 23.1, а в примере 23.1 продемонстрировано использование контроллера для решения задачи о плачущем ребенке.

Контроллеры являются обобщением условных планов, которые были рассмотрены в разделе 20.2. Условные планы представляют стратегии в виде деревьев, где каждому узлу детерминированно назначается действие, а каждому ребру задается уникальный узел-последователь. Контроллеры представляют стратегии в виде направленных графов, а действия могут иметь стохастиче-

---

[1] Такое представление стратегии также называется *контроллером конечного автомата* (finite state controller). Мы будем называть состояния контроллера «узлами», а не «состояниями», чтобы избежать путаницы с состояниями среды.

ские переходы к нескольким последующим узлам. В примере 23.2 сравниваются эти два представления.

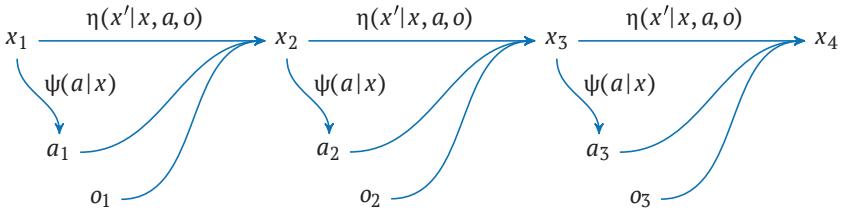
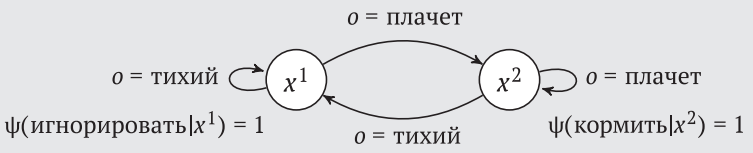


Рис. 23.1. При использовании контроллера очередное действие выбирается из распределения вероятностей действий. Это действие, а также последующее наблюдение, которое оно производит, используются вместе с предыдущим узлом *x* для создания последующего узла *x'*

Пример 23.1. Контроллер из двух узлов для решения задачи о плачущем ребенке. Это компактное представление простого решения задачи – немедленная реакция на самое последнее наблюдение

Построим простой контроллер для решения задачи о плачущем ребенке (приложение F.7). На рисунке ниже он изображен как граф с двумя узлами x^1 и x^2. В x^1 контроллер всегда игнорирует ребенка. В режиме x^2 контроллер всегда кормит ребенка. Если ребенок плачет, мы всегда переходим к x^2, а если ребенок молчит, мы всегда переходим к x^1.



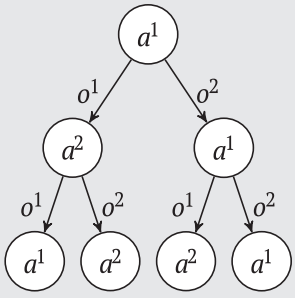
Алгоритм 23.1. Реализация стратегии контроллера конечного автомата для задачи POMDP $\mathcal{P}$. Узлы в `X` являются абстрактным представлением достижимых убеждений. Действия и узлы-преемники контроллера выбираются стохастически. Для заданного узла `x` действия выбираются в соответствии с распределением `ψ`. Функция `π(x)` реализует этот механизм стохастического выбора действий. После выполнения действия `a` в узле `x` и получения наблюдения `o` преемник выбирается в соответствии с распределением `η`. В функции `update` реализован механизм стохастического выбора узлов-преемников

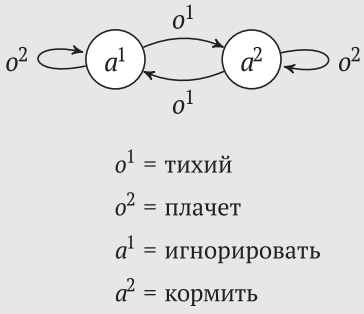
```
mutable struct ControllerPolicy
    𝒫 # problem
    X # set of controller nodes
    ψ # action selection distribution
    η # successor selection distribution
end
```

```
function (π::ControllerPolicy)(x)
    𝒜, ψ = π.𝒫.𝒜, π.ψ
    dist = [ψ[x, a] for a in 𝒜]
    return rand(SetCategorical(𝒜, dist))
end

function update(π::ControllerPolicy, x, a, o)
    X, η = π.X, π.η
    dist = [η[x, a, o, x′] for x′ in X]
    return rand(SetCategorical(X, dist))
end
```

Пример 23.2. Сравнение простого условного плана и простого детерминированного контроллера

Сравним трехшаговый условный план (слева) с более общим двухузловым контроллером конечного автомата (справа) из примера 23.1. В данном случае действия и преемники выбираются детерминированно. Детерминированное действие обозначено в центре узла, а исходящие ребра ведут к детерминированным последующим узлам. В этой задаче есть два действия (a^1 и a^2) и два наблюдения (o^1 и o^2).





Условный план сначала выполняет действие a^1, меняет ранее выбранное действие, если соблюдено условие o^1, и сохраняет ранее выбранное действие, если соблюдено условие o^2. Контроллер следует той же логике, но содержит на пять узлов меньше. Более того, контроллер идеально представляет описанную стратегию бесконечного горизонта только с двумя узлами (в отличие от семи в условном плане). Условный план не может охватить стратегию бесконечного горизонта, поскольку для этого потребуется дерево бесконечной глубины.

Контроллеры имеют несколько преимуществ перед условными планами. Во-первых, контроллеры обеспечивают более компактное представление. Количество узлов в условном плане экспоненциально растет с глубиной, но этого, как правило, не случается с контроллерами конечного автомата. Методы

аппроксимации из предыдущих глав также не очень эффективно работают с условными планами, поскольку приходится поддерживать большой набор допущений и соответствующих альфа-векторов. Контроллеры намного компактнее, потому что могут представить бесконечное множество возможных достижимых убеждений при помощи небольшого конечного количества узлов. Другое преимущество контроллеров состоит в том, что они не требуют сохранения убеждений. Каждый узел контроллера соответствует подмножеству пространства убеждений. Эти подмножества не обязательно являются взаимоисключающими. Переходя между узлами, контроллер осуществляет переходы между этими подмножествами, которые вместе охватывают достижимое пространство убеждений. Контроллер сам выбирает новый узел на основе каждого наблюдения, а не полагается на обновление убеждений, что в некоторых случаях бывает очень затратно.

Полезность следования стратегии контроллера можно вычислить, используя произведение MDP, пространство состояний которого равно $X \times \mathcal{S}$. Полезность пребывания в состоянии s с активным узлом x равна

$$U(x,s) = \sum_a \psi(a|x)\left(R(s,a) + \gamma \sum_{s'} T(s'|s,a) \sum_o O(o|a,s') \sum_{x'} \eta(x'|x,a,o) U(x',s')\right). \quad (23.1)$$

Оценка стратегии подразумевает решение системы линейных уравнений в (23.1). В качестве альтернативы мы можем применить итеративную оценку стратегии, как показано в алгоритме 23.2.

Если убеждение известно, то текущая полезность равна

$$U(x,b) = \sum_s b(s) U(x,s). \quad (23.2)$$

Мы можем рассматривать $U(x, s)$ как определение набора альфа-векторов, по одному для каждого узла x в X. Каждый альфа-вектор $\boldsymbol{\alpha}_x$ определяется как $\alpha_x(s) = U(x, s)$. Текущая полезность для данного альфа-вектора равна $U(x, b) = \mathbf{b}^\top \boldsymbol{\alpha}_x$.

Если нам даны контроллер и начальное убеждение, мы можем выбрать начальный узел, найдя максимум следующего выражения:

$$x^* = \arg\max_x U(x,b) = \arg\max_x \mathbf{b}^\top \boldsymbol{\alpha}_x. \quad (23.3)$$

Алгоритм 23.2. Алгоритм выполнения итеративной оценки по стратегиям для вычисления полезности контроллера конечного автомата `π` с `k_max` итераций. Функция полезности выполняет одношаговую оценку для текущего узла контроллера `x` и состояния `s` в соответствии с уравнением (23.1). Этот алгоритм был адаптирован из алгоритма 7.3, который применяет итеративную оценку по стратегиям к задаче MDP

```
function utility(π::ControllerPolicy, U, x, s)
    𝒮, 𝒜, 𝒪 = π.𝒫.𝒮, π.𝒫.𝒜, π.𝒫.𝒪
    T, O, R, γ = π.𝒫.T, π.𝒫.O, π.𝒫.R, π.𝒫.γ
```

```
    X, ψ, η = π.X, π.ψ, π.η
    U′(a,s′,o) = sum(η[x,a,o,x′]*U[x′,s′] for x′ in X)
    U′(a,s′) = T(s,a,s′)*sum(O(a,s′,o)*U′(a,s′,o) for o in 𝒪)
    U′(a) = R(s,a) + γ*sum(U′(a,s′) for s′ in 𝒮)
    return sum(ψ[x,a]*U′(a) for a in 𝒜)
end

function iterative_policy_evaluation(π::ControllerPolicy, k_max)
    𝒮, X = π.𝒫.𝒮, π.X
    U = Dict((x, s) => 0.0 for x in X, s in 𝒮)
    for k in 1:k_max
        U = Dict((x, s) => utility(π, U, x, s) for x in X, s in 𝒮)
    end
    return U
end
```

23.2. Итерация по стратегиям

В разделе 20.5 было рассмотрено постепенное добавление узлов в условный план для получения оптимальной стратегии конечного горизонта (алгоритм 20.8). В этом разделе мы покажем, как постепенно добавлять узлы в контроллер, чтобы оптимизировать решение задач с бесконечным горизонтом. Хотя представление отличается, версия алгоритма итерации по стратегиям для частично наблюдаемых задач, упоминаемая в этом разделе[2], имеет некоторое сходство с алгоритмом итерации по стратегиям для полностью наблюдаемых задач (раздел 7.4).

Итерация по стратегиям (алгоритм 23.3) начинается с произвольного начального контроллера, а затем происходит чередование между оценкой и улучшением стратегии. На этапе оценки стратегии вычисляют полезность $U(x, s)$, решая уравнение (23.1). На этапе улучшения стратегии вводят новые узлы в контроллер. В частности, вводят новый узел x' для каждой комбинации детерминированных присваиваний действий $\psi(a_i|x') = 1$ и детерминированных распределений выбора преемника $\eta(x|x', a, o)$. Этот процесс добавляет $|\mathcal{A}||X^{(k)}|^{|\mathcal{O}|}$ новых узлов контроллера в набор узлов $X^{(k)}$ на итерации k[3]. Шаг улучшения показан в примере 23.3.

---

[2] Приведенный здесь метод итерации по стратегиям был предложен в докладе E. A. Hansen, *Solving POMDPs by Searching in Policy Space* на конференции по неопределенности в искусственном интеллекте.

[3] Добавление всех возможных комбинаций зачастую невозможно. Альтернативный алгоритм, называемый *ограниченной итерацией по стратегиям*, добавляет только один узел. P. Poupart, C. Boutilier, *Bounded Finite State Controllers*, in Advances in Neural Information Processing Systems (NIPS), 2003. Алгоритмы также могут добавлять промежуточное число. *Итерация по значимости Монте-Карло*, например, добавляет $O(n|\mathcal{A}||X^{(k)}|)$ новых узлов на каждой итерации k, где n – параметр. H. Bai, D. Hsu, W. S. Lee, V. A. Ngo, *Monte Carlo Value Iteration for Continuous-State POMDPs*, in International Workshop on the Algorithmic Foundations of Robotics (WAFR), 2011.

Улучшение стратегии не может ухудшить ожидаемую полезность стратегии контроллера. Полезность любых узлов в $X^{(k)}$ остается неизменной, поскольку они сами и их достижимые последующие узлы остаются неизменными. Гарантируется, что если $X^{(k)}$ не является оптимальным контроллером, то по крайней мере один из новых узлов, введенных при улучшении стратегии, будет иметь лучшие ожидаемые полезности для некоторых состояний и, следовательно, будет улучшен контроллер в целом.

Алгоритм 23.3. Итерация по стратегиям для задачи POMDP $\mathcal{P}$ при фиксированном количестве итераций k_max и количестве итераций оценки стратегии eval_max. Алгоритм итеративно чередует оценку стратегии (алгоритм 23.2) и ее улучшение. Метод сокращения реализован в алгоритме 23.4

```
struct ControllerPolicyIteration
    k_max # количество итераций
    eval_max # количество оценочных итераций
end

function solve(M::ControllerPolicyIteration, 𝒫::POMDP)
    𝒜, 𝒪, k_max, eval_max = 𝒫.𝒜, 𝒫.𝒪, M.k_max, M.eval_max
    X = [1]
    ψ = Dict((x, a) => 1.0 / length(𝒜) for x in X, a in 𝒜)
    η = Dict((x, a, o, x′) => 1.0 for x in X, a in 𝒜, o in 𝒪, x′ in X)
    π = ControllerPolicy(𝒫, X, ψ, η)
    for i in 1:k_max
        prevX = copy(π.X)
        U = iterative_policy_evaluation(π, eval_max)
        policy_improvement!(π, U, prevX)
        prune!(π, U, prevX)
    end
    return π
end

function policy_improvement!(π::ControllerPolicy, U, prevX)
    𝒮, 𝒜, 𝒪 = π.𝒫.𝒮, π.𝒫.𝒜, π.𝒫.𝒪
    X, ψ, η = π.X, π.ψ, π.η
    repeatX𝒪 = fill(X, length(𝒪))
    assign𝒜X′ = vec(collect(product(𝒜, repeatX𝒪...)))
    for ax′ in assign𝒜X′
        x, a = maximum(X) + 1, ax′[1]
        push!(X, x)
        successor(o) = ax′[findfirst(isequal(o), 𝒪) + 1]
        U′(o,s′) = U[successor(o), s′]
        for s in 𝒮
            U[x, s] = lookahead(π.𝒫, U′, s, a)
        end
        for a′ in 𝒜
            ψ[x, a′] = a′ == a ? 1.0 : 0.0
            for (o, x′) in product(𝒪, prevX)
```

```
                η[x, a′, o, x′] = x′ == successor(o) ? 1.0 : 0.0
            end
        end
    end
    for (x, a, o, x′) in product(X, 𝒜, 𝒪, X)
        if !haskey(η, (x, a, o, x′))
            η[x, a, o, x′] = 0.0
        end
    end
end
```

Алгоритм 23.4. Этап сокращения в методе итерации по стратегиям. Он уменьшает количество узлов в текущей стратегии `π`, используя полезности `U`, вычисленные при оценке стратегии, и список предыдущих узлов `prevX`. На первом шаге помечаются все ранее существовавшие узлы, для которых нашлись соответствующие доминирующие узлы. На втором этапе помечаются все вновь добавленные узлы, идентичные предыдущим узлам. На третьем шаге помечаются новые узлы, над которыми доминируют другие новые узлы. Наконец, все помеченные узлы сокращаются

```
function prune!(π::ControllerPolicy, U, prevX)
    𝒮, 𝒜, 𝒪, X, ψ, η = π.𝒫.𝒮, π.𝒫.𝒜, π.𝒫.𝒪, π.X, π.ψ, π.η
    newX, removeX = setdiff(X, prevX), []
    # сокращение ранее существовавших узлов
    dominated(x,x′) = all(U[x,s] ≤ U[x′,s] for s in 𝒮)
    for (x,x′) in product(prevX, newX)
        if x′ ∉ removeX && dominated(x, x′)
            for s in 𝒮
                U[x,s] = U[x′,s]
            end
            for a in 𝒜
                ψ[x,a] = ψ[x′,a]
                for (o,x″) in product(𝒪, X)
                    η[x,a,o,x″] = η[x′,a,o,x″]
                end
            end
            push!(removeX, x′)
        end
    end
    # сокращение совпадающих узлов
    identical_action(x,x′) = all(ψ[x,a] ≈ ψ[x′,a] for a in 𝒜)
    identical_successor(x,x′) = all(η[x,a,o,x″] ≈ η[x′,a,o,x″]
```

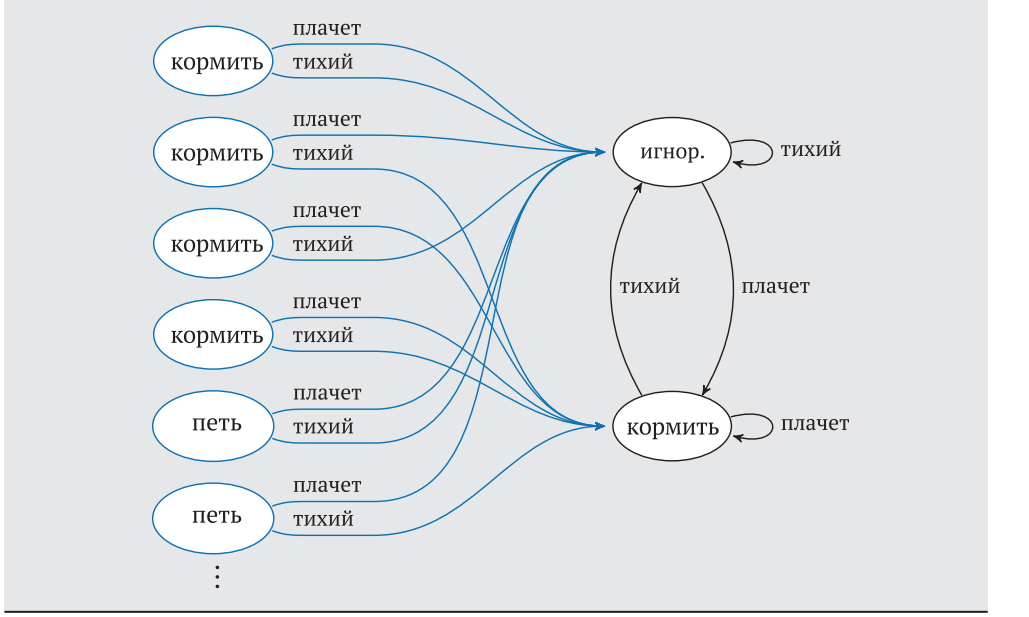
```
            for a in 𝒜, o in 𝒪, x″ in X)
        identical(x,x′) = identical_action(x,x′) && identical_successor(x,x′)
        for (x,x′) in product(prevX, newX)
            if x′ ∉ removeX && identical(x,x′)
                push!(removeX, x′)
            end
        end
        # сокращение новых узлов, над которыми доминируют другие узлы
        for (x,x′) in product(X, newX)
            if x′ ∉ removeX && dominated(x′,x) && x ≠ x′
                push!(removeX, x′)
            end
        end
        # обновление контроллера
        π.X = setdiff(X, removeX)
        π.ψ = Dict(k => v for (k,v) in ψ if k[1] ∉ removeX)
        π.η = Dict(k => v for (k,v) in η if k[1] ∉ removeX)
    end
```

Пример 23.3. Иллюстрация этапа улучшения в рамках итерации по стратегиям для задачи о плачущем ребенке и представление стратегии в виде контроллера

Применим этап улучшения стратегии к контроллеру задачи о плачущем ребенке из примера 23.1. Действия $\mathcal{A}$ = {кормить, петь, игнорировать}, а наблюдения $\mathcal{O}$ = {плачет, тихий}. Улучшение стратегии в итоге дает нам $|\mathcal{A}||X^{(1)}|^{|\mathcal{O}|} = 3 \times 2^2 = 12$ новых узлов. Новая стратегия контроллера имеет узлы $\{x^1, \ldots, x^{14}\}$ и распределения, показанные в таблице ниже:

Узел	Действие	Последователи (для всех a ниже)
x^3	ψ(кормить$\|x^3$) = 1	$\eta(x^1\|x^3, a,$ плачет) = $\eta(x^1\|x^3, a,$ плачет) = 1
x^4	ψ(кормить$\|x^4$) = 1	$\eta(x^1\|x^4, a,$ плачет) = $\eta(x^2\|x^4, a,$ плачет) = 1
x^5	ψ(кормить$\|x^5$) = 1	$\eta(x^2\|x^5, a,$ плачет) = $\eta(x^1\|x^5, a,$ плачет) = 1
x^6	ψ(кормить$\|x^6$) = 1	$\eta(x^2\|x^6, a,$ плачет) = $\eta(x^2\|x^6, a,$ плачет) = 1
x^7	ψ(кормить$\|x^7$) = 1	$\eta(x^1\|x^7, a,$ плачет) = $\eta(x^1\|x^7, a,$ плачет) = 1
x^8	ψ(кормить$\|x^8$) = 1	$\eta(x^1\|x^8, a,$ плачет) = $\eta(x^2\|x^8, a,$ плачет) = 1
⋮	⋮	⋮

У нас получился следующий контроллер с новыми узлами синего цвета и двумя исходными узлами черного цвета:



Многие из узлов, добавленных на этапе улучшения стратегии, как правило, не приводят к реальному улучшению. Для устранения ненужных узлов после оценки стратегии выполняется *сокращение* (pruning). Эта операция не ухудшает оптимальную функцию полезности контроллера. Сокращение способствует ограничению экспоненциального роста узлов, связанного с этапом улучшения. В некоторых случаях обрезка может привести к образованию петель, что дает компактные контроллеры.

Операция сокращения удаляет любые новые узлы, которые идентичны существующим узлам. Также подлежат удалению любые новые узлы, над которыми *доминируют* другие узлы. Над узлом x доминирует другой узел x', когда

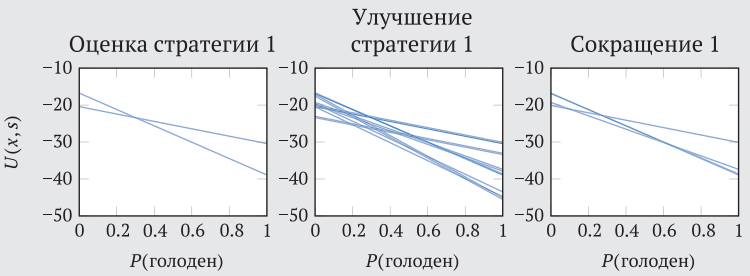
$$U(x, s) \leq U(x', s) \text{ для всех } s. \tag{23.4}$$

Существующие узлы также могут быть удалены. Всякий раз, когда новый узел доминирует над существующим узлом, алгоритм удаляет существующий узел из контроллера, а любые переходы к удаленному узлу перенаправляются на доминирующий узел. Пример 23.4 демонстрирует оценку, улучшение и сокращение контроллера для задачи о плачущем ребенке.

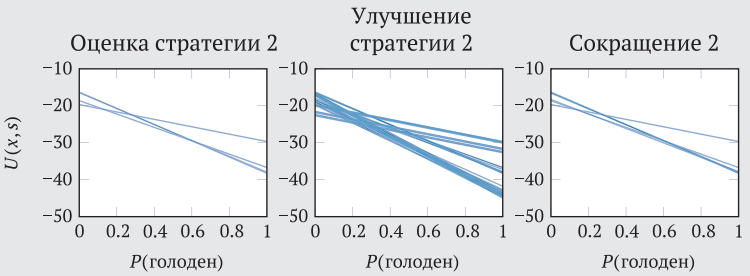
Пример 23.4. Итерация по стратегиям, иллюстрирующая этапы оценки, улучшения и сокращения при решении задачи о плачущем ребенке с представлением стратегии в виде контроллера

Вспомним пример 23.3. Здесь мы показываем первую итерацию по стратегиям с использованием того же начального контроллера. Он состоит из двух

основных этапов: оценки стратегии (слева) и улучшения стратегии (в центре), а также необязательного этапа сокращения (справа).



Вторая итерация следует той же схеме:



Полезность значительно улучшилась после второй итерации – до почти оптимальных значений. Мы видим, что шаг сокращения удаляет более «слабые» и повторяющиеся узлы из предыдущих итераций, а также новые узлы текущей итерации.

23.3. Нелинейное программирование

Задачу улучшения стратегии можно записать в виде общей формулы *нелинейного программирования* (алгоритм 23.5), которая содержит одновременную оптимизацию ψ и η во всех узлах[4]. Эта формула позволяет применять решатели общего назначения. Метод нелинейного программирования выполняет прямой поиск в пространстве контроллеров, чтобы максимизировать полезность исходного убеждения, удовлетворяя при этом уравнению ожидания Беллмана (23.1). Нет чередования между этапами оценки и улучшения стратегии, а количество узлов контроллера остается неизменным.

[4] C. Amato, D. S. Bernstein, S. Zilberstein, *Optimizing Fixed-Size Stochastic Controllers for POMDPs and Decentralized POMDPs*, Autonomous Agents and Multi-Agent Systems, vol. 21, no. 3, pp. 293–320, 2010.

Обозначим за x^1 начальный узел, соответствующий заданному начальному убеждению b. Тогда задача оптимизации заключается в следующем:

$$\begin{aligned}
&\underset{U,\psi,\eta}{\text{максимизировать}} \sum_s b(s)U(x^1,s),\\
&\text{при условии что}\\
&U(x,s) = \sum_a \psi(a|x)\left(R(s,a) + \gamma\sum_{s'} T(s'|s,a)\sum_o O(o|a,s')\sum_{x'}\eta(x'|x,a,o)U(x',s')\right)\\
&\qquad\qquad \text{для всех } x,s;\\
&\psi(a|x) \geq 0 \text{ для всех } x,a;\\
&\sum_a \psi(a|x) = 1 \text{ для всех } x;\\
&\eta(x'|x,a,o) \geq 0 \text{ для всех } x,a,o,x';\\
&\sum_{x'} \eta(x'|x,a,o) = 1 \text{ для всех } x,a,o.
\end{aligned} \tag{23.5}$$

Эта задача может быть записана в виде *линейной программы с квадратичными ограничениями* (quadratically constrained linear program, QCLP), которую можно эффективно решить с помощью специального решателя[5]. Пример 23.5 демонстрирует данный подход.

Алгоритм 23.5. Применение нелинейного программирования для вычисления оптимальной стратегии контроллера фиксированного размера для задачи POMDP $\mathcal{P}$ начиная с первоначального убеждения `b`. Размер контроллера конечного автомата определяется количеством узлов ℓ

```
struct NonlinearProgramming
    b # начальное убеждение
    ℓ # количество узлов
end

function tensorform(𝒫::POMDP)
    𝒮, 𝒜, 𝒪, R, T, O = 𝒫.𝒮, 𝒫.𝒜, 𝒫.𝒪, 𝒫.R, 𝒫.T, 𝒫.O
    𝒮′ = eachindex(𝒮)
    𝒜′ = eachindex(𝒜)
    𝒪′ = eachindex(𝒪)
    R′ = [R(s,a) for s in 𝒮, a in 𝒜]
    T′ = [T(s,a,s′) for s in 𝒮, a in 𝒜, s′ in 𝒮]
    O′ = [O(a,s′,o) for a in 𝒜, s′ in 𝒮, o in 𝒪]
    return 𝒮′, 𝒜′, 𝒪′, R′, T′, O′
end
```

---

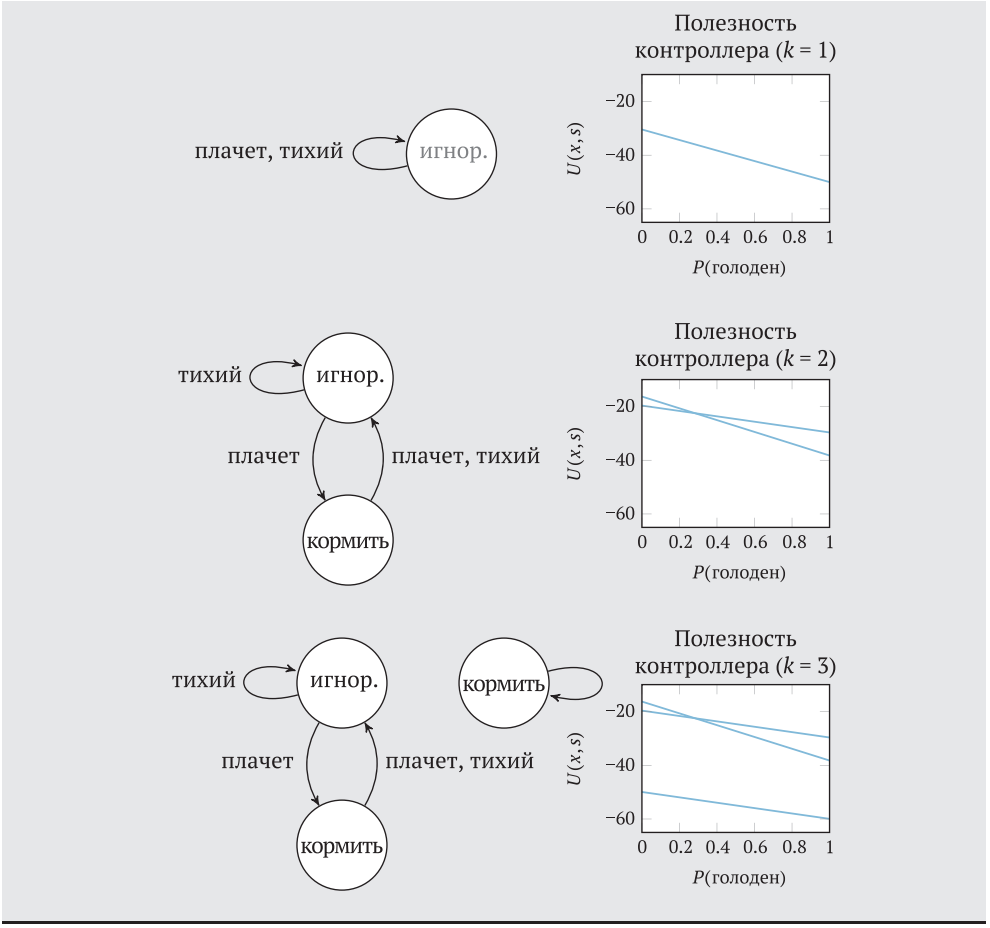
[5] Решение общей задачи QCLP является NP-трудным, но специальные решатели предлагают эффективные приближения.

```
function solve(M::NonlinearProgramming, 𝒫::POMDP)
    x1, X = 1, collect(1:M.ℓ)
    𝒫, γ, b = 𝒫, 𝒫.γ, M.b
    𝒮, 𝒜, 𝒪, R, T, O = tensorform(𝒫)
    model = Model(Ipopt.Optimizer)
    @variable(model, U[X,𝒮])
    @variable(model, ψ[X,𝒜] ≥ 0)
    @variable(model, η[X,𝒜,𝒪,X] ≥ 0)
    @objective(model, Max, b⋅U[x1,:])
    @NLconstraint(model, [x=X,s=𝒮],
        U[x,s] == (sum(ψ[x,a]*(R[s,a] + γ*sum(T[s,a,s′]*sum(O[a,s′,o]
        *sum(η[x,a,o,x′]*U[x′,s′] for x′ in X)
        for o in 𝒪) for s′ in 𝒮)) for a in 𝒜)))
    @constraint(model, [x=X], sum(ψ[x,:]) == 1)
    @constraint(model, [x=X,a=𝒜,o=𝒪], sum(η[x,a,o,:]) == 1)
    optimize!(model)
    ψ′, η′ = value.(ψ), value.(η)
    return ControllerPolicy(𝒫, X,
        Dict((x, 𝒫.𝒜[a]) => ψ′[x, a] for x in X, a in 𝒜),
        Dict((x, 𝒫.𝒜[a], 𝒫.𝒪[o], x′) => η′[x, a, o, x′]
            for x in X, a in 𝒜, o in 𝒪, x′ in X))
end
```

Пример 23.5. Алгоритм нелинейного программирования для контроллеров с фиксированным размером *k*, равным 1, 2 и 3. Каждый горизонтальный ряд рисунков показывает стратегию и соответствующие ей полезности (альфа-векторы) слева и справа соответственно. Стохастические контроллеры показаны в виде кругов с наиболее вероятным действием в середине. Исходящие ребра показывают выбор последующих узлов с учетом наблюдения. Серый цвет надписи означает более низкую вероятность

На рисунках ниже изображены оптимальные контроллеры фиксированного размера, вычисленные с помощью нелинейного программирования для задачи о плачущем ребенке с $b_0 = [0.5, 0.5]$. Верхним является узел x_1.

При $k = 1$ оптимальная стратегия состоит в том, чтобы просто всегда игнорировать ребенка. При $k = 2$ оптимальной стратегией является игнорирование до тех пор, пока не будет наблюдаться плач, после чего лучшим действием будет покормить ребенка, а затем вернуться к игнорированию. Эта стратегия близка к оптимальной для задачи о плачущем ребенке с бесконечным горизонтом POMDP. При $k = 3$ оптимальная стратегия практически не меняется по сравнению с $k = 2$.



23.4. Градиентный подъем

Стратегия контроллера фиксированного размера поддается итеративному улучшению с помощью градиентного подъема, описанного в приложении A.11[6]. Хотя градиент сложно вычислить, это открывает возможность оптимизации контроллера с помощью широкого спектра методов оптимизации на основе градиента. В алгоритме 23.6 реализована оптимизация контроллера методом градиентного подъема с использованием алгоритма 23.7.

[6] N. Meuleau, K.-E. Kim, L. P. Kaelbling, A. R. Cassandra, *Solving POMDPs by Searching the Space of Finite Policies*, in Conference on Uncertainty in Artificial Intelligence (UAI), 1999.

Алгоритм 23.6. Оптимизация контроллера методом градиентного подъема для задачи POMDP $\mathcal{P}$ при начальном убеждении `b`. Сам контроллер имеет фиксированный размер ℓ узлов. Он улучшается в течение `k_max` итераций, следуя градиенту контроллера с шагом `a`, чтобы максимально улучшить полезность исходного убеждения

```
struct ControllerGradient
    b # initial belief
    ℓ # number of nodes
    α # gradient step
    k_max # maximum iterations
end

function solve(M::ControllerGradient, 𝒫::POMDP)
    𝒜, 𝒪, ℓ, k_max = 𝒫.𝒜, 𝒫.𝒪, M.ℓ, M.k_max
    X = collect(1:ℓ)
    ψ = Dict((x, a) => rand() for x in X, a in 𝒜)
    η = Dict((x, a, o, x′) => rand() for x in X, a in 𝒜, o in 𝒪, x′ in X)
    π = ControllerPolicy(𝒫, X, ψ, η)
    for i in 1:k_max
        improve!(π, M, 𝒫)
    end
    return π
end

function improve!(π::ControllerPolicy, M::ControllerGradient, 𝒫::POMDP)
    𝒮, 𝒜, 𝒪, X, x1, ψ, η = 𝒫.𝒮, 𝒫.𝒜, 𝒫.𝒪, π.X, 1, π.ψ, π.η
    n, m, z, b, ℓ, α = length(𝒮), length(𝒜), length(𝒪), M.b, M.ℓ, M.α
    ∂U′∂ψ, ∂U′∂η = gradient(π, M, 𝒫)
    UIndex(x, s) = (s - 1) * ℓ + (x - 1) + 1
    E(U, x1, b) = sum(b[s]*U[UIndex(x1,s)] for s in 1:n)
    ψ′ = Dict((x, a) => 0.0 for x in X, a in 𝒜)
    η′ = Dict((x, a, o, x′) => 0.0 for x in X, a in 𝒜, o in 𝒪, x′ in X)
    for x in X
        ψ′x = [ψ[x, a] + α * E(∂U′∂ψ(x, a), x1, b) for a in 𝒜]
        ψ′x = project_to_simplex(ψ′x)
        for (aIndex, a) in enumerate(𝒜)
            ψ′[x, a] = ψ′x[aIndex]
        end
        for (a, o) in product(𝒜, 𝒪)
            η′x = [(η[x, a, o, x′] +
                    α * E(∂U′∂η(x, a, o, x′), x1, b)) for x′ in X]
            η′x = project_to_simplex(η′x)
            for (x′Index, x′) in enumerate(X)
                    η′[x, a, o, x′] = η′x[x′Index]
            end
        end
    end
    π.ψ, π.η = ψ′, η′
end
```

```
function project_to_simplex(y)
    u = sort(copy(y), rev=true)
    i = maximum([j for j in eachindex(u)
                 if u[j] + (1 - sum(u[1:j])) / j > 0.0])
    δ = (1 - sum(u[j] for j = 1:i)) / i
    return [max(y[j] + δ, 0.0) for j in eachindex(u)]
end
```

Алгоритм 23.7. Функция `gradient` для оптимизации стратегии контроллера методом градиентного подъема. Она строит градиенты полезности `U` относительно градиентов стратегии `∂U′∂ψ` и `∂U′∂η`

```
function gradient(π::ControllerPolicy, M::ControllerGradient, 𝒫::POMDP)
    𝒮, 𝒜, 𝒪, T, O, R, γ = 𝒫.𝒮, 𝒫.𝒜, 𝒫.𝒪, 𝒫.T, 𝒫.O, 𝒫.R, 𝒫.γ
    X, x1, ψ, η = π.X, 1, π.ψ, π.η
    n, m, z = length(𝒮), length(𝒜), length(𝒪)
    X𝒮 = vec(collect(product(X, 𝒮)))
    T′ = [sum(ψ[x, a] * T(s, a, s′) * sum(O(a, s′, o) * η[x, a, o, x′]
          for o in 𝒪) for a in 𝒜) for (x, s) in X𝒮, (x′, s′) in X𝒮]
    R′ = [sum(ψ[x, a] * R(s, a) for a in 𝒜) for (x, s) in X𝒮]
    Z = 1.0I(length(X𝒮)) - γ * T′
    invZ = inv(Z)
    ∂Z∂ψ(hx, ha) = [x == hx ? (-γ * T(s, ha, s′)
                    * sum(O(ha, s′, o) * η[hx, ha, o, x′]
                          for o in 𝒪)) : 0.0
                    for (x, s) in X𝒮, (x′, s′) in X𝒮]
    ∂Z∂η(hx, ha, ho, hx′) = [x == hx && x′ == hx′ ? (-γ * ψ[hx, ha]
                    * T(s, ha, s′) * O(ha, s′, ho)) : 0.0
                  for (x, s) in X𝒮, (x′, s′) in X𝒮]
    ∂R′∂ψ(hx, ha) = [x == hx ? R(s, ha) : 0.0 for (x, s) in X𝒮]
    ∂R′∂η(hx, ha, ho, hx′) = [0.0 for (x, s) in X𝒮]
    ∂U′∂ψ(hx, ha) = invZ * (∂R′∂ψ(hx, ha) - ∂Z∂ψ(hx, ha) * invZ * R′)
    ∂U′∂η(hx, ha, ho, hx′) = invZ * (∂R′∂η(hx, ha, ho, hx′)
                                - ∂Z∂η(hx, ha, ho, hx′) * invZ * R′)
    return ∂U′∂ψ, ∂U′∂η
end
```

Вспомним явное описание нелинейной задачи из раздела 23.3. Если даны начальное убеждение b и произвольный начальный узел контроллера x^1, мы стремимся максимизировать следующую сумму:

$$\sum_s b(s)U(x^1, s), \tag{23.6}$$

где $U(x, s)$ – полезность, определяемая уравнением оптимальности Беллмана для всех x и s:

$$U(x,s) = \sum_a \psi(a|x)\left(R(s,a) + \gamma \sum_{s'} T(s'|s,a) \sum_o O(o|a,s') \sum_{x'} \eta(x'|x,a,o)U(x',s')\right). \tag{23.7}$$

Кроме того, ψ и η должны представлять собой подходящие распределения вероятностей. Чтобы применить градиентный подъем, удобнее переписать эту задачу с использованием линейной алгебры.

Определим функцию перехода с помощью контроллера, который имеет пространство состояний $X \times \mathcal{S}$. Для любой стратегии контроллера фиксированного размера, параметризованной вектором $\boldsymbol{\theta} = (\psi, \eta)$, матрица перехода $\mathbf{T}_{\boldsymbol{\theta}} \in \mathbb{R}^{|X\times\mathcal{S}|\times|X\times\mathcal{S}|}$ будет следующей:

$$\mathbf{T}_{\boldsymbol{\theta}}((x,s),(x',s')) = \sum_a \psi(x,a)T(s,a,s')\sum_o O(a,s',o)\,\eta(x,a,o,x'). \tag{23.8}$$

Вознаграждение за параметрическую стратегию представлено как вектор $\mathbf{r}_{\boldsymbol{\theta}} \in \mathbb{R}^{|X\times\mathcal{S}|}$:

$$\mathbf{r}_{\boldsymbol{\theta}}((x,s)) = \sum_a \psi(x,a)R(s,a). \tag{23.10}$$

Уравнение ожидания Беллмана для полезности $\mathbf{u}_{\boldsymbol{\theta}} \in \mathbb{R}^{|X\times\mathcal{S}|}$ имеет вид:

$$\mathbf{u}_{\boldsymbol{\theta}} = \mathbf{r}_{\boldsymbol{\theta}} + \gamma\mathbf{T}_{\boldsymbol{\theta}}\mathbf{u}_{\boldsymbol{\theta}}. \tag{23.10}$$

Будем использовать начальный вектор узел–доверие $\boldsymbol{\beta}_{\boldsymbol{\theta}} \in \mathbb{R}^{|X\times\mathcal{S}|}$, где $\boldsymbol{\beta}_{xs} = b(s)$, если $x = x^1$, и $\boldsymbol{\beta}_{xs} = 0$ в противном случае. Вектор полезности $\mathbf{u}_{\boldsymbol{\theta}} \in \mathbb{R}^{|X\times\mathcal{S}|}$ также определяется по узлам X и состояниям $\mathcal{S}$ для любой из этих стратегий параметрического контроллера фиксированного размера $\boldsymbol{\theta} = (\psi, \eta)$. Теперь нам нужно максимизировать следующее произведение:

$$\boldsymbol{\beta}^\top\mathbf{u}_{\boldsymbol{\theta}}. \tag{23.11}$$

Сначала перепишем уравнение (23.10):

$$\mathbf{u}_{\boldsymbol{\theta}} = \mathbf{r}_{\boldsymbol{\theta}} + \gamma\mathbf{T}_{\boldsymbol{\theta}}\mathbf{u}_{\boldsymbol{\theta}}; \tag{23.12}$$

$$(\mathbf{I} - \gamma\mathbf{T}_{\boldsymbol{\theta}})\mathbf{u}_{\boldsymbol{\theta}} = \mathbf{r}_{\boldsymbol{\theta}}; \tag{23.13}$$

$$\mathbf{u}_{\boldsymbol{\theta}} = (\mathbf{I} - \gamma\mathbf{T}_{\boldsymbol{\theta}})^{-1}\mathbf{r}_{\boldsymbol{\theta}}; \tag{23.14}$$

$$\mathbf{u}_{\boldsymbol{\theta}} = \mathbf{Z}^{-1}\mathbf{r}_{\boldsymbol{\theta}}, \tag{23.15}$$

обозначив $\mathbf{Z} = \mathbf{I} - \gamma\mathbf{T}_{\boldsymbol{\theta}}$ для удобства. Чтобы выполнить градиентный подъем, нам нужно знать частные производные уравнения (23.15) по параметрам стратегии:

$$\frac{\partial\mathbf{u}_{\boldsymbol{\theta}}}{\partial\boldsymbol{\theta}} = \frac{\partial\mathbf{Z}^{-1}}{\partial\boldsymbol{\theta}}\mathbf{r}_{\boldsymbol{\theta}} + \mathbf{Z}^{-1}\frac{\partial\mathbf{r}_{\boldsymbol{\theta}}}{\partial\boldsymbol{\theta}} \tag{23.16}$$

$$= -\mathbf{Z}^{-1}\frac{\partial\mathbf{Z}}{\partial\boldsymbol{\theta}}\mathbf{Z}^{-1}\mathbf{r}_{\boldsymbol{\theta}} + \mathbf{Z}^{-1}\frac{\partial\mathbf{r}_{\boldsymbol{\theta}}}{\partial\boldsymbol{\theta}} \tag{23.17}$$

$$= \mathbf{Z}^{-1}\left(\frac{\partial\mathbf{r}_{\boldsymbol{\theta}}}{\partial\boldsymbol{\theta}} - \frac{\partial\mathbf{Z}}{\partial\boldsymbol{\theta}}\mathbf{Z}^{-1}\mathbf{r}_{\boldsymbol{\theta}}\right), \tag{23.18}$$

где $\partial\boldsymbol{\theta}$ для удобства относится как к $\partial\psi(\hat{x}, \hat{a})$, так и к $\partial\eta(\hat{x}, \hat{a}, \hat{o}, \hat{x}')$.

Вычисление частных производных от $\mathbf{Z}$ и $\mathbf{r}_{\boldsymbol{\theta}}$ дает нам четыре уравнения:

$$\frac{\partial \mathbf{r}_{\boldsymbol{\theta}}((x,s))}{\partial \psi(\hat{x},\hat{a})} = \begin{cases} R(s,a), & \text{если } x = \hat{x} \\ 0 & \text{в иных случаях} \end{cases}; \tag{23.19}$$

$$\frac{\partial \mathbf{r}_{\boldsymbol{\theta}}((x,s))}{\partial \eta(\hat{x},\hat{a},\hat{o},\hat{x}')} = 0; \tag{23.20}$$

$$\frac{\partial \mathbf{Z}((x,s),(x',s'))}{\partial \psi(\hat{x},\hat{a})} = \begin{cases} -\gamma T(s,\hat{a},s')\sum_o O(\hat{a},s',o)\eta(\hat{x},\hat{a},o,x'), & \text{если } x = \hat{x} \\ 0 & \text{в иных случаях} \end{cases}; \tag{23.21}$$

$$\frac{\partial \mathbf{Z}((x,s),(x',s'))}{\eta(\hat{x},\hat{a},\hat{o},\hat{x}')} = \begin{cases} -\gamma \psi(\hat{x},\hat{a})T(s,\hat{a},s')O(\hat{a},s',\hat{o})\eta(\hat{x},\hat{a},\hat{o},x'), & \text{если } x = \hat{x} \text{ и } x' = \hat{x}' \\ 0 & \text{в иных случаях} \end{cases}. \tag{23.22}$$

Наконец, подставим эти четыре градиента в уравнение (23.18) следующим образом:

$$\frac{\partial \mathbf{u}_{\boldsymbol{\theta}}}{\partial \psi(\hat{x},\hat{a})} = \mathbf{Z}^{-1}\left(\frac{\partial \mathbf{r}_{\boldsymbol{\theta}}}{\partial \psi(\hat{x},\hat{a})} - \frac{\partial \mathbf{Z}}{\partial \psi(\hat{x},\hat{a})}\mathbf{Z}^{-1}\mathbf{r}_{\boldsymbol{\theta}}\right); \tag{23.23}$$

$$\frac{\partial \mathbf{u}_{\boldsymbol{\theta}}}{\partial \eta(\hat{x},\hat{a},\hat{o},\hat{x}')} = \mathbf{Z}^{-1}\left(\frac{\partial \mathbf{r}_{\boldsymbol{\theta}}}{\partial \eta(\hat{x},\hat{a},\hat{o},\hat{x}')} - \frac{\partial \mathbf{Z}}{\partial \eta(\hat{x},\hat{a},\hat{o},\hat{x}')}\mathbf{Z}^{-1}\mathbf{r}_{\boldsymbol{\theta}}\right). \tag{23.24}$$

Теперь мы можем вернуться к исходной целевой функции в уравнении (23.11). Градиентный подъем контроллера начинается с фиксированного количества узлов в X и произвольных параметров стратегии ψ и η. На итерации k контроллер обновляет эти параметры следующим образом:

$$\psi^{k+1}(x,a) = \psi^{k}(x,a) + \mathrm{a}\mathbf{b}^{\top}\frac{\partial \mathbf{u}_{\boldsymbol{\theta}^k}}{\partial \psi^{k}(\hat{x},\hat{a})}; \tag{23.25}$$

$$\eta^{k+1}(x,a,o,x') = \eta^{k}(x,a,o,x') + \mathrm{a}\mathbf{b}^{\top}\frac{\partial \mathbf{u}_{\boldsymbol{\theta}^k}}{\partial \eta^{k}(\hat{x},\hat{a},\hat{o},\hat{x}')} \tag{23.26}$$

с шагом градиента $\alpha > 0$. После этого обновления ψ^{k+1} и η^{k+1} перестают быть действительными распределениями. Чтобы сделать их действительными, спроецируем их на симплекс вероятностей. Один из способов получения проекции вектора $\mathbf{y}$ на симплекс вероятностей состоит в том, чтобы найти ближайшее распределение в соответствии с L_2-нормой:

$$\begin{aligned} &\underset{\mathbf{b}}{\text{минимизировать}}\ \frac{1}{2}\|\mathbf{y}-\mathbf{b}\|_2^2, \\ &\text{при условии что}\quad \mathbf{b} \geq 0; \\ &\qquad\qquad\quad\ \mathbf{1}^{\top}\mathbf{b} = 1. \end{aligned} \tag{23.27}$$

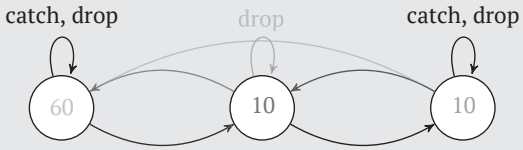
Эта оптимизация поддается точному вычислению с помощью простого алгоритма, включенного в алгоритм 23.6[7]. Пример 23.6 демонстрирует процесс обновления контроллера.

Целевая функция оптимизации в уравнении (23.6) не обязательно выпуклая[8]. Следовательно, в некоторых случаях обычный градиентный подъем может сходиться к локальному оптимуму. Для плавной и быстрой сходимости часто применяют алгоритмы адаптивного градиента.

Пример 23.6. Демонстрация работы алгоритма градиентной оптимизации для контроллеров с фиксированным размером $\ell = 3$. Здесь показано, что стратегия уточняется в ходе итераций алгоритма. Агент постепенно определяет, как лучше всего использовать свое фиксированное количество узлов, что приводит к разумной и интерпретируемой стратегии, обеспечивающей сходимость. Стохастические контроллеры показаны в виде кругов с наиболее вероятным действием в середине. Исходящие ребра показывают выбор последующих узлов с учетом наблюдения. Стохастичность в действиях узла и его преемниках показана оттенками серого (более темные – вероятность выше, более светлые – вероятность ниже)

Рассмотрим задачу о бросках мяча (приложение F.9) с равномерным начальным убеждением b_1. На рисунках ниже показано изменение полезности стратегии по мере прохождения итераций градиентного подъема применительно к задаче о бросках с $k = 3$ узлами. Левый узел равен x_1.

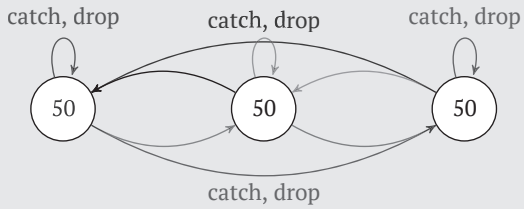
На итерации 1 стратегия, по существу, случайна как при выборе действия, так и при выборе преемника:



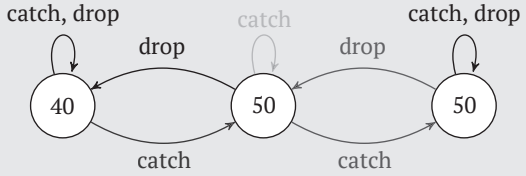
На итерации 50 агент определил разумное расстояние для броска мяча (50), но все еще не использовал свои три узла, чтобы запомнить что-либо полезное:

[7] J. Duchi, S. Shalev-Shwartz, Y. Singer, T. Chandra, *Efficient Projections onto the $\ell 1$-Ball for Learning in High Dimensions*, in International Conference on Machine Learning (ICML), 2008.

[8] Эта цель отличается от полезности $U(x, b) = \sum_s b(s)U(x, s)$, которая гарантированно является кусочно-линейной и выпуклой по отношению к убеждению b, как показано в разделе 20.3.



На итерации 500 стратегия образовала разумный план, использующий фиксированный набор из трех узлов с запоминанием:



Сначала агент пытается бросить мяч на расстояние 40. Если ребенок ловит мяч, расстояние увеличивается до 50. Агент использует последний узел, чтобы запомнить, сколько раз ребенок поймал мяч (до двух раз), чтобы выбрать расстояние.

23.5. Заключение

- Контроллеры – это способы представления стратегии, которые не нуждаются в исследовании или сохранении убеждений.
- Контроллеры состоят из узлов, функции выбора действия и функции выбора преемника.
- Узлы и граф контроллера абстрактны; однако их можно интерпретировать как наборы счетно-бесконечных достижимых убеждений.
- Функцию полезности для узла контроллера можно рассматривать как альфа-вектор.
- Итерация по стратегиям выполняет чередование между оценкой стратегии путем вычисления полезности для каждого узла и улучшением стратегии путем добавления новых узлов.
- Операция сокращения помогает избежать экспоненциального роста количества узлов с каждым шагом улучшения.
- Нелинейное программирование позволяет рассматривать поиск оптимального контроллера фиксированного размера как общую задачу оптимизации и применять готовые решатели и методы.
- При оптимизации контроллера градиентный подъем происходит в пространстве стратегий и направлен на непосредственное улучшение функции

полезности за счет явных шагов в направлении градиента, вычисляемых на основе метода для POMDP.

23.6. Упражнения

Упражнение 23.1. Перечислите все преимущества представления стратегии в виде контроллера по сравнению с древовидным условным планом и представлениями на основе убеждений.

Решение. В отличие от древовидных условных планов, контроллеры позволяют представлять стратегии, которые выполняются неограниченное время и, как правило, не склонны к экспоненциальному росту по мере удаления горизонта.

По сравнению с представлениями, основанными на убеждениях, в случае масштабных задач количество параметров в представлении контроллера, как правило, намного меньше, чем количество альфа-векторов убеждений. Кроме того, контроллеры намного легче оптимизировать для фиксированного объема памяти.

Во время выполнения контроллеры никогда не столкнутся с делением на ноль, как это происходит в стратегиях, представленных через убеждения. Методы, основанные на убеждениях, требуют постоянного наличия корректных убеждений. Фильтр дискретного состояния из уравнения (19.7) будет делить на ноль, если встретит невозможное наблюдение. Это может произойти, например, когда зашумленный датчик возвращает наблюдение, которое модели $T(s, a, s')$ и $O(o \mid a, s')$ не способны обработать.

Упражнение 23.2. Итеративное улучшение стратегии контроллера добавляет только узлы с детерминированными функциями выбора действий и распределениями последователей. Означает ли это, что результирующий контроллер обязательно неоптимален?

Решение. Итерация по стратегиям контроллера в пределе гарантированно сойдется к оптимальной стратегии. Однако метод не может найти более компактные представления оптимальных стратегий контроллера, для которых могут потребоваться стохастические узлы.

Упражнение 23.3. Докажите, что сокращение узла в процессе итерации по стратегиям не влияет на полезность.

Решение. Пусть x' будет новым узлом из некоторой итерации i, а x будет предыдущим узлом из итерации $i - 1$.

Из построения $\eta(x', a, o, x)$ следует, что все новые узлы x' имеют только предшественника x из предыдущей итерации. Таким образом, в каждом состоянии s функция полезности $U^{(i)}(x', s)$ суммирует только полезности $U^{(i-1)}(x, s')$ согласно уравнению (23.1). Это означает, что другие полезности на итерации i, в том числе самозамкнутого цикла при x, не влияют на полезность $U^{(i)}(x', s)$. Поскольку начальный узел выбирается по уравнению (23.3), мы должны убе-

диться, что полезность с сокращаемым узлом и без него при всех убеждениях будет одинаковой. Узел сокращают в одном из двух случаев:

1) узел x' получает более высокую полезность во всех состояниях, чем его сокращенный предшественник x. $U^{(i)}(x, s) \leq U^{(i-1)}(x', s)$ для всех s. На этапе сокращения x заменяется на x', включая U, ψ и η. Следовательно, значение U не уменьшилось ни в одном состоянии s;
2) узел x идентичен существующему предыдущему узлу x'. Заметим, что это означает переход $\eta(x, a, o, x') = \eta(x', a, o, x')$. Это означает, что полезность идентична, за исключением того, что полезность x уменьшается на дисконт γ; другими словами, $\gamma U^{(i)}(x, s) = U^{(i-1)}(x, s)$ по уравнению (23.1). Сокращение x не влияет на окончательную полезность.

Упражнение 23.4. Разработайте алгоритм, который использует нелинейное программирование для нахождения минимального контроллера фиксированного размера, необходимого для достижения оптимальности большого контроллера фиксированного размера ℓ. В данном случае можете предположить, что нелинейный оптимизатор возвращает оптимальную стратегию.

Решение. Идея состоит в том, чтобы создать внешний цикл, который постепенно увеличивает фиксированный размер меньшего контроллера, зная полезность большого контроллера. Сначала вычислим полезность большого контроллера фиксированного размера $U^* = \sum_s b_1(s)U(x_1, s)$ в начальном узле x_1 и при начальном убеждении b_1. Затем мы создаем цикл, который увеличивает размер ℓ контроллера. На каждом шаге мы оцениваем стратегию и вычисляем полезность U^ℓ. Согласно предположению, возвращаемый контроллер производит глобально оптимальную полезность для фиксированного размера ℓ. После достижения полезности U^ℓ и выполнения условия $U^\ell = U^*$ останавливаем итерации и возвращаем стратегию.

Упражнение 23.5. Проанализируйте выполнение шага градиента в алгоритме оптимизации контроллера методом градиентного подъема. Предположим, что $|\mathcal{S}|$ больше, чем $|\mathcal{A}|$ и $|\mathcal{O}|$. Какая часть шага градиента является наиболее затратной с вычислительной точки зрения? Как ее можно улучшить?

Решение. Вычисление обращенной матрицы $\mathbf{Z}^{-1} = (\mathbf{I} - \gamma \mathbf{T}_\theta)$ является наиболее ресурсоемкой частью шага градиента, как и всего алгоритма градиентного подъема. Матрица $\mathbf{Z}$ имеет размер $|X \times \mathcal{S}|$. Исключение по Гауссу–Жордану требует $O(|X \times \mathcal{S}|^3)$ операций, хотя 3 в показателе степени можно уменьшить до 2.3728639 с помощью современного алгоритма обращения матриц[9]. Создание временной матрицы T_θ также требует $O(|X \times \mathcal{S}|^2|\mathcal{A} \times \mathcal{O}|)$ операций для вычисления обращенной матрицы. Все остальные циклы и другие временные массивы требуют гораздо меньше операций. Алгоритм можно улучшить, используя методы приближенного обращения матриц.

[9] F. L. Gall, *Powers of Tensors and Fast Matrix Multiplication*, in International Symposium on Symbolic and Algebraic Computation (ISSAC), 2014.

Часть V

Многоагентные системы

Ранее мы рассматривали принятие решений с точки зрения одного агента. Теперь мы перенесем уже знакомые вам понятия на задачи, связанные с несколькими агентами. В *многоагентных системах* мы можем моделировать поведение других агентов как потенциальных союзников или противников, которые соответствующим образом адаптируются с течением времени. Такие задачи изначально намного сложнее, поскольку включают в себя взаимодействие агентов и рассуждения конкретного агента о других агентах, которые, в свою очередь, рассуждают об этом агенте, и так далее. Мы начнем с рассмотрения многоагентного логического вывода в играх и обрисуем в общих чертах, как вычислять равновесия из простых взаимодействий. Затем обсудим разработку алгоритмов для нескольких взаимодействующих агентов, опираясь на алгоритмы обучения, которые отдают предпочтение рациональной адаптации, а не сходимости к равновесию. Введение неопределенности состояния многократно увеличивает сложность задачи, и в этой части особое внимание уделяется возникающим проблемам. Последняя глава посвящена различным моделям и алгоритмам для сотрудничающих агентов, которые стремятся работать вместе для достижения общей цели.

24 Логический вывод в многоагентных системах

До сих пор мы рассматривали принятие рациональных решений одним агентом. Эти модели естественным образом расширяются на нескольких агентов. По мере взаимодействия агентов возникают новые проблемы; агенты могут помогать друг другу или действовать в своих интересах. Логический вывод в многоагентных системах является предметом *теории игр*[1]. В этой главе мы опираемся на представленные ранее понятия, распространяя их на многоагентные сценарии. Мы обсудим основные подходы теории игр к вычислению стратегий принятия решений и поиску многоагентного равновесия.

24.1. Простые игры

Простая игра (алгоритм 24.1) является фундаментальной моделью логического вывода с участием нескольких агентов[2]. Каждый агент $i \in \mathcal{I}$ выбирает действие a^i, чтобы максимизировать собственное накопительное вознаграждение r^i. Пространство совместных действий $\mathcal{A} = \mathcal{A}^1 \times \cdots \times \mathcal{A}^k$ состоит из всех возможных комбинаций действий $\mathcal{A}^i$, доступных каждому агенту. Действия, выбранные одновременно агентами, могут быть объединены для формирования *совместного действия* (joint action) $\mathbf{a} = (a^1, \ldots, a^k)$ в этом пространстве совместных действий[3]. *Функция совместного вознаграждения* (joint reward) $\mathbf{R}(\mathbf{a}) = (R^1(\mathbf{a}), \ldots, R^k(\mathbf{a}))$ возвращает вознаграждение, производимое совместным действием $\mathbf{a}$. *Совместное вознаграждение* записывают как $\mathbf{r} = (r^1, \ldots, r^k)$. Простые игры не содержат состояния или функции перехода. В примере 24.1 представлена простая игра.

---

[1] Теория игр – обширная область. В перечень общепризнанных вводных учебников входят D. Fudenberg, J. Tirole, *Game Theory*. MIT Press, 1991. R. B. Myerson, *Game Theory: Analysis of Conflict*. Harvard University Press, 1997. Y. Shoham, K. Leyton-Brown, *Multiagent Systems: Algorithmic, Game Theoretic, and Logical Foundations*. Cambridge University Press, 2009.

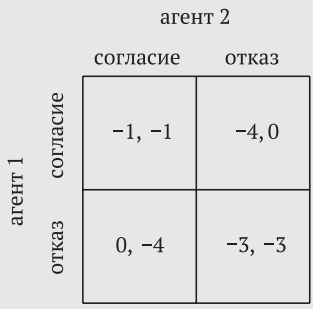
[2] Простые игры включают в себя игры нормальной формы (также называемые играми стандартной формы, или матричными играми), повторяющиеся игры с конечным горизонтом и повторяющиеся игры с дисконтом с бесконечным горизонтом. Y. Shoham, K. Leyton-Brown, *Multiagent Systems: Algorithmic, Game Theoretic, and Logical Foundations*. Cambridge University Press, 2009.

[3] Совместное действие также называется *профилем действия* (action profile).

Алгоритм 24.1. Структура данных для простой игры

```
struct SimpleGame
    γ # коэффициент дисконта
    ℐ # агенты
    𝒜 # пространство совместных действий
    R # совместная функция вознаграждения
end
```

Пример 24.1. Простая игра, известная как дилемма заключенного. Более подробное описание приведено в приложении F.10



Дилемма заключенного – это игра с двумя агентами и двумя действиями, в которой участвуют два заключенных, находящихся под следствием. Они могут сговориться и хранить молчание о своем общем преступлении или действовать исключительно в своих интересах и обвинить в своем преступлении другого. Если они придерживаются сговора и отрицают преступление, то оба отбывают заключение длительностью 1 год. Если агент i придерживается договоренности, а другой агент его предает и обвиняет в преступлении, то агент i получает 4 года, а другой выходит на свободу. Если оба агента отказываются от сговора и обвиняют друг друга, то каждый из них получает по 3 года.

Простые игры с двумя агентами можно представить в виде таблицы. Строки представляют действия для агента 1. Столбцы представляют действия для агента 2. Награды для агентов 1 и 2 размещены в каждой ячейке.

Совместная стратегия (joint policy) π определяет распределение вероятностей совместных действий агентов. Совместные стратегии можно разбить на стратегии отдельных агентов. Вероятность того, что агент i выберет действие a, равна $\pi^i(a)$. В теории игр детерминированная стратегия называется *чистой стратегией* (pure strategy), а стохастическая – *смешанной стратегией* (mixed strategy). Полезность совместной стратегии π с точки зрения агента i равна

$$U^i(\pi) \geq \sum_{\mathbf{a} \in \mathcal{A}} R^i(\mathbf{a}) \prod_{j \neq \mathcal{I}} \pi^j(a^j). \tag{24.1}$$

Алгоритм 24.2 реализует подпрограммы для представления стратегий и вычисления их полезности.

Алгоритм 24.2. Стратегия, относящаяся к определенному агенту, представлена словарем, который сопоставляет действия с вероятностями. Существуют разные способы построения стратегии. Один из способов – передать каталог словаря, и в этом случае вероятности нормализуются. Другой способ – передать генератор, который создает этот словарь. Мы также можем построить стратегию, передав действие, и в этом случае она присваивает вероятность 1 этому действию. Если у нас есть отдельная стратегия `πi`, мы можем вызвать `πi(ai)` для вычисления вероятности, которую стратегия связывает с действием `ai`. Если мы вызовем `πi()`, она вернет случайное действие в соответствии с этой стратегией. Для построения пространства совместных действий из $\mathcal{A}$ применяется функция `joint(`$\mathcal{A}$`)`. Для вычисления полезности, связанной со следованием совместной стратегии `π` в игре $\mathcal{P}$ с точки зрения агента `i`, предназначена функция `utility(`$\mathcal{P}$`, π, i)`

```
struct SimpleGamePolicy
    p # словарь перевода действий в вероятности

    function SimpleGamePolicy(p::Base.Generator)
        return SimpleGamePolicy(Dict(p))
    end

function SimpleGamePolicy(p::Dict)
    vs = collect(values(p))
    vs ./= sum(vs)
    return new(Dict(k => v for (k,v) in zip(keys(p), vs)))
end

    SimpleGamePolicy(ai) = new(Dict(ai => 1.0))
end

(πi::SimpleGamePolicy)(ai) = get(πi.p, ai, 0.0)

function (πi::SimpleGamePolicy)()
    D = SetCategorical(collect(keys(πi.p)), collect(values(πi.p)))
    return rand(D)
end

joint(X) = vec(collect(product(X...)))

joint(π, πi, i) = [i == j ? πi : πj for (j, πj) in enumerate(π)]

function utility(𝒫::SimpleGame, π, i)
    𝒜, R = 𝒫.𝒜, 𝒫.R
```

```
    p(a) = prod(πj(aj) for (πj, aj) in zip(π, a))
    return sum(R(a)[i]*p(a) for a in joint(𝒜))
end
```

Игра с нулевой суммой – это разновидность простой игры, в которой сумма вознаграждений всех агентов равна нулю. Здесь любой выигрыш одного агента приводит к проигрышу для других агентов. Игра с нулевой суммой с двумя агентами $\mathcal{I} = \{1, 2\}$ имеет противоположные функции вознаграждения $R^1(\mathbf{a}) = -R^2(\mathbf{a})$. Обычно при поиске оптимальной стратегии в игре с нулевой суммой применяют алгоритмы, специально разработанные для данной структуры вознаграждения. Пример 24.2 описывает одну из таких игр.

Пример 24.2. Хорошо известная игра «камень–ножницы–бумага» является примером игры с нулевой суммой. Дополнительное описание приведено в приложении F.11

«Камень–ножницы–бумага» – игра с нулевой суммой для двух агентов. Каждый агент выбирает камень, ножницы или бумагу. Камень побеждает ножницы, бумага побеждает камень, а ножницы побеждают бумагу с вознаграждением 1 для победителя и –1 для проигравшего. Если агенты выбирают одно и то же действие, оба получают нулевое вознаграждение. В общем виде повторяющиеся игры с двумя агентами можно представить в виде последовательности матриц выигрышей, как показано ниже:

$t = 1$

агент 1 \ агент 2	камень	бумага	ножницы
камень	0, 0	−1, 1	1, −1
бумага	1, −1	0, 0	−1, 1
ножницы	−1, 1	1, −1	0, 0

$t = 2$

агент 1 \ агент 2	камень	бумага	ножницы
камень	0, 0	−1, 1	1, −1
бумага	1, −1	0, 0	−1, 1
ножницы	−1, 1	1, −1	0, 0

…

…

24.2. Модели откликов

Прежде чем рассмотреть различные подходы к поиску совместной стратегии, мы начнем с моделирования *отклика* (response) одного агента i при заданных фиксированных стратегиях для других агентов. Мы будем использовать обозначение $-i$ как сокращенную запись для $(1, \ldots, i-1, i+1, \ldots, k)$. Соответственно,

совместное действие записывается как $\mathbf{a} = (a^i, \mathbf{a}^{-i})$, совместное вознаграждение – как $\mathbf{R}(a^i, \mathbf{a}^{-i})$, а совместная стратегия – как $\pi = (\pi^i, \pi^{-i})$. В этом разделе обсуждаются различные подходы к вычислению отклика на известную стратегию π^{-i}.

24.2.1. Наилучший отклик

Наилучшим откликом (или *наилучшим ответом*, best response) агента i на стратегию других агентов $\boldsymbol{\pi}^{-i}$ является стратегия π^i, удовлетворяющая выражению

$$U^i(\pi^i, \boldsymbol{\pi}^{-1}) \geq U^i(\pi^{i\prime}, \boldsymbol{\pi}^{-1}) \tag{24.2}$$

для всех других стратегий $\pi^{i\prime} \neq \pi^i$. Другими словами, наилучший отклик агента – это стратегия, при которой у него нет стимула менять свою стратегию при заданном наборе стратегий других агентов. Наилучших откликов может быть несколько.

Если мы ограничимся детерминированными стратегиями, *детерминированный лучший отклик* на стратегию противника π^{-i} вычислить несложно. Мы просто перебираем все действия агента i и возвращаем то из них, которое максимизирует полезность следующим образом:

$$\arg\max_{a^i \in \mathcal{A}^i} U^i(a^i, \boldsymbol{\pi}^{-i}). \tag{24.3}$$

Реализация этого подхода представлена в алгоритме 24.3.

Алгоритм 24.3. В случае простой игры $\mathcal{P}$ мы можем вычислить детерминированный лучший отклик для агента `i`, при условии что другие агенты используют стратегии из `π`

```
function best_response(𝒫::SimpleGame, π, i)
    U(ai) = utility(𝒫, joint(π, SimpleGamePolicy(ai), i), i)
    ai = argmax(U, 𝒫.𝒜[i])
    return SimpleGamePolicy(ai)
end
```

24.2.2. Отклик softmax

При моделировании того, как агент i будет выбирать свое действие, можно использовать *отклик softmax* (softmax response)[4]. Как было отмечено в разделе 6.7,

[4] Такой тип модели иногда называют *логит-откликом* (logit response), или *квантильным откликом* (quantal response). Мы представили аналогичные модели softmax ранее в этой книге в контексте стратегий направленного исследования для обучения с подкреплением (раздел 15.4).

люди редко руководствуются рациональными соображениями оптимальной ожидаемой полезности. Принцип, лежащий в основе модели отклика softmax, заключается в том, что агенты (как правило, люди) с большей вероятностью допускают ошибки при оптимизации, когда эти ошибки обходятся дешевле. При заданном *параметре точности* (precision parameter) $\lambda \geq 0$ эта модель выбирает действие a^i в соответствии с уравнением

$$\pi^i(a^i) \propto \exp(\lambda U^i(a^i, \boldsymbol{\pi}^{-1})). \tag{24.4}$$

Когда $\lambda \to 0$, агент нечувствителен к различиям в полезности и равномерно выбирает действия случайным образом. Когда $\lambda \to \infty$, стратегия сходится к детерминированному наилучшему отклику. Мы можем рассматривать λ как параметр, который можно узнать из данных, используя, например, оценку максимального правдоподобия (раздел 4.1). Этот подход, основанный на обучении, направлен на то, чтобы предсказывать, а не предписывать поведение, хотя наличие прогнозирующей модели других человеческих агентов может быть полезно и при построении системы, предписывающей оптимальное поведение. Алгоритм 24.4 содержит реализацию модели отклика softmax.

Алгоритм 24.4. Для простой игры $\mathcal{P}$ и конкретного агента `i` мы можем вычислить стратегию отклика softmax `πi`, при условии что другие агенты используют стратегии из `π`. Для вычислений необходимо указать параметр точности `λ`

```
function softmax_response(𝒫::SimpleGame, π, i, λ)
    𝒜i = 𝒫.𝒜[i]
    U(ai) = utility(𝒫, joint(π, SimpleGamePolicy(ai), i), i)
    return SimpleGamePolicy(ai => exp(λ*U(ai)) for ai in 𝒜i)
end
```

24.3. Равновесие доминирующей стратегии

В некоторых играх у агента есть *доминирующая стратегия* (dominant strategy), которая является наилучшей реакцией на все другие возможные стратегии агентов. Например, в дилемме заключенного (пример 24.1) наилучший отклик агента 1 – отказаться от участия в сговоре независимо от стратегии агента 2, что делает отказ доминирующей стратегией для агента 1. Совместная стратегия, при которой все агенты используют доминирующие стратегии, называется *равновесием с доминирующей стратегией* (dominant strategy equilibrium). В дилемме заключенного совместная стратегия, при которой оба агента отказываются от участия в сговоре, является равновесием с доминирующей стратегией[5]. Во многих играх нет равновесия с доминирующей стратегией. Например,

[5] Интересно, что когда оба агента выполняют жадные действия в отношении своей собственной функции полезности, это приводит к худшему результату для них обо-

в игре «камень–ножницы–бумага» (пример 24.2) наилучший отклик агента 1 зависит от стратегии агента 2.

24.4. Равновесие Нэша

В отличие от концепции равновесия с доминирующей стратегией, равновесие Нэша[6] всегда существует для игр с конечным пространством действия[7]. *Равновесие Нэша* (Nash equilibrium) – это совместная стратегия π, при которой все агенты следуют наилучшему отклику. Другими словами, равновесие Нэша – это совместная стратегия, при которой ни у одного агента нет стимула в одностороннем порядке менять свою стратегию.

В одной игре может существовать несколько равновесий Нэша (упражнение 24.2). Иногда равновесие Нэша может включать детерминированную стратегию, но это не всегда так (см. пример 24.3). Вычисление равновесия Нэша является PPAD-полным – это класс задач, который отличается от NP-полного (приложение С.2), но также не имеет известного алгоритма с полиномиальным временем выполнения[8].

Пример 24.3. Детерминированные и стохастические равновесия Нэша

Предположим, что мы хотим найти равновесие Нэша для дилеммы заключенного из примера 24.1. Если оба агента всегда отказываются соблюдать сговор, то оба получают вознаграждение –3. Одиночный отказ любого агента приведет к вознаграждению –4 для этого агента; следовательно, у агента нет никакого стимула менять стратегию. Отказ обоих агентов от сговора, таким образом, является равновесием Нэша для дилеммы заключенного.

Предположим, что теперь мы хотим найти равновесие Нэша для сценария «камень–ножницы–бумага» из примера 24.2. Любой детерминированной стратегии одного агента может легко противостоять другой агент. Например, если агент 1 выбирает камень, то лучшим ответом агента 2 будет бумага. Поскольку не существует детерминированного равновесия Нэша для модели «камень–ножницы–бумага», мы знаем, что должно быть равновесие, включающее стохастическую стратегию. Предположим, что каждый агент выбирает одно из действий случайным образом. Это решение дает ожидаемую полезность 0 для обоих агентов:

их. Если бы они оба придерживались сговора, то получили бы по одному году вместо трех лет.

[6] Название дано в честь американского математика Джона Форбса Нэша-младшего (1928–2015), который формализовал и обобщил эту идею. J. Nash, *Non-Cooperative Games*, Annals of Mathematics, pp. 286–295, 1951.

[7] В упражнении 24.1 исследуется случай, когда пространство действий бесконечно.

[8] C. Daskalakis, P. W. Goldberg, C. H. Papadimitriou, *The Complexity of Computing a Nash Equilibrium*, Communications of the ACM, vol. 52, no. 2, pp. 89–97, 2009.

$$\begin{aligned}U^i(\pi) &= 0\frac{1}{3}\frac{1}{3} - 1\frac{1}{3}\frac{1}{3} + 1\frac{1}{3}\frac{1}{3} \\ &\quad + 1\frac{1}{3}\frac{1}{3} + 0\frac{1}{3}\frac{1}{3} - 1\frac{1}{3}\frac{1}{3} \\ &\quad - 1\frac{1}{3}\frac{1}{3} + 1\frac{1}{3}\frac{1}{3} + 0\frac{1}{3}\frac{1}{3} \\ &= 0.\end{aligned}$$

Любое изменение стратегии агента уменьшит его ожидаемый выигрыш, а это означает, что мы нашли равновесие Нэша.

Задача нахождения равновесия Нэша может быть сформулирована как задача оптимизации:

$$\begin{aligned}&\underset{\boldsymbol{\pi}, U}{\text{минимизировать}} \sum_i (U^i - U^i(\boldsymbol{\pi})), \\ &\text{при условии что } U^i \geq U^i(a^i, \boldsymbol{\pi}^{-1}) \text{ для всех } i, a^i; \\ &\qquad\qquad \sum_{a^i} \pi^i(a^i) = 1 \text{ для всех } i; \\ &\qquad\qquad \pi^i(a^i) \geq 0 \text{ для всех } i, a^i.\end{aligned} \tag{24.5}$$

Переменные оптимизации соответствуют параметрам $\boldsymbol{\pi}$ и U. При сходимости цель будет равна 0, а U^i будет соответствовать полезности, связанной со стратегией $\boldsymbol{\pi}$, вычисленной в уравнении (24.1) для каждого агента i. Первое ограничение гарантирует, что ни один агент не добьется большего успеха, если в одностороннем порядке изменит свое действие. Как и цель, это первое ограничение является нелинейным, поскольку содержит произведение параметров в переменной оптимизации $\boldsymbol{\pi}$. Последние два ограничения являются линейными, гарантируя, что $\boldsymbol{\pi}$ представляет допустимый набор распределений вероятностей по действиям. Алгоритм 24.5 реализует эту процедуру оптимизации.

Алгоритм 24.5. Эта нелинейная программа вычисляет равновесие Нэша для простой игры $\mathcal{P}$

```
struct NashEquilibrium end

function tensorform(𝒫::SimpleGame)
    ℐ, 𝒜, R = 𝒫.ℐ, 𝒫.𝒜, 𝒫.R
    ℐ′ = eachindex(ℐ)
    𝒜′ = [eachindex(𝒜[i]) for i in ℐ]
    R′ = [R(a) for a in joint(𝒜)]
    return ℐ′, 𝒜′, R′
end
```

```
function solve(M::NashEquilibrium, 𝒫::SimpleGame)
    ℐ, 𝒜, R = tensorform(𝒫)
    model = Model(Ipopt.Optimizer)
    @variable(model, U[ℐ])
    @variable(model, π[i=ℐ, 𝒜[i]] ≥ 0)
    @NLobjective(model, Min,
        sum(U[i] - sum(prod(π[j,a[j]] for j in ℐ) * R[y][i]
            for (y,a) in enumerate(joint(𝒜))) for i in ℐ))
    @NLconstraint(model, [i=ℐ, ai=𝒜[i]],
        U[i] ≥ sum(
            prod(j==i ? (a[j]==ai ? 1.0 : 0.0) : π[j,a[j]] for j in ℐ)
            * R[y][i] for (y,a) in enumerate(joint(𝒜))))
    @constraint(model, [i=ℐ], sum(π[i,ai] for ai in 𝒜[i]) == 1)
    optimize!(model)
    πi′(i) = SimpleGamePolicy(𝒫.𝒜[i][ai] => value(π[i,ai]) for ai in 𝒜[i])
    return [πi′(i) for i in ℐ]
end
```

24.5. Согласованное равновесие

Согласованное равновесие (correlated equilibrium) обобщает концепцию равновесия Нэша, ослабляя предположение о том, что агенты действуют независимо. Совместное действие в этом случае извлекается из полного совместного распределения. *Согласованная совместная стратегия* (correlated joint policy) $\pi(\mathbf{a})$ представляет собой единое распределение по совместным действиям всех агентов. Следовательно, действия различных агентов могут быть согласованы, предотвращая разделение стратегий на отдельные стратегии $\pi^i(a^i)$. Алгоритм 24.6 демонстрирует представление такой стратегии.

Алгоритм 24.6. Согласованная совместная стратегия представлена словарем, который сопоставляет совместные действия с вероятностями. Если `π` является совместной коррелированной стратегией, вычисление `π(a)` вернет вероятность, связанную с совместным действием `a`

```
mutable struct JointCorrelatedPolicy
    p # словарь, отображающий совместные действия в вероятности
    JointCorrelatedPolicy(p::Base.Generator) = new(Dict(p))
end

(π::JointCorrelatedPolicy)(a) = get(π.p, a, 0.0)

function (π::JointCorrelatedPolicy)()
    D = SetCategorical(collect(keys(π.p)), collect(values(π.p)))
    return rand(D)
end
```

Согласованное равновесие – это согласованная совместная стратегия, при которой ни один агент i не может увеличить свою ожидаемую полезность, отклонившись от своего текущего действия a^i в сторону другого действия $a^{i'}$:

$$\sum_{\mathbf{a}^{-i}} R^i(a^i, \mathbf{a}^{-i})\pi(a^i, \mathbf{a}^{-i}) \geq \sum_{\mathbf{a}^{-i}} R^i(a^{i'}, \mathbf{a}^{-i})\pi(a^i, \mathbf{a}^{-i}). \tag{24.6}$$

Это определение проиллюстрировано в примере 24.4.

Пример 24.4. Вычисление согласованного равновесия в игре «камень–ножницы–бумага»

Вернемся к сценарию игры «камень–ножницы–бумага» из примера 24.2. В примере 24.3 мы обнаружили, что в равновесии Нэша для этой игры оба агента выбирают свои действия случайным образом. В согласованном равновесии мы используем согласованную совместную стратегию $\pi(\mathbf{a})$. Это означает, что нам нужно найти распределение по сочетаниям (камень, камень), (камень, бумага), (камень, ножницы), (бумага, камень) и т. д. Возможны девять совместных действий.

Для начала рассмотрим совместную стратегию, в которой агент 1 выбирает камень, а агент 2 выбирает ножницы. Полезности таковы:

$$U^1(\pi) = 0\frac{0}{9} - 1\frac{0}{9} + 1\frac{9}{9} + 1\frac{0}{9} + \cdots = 1;$$

$$U^2(\pi) = 0\frac{0}{9} + 1\frac{0}{9} - 1\frac{9}{9} - 1\frac{0}{9} + \cdots = -1.$$

Если бы агент 2 переключился на бумагу, он получил бы полезность, равную 1. Следовательно, это не согласованное равновесие.

Теперь рассмотрим согласованную совместную стратегию, в которой совместное действие было выбрано равномерно случайным образом с $\pi(\mathbf{a}) = 1/9$:

$$U^1(\pi) = 0\frac{1}{9} - 1\frac{1}{9} + 1\frac{1}{9} + 1\frac{1}{9} + \cdots = 0;$$

$$U^2(\pi) = 0\frac{1}{9} + 1\frac{1}{9} - 1\frac{1}{9} - 1\frac{1}{9} + \cdots = 0.$$

Любое отклонение от этой стратегии приводит к тому, что один агент приобретает полезность, а другой теряет. Следовательно, мы нашли согласованное равновесие для игры «камень–ножницы–бумага».

Каждое равновесие Нэша является согласованным равновесием, потому что мы всегда можем сформировать совместную стратегию из независимых стратегий:

$$\pi(\mathbf{a}) = \prod_{i=1}^{k} \pi^i(a^i). \tag{24.7}$$

Если отдельные стратегии удовлетворяют уравнению (24.2), то совместная стратегия будет удовлетворять уравнению (24.6). Однако не все согласованные равновесия являются равновесиями по Нэшу.

Согласованное равновесие можно вычислить с помощью линейного программирования (алгоритм 24.7):

$$\begin{aligned}
&\text{максимизировать}_{\boldsymbol{\pi}} \sum_i \sum_{\mathbf{a}} R^i(\mathbf{a})\pi(\mathbf{a}),\\
&\text{при условии что } \sum_{\mathbf{a}^{-1}} R^i(a^i, \mathbf{a}^{-i})\pi(a^i, \mathbf{a}^{-i}) \geq \sum_{\mathbf{a}^{-1}} R^i(a^{i\prime}, \mathbf{a}^{-i})\pi(a^i, \mathbf{a}^{-i})\\
&\qquad\qquad \text{для всех } i, a^i, a^{i\prime};\\
&\sum_{\mathbf{a}} \pi(\mathbf{a}) = 1;\\
&\pi(\mathbf{a}) \geq 0 \text{ для всех } \mathbf{a}.
\end{aligned} \tag{24.8}$$

Алгоритм 24.7. Согласованное равновесие – это более общее понятие оптимальности для простой игры $\mathcal{P}$, чем равновесие Нэша. Его можно вычислить с помощью линейной программы. Результирующие стратегии коррелированы, т. е. агенты стохастически выбирают свои совместные действия

```
struct CorrelatedEquilibrium end

function solve(M::CorrelatedEquilibrium, 𝒫::SimpleGame)
    ℐ, 𝒜, R = 𝒫.ℐ, 𝒫.𝒜, 𝒫.R
    model = Model(Ipopt.Optimizer)
    @variable(model, π[joint(𝒜)] ≥ 0)
    @objective(model, Max, sum(sum(π[a]*R(a) for a in joint(𝒜))))
    @constraint(model, [i=ℐ, ai=𝒜[i], ai′=𝒜[i]],
        sum(R(a)[i]*π[a] for a in joint(𝒜) if a[i]==ai)
        ≥ sum(R(joint(a,ai′,i))[i]*π[a] for a in joint(𝒜) if a[i]==ai))
    @constraint(model, sum(π) == 1)
    optimize!(model)
    return JointCorrelatedPolicy(a => value(π[a]) for a in joint(𝒜))
end
```

Хотя линейные программы могут быть выполнены за полиномиальное время, размер пространства совместных действий растет экспоненциально с увеличением числа агентов. Ограничения обеспечивают согласованное равновесие. Для выбора между различными действительными согласованными равновесиями можно использовать разные целевые функции. В табл. 24.1 представлено несколько распространенных вариантов.

Таблица 24.1. Альтернативные целевые функции для уравнения (24.8), которые можно использовать при выборе различных согласованных равновесий. Таблица адаптирована из доклада А. Гринвальда и К. Холла «Согласованное Q-обучение» на Международной конференции по машинному обучению (ICML), 2003 г.

Название	Описание	Целевая функция
Утилитарная	Максимизирует чистую полезность	максимизировать$_\pi \sum_i \sum_{\mathbf{a}} R^i(\mathbf{a})\pi(\mathbf{a})$
Эгалитарная	Максимизирует минимум всех полезностей агентов	максимизировать$_\pi$ минимизировать$_i$ $\sum_{\mathbf{a}} R^i(\mathbf{a})\pi(\mathbf{a})$
Плутократическая	Максимизирует максимум всех полезностей агентов	максимизировать$_\pi$ минимизировать$_i$ $\sum_{\mathbf{a}} R^i(\mathbf{a})\pi(\mathbf{a})$
Диктаторская	Максимизирует полезность агента i	максимизировать$_\pi \sum_{\mathbf{a}} R^i(\mathbf{a})\pi(\mathbf{a})$

24.6. Итеративный поиск лучшего отклика

Поскольку поиск равновесия Нэша может быть дорогостоящим в вычислительном отношении, альтернативный подход заключается в итеративном применении наилучших откликов в серии повторяющихся игр. В алгоритме *итеративного наилучшего отклика* (iterated best response, алгоритм 24.8) мы случайным образом переключаемся между агентами, по очереди находя стратегию наилучшего отклика для каждого агента. Этот процесс может сходиться к равновесию Нэша, но гарантия сходимости существует только для определенных классов игр[9]. Во многих задачах часто встречаются циклы.

Алгоритм 24.8. Итеративный поиск наилучшего отклика включает циклический перебор агентов и применение их наилучшего отклика к другим агентам. Алгоритм начинается с некоторой начальной стратегии и останавливается после `k_max` итераций. Для удобства у нас есть конструктор, который принимает в качестве входных данных простую игру и создает исходную стратегию, согласно которой каждый агент выбирает действия случайным образом. Та же самая функция `solve` будет повторно использована в следующей главе для более сложных форм игр

```
struct IteratedBestResponse
    k_max # количество итераций
    π # начальная стратегия
end

function IteratedBestResponse(𝒫::SimpleGame, k_max)
    π = [SimpleGamePolicy(ai => 1.0 for ai in 𝒜i) for 𝒜i in 𝒫.𝒜]
```

---

[9] Итерация по наилучшим откликам будет сходиться, например, для класса, известного как *потенциальные игры*, как показано в теореме 19.12 учебника N. Nisan, T. Roughgarden, É. Tardos, V. V. Vazirani, eds., *Algorithmic Game Theory*. Cambridge University Press, 2007.

```
    return IteratedBestResponse(k_max, π)
end

function solve(M::IteratedBestResponse, 𝒫)
    π = M.π
    for k in 1:M.k_max
        π = [best_response(𝒫, π, i) for i in 𝒫.ℐ]
    end
    return π
end
```

24.7. Иерархическая форма модели softmax

Область исследований, известная как *поведенческая теория игр* (behavioral game theory), направлена на моделирование поведения агента, который является человеком. При построении систем принятия решений, которые должны взаимодействовать с людьми, вычисление равновесия Нэша не всегда полезно. Люди часто не придерживаются равновесия Нэша. Кроме того, может быть не ясно, какое равновесие выбрать, если в игре много различных равновесий. Для игр с единственным равновесием человеку бывает трудно найти равновесие Нэша из-за когнитивных ограничений. Даже если агенты-люди находят равновесие Нэша, они могут усомниться в том, что их оппоненты способны сделать то же самое.

В литературе описано множество поведенческих моделей[10], но один из наиболее интересных подходов заключается в объединении итеративного подхода из предыдущего раздела с моделью softmax. Этот иерархический метод softmax (алгоритм 24.9)[11] моделирует глубину рациональности агента уровнем $k \geq 0$. Агент уровня 0 воспроизводит свои действия равномерно случайным образом. Агент уровня 1 предполагает, что другие игроки используют стратегии уровня 0, и выбирает действия в соответствии с откликом softmax с точностью λ. Агент уровня k выбирает действия в соответствии с моделью softmax других игроков, играющих на уровне $k - 1$. На рис. 24.1 показано применение этого подхода для простой игры.

Мы можем обучить параметры k и λ этой поведенческой модели на имеющихся данных. Если у нас есть набор совместных действий, выполняемых разными агентами, мы можем вычислить соответствующую вероятность для заданных k и λ. Затем мы можем использовать алгоритм оптимизации, чтобы попытаться найти значения k и λ, которые максимизируют правдоподобие. Эта оптимизация обычно не имеет аналитического решения, но мы можем ис-

[10] C. F. Camerer, *Behavioral Game Theory: Experiments in Strategic Interaction*. Princeton University Press, 2003.

[11] Этот подход иногда называют *квантильным k-уровнем*, или *k-логит-уровнем*. D. O. Stahl, P. W. Wilson, *Experimental Evidence on Players' Models of Other Players*, Journal of Economic Behavior & Organization, vol. 25, no. 3, pp. 309–327, 1994.

пользовать численные методы[12]. В качестве альтернативы некоторые исследователи предлагают байесовский подход к обучению параметров[13].

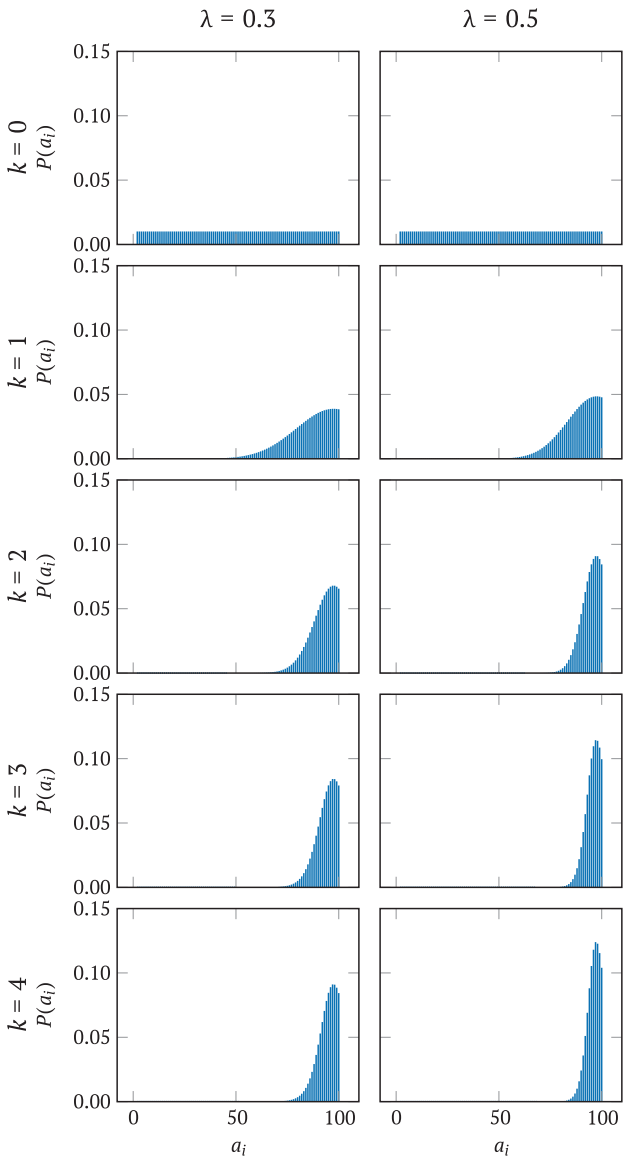


Рис. 24.1. Иерархическая модель softmax применительно к дилемме путешественника (описанной в приложении F.12) для различной глубины рациональности k и параметров точности λ. Люди склонны выбирать действия в диапазоне от 97 до 100 долларов, хотя равновесие Нэша составляет всего 2 доллара

[12] J. R. Wright, K. Leyton-Brown, *Beyond Equilibrium: Predicting Human Behavior in Normal Form Games*, in AAAI Conference on Artificial Intelligence (AAAI), 2010.

[13] J. R. Wright, K. Leyton-Brown, *Behavioral Game Theoretic Models: A Bayesian Framework for Parameter Analysis*, in International Conference on Autonomous Agents and Multiagent Systems (AAMAS), 2012.

Алгоритм 24.9. Иерархическая модель softmax с параметром точности λ и уровнем k. По умолчанию мы начинаем с начальной совместной стратегии, которая присваивает одинаковую вероятность всем индивидуальным действиям

```
struct HierarchicalSoftmax
    λ # precision parameter
    k # level
    π # initial policy
end

function HierarchicalSoftmax(𝒫::SimpleGame, λ, k)
    π = [SimpleGamePolicy(ai => 1.0 for ai in 𝒜i) for 𝒜i in 𝒫.𝒜]
    return HierarchicalSoftmax(λ, k, π)
end

function solve(M::HierarchicalSoftmax, 𝒫)
    π = M.π
    for k in 1:M.k
        π = [softmax_response(𝒫, π, i, M.λ) for i in 𝒫.ℐ]
    end
    return π
end
```

24.8. Фиктивная игра

Альтернативный подход к поиску стратегий для разных агентов состоит в том, чтобы они симулировали игру друг с другом и учились, как лучше всего реагировать. В алгоритме 24.10 представлена реализация цикла симуляции игры. На каждой итерации мы оцениваем различные стратегии для получения совместного действия, а затем это совместное действие агенты используют для обновления своих стратегий. Допускается использование нескольких способов обновления стратегий в ответ на наблюдаемые совместные действия. В этом разделе основное внимание уделяется *фиктивной игре* (fictitious play), в которой агенты используют оценки максимального правдоподобия (как описано в разделе 16.1) стратегий, которым следуют другие агенты. Каждый агент руководствуется наилучшим откликом, предполагая, что и другие агенты тоже следуют этим оценкам[14].

Чтобы вычислить оценку максимального правдоподобия, агент i отслеживает, сколько раз агент j выполняет действие a^j, сохраняя его в таблице $N^i(j, a^j)$. Эти счетчики могут быть инициализированы любым значением, но часто их инициализируют значением 1, чтобы создать начальную равномерную не-

[14] G. W. Brown, *Iterative Solution of Games by Fictitious Play*, Activity Analysis of Production and Allocation, vol. 13, no. 1, pp. 374–376, 1951. J. Robinson, *An Iterative Method of Solving a Game*, Annals of Mathematics, pp. 296–301, 1951.

определенность. Агент i вычисляет свой лучший отклик, предполагая, что каждый агент j следует стохастической стратегии:

$$\pi^j(a^j) \propto N^i(j, a^i). \tag{24.9}$$

Алгоритм 24.10. Симуляция совместной стратегии в простой игре $\mathcal{P}$ для `k_max` итераций. Совместная стратегия `π` – это вектор стратегий, которые можно обновлять по отдельности с помощью вызовов `update!(πi, a)`

```
function simulate(𝒫::SimpleGame, π, k_max)
    for k = 1:k_max
        a = [πi() for πi in π]
        for πi in π
            update!(πi, a)
        end
    end
    return π
end
```

На каждой итерации каждый агент действует в соответствии с наилучшим откликом, предполагая стохастические стратегии на основе значений счетчиков для других агентов. Затем происходит обновление счетчиков, исходя из предпринятых агентами действий. Алгоритм 24.11 реализует эту простую адаптивную процедуру. На рис. 24.2 и 24.3 показано, как стратегия меняется с течением времени при использовании фиктивной игры. Сходимость фиктивной игры к равновесию Нэша не гарантирована[15].

Алгоритм 24.11. Фиктивная игра – это простой алгоритм обучения агента `i` в простой игре $\mathcal{P}$, который хранит подсчеты вариантов действий других агентов `counts` с течением времени и усредняет их, предполагая, что это их стохастическая стратегия. Затем он вычисляет наилучший отклик на эту стратегию и выполняет соответствующее действие по максимизации полезности

```
mutable struct FictitiousPlay
    𝒫 # простая игра
    i # индекс агента
    N # массив словарей подсчетов действий
    πi # текущая стратегия
end

function FictitiousPlay(𝒫::SimpleGame, i)
    N = [Dict(aj => 1 for aj in 𝒫.𝒜[j]) for j in 𝒫.ℐ]
    πi = SimpleGamePolicy(ai => 1.0 for ai in 𝒫.𝒜[i])
    return FictitiousPlay(𝒫, i, N, πi)
```

[15] Краткий справочный материал представлен в U. Berger, *Brown's Original Fictitious Play*, Journal of Economic Theory, vol. 135, no. 1, pp. 572–578, 2007.

```
end

(ni::FictitiousPlay)() = ni.ni()

(ni::FictitiousPlay)(ai) = ni.ni(ai)

function update!(ni::FictitiousPlay, a)
    N, 𝒫, ℐ, i = ni.N, ni.𝒫, ni.𝒫.ℐ, ni.i
    for (j, aj) in enumerate(a)
        N[j][aj] += 1
    end
    p(j) = SimpleGamePolicy(aj => u/sum(values(N[j])) for (aj, u) in N[j])
    π = [p(j) for j in ℐ]
    ni.ni = best_response(𝒫, π, i)
end
```

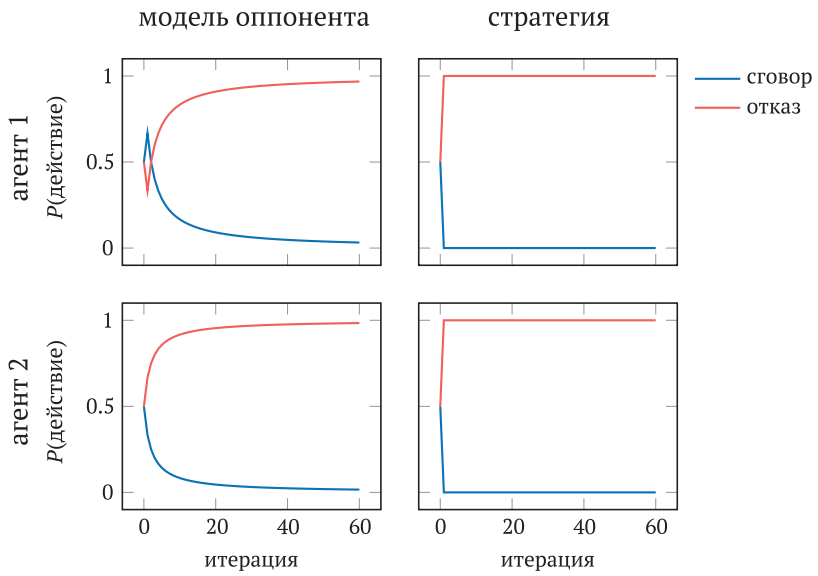


Рис. 24.2. Два вымышленных игровых агента учатся и адаптируются друг к другу в игре «дилемма заключенного». Первый ряд иллюстрирует изученную агентом 1 модель 2 (слева) и стратегию агента 1 (справа) в ходе итерации. Второй ряд следует той же схеме, но для агента 2. Чтобы проиллюстрировать различия в обучающем поведении, начальным счетчикам для модели поведения каждого агента относительно другого агента были присвоены случайные значения в интервале от 1 до 10

Существует множество вариантов фиктивной игры. Один вариант, называемый *сглаженной фиктивной игрой* (smooth fictitious play)[16], выбирает

[16] D. Fudenberg, D. Levine, *Consistency and Cautious Fictitious Play*, Journal of Economic Dynamics and Control, vol. 19, no. 5–7, pp. 1065–1089, 1995.

наилучший ответ, используя ожидаемую полезность плюс функцию сглаживания, такую как энтропия стратегии. Другой вариант называется *рациональным обучением* (rational learning), или *байесовским обучением*. Рациональное обучение расширяет модель фиктивной игры до любых убеждений относительно действий других агентов в форме байесовского априорного распределения. Затем для обновления убеждений с учетом истории совместных действий применяется правило Байеса. Традиционную фиктивную игру можно рассматривать как рациональное обучение с априорным распределением Дирихле (раздел 4.2.2).

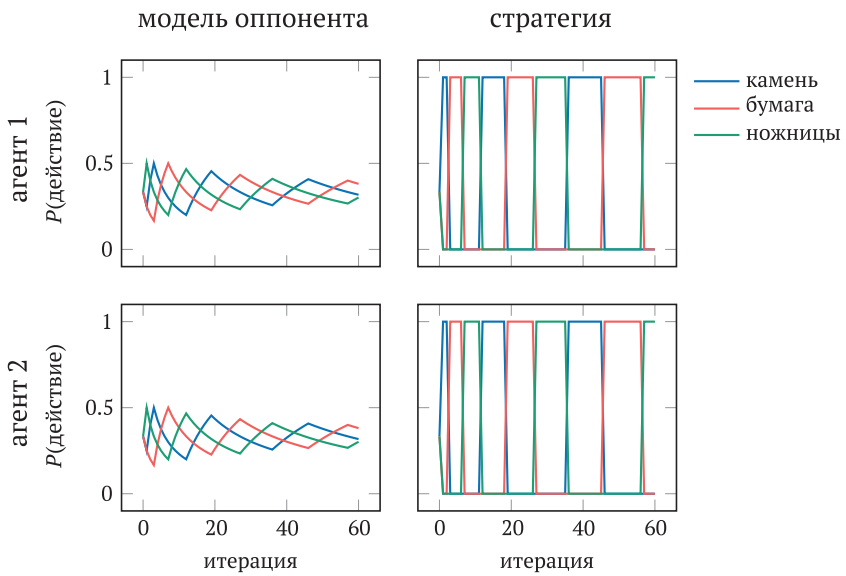


Рис. 24.3. Два вымышленных игровых агента учатся и адаптируются друг к другу в игре «камень–ножницы–бумага». Первый ряд иллюстрирует изученную агентом 1 модель 2 (слева) и стратегию агента 1 (справа) с течением времени. Второй ряд следует той же схеме, но для агента 2. Чтобы проиллюстрировать различия в обучающем поведении, начальным подсчетам для модели поведения каждого агента относительно другого агента были присвоены случайные значения в интервале от 1 до 10. В этой игре с нулевой суммой агенты фиктивной игры приближаются к сходимости своей стохастической стратегии равновесия по Нэшу

24.9. Градиентный подъем

Градиентный подъем (алгоритм 24.12) постепенно корректирует стратегию агента в градиенте в зависимости от полезности. В момент времени t градиент для агента i равен

$$\frac{\partial U^i(\boldsymbol{\pi}_t)}{\partial \pi_t^i(a^i)} = \frac{\partial}{\partial \pi_t^i}\left(\sum_{\mathbf{a}} R^i(\mathbf{a}) \prod_j \pi_t^i(a^i)\right) = \sum_{\mathbf{a}^{-i}} R^i(a^i, \mathbf{a}^{-1}) \prod_{j \neq i} \pi_t^j(a^j). \tag{24.10}$$

Затем мы можем использовать стандартный градиентный подъем

$$\pi_{t+1}^i(a^i) = \pi_t^i(a^i) + \alpha_t^i \frac{\partial U^i(\boldsymbol{\pi}_t)}{\partial \pi_t^i(a^i)} \tag{24.11}$$

со скоростью обучения α_t^i[17]. Стратегию π_{t+1}^i может потребоваться спроецировать обратно на действительное распределение вероятностей, как было показано в разделе 23.4 для стратегий POMDP.

Алгоритм 24.12. Реализация градиентного подъема для агента `i` простой игры $\mathcal{P}$. Алгоритм постепенно обновляет свое распределение вероятностей по действиям после градиентного подъема, стремясь улучшить ожидаемую полезность. Применение функции проекции из алгоритма 23.6 гарантирует, что результирующая стратегия останется действительным распределением вероятностей

```
mutable struct GradientAscent
    𝒫 # простая игра
    i # индекс агента
    t # временной шаг
    πi # текущая стратегия
end

function GradientAscent(𝒫::SimpleGame, i)
    uniform() = SimpleGamePolicy(ai => 1.0 for ai in 𝒫.𝒜[i])
    return GradientAscent(𝒫, i, 1, uniform())
end

(πi::GradientAscent)() = πi.πi()

(πi::GradientAscent)(ai) = πi.πi(ai)

function update!(πi::GradientAscent, a)
    𝒫, ℐ, 𝒜i, i, t = πi.𝒫, πi.𝒫.ℐ, πi.𝒫.𝒜[πi.i], πi.i, πi.t
    jointπ(ai) = [SimpleGamePolicy(j == i ? ai : a[j]) for j in ℐ]
    r = [utility(𝒫, jointπ(ai), i) for ai in 𝒜i]
    π′ = [πi.πi(ai) for ai in 𝒜i]
    π = project_to_simplex(π′ + r / sqrt(t))
```

---

[17] В методе *бесконечно малого градиентного подъема* (infinitesimal gradient ascent) используется скорость обучения, обратная квадратному корню, равная $\alpha_t^i = 1/\sqrt{t}$. Он называется бесконечно малым, поскольку $\alpha_t^i \to 0$ при $t \to \infty$. Мы используем эту скорость обучения в нашей реализации алгоритма. S. Singh, M. Kearns, Y. Mansour, *Nash Convergence of Gradient Dynamics in General-Sum Games*, in Conference on Uncertainty in Artificial Intelligence (UAI), 2000.

```
    ni.t = t + 1
    ni.ni = SimpleGamePolicy(ai => p for (ai, p) in zip(𝒜i, π))
end
```

На практике, однако, агент i знает только свою собственную стратегию π_t^i, а не стратегию других агентов, что затрудняет вычисление градиента. Но при этом агенты наблюдают за совместными действиями. Хотя мы могли бы попытаться оценить их стратегии как достигнутые в фиктивной игре, более простой подход состоит в том, чтобы предположить, что стратегия других агентов представляет собой воспроизведение самого последнего действия[18]. Следовательно, вычисление градиента сводится к

$$\frac{\partial U^i(\boldsymbol{\pi}_t)}{\partial \pi_t^i(a^i)} = R^i(a^i, \mathbf{a}^{-1}). \tag{24.12}$$

На рис. 24.4 показан этот подход для простой игры «камень–ножницы–бумага».

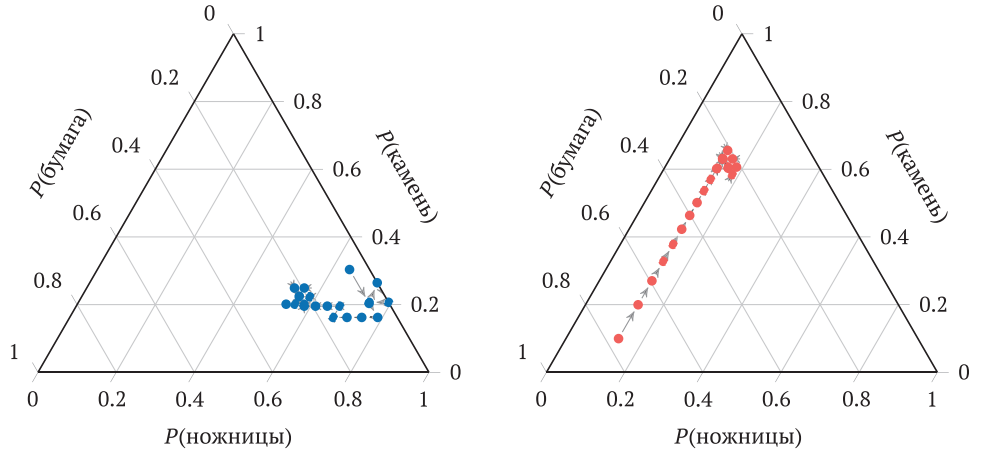


Рис. 24.4. Два агента, действующих по методу градиентного подъема со случайно инициализируемыми стратегиями в игре «камень–ножницы–бумага». Мы используем вариант алгоритма 24.12 со скоростью обучения $0.1/\sqrt{t}$. Здесь показаны 20 обновлений стратегии. Хотя различные симуляции будут сходиться, поскольку размер шага становится равным 0, разные выборки из стохастических стратегий могут сойтись к разным стратегиям

[18] Этот подход используется в *обобщенном бесконечно малом градиентном подъеме* (GIGA). M. Zinkevich, *Online Convex Programming and Generalized Infinitesimal Gradient Ascent*, in International Conference on Machine Learning (ICML), 2003. Вариант правила обновления градиента для стимуляции сходимости предложен в статье M. Bowling, *Convergence and No-Regret in Multiagent Learning*, in Advances in Neural Information Processing Systems (NIPS), 2005.

24.10. Заключение

- В простых играх несколько агентов соревнуются, чтобы максимизировать ожидаемое вознаграждение.
- Оптимальность менее очевидна в многоагентной среде с несколькими возможными концепциями решений для извлечения стратегий через критерий вознаграждения.
- Наилучший отклик агента на фиксированный набор стратегий других агентов – это стратегия, при которой нет стимула ее менять.
- Равновесие Нэша – это совместная стратегия, при которой все агенты следуют наилучшему отклику.
- Согласованное равновесие похоже на равновесие Нэша, за исключением того, что все агенты следуют одному распределению совместного действия, которое допускает корреляцию между агентами.
- Итерация по наилучшему отклику может быстро оптимизировать совместную стратегию путем многократного применения метода наилучших откликов, но гарантия сходимости в общем случае отсутствует.
- Иерархический softmax пытается моделировать агентов с точки зрения их глубины рациональности и точности, что можно узнать из прошлых совместных действий.
- Фиктивная игра – это обучающий алгоритм, который использует модели действий с максимальным правдоподобием для других агентов, чтобы найти наилучшую стратегию реагирования, которая может сойтись к равновесию Нэша.
- Для обучения стратегий можно использовать градиентное восхождение с последующей проекцией на симплекс вероятностей.

24.11. Упражнения

Упражнение 24.1. Приведите пример игры с двумя агентами и бесконечным числом действий, в которой равновесие Нэша не существует.

Решение. Предположим, что пространство действий каждого агента состоит из отрицательных действительных чисел, а их вознаграждение равно их действию. Поскольку не существует наибольшего отрицательного числа, равновесие Нэша не может существовать.

Упражнение 24.2. Приведите пример игры с двумя агентами, двумя действиями и двумя равновесиями Нэша, включающими детерминистские стратегии.

Решение. Рассмотрим пример из литературы[19]. Предположим, у нас есть два самолета, летящих строго встречными курсами, и пилоты каждого самолета

[19] За основу взят пример из книги M. J. Kochenderfer, *Decision Making Under Uncertainty: Theory and Application*. MIT Press, 2015.

должны выбирать между набором высоты или снижением, чтобы избежать столкновения. Если оба пилота выбирают один и тот же маневр, происходит катастрофа с полезностью –4 для обоих пилотов. Поскольку для набора высоты требуется больше топлива, чем для снижения, к любому пилоту, решившему набрать высоту, применяется дополнительный штраф –1.

		агент 2: подъем	агент 2: снижение
агент 1	подъем	−5, −5	−1, 0
агент 1	снижение	0, −1	−4, −4

Упражнение 24.3. Пусть нам дана стационарная совместная стратегия π, которая является равновесием Нэша для простой игры с горизонтом 1. Докажите, что она остается равновесием Нэша для той же простой игры, повторяемой до любого конечного или бесконечного горизонта.

Решение. Из определения равновесия Нэша следует, что все агенты i реализуют наилучший отклик π^i на все другие стратегии $\pi^{i\prime} \neq \pi^i$ в соответствии с уравнением (24.2):

$$U^i(\pi^i, \boldsymbol{\pi}^{-1}) \geq U^i(\pi^{i\prime}, \boldsymbol{\pi}^{-1}).$$

По определению U^i имеем

$$U^i(\boldsymbol{\pi}) = \sum_{\mathbf{a}\in\mathbf{A}} R^i(\mathbf{a}) \prod_{j=1}^{k} \pi^j(a^j).$$

Совместная стратегия остается неизменной во времени для всех агентов. Возьмем произвольный горизонт n с произвольным коэффициентом дисконтирования ($\gamma = 1$ для $n < \infty$; $\gamma < 1$ для $n \to \infty$). Полезность i-го агента после n шагов равна

$$\begin{aligned} U^{i,n}(\boldsymbol{\pi}) &= \sum_{t=1}^{n} \gamma^{t-1} \sum_{\mathbf{a}\in\mathbf{A}} R^i(\mathbf{a}) \prod_{j=1}^{k} \pi^j(a^j) \\ &= \sum_{\mathbf{a}\in\mathbf{A}} R^i(\mathbf{a}) \prod_{j=1}^{k} \pi^j(a^j) \sum_{t=1}^{n} \gamma^{t-1} \\ &= U^i(\boldsymbol{\pi}) \sum_{t=1}^{n} \gamma^{t-1} \\ &= U^i(\boldsymbol{\pi}) c. \end{aligned}$$

Коэффициент дисконтирования становится постоянным множителем $c > 0$. Следовательно, любое умножение обеих сторон уравнения (24.2) на постоянную величину приводит к одному и тому же неравенству, что завершает доказательство:

$$U^i(\pi^i, \boldsymbol{\pi}^{-i}) \geq U^i(\pi^{i\prime}, \boldsymbol{\pi}^{-i});$$

$$U^i(\pi^i, \boldsymbol{\pi}^{-i})c \geq U^i(\pi^{i\prime}, \boldsymbol{\pi}^{-i})c;$$

$$U^i(\pi^i, \boldsymbol{\pi}^{-i})\sum_{t=1}^{n}\gamma^{t-1} \geq U^i(\pi^{i\prime}, \boldsymbol{\pi}^{-i})\sum_{t=1}^{n}\gamma^{t-1};$$

$$\sum_{t=1}^{n}\gamma^{t-1}U^i(\gamma^i, \boldsymbol{\gamma}^{-i}) \geq \sum_{t=1}^{n}\gamma^{t-1}U^i(\pi^{i\prime}, \boldsymbol{\pi}^{-i});$$

$$U^{i,n}(\pi^i, \boldsymbol{\pi}^{-i}) \geq U^{i,n}(\pi^{i\prime}, \boldsymbol{\pi}^{-i}).$$

Упражнение 24.4. Докажите, что равновесие Нэша является согласованным равновесием.

Решение. Возьмем произвольную несогласованную совместную стратегию $\pi(\mathbf{a})$. Для любого агента i:

$$\pi(\mathbf{a}) = \prod_{j=1}^{k}\pi^j(a^j) = \pi^i(a^i)\prod_{j\neq i}\pi^j(a^j). \tag{24.13}$$

Достаточно показать, что согласованное равновесие при этом ограничении образует точное определение равновесия Нэша. Начнем с применения уравнения (24.13) к определению согласованного равновесия. Для всех i, любого a^i с ненулевой вероятностью[20] в π и всех $a^{i\prime}$:

$$\sum_{\mathbf{a}^{-1}} R^i(a^i, \mathbf{a}^{-1})\pi(a^i, \mathbf{a}^{-1}) \geq \sum_{\mathbf{a}^{-1}} R^i(a^{i\prime}, \mathbf{a}^{-1})\pi(a^i, \mathbf{a}^{-1});$$

$$\sum_{\mathbf{a}^{-1}} R^i(a^i, \mathbf{a}^{-1})\pi(a^i)\prod_{j\neq i}\pi^j(a^j) \geq \sum_{\mathbf{a}^{-1}} R^i(a^{i\prime}, \mathbf{a}^{-1})\pi(a^i)\prod_{j\neq i}\pi^j(a^j);$$

$$\sum_{\mathbf{a}^{-1}} R^i(a^i, \mathbf{a}^{-1})\prod_{j\neq i}\pi^j(a^j) \geq \sum_{\mathbf{a}^{-1}} R^i(a^{i\prime}, \mathbf{a}^{-1})\prod_{j\neq i}\pi^j(a^j). \tag{24.14}$$

Теперь возьмем определение полезности:

$$U^i(\pi^i, \boldsymbol{\pi}^{-i}) = \sum_{\mathbf{a}} R^i(a^i, \mathbf{a}^{-1})\prod_{j=1}^{k}\pi^j(a^j) \geq \sum_{a^i}\pi^i(a^i)\left(\sum_{\mathbf{a}^{-i}} R^i(a^{i\prime}, \mathbf{a}^{-1})\prod_{j\neq i}\pi^j(a^j)\right).$$

[20] То есть $\sum_{\mathbf{a}^{-i}}\pi(a^i, \mathbf{a}^{-i}) > 0$. Если оно равно нулю, то неравенство тривиально становится верным при $0 \geq 0$.

Затем применим уравнение (24.14) к членам в скобках:

$$\begin{aligned} U^i(\pi^i, \boldsymbol{\pi}^{-i}) &\geq \sum_{a^i} \pi^i(a^i) \left(\sum_{\mathbf{a}^{-i}} R^i(a^{i\prime}, \mathbf{a}^{-1}) \prod_{j \neq i} \pi^j(a^j) \right) \\ &= \left(\sum_{\mathbf{a}^{-i}} R^i(a^{i\prime}, \mathbf{a}^{-1}) \prod_{j \neq i} \pi^j(a^j) \right) \sum_{a^i} \pi^i(a^i) \\ &= \sum_{\mathbf{a}^{-i}} R^i(a^{i\prime}, \mathbf{a}^{-1}) \prod_{j \neq i} \pi^j(a^j). \end{aligned}$$

Это уравнение верно для любого действия $a^{i\prime}$. Следовательно, применение любого вероятностного взвешивания сохраняет правую часть этого неравенства. Запишем любую другую стратегию $\pi^{i\prime}$ как взвешенное представление:

$$U^i(\pi^i, \boldsymbol{\pi}^{-i}) \geq \sum_{a^i} \pi^{i\prime}(a^i) \sum_{\mathbf{a}^{-i}} R^i(a^i, \mathbf{a}^{-1}) \prod_{j \neq i} \pi^j(a^j) = U^i(\pi^{i\prime}, \boldsymbol{\pi}^{-i}).$$

Это неравенство является определением наилучшего отклика. Оно должно выполняться для всех агентов i и, таким образом, является определением равновесия Нэша. Следовательно, равновесие Нэша – это особый вид согласованного равновесия, которое ограничено несогласованной совместной стратегией.

Упражнение 24.5. Приведите пример игры с двумя агентами, у каждого из которых по два действия, когда согласованные равновесия не могут быть представлены в виде равновесия Нэша.

Решение. Рассмотрим игру, в которой два человека хотят пойти на свидание, но имеют противоречивые предпочтения относительно того, где провести время (в данном случае ужин или кино):

		агент 2: ужин	агент 2: кино
агент 1	ужин	2, 1	0, 0
агент 1	кино	0, 0	1, 2

Здесь существует стохастическое равновесие Нэша. Агент 1 следует стратегии π^1(ужин) = 2/3 и π^1(кино) = 1/3. Агент 2 следует стратегии π^2(ужин) = 1/3 и π^2(кино) = 2/3. Их полезности будут следующими:

$$U^1(\boldsymbol{\pi}) = \frac{2}{3}\cdot\frac{1}{3}\cdot 2 + \frac{2}{3}\cdot\frac{2}{3}\cdot 0 + \frac{1}{3}\cdot\frac{1}{3}\cdot 0 + \frac{1}{3}\cdot\frac{2}{3}\cdot 1 = \frac{2}{9}\cdot 2 + \frac{2}{9}\cdot 1 = \frac{2}{3};$$
$$U^2(\boldsymbol{\pi}) = \frac{2}{3}\cdot\frac{1}{3}\cdot 1 + \frac{2}{3}\cdot\frac{2}{3}\cdot 0 + \frac{1}{3}\cdot\frac{1}{3}\cdot 0 + \frac{1}{3}\cdot\frac{2}{3}\cdot 2 = \frac{2}{9}\cdot 1 + \frac{2}{9}\cdot 2 = \frac{2}{3}.$$

Однако если два агента согласовали свои действия путем честного подбрасывания монеты π(кино, кино) = π(обед, ужин) = 0.5, то они могли бы согласованно получить либо два похода на ужин, либо два похода в кино. Полезности в этом случае:

$$U^1(\boldsymbol{\pi}) = 0.5\cdot 2 + 0.0\cdot 0 + 0.0\cdot 0 + 0.5\cdot 1 = 0.5\cdot 2 + 0.5\cdot 1 = \frac{3}{2};$$
$$U^2(\boldsymbol{\pi}) = 0.5\cdot 1 + 0.0\cdot 0 + 0.0\cdot 0 + 0.5\cdot 2 = 0.5\cdot 1 + 0.5\cdot 2 = \frac{3}{2}.$$

Это невозможно при равновесии Нэша. Интуитивно понятно, что в данном примере это связано с тем, что вероятностный вес распределяется по каждой строке независимо для каждого игрока. И наоборот, согласованное равновесие может быть нацелено на конкретную ячейку (в данном случае с более высоким вознаграждением).

Упражнение 24.6. Такие алгоритмы, как итерация по наилучшим откликам и фиктивная игра, сходятся не для каждой игры. Предложите игру, демонстрирующую эту несходимость.

Решение. Итерация по наилучшим откликам расходится в игре «камень–ножницы–бумага». Вот пример первых 10 итераций со случайной инициализацией:

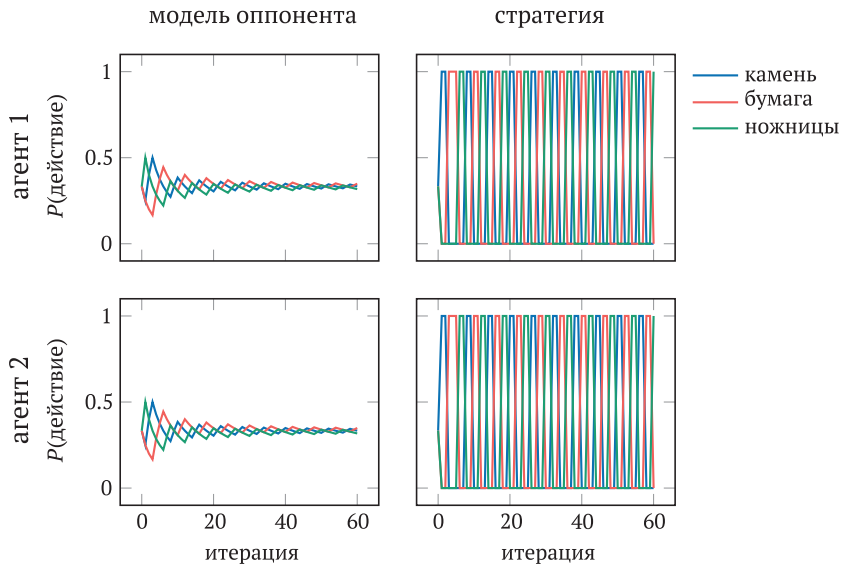
Итерация	Действие агента 1	Действие агента 1	Вознаграждение
1	бумага	камень	1.0, −1.0
2	бумага	ножницы	−1.0, 1.0
3	камень	ножницы	1.0, −1.0
4	камень	бумага	−1.0, 1.0
5	ножницы	бумага	1.0, −1.0
6	ножницы	камень	−1.0, 1.0
7	бумага	камень	1.0, −1.0
8	бумага	ножницы	−1.0, 1.0
9	камень	ножницы	1.0, −1.0
10	камень	бумага	−1.0, 1.0

Алгоритм фиктивной игры тоже не сойдется на этой игре[21]:

[21] Эта и многие другие игры более подробно обсуждаются в Y. Shoham, K. Leyton-Brown, *Multiagent Systems: Algorithmic, Game Theoretic, and Logical Foundations*. Cambridge University Press, 2009.

агент 1 \ агент 2	камень	бумага	ножницы
камень	0, 0	0, 1	1, 0
бумага	1, 0	0, 0	0, 1
ножницы	0, 1	1, 0	0, 0

Ниже показан пример игровых агентов, осуществляющих фиктивную игру на протяжении 60 итераций:



Упражнение 24.7. К чему сходится алгоритм итерации по наилучшим откликам в дилемме путешественника (приложение F.12)?

Решение. Алгоритм сходится к равновесию Нэша, равному 2 долларам.

25 Последовательные задачи

В этой главе простые игры расширяются до последовательной формы с несколькими состояниями. *Марковскую игру* (Markov game, MG) можно рассматривать как марковский процесс принятия решений с участием нескольких агентов, обладающих собственными функциями вознаграждения[1]. В этой форме игры переходы зависят от совместного действия, и все агенты стремятся максимизировать собственное вознаграждение. Мы изменим модели отклика и концепцию поиска равновесия Нэша для простых игр таким образом, чтобы учитывать модель перехода состояний. В завершающей части этой главы рассмотрены обучаемые модели, в которых агенты адаптируют свои стратегии на основе информации, полученной в результате наблюдаемых взаимодействий, и знаний о функциях вознаграждения и перехода.

25.1. Марковские игры

Марковская игра (алгоритм 25.1) расширяет понятие простой игры, включив в нее общее состояние $s \in \mathcal{S}$. Вероятность перехода из состояния s в состояние s' при совместном действии $\mathbf{a}$ определяется распределением переходов $T(s'|s, \mathbf{a})$. Каждый агент i получает вознаграждение в соответствии со своей функцией вознаграждения $R^i(s, \mathbf{a})$, которая теперь также зависит от состояния. В примере 25.1 показано, как можно представить в виде марковской игры задачу о построении дорожного маршрута.

Алгоритм 25.1. Структура данных для марковской игры `MG`

```
struct MG
    γ # коэффициент дисконтирования
    ℐ # агенты
```

---

[1] Марковские игры, также называемые стохастическими играми, первоначально изучались в 1950-х годах примерно в то же время, что и MDP. L. S. Shapley, *Stochastic Games*, Proceedings of the National Academy of Sciences, vol. 39, no. 10, pp. 1095–1100, 1953. Спустя десятилетия они заинтересовали сообщество мультиагентного искусственного интеллекта. M. L. Littman, *Markov Games as a Framework for Multi-Agent Reinforcement Learning*, in International Conference on Machine Learning (ICML), 1994.

```
    𝒮 # пространство состояний
    𝒜 # пространство совместных действий
    T # transition function
    R # joint reward function
end
```

Пример 25.1. Задача о построении дорожного маршрута в виде марковской игры. Эта задача не может быть решена с использованием одноагентной модели, такой как MDP, потому что мы не знаем поведения других агентов, а знаем только их вознаграждение. Мы можем попытаться найти равновесие или обучить стратегию посредством взаимодействия, аналогично тому, как мы делали в простых играх

Рассмотрим действия жителей, которые едут на работу на машине. У каждой машины есть начальная позиция и пункт назначения. Каждый автомобиль может добраться до места назначения по любой из нескольких доступных дорог, но эти дороги различаются по времени, затрачиваемому на поездку по ним. Чем больше автомобилей едет по определенной дороге, тем медленнее они все движутся.

Эта задача является типичной марковской игрой. Агенты – это люди в своих машинах, состояния – это расположение всех машин на дорогах, а действия соответствуют решениям о том, по какой дороге ехать дальше. Переход состояния перемещает всех автомобильных агентов вперед после их совместного действия. Отрицательное вознаграждение пропорционально времени, проведенному за рулем.

Совместная стратегия π в марковской игре определяет распределение вероятностей по совместным действиям при заданном текущем состоянии. Как и в случае с MDP, мы сосредоточимся на стратегиях, которые зависят от текущего состояния, а не от прошлой истории, потому что будущие состояния и вознаграждения условно не зависят от истории. Кроме того, мы сосредоточимся на стационарных стратегиях, которые не зависят от времени. Вероятность того, что агент i выберет действие a в состоянии s, равна $\pi^i(a|s)$. Мы будем часто использовать запись $\pi(s)$ для обозначения распределения вероятностей по совместным действиям.

Полезность совместной стратегии π с точки зрения агента i можно вычислить, используя способ оценки стратегии, представленный в разделе 7.2 для MDP. Вознаграждение агента i из состояния s при следовании совместной стратегии π равно

$$R^i(s, \boldsymbol{\pi}(s)) = \sum_{\mathbf{a}} R^i(s, \mathbf{a}) \prod_{j \in \mathcal{I}} \pi^j(a^j|s). \tag{25.1}$$

Вероятность перехода из состояния s в s' при следовании π равна

$$T(s'|s, \boldsymbol{\pi}(s)) = \sum_{\mathbf{a}} T(s'|s, \mathbf{a}) \prod_{j \in \mathcal{J}} \pi^j(a^j|s). \tag{25.2}$$

В игре с дисконтированием на бесконечном горизонте полезность для агента i из состояния s равна

$$U^{\boldsymbol{\pi},i}(s) = R^i(s, \boldsymbol{\pi}(s)) + \gamma \sum_{s'} T(s'|s, \boldsymbol{\pi}(s)) U^{\boldsymbol{\pi},i}(s'). \tag{25.3}$$

Это уравнение имеет точное решение (алгоритм 25.2).

Алгоритм 25.2. Стратегия марковской игры представляет собой сопоставление состояний со стратегиями простой игры, представленными в предыдущей главе. Мы можем построить ее, передав генератор для создания словаря. Вероятность того, что стратегия (либо для марковской, либо для простой игры) назначает действие `ai` из состояния `s`, равна `πi(s, ai)`. Также предусмотрены функции для вычисления $R^i(s, \pi(s))$ и $T(s'|s, \pi(s))$. Функция оценки стратегии будет вычислять вектор, представляющий $U^{\pi,i}$

```
struct MGPolicy
    p # словарь перевода состояний в стратегии простой игры
    MGPolicy(p::Base.Generator) = new(Dict(p))
end

(πi::MGPolicy)(s, ai) = πi.p[s](ai)
(πi::SimpleGamePolicy)(s, ai) = πi(ai)

probability(𝒫::MG, s, π, a) = prod(πj(s, aj) for (πj, aj) in zip(π, a))
reward(𝒫::MG, s, π, i) =
    sum(𝒫.R(s,a)[i]*probability(𝒫,s,π,a) for a in joint(𝒫.𝒜))
transition(𝒫::MG, s, π, s′) =
    sum(𝒫.T(s,a,s′)*probability(𝒫,s,π,a) for a in joint(𝒫.𝒜))

function policy_evaluation(𝒫::MG, π, i)
    𝒮, 𝒜, R, T, γ = 𝒫.𝒮, 𝒫.𝒜, 𝒫.R, 𝒫.T, 𝒫.γ
    p(s,a) = prod(πj(s, aj) for (πj, aj) in zip(π, a))
    R′ = [sum(R(s,a)[i]*p(s,a) for a in joint(𝒜)) for s in 𝒮]
    T′ = [sum(T(s,a,s′)*p(s,a) for a in joint(𝒜)) for s in 𝒮, s′ in 𝒮]
    return (I - γ*T′)\R′
end
```

25.2. Модели отклика

Мы можем распространить модели отклика, представленные в предыдущей главе, на марковские игры. Для этого необходимо учитывать модель перехода состояний.

25.2.1. Наилучший отклик

Стратегия отклика (response policy) для агента i – это стратегия π^i, которая максимизирует ожидаемую полезность при фиксированных стратегиях других агентов $\boldsymbol{\pi}^{-i}$. Если стратегии других агентов фиксированы, то задача сводится к MDP. Этот MDP имеет пространство состояний $\mathcal{S}$ и пространство действий $\mathcal{A}^i$. Мы можем определить функции перехода и вознаграждения следующим образом:

$$T'(s'|s, a^i) = T(s'|s, a^i, \boldsymbol{\pi}^{-i}(s)); \tag{25.4}$$

$$R'(s, a^i) = R^i(s, a^i, \boldsymbol{\pi}^{-i}(s)). \tag{25.5}$$

Поскольку это наилучший отклик для агента i, MDP использует только вознаграждение R^i. Решение этого MDP дает наилучшую стратегию отклика для агента i. Реализация представлена в алгоритме 25.3.

Алгоритм 25.3. Если дана марковская игра $\mathcal{P}$, мы можем вычислить детерминированную стратегию наилучшего отклика для агента `i`, полагая, что другие агенты используют стратегии из `π`. Точное решение задачи MDP можно найти, используя один из методов из главы 7

```
function best_response(𝒫::MG, π, i)
    𝒮, 𝒜, R, T, γ = 𝒫.𝒮, 𝒫.𝒜, 𝒫.R, 𝒫.T, 𝒫.γ
    T′(s,ai,s′) = transition(𝒫, s, joint(π, SimpleGamePolicy(ai), i), s′)
    R′(s,ai) = reward(𝒫, s, joint(π, SimpleGamePolicy(ai), i), i)
    πi = solve(MDP(γ, 𝒮, 𝒜[i], T′, R′))
    return MGPolicy(s => SimpleGamePolicy(πi(s)) for s in 𝒮)
end
```

25.2.2. Стратегия отклика softmax

По аналогии с предыдущей главой, мы можем определить *стратегию отклика softmax* (softmax response policy), которая связывает стохастический отклик со стратегиями других агентов в каждом состоянии. Как и при построении детерминированной стратегии наилучшего отклика, мы решаем задачу MDP, в которой агенты с фиксированными стратегиями π^{-i} объединены в среду. Затем мы извлекаем функцию полезности действия $Q(s, a)$, используя пошаговый предпросмотр. Стратегия отклика softmax определена следующим образом:

$$\pi^i(a^i|s) \propto \exp(\lambda Q(s, a^i)) \tag{25.6}$$

с параметром точности $\lambda \geq 0$. Реализация представлена в алгоритме 25.4. Этот подход можно использовать для создания иерархических решений softmax (раздел 24.7). На самом деле мы можем применить непосредственно алгоритм 24.9.

Алгоритм 25.4. Softmax-отклик агента `i` на совместную стратегию π с параметром точности λ

```
function softmax_response(𝒫::MG, π, i, λ)
    𝒮, 𝒜, R, T, γ = 𝒫.𝒮, 𝒫.𝒜, 𝒫.R, 𝒫.T, 𝒫.γ
    T′(s,ai,s′) = transition(𝒫, s, joint(π, SimpleGamePolicy(ai), i), s′)
    R′(s,ai) = reward(𝒫, s, joint(π, SimpleGamePolicy(ai), i), i)
    mdp = MDP(γ, 𝒮, joint(𝒜), T′, R′)
    πi = solve(mdp)
    Q(s,a) = lookahead(mdp, πi.U, s, a)
    p(s) = SimpleGamePolicy(a => exp(λ*Q(s,a)) for a in 𝒜[i])
    return MGPolicy(s => p(s) for s in 𝒮)
end
```

25.3. Равновесие Нэша

Понятие равновесия Нэша можно распространить на марковские игры[2]. Как и в простых играх, при равновесии все агенты наилучшим образом реагируют друг на друга и не имеют стимула отклоняться от выбранной стратегии. Все конечные марковские игры с дисконтированным бесконечным горизонтом имеют равновесие Нэша[3].

Мы можем найти равновесие Нэша, решив задачу нелинейной оптимизации, аналогичную той, которую мы решали в случае простых игр. Эта задача заключается в минимизации суммы отклонений полезности при использовании алгоритма предпросмотра и ограничивает стратегии следующими допустимыми распределениями:

$$\begin{aligned}
&\underset{\boldsymbol{\pi},U}{\text{минимизировать}} \sum_{i\in\mathcal{I}}\sum_{s}\left(U^i(s) - Q^i(s,\boldsymbol{\pi}(s))\right),\\
&\text{при условии что } U^i(s) \geq Q^i(s,a^i,\boldsymbol{\pi}^{-i}(s)) \text{ для всех } i,s,a^i;\\
&\qquad\qquad \sum_{a^i}\pi^i(a^i|s) = 1 \text{ для всех } i,s;\\
&\qquad\qquad \pi^i(a^i|s) \text{ для всех } i,s,a^i,
\end{aligned} \tag{25.7}$$

где

$$Q^i(s,\boldsymbol{\pi}(s)) = R^i(s,\boldsymbol{\pi}(s)) + \gamma\sum_{s'}T(s'|s,\boldsymbol{\pi}(s))U^i(s'). \tag{25.8}$$

---

[2] Поскольку мы предполагаем, что политика является *стационарной* в том смысле, что она не меняется во времени, рассматриваемые здесь равновесия Нэша являются *стационарными совершенными по Маркову равновесиями*.

[3] A. M. Fink, *Equilibrium in a Stochastic n-Person Game*, Journal of Science of the Hiroshima University, Series A-I, vol. 28, no. 1, pp. 89–93, 1964.

Эта задача нелинейной оптимизации реализована в алгоритме 25.5[4].

Алгоритм 25.5. Нелинейная программа, вычисляющая равновесие Нэша для марковской игры $\mathcal{P}$

```
function tensorform(𝒫::MG)
    ℐ, 𝒮, 𝒜, R, T = 𝒫.ℐ, 𝒫.𝒮, 𝒫.𝒜, 𝒫.R, 𝒫.T
    ℐ′ = eachindex(ℐ)
    𝒮′ = eachindex(𝒮)
    𝒜′ = [eachindex(𝒜[i]) for i in ℐ]
    R′ = [R(s,a) for s in 𝒮, a in joint(𝒜)]
    T′ = [T(s,a,s′) for s in 𝒮, a in joint(𝒜), s′ in 𝒮]
    return ℐ′, 𝒮′, 𝒜′, R′, T′
end

function solve(M::NashEquilibrium, 𝒫::MG)
    ℐ, 𝒮, 𝒜, R, T = tensorform(𝒫)
    𝒮′, 𝒜′, γ = 𝒫.𝒮, 𝒫.𝒜, 𝒫.γ
    model = Model(Ipopt.Optimizer)
    @variable(model, U[ℐ, 𝒮])
    @variable(model, π[i=ℐ, 𝒮, ai=𝒜[i]] ≥ 0)
    @NLobjective(model, Min,
        sum(U[i,s] - sum(prod(π[j,s,a[j]] for j in ℐ)
            * (R[s,y][i] + γ*sum(T[s,y,s′]*U[i,s′] for s′ in 𝒮))
            for (y,a) in enumerate(joint(𝒜))) for i in ℐ, s in 𝒮))
    @NLconstraint(model, [i=ℐ, s=𝒮, ai=𝒜[i]],
        U[i,s] ≥ sum(
            prod(j==i ? (a[j]==ai ? 1.0 : 0.0) : π[j,s,a[j]] for j in ℐ)
            * (R[s,y][i] + γ*sum(T[s,y,s′]*U[i,s′] for s′ in 𝒮))
            for (y,a) in enumerate(joint(𝒜))))
    @constraint(model, [i=ℐ, s=𝒮], sum(π[i,s,ai] for ai in 𝒜[i]) == 1)
    optimize!(model)
    π′ = value.(π)
    πi′(i,s) = SimpleGamePolicy(𝒜′[i][ai] => π′[i,s,ai] for ai in 𝒜[i])
    πi′(i) = MGPolicy(𝒮′[s] => πi′(i,s) for s in 𝒮)
    return [πi′(i) for i in ℐ]
end
```

25.4. Фиктивная марковская игра

Как и в случае с простыми играми, мы можем использовать подход, основанный на обучении, чтобы прийти к совместной стратегии путем моделирования действий агентов. Алгоритм 25.6 расширяет применение цикла модели-

[4] J. A. Filar, T. A. Schultz, F. Thuijsman, O. Vrieze, *Nonlinear Programming and Stationary Equilibria in Stochastic Games*, Mathematical Programming, vol. 50, no. 1–3, pp. 227–237, 1991.

рования, представленного в предыдущей главе, добавляя обработку переходов между состояниями. Стратегии, применяемые при моделировании, обновляются в зависимости от переходов состояний и действий, предпринимаемых агентами.

Алгоритм 25.6. Функции для выполнения случайного шага и запуска полных проходов моделирования в марковской игре. Функция `simulate` будет моделировать совместную стратегию `π` для `k_max` шагов, начиная с состояния, случайно выбранного из `b`

```
function randstep(𝒫::MG, s, a)
    s′ = rand(SetCategorical(𝒫.𝒮, [𝒫.T(s, a, s′) for s′ in 𝒫.𝒮]))
    r = 𝒫.R(s,a)
    return s′, r
end

function simulate(𝒫::MG, π, k_max, b)
    s = rand(b)
    for k = 1:k_max
        a = Tuple(πi(s)() for πi in π)
        s′, r = randstep(𝒫, s, a)
        for πi in π
            πi, s, a, s′)
        end
        s = s′
    end
    return π
end
```

Один из подходов к обновлению стратегий заключается в использовании обобщения *фиктивной игры* (алгоритм 25.7) из предыдущей главы[5], которое подразумевает наличие модели максимального правдоподобия в отношении стратегий других агентов. Модель максимального правдоподобия отслеживает состояние в дополнение к действию, предпринимаемому каждым агентом. Иначе говоря, мы отслеживаем, сколько раз агент j выполняет действие a^j в состоянии s, сохраняя подсчет в таблице $N(j, a^j, s)$, обычно инициализируемой значением 1. Затем мы можем вычислить наилучший отклик, предполагая, что каждый агент j следует стохастической стратегии, зависящей от состояния:

$$\pi^j(a^j|s) \propto N(j, a^j, s)). \tag{25.9}$$

[5] W. Uther, M. Veloso, *Adversarial Reinforcement Learning*, Carnegie Mellon University, Tech. Rep. CMU-CS-03-107, 1997. M. Bowling, M. Veloso, *An Analysis of Stochastic Game Theory for Multiagent Reinforcement Learning*, Carnegie Mellon University, Tech. Rep. CMU-CS-00-165, 2000.

Алгоритм 25.7. Реализация фиктивной игры для агента `i` в марковской игре $\mathcal{P}$, которая хранит подсчеты `Ni` других вариантов действий агентов с течением времени в каждом состоянии и усредняет их, предполагая, что это их стохастическая стратегия. Затем алгоритм вычисляет наилучший отклик на эту стратегию и выполняет соответствующее действие по максимизации полезности

```
mutable struct MGFictitiousPlay
    𝒫 # Марковская игра
    i # индекс агента
    Qi # оценки полезности состояния-действия
    Ni # счетчики состояния-действия
end

function MGFictitiousPlay(𝒫::MG, i)
    ℐ, 𝒮, 𝒜, R = 𝒫.ℐ, 𝒫.𝒮, 𝒫.𝒜, 𝒫.R
    Qi = Dict((s, a) => R(s, a)[i] for s in 𝒮 for a in joint(𝒜))
    Ni = Dict((j, s, aj) => 1.0 for j in ℐ for s in 𝒮 for aj in 𝒜[j])
    return MGFictitiousPlay(𝒫, i, Qi, Ni)
end

function (πi::MGFictitiousPlay)(s)
    𝒫, i, Qi = πi.𝒫, πi.i, πi.Qi
    ℐ, 𝒮, 𝒜, T, R, γ = 𝒫.ℐ, 𝒫.𝒮, 𝒫.𝒜, 𝒫.T, 𝒫.R, 𝒫.γ
    πi′(i,s) = SimpleGamePolicy(ai => πi.Ni[i,s,ai] for ai in 𝒜[i])
    πi′(i) = MGPolicy(s => πi′(i,s) for s in 𝒮)
    π = [πi′(i) for i in ℐ]
    U(s,π) = sum(πi.Qi[s,a]*probability(𝒫,s,π,a) for a in joint(𝒜))
    Q(s,π) = reward(𝒫,s,π,i) + γ*sum(transition(𝒫,s,π,s′)*U(s′,π)
                                     for s′ in 𝒮)
    Q(ai) = Q(s, joint(π, SimpleGamePolicy(ai), i))
    ai = argmax(Q, 𝒫.𝒜[πi.i])
    return SimpleGamePolicy(ai)
end

function update!(πi::MGFictitiousPlay, s, a, s′)
    𝒫, i, Qi = πi.𝒫, πi.i, πi.Qi
    ℐ, 𝒮, 𝒜, T, R, γ = 𝒫.ℐ, 𝒫.𝒮, 𝒫.𝒜, 𝒫.T, 𝒫.R, 𝒫.γ
    for (j,aj) in enumerate(a)
        πi.Ni[j,s,aj] += 1
    end
    πi′(i,s) = SimpleGamePolicy(ai => πi.Ni[i,s,ai] for ai in 𝒜[i])
    πi′(i) = MGPolicy(s => πi′(i,s) for s in 𝒮)
    π = [πi′(i) for i in ℐ]
    U(π,s) = sum(πi.Qi[s,a]*probability(𝒫,s,π,a) for a in joint(𝒜))
    Q(s,a) = R(s,a)[i] + γ*sum(T(s,a,s′)*U(π,s′) for s′ in 𝒮)
    for a in joint(𝒜)
        πi.Qi[s,a] = Q(s,a)
    end
end
```

После наблюдения за совместным действием $\mathbf{a}$ в состояниях s мы обновляем соответствующий счетчик

$$N(j, a^j, s) \leftarrow N(j, a^j, s) + 1 \tag{25.10}$$

для каждого агента j.

По мере изменения распределения действий других агентов мы должны обновлять полезности. Полезности в марковской игре вычисляются значительно сложнее, чем в простой игре, из-за необходимости учитывать зависимость от состояния. Как сказано в разделе 25.2.1, любое назначение фиксированных стратегий $\boldsymbol{\pi}^{-i}$ порождает задачу MDP. В фиктивной игре $\boldsymbol{\pi}^{-i}$ определяется уравнением (25.9). Вместо того чтобы заново решать MDP при каждом обновлении, обычно обновление применяется периодически – стратегия, заимствованная из асинхронной итерации по полезности. Иллюстрация фиктивной игры показана в примере 25.2.

Стратегия $\pi^i(s)$ для состояния s выводится из заданной модели оппонента $\boldsymbol{\pi}^{-i}$ и вычисленной полезности U^i. Затем мы выбираем лучший ответ:

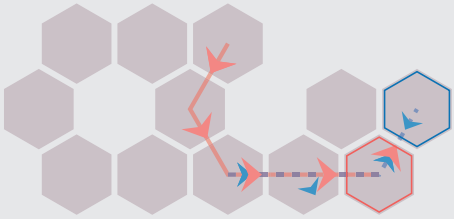
$$\arg\max_a Q^i(s, a, \boldsymbol{\pi}^{-i}). \tag{25.11}$$

В реализации алгоритма мы используем тот факт, что каждое состояние стратегии марковской игры является стратегией простой игры, наградой за которую является соответствующее Q^i.

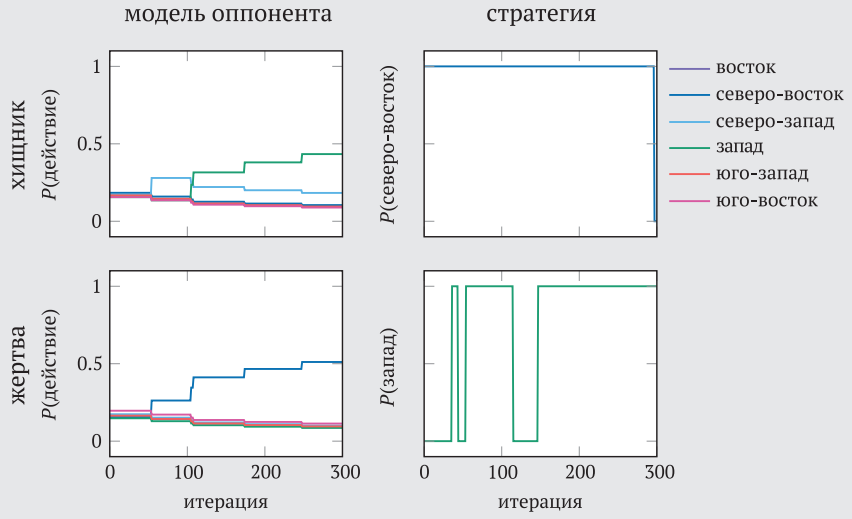
Пример 25.2. Фиктивная игра в задаче о гексамире, вариант «хищник–жертва». При инициализации стратегий была введена стохастичность, чтобы лучше отображать тенденции обучения

Марковская игра в задаче о гексамире «хищник–жертва» (приложение F.13) содержит одного хищника (красный цвет) и одну жертву (синий цвет). Если хищник ловит жертву, он получает вознаграждение в размере 10, а жертва получает отрицательное вознаграждение в размере –100. В противном случае оба агента получают отрицательное вознаграждение –1. Агенты двигаются одновременно. Применим фиктивную игру со сбросом в начальное состояние через каждые 10 шагов.

Мы наблюдаем, как хищник учится преследовать добычу, а жертва учится убегать. Интересно, что хищник также обучается, что добыча бежит в восточный угол и ждет там. Жертва обучается, что если она будет ждать в этом углу, она может немедленно убежать от хищника, когда он прыгает в ее сторону. В этом случае жертва уклоняется от хищника, двигаясь на запад, когда хищник движется на северо-восток.



Ниже показаны графики обученных моделей для выделенного на рисунке состояния как для хищника, так и для жертвы:



25.5. Градиентный подъем

В марковских играх мы можем использовать метод градиентного подъема (алгоритм 25.8) для обучения стратегий аналогично тому, как это делалось в предыдущей главе для простых игр. Но теперь необходимо учитывать состояние, что требует обучения функции полезности действия. На каждом временном шаге t все агенты выполняют совместные действия $\mathbf{a}_t$ в состоянии s_t. Как и при градиентном подъеме для простых игр, агент i предполагает, что стратегии агентов $\boldsymbol{\pi}_t^{-i}$ проявляются через наблюдаемые действия $\mathbf{a}_t^{-i}$. Градиент вычисляется следующим образом:

$$\frac{\partial U^{\pi_t,i}(s_t)}{\partial \boldsymbol{\pi}_t^i(a^i|s_t)} = \frac{\partial}{\partial \boldsymbol{\pi}^i(a^i|s_t)}\left(\sum_{\mathbf{a}}\prod_j \pi^j(a^j|s_t)Q^{\pi_t,i}(s_t,\mathbf{a}_t)\right) \tag{25.12}$$

$$= Q^{\pi_t,i}(s_t,a^i,\mathbf{a}_t^{-i}). \tag{25.13}$$

Шаг градиента следует той же схеме, что и в предыдущей главе, за исключением того, что теперь рассматривается состояние s и используется оценка ожидаемой полезности Q_t^i:

$$\pi_{t+1}^i(a^i|s_t) = \pi_t^i(a^i|s_t) + \alpha_{t+1}^i Q^i(s_t, a^i, \mathbf{a}^{-i}). \tag{25.14}$$

Данное обновление тоже может нуждаться в проекции на симплекс, чтобы гарантировать, что стратегия π_{t+1}^i в s_t является действительным распределением вероятностей.

Как и в случае фиктивной игры в предыдущем разделе, необходимо оценить Q_t^i. Можно использовать Q-обучение:

$$Q_{t+1}^i(s_t, \mathbf{a}_t) = Q_t^i(s_t, \mathbf{a}_t) + \alpha_t \left(R^i(s_t, \mathbf{a}_t) + \gamma \max_{a^{i\prime}} Q_t^i(s_{t+1}, a^{i\prime}, \mathbf{a}_t^{-i}) - Q_t^i(s_t, \mathbf{a}_t) \right). \tag{25.15}$$

Часто задают скорость обучения, обратно пропорциональную квадратному корню из времени: $\alpha_t = 1/\sqrt{t}$. В данном случае также требуется исследование. Мы можем применить ε-жадную стратегию исследования, возможно, также с $\varepsilon_t = 1/\sqrt{t}$.

Алгоритм 25.8. Градиентное восхождение агента `i` в марковской игре $\mathcal{P}$. Алгоритм постепенно обновляет свои распределения по действиям в посещенных состояниях после градиентного подъема, чтобы улучшить ожидаемую полезность. Функция проекции на симплекс из алгоритма 23.6 используется для того, чтобы результирующая стратегия оставалась допустимым распределением вероятностей

```
mutable struct MGGradientAscent
    𝒫 # Марковская игра
    i # индекс агента
    t # шаг времени
    Qi # оценки полезности состояния-действия
    πi # текущая стратегия
end

function MGGradientAscent(𝒫::MG, i)
    ℐ, 𝒮, 𝒜 = 𝒫.ℐ, 𝒫.𝒮, 𝒫.𝒜
    Qi = Dict((s, a) => 0.0 for s in 𝒮, a in joint(𝒜))
    uniform() = Dict(s => SimpleGamePolicy(ai => 1.0 for ai in 𝒫.𝒜[i])
    for s in 𝒮)
    return MGGradientAscent(𝒫, i, 1, Qi, uniform())
end

function (πi::MGGradientAscent)(s)
    𝒜i, t = πi.𝒫.𝒜[πi.i], πi.t
    ε = 1 / sqrt(t)
    πi′(ai) = ε/length(𝒜i) + (1-ε)*πi.πi[s](ai)
    return SimpleGamePolicy(ai => πi′(ai) for ai in 𝒜i)
end
```

```
function update!(ni::MGGradientAscent, s, a, s′)
    𝒫, i, t, Qi = ni.𝒫, ni.i, ni.t, ni.Qi
    ℐ, 𝒮, 𝒜i, R, γ = 𝒫.ℐ, 𝒫.𝒮, 𝒫.𝒜[ni.i], 𝒫.R, 𝒫.γ
    jointπ(ai) = Tuple(j == i ? ai : a[j] for j in ℐ)
    α = 1 / sqrt(t)
    Qmax = maximum(Qi[s′, jointπ(ai)] for ai in 𝒜i)
    ni.Qi[s, a] += α * (R(s, a)[i] + γ * Qmax - Qi[s, a])
    u = [Qi[s, jointπ(ai)] for ai in 𝒜i]
    π′ = [πi.πi[s](ai) for ai in 𝒜i]
    π = project_to_simplex(π′ + u / sqrt(t))
    ni.t = t + 1
    ni.πi[s] = SimpleGamePolicy(ai => p for (ai, p) in zip(𝒜i, π))
end
```

25.6. Q-обучение Нэша

Другим подходом, основанным на обучении, является *Q-обучение Нэша* (Nash Q-learning, алгоритм 25.9), которое основано на Q-обучении (раздел 17.2)[6]. Данный метод использует оценку функции полезности действия, которая уточняется по мере того, как агенты реагируют на изменение стратегии друг друга. В процессе обновления функции полезности действия алгоритм находит равновесие Нэша для моделирования поведения других агентов.

Алгоритм 25.9. Q-обучение Нэша для агента `i` в марковской игре $\mathcal{P}$. Алгоритм выполняет Q-обучение совместного действия, чтобы обучить функцию полезности состояния-действия для всех агентов. Далее с `Q` строится простая игра, и мы находим равновесие Нэша, используя алгоритм 24.5. Затем это равновесие используется для обновления функции полезности. В этой реализации также применяется переменная скорость обучения, пропорциональная количеству посещений пар состояние – совместное действие, которое хранится в `N`. Кроме того, алгоритм использует ε-жадное исследование, чтобы гарантированно изучить все состояния и действия

```
mutable struct NashQLearning
    𝒫 # Markov game
    i # agent index
    Q # state-action value estimates
    N # history of actions performed
end

function NashQLearning(𝒫::MG, i)
    ℐ, 𝒮, 𝒜 = 𝒫.ℐ, 𝒫.𝒮, 𝒫.𝒜
    Q = Dict((j, s, a) => 0.0 for j in ℐ, s in 𝒮, a in joint(𝒜))
```

[6] J. Hu, M. P. Wellman, *Nash Q-Learning for General-Sum Stochastic Games*, Journal of Machine Learning Research, vol. 4, pp. 1039–1069, 2003.

```
    N = Dict((s, a) => 1.0 for s in 𝒮, a in joint(𝒜))
    return NashQLearning(𝒫, i, Q, N)
end

function (πi::NashQLearning)(s)
    𝒫, i, Q, N = πi.𝒫, πi.i, πi.Q, πi.N
    ℐ, 𝒮, 𝒜, 𝒜i, γ = 𝒫.ℐ, 𝒫.𝒮, 𝒫.𝒜, 𝒫.𝒜[πi.i], 𝒫.γ
    M = NashEquilibrium()
    𝒢 = SimpleGame(γ, ℐ, 𝒜, a -> [Q[j, s, a] for j in ℐ])
    π = solve(M, 𝒢)
    ε = 1 / sum(N[s, a] for a in joint(𝒜))
    πi′(ai) = ε/length(𝒜i) + (1-ε)*π[i](ai)
    return SimpleGamePolicy(ai => πi′(ai) for ai in 𝒜i)
end

function update!(πi::NashQLearning, s, a, s′)
    𝒫, ℐ, 𝒮, 𝒜, R, γ = πi.𝒫, πi.𝒫.ℐ, πi.𝒫.𝒮, πi.𝒫.𝒜, πi.𝒫.R, πi.𝒫.γ
    i, Q, N = πi.i, πi.Q, πi.N
    M = NashEquilibrium()
    𝒢 = SimpleGame(γ, ℐ, 𝒜, a′ -> [Q[j, s′, a′] for j in ℐ])
    π = solve(M, 𝒢)
    πi.N[s, a] += 1
    α = 1 / sqrt(N[s, a])
    for j in ℐ
        πi.Q[j,s,a] += α*(R(s,a)[j] + γ*utility(𝒢,π,j) - Q[j,s,a])
    end
end
```

Агент, действующий по методу Q-обучения Нэша, выполняет оценку функции полезности совместного действия $\mathbf{Q}(s, \mathbf{a})$. Эта функция обновляется после каждого перехода состояния с использованием равновесия Нэша, вычисленного из простой игры, построенной на основе данной функции полезности. После перехода от s к s' в результате совместного действия $\mathbf{a}$ мы строим простую игру с тем же числом агентов и тем же пространством совместных действий, но функция вознаграждения равна оценочному значению в s', такому что $\mathbf{R}(\mathbf{a}') = \mathbf{Q}(s', \mathbf{a}')$. Агент вычисляет стратегию равновесия Нэша π' для следующего действия $\mathbf{a}'$. В соответствии с полученной стратегией ожидаемая полезность состояния-преемника равна

$$\mathbf{U}(s') = \sum_{\mathbf{a}'} \mathbf{Q}(s', \mathbf{a}') \prod_{j \in \mathcal{I}} \pi^{j\prime}(a^{j\prime}). \tag{25.16}$$

Затем агент обновляет свою функцию полезности:

$$\mathbf{Q}(s, \mathbf{a}) \leftarrow \mathbf{Q}(s, \mathbf{a}) + \alpha(\mathbf{R}(s, \mathbf{a}) + \gamma \mathbf{U}(s') - \mathbf{Q}(s, \mathbf{a})), \tag{25.17}$$

где скорость обучения α обычно является функцией количества состояний-действий $\alpha = \sqrt{N(s, \mathbf{a})}$.

Как и в случае обычного Q-обучения, нам необходимо применять стратегию исследования, чтобы убедиться, что все состояния и действия используются достаточно часто. В алгоритме 25.9 агент следует ε-жадной стратегии. С вероятностью $\varepsilon = 1/\sum_{\mathbf{a}}(N(s, \mathbf{a}))$ он равномерно выбирает действие случайным образом. В противном случае он будет использовать результат из равновесия Нэша.

25.7. Заключение

- Марковские игры являются расширением MDP для нескольких агентов или расширением простых игр для последовательных задач. В этих задачах несколько агентов соревнуются между собой и каждый со временем получает вознаграждение.
- Равновесие Нэша можно найти и для марковской игры, но теперь оно должно учитывать все действия всех агентов во всех состояниях.
- Задачу нахождения равновесия Нэша можно сформулировать как задачу нелинейной оптимизации.
- Метод фиктивной игры можно обобщить на марковские игры, используя известную функцию перехода и добавив оценки полезности действия.
- Алгоритмы градиентного подъема итеративно улучшают стохастическую стратегию, и им не нужно предполагать модель.
- Q-обучение Нэша адаптирует традиционное Q-обучение к многоагентным задачам и предусматривает нахождение равновесия Нэша в простой игре, построенной на основе моделей поведения других игроков.

25.8. Упражнения

Упражнение 25.1. Покажите, что марковские игры обобщают как MDP, так и простые игры. Для этого представьте в виде марковских игр MDP и простую игру.

Решение. Покажем, что марковские игры обобщают простые игры. Для любой простой игры с $\mathcal{I}$, $\mathcal{A}$ и $\mathbf{R}$ мы можем построить марковскую игру, имея только одно состояние, которое зациклено на себя. Другими словами, эта марковская игра имеет $\mathcal{S} = \{s^1\}$, $T(s^1|s^1, \mathbf{a}) = 1$ и $\mathbf{R}(s^1, \mathbf{a}) = \mathbf{R}(\mathbf{a})$.

Покажем, что марковские игры обобщают MDP. Для любого MDP с $\mathcal{S}$, $\mathcal{A}$, T и R мы можем построить марковскую игру, просто состоящую из единственного агента. Другими словами, эта марковская игра имеет $\mathcal{I} = \{1\}$, $\mathcal{A}^2 = \mathcal{A}$, $T(s'|s, \mathbf{a}) = T(s'|s', a)$ и $\mathbf{R}(s, a) = R(s, a)$.

Упражнение 25.2. Если мы рассматриваем агента i при фиксированной стратегии других агентов $\boldsymbol{\pi}^{-i}$, может ли существовать стохастический наилучший

отклик, дающий большую полезность, чем детерминированный наилучший отклик? Почему мы рассматриваем стохастическую стратегию в равновесии Нэша?

Решение. Нет, не может. Если заданы фиксированные стратегии других агентов $\boldsymbol{\pi}^{-i}$, детерминированный наилучший отклик достаточен для достижения наибольшей полезности. Лучший отклик можно представить в виде решения MDP, как описано в разделе 25.2. Было показано, что детерминированных стратегий достаточно для обеспечения оптимальной максимизации полезности. Следовательно, то же верно и для наилучшего отклика в марковской игре.

В равновесии Нэша наилучший отклик должен присутствовать для всех агентов. Хотя детерминированный наилучший отклик может быть по полезности равен стохастическому, для достижения равновесия могут потребоваться стохастические отклики, чтобы у других агентов не было стимула для отказа от стратегии равновесия.

Упражнение 25.3. В этой главе обсуждались только стационарные марковские стратегии. Какие еще существуют категории стратегий?

Решение. Так называемая *поведенческая стратегия* (behavioral policy) $\pi^i(\mathbf{h}_t)$ зависит от полной истории $\mathbf{h}_t = (s_{1:t}, \mathbf{a}_{1:t-1})$. Такие стратегии зависят от истории поведения других агентов. *Нестационарная марковская стратегия* $\pi^i(s, t)$ зависит от временного шага t, но не от всей истории. Например, в гексамире типа «хищник–жертва» в течение первых 10 временных шагов действие может заключаться в движении на восток, а после 10 шагов – в движении на запад.

Возможны равновесия Нэша в пространстве нестационарных немарковских совместных стратегий; стационарных немарковских совместных стратегий и т. д. Однако доказано, что любая стационарная марковская игра имеет стационарное марковское равновесие Нэша.

Упражнение 25.4. Использование метода фиктивной игры требует оценки полезности. Перечислите различные способы вычисления полезности с их преимуществами и недостатками.

Решение. Алгоритм 25.7 выполняет одну операцию дублирования для посещенных состояний s и всех совместных действий $\mathbf{a}$. Преимущество этого подхода в том, что он относительно эффективен, потому что выполняется единственное дублирование. Обновление всех совместных действий в этом состоянии приводит к исследованию действий, которые не наблюдались. Недостаток данного подхода заключается в том, что может потребоваться выполнить это обновление во всех состояниях много раз, чтобы получить подходящую стратегию.

Альтернативой является обновление только посещаемого состояния и фактического совместного действия, что ускоряет шаг обновления. Недостатком является то, что для изучения всего спектра совместных действий требуется гораздо больше шагов.

Другой альтернативой является выполнение итерации по полезности во всех состояниях s до сходимости на каждом шаге обновления. Напомним, что

модель оппонента меняется при каждом обновлении. Это порождает новый MDP, как было отмечено для детерминированного наилучшего отклика в разделе 25.2.1. Следовательно, нам нужно будет повторно запускать итерацию по полезности после каждого обновления. Преимущество этого подхода заключается в том, что он может привести к наиболее обоснованному решению на каждом этапе, поскольку полезности Q^i учитывают все состояния за рассматриваемый период времени. Недостаток заключается в том, что шаг обновления требует больших вычислительных ресурсов.

26 Неопределенность состояния

До сих пор при обсуждении многоагентных моделей мы предполагали, что все агенты могут наблюдать истинное состояние системы. Точно так же, как MDP может быть расширен на случаи частичной наблюдаемости, понятие марковской игры можно расширить до *частично наблюдаемой марковской игры* (partially observable Markov game, POMG)[1]. Фактически POMG охватывает все остальные задачи, представленные в этой книге. Эти сложные задачи можно использовать для представления сценариев, в которых несколько агентов получают частичные или зашумленные наблюдения за окружающей средой. Такое глубокое обобщение значительно усложняет моделирование и решение POMG в вычислительном отношении. В этой главе дано определение POMG, описаны представления стратегии и представлены методы решения.

26.1. Частично наблюдаемые марковские игры

POMG (алгоритм 26.1) можно рассматривать либо как расширение марковской игры для случая частичной наблюдаемости, либо как расширение POMDP для нескольких агентов. Каждый агент $i \in \mathcal{I}$ выбирает действие $a^i \in \mathcal{A}^i$, основываясь только на локальных наблюдениях o^i за общим состоянием s. Истинное состояние системы $s \in \mathcal{S}$ разделяют все агенты, но оно не обязательно полностью наблюдаемое. Начальное состояние выбирается из известного распределения начальных состояний b. Вероятность перехода из состояния s в состояние s' при совместном действии $\mathbf{a}$ вычисляется как $T(s'|s, \mathbf{a})$. Совместное вознаграждение $\mathbf{r}$ генерируется в соответствии с функцией $R^i(s, \mathbf{a})$, как и в марковской игре. Каждый агент стремится максимизировать собственное накопленное вознаграждение. После того как все агенты совершат совместное действие $\mathbf{a}$, среда порождает *совместное наблюдение* (joint observation) $\mathbf{o} = (o^1, \ldots, o^k)$ из *пространства совместного наблюдения* (joint observation space) $\mathcal{O} = \mathcal{O}^1 \times \cdots \times \mathcal{O}^k$. За-

[1] POMG также называют *частично наблюдаемой стохастической игрой* (partially observable stochastic game, POSG). POMG тесно связаны с экстенсивной формой игры в условиях несовершенной информации. H. Kuhn, *Extensive Games and the Problem of Information*, in Contributions to the Theory of Games II, H. Kuhn, A. Tucker, eds., Princeton University Press, 1953, pp. 193–216. Позже модель была предложена сообществу искусственного интеллекта. E. A. Hansen, D. S. Bernstein, S. Zilberstein, *Dynamic Programming for Partially Observable Stochastic Games*, in AAAI Conference on Artificial Intelligence (AAAI), 2004.

тем каждый агент получает индивидуальное наблюдение из этого совместного наблюдения. В примере 26.1 показано расширение задачи о плачущем ребенке на случай с несколькими агентами.

Алгоритм 26.1. Структура данных для POMG

```
struct POMG
    γ # коэффициент дисконтирования
    𝐼 # агенты
    𝒮 # пространство состояний
    𝒜 # пространство совместных действий
    𝒪 # пространство совместных наблюдений
    T # функция перехода
    O # функция совместных наблюдений
    R # функция совместного вознаграждения
end
```

Пример 26.1. Задача о плачущем ребенке с несколькими няньками, представленная как POMG. Дополнительное описание задачи дано в приложении F.14

Применим подход POMG к задаче о плачущем ребенке. У нас есть две няньки, которые ухаживают за ребенком. Как и в версии POMDP, есть два состояния ребенка – голоден или сыт. Действия каждой няньки заключаются в том, чтобы кормить, петь или игнорировать ребенка. Если обе няньки решают выполнить одно и то же действие, его вознаграждение уменьшается вдвое. Например, если ребенка кормят обе няньки, то вознаграждение составит всего −2.5 вместо −5. Однако нянькам недоступно идеальное наблюдение за состоянием малыша. Вместо этого они полагаются на зашумленные наблюдения за плачущим ребенком (обе получают одно и то же наблюдение). Благодаря такой структуре вознаграждения существует компромисс между помощью друг другу и жадным выбором менее затратного действия.

В POMDP мы могли обновлять убеждения, как обсуждалось в главе 19, но этот подход невозможен в POMG. Отдельные агенты не могут выполнять такие же обновления убеждений, как в POMDP, потому что совместные действия и совместные наблюдения не реализуются. Вывод распределения вероятностей по совместным действиям требует, чтобы каждый агент рассуждал о других агентах, рассуждающих друг о друге, которые, в свою очередь, рассуждают друг о друге, и т. д. Вывод распределения по другим наблюдениям так же сложен, потому что наблюдения зависят от действий других агентов[2].

[2] Интерактивная модель POMDP (IPOMDP) пытается отразить эту бесконечную регрессию. P. J. Gmytrasiewicz, P. Doshi, *A Framework for Sequential Planning in Multi-Agent Settings*, Journal of Artificial Intelligence Research, vol. 24, no. 1, pp. 49–79, 2005. Несмот-

Из-за сложности явного моделирования убеждений в POMG мы сосредоточимся на представлениях стратегии, которые не требуют наличия известного убеждения для определения действия. Мы можем использовать условный план в виде дерева и представление контроллера на основе графа, о которых было сказано в предыдущих главах, посвященных POMDP. Как и в обычной марковской игре, каждый агент в POMG действует в соответствии со стратегией π^i, или, что эквивалентно, агенты действуют сообща в соответствии с совместной стратегией $\boldsymbol{\pi} = (\pi^1, \ldots, \pi^k)$.

26.2. Оценка стратегии

В этом разделе мы рассмотрим оценку совместных стратегий, представленных либо в виде условных планов древовидной формы, либо в виде графовых контроллеров. В контексте POMDP мы используем условные планы для представления детерминированных стратегий и контроллеры для представления стохастических стратегий.

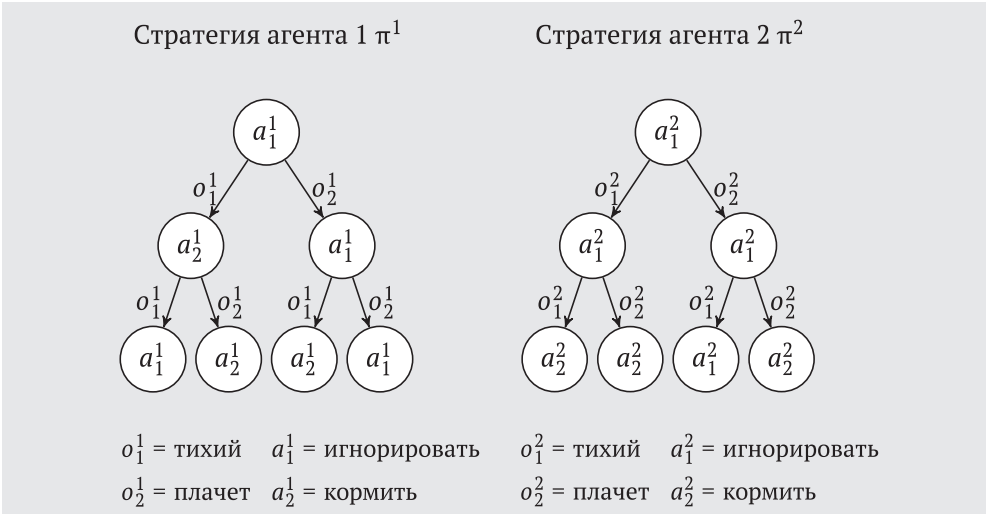
26.2.1. Оценка условных планов

Напомним, что условный план (раздел 20.2) – это дерево, в котором действия обозначены узлами, а наблюдения – ребрами. Каждый агент имеет собственное дерево и изначально выбирает действие, расположенное в корне. После наблюдения каждый агент перемещается по дереву, выбирая ребро, которое соответствует текущему наблюдению. Процесс выполнения действий и выбора ребер на основе наблюдений продолжается до тех пор, пока не будет достигнут конец дерева. В примере 26.2 показана совместная стратегия, состоящая из условного плана для каждого агента.

Пример 26.2. Двухэтапная совместная стратегия с участием двух агентов, представленная в форме условного плана, для решения задачи о плачущем ребенке с несколькими няньками

Так выглядит совместная стратегия $\boldsymbol{\pi} = (\pi^1, \pi^2)$, представленная в виде двухэтапных условных планов для задачи о плачущем ребенке с несколькими няньками:

ря на то что это вычислительно сложная структура, поскольку она учитывает как время, так и глубину, алгоритмы для таких моделей значительно продвинулись в направлении прикладного использования. E. Sonu, Y. Chen, P. Doshi, *Decision-Theoretic Planning Under Anonymity in Agent Populations*, Journal of Artificial Intelligence Research, vol. 59, pp. 725–770, 2017.



Мы можем вычислить совместную функцию полезности $\mathbf{U}^{\boldsymbol{\pi}}$ рекурсивно, подобно тому, как это было сделано в уравнении (20.8) для POMDP при старте из состояния s:

$$\mathbf{U}^{\boldsymbol{\pi}}(s) = \mathbf{R}(s, \boldsymbol{\pi}()) + \gamma \left[\sum_{s'} T(s'|s, \boldsymbol{\pi}()) \sum_{\mathbf{o}} O(\mathbf{o}|\boldsymbol{\pi}()), s') \mathbf{U}^{\boldsymbol{\pi}(\mathbf{o})}(s') \right], \tag{26.1}$$

где $\boldsymbol{\pi}()$ – вектор действий в корне дерева, обозначающего $\boldsymbol{\pi}$, а $\boldsymbol{\pi}(\mathbf{o})$ – вектор подпланов, связанных с различными агентами, наблюдающими свои компоненты совместного наблюдения $\mathbf{o}$.

Полезность, связанная со стратегией $\boldsymbol{\pi}$, следующей из начального распределения по состояниям b, определяется выражением

$$\mathbf{U}^{\boldsymbol{\pi}}(b) = \sum_{s} b(s) \mathbf{U}^{\boldsymbol{\pi}}(s). \tag{26.2}$$

Реализация этого подхода показана в алгоритме 26.2.

Алгоритм 26.2. Условные планы представляют стратегии в POMG с конечным горизонтом. Определение для одного агента было показано в алгоритме 20.1. Мы можем вычислить полезность, связанную с выполнением совместной стратегии `π`, представленной условными планами, при запуске из состояния `s`. Вычисление полезности из начального распределения по состояниям `b` включает получение средневзвешенного значения полезностей при запуске из разных состояний

```
function lookahead(𝒫::POMG, U, s, a)
    𝒮, 𝒪, T, O, R, γ = 𝒫.𝒮, joint(𝒫.𝒪), 𝒫.T, 𝒫.O, 𝒫.R, 𝒫.γ
```

```
    u′ = sum(T(s,a,s′)*sum(O(a,s′,o)*U(o,s′) for o in 𝒪) for s′ in 𝒮)
    return R(s,a) + γ*u′
end

function evaluate_plan(𝒫::POMG, π, s)
    a = Tuple(πi() for πi in π)
    U(o,s′) = evaluate_plan(𝒫, [πi(oi) for (πi, oi) in zip(π,o)], s′)
    return isempty(first(π).subplans) ? 𝒫.R(s,a) : lookahead(𝒫, U, s, a)
end

function utility(𝒫::POMG, b, π)
    u = [evaluate_plan(𝒫, π, s) for s in 𝒫.𝒮]
    return sum(bs * us for (bs, us) in zip(b, u))
end
```

26.2.2. Оценка стохастических контроллеров

Контроллер (раздел 23.1) представлен в виде стохастического графа. Контроллер, соответствующий агенту i, определяется распределением по действиям $\psi^i(a^i|x^i)$ и распределением по последователям $\eta^i(x^{i\prime}|x^i, a^i, o^i)$. Полезность пребывания в состоянии s с активным совместным узлом $\mathbf{x}$ и соблюдением совместной стратегии $\boldsymbol{\pi}$ составляет

$$\mathbf{U}^{\boldsymbol{\pi}}(\mathbf{x}, s) = \sum_{\mathbf{a}} \prod_i \psi^i(a^i|x^i)\Big(\mathbf{R}(s, \mathbf{a}) + \gamma \sum_{s'} T(s'|s, \mathbf{a}) \sum_{\mathbf{o}} O(\mathbf{o}|\mathbf{a}, s') \sum_{\mathbf{x}'} \prod_i \eta^i(x^{i\prime}|x^i, a^i, o^i)\mathbf{U}^{\boldsymbol{\pi}}(\mathbf{x}', s')\Big). \quad (26.3)$$

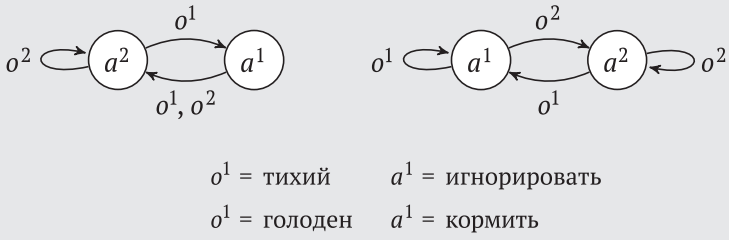
Оценка стратегии в этом контексте подразумевает решение системы линейных уравнений. В качестве альтернативы мы можем использовать итеративную оценку стратегии, аналогичную алгоритму 23.2 для POMDP. Полезность при старте из начального распределения по состояниям b и состояния совместного контроллера $\mathbf{x}$:

$$\mathbf{U}^{\boldsymbol{\pi}}(\mathbf{x}, b) = \sum_s b(s)\mathbf{U}(\mathbf{x}, s). \quad (26.4)$$

В примере 26.3 показан совместный стохастический контроллер.

Пример 26.3. Совместная стратегия двух агентов с использованием контроллеров для решения задачи о плачущем ребенке с несколькими няньками

Так выглядит совместная стратегия $\boldsymbol{\pi} = (\pi^1, \pi^2)$ в форме контроллера для двух нянек в задаче о плачущем ребенке. Каждый контроллер имеет два узла $X^i = \{x_1^i, x_2^i\}$:



26.3. Равновесие Нэша

Как и в случае с простыми и марковскими играми, равновесие Нэша для POMG достигнуто, когда все агенты действуют в соответствии со стратегией наилучшего отклика на действия друг друга, так что ни у одного агента нет стимула отклоняться от своей стратегии. Нахождение равновесия Нэша для POMG, как правило, является невероятно сложным в вычислительном отношении. Алгоритм 26.3 вычисляет d-шаговое равновесие Нэша для POMG. Он перебирает все возможные d-шаговые совместные условные планы для построения простой игры, как показано в примере 26.4. Равновесие Нэша для этой простой игры также является равновесием Нэша для POMG.

В простой игре действуют те же агенты, что и в POMG. Каждому совместному действию в простой игре соответствует совместный условный план в POMG. Вознаграждение, получаемое за каждое действие, эквивалентно полезности по совместному условному плану в POMG. Равновесие Нэша этой сконструированной простой игры можно непосредственно применять как равновесие Нэша для POMG.

Пример 26.4. Вычисление равновесия Нэша для задачи о плачущем ребенке с несколькими нянями путем преобразования ее в простую игру, в которой действия соответствуют условным планам

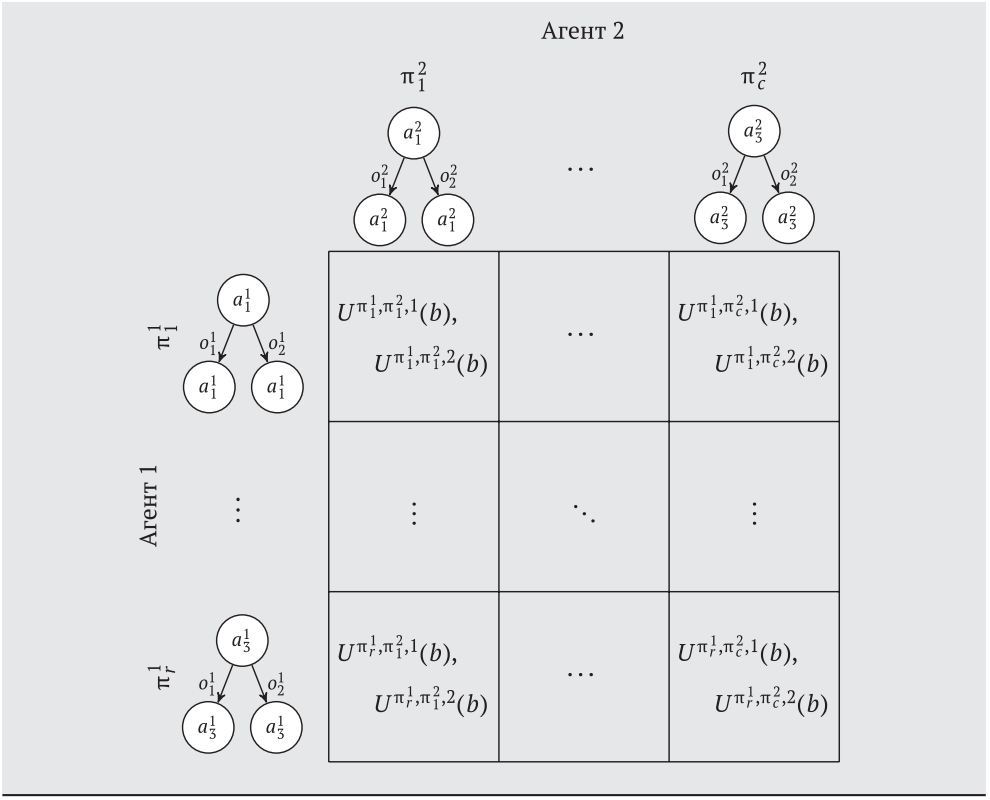
Рассмотрим задачу о плачущем ребенке с несколькими нянями и с горизонтом в два шага. Напомним, что каждому агенту i доступны три действия

$\mathcal{A}^i = \{a^i_1, a^i_2, a^i_3\}$ = {кормить, петь, игнорировать}

и два наблюдения

$\mathcal{O}^i = \{o^i_1, o^i_2\}$ = {плачет, тихий}.

Преобразование этой POMG в простую игру дает нам представленную ниже игровую таблицу. Каждая нянька выбирает действия простой игры, соответствующие законченному условному плану. Вознаграждение простой игры для каждого агента – это полезность, связанная с совместной стратегией.



Алгоритм 26.3. Вычисление равновесия Нэша для POMG $\mathcal{P}$ с начальным распределением по состояниям `b` путем создания простой игры для всех условных планов до некоторой глубины `d`. Мы находим равновесие Нэша в этой простой игре, используя алгоритм 24.5. Для простоты выбираем наиболее вероятную совместную стратегию. В качестве альтернативы можно случайным образом выбрать совместную стратегию при запуске

```
struct POMGNashEquilibrium
    b # начальное убеждение
    d # глубина условных планов
end

function create_conditional_plans(𝒫, d)
    ℐ, 𝒜, 𝒪 = 𝒫.ℐ, 𝒫.𝒜, 𝒫.𝒪
    Π = [[ConditionalPlan(ai) for ai in 𝒜[i]] for i in ℐ]
    for t in 1:d
        Π = expand_conditional_plans(𝒫, Π)
    end
    return Π
end
```

```
function expand_conditional_plans(𝒫, Π)
    ℐ, 𝒜, 𝒪 = 𝒫.ℐ, 𝒫.𝒜, 𝒫.𝒪
    return [[ConditionalPlan(ai, Dict(oi => πi for oi in 𝒪[i]))
        for πi in Π[i] for ai in 𝒜[i]] for i in ℐ]
end

function solve(M::POMGNashEquilibrium, 𝒫::POMG)
    ℐ, γ, b, d = 𝒫.ℐ, 𝒫.γ, M.b, M.d
    Π = create_conditional_plans(𝒫, d)
    U = Dict(π => utility(𝒫, b, π) for π in joint(Π))
    𝒢 = SimpleGame(γ, ℐ, Π, π -> U[π])
    π = solve(NashEquilibrium(), 𝒢)
    return Tuple(argmax(πi.p) for πi in π)
end
```

26.4. Динамическое программирование

Метод, использованный в предыдущем разделе для вычисления равновесия Нэша, обычно чрезвычайно затратен в вычислительном отношении, потому что рассматриваемые действия соответствуют всем возможным условным планам до некоторой глубины. К счастью, в данном случае можно адаптировать метод итерации по полезности для POMDP (раздел 20.5), где мы выполняли итеративное чередование между увеличением глубины набора рассматриваемых условных планов и сокращением неоптимальных планов. Хотя вычислительная сложность в наихудшем случае такая же, как при полном развертывании всех деревьев стратегий, этот поэтапный подход может значительно сэкономить вычислительные ресурсы.

Упомянутый метод динамического программирования реализован в алгоритме 26.4. Он начинается с построения всех одношаговых планов. Мы сокращаем все планы, над которыми доминирует другой план, а затем расширяем все комбинации одношаговых планов, чтобы получить двухшаговые планы. Эта процедура чередования расширения и сокращения повторяется до тех пор, пока не будет достигнут желаемый горизонт.

Алгоритм 26.4. Вычисление равновесия Нэша `π` для POMG $\mathcal{P}$ при заданном исходном убеждении `b` и глубине горизонта `d` при помощи динамического программирования. Алгоритм итеративно вычисляет деревья стратегий и их ожидаемую полезность на каждом шаге. Фаза сокращения на каждой итерации удаляет деревья стратегий, относительно которых существует хотя бы одно дерево стратегии, дающее более высокую ожидаемую полезность

```
struct POMGDynamicProgramming
    b # начальное убеждение
    d # глубина условных планов
end
```

```
function solve(M::POMGDynamicProgramming, 𝒫::POMG)
    ℐ, 𝒮, 𝒜, R, γ, b, d = 𝒫.ℐ, 𝒫.𝒮, 𝒫.𝒜, 𝒫.R, 𝒫.γ, M.b, M.d
    Π = [[ConditionalPlan(ai) for ai in 𝒜[i]] for i in ℐ]
    for t in 1:d
        Π = expand_conditional_plans(𝒫, Π)
        prune_dominated!(Π, 𝒫)
    end
    𝒢 = SimpleGame(γ, ℐ, Π, π -> utility(𝒫, b, π))
    π = solve(NashEquilibrium(), 𝒢)
    return Tuple(argmax(πi.p) for πi in π)
end

function prune_dominated!(Π, 𝒫::POMG)
    done = false
    while !done
        done = true
        for i in shuffle(𝒫.ℐ)
            for πi in shuffle(Π[i])
                if length(Π[i]) > 1 && is_dominated(𝒫, Π, i, πi)
                    filter!(πi′ -> πi′ ≠ πi, Π[i])
                    done = false
                    break
                end
            end
        end
    end
end

function is_dominated(𝒫::POMG, Π, i, πi)
    ℐ, 𝒮 = 𝒫.ℐ, 𝒫.𝒮
    jointΠnoti = joint([Π[j] for j in ℐ if j ≠ i])
    π(πi′, πnoti) = [j==i ? πi′ : πnoti[j>i ? j-1 : j] for j in ℐ]
    Ui = Dict((πi′, πnoti, s) => evaluate_plan(𝒫, π(πi′, πnoti), s)[i]
            for πi′ in Π[i], πnoti in jointΠnoti, s in 𝒮)
    model = Model(Ipopt.Optimizer)
    @variable(model, δ)
    @variable(model, b[jointΠnoti, 𝒮] ≥ 0)
    @objective(model, Max, δ)
    @constraint(model, [πi′=Π[i]],
        sum(b[πnoti, s] * (Ui[πi′, πnoti, s] - Ui[πi, πnoti, s])
        for πnoti in jointΠnoti for s in 𝒮) ≥ δ)
    @constraint(model, sum(b) == 1)
    optimize!(model)
    return value(δ) ≥ 0
end
```

На этапе сокращения удаляют все недостаточно эффективные стратегии. Стратегия π^i, принадлежащая агенту i, подлежит удалению, если существует другая стратегия $\pi^{i\prime}$, которая всегда работает не хуже, чем π^i. Несмотря на большие вычислительные затраты, это условие можно проверить, выполнив

линейную программу. Данный процесс соответствует отсечению узлов контроллера в POMDP (алгоритм 23.4).

Выполнить отдельную линейную программу для каждой возможной комбинации стратегий другого агента $\boldsymbol{\pi}^{-i}$ было бы сложно в вычислительном отношении. Вместо этого мы будем использовать более эффективный подход, который никогда не отбрасывает оптимальную стратегию, но, возможно, не сможет отбросить все неоптимальные политики. Стратегия π^i хуже, чем $\pi^{i\prime}$, если не существует $b(\boldsymbol{\pi}^{-i}, s)$ между другими совместными стратегиями $\boldsymbol{\pi}^{-i}$ и состояниями s, такими что

$$\sum_{\boldsymbol{\pi}^{-i}} \sum_{s} b(\boldsymbol{\pi}^{-i}, s) U^{\pi^{i\prime}, \boldsymbol{\pi}^{-i}, i}(s) \geq \sum_{\boldsymbol{\pi}^{-i}} \sum_{s} b(\boldsymbol{\pi}^{-i}, s) U^{\pi^{i}, \boldsymbol{\pi}^{-i}, i}(s). \tag{26.5}$$

Здесь b – совместное распределение вероятностей по стратегиям других агентов и состояний. Как упоминалось в начале этой главы, как правило, невозможно вычислить убеждения, но уравнение (26.5) проверяет пространство убеждений на доминирование индивидуальной стратегии.

Мы можем построить отдельную линейную программу для проверки условия (26.5)[3]. Если линейная программа выполнима, то это означает, что над стратегией π^i не доминирует никакая другая стратегия $\pi^{i\prime}$:

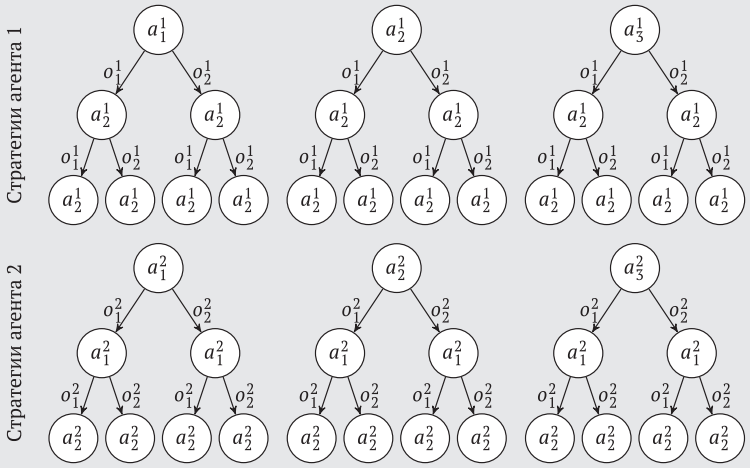
$$\begin{aligned}
&\text{максимизировать}_{\delta, b}\ \delta, \\
&\text{при условии что } b(\boldsymbol{\pi}^{-i}, s) \geq 0 \text{ для всех } \boldsymbol{\pi}^{-i}, s; \\
&\qquad \sum_{\boldsymbol{\pi}^{-i}} \sum_{s} b(\boldsymbol{\pi}^{-i}, s) = 1; \\
&\qquad \sum_{\boldsymbol{\pi}^{-i}} \sum_{s} b(\boldsymbol{\pi}^{-i}, s) \left(U^{\pi^{i\prime}, \boldsymbol{\pi}^{-i}, i}(s) - U^{\pi^{i}, \boldsymbol{\pi}^{-i}, i}(s) \right) \geq \delta \text{ для всех } \pi^{i\prime}.
\end{aligned} \tag{26.6}$$

На этапе сокращения худшие стратегии удаляются путем случайного выбора агента i и проверки доминирования каждой из его стратегий. Этот процесс повторяется до тех пор, пока при обходе всех агентов не перестанет находиться хотя бы одна доминирующая стратегия. В примере 26.5 показан данный процесс применительно к задаче о плачущем ребенке с несколькими няньками.

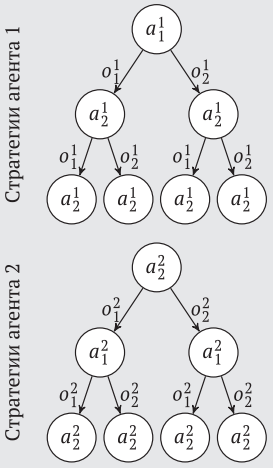
Пример 26.5. Иллюстрация динамического программирования и одного шага сокращения для задачи о плачущем ребенке с несколькими няньками

Рассмотрим решение задачи о плачущем ребенке с несколькими няньками при помощи динамического программирования. Первоначально стратегии на глубине $d = 2$ выглядят так:

[3] Аналогичная линейная программа была использована для сокращения альфа-векторов в POMDP в уравнении (20.16).



После этапа сокращения стратегии агентов приобретают вид:



В данном случае этап сокращения дает наилучшую совместную стратегию. Этот подход значительно уменьшает количество возможных совместных стратегий, которые необходимо учитывать на следующей итерации алгоритма.

26.5. Заключение

- POMG обобщают POMDP для нескольких агентов, а марковские игры – для частичной наблюдаемости.

- Поскольку агенты, как правило, не могут хранить и обрабатывать убеждения в POMG, стратегии обычно принимают форму условных планов или контроллеров с конечным состоянием.
- Равновесия Нэша в форме d-шаговых условных планов для POMG могут быть получены путем нахождения равновесия Нэша для простых игр, совместные действия которых состоят из всех возможных совместных стратегий POMG.
- Методы динамического программирования применяются для эффективного вычисления равновесий Нэша путем итеративного построения наборов более глубоких условных планов с одновременным сокращением менее эффективных планов для ограничения области поиска.

26.6. Упражнения

Упражнение 26.1. Покажите, что POMG обобщает как POMDP, так и марковские игры.

Решение. Для любого POMDP мы можем определить POMG с одним агентом $\mathcal{I} = \{1\}$. Состояния $\mathcal{S}$ идентичны, как и действия $\mathbf{A} = (\mathcal{A}^1)$ и наблюдения $\mathbf{O} = (\mathcal{O}^1)$. Таким образом, из POMDP непосредственно следуют переход состояния, функция наблюдения и вознаграждение POMG. Оптимизация по равновесию Нэша имеет только одного агента, поэтому она приводит к простой максимизации ожидаемой полезности, что идентично POMDP.

Для любой марковской игры мы можем определить POMG с теми же агентами $\mathcal{I}$, состояниями $\mathcal{S}$, совместными действиями $\mathbf{A}$, переходами T и совместными вознаграждениями $\mathbf{R}$. Отдельным наблюдениям назначаются состояния $\mathcal{O}^i = \mathcal{S}$. Затем функция наблюдения детерминистически предоставляет каждому агенту истинное состояние $O(\mathbf{o}|\mathbf{a}, s') = 1$, если $\mathbf{o} = (s', \ldots, s')$, и 0 в противном случае.

Упражнение 26.2. Как мы можем включить связь между агентами в структуру POMG?

Решение. Пространство действий для агентов можно расширить, включив в него коммуникационные действия. Другие агенты могут наблюдать за этими коммуникационными действиями в соответствии со своей моделью наблюдения.

Упражнение 26.3. Всегда ли у агентов есть стимул к общению?

Решение. Агенты в POMG часто конкурируют друг с другом, и в этом случае у них не будет стимула общаться с другими агентами. Если их интересы в какой-то степени совпадают, они могут быть склонны к общению.

Упражнение 26.4. Сколько может быть совместных условных планов для глубины d?

Решение. Напомним, что существует $|\mathcal{A}|^{(|\mathcal{O}|^d-1)/(|\mathcal{O}|-1)}$ возможных d-шаговых одноагентных условных планов. Мы можем построить совместную стратегию условных планов, используя каждую комбинацию этих одноагентных условных планов для всех агентов. Количество d-шаговых мультиагентных условных планов равно

$$\prod_{i \in \mathcal{I}} |\mathcal{A}^i|^{(|\mathcal{O}^i|^d-1)/(|\mathcal{O}^i|-1)}.$$

Упражнение 26.5. Определите наилучший отклик для POMG с точки зрения полезностей $U^{\boldsymbol{\pi},i}$ для агента i. Предложите реализацию итеративного поиска наилучшего отклика для POMG.

Решение. Наилучший отклик π^i агента i на стратегию $\boldsymbol{\pi}^{-i}$ других агентов определяется уравнением (24.2) для начального убеждения b:

$$U^{\pi^i,\boldsymbol{\pi}^{-i},i}(b) \geq U^{\pi^{i\prime},\boldsymbol{\pi}^{-i},i}(b)$$

относительно любой другой стратегии $\pi^{i\prime}$. Для условных планов $U^{\boldsymbol{\pi},i}$ определяется уравнениями (26.1) и (26.2).

Реализация итеративного поиска наилучшего отклика следует из раздела 24.2.1. Сначала можно создать условные планы и простую игру, как в алгоритме 26.3. Затем можно выполнить итерацию по наилучшему отклику, используя алгоритм 24.8.

27 Совместные действия агентов

В реальной жизни часто встречаются *совместные задачи*, когда все агенты действуют независимо в одной среде, работая над достижением общей цели. Области применения совместно действующих агентов варьируются от роботизированных поисково-спасательных служб до межпланетных исследовательских аппаратов. *Децентрализованный частично наблюдаемый марковский процесс принятия решений* (Dec-POMDP) охватывает общие принципы POMG, но сосредоточен на совместных действиях агентов[1]. Эта модель лучше подходит для реализации с помощью масштабируемых приблизительных алгоритмов благодаря единственной общей цели, в отличие от поиска равновесия между несколькими целями отдельных агентов. В этой главе представлена модель Dec-POMDP, выделены ее подклассы и описаны алгоритмы оптимального и приблизительного решений.

27.1. Децентрализованные частично наблюдаемые марковские процессы принятия решений

Dec-POMDP (алгоритм 27.1) – это POMG, в котором все агенты имеют одну и ту же цель. Каждый агент $i \in \mathcal{I}$ выбирает локальное действие $a^i \in \mathcal{A}^i$ на основе истории локальных наблюдений $o^i \in \mathcal{O}^i$. Истинное состояние системы $s \in \mathcal{S}$ распространяется на всех агентов. На основе состояния s и совместного действия $\mathbf{a}$ генерируется единое вознаграждение $R(s, \mathbf{a})$. Цель всех агентов состоит в том, чтобы максимизировать общее ожидаемое вознаграждение с течением времени в условиях частичной локальной наблюдаемости. В примере 27.1 представлена версия задачи «хищник–жертва» для модели Dec-POMDP.

---

[1] D. S. Bernstein, R. Givan, N. Immerman, S. Zilberstein, *The Complexity of Decentralized Control of Markov Decision Processes*, Mathematics of Operation Research, vol. 27, no. 4, pp. 819–840, 2002. Более подробное введение представлено в F. A. Oliehoek, C. Amato, *A Concise Introduction to Decentralized POMDPs*. Springer, 2016.

Алгоритм 27.1. Структура данных для Dec-POMDP. Функция `joint` из алгоритма 24.2 позволяет создавать все комбинации, такие как $\mathcal{A}$ или $\mathcal{O}$. Функция `tensorform` преобразует задачу Dec-POMDP $\mathcal{P}$ в тензорное представление

```
struct DecPOMDP
    γ # коэффициент дисконтирования
    ℐ # агенты
    𝒮 # пространство состояний
    𝒜 # пространство общих состояний
    𝒪 # пространство общих наблюдений
    T # функция перехода
    O # функция совместных наблюдений
    R # функция вознаграждения
end
```

Пример 27.1. Совместная задача «хищник–жертва» в форме Dec-POMDP. Дополнительная информация о задаче доступна в приложении F.15

Рассмотрим задачу гексамира типа «хищник–жертва», в которой команда хищников $\mathcal{I}$ пытается поймать одну убегающую жертву. Хищники передвигаются независимо друг от друга. Жертва случайным образом перемещается в соседнюю ячейку, не занятую хищником. Хищники должны работать вместе, чтобы поймать жертву.

Многие из проблем, свойственных POMG, переходят и в Dec-POMDP, например изначальная неспособность агентов хранить убеждения. Мы сосредоточимся на стратегиях, представленных в виде условных планов или контроллеров. Для оценки стратегий можно использовать алгоритмы, рассмотренные в предыдущей главе. Все, что требуется, – это создать POMG с $R^i(s, \mathbf{a})$ для каждого агента i, равным $R(s, \mathbf{a})$ из Dec-POMDP.

27.2. Подклассы

Известно много различных подклассов Dec-POMDP. Знание этих подклассов полезно при разработке алгоритмов, использующих преимущества их специфической структуры.

Отдельного внимания заслуживает такой аспект, как *совместная полная наблюдаемость* (joint full observability), когда каждый агент наблюдает за аспектом состояния, так что, если они объединят свои наблюдения, это однозначно выявит истинное состояние. Агенты, однако, не делятся своими наблюдениями. Это свойство гарантирует, что если $O(\mathbf{o}|\mathbf{a}, s') > 0$, то $P(s'|\mathbf{o}) = 1$. Dec-POMDP с совместной полной наблюдаемостью называется *децентрализованным мар-*

ковским процессом принятия решений (Dec-MDP). Оба процесса – Dec-POMDP и Dec-MDP – являются NEXP-полными, когда количество шагов в горизонте меньше, чем количество состояний[2].

Во многих случаях пространство состояний Dec-POMDP *факторизовано* (разложено на компоненты): по одному для каждого агента и одно для среды. Задачи такого типа называются *факторизованными Dec-POMDP*. У нас есть $\mathcal{S} = \mathcal{S}^0 \times \mathcal{S}^1 \times \mathcal{S}^k$, где $\mathcal{S}^i$ – компонент факторизованного состояния, связанный с агентом i, а $\mathcal{S}^0$ – компонент факторизованного состояния, связанный с общей средой. Например, в совместной задаче «хищник–жертва» каждый агент имеет свой собственный фактор состояния для своего местоположения, а положение жертвы связано с фактором среды в пространстве состояний.

В ряде случаев факторизованный Dec-POMDP может иметь одно или несколько из следующих свойств:

- *независимость перехода* (transition independence), когда агенты не могут влиять на состояние друг друга:

$$T(\mathbf{s}'|\mathbf{s},\mathbf{a}) = T^0(s^{0\prime}|s^0)\prod_i T^i(s^{i\prime}|s^i,a^i); \tag{27.1}$$

- *независимость наблюдения* (observation independence), когда наблюдения агентов зависят только от их локального состояния и действий:

$$O(\mathbf{o}|\mathbf{a},\mathbf{s}') = \prod_i O^i(o^i|a^i,s^{i\prime}); \tag{27.2}$$

- *независимость вознаграждения* (reward independence), когда вознаграждение может быть разделено на несколько независимых частей[3]:

$$R(\mathbf{s},\mathbf{a}) = R^0(s^0) + \sum_i R^i(s^i,a^i). \tag{27.3}$$

Вычислительная сложность может значительно различаться в зависимости от того, какое из этих свойств независимости выполняется, как показано в табл. 27.1. При моделировании задачи важно учитывать эти независимости для улучшения масштабируемости.

Сетевой распределенный частично наблюдаемый марковский процесс принятия решений (ND-POMDP) представляет собой Dec-POMDP с независимостью перехода и наблюдения и специальной структурой вознаграждения, представ-

---

[2] В отличие от классов сложности NP и PSPACE, известно, что NEXP не входит в P. Следовательно, мы можем доказать, что Dec-MDP и Dec-POMDP не решаемы при помощи алгоритмов с полиномиальным временем. D. S. Bernstein, R. Givan, N. Immerman, S. Zilberstein, *The Complexity of Decentralized Control of Markov Decision Processes*, Mathematics of Operation Research, vol. 27, no. 4, pp. 819–840, 2002.

[3] Здесь мы показываем комбинацию компонентов вознаграждения как сумму, но вместо этого можно использовать любую монотонно неубывающую функцию и сохранить независимость вознаграждения.

ленной в виде координационного графа. В отличие от графов, использованных ранее в этой книге, *координационный граф* (coordination graph) – это тип гиперграфа, который позволяет ребрам соединять любое количество узлов. Узлы в гиперграфе ND-POMDP соответствуют различным агентам. Ребра обозначают взаимодействия между агентами в функции вознаграждения. ND-POMDP связывает с каждым ребром j в гиперграфе компонент вознаграждения R_j, который зависит от компонентов состояния и действия, с которыми соединяется ребро. Функция вознаграждения в ND-POMDP представляет собой просто сумму компонентов вознаграждения, связанных с ребрами. На рис. 27.1 показан координационный граф, образующий функцию вознаграждения, которую можно разложить следующим образом:

$$R_{123}(s_1, s_2, s_3, a_1, a_2, a_3) + R_{34}(s_3, s_4, a_3, a_4) + R_5(s_5, a_5). \tag{27.4}$$

Таблица 27.1. Сложность факторизованных Dec-POMDP с различными предположениями о независимости

Независимость	Сложность
Переходы, наблюдения, вознаграждения	P-полная
Переходы и наблюдения	NP-полная
Любые другие подмножества	NEXP-полная

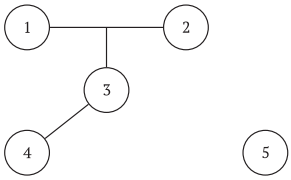


Рис. 27.1. Структура ND-POMDP с пятью агентами. В ней есть три гиперребра: одно с участием агентов 1, 2 и 3, другое с участием агентов 3 и 4, и еще одно для агента 5, который действует сам по себе (поэтому ребро не изображено)

Сети датчиков и задачи отслеживания целей часто оформляют как ND-POMDP.

Модель ND-POMDP аналогична модели Dec-MDP, независимой от перехода и наблюдения, но в ней не делается допущение о совместной полной наблюдаемости. Даже если все наблюдения станут общими, истинное состояние мира может быть неизвестно. Кроме того, даже с факторизованными переходами и наблюдениями стратегия ND-POMDP представляет собой сопоставление истории наблюдения с действиями, в отличие от случая перехода и наблюдения Dec-MDP, когда стратегии представляют собой сопоставление локальных состояний с действиями. Сложность в наихудшем случае остается такой же, как и для Dec-POMDP, но алгоритмы для ND-POMDP обычно гораздо более масштабируемы по количеству агентов. Масштабируемость может увеличиваться по мере того, как координационный граф становится менее связным.

Если агенты могут без ущерба для себя сообщать о своих действиях и наблюдениях, то они способны формировать и поддерживать состояние коллективного убеждения. Такая модель называется *многоагентным MDP* (MMDP), или *многоагентным POMDP* (MPOMDP). Сценарии MMDP и MPOMDP также могут возникать, когда существует независимость перехода, наблюдения и возна-

граждения. Для решения этих задач может применяться любой алгоритм MDP или POMDP, обсуждавшийся в предыдущих главах.

В табл. 27.2 приведены некоторые из этих подклассов. На рис. 27.2 показаны отношения между моделями, обсуждаемыми в данной книге.

Таблица 27.2. Подклассы Dec-POMDP классифицируются по типу и вычислительной сложности. Наблюдаемость здесь означает степени наблюдаемости общего состояния. Степень коммуникации означает, могут ли сотрудничающие агенты свободно обмениваться всеми наблюдениями друг с другом. Свободная коммуникация происходит вне модели (например, высокоскоростное беспроводное соединение у роботов). Общая коммуникация – это когда у агентов нет такой возможности и они должны общаться (как правило, несовершенно) посредством своих действий

Агенты	Наблюдаемость	Коммуникация	Модель
Один	Полная	—	MDP
Один	Частичная	—	POMDP
Много	Полная	Свободная	MMDP
Много	Полная	Общая	MMDP
Много	Совместно полная	Свободная	MMDP
Много	Совместно полная	Общая	Dec-MDP
Много	Частичная	Свободная	MPOMDP
Много	Частичная	Общая	Dec-POMDP

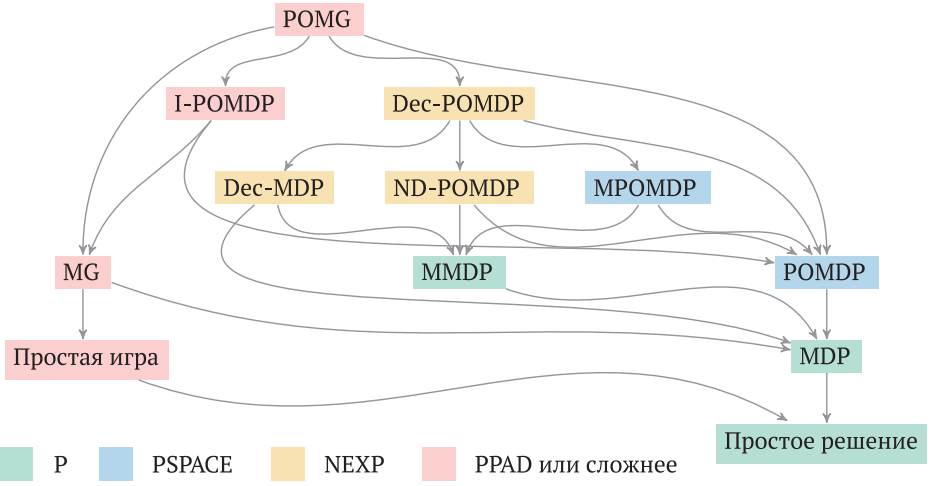


Рис. 27.2. Таксономия моделей, обсуждаемых в этой книге. Стрелками показано обобщение потомков родителями. Например, модель Dec-POMDP включает в себя POMDP путем поддержки нескольких агентов. Цвет узлов указывает на вычислительную сложность, как указано в нижней части рисунка. Обозначенные здесь сложности относятся к общей модели, стратегии и формулировке цели, представленным в книге. Более подробно об этом сказано в C. Papadimitriou, J. Tsitsiklis, *The Complexity of Markov Decision Processes*, Mathematics of Operation Research, vol. 12, no. 3, pp. 441–450, 1987. Также см. S. Seuken, S.Zilberstein, *Formal Models and Algorithms for Decentralized Decision Making Under Uncertainty*, Autonomous Agents and Multi-Agent Systems, vol. 17, no. 2, pp. 190–250, 2008

27.3. Динамическое программирование

Алгоритм динамического программирования для задачи Dec-POMDP применяет оператор Беллмана на каждом этапе и сокращает «отстающие» стратегии. Этот процесс идентичен динамическому программированию для POMG, за исключением того, что каждый агент получает одинаковое вознаграждение. Процедура реализована в алгоритме 27.2.

Алгоритм 27.2. Метод динамического программирования вычисляет оптимальную совместную стратегию π для задачи Dec-POMDP $\mathcal{P}$, учитывая начальное убеждение `b` и глубину горизонта `d`. Мы можем напрямую использовать алгоритм POMG, поскольку Dec-POMDP – это особый совместный класс POMG

```
struct DecPOMDPDynamicProgramming
    b # начальное убеждение
    d # глубина условных планов
end

function solve(M::DecPOMDPDynamicProgramming, 𝒫::DecPOMDP)
    ℐ, 𝒮, 𝒜, 𝒪, T, O, R, γ = 𝒫.ℐ, 𝒫.𝒮, 𝒫.𝒜, 𝒫.𝒪, 𝒫.T, 𝒫.O, 𝒫.R, 𝒫.γ
    R′(s, a) = [R(s, a) for i in ℐ]
    𝒫′ = POMG(γ, ℐ, 𝒮, 𝒜, 𝒪, T, O, R′)
    M′ = POMGDynamicProgramming(M.b, M.d)
    return solve(M′, 𝒫′)
end
```

27.4. Итерация по наилучшим откликам

Вместо того чтобы напрямую исследовать совместные стратегии, мы можем применить разновидность *итерации по наилучшим откликам* (iterated best response, алгоритм 27.3). В соответствии с этим алгоритмом мы итеративно выбираем агента и вычисляем стратегию наилучшего отклика, предполагая, что другие агенты следуют фиксированной стратегии[4]. Этот приблизительный алгоритм обычно работает быстро, потому что он выбирает наилучшую стратегию только для одного агента за раз. Более того, поскольку все агенты получают одно и то же вознаграждение, обычно он завершается после относительно небольшого количества итераций.

---

[4] Этот тип алгоритма также называется поиском стратегий на основе совместного равновесия (JESP). R. Nair, M. Tambe, M. Yokoo, D. Pynadath, S. Marsella, *Taming Decentralized POMDPs: Towards Efficient Policy Computation for Multiagent Settings*, in International Joint Conference on Artificial Intelligence (IJCAI), 2003. Он может быть улучшен путем динамического программирования.

Алгоритм 27.3. Метод итерации по наилучшим откликам для совместной задачи Dec-POMPDP $\mathcal{P}$ выполняет поиск детерминированного наилучшего отклика для каждого агента, чтобы быстро найти пространство стратегий условного плана. Функция `solve` выполняет эту процедуру до `k_max` шагов, максимизируя полезность при начальном убеждении `b` для условных планов глубины `d`

```
struct DecPOMDPIteratedBestResponse
    b # начальное убеждение
    d # глубина условных планов
    k_max # количество итераций
end

function solve(M::DecPOMDPIteratedBestResponse, 𝒫::DecPOMDP)
    ℐ, 𝒮, 𝒜, 𝒪, T, O, R, γ = 𝒫.ℐ, 𝒫.𝒮, 𝒫.𝒜, 𝒫.𝒪, 𝒫.T, 𝒫.O, 𝒫.R, 𝒫.γ
    b, d, k_max = M.b, M.d, M.k_max
    R′(s, a) = [R(s, a) for i in ℐ]
    𝒫′ = POMG(γ, ℐ, 𝒮, 𝒜, 𝒪, T, O, R′)
    Π = create_conditional_plans(𝒫, d)
    π = [rand(Π[i]) for i in ℐ]
    for k in 1:k_max
        for i in shuffle(ℐ)
            π′(πi) = Tuple(j == i ? πi : π[j] for j in ℐ)
            Ui(πi) = utility(𝒫′, b, π′(πi))[i]
            π[i] = argmax(Ui, Π[i])
        end
    end
    return Tuple(π)
end
```

Итерация по наилучшим откликам начинается со случайной начальной совместной стратегии $\boldsymbol{\pi}_1$. Процесс случайным образом повторяет агенты. Если выбран агент i, его стратегия π^i обновляется с учетом наилучшего отклика на фиксированные стратегии $\boldsymbol{\pi}^{-i}$ других агентов с начальным распределением по убеждениям b:

$$\pi^i \leftarrow \arg\max_{\pi^{i'}} U^{\pi^{i'},\boldsymbol{\pi}^{-i}}(b) \tag{27.5}$$

с зависимостями в пользу текущей стратегии. Процесс завершается, когда ни один агент не может изменить свою стратегию.

Хотя этот алгоритм работает быстро и гарантированно сходится, он не всегда находит наилучшую совместную стратегию. Он основан на итерации по наилучшему отклику для нахождения равновесия Нэша, но дело в том, что могут существовать несколько равновесий Нэша, которым соответствуют различные полезности. Данный алгоритм найдет только одно из них.

27.5. Эвристический поиск

Вместо расширения всех совместных стратегий *эвристический поиск* (алгоритм 27.4) исследует фиксированное количество стратегий[5], которые сохраняются во время итерирования, предотвращая экспоненциальный рост объема вычислений. Эвристическое исследование направляет поиск, пытаясь расширить наилучшие совместные стратегии только до тех пор, пока не будет достигнута глубина d.

Алгоритм 27.4. Эвристический поиск с ограниченной памятью использует эвристическую функцию для поиска в пространстве условных планов задачи Dec-POMDP $\mathcal{P}$. Функция `solve` пытается максимизировать полезность при начальном убеждении `b` для совместных условных планов глубины `d`. Функция `explore` генерирует убеждение на `t` шагов в будущее путем применения случайных действий и моделирования действий и наблюдений. Алгоритм обладает ограниченной памятью, сохраняя только `n_max` условных планов на агента

```
struct DecPOMDPHeuristicSearch
    b # начальное убеждение
    d # глубина условных планов
    n_max # количество стратегий
end

function solve(M::DecPOMDPHeuristicSearch, 𝒫::DecPOMDP)
    ℐ, 𝒮, 𝒜, 𝒪, T, O, R, γ = 𝒫.ℐ, 𝒫.𝒮, 𝒫.𝒜, 𝒫.𝒪, 𝒫.T, 𝒫.O, 𝒫.R, 𝒫.γ
    b, d, n_max = M.b, M.d, M.n_max
    R′(s, a) = [R(s, a) for i in ℐ]
    𝒫′ = POMG(γ, ℐ, 𝒮, 𝒜, 𝒪, T, O, R′)
    Π = [[ConditionalPlan(ai) for ai in 𝒜[i]] for i in ℐ]
    for t in 1:d
        allΠ = expand_conditional_plans(𝒫, Π)
        Π = [[] for i in ℐ]
        for z in 1:n_max
            b′ = explore(M, 𝒫, t)
            π = argmax(π -> first(utility(𝒫′, b′, π)), joint(allΠ))
            for i in ℐ
                push!(Π[i], π[i])
                filter!(πi -> πi != π[i], allΠ[i])
            end
        end
    end
```

---

[5] Этот подход также известен как *динамическое программирование с ограничением памяти* (MBDP). S. Seuken and S. Zilberstein, *Memory-Bounded Dynamic Programming for Dec-POMDPs*, in International Joint Conference on Artificial Intelligence (IJCAI), 2007. Существуют и другие алгоритмы эвристического поиска, такие как *мультиагентный* A* (MMA*). D. Szer, F. Charpillet, S. Zilberstein, *MAA*: A Heuristic Search Algorithm for Solving Decentralized POMDPs*, in Conference on Uncertainty in Artificial Intelligence (UAI), 2005.

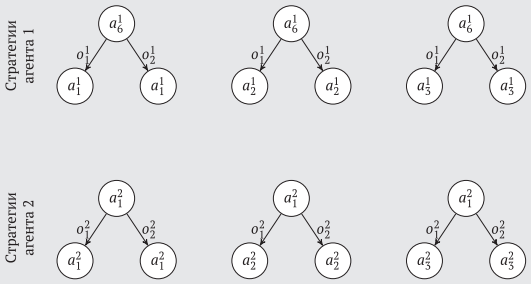
```
    return argmax(π -> first(utility(𝒫′, b, π)), joint(Π))
end

function explore(M::DecPOMDPHeuristicSearch, 𝒫::DecPOMDP, t)
    ℐ, 𝒮, 𝒜, 𝒪, T, O, R, γ = 𝒫.ℐ, 𝒫.𝒮, 𝒫.𝒜, 𝒫.𝒪, 𝒫.T, 𝒫.O, 𝒫.R, 𝒫.γ
    b = copy(M.b)
    b′ = similar(b)
    s = rand(SetCategorical(𝒮, b))
    for τ in 1:t
        a = Tuple(rand(𝒜i) for 𝒜i in 𝒜)
        s′ = rand(SetCategorical(𝒮, [T(s,a,s′) for s′ in 𝒮]))
        o = rand(SetCategorical(joint(𝒪), [O(a,s′,o) for o in joint(𝒪)]))
        for (i′, s′) in enumerate(𝒮)
            po = O(a, s′, o)
            b′[i′] = po*sum(T(s,a,s′)*b[i] for (i,s) in enumerate(𝒮))
        end
        normalize!(b′, 1)
        b, s = b′, s′
    end
    return b′
end
```

Каждая итерация k алгоритма сохраняет набор совместных стратегий $\mathbf{\Pi}_k$. Этот набор изначально состоит из всех одношаговых условных планов. Последующие итерации начинаются с полного расширения условных планов. Цель состоит в том, чтобы добавить их фиксированное количество для следующей итерации.

Мы отдаем приоритет стратегиям, которые с большей вероятностью максимизируют полезность при выборе условных планов для добавления в набор. Однако, поскольку мы расширяем условные планы снизу вверх, то не можем просто оценивать стратегии из исходного состояния убеждения b. Вместо этого нам нужна оценка убеждения на $d - k$ шагов в будущее, которую мы вычисляем, предпринимая случайные действия и моделируя переходы состояний и наблюдения и обновляя убеждение по ходу дела. Это убеждение на итерации k обозначается как b_k. Чтобы найти совместную стратегию с максимальной полезностью, которую нужно добавить в набор, для каждой доступной совместной стратегии $\boldsymbol{\pi} \in \mathbf{\Pi}_k$ проверяется полезность $U^{\pi}(b_k)$. Этот процесс демонстрирует пример 27.2.

Пример 27.2. Исследование путем эвристического поиска и расширение условного плана для совместной задачи гексамира типа «хищник–жертва», показанной справа. Хищники красные и зеленые, добыча синяя

Рассмотрим алгоритм решения этой задачи. Применим эвристический поиск на глубину $d = 3$ с сохранением трех стратегий на каждой итерации. После итерации $k = 1$ стратегии таковы:



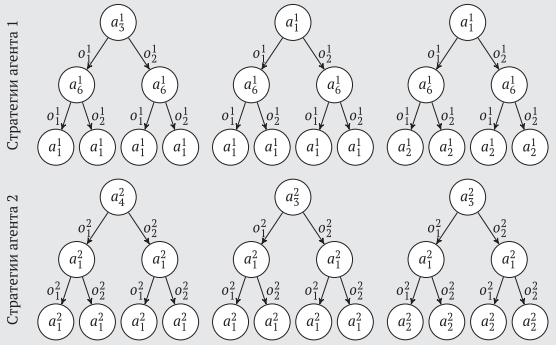
На следующей итерации $k = 2$ эвристический поиск вновь начинается с начального убеждения и занимает $d - k = 3 - 2 = 1$ шаг после эвристического исследования. Исследуемые убеждения, используемые для выбора следующих трех условных планов:

$b_1 = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.17$
$0.0, 0.03, 0.01, 0.0, 0.0, 0.05, 0.0$
$0.01, 0.23, 0.0, 0.08, 0.01, 0.0, 0.0$
$0.14, 0.0, 0.03, 0.22, 0.0, 0.01]$

$b_2 = [0.0, 0.21, 0.03, 0.0, 0.04, 0.01, 0.0$
$0.05, 0.01, 0.0, 0.08, 0.03, 0.0, 0.0$
$0.01, 0.0, 0.0, 0.01, 0.08, 0.34, 0.03$
$0.02, 0.05, 0.01, 0.0, 0.01, 0.0]$

$b_3 = [0.0, 0.03, 0.01, 0.0, 0.03, 0.01, 0.0$
$0.15, 0.05, 0.0, 0.01, 0.0, 0.0, 0.0$
$0.0, 0.0, 0.0, 0.03, 0.06, 0.11, 0.32$
$0.06, 0.03, 0.01, 0.01, 0.04, 0.06]$

Стратегии после итерации $k = 2$:



Убеждения были использованы для определения действий корневого узла и двух поддеревьев под ним. Эти поддеревья строятся из деревьев предыдущей итерации.

27.6. Нелинейное программирование

Для поиска оптимального представления стратегии совместного контроллера фиксированного размера можно использовать нелинейное программирование (алгоритм 27.5)[6]. Этот метод обобщает подход нелинейного программирования для задачи POMDP из раздела 23.3.

Если даны фиксированный набор узлов X^i для каждого агента i, начальное убеждение b и начальные совместные узлы x_1, задача оптимизации представлена условиями

$$\underset{U,\boldsymbol{\psi},\boldsymbol{\eta}}{\text{максимизировать}} \sum_s b(s)U(\mathbf{x}_1, s),$$

при условии что

$$\begin{aligned}
&U(\mathbf{x}, s) = \sum_{\mathbf{a}} \prod_i \psi^i(a^i|x^i) \\
&\qquad \times \left(R(s, \mathbf{a}) + \gamma \sum_{s'} T(s'|s, \mathbf{a}) \sum_{\mathbf{o}} O(\mathbf{o}|\mathbf{a}, s') \sum_{\mathbf{x}'} \prod_i \eta^i(x^{i\prime}|x^i, a^i, o^i) U(\mathbf{x}', s') \right) \\
&\qquad \text{для всех } \mathbf{x}, s; \\
&\psi^i(a^i|x^i) \geq 0 \text{ для всех } i, x^i, a^i; \\
&\sum_{x^{i\prime}} \psi^i(a^i|x^i) = 1 \text{ для всех } i, x^i; \\
&\eta^i(x^{i\prime}|x^i, a^i, o^i) \geq 0 \text{ для всех } i, x^i, a^i, o^i, x^{i\prime}; \\
&\sum_a \eta^i(x^{i\prime}|x^i, a^i, o^i) = 1 \text{ для всех } i, x^i, a^i, o^i.
\end{aligned} \tag{27.6}$$

Алгоритм 27.5. В соответствии с методом нелинейного программирования вычисляем оптимальную стратегию совместного контроллера `π` для Dec-POMDP $\mathcal{P}$, учитывая начальное убеждение `b` и количество узлов контроллера `ℓ` для каждого агента. Фактически это расширение алгоритма 23.5 на задачу Dec-POMDP

```
struct DecPOMDPNonlinearProgramming
    b # начальное убеждение
    ℓ # количество узлов для каждого агента
end

function tensorform(𝒫::DecPOMDP)
    ℐ, 𝒮, 𝒜, 𝒪, R, T, O = 𝒫.ℐ, 𝒫.𝒮, 𝒫.𝒜, 𝒫.𝒪, 𝒫.R, 𝒫.T, 𝒫.O
    ℐ′ = eachindex(ℐ)
```

[6] C. Amato, D. S. Bernstein, S. Zilberstein, *Optimizing Fixed-Size Stochastic Controllers for POMDPs and Decentralized POMDPs*, Autonomous Agents and Multi-Agent Systems, vol. 21, no. 3, pp. 293–320, 2010.

```
    𝒮′ = eachindex(𝒮)
    𝒜′ = [eachindex(𝒜i) for 𝒜i in 𝒜]
    𝒪′ = [eachindex(𝒪i) for 𝒪i in 𝒪]
    R′ = [R(s,a) for s in 𝒮, a in joint(𝒜)]
    T′ = [T(s,a,s′) for s in 𝒮, a in joint(𝒜), s′ in 𝒮]
    O′ = [O(a,s′,o) for a in joint(𝒜), s′ in 𝒮, o in joint(𝒪)]
    return ℐ′, 𝒮′, 𝒜′, 𝒪′, R′, T′, O′
end

function solve(M::DecPOMDPNonlinearProgramming, 𝒫::DecPOMDP)
    𝒫, γ, b = 𝒫, 𝒫.γ, M.b
    ℐ, 𝒮, 𝒜, 𝒪, R, T, O = tensorform(𝒫)
    X = [collect(1:M.ℓ) for i in ℐ]
    jointX, joint𝒜, joint𝒪 = joint(X), joint(𝒜), joint(𝒪)
    x1 = jointX[1]
    model = Model(Ipopt.Optimizer)
    @variable(model, U[jointX,𝒮])
    @variable(model, ψ[i=ℐ,X[i],𝒜[i]] ≥ 0)
    @variable(model, η[i=ℐ,X[i],𝒜[i],𝒪[i],X[i]] ≥ 0)
    @objective(model, Max, b⋅U[x1,:])
    @NLconstraint(model, [x=jointX,s=𝒮],
        U[x,s] == (sum(prod(ψ[i,x[i],a[i]] for i in ℐ)
                    *(R[s,y] + γ*sum(T[s,y,s′]*sum(O[y,s′,z]
                        *sum(prod(η[i,x[i],a[i],o[i],x′[i]] for i in ℐ)
                            *U[x′,s′] for x′ in jointX)
                        for (z, o) in enumerate(joint𝒪)) for s′ in 𝒮))
                    for (y, a) in enumerate(joint𝒜))))
    @constraint(model, [i=ℐ,xi=X[i]],
                sum(ψ[i,xi,ai] for ai in 𝒜[i]) == 1)
    @constraint(model, [i=ℐ,xi=X[i],ai=𝒜[i],oi=𝒪[i]],
                sum(η[i,xi,ai,oi,xi′] for xi′ in X[i]) == 1)
    optimize!(model)
    ψ′, η′ = value.(ψ), value.(η)
    return [ControllerPolicy(𝒫, X[i],
            Dict((xi,𝒫.𝒜[i][ai]) => ψ′[i,xi,ai]
                   for xi in X[i], ai in 𝒜[i]),
            Dict((xi,𝒫.𝒜[i][ai],𝒫.𝒪[i][oi],xi′) => η′[i,xi,ai,oi,xi′]
                    for xi in X[i], ai in 𝒜[i], oi in 𝒪[i], xi′ in X[i]))
        for i in ℐ]
end
```

27.7. Заключение

- Задачи Dec-POMDP – это полностью совместные POMG, моделирующие группу агентов, работающих вместе для достижения общей цели, каждый из которых действует индивидуально, используя только локальную информацию.

- Поскольку определить состояние доверия невозможно, как и в POMG, стратегии в Dec-POMDP обычно представляют в виде условных планов или контроллеров, что позволяет каждому агенту сопоставлять отдельные последовательности наблюдений с отдельными действиями.
- Существует множество подклассов Dec-POMDP с различной степенью вычислительной сложности.
- Метод динамического программирования итеративно вычисляет функцию полезности, сокращая менее удачные стратегии по мере итерации с использованием линейной программы.
- Метод итерации по наилучшему отклику вычисляет стратегию наилучшего отклика с максимальной полезностью для одного агента за раз, итеративно сходясь к совместному равновесию.
- Эвристический поиск ищет фиксированное подмножество стратегий на каждой итерации, руководствуясь эвристикой.
- Для создания контроллеров фиксированного размера можно использовать метод нелинейного программирования.

27.8. Упражнения

Упражнение 27.1. Почему Dec-MDP с полной совместной наблюдаемостью отличается от сценария агентов, знающих состояние?

Решение. Полная совместная наблюдаемость означает, что если агенты делятся своими индивидуальными наблюдениями, то команда знает истинное состояние. Это можно сделать в офлайн-режиме во время планирования. Таким образом, в Dec-MDP истинное состояние, по существу, известно во время планирования. Проблема в том, что для этого требуется, чтобы агенты делились своими индивидуальными наблюдениями, а это невозможно сделать в онлайн-режиме во время выполнения. Следовательно, при планировании по-прежнему необходимо учитывать неопределенные наблюдения, сделанные другими агентами.

Упражнение 27.2. Предложите быстрый алгоритм для Dec-MDP с независимостью перехода, наблюдения и вознаграждения. Докажите, что это правильный выбор.

Решение. Если факторизованный Dec-MDP удовлетворяет всем трем предположениям о независимости, то мы можем решить его как $|\mathcal{I}|$ отдельных MDP. Результирующая стратегия π^i для MDP каждого агента i может затем быть объединена для формирования оптимальной совместной стратегии. Чтобы доказать этот факт, рассмотрим полезность отдельного MDP каждого агента:

$$U^{\pi^i}(s^i) = R(s^i, \pi^i()) + \gamma \left[\sum_{s^{i\prime}} T^i(s^{i\prime} | s^i, \pi^i()) \sum_{o^i} O^i(o^i | \pi^i(), s^{i\prime}) U^{\pi^i(o^i)}(s^{i\prime}) \right].$$

Как и в уравнении (26.1), $\pi^i()$ обозначает корневое действие условного плана i, а $\pi^i(o^i)$ обозначает подпланы i после наблюдения за o^i. Просуммируем каждый из их индивидуальных вкладов следующим образом:

$$\sum_i U^{\pi^i}(s) = \sum_i \left[R(s^i, \pi^i()) + \gamma \left[\sum_{s^{i\prime}} T^i(s^{i\prime} | s^i, \pi^i()) \sum_{o^i} O^i(o^i | \pi^i(), s^{i\prime}) U^{\pi^i(o^i)}(s^{i\prime}) \right] \right].$$

Мы можем объединить T^i и O^i в одно распределение вероятностей P, перенести сумму и применить определение независимости вознаграждения:

$$\begin{aligned}
\sum_i U^{\pi^i}(s) &= \sum_i \left[R(s^i, \pi^i()) + \gamma \left[\sum_{s^{i\prime}} P(s^{i\prime} | s^i, \pi^i()) \sum_{o^i} P(o^i | \pi^i(), s^{i\prime}) U^{\pi^i(o^i)}(s^{i\prime}) \right] \right] \\
&= \sum_i R(s^i, \pi^i()) + \sum_i \left[\gamma \left[\sum_{s^{i\prime}} P(s^{i\prime} | s^i, \pi^i()) \sum_{o^i} P(o^i | \pi^i(), s^{i\prime}) U^{\pi^i(o^i)}(s^{i\prime}) \right] \right] \\
&= R(s, \pi()) + \sum_i \left[\gamma \left[\sum_{s^{i\prime}} P(s^{i\prime} | s^i, \pi^i()) \sum_{o^i} P(o^i | \pi^i(), s^{i\prime}) U^{\pi^i(o^i)}(s^{i\prime}) \right] \right].
\end{aligned}$$

Теперь выполним маргинализацию по всем преемникам s и наблюдениям o. Из-за независимости перехода и наблюдения мы можем свободно обусловливать распределения этими другими не i-состояниями и факторами наблюдения, что аналогично обусловливанию s и o. Затем применим определение независимости перехода и наблюдения. Наконец, перенесем суммирование и обозначим результат как $U^{\pi}(s)$:

$$\begin{aligned}
\sum_i U^{\pi^i}(s) &= R(s, \boldsymbol{\pi}()) + \sum_i \left[\gamma \left[\sum_{s'} P(s' | s^i, \pi^i()) \sum_o P(o | \pi^i(), s^{i\prime}) U^{\pi^i(o^i)}(s^{i\prime}) \right] \right] \\
&= R(s, \boldsymbol{\pi}()) + \sum_i \left[\gamma \left[\sum_{s'} P(s^{0\prime} | s^0) \prod_i P(s^{i\prime} | s^i, \pi^i()) \sum_o \prod_j P(o^j | \pi^i(), s^{i\prime}) U^{\pi^i(o^i)}(s^{i\prime}) \right] \right] \\
&= R(s, \boldsymbol{\pi}()) + \sum_i \left[\gamma \left[\sum_{s'} P(s^{0\prime} | s^0) \prod_i P(s^{i\prime} | s, \pi()) \sum_o \prod_j P(o^j | \pi(), s') U^{\pi^i(o^i)}(s^{i\prime}) \right] \right] \\
&= R(s, \boldsymbol{\pi}()) + \sum_i \left[\gamma \left[\sum_{s'} T(s' | s, \boldsymbol{\pi}()) \sum_o O(o | \boldsymbol{\pi}(), s') U^{\pi^i(o^i)}(s^{i\prime}) \right] \right] \\
&= R(s, \boldsymbol{\pi}()) + \gamma \left[\sum_{s'} T(s' | s, \boldsymbol{\pi}()) \sum_o O(o | \boldsymbol{\pi}(), s') \left[\sum_i U^{\pi^i(o^i)}(s^{i\prime}) \right] \right] \\
&= R(s, \boldsymbol{\pi}()) + \gamma \left[\sum_{s'} T(s' | s, \boldsymbol{\pi}()) \sum_o O(o | \boldsymbol{\pi}(), s') U^{\boldsymbol{\pi}(o)}(s') \right] \\
&= U^{\boldsymbol{\pi}}(s).
\end{aligned}$$

Это функция полезности Dec-MDP, полученная из уравнения (26.1), что и требовалось доказать.

Упражнение 27.3. Как мы можем использовать MMDP или MPOMDP в качестве эвристики в эвристическом поиске Dec-POMDP?

Решение. Предположим наличие свободной коммуникации между агентами в целях планирования. На каждом временном шаге t все агенты знают $\mathbf{a}_t$ и $\mathbf{o}_t$, что позволяет иметь многоагентное убеждение b_t и приводит к MPOMDP. Решение MPOMDP можно использовать в качестве эвристики для поиска деревьев стратегий. В качестве альтернативы создадим эвристику, где предположим, что истинное состояние и совместные действия известны. Это приводит к задаче MMDP, решение которой также можно использовать в качестве эвристики. Эти предположения используются только для планирования. Выполнение по-прежнему представляет собой Dec-POMDP, в котором агенты получают индивидуальные наблюдения без свободной коммуникации. Результатом любой эвристики является совместная стратегия $\hat{\boldsymbol{\pi}}$ для эвристического исследования.

Упражнение 27.4. Как можно вычислить контроллер наилучшего отклика? Покажите, как вашу рекомендацию можно использовать в итерации по наилучшему отклику.

Решение. Для агента i контроллер наилучшего отклика X^i, ψ^i и η^i можно вычислить при помощи нелинейной программы. Она аналогична приведенной в разделе 27.6, за исключением того, что $\mathbf{X}^{-i}$, $\boldsymbol{\psi}^{-i}$ и $\boldsymbol{\eta}^{-i}$ теперь строго заданы и больше не являются переменными:

$$\text{максимизировать} \underset{U, \psi^i, \eta^i}{\sum} b(s) U(\mathbf{x}_1, s),$$

при условии что

$$U(\mathbf{x}, s) = \sum_{\mathbf{a}} \prod_i \psi^i(a^i | x^i) \times \left(R(s, \mathbf{a}) + \gamma \sum_{s'} T(s' | s, \mathbf{a}) \sum_{\mathbf{o}} O(\mathbf{o} | \mathbf{a}, s') \sum_{\mathbf{x}'} \prod_i \eta^i(x^{i\prime} | x^i, a^i, o^i) U(\mathbf{x}', s') \right)$$

для всех $\mathbf{x}, s$;

$$\psi^i(a^i | x^i) \geq 0 \text{ для всех } x^i, a^i;$$

$$\sum_a \psi^i(a^i | x^i) = 1 \text{ для всех } x^i;$$

$$\eta^i(x^{i\prime} | x^i, a^i, o^i) \geq 0 \text{ для всех } x^i, a^i, o^i, x^{i\prime};$$

$$\sum_{x^{i\prime}} \eta^i(x^{i\prime} | x^i, a^i, o^i) = 1 \text{ для всех } x^i, a^i, o^i.$$

Адаптируя алгоритм 27.3 для стратегий контроллера, эта программа заменяет внутреннюю операцию поиска наилучшего отклика.

A Основные математические понятия

В этом приложении дается краткий обзор некоторых математических понятий, используемых в книге.

A.1. Пространство с мерой

Прежде чем ввести определение пространства с мерой, сначала введем понятие сигма-алгебры над множеством Ω. *Сигма-алгебра* – это набор подмножеств Σ таких, что

1) $\Omega \in \Sigma$;
2) если $E \in \Sigma$, то $\Omega \setminus E \in \Sigma$ (*замкнуто относительно дополнения*);
3) если $E_1, E_2, E_3, \ldots \in \Sigma$, то $E_1 \cup E_2 \cup E_3 \ldots \in \Sigma$ (*замкнуто относительно счетных объединений*).

Элемент $E \in \Sigma$ называется *измеримым множеством*.

Пространство с мерой (measure space) определяется множеством Ω, сигма-алгеброй Σ и мерой μ: $\Omega \to \mathbb{R} \cup \{\infty\}$. Чтобы μ было мерой, должны выполняться следующие условия:

1. Если $E \in \Sigma$, то $\mu(E) \geq 0$ (*неотрицательность*).
2. $\mu(\emptyset) = 0$.
3. Если $E_1, E_2, E_3, \ldots \in \Sigma$ попарно не пересекаются, то $\mu(E_1 \cup E_2 \cup E_3 \ldots) = \mu(E_1) +$
4. $\mu(E_2) + \mu(E_3) + \cdots$ (*счетная аддитивность*).

A.2. Вероятностное пространство

Вероятностное пространство (probability space) – это пространство с мерой (Ω, Σ, μ) с требованием, чтобы $\mu(\Omega) = 1$. В контексте вероятностных пространств

Ω называется *выборочным пространством* (sample space), Σ – *пространством событий* (event space), а μ (или чаще P) – *вероятностной мерой* (probability measure). *Аксиомы вероятности*[1] устанавливают свойства неотрицательности и счетной аддитивности пространств с мерой вместе с требованием $\mu(\Omega) = 1$.

A.3. Метрическое пространство

Множество с *метрикой* называется *метрическим пространством* (metric space). Метрика d, иногда называемая *метрикой расстояния* (distance metric), представляет собой функцию, которая отображает пары элементов в X в неотрицательные действительные числа, такие что для всех $x, y, z \in X$:

1) $d(x, y) = 0$ тогда и только тогда, когда $x = y$ (*тождество неразличимых*);
2) $d(x, y) = d(y, x)$ (*симметрия*);
3) $d(x, y) \leq d(x, z) + d(z, y)$ (*неравенство треугольника*).

A.4. Нормированное векторное пространство

Нормированное векторное пространство (normed vector space) состоит из *векторного пространства* X и нормы $||\cdot||$, которая отображает элементы X в неотрицательные действительные числа такие, что для всех скаляров α и векторов $\mathbf{x}, \mathbf{y} \in X$:

1) $||\mathbf{x}|| = 0$ тогда и только тогда, когда $\mathbf{x} = 0$;
2) $||\alpha\mathbf{x}|| = |\alpha|||\mathbf{x}||$ (*абсолютно однородный*);
3) $||\mathbf{x} + \mathbf{y}|| \leq ||x|| + ||y||$ (*неравенство треугольника*).

Нормы L_p – это обычно используемый набор норм, параметризованных скаляром $p \geq 1$. Норма L_p вектора $\mathbf{x}$ равна

$$||\mathbf{x}||_p = \lim_{\rho \to p}(|x_1|^\rho + |x_1|^\rho + \cdots + |x_1|^\rho)^{\frac{1}{\rho}}, \tag{A.1}$$

где предел необходим для определения бесконечной нормы L_∞. Несколько норм L_p показаны на рис. A.1.

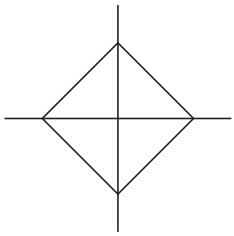
Нормы можно использовать для получения метрик расстояния в векторных пространствах, задав метрику $d(\mathbf{x}, \mathbf{y}) = ||\mathbf{x} - \mathbf{y}||$. Затем мы можем, например, использовать норму L_p для определения расстояний.

---

[1] Эти аксиомы иногда называют *аксиомами Колморогова. Колмогоров А. Н.* Основные понятия теории вероятностей. 2-е изд. М.: Наука, 1974. 120 с.

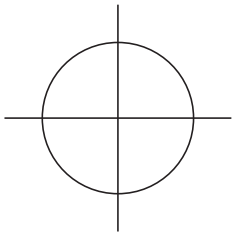
L_1: $||\mathbf{x}||_1 = |x_1| + |x_2| + \cdots + |x_n|$

Эту метрику часто называют *нормой такси* (taxicab norm)



L_2: $||\mathbf{x}||_2 = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$

Эту метрику часто называют *евклидовой нормой* (Euclidean norm)



L_∞: $||\mathbf{x}||_\infty = \max(|x_1|, |x_2|, \cdots, |x_n|)$

Эту метрику часто называют максимальной нормой (max norm), *нормой Чебышева* или *нормой шахматной доски* (chessboard norm). Последнее название происходит от минимального количества ходов, которое необходимо королю, чтобы переместиться между двумя клетками в шахматах

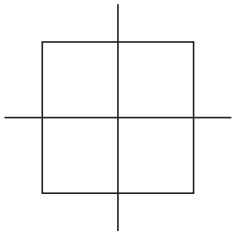


Рис. А.1. Распространенные нормы L_p.
Справа показана форма контуров нормы в двух измерениях.
Все точки контура равноудалены от начала координат по этой норме

А.5. Положительная определенность

Симметричная матрица **A** *положительно определена*, если $\mathbf{x}^T\mathbf{A}\mathbf{x}$ является положительным для всех точек, кроме начала координат. Другими словами, $\mathbf{x}^T\mathbf{A}\mathbf{x} > 0$ для всех $\mathbf{x} \neq 0$. Симметричная матрица **A** является *положительно полуопределенной*, если $\mathbf{x}^T\mathbf{A}\mathbf{x}$ всегда остается неотрицательным. Другими словами, $\mathbf{x}^T\mathbf{A}\mathbf{x} \geq 0$ для всех **x**.

А.6. Выпуклость

Выпуклая комбинация (convex combination) двух векторов **x** и **y** является результатом операции

$$\alpha\mathbf{x} + (1 - \alpha)\mathbf{y} \tag{A.2}$$

для некоторого $\alpha \in [0, 1]$. Выпуклые комбинации можно составить из m векторов:

$$w_1\mathbf{v}^{(1)} + w_2\mathbf{v}^{(2)} + \cdots + w_m\mathbf{v}^{(m)} \tag{A.3}$$

с неотрицательными весами $\mathbf{w}$, сумма которых равна 1.

Выпуклое множество (convex set) – это множество, для которого линия, проведенная между любыми двумя точками множества, полностью находится внутри множества. Математически множество $\mathcal{S}$ является выпуклым, если мы имеем

$$\alpha\mathbf{x} + (1 - \alpha)\mathbf{y} \in \mathcal{S} \tag{A.4}$$

для всех $\mathbf{x}$, $\mathbf{y}$ в $\mathcal{S}$ и для всех α в $[0, 1]$. Выпуклые и невыпуклые множества показаны на рис. А.2.

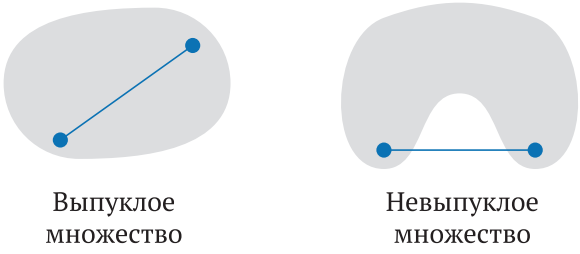


Рис. А.2. Выпуклые и невыпуклые множества

Выпуклая функция (convex function) – это *чашеобразная* функция, областью определения которой является выпуклое множество. Под «чашеобразной» мы подразумеваем такую функцию, что любая линия, проведенная между двумя точками в ее области определения, не лежит ниже функции. Функция f называется выпуклой над выпуклым множеством $\mathcal{S}$, если для всех $\mathbf{x}$, $\mathbf{y}$ в $\mathcal{S}$ и для всех α в $[0, 1]$

$$f(\alpha\mathbf{x} + (1 - \alpha)\mathbf{y}) \leq \alpha f(\mathbf{x}) + (1 - \alpha)f(\mathbf{y}). \tag{A.5}$$

Выпуклая и вогнутая области функции показаны на рис. А.3.

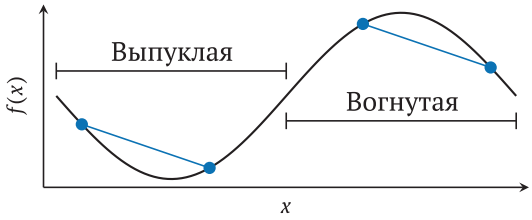
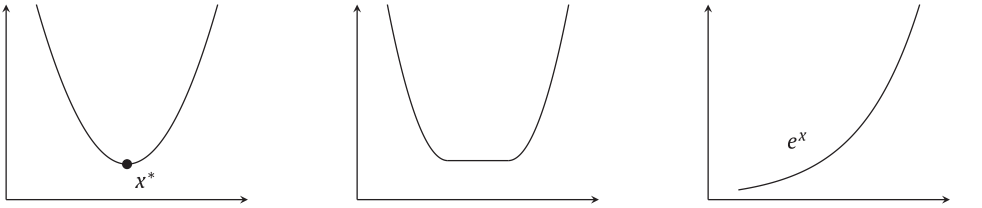


Рис. А.3. Выпуклая и вогнутая области функции

Функция f является *строго выпуклой* (strictly convex) на выпуклом пространстве $\mathcal{S}$ для всех $\mathbf{x}$, $\mathbf{y}$ в $\mathcal{S}$ и α в (0, 1), если

$$f(\alpha\mathbf{x} + (1 - \alpha)\mathbf{y}) < \alpha f(\mathbf{x}) + (1 - \alpha)f(\mathbf{y}). \tag{A.6}$$

Строго выпуклые функции имеют не более одного минимума, тогда как выпуклая функция может иметь плоские области[2]. Примеры строгой и нестрогой выпуклости показаны на рис. А.4.



Строго выпуклая функция с одним глобальным минимумом

Выпуклая функция без уникального глобального минимума

Строго выпуклая функция без глобального минимума

Рис. А.4. Не все выпуклые функции имеют один глобальный минимум

Функция f *вогнута* (concave), если $-f$ выпукла. Кроме того, f *строго вогнута*, если $-f$ строго выпукла.

A.7. Объем информации

Если у нас есть дискретное распределение, которое присваивает вероятность $P(x)$ значению x, *объем информации* (information content, *информационное наполнение*)[3] наблюдения x определяется выражением

$$I(x) = -\log P(x). \tag{A.7}$$

Единица объема информации зависит от основания логарифма. Обычно мы используем натуральные логарифмы (с основанием e) и единицу *нат* (nat), что является сокращением от natural. В области теории информации основание часто равно 2, что дает единицу *бит* (bit). Мы можем рассматривать эту величину как количество битов, необходимых для передачи значения x в соответствии с оптимальным кодированием сообщения, когда распределение вероятностей по сообщениям следует заданному распределению.

---

[2] Оптимизации выпуклых функций посвящен учебник S. Boyd, L. Vandenberghe, Convex Optimization. Cambridge University Press, 2004.

[3] Иногда информационное наполнение называют *информацией Шеннона* в честь Клода Шеннона, основателя области теории информации. C. E. Shannon, *A Mathematical Theory of Communication*, Bell System Technical Journal, vol. 27, no. 4, pp. 623–656, 1948.

A.8. Энтропия

Энтропия – это мера неопределенности в теории информации. Энтропия, связанная с дискретной случайной величиной X, представляет собой ожидаемое информационное наполнение:

$$H(X) = \mathbb{E}_X[I(x)] = \sum_x P(x)I(x) = -\sum_x P(x)\log P(x), \tag{A.8}$$

где $P(x)$ – вероятностная масса, относящаяся к x.

Для непрерывного распределения, где $p(x)$ – плотность, связанная с x, *дифференциальная энтропия* (также известная как *непрерывная энтропия*) определяется следующим образом:

$$h(X) = \int p(x)I(x)dx = -\int p(x)\log p(x)\mathrm{d}x. \tag{A.9}$$

A.9. Взаимная энтропия

Взаимная энтропия (или *перекрестная энтропия*, cross entropy) одного распределения относительно другого может быть определена через ожидаемое информационное наполнение. Если у нас есть одно дискретное распределение с функцией распределения масс $P(x)$, а другое с функцией $Q(x)$, то взаимная энтропия P относительно Q определяется выражением

$$H(P,Q) = -\mathbb{E}_{x\sim P}[\log Q(x)] = -\sum_x P(x)\log Q(x). \tag{A.10}$$

Для непрерывных распределений с функциями плотности $p(x)$ и $q(x)$ имеем:

$$H(p,q) = -\int p(x)\log q(x)\mathrm{d}x. \tag{A.11}$$

A.10. Относительная энтропия

Относительная энтропия, также называемая *расхождением Кульбака–Лейблера* (KL), является мерой того, насколько данное распределение вероятностей отличается от эталонного распределения[4]. Если $P(x)$ и $Q(x)$ являются функциями распределения масс, то расхождение KL от Q до P представляет собой матема-

[4] Назван в честь двух американских математиков, которые ввели эту меру, Соломона Кульбака (1907–1994) и Ричарда А. Лейблера (1914–2003). S. Kullback, R. A. Leibler, *On Information and Sufficiency*, Annals of Mathematical Statistics, vol. 22, no. 1, pp. 79–86, 1951. S. Kullback, Information Theory and Statistics. Wiley, 1959.

тическое ожидание логарифмических разностей с математическим ожиданием, использующим P:

$$D_{\mathrm{KL}}(P \,||\, Q) = \sum_x P(x) \log \frac{P(x)}{Q(x)} = -\sum_x P(x) \log \frac{Q(x)}{P(x)}. \tag{A.12}$$

Эта величина определяется только в том случае, если носитель (несущее множество) P является подмножеством носителя Q. Суммирование производится по носителю P, чтобы избежать деления на ноль.

Для непрерывных распределений с функциями плотности $p(x)$ и $q(x)$ имеем:

$$D_{\mathrm{KL}}(p \,||\, q) = \int p(x) \log \frac{p(x)}{q(x)} = -\int p(x) \log \frac{q(x)}{p(x)} \mathrm{d}x. \tag{A.13}$$

Точно так же эта величина определяется только в том случае, если носитель p является подмножеством носителя q. Интеграл вычисляется по носителю p, чтобы избежать деления на ноль.

A.11. Градиентный подъем

Градиентный подъем (gradient ascent) – это общий подход к поиску максимума функции $f(\mathbf{x})$, если f – дифференцируемая функция. Мы начинаем с точки $\mathbf{x}$ и итеративно применяем следующее правило обновления:

$$\mathbf{x} \leftarrow \mathbf{x} + \alpha \nabla f(\mathbf{x}), \tag{A.14}$$

где $\alpha > 0$ называется *коэффициентом шага* (step factor). Идея этого подхода к оптимизации заключается в том, что мы делаем шаги в направлении градиента, пока не достигнем локального максимума. При использовании этого метода нет никакой гарантии, что будет найден глобальный максимум. Малые значения α обычно требуют большего количества итераций, чтобы приблизиться к локальному максимуму. Большие значения α часто приводят к «перескоку» через локальный оптимум, не достигая его. Если коэффициент α является постоянным на всем протяжении итераций, его иногда называют *скоростью обучения* (learning rate). Во многих приложениях используется *затухающий коэффициент шага* (decaying step factor), когда в дополнение к обновлению $\mathbf{x}$ на каждой итерации мы обновляем α в соответствии с выражением

$$\alpha \leftarrow \gamma\alpha, \tag{A.15}$$

где $0 < \gamma < 1$ – *коэффициент затухания*.

A.12. Разложение Тейлора

Разложение Тейлора (Taylor expansion)[5], также называемое *рядом Тейлора*, важно для многих приближенных представлений функций, используемых в этой книге. Из *первой основной теоремы исчисления*[6] мы знаем, что

$$f(x+h) = f(x) + \int_0^h f'(x+a)\,\mathrm{d}a. \tag{A.16}$$

Рекурсивная подстановка этого определения дает разложение Тейлора функции f относительно x:

$$f(x+h) = f(x) + \int_0^h \left(f'(x) + \int_0^a f''(x+b)\,\mathrm{d}b \right) \mathrm{d}a \tag{A.17}$$

$$= f(x) + f'(x)h + \int_0^h \int_0^a f''(x+b)\,\mathrm{d}b\,\mathrm{d}a \tag{A.18}$$

$$= f(x) + f'(x)h + \int_0^h \int_0^a \left(f''(x) + \int_0^b f'''(x+c)\,\mathrm{d}c \right) \mathrm{d}b\,\mathrm{d}a \tag{A.19}$$

$$= f(x) + f'(x)h + \frac{f''(x)}{2!}h^2 + \int_0^h \int_0^a \int_0^b f'''(x+c)\,\mathrm{d}c\,\mathrm{d}b\,\mathrm{d}a \tag{A.20}$$

$$\vdots \tag{A.21}$$

$$= f(x) + \frac{f'(x)}{1!}h + \frac{f''(x)}{2!}h^2 + \frac{f'''(x+c)}{3!}h^3 + \ldots \tag{A.22}$$

$$= \sum_{n=0}^{\infty} \frac{f^{(n)}(x)}{n!}h^n. \tag{A.23}$$

В приведенной выше записи x обычно фиксируется, а функция вычисляется через h. Часто бывает удобнее написать разложение Тейлора для $f(x)$ относительно точки a так, чтобы она оставалась функцией от x:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n. \tag{A.24}$$

Разложение Тейлора представляет функцию как бесконечную сумму полиномиальных членов, основанную на повторяющихся производных в одной точке. Любая аналитическая функция может быть представлена своим разложением Тейлора в локальной окрестности.

---

[5] Названо в честь английского математика Брука Тейлора (1685–1731), который предложил эту идею.

[6] Первая основная теорема исчисления связывает функцию с интегралом от ее производной: $f(b) - f(a) = \int_a^b f'(x)\mathrm{d}x$.

Функция может быть локально аппроксимирована с помощью первых нескольких членов разложения Тейлора. На рис. A.5 показаны все более совершенные приближения для функции cos(*x*) относительно $x = 1$. Использование большего количества членов разложения повышает точность локальной аппроксимации, но ошибка все равно возрастает по мере удаления от точки разложения.

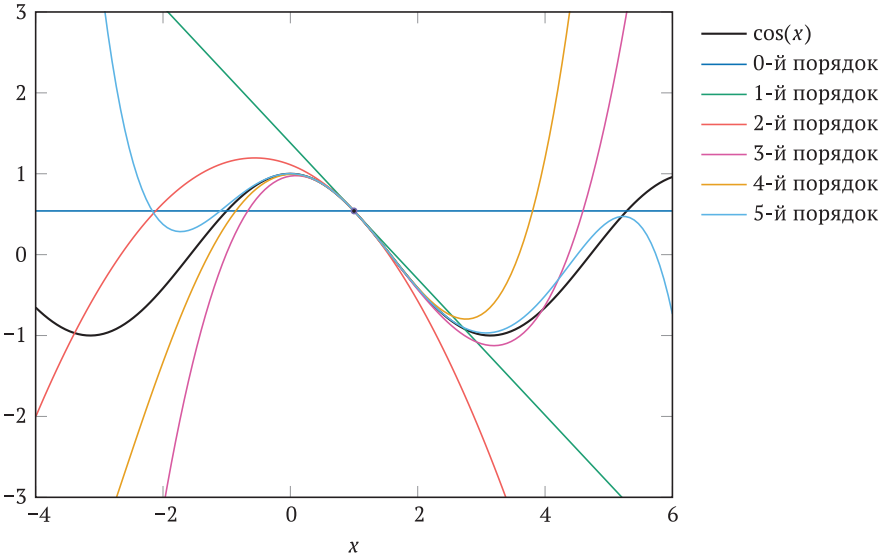


Рис. A.5. Последовательные приближения cos(x) около 1 на основе первых *n* членов разложения Тейлора

Линейное *приближение Тейлора* использует первые два члена разложения Тейлора:

$$f(x) \approx f(a) + f'(a)(x - a). \tag{A.25}$$

В квадратичном приближении Тейлора используются первые три члена:

$$f(x) \approx f(a) + f'(a)(x-a) + \frac{1}{2} f''(a)(x-a)^2 \tag{A.26}$$

и т. д.

В нескольких измерениях разложение Тейлора в окрестности точки **a** обобщается до

$$f(\mathbf{x}) = f(\mathbf{a}) + \nabla f(\mathbf{a})^\top (\mathbf{x} - \mathbf{a}) + \frac{1}{2}(\mathbf{x} - \mathbf{a})^\top \nabla^2 f(\mathbf{a})(\mathbf{x} - \mathbf{a}) + \ldots. \tag{A.27}$$

Первые два члена образуют касательную плоскость в точке **a**. Третий член включает локальную кривизну. В этой книге будут использоваться только первые три приведенных здесь члена.

A.13. Оценка по методу Монте-Карло

Оценка по методу Монте-Карло (Monte Carlo estimation) позволяет нам оценить математическое ожидание функции f, когда ее вход x следует функции плотности вероятности p:

$$\mathbb{E}_{x\sim p}[f(x)] = \int f(x)p(x)\mathrm{d}x \approx \frac{1}{n}\sum_i f(x^{(i)}), \tag{A.28}$$

где $x^{(1)}, \ldots, x^{(n)}$ извлечены из p. Дисперсия оценки равна $\mathrm{Var}_{x\sim p}[f(x)]/n$.

A.14. Выборка по значимости

Выборка по значимости (importance sampling) позволяет нам вычислить $\mathbb{E}_{x\sim p}[f(x)]$ из выборок, извлеченных из другого распределения q:

$$\mathbb{E}_{x\sim p}[f(x)] = \int f(x)p(x)\mathrm{d}x \tag{A.29}$$

$$= \int f(x)p(x)\frac{q(x)}{q(x)}\mathrm{d}x \tag{A.30}$$

$$= \int f(x)\frac{p(x)}{q(x)}q(x)\mathrm{d}x \tag{A.31}$$

$$= \mathbb{E}_{x\sim q}\left[f(x)\frac{p(x)}{q(x)}\right]. \tag{A.32}$$

Приведенное выше уравнение можно аппроксимировать, используя выборки $x^{(1)}, \ldots, x^{(n)}$, извлеченные из q:

$$\mathbb{E}_{x\sim p}[f(x)] = \mathbb{E}_{x\sim q}\left[f(x)\frac{p(x)}{q(x)}\right] \approx \frac{1}{n}\sum_i f(x^{(i)})\frac{p(x^{(i)})}{q(x^{(i)})}. \tag{A.33}$$

A.15. Сжимающее отображение

Сжимающее отображение (или *сжатое отображение*, contraction mapping) f определено относительно функции над метрическим пространством такой, что

$$d(f(x), f(y)) \leq \alpha d(x, y), \tag{A.34}$$

где d – метрика расстояния, связанная с метрическим пространством, и $0 \leq \alpha < 1$. Таким образом, сжимающее отображение уменьшает расстояние между

любыми двумя элементами множества. Такую функцию иногда называют просто *сжатием*.

Следствием многократного применения сжимающего отображения является то, что расстояние между любыми двумя элементами множества становится равным 0. *Теорема о сжимающем отображении*, или *теорема Банаха о неподвижной точке*[7], утверждает, что каждое сжимающее отображение на полное[8] непустое метрическое пространство имеет единственную неподвижную точку. Кроме того, для любого элемента x в этом множестве многократное применение сжимающего отображения к этому элементу приводит к сходимости в данной фиксированной точке.

Демонстрация того, что функция f является сжимающим отображением в метрическом пространстве, полезна в различных доказательствах сходимости, связанных с понятиями, представленными выше. Например, мы можем показать, что оператор Беллмана является сжимающим отображением на пространстве функций полезности с max-нормой. Применение теоремы сжимающего отображения позволяет нам доказать, что повторное применение оператора Беллмана приводит к сходимости к единственной функции полезности. В примере А.1 показано простое сжимающее отображение.

Пример А.1. Сжимающее отображение для $\mathbb{R}^2$

Рассмотрим функцию $f(\mathbf{x}) = [x_2/2 + 1, x_1/2 + 1/2]$. Мы можем показать, что f является сжимающим отображением для множества $\mathbb{R}^2$ и евклидовой функции расстояния:

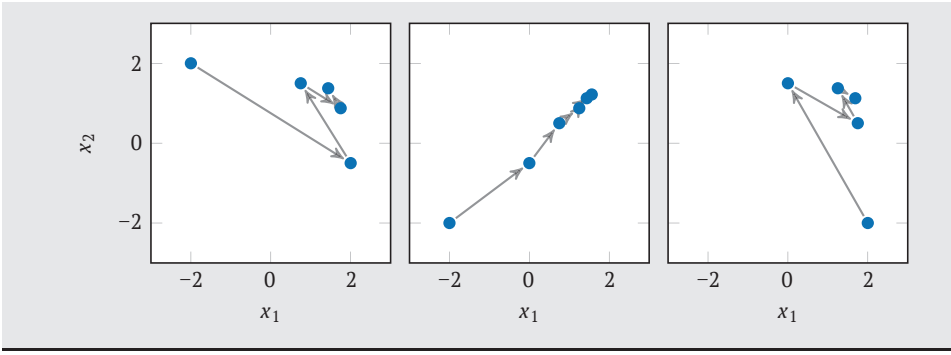
$$\begin{aligned} d(\mathbf{f}(\mathbf{x}), \mathbf{f}(\mathbf{y})) &= ||\mathbf{f}(\mathbf{x}) - \mathbf{f}(\mathbf{y})||_2 \\ &= ||[x_2/2 + 1, x_1/2 + 1/2] - [y_2/2 + 1, y_1/2 + 1/2]||_2 \\ &= ||[\frac{1}{2}(x_2 - y_2), (x_1 - y_1)]||_2 \\ &= \frac{1}{2}||[(x_2 - y_2), (x_1 - y_1)]||_2 \\ &= \frac{1}{2}d(\mathbf{x}, \mathbf{y}). \end{aligned}$$

Визуализируем эффект повторных применений f к точкам в $\mathbb{R}^2$ и покажем, как они сходятся к точке [5/3, 4/3]:

---

[7] Названа в честь польского математика Стефана Банаха (1892–1945), впервые сформулировавшего теорему.

[8] Полное метрическое пространство – это такое пространство, в котором каждая последовательность Коши сходится к точке в этом пространстве. Последовательность $x_1, x_2, \ldots$ является последовательностью Коши, если для каждого положительного действительного числа $\varepsilon > 0$ существует натуральное число n такое, что для всех положительных целых чисел $i, j > n$ выполняется условие $d(x_i, x_j) < \varepsilon$.



A.16. Графы

Граф $G = (V, E)$ определяется множеством *узлов* (также называемых *вершинами*) V и *ребер* E. На рис. А.6 показан пример графа. Ребро $e \in E$ – это пара узлов (v_i, v_j). Мы сосредоточимся в первую очередь на ориентированных графах, где ребра имеют направление и определяют отношения родитель–потомок. Ребро $e = (v_i, v_j)$ часто изображают графически в виде стрелки от v_i до v_j, где v_i является *родителем*, а v_j – *потомком*. Если существует ребро, соединяющее v_i и v_j, то говорят, что v_i и v_j – *соседи*. Множество всех родителей узла v_i обозначается как $\text{Pa}(v_i)$.

Путь от узла v_i к узлу v_j – это последовательность ребер, соединяющих v_i с v_j. Если этот путь можно пройти от узла к узлу строго вдоль направления ребер, то мы называем его *направленным путем*. В свою очередь, *ненаправленный путь* – это путь без учета направления ребер. Узел v_j является потомком v_i, если существует направленный путь из v_i в v_j. *Цикл*, или *петля*, – это направленный путь от узла к самому себе. Если граф не содержит циклов, он называется *ациклическим* графом.

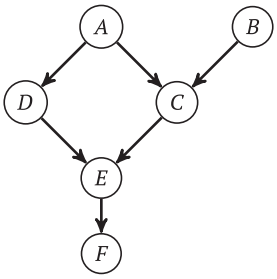
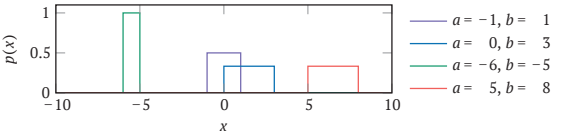
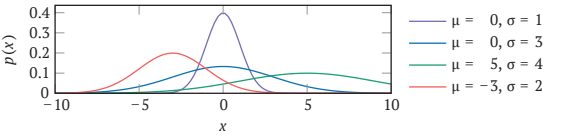
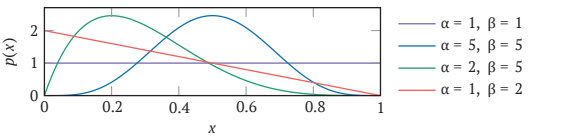


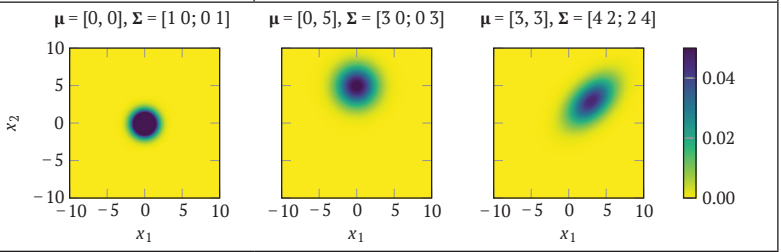
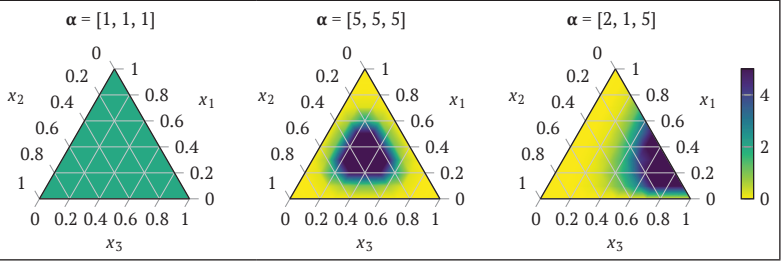
Рис. А.6. Пример графа. Здесь $\text{Pa}(C) = \{A, B\}$. Последовательность (A, C, E, F) – направленный путь, а (A, C, B) – ненаправленный путь. Узел A является родительским для C и D. Узел E является потомком B. Соседями C являются A, B и E

B Распределения вероятностей

В этом приложении представлено несколько семейств распределений вероятностей, имеющих отношение к темам, рассмотренным в этой книге[1]. Распределения представлены либо функциями массы, либо функциями плотности вероятностей в сопровождении соответствующих функций вместе с параметрами, определяющими каждое распределение. На графиках показано, как различные параметры влияют на распределение. Некоторые распределения *одномерные*, т. е. являются распределениями по скалярной переменной; другие – *многомерные*, т. е. являются распределениями по нескольким переменным одновременно (по вектору).

Название	Параметры	Функция распределения
Равномерное $\mathcal{U}(a, b)$	a – нижняя граница b – верхняя граница	$p(x) = \frac{1}{b-a}$, где $x \in [a, b]$
		
Гауссово (одномерное) $\mathcal{N}(\mu, \sigma)$	μ – среднее σ^2 – дисперсия	$p(x) = \frac{1}{\sigma}\varphi\left(\frac{x-\mu}{\sigma}\right)$, где $\varphi(x) = \frac{1}{\sqrt{2\pi}}\exp(-x^2/2)$ и где $x \in \mathbb{R}$
		
Бета Beta(α, β)	$\alpha > 0$ – форма $\beta > 0$ – форма	$p(x) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1}(1-x)^{\beta-1}$, где $x \in (0, 1)$
		

[1] Эти распределения реализованы в библиотеке `Distributions.jl`. M. Besançon, T. Papamarkou, D. Anthoff, A. Arslan, S. Byrne, D. Lin, J. Pearson, *Distributions.jl: Definition and Modeling of Probability Distributions in the JuliaStats Ecosystem*, 2019. arXiv: 1907.08611v1.

<table>
<tr><td rowspan="2">Гауссово (многомерное)
$\mathcal{N}(\mu, \Sigma)$</td><td>μ – среднее
Σ – ковариация</td><td>$p(x) = \frac{1}{(2\pi)^{n/2}\lvert\boldsymbol{\Sigma}\rvert^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\top}\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right)$,
где $n = \dim(\mathbf{x})$ и $\mathbf{x} \in \mathbb{R}^n$</td></tr>
<tr><td colspan="2"> 
</td></tr>
<tr><td rowspan="2">Дирихле
Dir(α)</td><td>α > 0 – концентрация</td><td>$p(x) = \frac{\Gamma(\alpha_0)}{\prod_{i=1}^{n}\Gamma(\alpha_i)} \prod_{i=1}^{n} x_i^{\alpha_i - 1}$, где $\alpha_0 = \sum_i \alpha_i$,
а также $x_i \in (0, 1)$ и $\sum_i x_i = 1$</td></tr>
<tr><td colspan="2"> 
</td></tr>
</table>

C Вычислительная сложность

При обсуждении различных алгоритмов полезно анализировать их *вычислительную сложность* (computational complexity), которая характеризует количество ресурсов, необходимых для их выполнения[1]. Обычно нас интересует либо временна́я, либо пространственная сложность. В этом приложении представлены асимптотические обозначения, которые обычно используются для характеристики сложности. Затем мы рассмотрим несколько классов сложности, относящихся к алгоритмам в книге, и обсудим проблему разрешимости.

C.1. Асимптотические обозначения

Асимптотическое обозначение часто используют, чтобы охарактеризовать рост функции. Это обозначение иногда называют записью «*О большое*»; букву *O* выбрали потому, что скорость роста функции часто называют ее *порядком* (order). Обозначение «*O* большое» может применяться для описания ошибки, связанной с численным методом или временно́й либо пространственной сложностью алгоритма. Это обозначение указывает на верхнюю границу функции, когда ее аргумент приближается к определенному значению.

С математической точки зрения, если $f(x) = O(g(x))$ при $x \to a$, то абсолютное значение $f(x)$ ограничено абсолютным значением $g(x)$, умноженным на некоторое положительное и конечное число c для значений x, достаточно близких к a:

$$|f(x)| \leq c|g(x)| \quad \text{для } x \to a. \tag{C.1}$$

Запись вида $f(x) = O(g(x))$ является распространенным некорректным применением знака равенства. Например, $x^2 = O(x^2)$ и $2x^2 = O(x^2)$, но, конечно же, $x^2 \neq 2x^2$. В некоторых математических публикациях $O(g(x))$ представляет собой множество всех функций, которые не растут быстрее, чем $g(x)$. Например, $5x^2 \in O(x^2)$. Использование асимптотического обозначения показано в примере C.1.

---

[1] Анализ алгоритмов представляет собой обширную область компьютерных наук. Вводный учебник см. в O. Goldreich, *Computational Complexity: A Conceptual Perspective*. Cambridge University Press, 2008. Для строгого определения необходимо ввести концепции и вычислительные модели, такие как машины Тьюринга, которые мы не рассматриваем в этой книге.

Если $f(x)$ – *линейная комбинация* членов[2], то $O(f)$ соответствует порядку самого быстрорастущего члена. В примере С.2 сравниваются порядки нескольких членов.

Пример С.1. Асимптотическая запись константы, умноженной на функцию

Рассмотрим функцию $f(x) = 10^6 e^x$ при $x \to \infty$. Здесь f – произведение константы 10^6 и e^x. Константу можно просто включить в граничную константу c следующим образом:

$|f(x)| \leq c|g(x)|$;

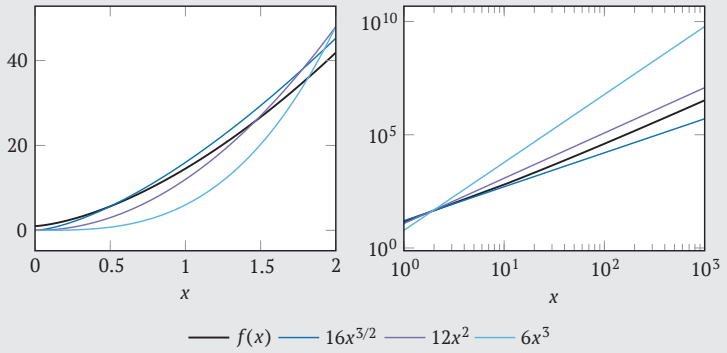
$10^6|e^x| \leq c|g(x)|$;

$|e^x| \leq c|g(x)|$.

Таким образом, $f = O(e^x)$ при $x \to \infty$.

Пример С.2. Нахождение порядка линейной комбинации членов

Рассмотрим функцию $f(x) = \cos(x) + x + 10x^{3/2} + 3x^2$. Здесь f – линейная комбинация членов. Члены $\cos(x)$, x, $x^{3/2}$, x^2 расположены в порядке возрастания значения по мере приближения x к бесконечности. Построим график $f(x)$ вместе с $c|g(x)|$, где значение c было выбрано для каждого члена так, что $c|g(x = 2)|$ превышает $f(x = 2)$.



Не существует константы c такой, что $f(x)$ всегда меньше, чем $c|x^{3/2}|$, при достаточно больших значениях x. То же самое верно для $\cos(x)$ и x.

Мы находим, что $f(x) = O(x^3)$, и в общем виде $f(x) = O(x^m)$ для $m \geq 2$, наряду с другими классами функций, такими как $f(x) = e^x$. Обычно мы рассматриваем порядок, обеспечивающий самую точную верхнюю границу. Таким образом, $f = O(x^2)$ при $x \to \infty$.

[2] Линейная комбинация представляет собой взвешенную сумму членов. Если условия находятся в векторе $\mathbf{x}$, то линейная комбинация имеет вид $w_1x_1 + w_2x_2 + \cdots = w^\top x$.

C.2. Классы временной сложности

Сложность решения тех или иных задач можно сгруппировать по разным классам временно́й сложности. Важные классы, которые часто встречаются в этой книге:

- *P*: задачи, которые можно решить за полиномиальное время;
- *NP*: задачи, решения которых можно проверить за полиномиальное время;
- *NP-сложные*: задачи, которые не менее сложны, чем самые сложные задачи в NP;
- *NP-полные*: задачи, которые являются одновременно NP-сложными и NP.

Формальные определения этих классов сложности весьма запутаны. Обычно считается, что P ≠ NP, но это не доказано и остается одним из важнейших открытых вопросов математики. Фактически вся современная криптография держится на том факте, что не существует известных эффективных (т. е. полиномиальных по времени) алгоритмов для решения NP-сложных задач. На рис. C.1 показаны отношения между классами сложности в предположении, что P ≠ NP.

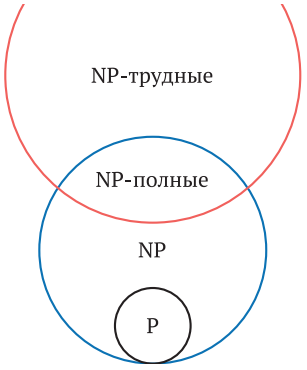


Рис. C.1. Классы сложности

Обычный подход к доказательству того, является ли конкретная задача *Q* NP-сложной, состоит в том, чтобы найти полиномиальное преобразование известной NP-*полной* задачи[3] *Q′* к экземпляру *Q*. Задача 3SAT является первой известной NP-полной задачей и обсуждается в примере C.3.

Пример C.3. Задача 3SAT – первая известная NP-полная задача

Задача *логической выполнимости* (Boolean satisfiability) включает в себя выяснение того, *выполнима* ли булева формула. Булева формула состоит из конъюнкций (∧), дизъюнкций (∨) и отрицаний (¬) и содержит *n* логических переменных $x_1 \ldots x_n$. Литерал – это переменная x_i или ее отрицание $\neg x_i$. Оператор 3SAT – это дизъюнкция до трех литералов (например, $x_3 \vee \neg x_5 \vee x_6$). Формула 3SAT представляет собой соединение таких операторов 3SAT, как

$$F(x_1, x_2, x_3, x_4) = \begin{matrix} (\ x_1 \vee x_2 \vee x_3\ ) \wedge \\ (\ \neg x_1 \vee \neg x_2 \vee x_3\ ) \wedge . \\ (\ x_2 \vee \neg x_3 \vee x_4\ ) \end{matrix}$$

[3] Существует много хорошо известных NP-полных задач, как указано в обзоре R. M. Karp, *Reducibility Among Combinatorial Problems*, in Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum, 1972, pp. 85–103.

Задача 3SAT состоит в том, чтобы определить, существует ли возможное распределение значений истинности по переменным, которое делает формулу истинной. В приведенной выше формуле

F(истина, ложь, ложь, истина) = истина.

Следовательно, формула выполнима. Хотя выполнимое сочетание значений для некоторых задач 3SAT находится довольно легко, иногда просто путем быстрой проверки, в общем случае их трудно решить. Один из способов определить, может ли быть выполнено искомое присвоение значений, состоит в том, чтобы перебрать 2^n возможных значений истинности всех переменных. Хотя сложно определить, существует ли дающее истину сочетание входных значений, проверка того, дает ли истину определенное сочетание, может быть выполнена за линейное время.

C.3. Классы пространственной сложности

Второй набор классов сложности характеризует *пространство*, а точнее объем памяти, необходимой для выполнения алгоритма до его завершения. Класс сложности PSPACE содержит множество всех задач, которые могут быть решены с полиномиальным объемом памяти без учета времени. Между сложностью во времени и в пространстве существует фундаментальная разница: время нельзя использовать повторно, а пространство можно. Мы знаем, что P и NP являются подмножествами PSPACE. Пока неизвестно, но есть предположение, что PSPACE включает в себя задачи, которых нет в NP. С помощью полиномиальных преобразований времени мы можем определить *PSPACE-сложные* и *PSPACE-полные* классы точно так же, как мы делали это с NP-сложными и NP-полными классами.

C.4. Разрешимость

Неразрешимая задача не всегда может быть решена за конечное время. Возможно, одной из самых известных неразрешимых задач является *задача остановки* (halting problem). По условию этой задачи мы получаем произвольную программу, написанную на достаточно выразительном языке[4], и должны определить, завершит ли она свое выполнение. Было доказано, что не существует алгоритма, который может выполнять обобщенный анализ такого рода. Хотя существуют алгоритмы, которые могут правильно определить, завершатся ли некоторые программы, не существует алгоритма, который может определить, завершится ли любая произвольная программа.

[4] Техническое требование состоит в том, чтобы язык был полным по Тьюрингу или вычислительно универсальным, что означает, что его можно использовать для моделирования любой машины Тьюринга.

D Представление функций в форме нейронных сетей

Нейронные сети – это параметрические представления нелинейных функций[1]. Функция, представленная в виде нейронной сети, является дифференцируемой, что позволяет алгоритмам градиентной оптимизации, таким как стохастический градиентный спуск, оптимизировать параметры сети, чтобы лучше аппроксимировать отношения ввода-вывода[2]. Нейронные представления могут быть полезны в различных контекстах, связанных с принятием решений, таких как построение вероятностных моделей, функций полезности и стратегий принятия решений. В этом приложении описывается несколько архитектур нейронных сетей.

D.1. Нейронные сети

С точки зрения математики нейронная сеть представляет собой дифференцируемую функцию $\mathbf{y} = \mathbf{f}_{\boldsymbol{\theta}}(\mathbf{x})$, которая отображает входные данные $\mathbf{x}$ на выходные данные $\mathbf{y}$ и имеет набор параметров $\boldsymbol{\theta}$. Современные нейронные сети могут иметь миллионы параметров и применяться для преобразования многомерных входных данных (например, изображения или видео) в многомерные выходные данные, такие как классификация по множеству классов или речь.

Параметры сети $\boldsymbol{\theta}$ обычно настраивают таким образом, чтобы минимизировать скалярную *функцию потерь* $\ell(\mathbf{f}_{\boldsymbol{\theta}}(\mathbf{x}), \mathbf{y})$, которая показывает, насколько далек реальный выход сети от желаемого выхода. И функция потерь, и нейронная сеть являются дифференцируемыми, что позволяет нам использовать градиент функции потерь относительно параметров $\nabla_{\boldsymbol{\theta}}\ell$ для итеративного улучшения параметризации. Этот процесс обычно называют *обучением* нейронной

---

[1] Название происходит от довольно слабой аналогии с сетью нейронов в биологическом мозге. Мы не будем обсуждать биологические аспекты; обзор и историческая перспектива предоставлены в книге B. Müller, J. Reinhardt, M. T. Strickland, *Neural Networks*. Springer, 1995.

[2] Этот процесс оптимизации применительно к многослойным нейронным сетям часто называют *глубоким обучением*. Среди популярных учебников по этой теме особенно выделяется I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning*. MIT Press, 2016. Пакет Julia `Flux.jl` обеспечивает эффективные реализации различных алгоритмов обучения.

сети, или *настройкой параметров*. Обучение нейронной сети продемонстрировано в примере D.1.

Пример D.1. Основы нейронных сетей и настройка параметров

Рассмотрим очень простую нейронную сеть $f_{\boldsymbol{\theta}}(x) = \theta_1 + \theta_2 x$. Нам нужно, чтобы нейронная сеть получала на вход площадь дома x и предсказывала его цену y_{pred}. Мы стремимся минимизировать квадратичное расхождение между прогнозируемой и истинной ценами на жилье с помощью функции потерь $\ell(y_{\text{pred}}, y_{\text{true}}) = (y_{\text{pred}} - y_{\text{true}})^2$. Используя обучающие пары значений вход–выход, мы можем вычислить градиент:

$$\nabla_{\boldsymbol{\theta}} \ell(f(x), y_{\text{true}}) = \nabla_{\boldsymbol{\theta}} (\theta_1 + \theta_2 x - y_{\text{true}})^2 = \begin{bmatrix} 2(\theta_1 + \theta_2 x - y_{\text{true}}) \\ 2(\theta_1 + \theta_2 x - y_{\text{true}})x \end{bmatrix}.$$

Допустим, у нас есть начальные параметры $\boldsymbol{\theta} = [10\,000, 123]$ и обучающая пара вход–выход ($x = 2500$, $y_{\text{true}} = 360\,000$). Градиент потерь в этом случае $\nabla\theta\ell = [-85\,000, -2.125 \times 10^8]$. Нам следует сделать небольшой шаг в противоположном направлении, чтобы улучшить аппроксимацию функции.

Нейронные сети обычно обучаются на наборе данных в виде пар ввода–вывода $\mathbf{D}$. Обучение сводится к подбору параметров таким образом, чтобы минимизировать совокупные потери по набору данных:

$$\arg\min_{\boldsymbol{\theta}} \sum_{(\mathbf{x},\mathbf{y})\in\mathbf{D}} \ell(\mathbf{f}_{\boldsymbol{\theta}}, (\mathbf{x}), \mathbf{y}). \tag{D.1}$$

Наборы данных для современных задач, как правило, очень велики, что делает поиск градиента уравнения (D.1) вычислительно дорогостоящим. Обычно на каждой итерации выбирают случайные подмножества – *пакеты* (batch) обучающих данных, используя их для вычисления градиента потерь. В дополнение к сокращению объема вычислений использование пакетов привносит некоторую стохастичность в градиент, что помогает обучению не застревать в локальных минимумах.

D.2. Сети прямого распространения

В нейронной сети обычно происходит передача входных данных через несколько последовательных слоев[3]. Сети с несколькими слоями часто называют

[3] Достаточно большая однослойная нейронная сеть теоретически может аппроксимировать любую функцию. См. A. Pinkus, *Approximation Theory of the MLP Model in Neural Networks*, Acta Numerica, vol. 8, pp. 143–195, 1999.

глубокими. В сетях *прямого распространения* (feedforward network) каждый уровень применяет аффинное преобразование, за которым следует нелинейная *функция активации* (activation function), применяемая поэлементно[4]:

$$\mathbf{x}' = \varphi.(\mathbf{W}\mathbf{x} + \mathbf{b}), \tag{D.2}$$

где матрица $\mathbf{W}$ и вектор $\mathbf{b}$ – параметры, связанные со слоем. Полностью связанный слой показан на рис. D.1. Размерность выхода отличается от размера входа, если матрица $\mathbf{W}$ не квадратная. На рис. D.2 показано более компактное изображение той же сети.

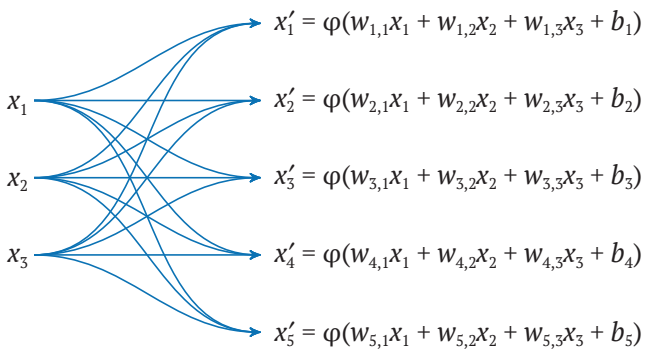


Рис. D.1. Полносвязный слой с трехкомпонентным входом и пятикомпонентным выходом

Рис. D.2. Более компактное представление рис. D.1. Слои нейронной сети часто изображают в виде блоков или пластин для простоты

Если между слоями нет нелинейных функций активации, несколько последовательных аффинных преобразований можно свернуть в одно эквивалентное аффинное преобразование:

$$\mathbf{W}_2(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 = \mathbf{W}_2\mathbf{W}_1\mathbf{x} + (\mathbf{W}_2\mathbf{b}_1 + \mathbf{b}_2). \tag{D.3}$$

Следовательно, нелинейности необходимы, чтобы нейронная сеть могла подстроиться под произвольные целевые функции. На рис. D.3 показаны вы-

[4] Нелинейность, вносимая функцией активации, обеспечивает поведение, аналогичное активационному поведению биологических нейронов, когда накопление входных данных в конечном итоге вызывает срабатывание нейрона. A. L. Hodgkin, A. F. Huxley, *A Quantitative Description of Membrane Current and Its Application to Conduction and Excitation in Nerve*, Journal of Physiology, vol. 117, no. 4, pp. 500–544, 1952.

ходные данные нейронной сети, которая обучена аппроксимировать некую нелинейную функцию.

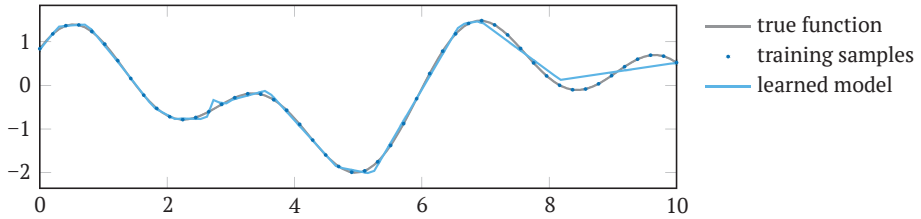


Рис. D.3. Глубокая нейронная сеть обучена на выборках из нелинейной функции путем минимизации квадрата ошибки. Эта нейронная сеть имеет четыре аффинных слоя, по 10 нейронов в каждом промежуточном представлении

Есть много разновидностей функций активации, которые часто используются в нейронных сетях. Подобно их биологическому прообразу, они, как правило, близки к нулю при низком значении на входе и выдают большое значение с ростом входа. Некоторые распространенные функции активации показаны на рис. D.4.

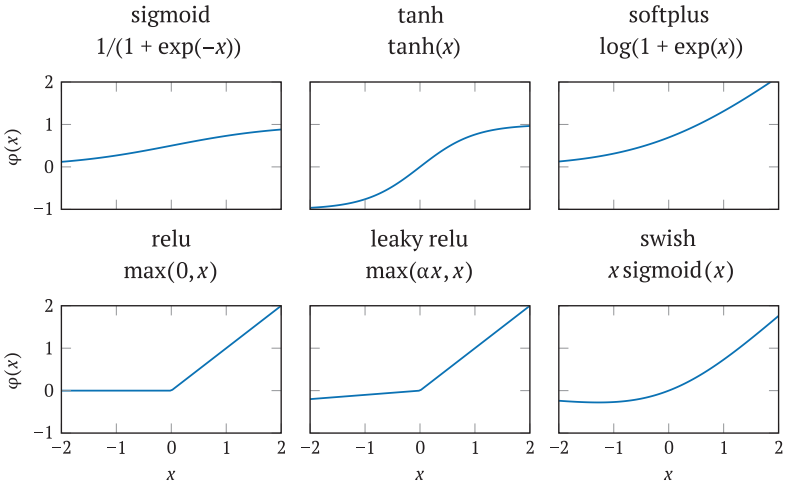


Рис. D.4. Несколько распространенных функций активации

Иногда для достижения определенных эффектов в нейронную сеть добавляют специальные слои. Например, на рис. D.5 показано использование слоя softmax в конце, чтобы заставить выходные данные представлять двухэлементное категориальное распределение. Функция softmax применяет экспоненциальную функцию к каждому элементу, что гарантирует положительное значение на выходе, а затем повторно нормирует полученные значения:

$$\text{softmax}(\mathbf{x})_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)}. \tag{D.4}$$

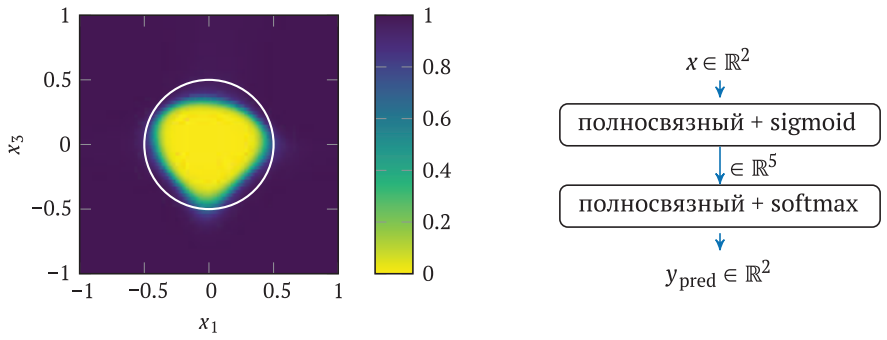


Рис. D.5. Простая двухслойная полностью связанная сеть, обученная классифицировать, находится ли заданная координата в пределах круга (показанного белым цветом). Нелинейности позволяют нейронным сетям формировать сложные нелинейные границы решений

Градиенты для нейронных сетей обычно вычисляются с использованием *обратного накопления* (reverse accumulation)[5]. Метод начинается с этапа прямого распространения, на котором нейронная сеть оценивается с использованием всех входных параметров. На этапе обратного распространения вычисляется градиент каждого интересующего члена по направлению от выхода обратно к входу. Обратное накопление основано на цепном правиле для производных:

$$\frac{\partial \mathbf{f}(\mathbf{g}(\mathbf{h}(\mathbf{x})))}{\partial \mathbf{x}} = \frac{\partial \mathbf{f}(\mathbf{g}(\mathbf{h}))}{\partial \mathbf{h}} \frac{\partial \mathbf{h}(\mathbf{x})}{\partial \mathbf{x}} = \left(\frac{\partial \mathbf{f}(\mathbf{g})}{\partial \mathbf{g}} \frac{\partial \mathbf{g}(\mathbf{h})}{\partial \mathbf{h}} \right) \frac{\partial \mathbf{h}(\mathbf{x})}{\partial \mathbf{x}}. \tag{D.5}$$

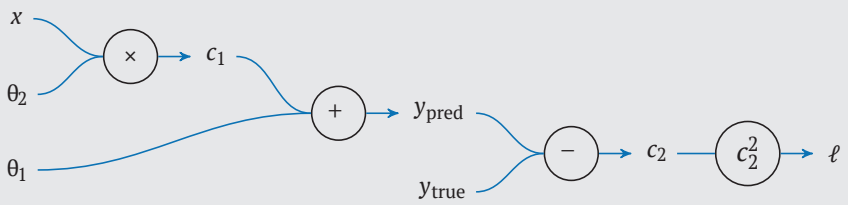
Пример D.2 демонстрирует этот процесс. Многие пакеты глубокого обучения вычисляют градиенты, используя подобные методы автоматического дифференцирования[6]. Пользователям редко приходится предоставлять свои собственные градиенты.

Пример D.2. Использование обратного накопления для вычисления градиентов параметров на основе обучающих данных

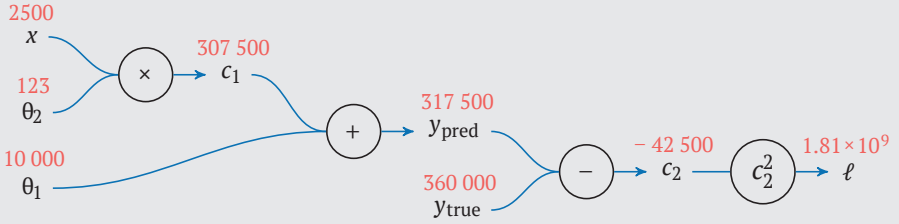
Вспомним нейронную сеть и функцию потерь из примера D.1. Ниже изображен вычислительный граф для расчета потерь:

[5] Этот процесс обычно называют обратным распространением, что конкретно относится к обратному накоплению, применяемому к скалярной функции потерь. D. E. Rumelhart, G. E. Hinton, R. J. Williams, *Learning Representations by Back-Propagating Errors*, Nature, vol. 323, pp. 533–536, 1986.

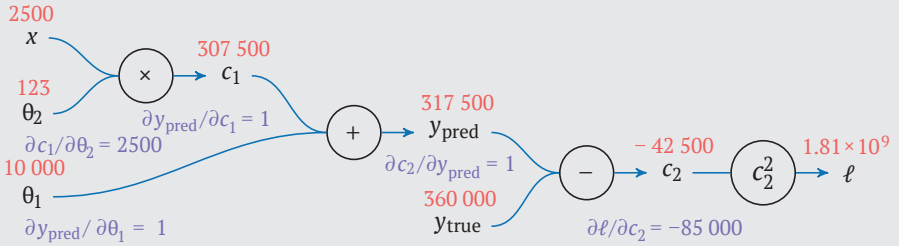
[6] A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, 2nd ed. SIAM, 2008.



Обратное накопление начинается с прямого прохода, в котором оценивается вычислительный граф. Мы снова будем использовать параметры $\boldsymbol{\theta} = [10\,000, 123]$ и обучающую пару вход–выход ($x = 2500$, $y_{\text{true}} = 360\,000$) следующим образом:



Затем вычисляем градиент, двигаясь в обратном направлении:



Наконец, мы вычисляем:

$$\frac{\partial \ell}{\partial \theta_1} = \frac{\partial \ell}{\partial c_2}\frac{\partial c_2}{\partial y_{\text{pred}}}\frac{\partial y_{\text{pred}}}{\partial \theta_1} = -85\,000 \cdot 1 \cdot 1 = -85\,000;$$

$$\frac{\partial \ell}{\partial \theta_2} = \frac{\partial \ell}{\partial c_2}\frac{\partial c_2}{\partial y_{\text{pred}}}\frac{\partial y_{\text{pred}}}{\partial c_1}\frac{\partial c_1}{\partial \theta_2} = -85\,000 \cdot 1 \cdot 1 \cdot 2500 = -2.125 \times 10^8.$$

D.3. Регуляризация параметров

Нейронные сети обычно *недоопределены*, а это означает, что существует несколько экземпляров параметров, которые могут привести к одной и той же

оптимальной потере при обучении[7]. Обычно используется *регуляризация параметров*, также называемая *регуляризацией веса*. В функцию потерь вводят дополнительный член, который штрафует большие значения параметров. Регуляризация также помогает предотвратить *переобучение*, которое случается, когда сеть чрезмерно специализируется на обучающих данных, но не может распространить эти знания на незнакомые данные.

Регуляризация часто принимает форму L_2-нормы вектора параметризации:

$$\arg\min_{\boldsymbol{\theta}} \sum_{(x,y)\in\mathbf{D}} \ell(f_{\boldsymbol{\theta}}(x), y) - \beta||\boldsymbol{\theta}||^2, \tag{D.6}$$

где положительная скалярная величина β определяет величину регуляризации параметра. Скаляр нередко довольно мал, со значениями около 10^{-6}, чтобы свести к минимуму снижение качества обучения вследствие добавления регуляризации.

D.4. Сверточные нейронные сети

На вход нейросети могут поступать изображения или другие многомерные структуры, такие как данные лидарного сканирования пространства. Даже относительно небольшое RGB-изображение 256×256 (как на рис. D.6) содержит 256 × 256 × 3 = 196 608 элементов. Любой полносвязный слой, принимающий изображение размером m×m×3 в качестве входных данных и создающий вектор из n выходных данных, будет иметь весовую матрицу с $3m^2n$ значениями. Чрезмерно большое количество параметров для обучения не только требует огромных вычислительных ресурсов, но и бесполезно. Информация в изображениях обычно инвариантна к трансляции; объект на изображении, сдвинутый вправо на 1 пиксель, должен давать схожий, если не идентичный результат на выходе нейронной сети.

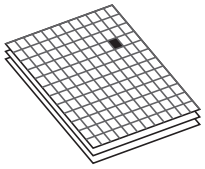


Рис. D.6. Многомерные входные данные, такие как изображения, обобщают векторы в тензоры. Здесь показано трехслойное изображение RGB. Такие входные данные могут иметь очень много элементов

Сверточные слои[8] значительно сокращают объем вычислений и обеспечивают инвариантность трансляции, перемещая небольшое полносвязное окно

---

[7] Например, предположим, что у нас есть нейронная сеть с последним слоем softmax. Входные данные для этого слоя можно масштабировать, производя тот же результат и, следовательно, те же потери.

[8] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, *Gradient-Based Learning Applied to Document Recognition*, Proceedings of the IEEE, vol. 86, no. 11, pp. 2278–2324, 1998.

по входу для получения выходных данных. Количество обучаемых параметров значительно снижается, а сами параметры восприимчивы к локальным текстурам почти так же, как нейроны зрительной коры реагируют на стимулы в своих рецептивных полях.

Сверточный слой состоит из набора *признаков* (feature) или *ядер* (kernel). Каждое ядро эквивалентно полносвязному слою, в который можно ввести меньшую область входного тензора. Одно ядро применяется один раз, как показано на рис. D.7. Признаки имеют полную глубину, а это означает, что если входной тензор имеет размерность $n \times m \times \mathbf{d}$, признаки также будут иметь третье измерение d. Признаки применяются много раз путем перемещения их по входным данным как в первом, так и во втором измерении. Если *шаг* перемещения (stride) равен 1×1, то все k фильтров применяются к каждой возможной позиции, и вывод будет иметь размерность $n \times m \times k$. Если шаг равен 2×2, то фильтры сдвигаются на 2 в первом и втором измерениях при каждом применении, что дает вывод с размерностью $n/2 \times m/2 \times k$. Сверточные нейронные сети обычно увеличиваются в третьем измерении и уменьшаются в первых двух измерениях с каждым слоем.

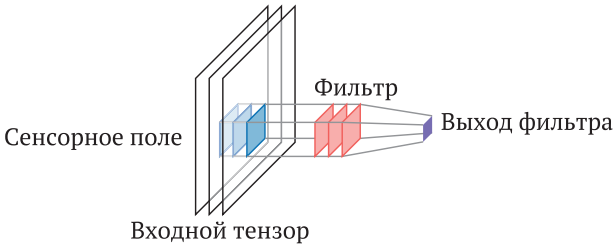
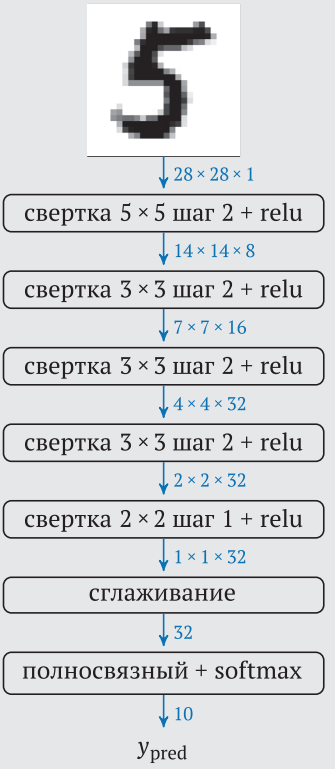


Рис. D.7. Сверточный слой многократно применяет фильтры к входному тензору (например, изображению) для создания выходного тензора. Здесь показано, что фильтр действует как небольшой полностью связанный слой, применяемый к небольшому рецептивному полю для создания одной записи в выходном тензоре. Каждый фильтр перемещается по входному полю в соответствии с заданным шагом

Сверточные слои инвариантны к трансляции, поскольку каждый фильтр ведет себя одинаково независимо от позиции входных данных. Это свойство особенно полезно при пространственной обработке, поскольку смещенные входные изображения благодаря инвариантности фильтра могут давать одинаковые выходные данные, что облегчает нейронным сетям извлечение общих признаков. Отдельные признаки, как правило, обучаются распознавать локальные атрибуты, такие как цвета и текстуры. В примере D.3 представлена классическая сверточная сеть для распознавания рукописных цифр из набора MNIST.

Пример D.3. Сверточная нейтральная сеть для набора данных MNIST. Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, *Gradient-Based Learning Applied to Document Recognition*, Proceedings of the IEEE, vol. 86, no. 11, pp. 2278–2324, 1998

Набор данных MNIST содержит рукописные цифры в виде монохроматических изображений 28×28. Его часто используют для тестирования сетей классификации изображений. На рисунке справа показан пример сверточной нейронной сети, которая принимает изображение MNIST в качестве входных данных и создает категориальное распределение вероятностей по 10 возможным цифрам. Для эффективного извлечения признаков применяются сверточные слои. Модель сжимается в первых двух измерениях и расширяется в третьем измерении (количестве признаков) по мере увеличения глубины сети. В конечном итоге достижение первого и второго измерений, равных 1, гарантирует, что информация со всего изображения может повлиять на каждый признак. Операция сглаживания берет входные данные 1×1×32 и преобразует их в «плоский» 32-компонентный выход. Такие операции часто применяются при переходе между сверточными и полносвязными слоями. Эта модель имеет 19 722 параметра. Параметры можно настроить, чтобы максимизировать правдоподобие на обучающих данных.



D.5. Рекуррентные сети

Рассмотренные ранее архитектуры нейронных сетей плохо подходят для распределенных во времени или последовательных входных данных. Операции над последовательностями выполняются при обработке изображений из видео, при переводе последовательности слов или при отслеживании данных временных рядов. В таких случаях выходные данные зависят не только от самого последнего ввода, но и от предшествующих. Кроме того, эти архитектуры нейронных сетей по своей природе не способны производить выходные данные переменной длины. Например, на основе обычной полносвязной архитектуры очень сложно построить нейронную сеть, которая должна писать эссе на различные темы.

Когда нейронная сеть должна иметь последовательный ввод, последовательный вывод или и то, и другое (рис. D.8), целесообразно использовать *рекур-*

рентную нейронную сеть, выполняющую несколько итераций. Такие нейронные сети хранят рекуррентное состояние **r**, иногда называемое *памятью*, для передачи информации о предыдущих вводах в следующие итерации в течение некоторого времени. Например, в системах машинного перевода слово, встреченное в начале предложения, может влиять на правильность перевода слов в конце предложения. На рис. D.9 показана базовая структура рекуррентной нейронной сети и ее эквивалентное представление в виде более крупной сети, развернутой во времени.

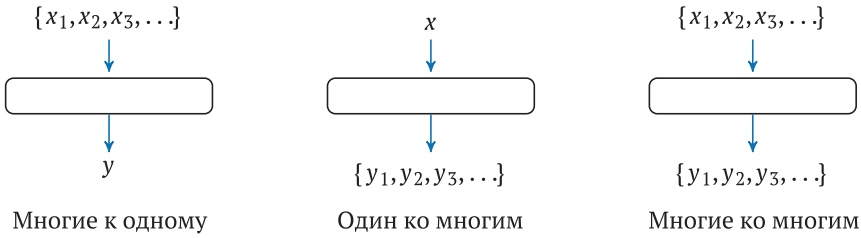


Рис. D.8. Традиционные нейронные сети по своей природе не способны работать с данными переменной длины на входе и выходе

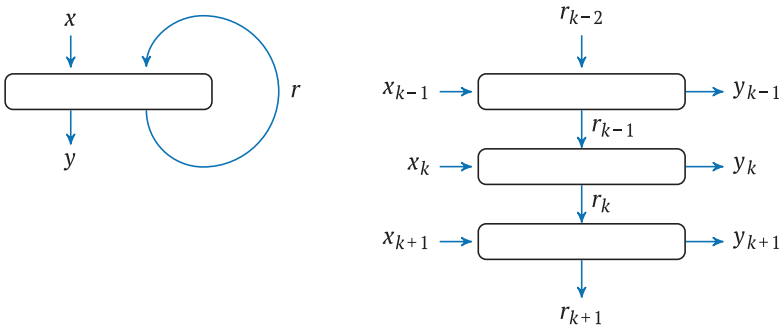
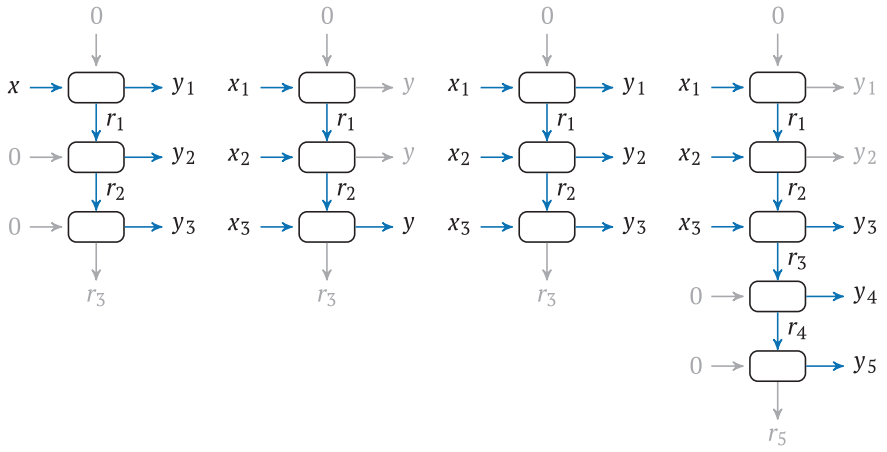


Рис. D.9. Рекуррентная нейронная сеть (слева) и та же рекуррентная нейронная сеть, развернутая во времени (справа). Эти сети хранят рекуррентное состояние **r**, которое позволяет сети развивать своего рода память, передавая информацию через итерации

Развернутую структуру рекуррентной сети можно использовать для создания разнообразных последовательных нейронных сетей, как показано на рис. D.10. Структура «многие ко многим» может иметь разные формы. В одном случае выходная последовательность начинается с входной последовательности. В другом случае выходная последовательность не начинается с входной последовательности. При использовании выходных данных переменной длины сами выходные данные нейронной сети часто содержат указатели на то, когда последовательность начинается или заканчивается. Рекуррентное состояние нередко инициализируют нулем, как и дополнительные входные данные, если входная последовательность слишком коротка, но это не всегда так.



Один ко многим Многие к одному Многие ко многим Многие ко многим

Рис. D.10. Рекуррентную нейронную сеть можно развернуть во времени для создания различных отношений. Неиспользуемые или стандартные входы и выходы показаны серым цветом

Многослойные рекуррентные нейронные сети, развернутые на несколько временных шагов, быстро и эффективно образуют очень глубокую нейронную сеть. Во время обучения градиенты вычисляются относительно функции потерь. По мере удаления от функции потерь влияние слоев на градиент быстро уменьшается. Это приводит к проблеме *исчезающего градиента* (vanishing gradient), когда глубокие нейронные сети имеют исчезающе малые градиенты в своих верхних слоях. Эти небольшие градиенты замедляют обучение.

Очень глубокие нейронные сети также могут страдать от *градиентного взрыва* (exploding gradient), когда последовательные вклады в градиент от каждого слоя складываются и образуют очень большие значения. Слишком большие значения градиента делают обучение нестабильным. В примере D.4 показаны как градиентный взрыв, так и исчезающий градиент.

Пример D.4. Демонстрация того, как в глубоких нейронных сетях происходят исчезновение градиента и градиентный взрыв. В этом примере используется очень простая нейронная сеть. Но эти принципы справедливы и для более крупных полносвязных слоев

Чтобы проиллюстрировать исчезновение и взрыв градиента, рассмотрим глубокую нейронную сеть, состоящую из одномерных, полностью связанных слоев с relu-активациями. Например, если сеть имеет три слоя, ее выход равен

$$f_\theta(x) = \text{relu}(w_3\,\text{relu}(w_2\,\text{relu}(w_1 x_1 + b_1) + b_2) + b_3).$$

Градиент по отношению к функции потерь зависит от градиента f_θ.

Мы получим исчезающие градиенты параметров первого слоя w_1 и b_1, если вклады градиента в последовательных слоях меньше 1. Например, если какой-либо из слоев передает отрицательный вход в свою функцию relu, градиент его входных данных будет равен нулю, поэтому градиент полностью исчезнет. В менее экстремальном случае предположим, что все веса $\mathbf{w} = 0.5\,\mathbf{1}$, все смещения $\mathbf{b} = 0$, а вход x положителен. В этом случае градиент по w_1 равен

$$\frac{\partial f}{\partial w_1} = x_1 \cdot w_2 \cdot w_3 \cdot w_4 \cdot w_5 \ldots$$

Чем глубже сеть, тем меньше будет градиент.

Мы можем получить взрывные градиенты в параметрах первого слоя, если вклад градиента в последовательных слоях больше 1. Если мы просто увеличим наши веса до $\mathbf{w} = 2\,\mathbf{1}$, тот же самый градиент внезапно удвоится на каждом слое.

Хотя проблему градиентного взрыва часто решают с помощью отсечения градиента, регуляризации и инициализации параметров малыми значениями, фактически эти решения просто сдвигают проблему в сторону исчезающих градиентов. Рекуррентные нейронные сети часто используют слои, специально созданные для смягчения проблемы исчезающих градиентов.

Они функционируют, выборочно решая, сохранять ли память, и эти управляющие элементы, называемые *ключами*, или *гейтами* (gate), помогают поддерживать компромисс между памятью и градиентом. К наиболее распространенным рекуррентным уровням относятся *долгая краткосрочная память* (long short-term memory, LSTM)[9] и *управляемые рекуррентные блоки* (gated recurrent unit, GRU)[10].

D.6. Автокодировщики

Нейронные сети часто применяют для обработки многомерных входных данных, таких как изображения или облака точек. Эти многомерные входные данные обычно сильно структурированы, при этом фактическое информационное содержание имеет гораздо более низкую размерность, чем многомерное пространство, в котором оно представлено. Пиксели на изображениях, как правило, сильно коррелируют со своими соседями, а облака точек часто имеют много областей непрерывности. На практике нередко бывает необходимо по-

[9] S. Hochreiter, J. Schmidhuber, *Long Short-Term Memory*, Neural Computation, vol. 9, no. 8, pp. 1735–1780, 1997.

[10] K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, Y. Bengio, *Learning Phrase Representations Using RNN Encoder-Decoder for Statistical Machine Translation*, in Conference on Empirical Methods in Natural Language Processing (EMNLP), 2014.

лучить обобщенное представление об информационном содержании наборов данных, преобразовав их в гораздо меньший набор признаков, который называется *встраиванием* (embedding). Это сжатие, или *обучение представлениям* (representation learning), имеет много преимуществ[11]. Представления меньшей размерности позволяют применить к многомерным данным традиционные методы машинного обучения, такие как байесовские сети, что в противном случае было бы невозможно. Признаки можно исследовать, чтобы понять информационное содержание набора данных, а также использовать в качестве входных данных для других моделей.

Автокодировщик (autoencoder) – это нейронная сеть, обученная находить низкоразмерное представление признаков входных данных более высокого уровня. Сеть автокодировщика принимает многомерный вход $\mathbf{x}$ и создает выход $\mathbf{x}'$ с той же размерностью. Архитектуру сети строят таким образом, чтобы признаки проходили через промежуточное представление меньшей размерности, называемое *сужением* (bottleneck). Активации $\mathbf{z}$ в сужении – это и есть наши низкоразмерные признаки, которые существуют в *скрытом пространстве* (latent space), не наблюдаемом в явном виде. Данная архитектура показана на рис. D.11.

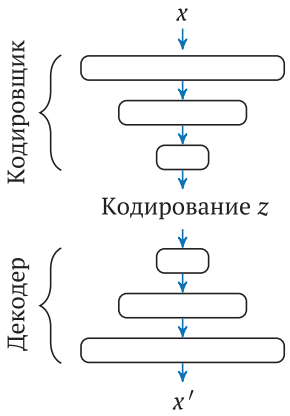


Рис. D.11. Автокодировщик пропускает входные данные высокой размерности через сужение с низкой размерностью, а затем восстанавливает исходные данные из сжатых признаков. Эффективного кодирования признаков с низкой размерностью достигают путем минимизации потерь при восстановлении (сравнивая восстановленные данные с исходными)

Мы обучаем автокодировщик максимально достоверно воспроизводить входные данные на выходе. Например, чтобы добиться от выхода $\mathbf{x}'$ максимального совпадения с входом $\mathbf{x}$, можно просто минимизировать L_2-норму:

$$\underset{\boldsymbol{\theta}}{\text{минимизировать}} \; \underset{\mathbf{x}\in\mathbf{D}}{\mathbb{E}}[||f_{\boldsymbol{\theta}}(\mathbf{x}) - \mathbf{x}||_2]. \tag{D.7}$$

Для более надежного встраивания признаков к входным данным часто добавляют шум:

[11] Уменьшение размерности также может быть выполнено с использованием традиционных методов машинного обучения, таких как метод главных компонент. Нейронные модели обеспечивают большую гибкость и могут обрабатывать нелинейные представления.

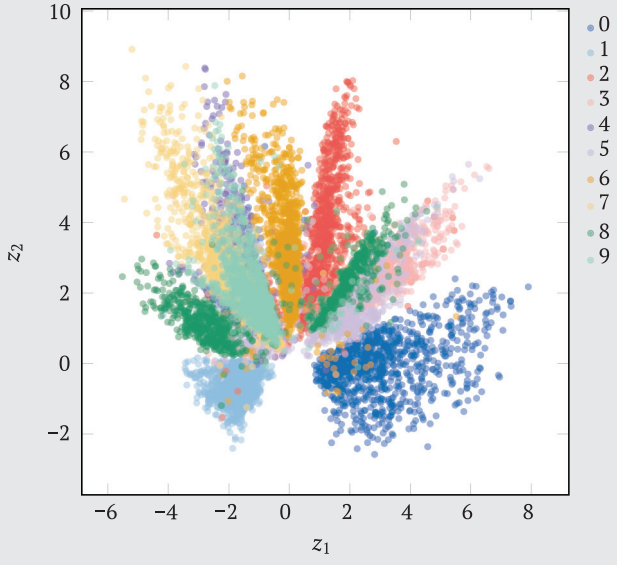
$$\underset{\boldsymbol{\theta}}{\text{минимизировать}} \quad \underset{\mathbf{x}\in\mathbf{D}}{\mathbb{E}}[||f_{\boldsymbol{\theta}}(\mathbf{x}+\boldsymbol{\mu})-\mathbf{x}||_2]. \tag{D.8}$$

Стремление к минимизации потерь при восстановлении заставляет автокодировщик находить наиболее эффективное низкоразмерное представление признаков, достаточное для точного восстановления исходного ввода. Кроме того, при обучении не нужен учитель, поскольку нет необходимости изучить конкретный набор признаков.

После обучения верхнюю часть автокодировщика над сужением можно использовать в качестве отдельного кодировщика, который преобразует входные данные в представление признаков. Нижнюю часть автокодировщика можно использовать в качестве декодера, который преобразует набор признаков во входное представление. Декодирование полезно при обучении нейронных сетей генерации изображений или других многомерных выходных данных. В примере D.5 показано вложение, обученное на рукописных цифрах.

Пример D.5. Визуализация двумерного вложения, полученного для рукописных цифр из набора MNIST

Попробуем применить автокодировщик для обучения встраивания на наборе данных MNIST. В этом примере мы используем кодировщик, аналогичный сверточной сети в примере D.3, за исключением двумерного вывода и отсутствия слоя softmax. Мы создаем декодер, восстанавливающий вход после кодировщика, и обучаем всю сеть, стремясь свести к минимуму потери при восстановлении. На рисунке ниже показаны закодированные представления для 10 000 изображений из набора данных MNIST после обучения. Каждое представление окрашено согласно соответствующей цифре:



Мы видим, что представления цифр образуют кластеры, которые примерно радиально распределены от начала координат. Обратите внимание, что закодированные представления 1 и 7 похожи, поскольку две цифры выглядят почти одинаково. Напомним, что обучение происходит без учителя и сеть не получает извне никакой информации о значениях цифр. Тем не менее она обучается выполнять качественную кластеризацию.

Вариационный автокодировщик (variational autoencoder), показанный на рис. D.12, дополняет структуру автокодировщика для обучения вероятностному кодированию[12]. Вместо того чтобы выводить детерминированную выборку, такой кодировщик создает распределение вероятностей по закодированным представлениям, что позволяет модели присвоить достоверность представлению. Для удобства часто используют многомерные распределения Гаусса с диагональными ковариационными матрицами. В таком случае кодировщик выводит как среднее значение кодированного представления, так и диагональную ковариационную матрицу.

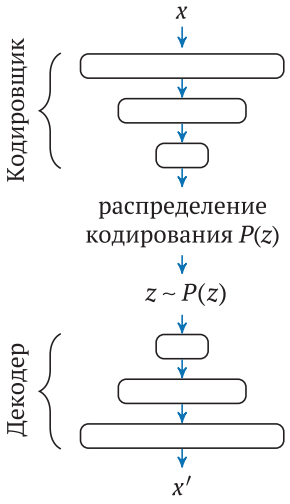


Рис. D.12. Вариационный автокодировщик пропускает входные данные высокой размерности через сужение низкой размерности, которое создает распределение вероятностей при кодировании. Декодер реконструирует выборки из этого закодированного представления, чтобы восстановить исходный вход. Вариационные автокодировщики назначают показатель достоверности каждому закодированному признаку. После этого декодер можно использовать в качестве генеративной модели

Вариационные автокодировщики обучены минимизировать ожидаемые потери при восстановлении, сохраняя компоненты кодирования близкими к распределению Гаусса (с нулевым средним и единичной дисперсией). Первое условие достигается путем взятия одиночной выборки $\mathbf{z} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\sigma}^{\top}\mathbf{I}\boldsymbol{\sigma})$ из распределения кодирования при каждом проходе. Чтобы работало обратное распространение, ко входу обычно примешивают случайный шум $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ и получают окончательную выборку $\mathbf{z} = \boldsymbol{\mu} + \mathbf{w} \odot \boldsymbol{\sigma}$.

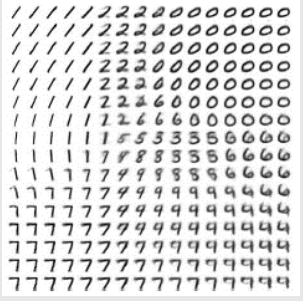
[12] D. Kingma, M. Welling, *Auto-Encoding Variational Bayes*, in International Conference on Learning Representations (ICLR), 2013.

Компоненты остаются близкими к распределению Гаусса за счет минимизации расхождения KL (приложение A.10)[13]. Эта целевая функция поощряет гладкое представление скрытого пространства. Сеть штрафуется за распространение скрытых представлений (большие значения для $||\boldsymbol{\mu}||$) и за концентрацию каждого представления в очень маленьком пространстве кодирования (малые значения для $||\boldsymbol{\sigma}||$), что обеспечивает лучший охват скрытого пространства. В результате плавные изменения в декодере приводят к плавно меняющимся выходным сигналам. Это свойство позволяет использовать декодеры в качестве генеративных моделей, когда выборки из многомерного гауссова распределения подают в декодер для получения реалистичных выборок в исходном пространстве. Комбинированная функция потерь имеет вид:

$$\begin{aligned}&\text{минимизировать}\ \underset{\boldsymbol{\theta}}{\mathbb{E}}\underset{\mathbf{x}\in\mathbf{D}}{}\left[||\mathbf{x}'-\mathbf{x}||_2+c\sum_{i=1}^{|\boldsymbol{\mu}|}D_{\mathrm{KL}}\left(\mathcal{N}(\mu_i,\sigma_i^2)\,||\,\mathcal{N}(0,1)\right)\right],\\&\text{при условии что}\ \boldsymbol{\mu},\boldsymbol{\sigma}=\mathrm{encoder}(\mathbf{x}+\boldsymbol{\varepsilon})\\&\qquad\qquad\mathbf{x}'=\mathrm{decoder}(\boldsymbol{\mu}+\mathbf{w}\odot\boldsymbol{\sigma}),\end{aligned}\tag{D.9}$$

где компромисс между двумя потерями определяется скаляром $c > 0$. Пример D.6 демонстрирует этот процесс на скрытом пространстве, полученном из рукописных цифр.

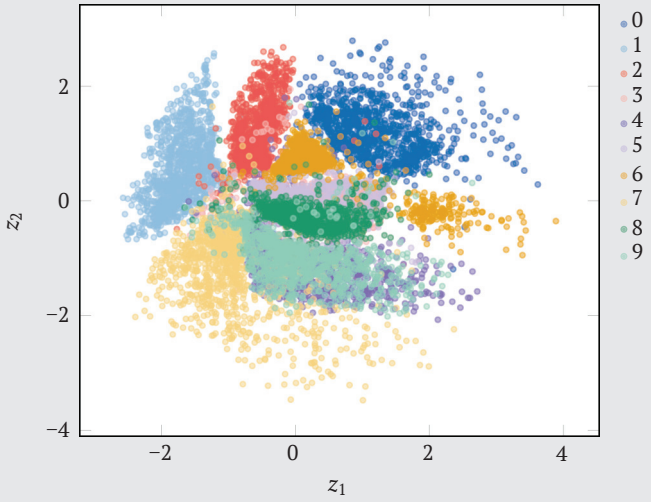
Пример D.6. Визуализация двумерного встраивания, полученного с помощью вариационного автокодировщика для цифр MNIST. На рисунке ниже показаны декодированные выходные данные на основе входных данных, панорамированные по пространству кодирования



В примере D.5 мы обучили автокодировщик набору данных MNIST. Мы можем адаптировать эту сеть для получения двумерных векторов среднего и дисперсии вместо двумерного встраивания, а затем обучить ее таким образом, чтобы минимизировать как потери при восстановлении, так и рас-

[13] Расхождение Кульбака–Лейблера для двух гауссианов равно $\log\left(\frac{\sigma_2}{\sigma_1}\right)+\frac{\sigma_1^2+(\mu_1-\mu_2)^2}{2\sigma_2^2}-\frac{1}{2}$.

хождение KL. Ниже показаны средние кодированные представления для тех же 10 000 изображений из набора данных MNIST. Каждое представление снова окрашено в соответствии с определенной цифрой:



Вариационный автокодировщик тоже создает для каждой цифры кластеры в пространстве встраиваний, но на этот раз они распределены примерно в соответствии с распределением Гаусса с нулевым средним и единичной дисперсией. Мы снова видим, насколько похожи некоторые представления, например для цифр 4 и 9.

Вариационные автокодировщики получаются путем представления кодировщика в виде условного распределения $q(\mathbf{z}|\mathbf{x})$, где $\mathbf{x}$ принадлежит наблюдаемому входному пространству, а $\mathbf{z}$ находится в ненаблюдаемом пространстве встраиваний. Декодер выполняет вывод в обратном направлении, представляя $p(\mathbf{x}|\mathbf{z})$, и в данном случае он также выводит распределение вероятностей. При обучении мы стараемся минимизировать расхождение KL между $q(\mathbf{z}|\mathbf{x})$ и $p(\mathbf{x}|\mathbf{z})$, что равносильно минимизации $\mathbb{E}[\log p(\mathbf{x}|\mathbf{z})] - D_{\mathrm{KL}}(q(\mathbf{z}|\mathbf{x})||p(\mathbf{z}))$, где $p(\mathbf{z})$ – априорное распределение, многомерное по Гауссу, к которому мы стремим наше распределение кодирования. Таким образом, для обучения применяются потери при восстановлении и расхождение KL.

D.7. Состязательные сети

Часто возникает необходимость научить нейронные сети генерировать многомерные выходные данные, такие как изображения или последовательности команд управления вертолетом. Проблема в том, что когда выходное пространство велико, обучающие данные могут охватывать только очень малень-

кую область пространства состояний. Следовательно, обучение исключительно на доступных данных может привести к нереалистичным результатам или переобучению. Обычно все хотят, чтобы обученная нейронная сеть выдавала правдоподобные результаты. При генерации изображений мы хотим получать реалистичные рисунки. При имитации движения автомобиля (например, при имитационном обучении, глава 18) нам нужно, чтобы транспортное средство, как правило, оставалось на своей полосе и реалистично реагировало на другие транспортные средства.

Одним из распространенных подходов к наказанию за нестандартные выходные данные или поведение является использование *состязательного обучения* (adversarial learning) путем добавления *дискриминатора*, как показано на рис. D.13[14]. Дискриминатор – это нейронная сеть, действующая как двоичный классификатор, который принимает выходные данные нейронной сети и учится отличать их от реальных обучающих данных. Первичная нейронная сеть, также называемая *генератором*, в свою очередь обучается обманывать дискриминатор, пытаясь создать выходные данные, которые труднее отличить от обучающего набора. Основное преимущество этого метода заключается в том, что нам не нужно использовать специальные признаки для выяснения или количественного оценивания того, почему выходные данные не соответствуют обучающим данным, – дискриминатор естественным образом находит такие различия в процессе обучения.

Обучение называют состязательным, потому что в процессе обучения соревнуются две нейронные сети: основная, которая учится генерировать реалистичные выходные данные, и сеть дискриминатора, которая учится отличать сгенерированные данные от реальных. Обе сети стараются совершенствовать свои навыки, чтобы превзойти соперника. Обучение – это повторяющийся процесс, в котором каждая сеть совершенствуется по очереди. Иногда бывает трудно найти баланс качества между генератором и дискриминатором; если одна сеть становится слишком хорошей, другая может застрять на одном месте.

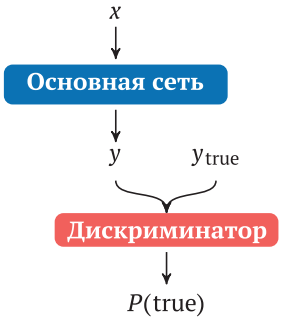


Рис. D.13. Генеративно-состязательная сеть делает вывод основной сети более реалистичным за счет использования дискриминатора, заставляющего основную сеть выдавать более реалистичные данные

[14] Эти методы были представлены в статье I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, Y. Bengio, *Generative Adversarial Nets*, in Advances in Neural Information Processing Systems (NIPS), 2014.

E Алгоритмы поиска

Задача поиска (search problem) заключается в нахождении подходящей последовательности действий для максимизации полученного вознаграждения вследствие последующих детерминированных переходов. Задачи поиска представляют собой марковские процессы принятия решений (часть II этой книги) с детерминированными функциями перехода. К широко известным задачам поиска относятся головоломки со скользящими плитками, кубик Рубика, сокобан и поиск кратчайшего пути к месту назначения.

E.1. Задачи поиска

В задаче поиска агент выбирает действие a_t в момент времени t на основе наблюдения за состоянием s_t, а затем получает вознаграждение r_t. *Пространство действий* $\mathcal{A}$ – это множество возможных действий, а *пространство состояний* $\mathcal{S}$ – это множество возможных состояний. Некоторые алгоритмы предполагают, что эти множества конечны, но в общем случае это не обязательно. Состояние развивается детерминированно и зависит только от текущего состояния и предпринятых действий. Мы используем $\mathcal{A}(s)$ для обозначения набора допустимых действий из состояния s. Когда в определенном состоянии у агента нет допустимых действий, оно считается *поглощающим* (absorbing) и дает нулевое вознаграждение для всех будущих временных шагов. Целевые состояния, как правило, являются поглощающими.

Детерминированная функция перехода состояния $T(s, a)$ дает последующее состояние s'. Функция вознаграждения $R(s, a)$ дает вознаграждение, полученное при выполнении действия a из состояния s. Задачи поиска обычно не содержат коэффициент дисконтирования γ, который снижает вознаграждение в будущем. Для решения задачи поиска необходимо найти последовательность действий, которая максимизирует сумму вознаграждений, или *доход* (return), агента. Алгоритм E.1 содержит структуру данных для представления задач поиска.

Алгоритм E.1. Структура данных задачи поиска

```
struct Search
    𝒮 # пространство состояний
    𝒜 # функция допустимого действия
```

```
    T # функция перехода
    R # функция вознаграждения
end
```

E.2. Графы поиска

Задача поиска с конечными пространствами состояний и действий может быть представлена в виде *графа поиска* (search graph). Узлы графа обозначают состояния, а ребра – переходы между состояниями. С каждым ребром, направленным из исходного в целевое состояние, связано как действие, которое приводит к переходу в это состояние, так и ожидаемое вознаграждение при выполнении этого действия из исходного состояния. На рис. E.1 изображено подмножество такого графа поиска для головоломки со скользящей плиткой на игровом поле 3×3.

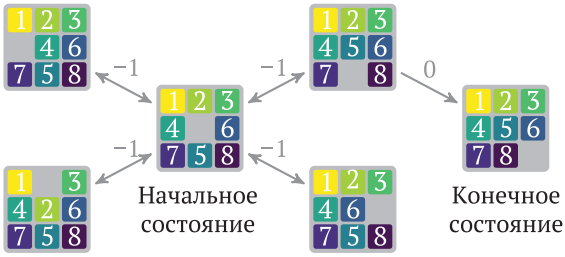


Рис. E.1. Несколько состояний в головоломке со скользящей плиткой, изображенной в виде графа. Для достижения конечного состояния доступны два перехода из начального состояния. Числа возле ребер графа обозначают вознаграждения

Многие алгоритмы поиска по графу выполняют поиск из начального состояния и постепенно расширяют его. При этом эти алгоритмы отслеживают *дерево поиска* (search tree). Начальным состоянием является корневой узел, и каждый раз, когда агент переходит от s к s' во время поиска, к дереву добавляется ребро от s к новому узлу s'. Дерево поиска для той же головоломки со скользящей плиткой показано на рис. E.2.

E.3. Прямой поиск

Возможно, самым простым алгоритмом поиска по графу является *прямой поиск* (forward search, алгоритм E.2), который определяет наилучшее действие из начального состояния s путем просмотра всех возможных переходов действие–состояние до глубины (или горизонта) d. На глубине d алгоритм вычисляет

оценку полезности $U(s)$[1]. Алгоритм рекурсивно вызывает сам себя с приоритетом движения в глубину, строит дерево поиска и возвращает кортеж с оптимальным действием a и его ожидаемой полезностью u на конечном горизонте.

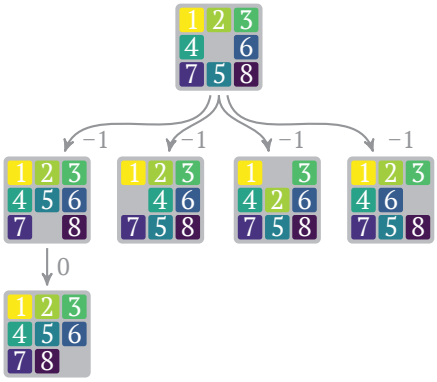


Рис. E.2. Граф поиска для головоломки со скользящей плиткой на поле 3×3 (рис. E.1) может быть представлен как задача поиска по дереву. Поиск начинается с корневого узла и продолжается вниз по дереву. В таком случае агент может пройти весь путь до желаемого конечного состояния

Алгоритм E.2. Алгоритм прямого поиска приблизительно оптимального действия для задачи дискретного поиска $\mathcal{P}$ из текущего состояния `s`. Поиск выполняется на глубину `d`, после чего конечная полезность оценивается с помощью аппроксимированной функции полезности `U`. Возвращаемый именованный кортеж состоит из наилучшего действия `a` и его ожидаемой полезности `u` на конечном горизонте

```
function forward_search(𝒫::Search, s, d, U)
    𝒜, T, R = 𝒫.𝒜(s), 𝒫.T, 𝒫.R
    if isempty(𝒜) || d ≤ 0
        return (a=nothing, u=U(s))
    end
    best = (a=nothing, u=-Inf)
    for a in 𝒜
        s′ = T(s,a)
        u = R(s,a) + forward_search(𝒫, s′, d-1, U).u
        if u > best.u
            best = (a=a, u=u)
        end
    end
    return best
end
```

[1] Ожидается, что аппроксимированные функции полезности в этой главе будут возвращать 0, когда они находятся в состоянии без доступных действий.

На рис. E.3 показан пример дерева поиска, полученного путем прямого поиска в задаче о головоломке со скользящей плиткой. Поиск в глубину может быть расточительным; посещаются все достижимые состояния до заданной глубины. Поиск до глубины d приведет к построению дерева поиска с количеством узлов $O(|\mathcal{A}|)^d$ для задачи с $|\mathcal{A}|$ действиями.

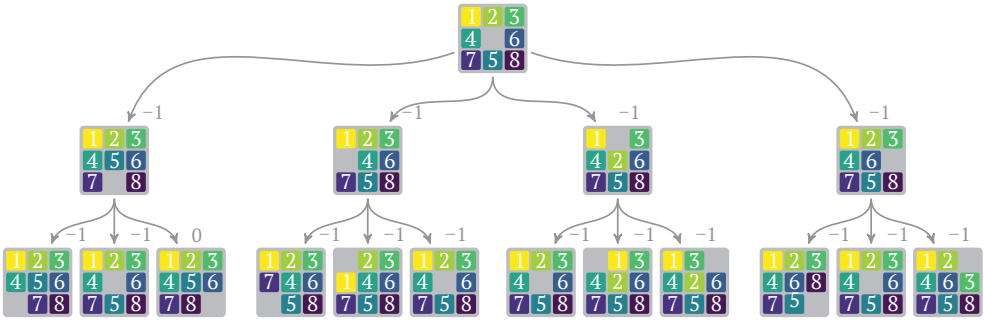


Рис. E.3. Дерево поиска, построенное в результате прямого поиска на глубину 2 в головоломке со скользящей плиткой. Посещаются все состояния, достижимые за два шага, причем некоторые посещаются более одного раза. Мы находим, что существует один путь к конечному узлу. Этот путь дает вознаграждение –1, тогда как все остальные пути дают вознаграждение –2

E.4. Метод ветвей и границ

Широко распространенный метод *ветвей и границ* (алгоритм E.3) может значительно сократить объем вычислений за счет использования информации о верхних и нижних границах ожидаемого вознаграждения, основанной на априорном знании предметной области. Верхняя граница вознаграждения за действие a из состояния s равна $\overline{Q}(s, a)$. Нижняя граница вознаграждения за действие a из состояния s равна $\underline{U}(s)$. Алгоритм ветвей и границ следует той же процедуре, что и поиск в глубину, но перебирает действия в соответствии с их верхней границей вознаграждения и переходит к последующему узлу только в том случае, если наилучшее возможное вознаграждение, которое он может дать, выше, чем то, что уже было обнаружено на предыдущих шагах. В примере E.1 алгоритм прямого поиска сравнивается с алгоритмом ветвей и границ.

Алгоритм E.3. Алгоритм поиска по методу ветвей и границ для нахождения приблизительно оптимального действия в задаче дискретного поиска $\mathcal{P}$ из текущего состояния `s`. Поиск выполняется на глубину `d` с нижней границей функции полезности `Ulo` и верхней границей функции полезности действия `Qhi`. Возвращаемый именованный кортеж состоит из наилучшего действия `a` и его ожидаемой полезности `u` на конечном горизонте

```
function branch_and_bound(𝒫::Search, s, d, Ulo, Qhi)
    𝒜, T, R = 𝒫.𝒜(s), 𝒫.T, 𝒫.R
```

```
    if isempty(𝒜) || d ≤ 0
        return (a=nothing, u=Ulo(s))
    end
    best = (a=nothing, u=-Inf)
    for a in sort(𝒜, by=a->Qhi(s,a), rev=true)
        if Qhi(s,a) ≤ best.u
            return best # защищен от сокращения
        end
        u = R(s,a) + branch_and_bound(𝒫,T(s,a),d-1,Ulo,Qhi).u
        if u > best.u
            best = (a=a, u=u)
        end
    end
    return best
end
```

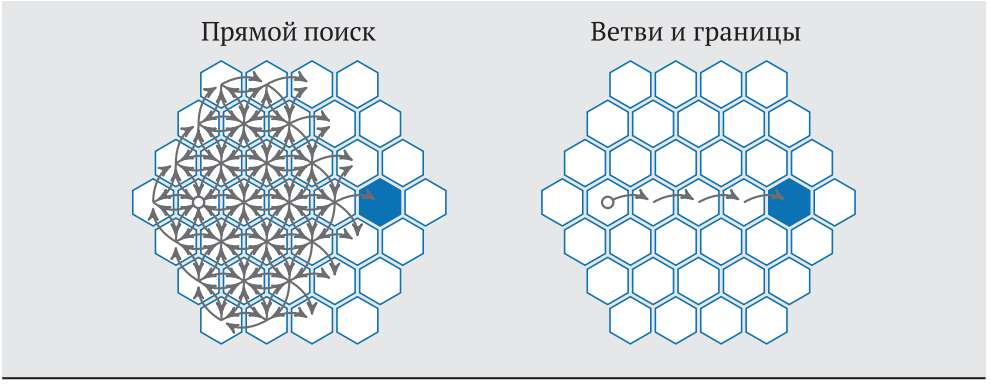
Пример E.1. Демонстрация экономии вычислений, которую дает алгоритм ветвей и границ по сравнению с прямым поиском. При правильном выборе параметров метод ветвей и границ может быть намного эффективнее прямого поиска

Рассмотрим использование метода ветвей и границ для поиска в гексамире. Действия в задачах поиска вызывают детерминированные переходы, поэтому, в отличие от сценария MDP, агент всегда правильно переходит между соседними ячейками, когда выполняется соответствующее действие.

Кружок указывает на начальное состояние. Все переходы влекут за собой отрицательное вознаграждение в размере –1. Синяя ячейка является конечной и дает вознаграждение 5 при попадании на нее.

Ниже показаны деревья поиска как для метода прямого поиска, так и для метода ветвей и границ с глубиной 4. Для метода ветвления и границы были выбраны нижняя граница $\underline{U}(s) = -6$ и верхняя граница $\overline{Q}(s, a) = 5 - \delta(T(s, a))$, где функция $\delta(s)$ – это минимальное количество шагов от заданного состояния до конечного состояния с вознаграждением. Дерево поиска ветвей и границ является подмножеством дерева прямого поиска, потому что алгоритм ветвей и границ может игнорировать части, о которых известно, что они не оптимальны.

Благодаря наличию верхней границы алгоритм сначала оценивает движение вправо, и, поскольку оно оказывается оптимальным, он может сразу определить оптимальную последовательность и избежать изучения других действий. Если бы начальное и целевое состояния поменялись местами, дерево поиска было бы больше. В худшем случае оно может достигать размера дерева прямого поиска.



Алгоритм ветвей и границ не гарантирует сокращения вычислений по сравнению с прямым поиском. В наихудшем случае оба метода имеют одинаковую временную сложность. Эффективность алгоритма сильно зависит от эвристики.

E.5. Динамическое программирование

Рассмотренные в предыдущих разделах алгоритмы поиска не запоминают, посещалось ли состояние ранее; каждый алгоритм тратит вычислительные ресурсы впустую, оценивая состояния по нескольку раз. Метод *динамического программирования* (dynamic programming, алгоритм E.4) позволяет избежать дублирования вычислений, запоминая, какая конкретная подзадача уже решалась ранее. Динамическое программирование применимо к задачам, в которых оптимальное решение может быть построено из оптимальных решений подзадач – свойство, называемое *оптимальной подструктурой* (optimal substructure). Например, если оптимальная последовательность действий от s_1 до s_3 проходит через s_2, то участки пути от s_1 до s_2 и от s_2 до s_3 также оптимальны. Эта подструктура показана на рис. E.4.

Рис. E.4. Последовательность состояний слева направо образует оптимальный путь от начального состояния до конечного состояния. Задачи кратчайшего пути имеют оптимальную подструктуру, а это означает, что последовательность от начального состояния к промежуточному состоянию так же оптимальна, как и последовательность от промежуточного состояния к конечному состоянию

Алгоритм Е.4. Применение метода динамического программирования для прямого поиска с использованием таблицы обращений `M`. В данном случае `M` – это словарь, в котором хранятся кортежи глубина–состояние из предыдущих вычислений, что позволяет методу возвращать ранее вычисленные результаты. Поиск выполняется на глубину `d`, после чего конечное вознаграждение оценивается с помощью аппроксимированной функции полезности `U`. Возвращаемый именованный кортеж состоит из наилучшего действия `a` и его ожидаемой полезности `u` на конечном горизонте

```
function dynamic_programming(𝒫::Search, s, d, U, M=Dict())
    if haskey(M, (d,s))
        return M[(d,s)]
    end
    𝒜, T, R = 𝒫.𝒜(s), 𝒫.T, 𝒫.R
    if isempty(𝒜) || d ≤ 0
        best = (a=nothing, u=U(s))
    else
        best = (a=first(𝒜), u=-Inf)
        for a in 𝒜
            s′ = T(s,a)
            u = R(s,a) + dynamic_programming(𝒫, s′, d-1, U, M).u
            if u > best.u
                best = (a=a, u=u)
            end
        end
    end
    M[(d,s)] = best
    return best
end
```

В случае поиска по графу при оценке состояния мы сначала проверяем таблицу обращений, чтобы увидеть, посещалось ли состояние ранее, и если да, то возвращаем его сохраненную полезность[2]. В противном случае мы оцениваем состояние как обычно и сохраняем результат в таблице. Сравнение этого метода с прямым поиском показано на рис. Е.5.

Е.6. Эвристический поиск

Эвристический поиск[3] (алгоритм Е.5) фактически является усовершенствованным методом ветвей и границ. Он упорядочивает действия при помощи эвристической функции $\overline{U}(s)$, которая является верхней границей вознаграждения. Подобно динамическому программированию, эвристический поиск имеет

[2] Кеширование результатов дорогостоящих вычислений, чтобы их можно было извлечь, а не пересчитывать в будущем, называется *мемоизацией* (memoization).

[3] Эвристический поиск также называют *информированным поиском*, или *поиском по первому наилучшему совпадению*.

механизм кеширования оценок состояния, чтобы избежать избыточных вычислений. Более того, эвристический поиск не нуждается в функции нижней границы полезности, необходимой для метода ветвей и границ[4].

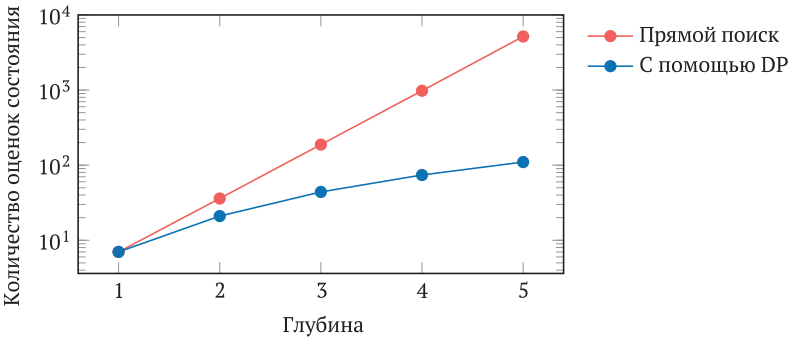


Рис. E.5. Сравнение количества вычислений состояния для простого прямого поиска и поиска, дополненного динамическим программированием, в задаче поиска по гексамиру из примера E.1. Динамическое программирование позволяет избежать экспоненциального роста посещаемости состояний за счет кеширования результатов

Алгоритм E.5. Алгоритм эвристического поиска для решения задачи $\mathcal{P}$, начиная с состояния `s` и до максимальной глубины `d`. Для управления поиском используется эвристика `Uhi`, аппроксимированная функция полезности `U` оценивается в конечных состояниях, а таблица посещений `M` в форме словаря, содержащего кортежи глубина–состояние, позволяет алгоритму кешировать полезности ранее исследованных состояний

```
function heuristic_search(𝒫::Search, s, d, Uhi, U, M)
    if haskey(M, (d,s))
        return M[(d,s)]
    end
    𝒜, T, R = 𝒫.𝒜(s), 𝒫.T, 𝒫.R
    if isempty(𝒜) || d ≤ 0
        best = (a=nothing, u=U(s))
    else
        best = (a=first(𝒜), u=-Inf)
        for a in sort(𝒜, by=a->R(s,a) + Uhi(T(s,a)), rev=true)
            if R(s,a) + Uhi(T(s,a)) ≤ best.u
                break
            end
            s′ = T(s,a)
            u = R(s,a) + heuristic_search(𝒫, s′, d-1, Uhi, U, M).u
            if u > best.u
                best = (a=a, u=u)
```

[4] В нашей реализации используются две функции полезности: эвристика для управления поиском и приближенная функция полезности для оценки конечных состояний.

```
            end
        end
    end
    M[(d,s)] = best
    return best
end
```

Действия сортируются на основе немедленного вознаграждения плюс эвристическая оценка будущего дохода:

$$R(s, a) + \overline{U}(T(s, a)). \tag{E.1}$$

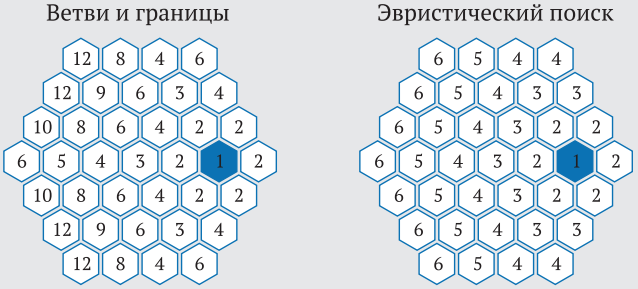
Чтобы гарантировать оптимальность, эвристическая оценка должна быть одновременно *допустимой* (admissible) и *непротиворечивой* (consistent). Допустимая эвристика – это верхняя граница истинно функции полезности. Непротиворечивая эвристика никогда не бывает меньше, чем ожидаемое вознаграждение, полученное при переходе в соседнее состояние:

$$\overline{U}(s) \geq R(s, a) + \overline{U}(T(s, a)). \tag{E.2}$$

Сравнение эвристического поиска с методом ветвей и границ приведено в примере E.2.

Пример E.2. Сравнение эвристического поиска с поиском методом ветвей и границ. Эвристический поиск автоматически упорядочивает действия в соответствии с их упреждающей эвристической полезностью

Мы можем применить эвристический поиск к той же задаче поиска в гексамире, что и в примере E.1. Применим эвристику $\overline{U}(s) = 5 - \delta(s)$, где $\delta(s)$ – количество шагов от заданного состояния до конечного состояния вознаграждения. На рисунке ниже показано количество состояний, посещенных при запуске из начального состояния для метода ветвей и границ (слева) и эвристического поиска (справа). Метод ветвей и границ по-прежнему эффективен в состояниях, близких к целевому состоянию и слева от него, в то время как эвристический поиск эффективно работает из любого начального состояния.



F Задачи принятия решений

В этом приложении рассматриваются задачи принятия решений, используемые в данной книге. В табл. F.1 приведены некоторые важные свойства этих задач.

Таблица F.1. Краткий обзор задач, реализованных в пакете `DecisionMakingProblems.jl`

Задача	$\lvert\mathcal{I}\rvert$	$\lvert\mathcal{S}\rvert$	$\lvert\mathcal{A}\rvert$	$\lvert\mathcal{O}\rvert$	γ
Шестиугольный мир	—	разное	6	—	0.9
Игра 2048	—	∞	4	—	1
Обратный маятник	—	($\subset \mathbb{R}^4$)	2	—	1
Горная машина	—	($\subset \mathbb{R}^2$)	3	—	1
Простой регулятор	—	($\subset \mathbb{R}$)	($\subset \mathbb{R}$)	—	1 или 0.9
Предотвращение столкновений	—	($\subset \mathbb{R}^3$)	3	—	1
Плачущий ребенок	—	2	3	2	0.9
Ремонт машины	—	3	4	2	1
Броски	—	4	10	2	0.9
Дилемма заключенного	2	—	2 / агент	—	1
Камень–ножницы–бумага	2	—	3 / агент	—	1
Дилемма путешественника	2	—	99 / агент	—	1
Хищник–жертва (гексамир)	разное	разное	6 / агент	—	0.9
Плачущий ребенок с няньками	2	2	3 / агент	2 / агент	0.9
Совместная охота в гексамире	разное	разное	6 / агент	2 / агент	0.9

F.1. Задача о шестиугольном мире

Задача о шестиугольном мире (будем для краткости называть его *гексамиром*) – это простая MDP, в которой агент должен пройти по карте из шестиугольных плиток (ячеек), чтобы достичь целевого состояния. Каждая ячейка на мозаичной карте представляет состояние в MDP. Агент может двигаться в любом из шести направлений. Эффекты этих действий являются стохастическими. Как показано на рис. F.1, агент делает 1 шаг в указанном стрелкой направлении с вероятностью 0.7 и делает 1 шаг в одном из соседних направлений, каждое с вероятностью 0.15. Если агент натыкается на внешнюю границу сетки, то он вообще не двигается, и стоимость такого решения равна 1.0.

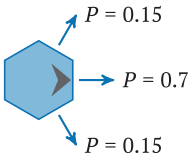


Рис. F.1. Действия в задаче о шестиугольном мире имеют вероятностный характер

Любое действие в заданных ячейках дает агенту определенное вознаграждение, а затем переносит его в конечное состояние. В конечном состоянии дальнейшее вознаграждение не предусмотрено. Таким образом, общее количество состояний в задаче о гексамире равно количеству плиток плюс 1 для конечного состояния. На рис. F.2 показана оптимальная политика для двух конфигураций гексамира, используемых в этой книге. Мы называем более обширный вариант стандартным гексамиром, а меньший, более простой вариант – прямолинейным гексамиром[1]. Вариант прямолинейного гексамира используется для иллюстрации того, как вознаграждение распространяется назад из единственного состояния, приносящего вознаграждение, находящегося в самой правой ячейке.

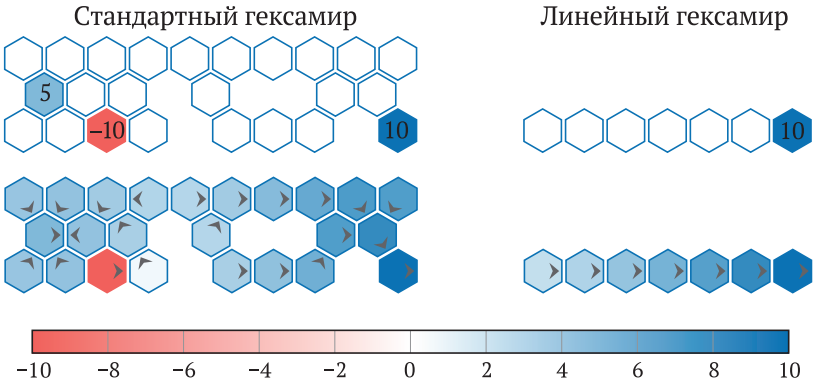


Рис. F.2. Стандартный гексамир и прямолинейный гексамир. В верхнем ряду показана базовая постановка задачи и цветные ячейки с конечными вознаграждениями. В нижней строке показана оптимальная политика для каждой задачи, окрашенная в соответствии с ожидаемой полезностью. Стрелки указывают действие, которое следует предпринять в каждом состоянии

F.2. Задача об игре 2048

Задача об игре 2048 основана на популярной игре с плитками, в которую играют на доске 4×4[2]. Она имеет дискретные пространства состояний и действий. Дос-

[1] Линейная формулировка аналогична *задаче о комнате* (hall problem) – обычному стандартному MDP. См., например, L. Baird, *Residual Algorithms: Reinforcement Learning with Function Approximation*, in International Conference on Machine Learning (ICML), 1995.

[2] Эта игра была разработана Габриэле Чирулли в 2014 году.

ка изначально пуста, за исключением двух плиток, каждая из которых может иметь значение 2 или 4. Случайно выбранное начальное состояние показано на рис. F.3.

Рис. F.3. Случайное начальное состояние в задаче 2048 состоит из двух плиток, каждая со значением 2 или 4

Агент может перемещать все плитки влево (`left`), вниз (`down`), вправо (`right`) или вверх (`up`). Выбор направления приводит к перемещению всех плиток в этом направлении. Плитка останавливается, когда сталкивается с краем поля или с плиткой, имеющей другое значение. Плитка, которая касается другой плитки с таким же значением, сливается с этой плиткой, образуя новую плитку с суммарным значением. После смещения и слияния в случайном месте свободного игрового поля возникает новая плитка со значением 2 или 4. Этот процесс показан на рис. F.4.

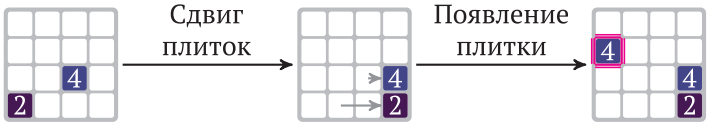


Рис. F.4. Действие в игре 2048 сдвигает все плитки в выбранном направлении, а затем создает новую плитку на пустом месте

Игра заканчивается, когда агент больше не может сдвигать плитки, чтобы образовалось пустое место. Вознаграждение получается только при слиянии двух плиток, и оно равно стоимости получившейся плитки. Пример перехода состояние-действие со слиянием показан на рис. F.5.

Рис. F.5. Здесь действие `down` используется для смещения всех плиток, что приводит к слиянию двух плиток 4 для получения плитки 8 и получения вознаграждения в размере 8

Распространенная стратегия состоит в том, чтобы выбрать угол и чередовать два действия, ведущих в направлении этого угла. Это ведет к разделению плиток таким образом, что плитки с большим значением находятся в углу, а недавно созданные плитки – на периферии.

F.3. Задача об обратном маятнике

В *задаче об обратном маятнике*[3], также иногда называемой задачей о балансировке шеста, агент перемещает тележку вперед и назад. Как показано на рис. F.6, эта тележка имеет шарнир, к которому прикреплен жесткий шест, так что при движении тележки вперед и назад шест начинает отклоняться. Цель состоит в том, чтобы удерживать шест в вертикальном равновесии, не позволяя тележке выходить за пределы допустимого бокового смещения. Таким образом, агент получает вознаграждение 1 на каждом временном шаге, на котором выполняются эти условия, а переход в конечное состояние с нулевым вознаграждением происходит всякий раз, когда они не выполняются.

Действия заключаются в приложении к тележке силы F слева или справа. Пространство состояний определяется четырьмя непрерывными переменными: боковым положением тележки x, ее поперечной скоростью v, углом наклона шеста θ и угловой скоростью шеста w. Условие задачи включает различные параметры, а именно массу тележки m_{cart}, массу шеста m_{pole}, длину шеста ℓ, величину силы $|F|$, ускорение свободного падения g, шаг времени Δt, максимальное отклонение по x, максимальное угловое отклонение и потери на трение между тележкой и шестом или между тележкой и опорной поверхностью[4].

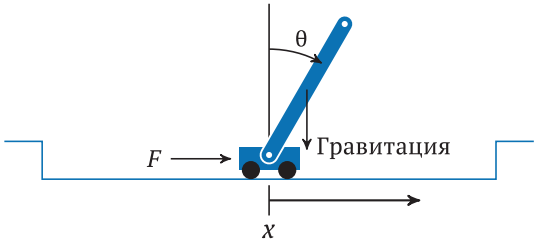


Рис. F.6. В задаче об обратном маятнике необходимо чередовать ускорение тележки влево и вправо, чтобы удерживать шест как можно ближе к вертикальному положению. Шест не может отклониться больше, чем на заданный угол, а тележка не может выехать за заданные боковые пределы

При приложенной силе F угловое ускорение шеста равно

$$\alpha = \frac{g\sin(\theta) - \tau\cos(\theta)}{\frac{\ell}{2}\left(\frac{4}{3} - \frac{m_{\text{pole}}}{m_{\text{cart}} + m_{\text{pole}}}\cos(\theta)^2\right)}, \tag{F.1}$$

---

[3] A. G. Barto, R. S. Sutton, C. W. Anderson, *Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems*, IEEE Transactions on Systems, Man, and Cybernetics, vol. SMC-13, no. 5, pp. 834–846, 1983.

[4] Мы используем параметры, реализованные в OpenAI Gym. G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, W. Zaremba, *OpenAI Gym*, 2016. arXiv: 1606.01540v1.

где

$$\tau = \frac{F + \omega^2 \ell \sin\theta/2}{m_{\text{cart}} + m_{\text{pole}}}, \tag{F.2}$$

а боковое ускорение тележки равно

$$a = \tau - \frac{\ell}{2}\alpha\cos(\theta)\frac{m_{\text{pole}}}{m_{\text{cart}} + m_{\text{pole}}}. \tag{F.3}$$

Состояние обновляется с помощью интегрирования Эйлера:

$$\begin{aligned} x &\leftarrow x + v\Delta t; \\ v &\leftarrow v + a\Delta t; \\ \theta &\leftarrow \theta + \omega\Delta t; \\ \omega &\leftarrow \omega + \alpha\Delta t. \end{aligned} \tag{F.4}$$

Задача об обратном маятнике обычно инициализируется случайными значениями, взятыми из $U(-0.05, 0.05)$. Перемещения тележки выполняют до тех пор, пока не будут превышены боковые или угловые отклонения.

F.4. Задача о горной машине

В *задаче о горной машине*[5] транспортное средство должно двигаться вправо из ложбины между двумя склонами холмов. Склоны настолько круты, что попытка прямого движения к цели с недостаточной скоростью приводит к тому, что машина останавливается на склоне и соскальзывает обратно. Агент должен научиться сначала заезжать на левый склон, чтобы на обратном пути набрать достаточную скорость и подняться в гору справа.

Состояние в данной задаче – это горизонтальное положение автомобиля $x \in [-1.2, 0.6]$ и его скорость $v \in [-0.07, 0.07]$. На любом заданном временном шаге транспортное средство может ускоряться влево ($a = -1$), ускоряться вправо ($a = 1$) или двигаться по инерции ($a = 0$). Агент получает вознаграждение –1 за каждый ход и прекращает работу, когда машина преодолевает правый склон с $x = 0.6$. Задача представлена в графическом виде на рис. F.7.

[5] Эта задача была рассмотрена в работе Moore, *Efficient Memory-Based Learning for Robot Control*, Ph.D. dissertation, University of Cambridge, 1990. Популярная и более простая форма с дискретным пространством действий была впервые предложена в S. P. Singh, R. S. Sutton, *Reinforcement Learning with Replacing Eligibility Traces*, Machine Learning, vol. 22, pp. 123–158, 1996.

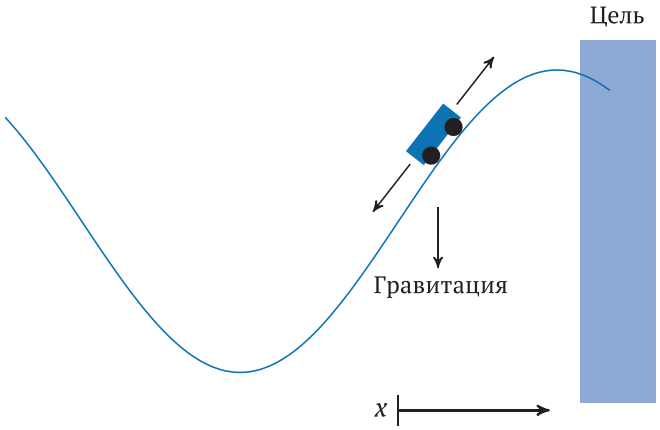


Рис. F.7. Горная машина должна чередовать ускорение влево и вправо, чтобы подняться по правому склону. Область цели показана закрашенным прямоугольником

Переходы в задаче о горной машине детерминированы:

$$v' \leftarrow v + 0.001a - 0.0025\cos(3x);$$

$$x' \leftarrow x + v'.$$

Член гравитации в формуле обновления скорости – это компонент, который отбрасывает маломощную машину обратно ко дну долины. Переходы привязаны к границам пространства состояний.

Задача о горной машине – хороший пример *задачи с отложенным доходом*. Чтобы достичь целевого состояния, требуется множество действий, что затрудняет для неподготовленного агента получение чего-либо, кроме постоянных штрафов. Лучшие алгоритмы обучения способны эффективно распространять знания от траекторий, ведущих к цели, назад в остальную часть пространства состояний.

F.5. Задача о простом регуляторе

Задача о простом регуляторе – это несложная задача линейного квадратичного регулятора с одним состоянием. Это MDP с одним действительным состоянием и одним допустимым действием. Переходы являются линейными по Гауссу, так что последующее состояние s' получается из гауссова распределения $\mathcal{N}(s + a, 0.1^2)$. Вознаграждение вычисляется по квадратичной формуле $R(s, a) = -s^2$ и не зависит от действия. В примерах в этой книге используется распределение начального состояния $\mathcal{N}(0.3, 0.1^2)$.

Оптимальные стратегии конечного горизонта не могут быть получены с использованием методов из раздела 7.8. В этом случае $\mathbf{T}_s = [1]$, $\mathbf{T}_a = [1]$, $\mathbf{R}_s = [-1]$,

$\mathbf{R}_a = [0]$, а w берется из распределения $\mathcal{N}(0, 0.1^2)$. Уравнение Риккати требует, чтобы $\mathbf{R}_a$ было отрицательно определено, что в данном случае не так.

Оптимальная стратегия $\pi(s) = -s$, что приводит к распределению последующих состояний с центром в начале координат. В главах, посвященных градиентному поиску стратегий, мы изучили параметрические стратегии вида $\pi_{\boldsymbol{\theta}}(s) = \mathcal{N}(\theta_1 s, \theta_2^2)$. В таких случаях оптимальная параметризация для задачи о простом регуляторе будет следующей: $\theta_1 = -1$, а θ_2 асимптотически стремится к нулю.

Оптимальная функция полезности для задачи о простом регуляторе также сосредоточена вокруг начала координат, при этом вознаграждение уменьшается по квадратичному закону:

$$U(s) = -s^2 + \frac{\gamma}{1-\gamma}\mathbb{E}_{s\sim\mathcal{N}(0,0.1^2)}[-s^2]$$
$$\approx -s^2 - 0.010\frac{\gamma}{1-\gamma}.$$

F.6. Задача о предотвращении столкновения самолетов

Задача о предотвращении столкновения самолетов заключается в принятии решения о том, когда выдавать воздушному судну рекомендацию о наборе высоты или снижении, чтобы избежать столкновения со встречным самолетом[6]. Агенту доступны три действия: отсутствие рекомендации, команда снижения со скоростью 5 м/с и команда набора высоты со скоростью 5 м/с. Встречный самолет приближается строго встречным курсом с постоянной горизонтальной скоростью сближения. Состояние определяется высотой h нашего самолета, измеренной относительно встречного самолета, нашей вертикальной скоростью $\dot{h}$, предыдущим действием a_{prev} и временем до потенциального столкновения t_{col}. Графический сценарий задачи показан на рис. F.8.

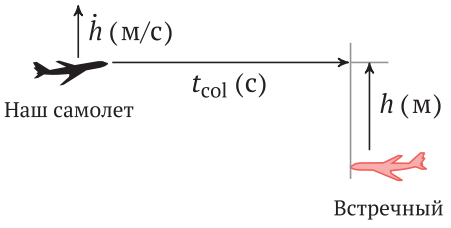


Рис. F.8. Переменные состояния в задаче о предотвращении столкновения самолетов

[6] Эта формулировка является сильно упрощенной версией задачи, описанной в M. J. Kochenderfer, J. P. Chryssanthacopoulos, *Robust Airborne Collision Avoidance Through Dynamic Programming*, Massachusetts Institute of Technology, Lincoln Laboratory, Project Report ATC-371, 2011.

При выполнении действия a переменные состояния обновляются следующим образом:

$$h \leftarrow h + \dot{h}\Delta t; \tag{F.5}$$

$$\dot{h} \leftarrow \dot{h} + (\ddot{h} + v)\Delta t; \tag{F.6}$$

$$a_{\text{prev}} \leftarrow a; \tag{F.7}$$

$$t_{\text{col}} \leftarrow t_{\text{col}} - \Delta t, \tag{F.8}$$

где $\Delta t = 1$ с, а v выбирается из дискретного распределения -2, 0 или 2 м/с^2 с соответствующими вероятностями 0.25, 0.5 и 0.25. Значение $\ddot{h}$ определяется выражением

$$\ddot{h} = \begin{cases} 0, & \text{если } a = \text{нет рекомендаций} \\ a/\Delta t, & \text{если } |a - \dot{h}|/\Delta t < \ddot{h}_{\text{limit}} \\ \text{sign}(a - \dot{h})\ddot{h}_{\text{limit}} & \text{в ином случае} \end{cases}, \tag{F.9}$$

где $\ddot{h}_{\text{limit}} = 1$ м/с^2.

Задача завершается при совершении действия, ведущего к $t_{\text{col}} < 0$. Предусмотрен штраф 1, когда встречный самолет приближается на 50 м (когда $t_{\text{col}} = 0$), и штраф 0,01, когда $a \neq a_{\text{prev}}$.

Задача о предотвращении столкновения самолетов может быть эффективно решена на дискретизированной сетке с использованием итерации по полезности путем обратной индукции (раздел 7.6), поскольку динамика детерминированно снижает t_{col}. Срезы оптимальной функции полезности и стратегии изображены на рис. F.9.

F.7. Задача о плачущем ребенке

Задача о плачущем ребенке[7] – это простая POMDP с двумя состояниями, тремя действиями и двумя наблюдениями. Наша цель – заботиться о ребенке, и мы делаем это, выбирая на каждом временном шаге, накормить ли его, спеть ли ему колыбельную или игнорировать.

Ребенок со временем становится голодным. Мы не можем наблюдать напрямую, голоден ли ребенок; вместо этого мы получаем зашумленное наблюдение о том, плачет ли он. Пространства состояния, действия и наблюдения следующие:

$\mathcal{S}$ = {сытый, голодный};

$\mathcal{A}$ = {кормить, петь, игнорировать};

$\mathcal{O}$ = {плачет, молчит}.

[7] Вариант задачи о плачущем ребенке, представленный в этой книге, является расширением оригинальной, более простой задачи о плачущем ребенке из книги M. J. Kochenderfer, *Decision Making Under Uncertainty: Theory and Application*. MIT Press, 2015.

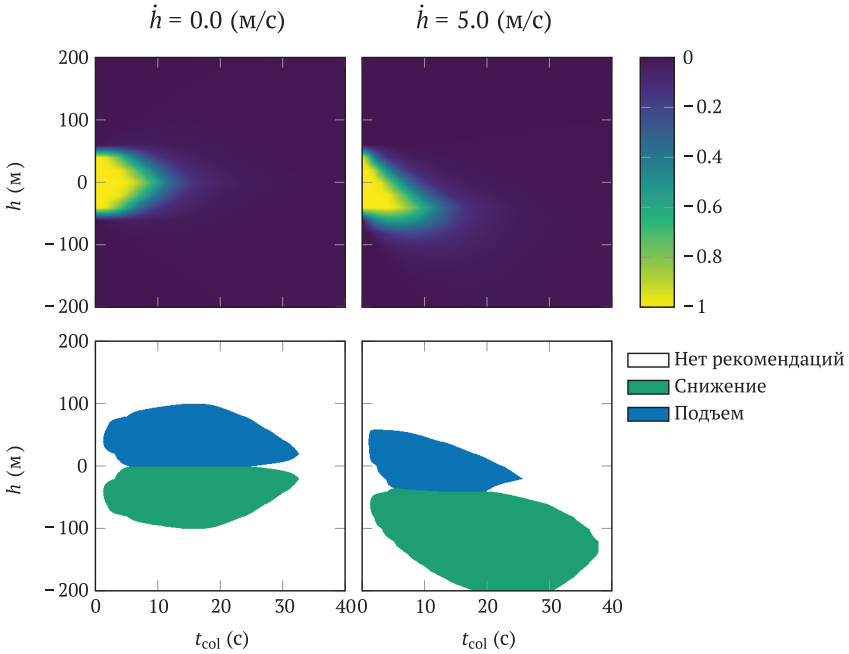


Рис. F.9. Срезы оптимальной функции полезности (вверху) и стратегии (внизу) для задачи о предотвращении столкновения самолетов. Функция полезности и стратегия симметричны относительно 0, когда коэффициент вертикального разделения равен 0, но искажены, когда коэффициент вертикального разделения отличен от нуля. В целом нашему самолету не нужно предпринимать никаких действий, пока встречный самолет не окажется достаточно близко

Кормление всегда насыщает ребенка. Игнорирование может привести к тому, что сытый ребенок со временем станет голодным, а голодный ребенок останется голодным. Пение ребенку – это действие по сбору информации с той же динамикой перехода, что и игнорирование, но после пения ребенок гарантированно перестает плакать, если он сыт (не голоден), и с высокой вероятностью продолжает плакать, если он голоден.

Динамика перехода следующая:

T(сыт | голоден, кормить) = 100 %;

T(голоден | голоден, петь) = 100 %;

T(голоден | голоден, игнорировать) = 100 %;

T(сыт | сытый, кормить) = 100 %;

T(голоден | сыт, петь) = 10 %;

T(голоден | сыт, игнорировать) = 10 %.

Динамика наблюдения следующая:

O(плачет | кормить, голоден) = 80 %;

O(плачет | петь, голоден) = 90 %;

O(плачет | игнорировать, голоден) = 80 %;

O(плачет | кормить, сыт) = 10 %;

O(плачет | петь, сыт) = 0 %;

O(плачет | игнорировать, сыт) = 10 %.

Функция вознаграждения назначает награду –10, если ребенок голоден, независимо от предпринятого действия. Кормление ребенка добавляет вознаграждение –5, тогда как пение добавляет вознаграждение –0.5. Агент, ухаживающий за ребенком, ищет оптимальную стратегию бесконечного горизонта с коэффициентом дисконтирования $\gamma = 0.9$. На рис. F.10 показаны оптимальная функция полезности и связанная с ней стратегия

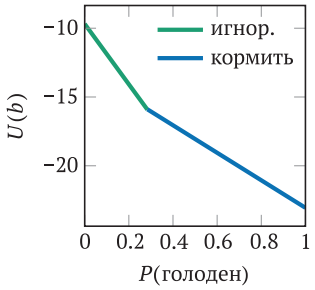


Рис. F.10. Оптимальная стратегия для задачи о плачущем ребенке. Это решение с бесконечным горизонтом не рекомендует пение для любого состояния-убеждения. Как показано на рис. 20.3, пение является оптимальным для некоторых вариантов этой задачи с конечным горизонтом

F.8. Задача о ремонте машины

Задача о ремонте машины представляет собой дискретную задачу POMDP, в которой агент занят обслуживанием машины, выпускающей некую продукцию[8]. Эта задача часто используется из-за ее относительной простоты и различных размеров и форм областей оптимальной стратегии. Оптимальная стратегия для определенных горизонтов даже может иметь непересекающиеся области, в которых одно и то же действие является оптимальным, как показано на рис. F.11.

[8] R. D. Smallwood, E. J. Sondik, *The Optimal Control of Partially Observable Markov Processes over a Finite Horizon*, Operations Research, vol. 21, no. 5, pp. 1071–1088, 1973. Первоначальная формулировка задачи включает *ликвидационную стоимость* или *конечное вознаграждение*, равное количеству исправных частей. В этой книге мы не моделируем конечное вознаграждение отдельно. Конечные вознаграждения можно было бы рассматривать, явно включив горизонт в условие задачи.

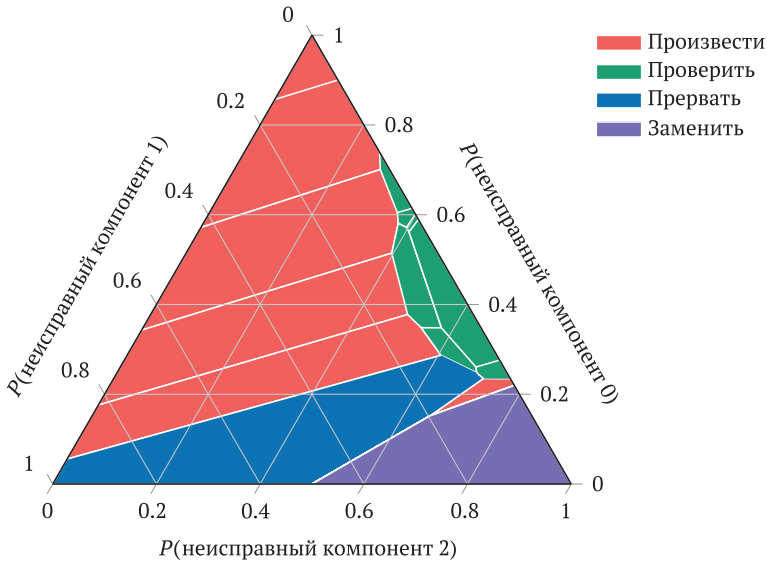


Рис. F.11. Оптимальная политика из 14 шагов для задачи о ремонте машины имеет непересекающиеся области, в которых производство является оптимальным. Каждый многоугольник соответствует области, в которой преобладает определенный альфа-вектор

Машина производит качественную продукцию, когда работает исправно. Со временем два основных компонента машины могут выйти из строя вместе или по отдельности, что приведет к выпуску бракованной продукции. Мы можем косвенно наблюдать за исправностью машины, исследуя продукцию, или напрямую – исследуя компоненты машины.

Задача имеет состояния $\mathcal{S} = \{0, 1, 2\}$, соответствующие количеству неисправных внутренних компонентов. В каждом производственном цикле возможны четыре действия:

1) *произвести* – произвести продукцию и не проверять ее качество;
2) *проверить* – произвести продукцию и проверять качество;
3) *прервать* – прервать выпуск продукции, проверить и заменить неисправные компоненты;
4) *заменить* – заменить оба компонента после остановки производства.

Когда мы проверяем продукцию, мы можем наблюдать, является ли она бракованной. Все остальные действия подразумевают качественную продукцию.

Компоненты машины независимо друг от друга имеют вероятность 10%-го выхода из строя при каждом производственном цикле. Каждый неисправный компонент увеличивает вероятность выпуска бракованной продукции с показателем 50 %. Качественная продукция дает вознаграждение 1, а бракованная – 0. Динамика перехода предполагает, что поломка компонентов определяется до того, как продукция будет изготовлена, поэтому действие «Произвести» на полностью исправной машине не имеет 100%-ной вероятности получения вознаграждения 1.

Действие «Произвести» не влечет за собой штрафа. Проверка качества продукции стоит 0.25. Прерывание производства обходится в 0.5 за осмотр машины, приводит к остановке выпуска продукции и влечет за собой затраты 1 за каждый сломанный компонент. Простая замена обоих компонентов всегда стоит 2, но не требует затрат на осмотр.

Функции перехода, наблюдения и вознаграждения приведены в табл. F.2. Оптимальные стратегии для увеличивающихся горизонтов показаны на рис. F.12.

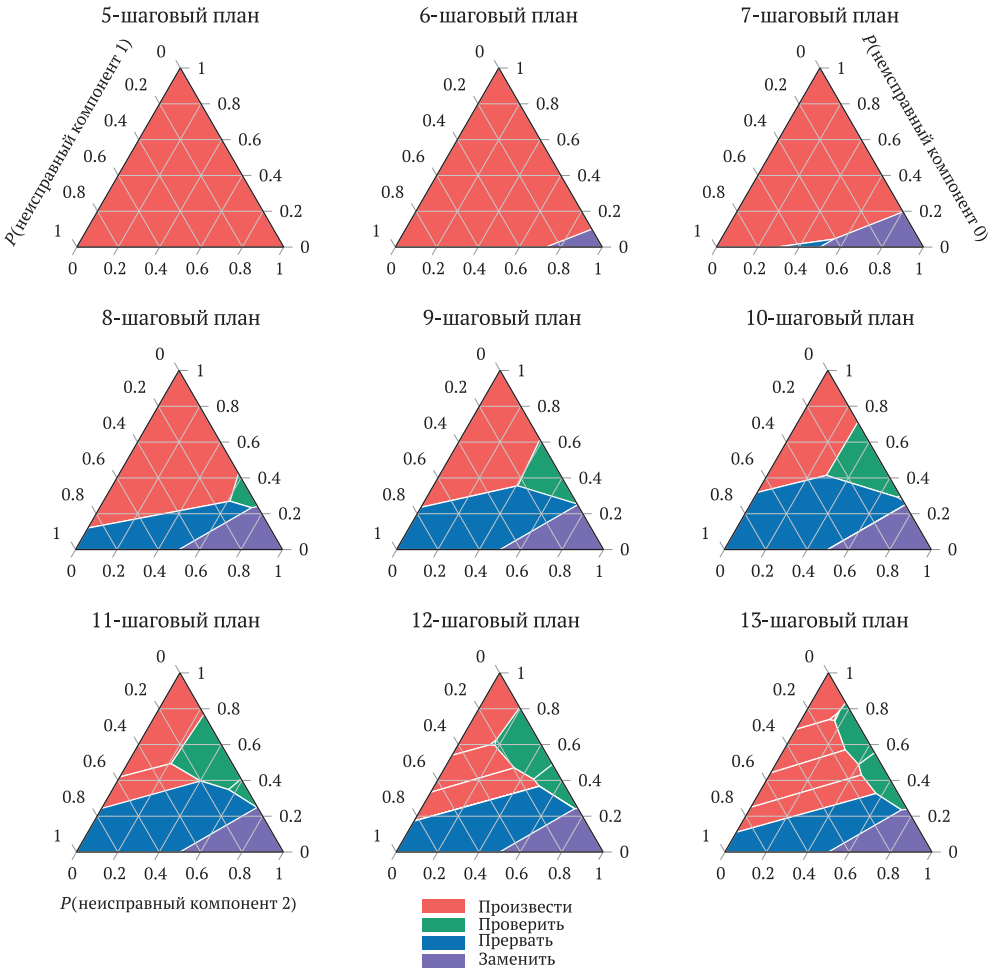


Рис. F.12. Оптимальные стратегии для задачи о ремонте машины. Каждый многоугольник соответствует области, в которой доминирует определенный альфа-вектор

Таблица F.2. Функции перехода, наблюдения и вознаграждения для задачи о ремонте машины

Действие	$T(s'\|s, a)$	$O(o\|a, s')$	$R(s, a)$
Производить	s' $s\begin{bmatrix}0.81 & 0.18 & 0.01\\ 0 & 0.9 & 0.1\\ 0 & 0 & 1\end{bmatrix}$	o $s'\begin{bmatrix}1 & 0\\ 1 & 0\\ 1 & 0\end{bmatrix}$	$\begin{bmatrix}0.9025\\ 0.475\\ 0.25\end{bmatrix}$
Проверять	$s\begin{bmatrix}0.81 & 0.18 & 0.01\\ 0 & 0.9 & 0.1\\ 0 & 0 & 1\end{bmatrix}$	$s'\begin{bmatrix}1 & 0\\ 0.5 & 0.5\\ 0.25 & 0.75\end{bmatrix}$	$\begin{bmatrix}0.6525\\ 0.225\\ 0\end{bmatrix}$
Прервать	$s\begin{bmatrix}1 & 0 & 0\\ 1 & 0 & 0\\ 1 & 0 & 0\end{bmatrix}$	$s'\begin{bmatrix}1 & 0\\ 1 & 0\\ 1 & 0\end{bmatrix}$	$\begin{bmatrix}-0.5\\ -1.5\\ -2.5\end{bmatrix}$
Заменить	$s\begin{bmatrix}1 & 0 & 0\\ 1 & 0 & 0\\ 1 & 0 & 0\end{bmatrix}$	$s'\begin{bmatrix}1 & 0\\ 1 & 0\\ 1 & 0\end{bmatrix}$	$\begin{bmatrix}-2\\ -2\\ -2\end{bmatrix}$

F.9. Задача о бросках мяча

В *задаче о бросках мяча* Джонни старается поймать мяч, который бросает его отец, и предпочитает ловить броски с дальней дистанции. Однако он не уверен в связи между расстоянием броска и вероятностью поимки мяча. Он знает также, что вероятность поимки мяча не зависит от того, кто кому бросает мяч; и у него есть конечное число попыток, чтобы максимизировать ожидаемую полезность, прежде чем его позовут домой.

Как показано на рис. F.13, Джонни моделирует вероятность успешной поимки мяча, брошенного на расстояние d, как

$$P(\text{catch}|d) = 1 - \frac{1}{1 + \exp\left(-\frac{d-s}{15}\right)}, \tag{F.10}$$

где мастерство s неизвестно и не меняется со временем. Чтобы не усложнять задачу, он предполагает, что s принадлежит дискретному множеству $\mathcal{S}$ = {20, 40, 60, 80}.

Вознаграждение за удачную поимку мяча равно расстоянию. Если мяч не пойман, то вознаграждение равно нулю. Джонни хочет максимизировать вознаграждение за конечное число попыток бросков. При каждом броске Джонни выбирает расстояние из дискретного множества $\mathcal{A}$ = {10, 20, ..., 100}. Он начинает с равномерного распределения вероятностей по $\mathcal{S}$.

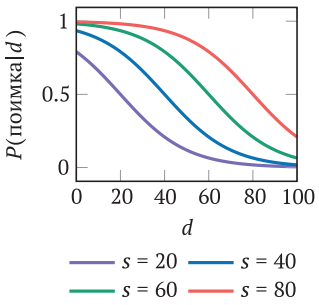


Рис. F.13. Вероятность поимки мяча как функция дальности броска d для четырех уровней мастерства s

F.10. Дилемма заключенного

Дилемма заключенного – классическая задача теории игр, в которой участвуют агенты с противоречивыми целями. Есть двое заключенных, которые готовятся к суду. Они могут выбрать сговор между собой и не признаваться в общем преступлении, или отказаться от сговора, обвинив другого в преступлении. Если оба заключенных придерживаются сговора и не сознаются, они оба отбывают срок в один год. Если агент i придерживается договоренности, а другой агент $-i$ отказывается от нее, то i отсидит четыре года, а $-i$ будет освобожден. Если оба откажутся от сговора, то получат по три года[9].

В игре есть два агента, $\mathcal{I} = \{1, 2\}$ и $\mathcal{A} = \mathcal{A}^1 \times \mathcal{A}^2$, каждый из которых $\mathcal{A}^i = \{$сговорился, отказался$\}$. На рис. F.14 представлены индивидуальные вознаграждения. Строки представляют действия для агента 1. Столбцы представляют действия для агента 2. Вознаграждения для агентов 1 и 2 показаны в каждой ячейке: $R^1(a^1, a^2)$, $R^2(a^1, a^2)$. В игру можно играть один раз или повторять произвольное количество раз. В случае бесконечного горизонта мы используем коэффициент дисконтирования $\gamma = 0.9$.

		Агент 2	
		сговор	отказ
Агент 1	сговор	−1, −1	−4, 0
	отказ	0, −4	−3, −3

Рис. F.14. Вознаграждения, связанные с дилеммой заключенного

[9] А. В. Такер придумал название и сюжет этой игры. Он был основан на первоначальной формулировке задачи Меррилла Флада и Мелвина Дрешера из RAND, предложенной в 1950 г. История опубликована в книге W. Poundstone, *Prisoner's Dilemma*. Doubleday, 1992.

F.11. Камень–ножницы–бумага

Одна из самых распространенных игр, в которую играют во всем мире, – «камень–ножницы–бумага». Есть два агента, каждый из которых может выбрать камень, ножницы или бумагу. Камень побеждает ножницы, в результате чего агент, выбравший камень, получает вознаграждение 1, а агент, выбравший ножницы, получает штраф 1. Ножницы побеждают бумагу, в результате чего агент, выбравший ножницы, получает вознаграждение 1, а агент, выбравший бумагу, получает штраф 1. Наконец, бумага побеждает камень, в результате чего агент, выбравший бумагу, получает вознаграждение 1, а агент, выбравший камень, получает штраф 1.

У нас есть $\mathcal{I} = \{1, 2\}$ и $\mathcal{A} = \mathcal{A}^1 \times \mathcal{A}^2$ для каждого $\mathcal{A}^i = \{$камень, ножницы, бумага$\}$. На рис. F.15 показаны вознаграждения, связанные с игрой, где каждая ячейка обозначает $R^1(a^1, a^2), R^2(a^1, a^2)$. В игру можно играть один раз или повторять произвольное количество раз. В случае бесконечного горизонта мы используем коэффициент дисконтирования $\gamma = 0.9$.

Агент 1 \ Агент 2	камень	бумага	ножницы
камень	0, 0	−1, 1	1, −1
бумага	1, −1	0, 0	−1, 1
ножницы	−1, 1	1, −1	0, 0

Рис. F.15. Вознаграждения, связанные с игрой «камень–ножницы–бумага»

F.12. Дилемма путешественника

Дилемма путешественника – это игра, в которой авиакомпания теряет два одинаковых чемодана у двух путешественников[10]. Авиакомпания просит путешественников указать стоимость их чемоданов, которая может составлять от 2 до 100 долларов включительно. Если путешественники указали одинаковую стоимость, то они оба получают выплату, равную этой стоимости. В противном случае путешественник, указавший меньшую стоимость, получает эту стоимость плюс 2 доллара, а путешественник, указавший большую стоимость, получает меньшую стоимость минус 2 доллара. Другими словами, функция вознаграждения выглядит следующим образом:

[10] K. Basu, *The Traveler's Dilemma: Paradoxes of Rationality in Game Theory*, American Economic Review, vol. 84, no. 2, pp. 391–395, 1994.

$$R_i(a_i, a_{-i}) = \begin{cases} a_i, & \text{если } a_i = a_{-i} \\ a_i + 2, & \text{если } a_i < a_{-i} \\ a_{-i} - 2 & \text{в ином случае} \end{cases}. \tag{F.11}$$

Большинство людей, как правило, указывают стоимость от 97 до 100 долларов. Однако, как это ни парадоксально, существует уникальное равновесие Нэша при оценке всего в 2 доллара.

F.13. Задача о хищнике и жертве в шестиугольном мире

Эта задача расширяет динамику гексамира, добавляя агентов, представляющих собой хищника и жертву. Хищник старается схватить жертву как можно быстрее, а жертва старается как можно дольше убегать от хищника. Исходное состояние гексамира показано на рис. F.16. В этой игре нет конечных состояний.

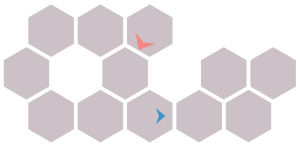


Рис. F.16. Исходное состояние в гексамире «хищник–жертва». Хищник обозначен красным цветом, а жертва – синим. Стрелки указывают возможные действия, предпринятые отдельными агентами из их начальных ячеек

В более общем случае существуют множество хищников $\mathcal{I}_{\text{pred}}$ и множество жертв $\mathcal{I}_{\text{prey}}$, где $\mathcal{I} = \mathcal{I}_{\text{pred}} \cup \mathcal{I}_{\text{prey}}$. Состояния содержат местоположения каждого агента $\mathcal{S} = \mathcal{S}^1 \times \cdots \times \mathcal{S}^{|\mathcal{I}|}$, где каждое $\mathcal{S}^i$ равно всем местоположениям на шестиугольных плитках. Пространство совместных действий равно $\mathcal{A} = \mathcal{A}^1 \times \cdots \times \mathcal{A}^{|\mathcal{I}|}$, где каждое $\mathcal{A}^i$ состоит из всех шести направлений движения.

Если хищник $i \in \mathcal{I}_{\text{pred}}$ и жертва $j \in \mathcal{I}_{\text{prey}}$ оказываются в одной и той же ячейке с $s_i = s_j$, то жертва погибает. Затем жертва j переносится в случайную ячейку, что олицетворяет рождение новой добычи в гексамире. В иных случаях переходы между состояниями независимы и соответствуют условиям исходного гексамира.

Один или несколько хищников могут поймать одну или несколько жертв, если все они оказались в одной ячейке. Если n хищников и m жертв оказались в одной ячейке, хищники получают вознаграждение в размере m/n. Например, если два хищника поймали одну жертву, каждый из них получает вознаграждение в размере 1/2. Если три хищника поймают пять жертв, каждый из них получит вознаграждение в размере 5/3. Движущиеся хищники получают единичный штраф за каждое перемещение без добычи. Добыча может двигаться без штрафа, но она получает штраф 100, если попадется хищникам.

F.14. Задача о плачущем ребенке с несколькими няньками

Задача о плачущем ребенке с несколькими няньками является многоагентной версией задачи о плачущем ребенке (для простоты пусть будет две няньки). Для каждой няньки $i \in \mathcal{I} = \{1, 2\}$ состояния, действия и наблюдения следующие:

$$\mathcal{S} = \{\text{голоден, сыт}\}; \tag{F.12}$$

$$\mathcal{A}^i = \{\text{кормить, петь, игнорировать}\}; \tag{F.13}$$

$$\mathcal{O}^i = \{\text{плачет, тихий}\}. \tag{F.14}$$

Динамика перехода аналогична исходной задаче о плачущем ребенке, за исключением того, что любая нянька может накормить ребенка:

$$T(\text{сыт} \mid \text{голоден}, (\text{кормить}, \bigstar)) = T(\text{сыт} \mid \text{голоден}, (\bigstar, \text{кормить})) = 100\ \%, \tag{F.15}$$

где ★ указывает на все возможные значения переменных. В противном случае, если действия не связаны с кормлением, ребенок переходит от сытого состояния к голодному, как и раньше:

$$T(\text{голоден} \mid \text{голоден}, (\bigstar, \bigstar)) = 100\ \%; \tag{F.16}$$

$$T(\text{сыт} \mid \text{сыт}, (\bigstar, \bigstar)) = 50\ \%. \tag{F.17}$$

Динамика наблюдения также аналогична версии с одним агентом, но модель гарантирует, что обе няньки осуществляют одинаковое наблюдение за ребенком, но не обязательно наблюдают за выбором действий другой няньки:

$$O((\text{плачет, плачет}) \mid (\text{петь}, \bigstar), \text{голоден}) = O((\text{плачет, плачет}) \mid (\bigstar, \text{петь}), \text{голоден}) = 90\ \%; \tag{F.18}$$

$$O((\text{тихий, тихий}) \mid (\text{петь}, \bigstar), \text{голоден}) = O((\text{тихий, тихий}) \mid (\bigstar, \text{петь}), \text{голоден}) = 10\ \%; \tag{F.19}$$

$$O((\text{плачет, плачет}) \mid (\text{петь}, \bigstar), \text{сытый}) = O((\text{плачет, плачет}) \mid (\bigstar, \text{петь}), \text{сытый}) = 0\ \%. \tag{F.20}$$

Если выбрано поведение «не петь», то наблюдения следующие:

$$O((\text{плачет, плачет}) \mid (\bigstar, \bigstar), \text{голоден}) = O((\text{плачет, плачет}) \mid (\bigstar, \bigstar), \text{голоден}) = 90\ \%; \tag{F.21}$$

$$O((\text{тихий, тихий}) \mid (\bigstar, \bigstar), \text{голоден}) = O((\text{тихий, тихий}) \mid (\bigstar, \bigstar), \text{голоден}) = 10\ \%; \tag{F.22}$$

$O((\text{плачет, плачет})\,|\,(★, ★), \text{сыт}) = O((\text{плачет, плачет})\,|\,(★, ★), \text{сыт}) = 0\ \%;$ (F.23)

$O((\text{тихий, тихий})\,|\,(★, ★), \text{сыт}) = O((\text{тихий, тихий})\,|\,(★, ★), \text{сыт}) = 100\ \%.$ (F.24)

Обе няньки хотят помочь ребенку, когда он голоден, что дает обоим одинаковый штраф –10.0. Однако первая нянька предпочитает кормление, а вторая – пение. За кормление первая нянька получает дополнительный штраф всего –2.5, а вторая получает дополнительный штраф –5.0. За пение первая нянька получает штраф –0.5, а вторая – всего –0.25.

F.15. Совместная задача о хищниках и жертвах в шестиугольном мире

Это вариант гексамира с хищниками и жертвами, в котором группа хищников *совместно* преследует одну движущуюся добычу. Хищники должны работать вместе, чтобы поймать жертву. Жертва случайным образом перемещается в соседнюю клетку, не занятую хищником.

Хищники также производят лишь зашумленные локальные наблюдения за окружающей средой. Каждый хищник i определяет, находится ли жертва в соседней ячейке $O^i = \{\text{добыча, пусто}\}$. Хищников наказывают штрафом –1 за каждое перемещение. Они получают награду 10, если оказываются в той же ячейке, что и жертва. В этот момент жертве случайным образом назначается новая ячейка, что означает рождение новой добычи, чтобы хищники снова начали охоту.

G Язык программирования Julia

Julia – это свободно распространяемый научный язык программирования с открытым исходным кодом[1]. Это относительно новый язык, создатели которого черпают вдохновение из таких языков, как Python, MATLAB и R. Мы выбрали его для использования в этой книге, потому что это язык достаточно высокого уровня[2], так что примеры кода получаются компактными и легко читаемыми и в то же время быстро работают. Примеры в книге совместимы с версией Julia 1.7. В этом приложении рассказано об основных компонентах языка, необходимых для понимания примеров кода. Изучение многих расширенных функций мы оставляем на усмотрение читателя. (На русском языке издан учебник М. Шеррингтона «Осваиваем язык Julia»: https://dmkpress.com/catalog/computer/programming/978-5-97060-370-3/. – *Прим. перев.*)

G.1. Типы

В Julia есть разнообразные базовые типы, которые могут представлять данные в виде значений истинности, чисел, строк, массивов, кортежей и словарей. Пользователи также могут определять свои типы. В этом разделе показано, как использовать некоторые из основных типов и определять новые типы.

G.1.1. Логические значения

Логический тип (Boolean) в Julia, обозначаемый как `Bool`, содержит лишь два значения – `true` и `false`. Мы можем присвоить эти значения переменным. Имена переменных могут быть любой строкой символов Unicode, с некоторыми ограничениями.

```
α = true
done = false
```

Имя переменной располагается слева от знака равенства; значение, которое должно быть присвоено переменной, находится справа.

---

1 Установочные пакеты и исходный код можно скачать на сайте https://julialang.org/.

2 В отличие от таких языков, как C++, Julia не заставляет программистов заботиться об управлении памятью и других аспектах выполнения кода на низком уровне, но при необходимости позволяет осуществлять низкоуровневое управление.

Мы можем выполнять присваивание непосредственно в консоли Julia. Консоль (иногда называемая REPL, от терминов read, eval, print, loop – чтение, вычисление, печать, цикл) вернет ответ на заданное выражение. Символ # означает, что оставшаяся часть строки является комментарием.

```
julia> x = true
true
julia> y = false; # точка с запятой подавляет вывод консоли
julia> typeof(x)
Bool
julia> x == y # проверка на равенство
false
```

Julia поддерживает стандартные логические операторы:

```
julia> !x # not
false
julia> x && y # and
false
julia> x || y # or
true
```

G.1.2. Числа

Julia поддерживает целые числа и числа с плавающей запятой, как показано ниже:

```
julia> typeof(42)
Int64
julia> typeof(42.0)
Float64
```

Здесь `Int64` обозначает 64-битное целое число, а `Float64` – 64-битное значение с плавающей запятой[3]. Мы можем выполнить стандартные математические операции:

```
julia> x = 4
4
julia> y = 2
2
julia> x + y
6
julia> x - y
2
julia> x * y
```

[3] На 32-разрядных машинах целочисленный литерал, такой как `42`, интерпретируется как `Int32`.

```
8
julia> x / y
2.0
julia> x ^ y # возведение в степень
16
julia> x % y # остаток от деления
0
julia> div(x, y) # усеченное деление возвращает целое число
2
```

Обратите внимание, что результатом операции `x / y` является `Float64`, даже если `x` и `y` представляют собой целые числа. Допускается выполнять математические операции одновременно с присваиванием. Например, `x += 1` является сокращенной записью для `x = x + 1`.

Julia поддерживает различные операции сравнения:

```
julia> 3 > 4
false
julia> 3 >= 4
false
```

julia> 3 ≥ 4 # Юникод тоже работает, используйте \ge[tab] в консоли

```
false
julia> 3 < 4
true
julia> 3 <= 4
true
```

julia> 3 ≤ 4 # Юникод тоже работает, используйте \le[tab] в консоли

```
true
julia> 3 == 4
false
julia> 3 < 4 < 5
true
```

G.1.3. Строки

Строка представляет собой массив символов. В этой книге строки используются редко, за исключением сообщений об определенных ошибках. Объект типа `String` может быть создан с использованием символов « (двойные кавычки). Например:

```
julia> x = "optimal"
"optimal"
julia> typeof(x)
String
```

G.1.4. Символы

Символ представляет собой идентификатор. Его можно записать с помощью оператора : (двоеточие) или получить из строки:

```
julia> :A
:A
julia> :Battery
:Battery
julia> Symbol("Failure")
:Failure
```

G.1.5. Векторы

Вектор – это одномерный массив, в котором хранится последовательность значений. Вектор можно задать путем перечисления значений в квадратных скобках, разделенных запятыми:

```
julia> x = [];                      # пустой вектор
julia> x = trues(3);                # логический вектор, содержащий три истины
julia> x = ones(3);                 # вектор из трех единиц
julia> x = zeros(3);                # вектор из трех нулей
julia> x = rand(3);                 # вектор из трех случайных чисел от 0 до 1
julia> x = [3, 1, 4];               # вектор целых чисел
julia> x = [3.1415, 1.618, 2.7182]; # вектор чисел с плавающей запятой
```

Для создания векторов можно использовать *генератор массивов* (array comprehension):

```
julia> [sin(x) for x in 1:5]
5-element Vector{Float64}:
0.8414709848078965
0.9092974268256817
0.1411200080598672
-0.7568024953079282
-0.9589242746631385
```

Можно проверить тип вектора:

```
julia> typeof([3, 1, 4])              # одномерный массив Int64
Vector{Int64} (alias for Array{Int64, 1})
julia> typeof([3.1415, 1.618, 2.7182]) # одномерный массив Float64s
Vector{Float64} (alias for Array{Float64, 1})
julia> Vector{Float64}                # псевдоним для одномерного массива
Vector{Float64} (alias for Array{Float64, 1})
```

Векторы индексируют с помощью квадратных скобок:

```
julia> x[1] # первый элемент имеет индекс 1
3.1415
```

```
julia> x[3] # третий элемент
2.7182
julia> x[end] # используйте end для ссылки на конец массива
2.7182
julia> x[end-1] # возвращает предпоследний элемент
1.618
```

Из массива можно извлечь диапазон элементов. Диапазоны указываются с помощью двоеточия:

```
julia> x = [1, 2, 5, 3, 1]
5-element Vector{Int64}:
 1
 2
 5
 3
 1
julia> x[1:3] # извлекаем первые три элемента
3-element Vector{Int64}:
 1
 2
 5
julia> x[1:2:end] # извлекаем все остальные элементы
3-element Vector{Int64}:
 1
 5
 1
julia> x[end:-1:1] # извлекаем все элементы в обратном порядке
5-element Vector{Int64}:
 1
 3
 5
 2
 1
```

Над массивами можно выполнять различные операции. Восклицательный знак на конце имен функций означает, что функция *мутирует* (т. е. изменяет) ввод:

```
julia> length(x)
5
julia> [x, x]             # конкатенация
2-element Vector{Vector{Int64}}:
 [1, 2, 5, 3, 1]
 [1, 2, 5, 3, 1]
julia> push!(x, -1)       # добавляем элемент в конец
6-element Vector{Int64}:
  1
  2
  5
  3
```

```
  1
 -1
julia> pop!(x)            # удаляем элемент в конце
 -1
julia> append!(x, [2, 3]) # добавляем [2, 3] в конец x
7-element Vector{Int64}:
 1
 2
 5
 3
 1
 2
 3
julia> sort!(x)           # сортируем элементы, изменяя один и тот же вектор
7-element Vector{Int64}:
 1
 1
 2
 2
 3
 3
 5
julia> sort(x);           # сортируем элементы как новый вектор
julia> x[1] = 2; print(x) # изменить первый элемент на 2
[2, 1, 2, 2, 3, 3, 5]
julia> x = [1, 2];
julia> y = [3, 4];
julia> x + y              # сложить векторы
2-element Vector{Int64}:
 4
 6
julia> 3x - [1, 2]        # перемножить как скаляры и вычесть
2-element Vector{Int64}:
 2
 4
julia> using LinearAlgebra
julia> dot(x, y)          # скалярное произведение с использованием LinearAlgebra
11
julia> x⋅y # скалярное произведение символов unicode character, используйте \
cdot[tab] в консоли
11
julia> prod(y)            # произведение всех элементов в y
12
```

Часто бывает необходимо применять к векторам различные функции поэлементно. Это разновидность *широковещательной операции* (broadcasting). Перед инфиксными операторами (например, +, * и ^) ставится точка, указывающая на поэлементное обращение. В таких функциях, как `sqrt` и `sin`, точка ставится после:

```
julia> x .* y # поэлементное умножение
2-element Vector{Int64}:
```

```
 3
 8
julia> x .^ 2 # поэлементное возведение в квадрат
2-element Vector{Int64}:
 1
 4
julia> sin.(x) # поэлементное применение sin
2-element Vector{Float64}:
 0.8414709848078965
 0.9092974268256817
julia> sqrt.(x) # поэлементное применение sqrt
2-element Vector{Float64}:
 1.0
 1.4142135623730951
```

G.1.6. Матрицы

Матрица представляет собой двумерный массив. Как и вектор, он строится с помощью квадратных скобок. Пробелы используют для разделения элементов в одной строке, а точки с запятой – для разделения строк. По аналогии с одномерными массивами можно индексировать матрицу и выводить подматрицы, используя диапазоны:

```
julia> X = [1 2 3; 4 5 6; 7 8 9; 10 11 12];
julia> typeof(X)              # двумерный массив Int64
Matrix{Int64} (alias for Array{Int64, 2})
julia> X[2]                   # второй элемент с упорядочением по столбцам
4
julia> X[3,2]                 # элемент в третьей строке и втором столбце
8
julia> X[1,:]                 # извлечь первую строку
3-element Vector{Int64}:
 1
 2
 3
julia> X[:,2]                 # извлечь второй столбец
4-element Vector{Int64}:
  2
  5
  8
 11
julia> X[:,1:2]               # извлечь первые два столбца
4×2 Matrix{Int64}:
  1  2
  4  5
  7  8
 10 11
julia> X[1:2,1:2]             # извлечь подматрицу 2×2 из верхнего левого угла x
2×2 Matrix{Int64}:
 1  2
```

```
 4  5
julia> Matrix{Float64}                    # псевдоним для двумерного массива
Matrix{Float64} (alias for Array{Float64, 2})
```

Можно также строить различные специальные матрицы и использовать генераторы массивов:

```
julia> Matrix(1.0I, 3, 3)          # единичная матрица 3×3
3×3 Matrix{Float64}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0
julia> Matrix(Diagonal([3, 2, 1])) # диагональная матрица 3×3 с 3, 2, 1
                                   # по диагонали
3×3 Matrix{Int64}:
 3  0  0
 0  2  0
 0  0  1
julia> zeros(3,2)                  # нулевая матрица 3×2
3×2 Matrix{Float64}:
 0.0  0.0
 0.0  0.0
 0.0  0.0
julia> rand(3,2)                   # случайная матрица 3×2
3×2 Matrix{Float64}:
 0.41794  0.881486
 0.14916  0.534639
 0.736357 0.850574
julia> [sin(x + y) for x in 1:3, y in 1:2] # генератор массива
3×2 Matrix{Float64}:
  0.909297   0.14112
  0.14112   -0.756802
 -0.756802  -0.958924
```

Julia поддерживает следующие операции с матрицами:

```
julia> X'              # комплексно-сопряженное транспонирование
3×4 adjoint(::Matrix{Int64}) with eltype Int64:
 1  4  7  10
 2  5  8  11
 3  6  9  12
julia> 3X .+ 2         # умножение на скаляр и сложение скаляра
4×3 Matrix{Int64}:
  5   8  11
 14  17  20
 23  26  29
 32  35  38
julia> X = [1 3; 3 1]; # создание обратимой матрицы
julia> inv(X)          # обращение
2×2 Matrix{Float64}:
 -0.125  0.375
```

```
  0.375 -0.125
julia> pinv(X)         # псевдообращение (требуется LinearAlgebra)
2×2 Matrix{Float64}:
 -0.125  0.375
  0.375 -0.125
julia> det(X)          # определитель матрицы (требуется LinearAlgebra)
-8.0
julia> [X X]           # горизонтальная конкатенация, то же, что hcat(X, X)
2×4 Matrix{Int64}:
 1  3  1  3
 3  1  3  1
julia> [X; X]          # вертикальная конкатенация, то же, что vcat(X, X)
4×2 Matrix{Int64}:
 1  3
 3  1
 1  3
 3  1
julia> sin.(X)         # поэлементное применение sin
2×2 Matrix{Float64}:
 0.841471  0.14112
julia> map(sin, X)     # поэлементное применение map
2×2 Matrix{Float64}:
 0.841471  0.14112
 0.14112   0.841471
julia> vec(X)          # преобразование массива в вектор
4-element Vector{Int64}:
 1
 3
 3
 1
```

G.1.7. Кортежи

Кортеж – это упорядоченный список значений, причем значения в одном кортеже могут быть различных типов. Кортежи строятся при помощи круглых скобок. Они похожи на векторы, но их нельзя изменять:

```
julia> x = ()    # пустой кортеж
()
julia> isempty(x)
true
julia> x = (1,)  # кортежи из одного элемента требуют запятой в конце
(1,)
julia> typeof(x)
Tuple{Int64}
julia> x = (1, 0, [1, 2], 2.5029, 4.6692) # третий элемент – вектор
(1, 0, [1, 2], 2.5029, 4.6692)
julia> typeof(x)
Tuple{Int64, Int64, Vector{Int64}, Float64, Float64}
julia> x[2]
```

```
0
julia> x[end]
4.6692
julia> x[4:end]
(2.5029, 4.6692)
julia> length(x)
5
julia> x = (1, 2)
(1, 2)
julia> a, b = x;
julia> a
1
julia> b
2
```

G.1.8. Именованные кортежи

Именованный кортеж похож на обычный, но у каждого элемента есть собственное имя:

```
julia> x = (a=1, b=-Inf)
(a = 1, b = -Inf)
julia> x isa NamedTuple
true
julia> x.a
1
julia> a, b = x;
julia> a
1
julia> (; :a=>10)
(a = 10,)
julia> (; :a=>10, :b=>11)
(a = 10, b = 11)
julia> merge(x, (d=3, e=10)) # слияние двух именованных кортежей
(a = 1, b = -Inf, d = 3, e = 10)
```

G.1.9. Словари

Словарь – это набор пар ключ–значение. Пары ключ–значение обозначаются оператором двойной стрелки =>. Мы можем индексировать словарь с помощью квадратных скобок, как и в случае с массивами и кортежами:

```
julia> x = Dict();             # пустой словарь
julia> x[3] = 4                # связать ключ 3 со значением 4
4
julia> x = Dict(3=>4, 5=>1)    # создать словарь с двумя парами ключ-значение
Dict{Int64, Int64} with 2 entries:
  5 => 1
  3 => 4
```

```
julia> x[5]                # вернуть значение, связанное с ключом 5
1
julia> haskey(x, 3)        # проверяем, есть ли в словаре ключ 3
true
julia> haskey(x, 4)        # проверяем, есть ли в словаре ключ 4
false
```

G.1.10. Составные типы

Составной тип представляет собой набор именованных полей. По умолчанию экземпляр составного типа является немутируемым (т. е. не может изменяться). Для определения составного типа сначала используют ключевое слово `struct`, а затем дают новому типу имя и перечисляют имена полей:

```
struct A
    a
    b
end
```

Добавление ключевого слова `mutable` позволяет изменять экземпляр:

```
mutable struct B
    a
    b
end
```

Составные типы строятся с помощью круглых скобок, между которыми передаются значения для каждого поля:

```
x = A(1.414, 1.732)
```

Оператор `::` (двойное двоеточие) применяется для указания типа поля:

```
struct A
    a::Int64
    b::Float64
end
```

Аннотации этих типов требуют, чтобы мы передавали значения `Int64` для первого поля и `Float64` для второго поля. Для компактности в этой книге не используются аннотации типов, но за это приходится расплачиваться снижением производительности. Аннотирование типов позволяет Julia повысить быстродействие во время выполнения, поскольку компилятор может оптимизировать базовый код для определенных типов.

G.1.11. Абстрактные типы

До сих пор мы обсуждали *явные типы*, то есть типы, которые мы можем конструировать. Однако конкретные типы являются лишь частью иерархии типов.

Существуют также *абстрактные типы*, которые являются супертипами явных типов и других абстрактных типов.

Мы можем исследовать иерархию типов для `Float64`, показанную на рис. G.1, используя функции `supertype` и `subtypes`:

```
julia> supertype(Float64)
AbstractFloat
julia> supertype(AbstractFloat)
Real
julia> supertype(Real)
Number
julia> supertype(Number)
Any
julia> supertype(Any)          # Any находится наверху иерархии
Any
julia> using InteractiveUtils  # требуется для использования подтипов в скриптах
julia> subtypes(AbstractFloat) # различные типы AbstractFloats
4-element Vector{Any}:
 BigFloat
 Float16
 Float32
 Float64
julia> subtypes(Float64)       # Float64 не имеет подтипов
Type[]
```

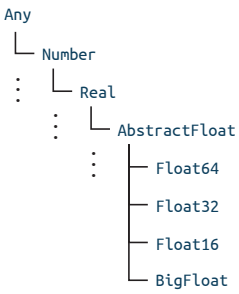


Рис. G.1. Иерархия типов для `Float64`

Можно определить собственные абстрактные типы:

```
abstract type D <: C end # D является абстрактным подтипом C
struct E <: D            # E – это составной тип, который является подтипом D
    a
end
```

G.1.12. Параметрические типы

Julia поддерживает *параметрические типы*, которые принимают параметры. Эти параметры задаются в фигурных скобках и разделяются запятыми. Вы уже видели параметрический тип в примере со словарем:

```
julia> x = Dict(3=>1.4, 1=>5.9)
Dict{Int64, Float64} with 2 entries:
  3 => 1.4
  1 => 5.9
```

В случае словарей первый параметр указывает тип ключа, а второй параметр указывает тип значения. В примере применяются ключи `Int64` и значения `Float64`, что делает словарь типа `Dict{Int64,Float64}`. Julia автоматически выводит эти типы на основе входных данных, но мы могли указать их явно:

```
julia> x = Dict{Int64,Float64}(3=>1.4, 1=>5.9)
```

Хотя можно определить собственные параметрические типы, в примерах данной книги в этом нет необходимости.

G.2. Функции

Функция сопоставляет аргументы, заданные в виде кортежа, с возвращаемым результатом.

G.2.1. Именованные функции

Один из способов определить *именованную функцию* – использовать ключевое слово `function`, за которым следует имя функции и кортеж имен аргументов:

```
function f(x, y)
    return x + y
end
```

Допускается компактное определение функции в виде присваивания:

```
julia> f(x, y) = x + y;
julia> f(3, 0.1415)
3.1415
```

G.2.2. Анонимные функции

У *анонимной функции* отсутствует имя, хотя ее можно присвоить именованной переменной. Один из способов определить анонимную функцию – использовать оператор стрелки:

```
julia> h = x -> x^2 + 1          # присвоить анонимную функцию с входом x
                                 # переменной h
#1 (generic function with 1 method)
julia> h(3)
10
julia> g(f, a, b) = [f(a), f(b)]; # применить функцию f к a и b и вернуть массив
```

```
julia> g(h, 5, 10)
2-element Vector{Int64}:
  26
  101
julia> g(x->sin(x)+1, 10, 20)
2-element Vector{Float64}:
 0.4559788891106302
```

G.2.3. Вызываемые объекты

Мы можем определить тип и связать с ним функции, создав таким образом *вызываемый* объект данного типа:

```
julia> (x::A)() = x.a + x.b    # добавление функции без аргументов к типу A,
                               # определенному ранее
julia> (x::A)(y) = y*x.a + x.b # добавление функции с одним аргументом
julia> x = A(22, 8);
julia> x()
30
julia> x(2)
52
```

G.2.4. Необязательные аргументы

Аргументу можно присвоить значение по умолчанию. Такой аргумент не обязательно передавать при вызове функции:

```
julia> f(x=10) = x^2;
julia> f()
100
julia> f(3)
9
julia> f(x, y, z=1) = x*y + z;
julia> f(1, 2, 3)
5
julia> f(1, 2)
3
```

G.2.5. Именованные аргументы

Функции могут использовать *именованные аргументы*, которые представляют собой аргументы, именуемые при вызове функции. Такие аргументы следуют после всех позиционных аргументов. Перед любыми именованными аргументами ставится точка с запятой, отделяющая их от других аргументов:

```
julia> f(; x = 0) = x + 1;
julia> f()
1
```

```
julia> f(x = 10)
11
julia> f(x, y = 10; z = 2) = (x + y)*z;
julia> f(1)
22
julia> f(2, z = 3)
36
julia> f(2, 3)
10
julia> f(2, 3, z = 1)
5
```

G.2.6. Диспетчеризация

Типы аргументов, передаваемых функции, можно указать с помощью оператора `::` (двойное двоеточие). Если существуют несколько методов одной и той же функции, Julia выполнит наиболее подходящий метод. Механизм выбора метода для выполнения называется *диспетчеризацией* (dispatch):

```
julia> f(x::Int64) = x + 10;
julia> f(x::Float64) = x + 3.1415;
julia> f(1)
11
julia> f(1.0)
4.141500000000001
julia> f(1.3)
4.4415000000000004
```

Будет использоваться метод с сигнатурой типа, который лучше всего соответствует типам переданных аргументов:

```
julia> f(x) = 5;
julia> f(x::Float64) = 3.1415;
julia> f([3, 2, 1])
5
julia> f(0.00787499699)
3.1415
```

G.2.7. Разделение вектора на аргументы

Часто бывает полезно разделить элементы вектора или кортежа на аргументы функции с помощью оператора `...` (троеточие):

```
julia> f(x,y,z) = x + y - z;
julia> a = [3, 1, 2];
julia> f(a...)
2
julia> b = (2, 2, 0);
julia> f(b...)
```

```
4
julia> c = ([0,0],[1,1]);
julia> f([2,2], c...)
2-element Vector{Int64}:
1
1
```

G.3. Управление ходом выполнения

Julia позволяет управлять ходом выполнения программы при помощи операторов условий и циклов. В этом разделе представлены некоторые элементы синтаксиса, используемые в книге.

G.3.1. Условное выполнение

При условном выполнении проверяется значение логического выражения, а затем выполняется соответствующий блок кода. Чаще всего это делается с помощью оператора `if`:

```
if x < y
    # выполнить этот блок, если x < y
elseif x > y
    # выполнить этот блок, если x > y
else
    # выполнить этот блок, если x == y
end
```

Допускается использование *тернарного оператора*, состоящего из вопросительного знака и двоеточия. Он проверяет логическое выражение перед вопросительным знаком. Если выражение оценивается как истинное, то оператор возвращает значение перед двоеточием; в противном случае возвращается значение после двоеточия:

```
julia> f(x) = x > 0 ? x : 0;
julia> f(-10)
0
julia> f(10)
10
```

G.3.2. Циклы

Цикл позволяет многократно повторять выполнение внутреннего блока операций. Одной из разновидностей цикла является `while`, который многократно выполняет блок операций до тех пор, пока не будет соблюдено условие, указанное после ключевого слова `while`. В следующем примере суммируются значения массива `x`:

```
X = [1, 2, 3, 4, 6, 8, 11, 13, 16, 18]
s = 0
while !isempty(X)
    s += pop!(X)
end
```

Еще одна базовая разновидность цикла обозначается ключевым словом `for`. Следующий пример также суммирует значения в массиве `x`, но не изменяет его:

```
X = [1, 2, 3, 4, 6, 8, 11, 13, 16, 18]
s = 0
for y in X
    s += y
end
```

Ключевое слово `in` можно заменить на `=` или `∈`. Следующий блок кода эквивалентен предыдущему:

```
X = [1, 2, 3, 4, 6, 8, 11, 13, 16, 18]
s = 0
for i = 1:length(X)
    s += X[i]
end
```

G.3.3. Итераторы

Мы можем выполнять итерацию по коллекции в таких контекстах, как циклы `for` и генераторы массивов. Чтобы продемонстрировать в действии различные итераторы, мы будем использовать функцию `collect`, которая возвращает массив всех элементов, сгенерированных итератором:

```
julia> X = ["feed", "sing", "ignore"];
julia> collect(enumerate(X))     # возвращает счетчик и элемент
3-element Vector{Tuple{Int64, String}}:
 (1, "feed")
 (2, "sing")
 (3, "ignore")
julia> collect(eachindex(X))     # эквивалентно 1:length(X)
3-element Vector{Int64}:
 1
 2
 3
julia> Y = [-5, -0.5, 0];
julia> collect(zip(X, Y))        # проход по нескольким итераторам одновременно
3-element Vector{Tuple{String, Float64}}:
 ("feed", -5.0)
 ("sing", -0.5)
 ("ignore", 0.0)
julia> import IterTools: subsets
```

```
julia> collect(subsets(X))       # итерация по всем подмножествам
8-element Vector{Vector{String}}:
 []
 ["feed"]
 ["sing"]
 ["feed", "sing"]
 ["ignore"]
 ["feed", "ignore"]
 ["sing", "ignore"]
 ["feed", "sing", "ignore"]
julia> collect(eachindex(X))     # итерация по индексам в коллекции
3-element Vector{Int64}:
 1
 2
 3
julia> Z = [1 2; 3 4; 5 6];
julia> import Base.Iterators: product
julia> collect(product(X,Y))     # перебираем декартово произведение нескольких
                                 # итераторов
3×3 Matrix{Tuple{String, Float64}}:
 ("feed", -5.0)   ("feed", -0.5)   ("feed", 0.0)
 ("sing", -5.0)   ("sing", -0.5)   ("sing", 0.0)
 ("ignore", -5.0) ("ignore", -0.5) ("ignore", 0.0)
```

G.4. Пакеты

Пакет – это набор кода Julia и, возможно, других внешних библиотек, которые можно импортировать для получения доступа к дополнительным функциям. В этом разделе кратко рассмотрены некоторые ключевые пакеты, использованные в примерах кода для этой книги. Чтобы добавить зарегистрированный пакет, такой как `Distributions.jl`, достаточно выполнить следующую команду:

```
using Pkg
Pkg.add("Distributions")
```

Для обновления пакетов используйте команду

```
Pkg.update()
```

Чтобы использовать пакет, необходимо указать ключевое слово `using` и имя пакета следующим образом:

```
using Distributions
```

G.4.1. Пакет Graphs.jl

Пакет `Graphs.jl` (версия 1.4) предназначен для представления графов и выполнения над ними операций:

```
julia> using Graphs
julia> G = SimpleDiGraph(3); # создаем ориентированный граф с тремя узлами
julia> add_edge!(G, 1, 3);   # добавляем ребро от узла 1 до узла 3
julia> add_edge!(G, 1, 2);   # добавляем ребро от узла 1 до узла 2
julia> rem_edge!(G, 1, 3);   # удаляем ребро от узла 1 до 3
julia> add_edge!(G, 2, 3);   # добавляем ребро от узла 2 до 3
julia> typeof(G)
Graphs.SimpleGraphs.SimpleDiGraph{Int64}
julia> nv(G)                 # количество узлов (также называемых вершинами)
3
julia> outneighbors(G, 1)    # список исходящих соседей для узла 1
1-element Vector{Int64}:
 2
julia> inneighbors(G, 1)     # список входящих соседей для узла 1
Int64[]
```

G.4.2. Пакет Distributions.jl

Пакет `Distributions.jl` (версия 0.24) предназначен для представления вероятностных распределений, подгонки их под данные и выборки из них:

```
julia> using Distributions
julia> dist = Categorical([0.3, 0.5, 0.2]) # создать категориальное распределение
Distributions.Categorical{Float64, Vector{Float64}}(support=Base.OneTo(3),
p=[0.3, 0.5, 0.2])
julia> data = rand(dist)                   # создать выборку
2
julia> data = rand(dist, 2)                # создать две выборки
2-element Vector{Int64}:
 2
 3
julia> μ, σ = 5.0, 2.5;                   # определить параметры нормального
                                           # распределения
julia> dist = Normal(μ, σ)                 # создать нормальное распределение
Distributions.Normal{Float64}(μ=5.0, σ=2.5)
julia> data = rand(dist, 3)                # создать три выборки
3-element Vector{Float64}:
 10.860475998911657
  1.519358465527894
  3.0194180096515186
julia> data = rand(dist, 1000);            # создать много выборок
julia> Distributions.fit(Normal, data)     # подогнать нормальное распределение
                                           # по выборкам
Distributions.Normal{Float64}(μ=5.085987626631449, σ=2.4766229761489367)
julia> μ = [1.0, 2.0];
julia> Σ = [1.0 0.5; 0.5 2.0];
julia> dist = MvNormal(μ, Σ)               # создать многомерное нормальное
                                           # распределение
FullNormal(
```

```
dim: 2
μ: [1.0, 2.0]
Σ: [1.0 0.5; 0.5 2.0]
)
julia> rand(dist, 3)                    # создать три выборки
2×3 Matrix{Float64}:
 0.834945 -0.527494 -0.098257
 1.25277  -0.246228  0.423922
julia> dist = Dirichlet(ones(3))        # создать распределение Дирихле Dir(1,1,1)
Distributions.Dirichlet{Float64, Vector{Float64}, Float64}(alpha=[1.0, 1.0, 1.0])
julia> rand(dist)                       # выборка из распределения
3-element Vector{Float64}:
 0.19658106436589923
 0.6128478073834874
 0.1905711282506134
```

G.4.3. Пакет JuMP.jl

Пакет `JuMP.jl` (версия 0.21) предназначен для определения задач оптимизации, которые затем можно решить с помощью различных решателей, таких как `GLPK.jl` и `Ipopt.jl`:

```
julia> using JuMP
julia> using GLPK
julia> model = Model(GLPK.Optimizer) # создать модель и использовать GLPK
                                     # в качестве решателя
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: GLPK
julia> @variable(model, x[1:3])      # определить переменные x[1], x[2] и x[3]
3-element Vector{JuMP.VariableRef}:
 x[1]
 x[2]
 x[3]
julia> @objective(model, Max, sum(x) - x[2]) # определить цель максимизации
x[1] + 0 x[2] + x[3]
julia> @constraint(model, x[1] + x[2] ≤ 3)   # добавить ограничение
x[1] + x[2] <= 3.0
julia> @constraint(model, x[2] + x[3] ≤ 2)   # добавить еще одно ограничение
x[2] + x[3] <= 2.0
julia> @constraint(model, x[2] ≥ 0)          # добавить еще одно ограничение
x[2] >= 0.0
julia> optimize!(model)                      # решить
julia> value.(x)                             # извлечь оптимальные значения
                                             # для элементов в x
3-element Vector{Float64}:
 3.0
```

```
 0.0
 2.0
```

G.5. Вспомогательные функции

В Julia есть несколько функций, которые позволяют получать более компактный код (мы их использовали в этой книге). Следующие функции полезны при работе со словарями и именованными кортежами:

```
Base.Dict{Symbol,V}(a::NamedTuple) where V =
    Dict{Symbol,V}(n=>v for (n,v) in zip(keys(a), values(a)))
Base.convert(::Type{Dict{Symbol,V}}, a::NamedTuple) where V =
    Dict{Symbol,V}(a)
Base.isequal(a::Dict{Symbol,<:Any}, nt::NamedTuple) =
    length(a) == length(nt) &&
    all(a[n] == v for (n,v) in zip(keys(nt), values(nt)))

julia> a = Dict{Symbol,Integer}((a=1, b=2, c=3))
Dict{Symbol, Integer} with 3 entries:
  :a => 1
  :b => 2
  :c => 3
julia> isequal(a, (a=1, b=2, c=3))
true
julia> isequal(a, (a=1, c=3, b=2))
true
julia> Dict{Dict{Symbol,Integer},Float64}((a=1, b=1)=>0.2, (a=1, b=2)=>0.8)
Dict{Dict{Symbol, Integer}, Float64} with 2 entries:
  Dict(:a=>1, :b=>1) => 0.2
  Dict(:a=>1, :b=>2) => 0.8
```

Определим `SetCategorical` для представления распределений в виде дискретных наборов:

```
struct SetCategorical{S}
    elements::Vector{S}      # набор элементов (могут повторяться)
    distr::Categorical       # категориальное распределение по элементам набора

    function SetCategorical(elements::AbstractVector{S}) where S
        weights = ones(length(elements))
        return new{S}(elements, Categorical(normalize(weights, 1)))
    end

    function SetCategorical(
            elements::AbstractVector{S},
            weights::AbstractVector{Float64}
        ) where S

        ℓ1 = norm(weights,1)
        if ℓ1 < 1e-6 || isinf(ℓ1)
```

```
            return SetCategorical(elements)
        end
        distr = Categorical(normalize(weights, 1))
        return new{S}(elements, distr)
    end
end

Distributions.rand(D::SetCategorical) = D.elements[rand(D.distr)]
Distributions.rand(D::SetCategorical, n::Int) = D.elements[rand(D.distr, n)]
function Distributions.pdf(D::SetCategorical, x)
    sum(e == x ? w : 0.0 for (e,w) in zip(D.elements, D.distr.p))
end
julia> D = SetCategorical(["up", "down", "left", "right"],[0.4, 0.2, 0.3, 0.1]);
julia> rand(D)
"up"
julia> rand(D, 5)
5-element Vector{String}:
 "left"
 "up"
 "down"
 "up"
 "left"
julia> pdf(D, "up")
0.3999999999999999
```

Предметный указатель

Г

Д

Ж

З

И

К

Л

М

H

O

П

Р

С

Т

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38, оф. 10;
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **http://www.galaktika-dmk.com/**.

Микель Кохендерфер, Тим Уилер, Кайл Рэй

Алгоритмы принятия решений

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Зам. главного редактора *Сенченкова Е. А.*
Перевод *Яценков В. С.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 55,58. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**