Древняя стратегическая игра го представляет собой отличный пример для демонстрации возможностей искусственного интеллекта. В 2016 году система, основанная на принципах глубокого обучения, потрясла мир го, победив одного из чемпионов. Вскоре после этого модернизированный алгоритм AlphaGo Zero сокрушил оригинальную версию бота благодаря использованию при освоении игры методов глубокого обучения с подкреплением. Теперь и вы можете освоить те же самые методы глубокого обучения, создав собственный бот для игры в го!

В данной книге вы познакомитесь с методами глубокого обучения и научитесь создавать гоботов. По мере чтения вы будете применять все более сложные методы и стратегии обучения, используя библиотеку глубокого обучения Keras, написанную на языке Python. Вы будете с удовольствием наблюдать за тем, как ваш бот осваивает игру го, и по ходу дела узнаете о вариантах применения полученных навыков глубокого обучения к широкому кругу других задач!

С этой книгой вы научитесь:

- создавать и обучать самосовершенствующиеся игровые ИИ;
- улучшать системы классического игрового ИИ с помощью глубокого обучения;
- использовать для реализации глубокого обучения нейронные сети.

Макс Памперла (Max Pumperla) и Кевин Фергюсон (Kevin Ferguson) *являются специалистами по глубокому обучению, имеют опыт работы в сфере распределенных систем и анализа данных. Вместе Макс и Кевин создали бота с открытым исходным кодом BetaGo.*

Все, что вам потребуется, – это базовое знание языка Python и математики на уровне средней школы. Наличие опыта глубокого обучения необязательно.

Интернет-магазин: www.dmkpress.com

Оптовая продажа: КТК "Галактика"
books@alians-kniga.ru

www.дмк.рф

ISBN 978-5-97060-769-5

Глубокое обучение и игра в го

«Легко читаемое и увлекательное введение в тему искусственного интеллекта и машинного обучения!»
— из предисловия Тора Грепеля, DeepMind

«Вдохновляющая демонстрация возможностей машинного обучения на примере игры в го. Очень рекомендую!»
— Бурк Хуфнагель, Daugherty Business Solutions

«Блестящее знакомство с одной из самых захватывающих технологий нашего времени».
— Хельмут Хаушильд, HSEC

«Замечательный, легко читаемый код на языке Python».
—Доминго Салазар, AstraZeneca

Глубокое обучение и игра в го

Глубокое обучение и игра в го

Макс Памперла
Кевин Фергюсон

MANNING

ДМК ИЗДАТЕЛЬСТВО

Макс Памперла и Кевин Фергюсон

Глубокое обучение и игра в го

Max Pumperla and Kevin Ferguson

Deep Learning and the Game of Go

MANNING
Shelter Island

Макс Памперла и Кевин Фергюсон

Глубокое обучение и игра в го

Москва, 2020

УДК 004.891
ББК 32.972.13
П15

Памперла М., Фергюсон К.
П15 Глубокое обучение и игра в го / пер. с анг. М. А. Райтмана. – М.: ДМК Пресс, 2020. – 372 с.: ил.

ISBN 978-5-97060-769-5

Древняя стратегическая игра го представляет собой отличный пример для демонстрации возможностей искусственного интеллекта. В 2016 году система, основанная на принципах глубокого обучения, потрясла мир го, победив одного из чемпионов.

В данной книге вы познакомитесь с методами глубокого обучения и научитесь создавать го-ботов. По мере чтения вы будете применять все более сложные методы и стратегии, используя библиотеку глубокого обучения Keras, написанную на языке Python. Вы будете с удовольствием наблюдать за тем, как ваш бот осваивает игру го, и узнаете о вариантах применения полученных навыков ко множеству других задач!

Издание предназначено широкому кругу читателей, знакомых с языком Python и желающих на практике познакомиться с методами глубокого обучения.

В России можно отыскать десятки тысяч ценителей игры го. Во многих городах страны функционируют го-клубы. Под эгидой Российской федерации го ежегодно проводятся сотни турниров. Их полный список можно найти на сайте **https://gofederation.ru/**.

УДК 004.891
ББК 32.972.13

ISBN 978-1-617-29532-4 (анг.)
ISBN 978-5-97060-769-5 (рус.)

Содержание

Предисловие

Для нас, членов команды AlphaGo, история этого алгоритма стала главным приключением всей жизни. Все началось, как это часто бывает, с небольшого шага – обучения простой сверточной нейронной сети на записях партий в го, сыгранных сильными игроками-людьми. Это привело к кардинальным прорывам в области машинного обучения, а также подарило нам несколько незабываемых событий, включая матчи против таких грозных профессиональных игроков, как Фань Хуэй, Ли Седоль и Ки Джи. Мы гордимся тем, что эти матчи не только повлияли на манеру игры в го по всему миру, но и привлекли внимание множества людей к теме искусственного интеллекта.

Но почему, спросите вы, нас должны интересовать игры? Подобно тому как дети используют игры для изучения тех или иных аспектов реального мира, исследователи в области машинного обучения используют их для подготовки программ-агентов. В этом смысле проект AlphaGo является частью стратегии компании DeepMind по использованию игр в качестве симулированных микрокосмов реальности. Это помогает нам развивать область искусственного интеллекта и тренировать обучающихся агентов, чтобы в будущем создавать интеллектуальные системы, способные решать самые сложные мировые проблемы.

Работа алгоритма AlphaGo напоминает два режима мышления, которые нобелевский лауреат Даниэль Канеман описал в своей книге «Думай медленно, решай быстро», посвященной вопросам человеческого познания. В случае с AlphaGo аналогом медленного режима мышления является алгоритм планирования, называемый *поиском по дереву методом Монте-Карло*, который просчитывает последовательность игровых состояний, начиная с заданной позиции, путем расширения дерева игры, включающего возможные будущие ходы и действия противника. Однако при наличии около 10^{170} (1 со 170 нулями) возможных игровых позиций просчет всех последовательностей оказывается невозможным. Чтобы обойти эту проблему и сократить пространство поиска, мы объединили алгоритм поиска по дереву методом Монте-Карло с компонентом *глубокого обучения*, состоящего из двух нейронных сетей, способных оценивать вероятность победы каждого из игроков и выбирать наиболее перспективные ходы.

В более поздней версии алгоритма, AlphaZero, используются принципы *обучения с подкреплением*, что позволяет программе играть против самой себя, не полагаясь на записи партий, сыгранных человеком. Этот алгоритм с нуля обучился игре в го (а также в шахматы и сёги), часто обнаруживая (и в дальнейшем отбрасывая) многие из стратегий, разработанных за сотни лет игроками-людьми, и создав множество собственных уникальных стратегий.

Авторы этой книги Макс Памперла и Кевин Фергюсон станут вашими проводниками в увлекательном путешествии от AlphaGo до более поздних версий этого алгоритма. В ходе чтения данного руководства вы не только реализуете движок для игры в го в стиле AlphaGo, но и получите отличное практическое понимание некоторых из основополагающих строительных блоков современных алгоритмов ИИ: поиска по дереву методом Монте-Карло, глубокого обучения и обуче-

ния с подкреплением. Авторы искусно объединили эти темы, используя игру го в качестве захватывающего и доступного для понимания примера. В дополнение к этому вы изучите основы одной из самых красивых и сложных игр, когда-либо изобретенных человечеством.

Кроме того, эта книга призывает вас с самого начала приступить к созданию работающего го-бота, который будет постепенно эволюционировать от алгоритма, выбирающего ходы совершенно случайным образом, до сложного самообучающегося ИИ для игры в го. Помимо исчерпывающих объяснений основополагающих концепций, авторы предоставили исполняемый код на языке Python. Кроме того, они не пренебрегли такими темами, как форматы данных, развертывание бота и облачные вычисления, необходимыми для обеспечения работоспособности программы для игры в го.

Таким образом, книга «Глубокое обучение и игра в го» представляет собой легко читаемое и увлекательное введение в тему искусственного интеллекта и машинного обучения. Объединяя в себе некоторые из самых захватывающих этапов развития области искусственного интеллекта, она превращается в интереснейший вводный курс по данному предмету. Любой читатель, прошедший этот путь от начала до конца, приобретет необходимые знания для понимания и создания современных систем ИИ, применяемых в ситуациях, требующих сочетания «быстрого» сопоставления образов с «медленным» планированием, которые являются аналогом двух режимов мышления, необходимых для осуществления базового процесса познания.

– *Тор Грепель*, научный сотрудник компании DeepMind,
от имени команды AlphaGo компании DeepMind

Введение

Когда в начале 2016 года о программе AlphaGo заговорили в новостях, мы были чрезвычайно взволнованы этим новаторским достижением в сфере компьютерных игр. В то время считалось, что до создания искусственного интеллекта для игры в го, способного играть на человеческом уровне, оставалось не менее 10 лет. Мы тщательно следили за играми и были готовы пожертвовать сном ради того, чтобы посмотреть трансляции матчей в прямом эфире. Но мы были в хорошей компании – миллионы людей по всему миру были зачарованы играми AlphaGo против Фань Хуэя, Ли Седоля, Ки Цжи и других профессиональных игроков.

Вскоре после появления этого алгоритма мы приступили к работе над небольшой библиотекой с открытым исходным кодом, которую назвали BetaGo (**github.com/maxpumperla/betago**), чтобы посмотреть, сможем ли мы самостоятельно реализовать некоторые базовые механизмы, лежащие в основе алгоритма AlphaGo. Идея BetaGo состояла в том, чтобы продемонстрировать интересующимся разработчикам некоторые из методов, использующихся в этой программе. Несмотря на то что наших ресурсов (времени, вычислительной мощности или интеллекта) было недостаточно, чтобы конкурировать с невероятным достижением компании DeepMind, мы получили огромное удовольствие в процессе создания собственного го-бота.

С тех пор нам много раз предоставлялась возможность рассказать об ИИ для игры в го. Поскольку мы являемся не только поклонниками этой игры, но и практиками машинного обучения, мы иногда забывали о том, как мало широкая публика могла вынести из событий, за которыми мы так пристально следили. Ирония заключалась в том, что наблюдающие за матчами миллионы людей, по-видимому, делились на две группы:

- те, кто понимает и любит игру го, но мало знает о машинном обучении;
- те, кто понимает и ценит машинное обучение, но практически незнаком с правилами игры в го.

Для далекого от этих тем человека обе сферы могут казаться одинаково туманными, сложными и трудными для освоения. Несмотря на то что в последние годы все большее количество разработчиков программного обеспечения задействует методы машинного обучения и, в частности, *глубокого обучения*, игра в го остается в значительной степени неизвестной многим жителям стран Запада. Мы считаем это весьма прискорбным и искренне надеемся, что данная книга позволит сблизить две вышеупомянутые группы людей.

Мы убеждены в том, что использованию принципов, лежащих в основе алгоритма AlphaGo, можно на практике обучить широкую аудиторию разработчиков программного обеспечения. Наслаждение игрой го и ее понимание приходит в процессе игры и экспериментов. То же самое можно сказать о машинном обучении и любой другой дисциплине.

Если в ходе изучения этой книги вы проникнетесь энтузиазмом в отношении игры го или машинного обучения (надеемся, и того, и другого!), мы будем считать, что выполнили свою задачу. Если, помимо этого, вы научитесь создавать и развертывать боты для игры в го, а также проводить собственные эксперименты, вам станет доступно множество других интересных ИИ-приложений. Наслаждайтесь путешествием!

Благодарности

Мы хотели бы поблагодарить всех сотрудников издательства Manning, сделавших публикацию этой книги возможной. В частности, мы благодарим наших неутомимых редакторов: Марину Майклз за то, что она помогла нам преодолеть первые 80 % пути, и Дженни Стаут за помощь в преодолении вторых 80 %. Выражаем благодарность нашему техническому редактору Чарльзу Федуке и техническому корректору Тане Вилке за проверку кода.

Благодарим всех рецензентов, предоставивших ценные отзывы: Александра Ерофеева, Алессандро Пузиелли, Алекса Орланди, Бурка Хуфнагеля, Крейга С. Коннелла, Даниэля Береца, Дениса Крайса, Доминго Салазара, Хельмута Хаушильда, Джеймса А. Худа, Джасбу Симпсона, Джин Лазароу, Мартина Мёллера, Скарбиникса Педерсена, Матиаса Поллигкайта, Ната Луенгнарюмитчая, Пьерлуиджи Рити, Сэма Де Костера, Шона Линдсея, Тайлера Коваллиса и Урсина Стосса.

Также спасибо всем, кто экспериментировал или участвовал в разработке нашего проекта BetaGo, особенно Эллиоту Герчаку и Кристоферу Мэлону.

Наконец, благодарим всех, кто когда-либо пытался научить компьютер играть в го и поделился результатами своих исследований.

Я хотел бы поблагодарить Карли за ее терпение и поддержку, а также папу и Джиллиан за то, что научили меня писать.

– Кевин Фергюсон

Особая благодарность Кевину за помощь в разъяснении материала, Андреасу за множество плодотворных дискуссий и Энн за ее постоянную поддержку.

– Макс Памперла

Об авторах

Макс Памперла является специалистом по работе с данными и инженером, занимающимся глубоким обучением в компании-разработчике ИИ-систем Skymind. Также он является сооснователем платформы глубокого обучения **aetros.com**.

Кевин Фергюсон на протяжении 18 лет работал в области создания распределенных систем и анализа данных. Он является специалистом по анализу данных в компании Honor и имеет опыт работы в таких компаниях, как Google и Meebo. Вместе Макс и Кевин разработали BetaGo, один из очень немногих го-ботов с открытым исходным кодом, созданных на языке Python.

Об иллюстрации на обложке

На обложке книги «Глубокое обучение и игра в го» изображен император Монтоку, правивший Японией с 850 по 858 год. Этот портрет был написан акварелью на шелке неизвестным художником. В 2006 году его репродукция была включена в раздел «Императоры и императрицы прошлого» японского исторического журнала *Bessatsu Rekishi Dokuhon*.

Подобные изображения напоминают нам об уникальности и индивидуальности древних городов и регионов мира. В то время по одежде можно было однозначно сказать, к какому из двух городов, разделенных несколькими десятками километров, принадлежит тот или иной человек.

С тех пор дресс-код изменился, и от былой самобытности разных уголков мира не осталось и следа. Сейчас жителей разных континентов бывает трудно отличить друг от друга. Возможно, мы променяли культурное и визуальное разнообразие на более разнообразную и интересную личную и интеллектуальную жизнь, или на более разнообразную жизнь в плане технологий. Мы, сотрудники издательства Manning, стремимся дополнить изобретательность, новаторство и увлеченность, пробуждаемые книгами по компьютерной тематике, обложками, отражающими богатый и разнообразный местный колорит прошлого.

Об этой книге

Книга «Глубокое обучение и игра в го» призвана познакомить читателя с современными концепциями машинного обучения на практическом примере создания искусственного интеллекта для игры в го. К концу главы 3 у вас будет готовая, хоть и очень примитивная программа, играющая в эту игру. В каждой следующей главе будет представлен новый метод улучшения ИИ вашего бота. В ходе экспериментов вы узнаете о преимуществах и недостатках каждого из этих методов. А в последних главах мы покажем, как алгоритмы AlphaGo и AlphaGo Zero позволяют интегрировать все представленные в книге методы в невероятно мощный ИИ.

Для кого предназначена эта книга

Эта книга рассчитана на разработчиков программного обеспечения, желающих приступить к экспериментам в области машинного обучения и предпочитающих практический подход математическому. Мы предполагаем, что вы обладаете практическим опытом программирования на языке Python, однако описанные алгоритмы можно реализовать на любом современном языке. Мы не предполагаем, что вы знакомы с игрой го. Если вы предпочитаете шахматы или какую-либо другую игру подобного рода, то сможете адаптировать к ней большинство описанных методов. Если же вы сами являетесь игроком в го, то получите массу удовольствия, наблюдая за прогрессом вашего бота!

Структура книги

Книга состоит из трех частей, включающих 14 глав и 5 приложений.

Часть I «Основы» знакомит читателя с основными концепциями, подробно описанными в остальных частях книги.

- Глава 1 «На пути к глубокому обучению» предоставляет беглый высокоуровневый обзор таких дисциплин, как искусственный интеллект, машинное обучение и глубокое обучение. В этой главе мы объясняем, как они связаны между собой и что вы можете и не можете сделать с помощью методов, применяемых в этих областях.
- Глава 2 «Игра го как проблема машинного обучения» знакомит читателя с правилами игры го и объясняет, чему мы намерены обучить компьютер.
- В главе 3 «Реализация первого бота для игры в го» мы реализуем доску для игры в го, раскладываем камни и играем в игры с помощью средств языка Python. В конце этой главы вы сможете создать простейший ИИ для игры в го.

Часть II «Машинное обучение и игровой ИИ» посвящена техническим и методологическим основам создания мощного ИИ. В частности, там описаны три метода, которые очень эффективно используются в алгоритме AlphaGo: *поиск по дереву*, *нейронные сети* и *обучение с подкреплением*.

Поиск по дереву

- Глава 4 «Игры и поиск по дереву» содержит обзор алгоритмов, которые отвечают за поиск и оценку последовательностей игрового процесса. Мы начнем с простого поиска путем минимаксного перебора, а затем перейдем к таким сложным алгоритмам, как альфа-бета-отсечение и метод Монте-Карло.

Нейронные сети

- Глава 5 «Знакомство с нейронными сетями» представляет собой практическое введение в тему искусственных нейронных сетей. Вы научитесь предсказывать рукописные цифры, создав нейронную сеть с нуля средствами Python.
- Глава 6 «Проектирование нейронной сети для данных го» объясняет сходство данных го с данными изображения, а также представляет сверточные нейронные сети для прогнозирования ходов. В этой главе мы начнем использовать для построения моделей популярную библиотеку глубокого обучения Keras.
- В главе 7 «Глубокое обучение бота на основе данных» мы применим знания, полученные в предыдущих двух главах, чтобы создать бота для игры в го на основе глубоких нейронных сетей. Мы обучим этого бота на фактических игровых данных сильных любительских партий и поговорим об ограничениях данного подхода.
- В главе 8 «Использование ботов» вы узнаете о том, как обеспечить игру ботов с противниками-людьми через пользовательский интерфейс. Кроме того, вы узнаете, как организовать игру ботов с другими ботами локально и на сервере для игры в го.

Обучение с подкреплением

- Глава 9 «Обучение на практике: обучение с подкреплением» посвящена основам обучения с подкреплением и описанию способов его использования для игры в го с самим собой.
- Глава 10 «Обучение с подкреплением и градиенты политики» знакомит читателя с градиентами политики, жизненно важным методом для повышения эффективности прогнозирования ходов, о котором мы говорили в главе 7.
- Глава 11 «Обучение с подкреплением и методы на основе значений» демонстрирует процесс оценки состояний доски с помощью так называемых методов на основе значений, которые являются мощным инструментом в сочетании с поиском по дереву, описанным в главе 4.
- Глава 12 «Обучение с подкреплением и методы типа “актор–критик”» знакомит читателя с методами, позволяющими предсказать долгосрочное значение конкретной позиции на доске и конкретного следующего хода для более эффективного выбора ходов.

В последней части III «Больше, чем сумма всех частей» мы соберем все описанные ранее строительные блоки, чтобы создать приложение, принципом работы напоминающее программу AlphaGo.

- Глава 13 «AlphaGo: Собираем все вместе», по сути, является технической и математической кульминацией этой книги. В ней мы обсуждаем, как

тренировка нейронной сети на данных го (главы 5–7) и последующая игра с самим собой (главы 8–11) в сочетании с поиском по дереву (глава 4) позволяют создать бота для игры в го сверхчеловеческого уровня.

- Глава 14 «AlphaGo Zero: Интеграция поиска по дереву и обучения с подкреплением» посвящена описанию современного состояния ИИ для настольных игр. В ней мы подробно поговорим об инновационной комбинации поиска по дереву и обучения с подкреплением, лежащей в основе программы AlphaGo Zero.

В приложениях мы рассмотрим следующие темы:

- в приложении А «Математические основы» кратко изложены некоторые базовые концепции линейной алгебры и математического анализа и приведены способы представления некоторых структур линейной алгебры с помощью библиотеки Python NumPy;
- в приложении Б «Алгоритм обратного распространения ошибки» более подробно объясняется процедура обучения большинства нейронных сетей, которые мы начнем использовать в главе 5;
- в приложении В «Программы и серверы для игры го» перечислены некоторые ресурсы для читателей, желающих больше узнать о данной игре;
- приложение Г «Обучение и развертывание ботов с помощью Amazon Web Services» представляет собой краткое руководство по запуску вашего бота на облачном сервере Amazon;
- в приложении Д «Отправка бота на онлайн-сервер для игры в го» говорится о том, как подключить бота к популярному серверу для игры в го, где вы можете протестировать его в игре с игроками со всего мира.

Структура книги схематически представлена на рисунке на следующей странице.

О КОДЕ

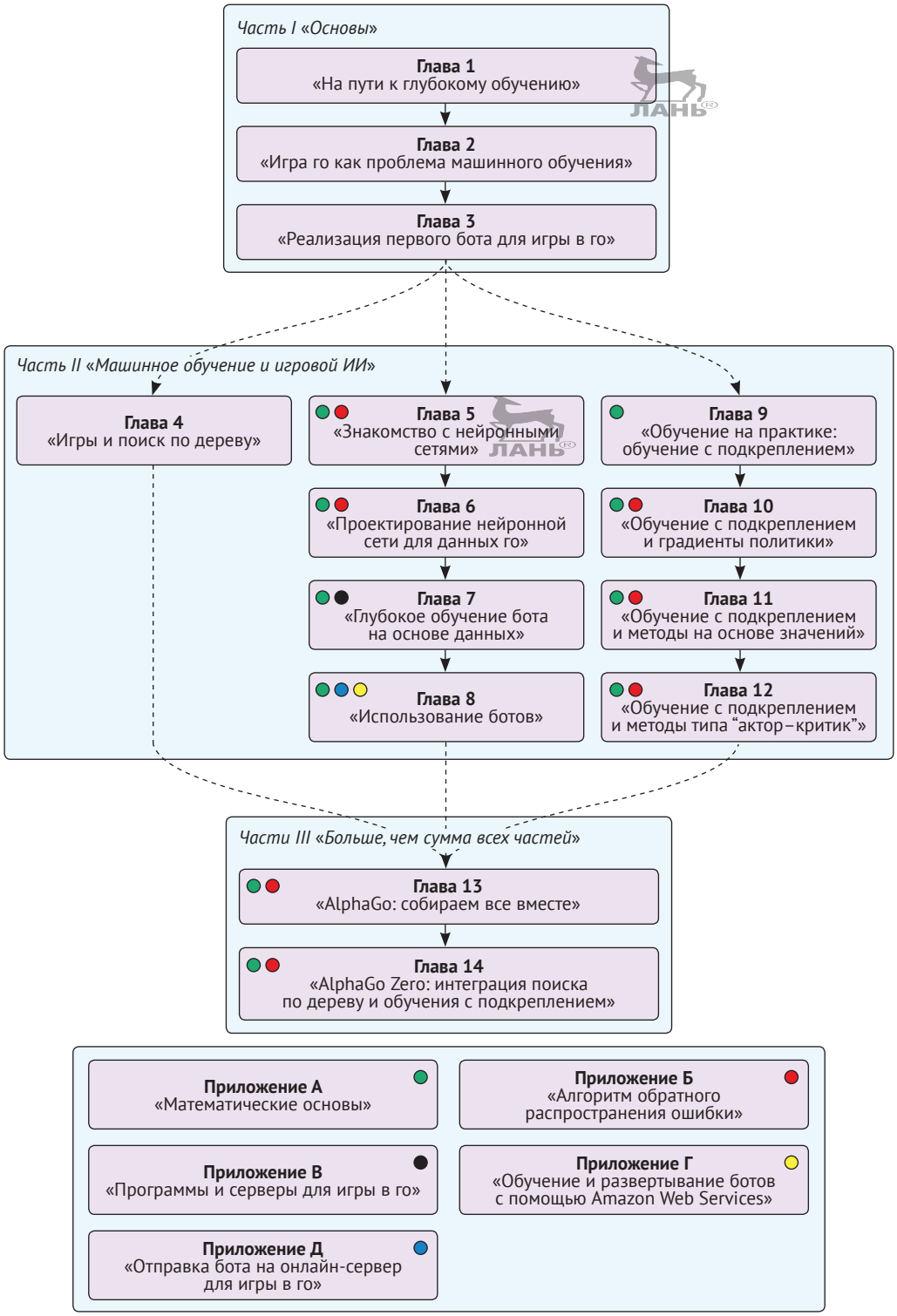
Эта книга содержит много примеров исходного кода как в листингах с пронумерованными строками, так и непосредственно в тексте. В обоих случаях код отформатирован `моноширинным` шрифтом, что позволяет отличить его от обычного текста. **`Полужирным моноширинным`** шрифтом выделяются изменения в коде, например новая функция, добавленная в уже существующую строку.

Во многих случаях мы переформатировали исходный код, добавив разрывы строк и изменив отступы с учетом доступного места на странице. В редких случаях даже этого оказалось недостаточно, поэтому мы включили в листинги символы продолжения строки (➥). Кроме того, мы убрали из листинга комментарии, относящиеся к коду, описанному в тексте. Многие листинги сопровождаются аннотациями, содержащими описание важных понятий.

Все примеры кода, а также некоторые дополнительные фрагменты связующего кода вы можете найти на сайте GitHub по адресу: **github.com/maxpumperla/deep_learning_and_the_game_of_go**.

Такая пиктограмма обозначает совет или рекомендацию.

Такая пиктограмма обозначает указание или примечание общего характера.


Часть I «Основы»
Глава 1
«На пути к глубокому обучению»
Глава 2
«Игра го как проблема машинного обучения»
Глава 3
«Реализация первого бота для игры в го»
Часть II «Машинное обучение и игровой ИИ»
Глава 4
«Игры и поиск по дереву»
Глава 5
«Знакомство с нейронными сетями»
Глава 6
«Проектирование нейронной сети для данных го»
Глава 7
«Глубокое обучение бота на основе данных»
Глава 8
«Использование ботов»
Глава 9
«Обучение на практике: обучение с подкреплением»
Глава 10
«Обучение с подкреплением и градиенты политики»
Глава 11
«Обучение с подкреплением и методы на основе значений»
Глава 12
«Обучение с подкреплением и методы типа “актор–критик”»
Части III «Больше, чем сумма всех частей»
Глава 13
«AlphaGo: собираем все вместе»
Глава 14
«AlphaGo Zero: интеграция поиска по дереву и обучения с подкреплением»
Приложение А
«Математические основы»
Приложение Б
«Алгоритм обратного распространения ошибки»
Приложение В
«Программы и серверы для игры в го»
Приложение Г
«Обучение и развертывание ботов с помощью Amazon Web Services»
Приложение Д
«Отправка бота на онлайн-сервер для игры в го»

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 прямо на нашем сайте **www.dmkpress.com**, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу **dmkpress@gmail.com**, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы 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**.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте **www.dmkpress.com** или **www.дмк.рф** на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу **dmkpress@gmail.com**, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Manning очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу электронной почты **dmkpress@gmail.com** со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Часть I

ОСНОВЫ

Что такое машинное обучение? Что собой представляет игра в го, и почему она является важной вехой на пути развития игрового ИИ? Чем обучение компьютера игре в го отличается от его обучения игре в шахматы или шашки?

В этой части мы ответим на все эти вопросы, после чего вы создадите гибкую библиотеку игровой логики го, которая станет основой для упражнений, приведенных в остальной части книги.

Глава 1

На пути к глубокому обучению: введение в машинное обучение

В этой главе:

- машинное обучение и его отличия от традиционного программирования;
- проблемы, которые можно и нельзя решить с помощью машинного обучения;
- связь машинного обучения и искусственного интеллекта;
- структура системы машинного обучения;
- дисциплины, относящиеся к области машинного обучения.

На протяжении всего срока существования компьютеров программисты интересовались *искусственным интеллектом* (ИИ), который позволил бы реализовать человеческое поведение с помощью компьютера. С давних пор популярным предметом исследований в области ИИ являются игры. В эпоху персональных компьютеров ИИ побеждал людей в игре в шашки, нарды, шахматы и почти во все остальные классические настольные игры. Однако на протяжении многих десятилетий древняя стратегическая игра го оставалась недостижимой для компьютеров. В 2016 году ИИ AlphaGo компании Google DeepMind бросил вызов 14-кратному чемпиону мира Ли Седолю и одержал победу в четырех играх из пяти. Следующая версия AlphaGo оказалась совершенно недосягаемой для игроков-людей: она выиграла 60 игр подряд, победив практически всех известных игроков в го.

Прорыв AlphaGo заключался в расширении классических алгоритмов ИИ с помощью машинного обучения. В частности, программа AlphaGo использовала современные методы *глубокого обучения* – алгоритмы, способные организовывать необработанные данные в полезные уровни абстракции. Применение этих методов не ограничивается играми. Глубокое обучение используется в приложениях для распознавания изображений и речи, системах машинного перевода и управления роботами. Изучение основ глубокого обучения поможет вам разобраться в принципах работы всех этих приложений.

Зачем посвящать целую книгу компьютеру, играющему в го? Авторы не сошли с ума, дело в том, что, в отличие от ИИ для игры в шахматы или нарды, мощный ИИ для игры в го требует применения методов глубокого обучения. Первоклассный шахматный движок вроде Stockfish содержит обширные знания о логике

игры в шахматы. Для создания чего-то подобного вы должны обладать определенными знаниями об игре. Глубокое обучение позволяет научить компьютер подражать сильным игрокам в го, даже если вы не понимаете логику их действий. И эта мощная техника позволяет создавать всевозможные виды приложений как в игровом, так и в реальном мире.

ИИ для игры в шахматы и шашки призваны более точно прогнозировать игру по сравнению с людьми. Применение данного подхода к игре в го имеет две сложности. Во-первых, вы не можете заглядывать достаточно далеко вперед из-за слишком большого количества возможных ходов. Во-вторых, даже если бы вы могли прогнозировать игру на много ходов вперед, вы не смогли бы оценить результат. Ключом к решению обеих проблем является глубокое обучение.

Эта книга позволяет познакомиться с темой глубокого обучения путем рассмотрения методов, лежащих в основе алгоритма AlphaGo. Вам не придется подробно изучать игру в го, вместо этого вы познакомитесь с общими принципами обучения компьютера. Первая глава посвящена машинному обучению и проблемам, которые можно (и нельзя) решить с его помощью. Мы рассмотрим примеры, иллюстрирующие основные направления в сфере машинного обучения, и увидим, как глубокое обучение расширило область применения машинного обучения.

1.1. Что такое машинное обучение?

Рассмотрим задачу распознавания фотографии друга. Для большинства людей это не является проблемой, даже если фотография отличается плохим освещением, друг постригся или надел другую рубашку. Но что, если вы хотите запрограммировать компьютер для решения этой же задачи? С чего бы вы начали? Это та проблема, которую можно решить с помощью машинного обучения.

Традиционное компьютерное программирование подразумевает применение четких правил к структурированным данным. Разработчик-человек программирует компьютер с целью применения к данным набора инструкций для получения желаемого результата (см. рис. 1.1). Представьте себе налоговую декларацию: каждое поле имеет четко определенное значение, а его заполнение подчиняется конкретным правилам. Сложность этих правил зависит от того, где вы живете. При заполнении человек может легко допустить ошибку, однако компьютерные программы превосходно справляются с этой задачей.

В отличие от традиционной парадигмы программирования, *машинное обучение* представляет собой семейство методов, позволяющих вывести программу или алгоритм из некоторых данных вместо их непосредственной реализации. Таким образом, при машинном обучении вы, как и раньше, предоставляете данные своему компьютеру, но вместо навязывания конкретных инструкций и ожидания результата *вы предоставляете ожидаемый результат и позволяете машине самостоятельно найти алгоритм*.

Чтобы создать компьютерную программу для распознавания лица на фотографии, вы можете применить алгоритм, который анализирует большую коллекцию изображений вашего друга и генерирует соответствующую им функцию. Если вы все сделаете правильно, то сгенерированная функция также будет соответствовать новым фотографиям, которые вы еще не видели. Разумеется, программа не будет знать о своем назначении. Она способна лишь идентифицировать изображения, похожие на предоставленные ей оригиналы.

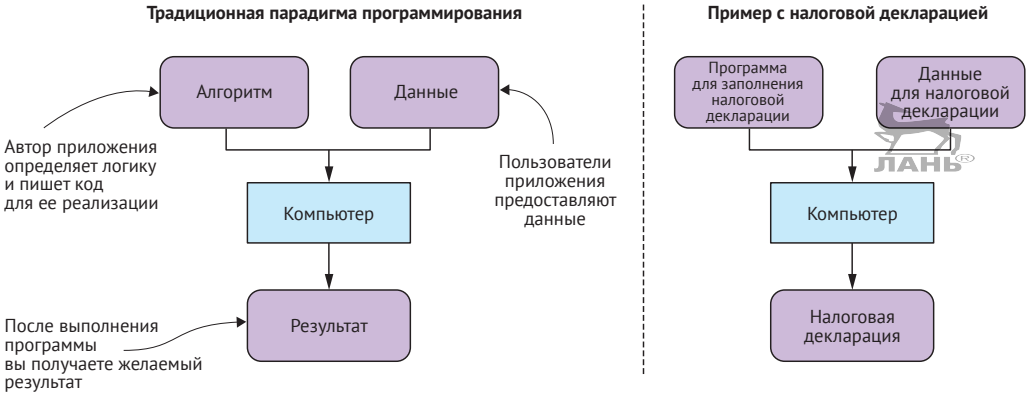


Рис. 1.1 ❖ Стандартная парадигма программирования, знакомая большинству разработчиков программного обеспечения. Разработчик идентифицирует алгоритм и реализует код, а пользователи предоставляют данные

В данном случае изображения, предоставляемые программе, называются *обучающими данными*, а имена людей на изображениях – *метками*. После *обучения* алгоритма вы сможете использовать его для *предсказания* меток на новых данных с целью его тестирования. На рис. 1.2 показан этот пример и схематически представлена парадигма машинного обучения.

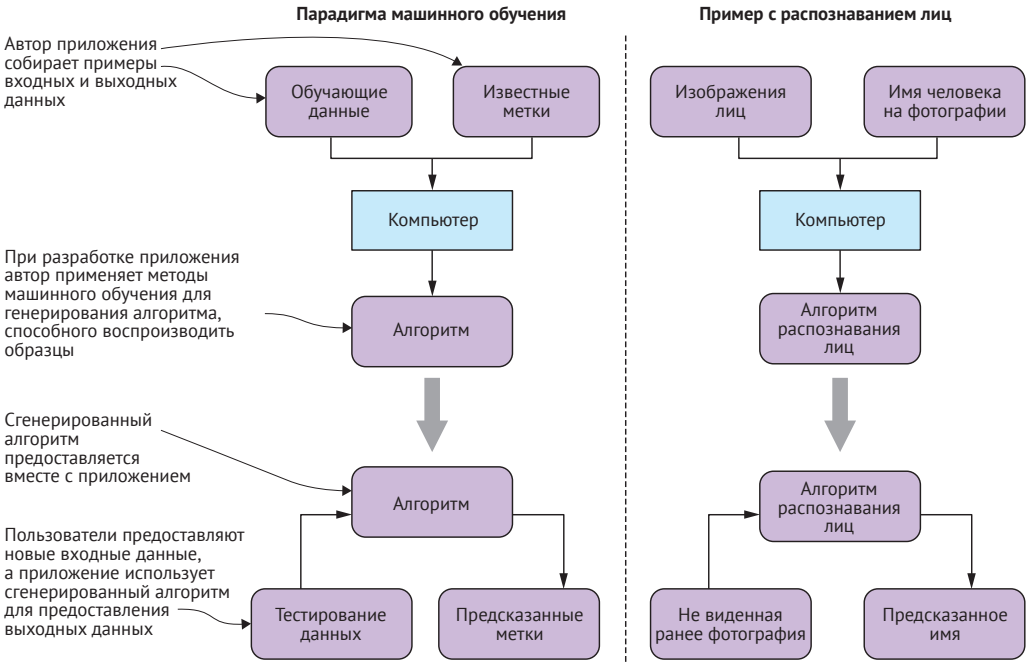


Рис. 1.2 ❖ Парадигма машинного обучения: в процессе разработки вы генерируете алгоритм на основании набора данных, а затем включаете его в итоговое приложение

Машинное обучение необходимо в ситуациях, когда правила не ясны. Оно может использоваться для решения проблем вроде «я узнаю это, когда увижу». Вместо непосредственного программирования функции вы предоставляете данные, указывающие, что эта функция должна делать, а затем методично генерируете функцию, соответствующую вашим данным.

На практике машинное обучение обычно комбинируется с традиционным программированием с целью создания полезной программы. В случае нашего приложения для распознавания лиц, прежде чем применять алгоритм машинного обучения, вы должны проинструктировать компьютер о том, как находить, загружать и преобразовывать примеры изображений. Кроме того, вы можете использовать эвристические алгоритмы, чтобы отделить изображения лиц от фотографий закатов и латте-арт, после чего применить методы машинного обучения для сопоставления имен и лиц. Часто сочетание традиционных методов программирования и продвинутых алгоритмов машинного обучения обеспечивает лучший результат, чем применение только одного из них.

1.1.1. Связь машинного обучения и искусственного интеллекта

Термин *искусственный интеллект* в широком смысле слова может быть применен к любому методу, позволяющему компьютерам имитировать человеческое поведение. ИИ предусматривает огромный спектр методов, включая следующие:

- логические продукционные системы, применяющие формальную логику для оценки утверждений;
- экспертные системы, в которых программисты пытаются закодировать человеческие знания непосредственно в программное обеспечение;
- нечеткая логика, определяющая алгоритмы, помогающие компьютерам обрабатывать неточные высказывания.

Такие методы, основанные на правилах, иногда называются *классическим ИИ*, или *GOFAI* (Good Old-Fashioned Artificial Intelligence, старый добрый искусственный интеллект).

Машинное обучение – это лишь одно из многих направлений в области ИИ, но, пожалуй, наиболее успешное на сегодняшний день. В частности, глубокое обучение позволило совершить некоторые из самых захватывающих прорывов в области искусственного интеллекта и решить задачи, к которым на протяжении многих десятилетий исследователи не могли подступиться. В случае классического ИИ исследователи изучают поведение человека и пытаются закодировать соответствующие ему правила. Машинное обучение и глубокое обучение ставят все с ног на голову: теперь вы собираете примеры человеческого поведения и выводите его правила с помощью математических и статистических методов.

Глубокое обучение используется повсеместно, поэтому для некоторых участников сообщества термины *ИИ* и *глубокое обучение* являются взаимозаменяемыми. Для ясности мы будем использовать понятие *ИИ*, говоря об общей проблеме имитации человеческого поведения с помощью компьютеров, а понятия *машинного обучения* или *глубокого обучения* – говоря о математических методах выведения алгоритмов из примеров.

1.1.2. Что можно и чего нельзя сделать с помощью машинного обучения

Машинное обучение представляет собой специализированную технику. Вы не будете использовать машинное обучение для обновления записей базы данных или визуализации пользовательского интерфейса. В следующих ситуациях предпочтительным является традиционное программирование:

- **проблему можно решить с помощью традиционных алгоритмов**: если для решения проблемы можно написать код, то его будет проще понимать, поддерживать, тестировать и отлаживать;
- **вы ожидаете идеальной точности**: все сложные программы содержат ошибки. Однако при традиционном подходе к разработке программного обеспечения вы методически выявляете и исправляете их. Это не всегда возможно при использовании машинного обучения. Вы можете улучшить систему машинного обучения, однако слишком большое внимание, уделенное конкретной ошибке, часто приводит к ухудшению системы в целом;
- **хорошо работают простые эвристические алгоритмы**: если вы можете реализовать достаточно хорошее правило с помощью нескольких строк кода, сделайте это и будьте счастливы. Простую четко реализованную эвристику будет легко понять и поддерживать. Функции, реализуемые с помощью машинного обучения, не ясны, а для их обновления требуется отдельный процесс обучения. (С другой стороны, если вам приходится поддерживать сложную последовательность эвристических алгоритмов, то их можно заменить машинным обучением.)

Часто существует тонкая грань между проблемами, которые можно решить с помощью традиционного программирования, и проблемами, которые практически невозможно решить даже с помощью машинного обучения. Распознавание лиц на изображениях и их сопоставление с именами – это только один пример. Другим примером является определение языка, на котором написан текст, и перевод текста на конкретный язык.

Мы часто выбираем традиционное программирование, когда машинное обучение могло бы помочь, например при чрезвычайно высокой сложности проблемы. Столкнувшись с очень сложными, насыщенными информацией сценариями, люди склонны прибегать к эмпирическим правилам и изложению фактов, например когда речь идет о макроэкономике, фондовом рынке или политике. Менеджеры процессов и так называемые эксперты часто могут извлечь большую пользу из идей, полученных в результате машинного обучения. Зачастую данные из реального мира оказываются более структурированными, чем это предполагалось, и мы только начинаем использовать преимущества автоматизации и усиления интеллекта во многих из этих областей.

1.2. Пример машинного обучения

Цель машинного обучения заключается в создании функции, которую было бы сложно реализовать напрямую. Для этого вы выбираете *модель*, большое семейство родовых функций. Затем вам требуется процедура для выбора из этого се-

мейства функции, соответствующей вашей цели. Этот процесс называется *обучением*, или *подбором*, модели. Разберем это на простом примере.

Допустим, вы собираете значения роста и веса нескольких людей и наносите их на график. На рис. 1.3 показаны данные, взятые из реестра членов профессиональной футбольной команды.

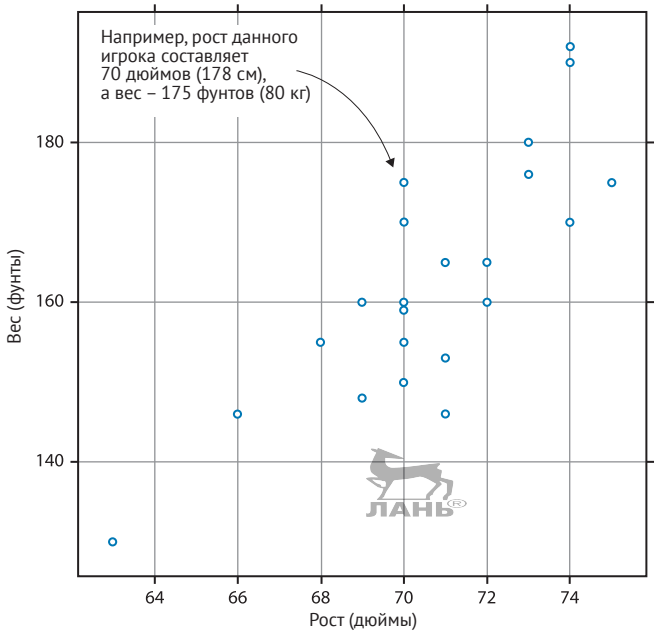


Рис. 1.3 ❖ Простой пример набора данных. Каждая точка на графике соответствует росту и весу футболиста. Ваша цель заключается в подборе модели, соответствующей этим точкам

Допустим, вы хотите описать эти точки с помощью математической функции. Во-первых, обратите внимание на то, что точки образуют почти прямую линию, направленную вверх и вправо. Из школьного курса алгебры вам известно, что прямые линии описываются функцией вида $f(x) = ax + b$. Вы можете предположить возможность нахождения таких значений a и b, при которых выражение $ax + b$ будет достаточно близко соответствовать вашим точкам данных. Значения a и b – это *параметры*, или *веса*, которые вам необходимо выяснить. Это ваша модель. Вы можете написать код на языке Python, способный генерировать любую функцию из этого семейства:

```
class GenericLinearFunction:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def evaluate(self, x):
        return self.a * x + self.b
```

Как узнать правильные значения *a* и *b*? Для этого можно использовать строгие алгоритмы, однако для получения быстрого и приблизительного решения вы можете просто начертить на графике линию с помощью линейки и попытаться вывести ее формулу. На рис. 1.4 показана линия, которая соответствует закономерности распределения точек данных.

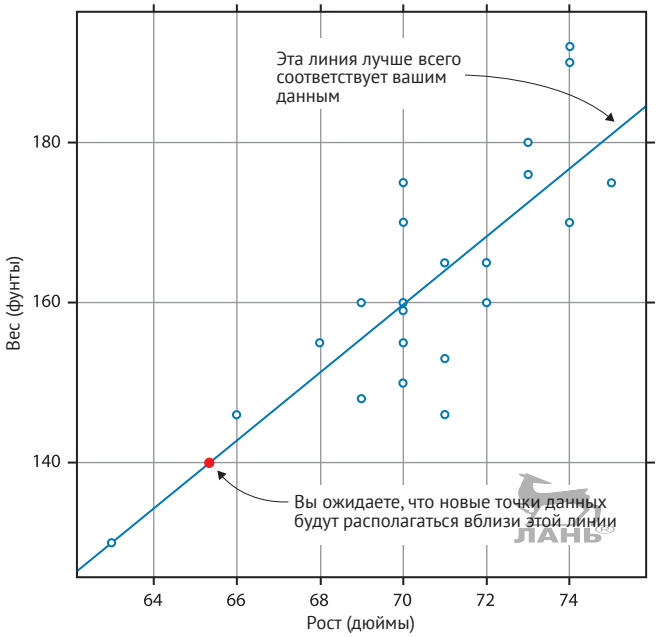


Рис. 1.4 ❖ Сначала вы замечаете, что точки данных распределены вдоль некоторой линии, а затем выводите формулу этой линии

Если вы посмотрите на пару точек, через которые проходит линия, то сможете вывести для этой линии формулу. В результате вы получите что-то вроде $f(x) = 4{,}2x - 137$. Теперь у вас есть конкретная функция, соответствующая вашим данным. Если вы измерите рост нового человека, то сможете использовать свою формулу для приблизительной оценки веса этого человека. Полученное значение будет не точным, но достаточно близким для того, чтобы оказаться полезным. Вы можете превратить `GenericLinearFunction` в конкретную функцию:

```
height_to_weight = GenericLinearFunction(a=4.2, b=-137)
height_of_new_person = 73
estimated_weight = height_to_weight.evaluate(height_of_new_person)
```

Эта функция позволит вам получить довольно хорошую оценку, при условии что ваш новый человек также будет являться профессиональным футболистом. Все данные в вашем наборе относятся к взрослым мужчинам в довольно узком возрастном диапазоне, которые каждый день занимаются одним и тем же видом спорта. Если вы попытаетесь применить свою функцию к женщинам-футболисткам, олимпийским тяжелоатлетам или детям, то получите крайне неточные результаты. Ваша функция соответствует только вашим обучающим данным.

Так выглядит простейший процесс машинного обучения. В данном случае вашей моделью является семейство всех функций вида $f(x) = ax + b$. На самом деле даже такие простые модели оказываются полезными, поэтому статистики используют их постоянно. При решении более сложных проблем применяются более сложные модели и более продвинутые методы обучения. Однако основная идея остается прежней: сначала описывается большое семейство возможных функций, среди которых затем выбирается лучшая.

Python и машинное обучение

Все примеры кода, приведенные в этой книге, написаны на языке программирования Python. Почему именно на нем? Во-первых, Python представляет собой выразительный высокоуровневый язык, предназначенный для разработки приложений. Кроме того, Python является одним из самых популярных языков для машинного обучения и математического программирования. Эта комбинация делает Python естественным выбором для создания приложения с элементами машинного обучения.

Язык Python широко применяется в сфере машинного обучения благодаря своей удивительной коллекции пакетов для числовых вычислений. В этой книге мы используем следующие из них:

- **NumPy** – эта библиотека предусматривает эффективные структуры данных для представления числовых векторов и массивов, а также обширную библиотеку быстрых математических операций. NumPy является ядром вычислительной экосистемы Python: все известные библиотеки для машинного обучения или статистики интегрированы с NumPy;
- **TensorFlow и Theano** – это две библиотеки для построения графа вычислений (*граф* – это сеть связанных между собой вершин). Они позволяют определять сложные последовательности математических операций, а затем генерировать высоко оптимизированные реализации;
- **Keras** – это высокоуровневая библиотека для глубокого обучения. Она обеспечивает удобный способ создания нейронных сетей, а для реализации вычислений использует библиотеки TensorFlow или Theano.

Примеры кода, приведенные в этой книге, написаны с учетом Keras 2.2 и TensorFlow 1.8. Внеся минимальные изменения, вы сможете использовать любую версию библиотеки Keras серии 2.x.

1.2.1. Использование машинного обучения в приложениях

В предыдущем разделе мы рассмотрели чисто математическую модель. Как можно применить машинное обучение к реальной программе?

Предположим, вы работаете над приложением для обмена фотографиями, в которое пользователи загрузили миллионы изображений с тегами. Вы хотите добавить функцию, которая предлагает тег для новой фотографии. Для создания этой функции идеально подойдут методы машинного обучения.

Во-первых, вы должны четко указать функцию, которую намерены обучить. Допустим, у вас была такая функция:

```
def suggest_tags(image_data):
    """Предлагает теги для изображения.

    Входные данные: image_data - это фотография в растровом формате

    Возвращает: ранжированный список рекомендуемых тегов
    """
```

Дальше все довольно просто. Однако не совсем ясно, с чего следует начинать реализацию такой функции, как `offer_tags`. И тут в игру вступает машинное обучение.

Обычная функция Python принимала бы в качестве входных данных некоторый объект `Image`, а в качестве результата, вероятно, возвращала бы список строк. Алгоритмы машинного обучения не так гибки в отношении своих входных и выходных данных, обычно они работают с векторами и матрицами. Поэтому сначала вы должны математически представить свои входные и выходные данные.

Если вы изменяете размер входной фотографии до фиксированной величины, скажем, 128×128 пикселов, то можете закодировать ее как матрицу, состоящую из 128 строк и 128 столбцов, – по одному значению с плавающей запятой на пиксел. А как насчет выходных данных? Одним из вариантов является ограничение набора идентифицируемых вами тегов: вы можете выбрать, например, 1000 самых популярных тегов в приложении. Выходные данные могли бы представлять собой вектор из 1000 элементов, каждый из которых соответствует определенному тегу. Позволив выходным значениям варьироваться в диапазоне от 0 до 1, вы сможете создать ранжированные списки предлагаемых тегов. На рис. 1.5 показаны концепции, лежащие в основе вашего приложения, и соответствующие им математические структуры.

Только что выполненный вами шаг предварительной обработки данных является неотъемлемой частью любой процедуры машинного обучения. Обычно вы загружаете необработанные данные и выполняете их предварительную обработку для создания *признаков* – входных данных, подаваемых алгоритму машинного обучения.

1.2.2. Обучение с учителем

Теперь вам нужен алгоритм для обучения вашей модели. В данном случае у вас уже есть миллионы правильных примеров – все фотографии, которые пользователи уже загрузили и вручную отметили в вашем приложении. Вы можете вывести функцию, максимально точно соответствующую этим примерам, в надежде, что она будет достаточно хорошо соответствовать и новым фотографиям. Этот метод называется *обучением с учителем*, поскольку в процессе обучения руководством служат *метки*, присвоенные человеком.

После завершения обучения вы сможете интегрировать окончательную функцию в ваше приложение. Каждый раз, когда пользователь загружает новую фотографию, вы передаете ее обученной функции и получаете вектор. Вы можете сопоставить каждое значение в векторе с соответствующим тегом, а затем выбрать теги с самыми большими значениями и показать их пользователю. Только что описанная процедура схематически представлена на рис. 1.6.

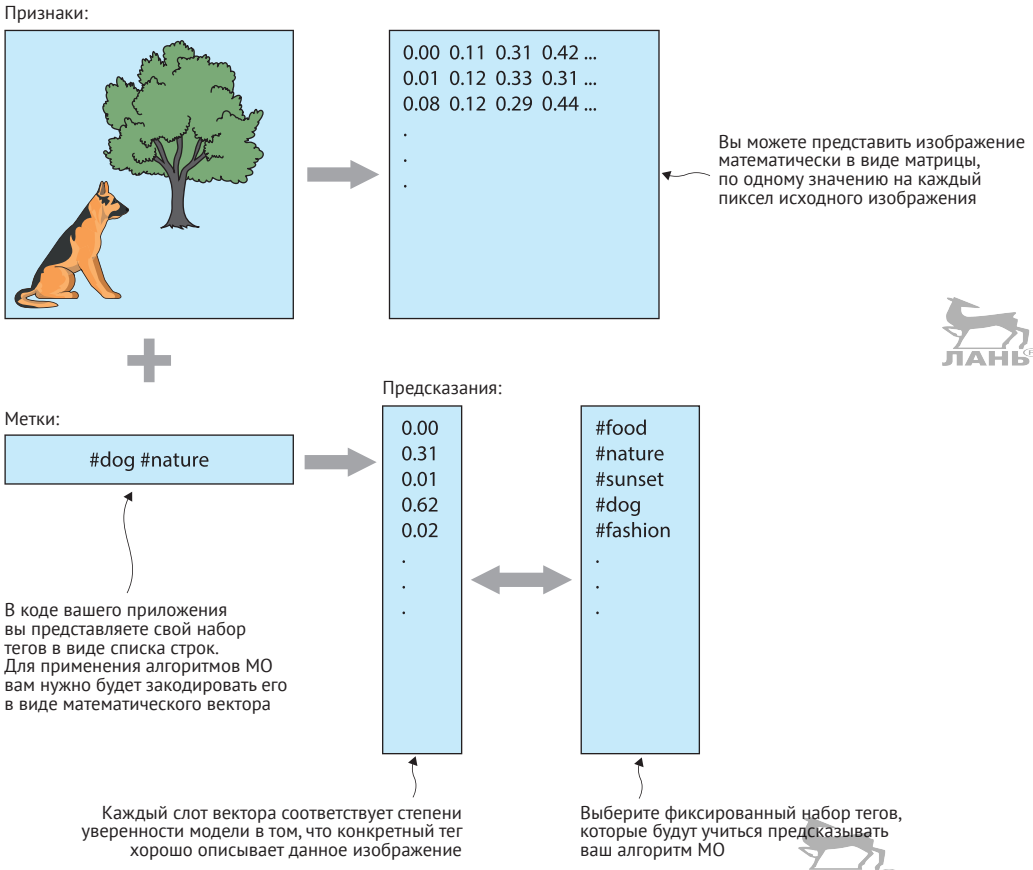


Рис. 1.5 ❖ Алгоритмы машинного обучения применяются к таким математическим структурам, как векторы и матрицы. Ваши теги для фотографий хранятся в стандартной компьютерной структуре данных – в списке строк. Это одна из возможных схем кодирования списка в виде математического вектора

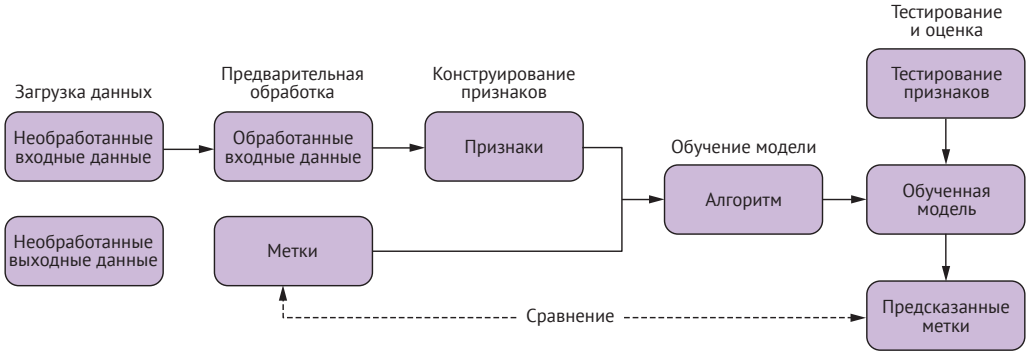


Рис. 1.6 ❖ Конвейер машинного обучения при обучении с учителем

Как протестировать обученную модель? Стандартная практика предполагает использование для этой цели некоторого количества исходных помеченных данных. До начала обучения вы можете выделить часть своих данных, например 10 %, в качестве *контрольного набора*. Этот контрольный набор *не включается* в набор данных, используемых для обучения. Затем вы можете применить свою обученную модель к изображениям из контрольного набора и сравнить предложенные теги с уже известными подходящими тегами. Это позволяет определить степень точности вашей модели. Если вы захотите поэкспериментировать с различными моделями, у вас будет единая метрика для определения лучшей из них.

В случае игрового ИИ вы можете извлечь помеченные данные для обучения из записей партий, сыгранных людьми. Отличным ресурсом данных для машинного обучения являются онлайн-игры, поскольку игровой сервер может хранить записи сыгранных пользователями партий. Обучение с учителем может быть применено к играм следующим образом:

- при наличии набора полных записей шахматных партий представьте игровое состояние в векторной или матричной форме и научитесь предсказывать следующий ход, исходя из имеющихся данных;
- научитесь прогнозировать вероятность выигрыша, исходя из текущего состояния доски.

1.2.3. Обучение без учителя

В отличие от обучения с учителем, *обучение без учителя* не предусматривает использования каких-либо меток, которые направляли бы процесс обучения. При обучении без учителя алгоритм должен научиться самостоятельно находить закономерности во входных данных. Единственное отличие от схемы на рис. 1.6 заключается в отсутствии меток, поэтому вы не можете оценить свои прогнозы, как раньше. Все остальные компоненты остаются прежними.

Примером может являться *обнаружение выбросов* – выявление точек данных, которые не согласуются с общей тенденцией набора. В наборе данных о футболистах выбросы указывают на игроков, телосложение которых не соответствует типичным показателям их товарищей по команде. Например, вы можете придумать алгоритм, измеряющий расстояние от точки рост–вес до намеченной линии. Если это расстояние превышает определенное значение, вы объявляете данную точку выбросом.

В ИИ для настольных игр естественным является вопрос о том, какие фигуры на доске образуют группу. В следующей главе вы узнаете, какое значение это имеет для игры в го. Нахождение групп связанных между собой элементов иногда называется *кластеризацией*, или *фрагментацией*. На рис. 1.7 это показано на примере шахмат.

1.2.4. Обучение с подкреплением

Обучение с учителем представляет собой мощный механизм, однако при его использовании серьезную проблему может представлять поиск качественных обучающих данных. Предположим, вы создаете робота для домашней уборки. Робот имеет различные датчики для детектирования находящихся рядом препятствий и моторы, позволяющие ему перемещаться по полу и поворачивать влево или вправо. Вам нужна система управления: функция, которая способна анализировать данные, полученные от датчика, и принимать решение относительно

дальнейшего движения робота. В данной ситуации обучение с учителем неприменимо. У вас нет примеров, которые можно было бы использовать в качестве обучающих данных, – ваш робот еще даже не существует.

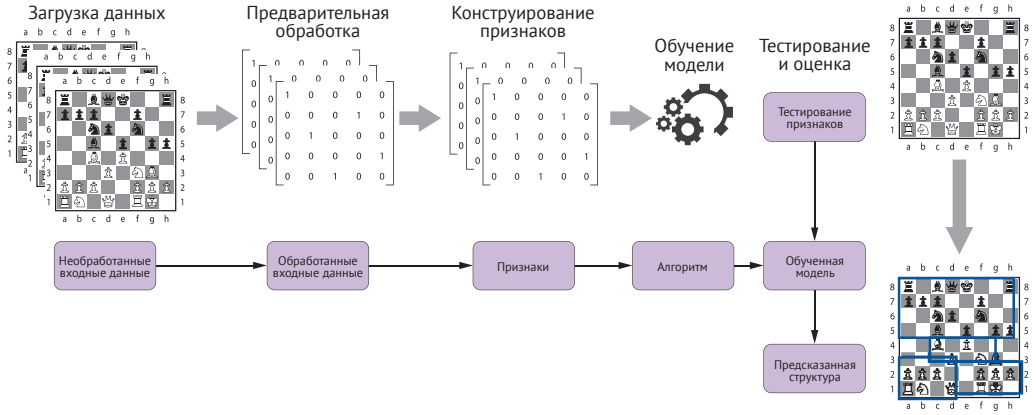


Рис. 1.7 ❖ Конвейер машинного обучения при обучении без учителя для нахождения кластеров шахматных фигур

Вместо этого вы можете воспользоваться своего рода методом проб и ошибок под названием *обучение с подкреплением*. Вы начинаете с неэффективной или неточной системы управления, а затем позволяете роботу выполнить свою задачу. Во время выполнения задачи вы записываете все входные данные, получаемые вашей системой управления, и решения, которые она принимает. После этого вам нужно будет оценить качество проделанной работы, например определить процент площади пола, который пропылесосил робот, и степень разряда батареи. В результате у вас появится небольшой объем обучающих данных, который вы сможете использовать для улучшения системы управления. Многократное повторение этого процесса позволит вам, в конце концов, вывести эффективную функцию управления. На рис. 1.8 этот процесс представлен в виде блок-схемы.

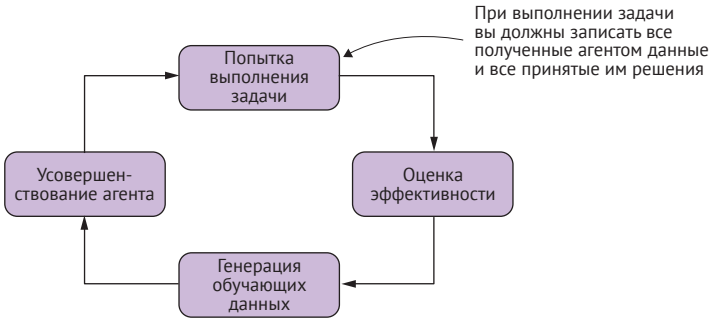


Рис. 1.8 ❖ При обучении с подкреплением агенты учатся взаимодействовать со своей средой методом проб и ошибок. Ваш агент многократно пытается выполнить свою задачу, собирая данные, на которых он может учиться. Каждый цикл позволяет вносить постепенные улучшения

1.3. Глубокое обучение

Текст этой книги состоит из предложений. Предложения состоят из слов, слова – из букв, буквы – из прямых и кривых линий, а эти линии, в свою очередь, – из крошечных чернильных точек. Обучение ребенка чтению начинается с самых маленьких фрагментов: сначала буквы, потом слова, затем предложения и, наконец, целые книги. (Обычно дети учатся распознавать линии самостоятельно.) Такая иерархия представляет собой естественный процесс изучения людьми сложных концепций. На каждом уровне игнорируются некоторые детали, и концепции становятся более абстрактными.

Глубокое обучение предполагает применение этой идеи к сфере машинного обучения. Глубокое обучение – это направление, в рамках которого используется определенное семейство моделей: последовательностей, состоящих из простых функций. Эти цепочки функций называются *нейронными сетями*, поскольку они несколько напоминают структуру естественного мозга. Основная идея глубокого обучения заключается в том, что эти последовательности функций могут проанализировать сложную концепцию как иерархию более простых. Первый слой глубокой модели может научиться принимать необработанные данные и организовывать их простыми способами, например группировать точки в линии. Каждый последующий слой организует предыдущий в более сложные и абстрактные концепции. Процесс изучения этих абстрактных понятий называется *репрезентативным обучением*.

Удивительным в глубоком обучении является то, что вам не нужно заранее знать, каковы эти промежуточные концепции. Если вы выберете модель с достаточным количеством слоев и предоставите достаточно обучающих данных, то в процессе обучения необработанные данные будут постепенно организованы в концепции все более высокого уровня. Откуда алгоритм обучения знает, какие концепции следует использовать? Он не знает, он просто организует входные данные таким способом, который позволяет обеспечить наилучшее соответствие обучающим примерам. Нет никаких гарантий, что это представление будет совпадать с человеческим способом обработки этих данных. На рис. 1.9 показано, как репрезентативное обучение вписывается в процесс обучения с учителем.

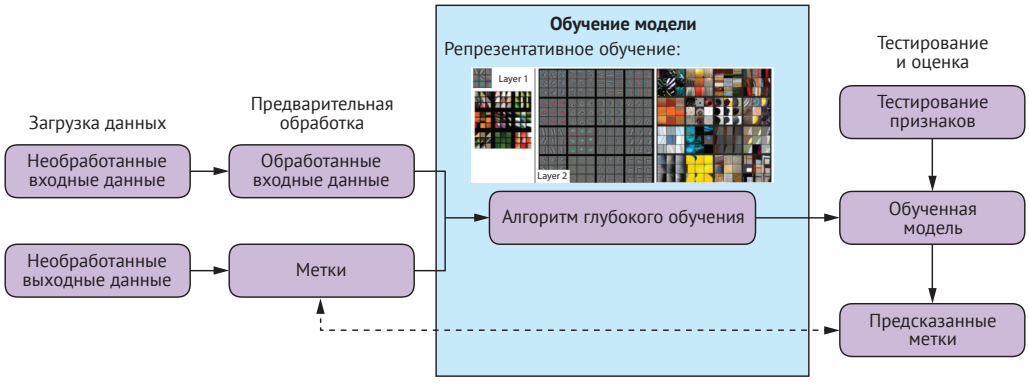


Рис. 1.9 ❖ Глубокое обучение и репрезентативное обучение

Вся эта мощь имеет свои издержки. Глубокие модели должны изучить огромное количество весов. Простой модели $ax + b$, которую вы применяли к набору данных о росте и весе футболистов, было достаточно двух весов. Глубокой модели, предназначенной для приложения с функцией тегирования изображений, может потребоваться миллион. Таким образом, глубокое обучение требует больших наборов данных, большей вычислительной мощности и более практичного подхода к обучению. Оба метода имеют свои сферы применения. Глубокое обучение является хорошим вариантом при следующих обстоятельствах:

- **ваши данные не структурированы**: изображения, аудиозаписи и тексты являются хорошими кандидатами на применение глубокого обучения. К таким данным можно применить простые модели, однако при этом обычно требуется сложная предварительная обработка;
- **вы обладаете большими объемами данных или знаете, как их получить**: как правило, чем сложнее модель, тем больше данных требуется для ее обучения;
- **у вас большие вычислительные мощности или много времени**. Глубокие модели требуют большого количества вычислений как в процессе обучения, так и в процессе оценки.

Традиционные модели с меньшим количеством параметров являются предпочтительными в следующих случаях:

- **ваши данные структурированы**: если ваши входные данные больше похожи на записи базы данных, вы зачастую можете применить простые модели напрямую;
- **вам нужна описательная модель**: в случае с простыми моделями вы можете посмотреть на выведенную в процессе обучения функцию и узнать, как отдельные входные данные влияют на результат. Это позволяет вам получить представление о том, как на самом деле работает изучаемая вами система. В случае с глубокими моделями связь между определенными входными данными и результатом является гораздо менее очевидной. Такую модель трудно интерпретировать.

Поскольку *глубокое обучение* относится к типу используемой модели, вы можете применить его к любому из основных направлений машинного обучения. Например, вы можете проводить обучение с учителем, используя глубокую или простую модель, в зависимости от типа имеющихся у вас обучающих данных.

1.4. Что вы узнаете из этой книги?

Эта книга представляет собой практическое введение в тему глубокого обучения и обучения с подкреплением. Чтобы получить максимальную пользу от этой книги, вы должны обладать навыком чтения и написания кода на языке Python, а также иметь некоторое представление о линейной алгебре и математическом анализе. Из этой книги вы узнаете, как:

- проектировать, обучать и тестировать нейронные сети с помощью библиотеки глубокого обучения Keras;
- осуществлять глубокое обучение с учителем;
- осуществлять обучение с подкреплением;
- интегрировать глубокое обучение в полезное приложение.

На протяжении всей книги мы будем создавать ИИ для игры в го. Наш бот будет сочетать в себе глубокое обучение со стандартными компьютерными алгоритмами. Мы будем использовать простой код Python для обеспечения соблюдения правил игры, отслеживания игрового состояния и прогнозирования последовательностей ходов. Глубокое обучение поможет боту определить перспективные ходы и лидера игры. На каждом этапе вы можете играть против своего бота и наблюдать за его прогрессом по мере применения все более изощренных приемов.

Если вас интересует сама игра го, то вы можете использовать созданного бота в качестве отправной точки для собственных экспериментов. Вы можете адаптировать описанные приемы к другим играм, а также добавлять функции, основанные на глубоком обучении, в другие приложения, не имеющие отношения к играм.

1.5. Резюме

- Машинное обучение – это семейство методов, позволяющих генерировать функции на основе данных вместо их непосредственного написания. Вы можете применить машинное обучение к проблемам, которые являются слишком неоднозначными для непосредственного решения.
- Как правило, процесс машинного обучения начинается с выбора *модели* – семейства математических функций. Затем вы *обучаете* эту модель, т. е. применяете алгоритм для нахождения лучшей функции в этом семействе. По большей части машинное обучение сводится к выбору подходящей модели и преобразованию конкретного набора данных для работы с ней.
- Тремя основными направлениями в сфере машинного обучения являются: обучение с учителем, обучение без учителя и обучение с подкреплением.
- Обучение с учителем предполагает выведение функции из заведомо верных примеров. При наличии примеров человеческого поведения или знаний вы можете имитировать их на компьютере с помощью методов обучения с учителем.
- Обучение без учителя предполагает извлечение из данных неизвестной заранее структуры. Обычно оно применяется для разбиения набора данных на логические группы.
- Обучение с подкреплением предполагает выведение функции методом проб и ошибок. Если вы способны написать код для оценки эффективности программы, то можете применить обучение с подкреплением для постепенного улучшения программы путем многократных испытаний.
- Глубокое обучение – это машинное обучение с использованием модели определенного типа, которая хорошо работает с такими неструктурированными входными данными, как изображения или текст. В настоящее время это одна из самых захватывающих областей информатики, которая постоянно расширяет наши представления о том, на что способны компьютеры.

Глава 2

Игра го как проблема машинного обучения

В этой главе:

- почему игры являются хорошим предметом для ИИ?
- почему игра го является подходящим случаем для применения глубокого обучения?
- каковы правила игры в го?
- какие игровые аспекты можно реализовать с помощью машинного обучения?

2.1. Почему игры?

Игры являются любимой темой для исследователей ИИ, и не только потому, что они доставляют массу удовольствия. Кроме того, они позволяют упростить некоторые сложности реальной жизни, чтобы вы могли сосредоточиться на изучаемых алгоритмах.

Представьте, что в Twitter или Facebook вы видите комментарий вроде: «Ой, я забыл свой зонт». Вы быстро придете к выводу о том, что ваш друг попал под дождь. Однако эта информация в предложении отсутствует. Как вы пришли к такому выводу? Во-первых, вы применили знания о том, для чего вообще нужны зонты. Во-вторых, вы применили знания о том, какие комментарии люди склонны оставлять в той или иной ситуации, – фраза «я забыл свой зонт» была бы неуместной в яркий солнечный день.

Люди легко учитывают весь этот контекст при чтении предложения. Но это нелегко сделать компьютерам. Современные методы глубокого обучения эффективны в плане обработки информации, которую вы им предоставляете. Однако вы ограничены в своей способности находить всю необходимую информацию и передавать ее компьютерам. Игры позволяют обойти эту проблему. Они имеют место в искусственной вселенной, где вся информация, необходимая для принятия решения, прописана в правилах.

К играм особенно хорошо применимо обучение с подкреплением. Как говорилось ранее, обучение с подкреплением предполагает многократный запуск программы с последующей оценкой ее результативности. Представьте, что вы используете обучение с подкреплением для того, чтобы обучить робота передвигаться по зданию. Прежде чем система управления будет точно настроена, существует риск

того, что робот упадет с лестницы или опрокинет вашу мебель. Другим вариантом является компьютерное моделирование среды, в которой роботу предстоит работать. Это исключает риски, связанные с работой необученного робота в реальном мире, но создает новые проблемы. Во-первых, вам необходимо инвестировать средства в разработку подробной компьютерной модели, что само по себе является важным проектом. Во-вторых, всегда есть вероятность того, что ваша симуляция окажется не вполне точной.

В случае с играми все, что вам нужно, – это заставить свой ИИ играть. Ничего страшного, если в процессе обучения он проиграет несколько сотен тысяч раз. В случае обучения с подкреплением игры являются необходимой составляющей серьезных исследований. Многие передовые алгоритмы были впервые продемонстрированы в таких видеоиграх Atari, как Breakout.

Вы *можете* успешно применять метод обучения с подкреплением к проблемам в физическом мире. Многие исследователи и инженеры делали это. Но если вы начнете с игр, то это позволит решить проблему создания реалистичной учебной среды и сосредоточиться на механике и принципах обучения с подкреплением.

В этой главе мы познакомим вас с правилами игры в го. Далее предоставим высокоуровневое описание структуры ИИ для настольной игры и определим этапы, на которых можно применить метод глубокого обучения. Наконец, мы расскажем о способах оценки прогресса игрового ИИ в процессе его разработки.

2.2. Краткое введение в игру го

Вам не обязательно быть сильным игроком в го, чтобы изучить эту книгу, но вы должны достаточно хорошо понимать правила этой игры, чтобы обеспечить их выполнение в компьютерной программе. К счастью, эти правила очень просты. Два игрока поочередно размещают на доске черные и белые камни, причем начинает игрок черными. Цель состоит в том, чтобы окружить своими камнями как можно большую часть доски.

Несмотря на простоту правил, стратегия игры в го отличается бесконечной глубиной, которую мы даже не будем пытаться описать в этой книге. Если вас интересует дополнительная информация, в конце этого раздела вы найдете ссылки на некоторые ресурсы.

2.2.1. Описание доски

Доска для игры в го представляет собой квадратную сетку (см. рис. 2.1). Камни помещаются на пересечения линий, а не внутри клеток. Стандартной является доска 19×19, но иногда игроки используют доску меньшего размера для проведения быстрой игры. Самыми популярными небольшими вариантами являются доски 9×9 и 13×13 (речь идет о количестве пересечений линий на доске, а не о количестве клеток).

Обратите внимание на девять точек. Эти точки называются *звездными пунктами*. Они позволяют игрокам определять расстояния на доске и никак не влияют на ход игры.

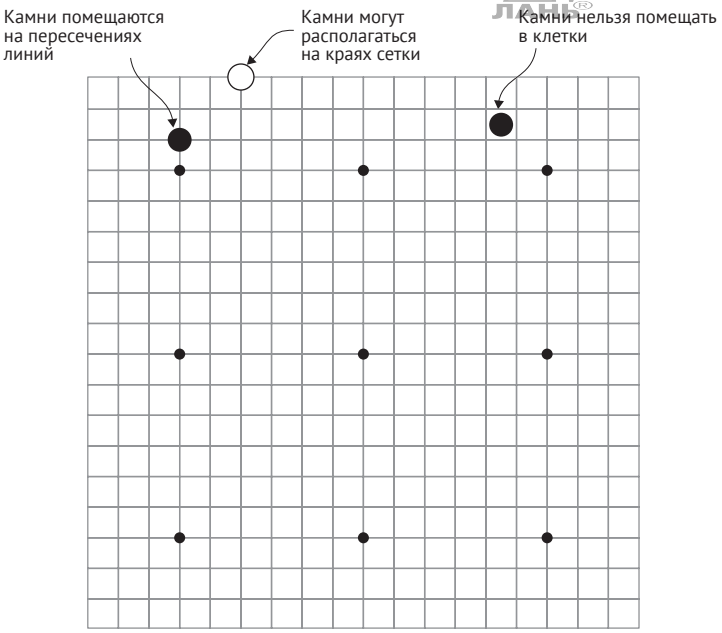


Рис. 2.1 ❖ Стандартная доска для игры в го 19×19. Пересечения, отмеченные точками, называются звездными пунктами и предназначены исключительно для ориентации игроков. Камни помещаются на пересечениях линий

2.2.2. Размещение и захват камней

Один игрок играет черными камнями, а другой – белыми. Два игрока поочередно размещают камни на доске, причем начинает игрок черными. Находящиеся на доске камни больше не передвигаются, однако они могут быть захвачены и удалены с доски. Чтобы захватить камни противника, вы должны полностью окружить их своими. Вот как это работает.

Соприкасающиеся камни одного цвета считаются соединенными, как показано на рис. 2.2. Камни могут соединяться только по прямой – вверх, вниз, влево или вправо. Соединение по диагонали не считается. Любая пустая точка, граничащая с соединенной группой камней, называется *степенью*, или *пунктом*, *свободы* этой группы. Каждой группе нужна хотя бы одна степень свободы, для того чтобы оставаться на доске. Вы можете захватывать камни противника, заполняя их пункты свободы.

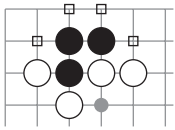


Рис. 2.2 ❖ Три соединенных черных камня. Квадратами отмечены четыре их пункта свободы. Игрок белыми камнями может захватить черные камни, поместив белые камни на все эти пункты свободы

Когда вы помещаете камень в последний пункт свободы группы противника, эта группа считается захваченной и убирается с доски. Вновь освобожденные пункты становятся доступными для совершения игроками разрешенных ходов. С другой стороны, вы не можете поместить камень в пункт с нулевой степенью свободы, *если при этом не происходит захват камней противника*.

Из этих правил вытекает интересное следствие. Если внутри группы камней имеется два не соприкасающихся между собой пункта свободы, то эта группа не может быть захвачена. На рис. 2.3 показана ситуация, когда черный камень не может быть помещен в точку A, поскольку в этом случае у данного черного камня не будет степеней свободы и его размещение не завершит захват из-за оставшегося пункта свободы в точке B. По той же причине черный камень не может быть помещен в точку B. Таким образом, игрок черными не может заполнить последние два пункта свободы белой группы. Эти внутренние пункты свободы называются *глазами*. Напротив, черный камень может быть помещен в точку C для захвата пяти белых камней, поскольку в этом случае, несмотря на отсутствие пунктов свободы у черного камня, он совершит захват. Эта белая группа имеет только один глаз, поэтому обречена быть захваченной на каком-то этапе.

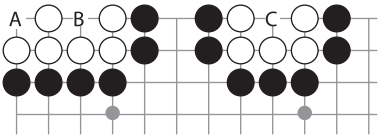


Рис. 2.3 ❖ Группа белых камней слева никогда не может быть захвачена: черный камень нельзя поместить ни в точку A, ни в точку B. Черный камень в этих пунктах не имел бы ни одной степени свободы, и поэтому такой ход не является разрешенным. С другой стороны, черный камень можно поместить в пункт C, чтобы захватить группу из пяти белых камней

Хотя это явно не указано в правилах, идея о том, что группа с двумя глазами не может быть захвачена, является базовой частью стратегии игры в го. Именно ее вам предстоит специально закодировать в логику своего бота. Более продвинутые стратегии игры в го будут выведены в процессе машинного обучения.

2.2.3. Завершение игры и подсчет очков

Любой игрок может пропустить ход, вместо того чтобы размещать на доске очередной камень. Когда оба игрока последовательно пропускают ход, игра заканчивается. Перед подсчетом очков игроки отмечают все мертвые камни, т. е. камни, которые не могут образовать группу с двумя глазами или соединиться с другими камнями того же цвета. При подведении итогов мертвые камни считаются эквивалентом захваченных. При возникновении разногласия игроки могут возобновить игру. Но это случается редко: если статус какой-либо группы не вполне очевиден, игроки обычно пытаются прояснить его еще до пропуска хода.

Цель игры состоит в том, чтобы добиться контроля над большей частью доски по сравнению с оппонентом. Существует два способа подсчета очков, но они почти всегда дают один и тот же результат.

Наиболее распространенным методом подсчета очков является *подсчет по территории*. В данном случае вы получаете одно очко за каждый пункт на доске, полностью окруженный вашими камнями, плюс одно очко за каждый захваченный вами камень противника. Победителем является игрок, набравший наибольшее количество очков.

Альтернативным методом является *подсчет по площади*. При этом вы получаете одно очко за каждый пункт территории, плюс одно очко за каждый ваш камень на доске. За исключением редких случаев, оба метода подсчета дают одного и того же победителя: если ни один из игроков не пропускал ход, разница в захваченных камнях будет равна разнице в камнях, оставшихся на доске.

Подсчет по территории чаще используется в повседневных играх, однако для компьютеров несколько более удобным является подсчет по площади. Создаваемый на протяжении всей книги ИИ будет применять именно такой метод, если не указано иное.

Кроме того, игрок белыми получает дополнительные очки в качестве компенсации за то, что он вступает в игру вторым. Эта компенсация называется *коми*. Как правило, она составляет 6,5 балла при подсчете по территории или 7,5 балла при подсчете по площади – дополнительные полбалла гарантируют отсутствие ничейного результата. В этой книге мы используем коми в 7,5 балла.

На рис. 2.4 показано итоговое игровое состояние игры 9×9.

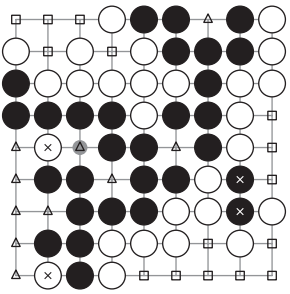


Рис. 2.4 ❖ Итоговое игровое состояние игры 9×9. Мертвые камни отмечены значком ×. Территория черных обозначена треугольниками. Территория белых – квадратами

Вот как подсчитать количество очков в данном случае:

1) камни, помеченные значком ×, считаются мертвыми: они приравниваются к захваченным, несмотря на то что не были захвачены в процессе игры. Игрок черными также осуществил захват ранее в игре (который не показан на рисунке). Таким образом, у игрока черными 3 захваченных камня, а у игрока белыми – 2;
2) игрок черными захватил 12 пунктов территории: 10 пунктов, помеченных треугольником, плюс 2 пункта под мертвыми белыми камнями;
3) игрок белыми захватил 17 пунктов территории: 15 пунктов, помеченных квадратом, плюс 2 пункта под мертвыми черными камнями;
4) после удаления мертвых черных камней на доске остается 27 черных камней;

5) после удаления мертвых белых камней на доске остается 25 белых камней;
6) при подсчете по территории игрок белыми получает 17 пунктов территории + 2 захваченных камня + коми 6,5, что в сумме составляет 25,5 очка. Игрок черными получает 12 пунктов территории + 3 захваченных камня, всего 15 очков;
7) при подсчете по площади игрок белыми получает 17 пунктов территории + 25 камней на доске + коми 7,5, что дает 49,5 очка. Игрок черными получает 12 пунктов территории + 27 камней на доске, что в сумме составляет 39 очков;
8) при любом способе подсчета игрок белыми выигрывает с перевесом в 10,5 очка.

Игра может закончиться еще одним способом: любой игрок может в любой момент выйти из игры. Опытные игроки считают вежливым выходить из игры при явном отставании. Чтобы наш ИИ стал хорошим противником, он должен научиться определять момент, когда ему следует покинуть игру.

2.2.4. Правило ко

Существует еще одно ограничение относительно того, куда вы можете помещать камни. Чтобы игра могла закончиться, ходы, возвращающие доску в предыдущее состояние, считаются незаконными. На рис. 2.5 показан пример того, как это может произойти.

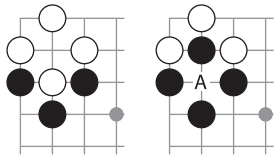


Рис. 2.5 ❖ Иллюстрация правила ко. Сначала игрок черными захватывает один белый камень. Игрок белыми хотел бы совершить ответный захват, поместив камень в пункт A, однако этот ход вернул бы доску в предыдущее состояние. Правило ко запрещает это в целях предотвращения бесконечного цикла поочередных захватов. Сначала игрок белыми должен поместить камень в другом месте доски. Это приведет к изменению игрового состояния, после чего игрок белыми сможет вернуться и осуществить захват, поместив камень в пункт A, при условии что игрок черными не защитит свои камни

На приведенной диаграмме видно, что игрок черными только что захватил один камень. Игроку белыми может захотеться поместить камень в пункт A, чтобы захватить черный камень. Однако это восстановило бы состояние, в котором доска находилась за два хода до этого. Вместо этого белые должны сначала сделать ход в другом месте доски. Это приведет к изменению игрового состояния, после чего игрок белыми сможет осуществить захват, поместив камень в точку A. Конечно, это дает игроку черными возможность защитить уязвимый камень. Чтобы захватить черный камень, игрок белыми должен осуществить отвлекающий маневр в любом другом месте на доске.

Эта ситуация называется *ко*, что в японском языке означает *вечность*. При ее возникновении игроки применяют специальные стратегии, что являлось пробле-

мой для предыдущих поколений программ для игры в го. В главе 7 мы покажем способ, с помощью которого нейронным сетям можно намекнуть на возможность освоения тактики ко. Это обычный метод эффективного обучения нейронных сетей. Даже если вы не можете сформулировать правила, которые должна выучить нейронная сеть, вы можете закодировать свои входные данные таким образом, чтобы подчеркнуть ситуации, на которые хотите обратить ее внимание.

2.3. Фора

Когда играют два игрока разного уровня, более слабый игрок получает право играть черными камнями и сделать несколько ходов. Это называется игрой с форой (гандикапом). Затем более сильный игрок берет белые камни и делает первый ход. Кроме того, в игре с форой коми обычно уменьшается до половины очка. Обычно цель коми заключается в том, чтобы скомпенсировать преимущество, полученное игроком благодаря праву первого хода. Однако смысл гандикапа заключается в том, чтобы дать игроку черными дополнительное преимущество. Оставшаяся половина очка призвана лишь гарантировать отсутствие ничейного результата.

Традиционно камни гандикапа размещаются в звездных пунктах, однако некоторые игроки позволяют противнику выбрать место их размещения.

2.4. Дополнительные ресурсы

Несмотря на то что мы рассмотрели правила, мы даже не прикоснулись к тому, что делает игру го такой увлекательной и захватывающей. Это выходит за рамки данной книги, поэтому мы рекомендуем вам сыграть несколько игр и самостоятельно выяснить подробности. Вот некоторые ресурсы для дальнейшего изучения:

- самый лучший способ познакомиться с го – сыграть в онлайн-игру. Популярный сервер Online Go (**onlinego.com**) позволяет играть прямо в веб-браузере. Даже если вы только что выучили правила, система ранжирования поможет вам найти подходящую игру. Другими популярными серверами для игры в го являются KGS Go Server (**gokgs.com**) и Tygem (**www.tygembaduk.com**);
- Sensei's Library (**senseis.xmp.net**) – справочник, содержащий многочисленные стратегии, советы, историческую справку и прочую информацию;
- серия Дженис Ким *Learn to Play Go* входит в число лучших англоязычных книг, посвященных игре го. Для начинающих мы настоятельно рекомендуем тома 1 и 2, которые быстро помогут вам вникнуть в смысл игры.

2.5. Чему можно научить машину?

Независимо от того, учите вы компьютер играть в го или в крестики-нолики, большинство ИИ для настольных игр имеет похожую общую структуру. В этом разделе мы предоставим общий обзор этой структуры и определим конкретные проблемы, которые должен решить искусственный интеллект. В зависимости от игры лучшие решения могут потребовать использования игровой логики, машинного обучения или и того, и другого.

2.5.1. Выбор ходов в дебюте

В начале игры сложно оценить конкретный ход из-за огромного количества вариантов дальнейшего развития ситуации. ИИ для игры в шахматы и го часто используют дебютную книгу: базу данных последовательностей открывающих ходов, взятых из партий опытных игроков-людей. Для создания такой книги вам потребуется набор записей партий сильных игроков. Игровые записи анализируются на предмет наличия общих позиций. Если в любой такой позиции существует твердое согласие в отношении следующего хода, скажем, 80 % всех последующих ходов приходится на один или два хода, вы вносите эти ходы в дебютную книгу. Затем в ходе игры бот может обратиться к этой книге. Если какая-либо ранняя игровая позиция встречается в книге дебютов, бот просто совершает ход эксперта.

ИИ для игры в шахматы и шашки, в которых по ходу игры фигуры удаляются с доски, содержат аналогичные эндшпильные базы данных: когда на доске остается всего несколько фигур, вы можете заранее рассчитать все варианты. Эта техника неприменима к игре го, в которой доска заполняется ближе к концу партии.

2.5.2. Поиск игровых состояний

В основе ИИ для настольной игры лежит поиск по дереву. Подумайте о том, как люди играют в стратегическую игру. Сначала мы рассматриваем свой возможный следующий ход. Затем думаем о том, как ответит наш оппонент, потом планируем свою реакцию и т. д. Мы просчитываем варианты на максимально возможное количество ходов вперед, а потом оцениваем, насколько хорош полученный результат. Затем возвращаемся на несколько ходов назад и рассматриваем другой вариант на предмет его возможного превосходства.

Это напоминает принцип работы алгоритмов поиска по дереву, используемых в игровом ИИ. Разумеется, люди могут держать в голове только несколько вариантов, в то время как компьютеры могут без проблем манипулировать миллионами. Люди компенсируют недостаток вычислительной мощности интуицией. Опытные шахматисты и игроки в го очень хорошо выявляют те несколько перспективных ходов.

В конечном счете вычислительные мощности одержали победу в шахматах. Однако ИИ для игры в го, способный конкурировать с ведущими игроками, сделал неожиданный поворот – наделил компьютер человеческой интуицией.

2.5.3. Сокращение количества рассматриваемых ходов

В контексте игрового поиска по дереву число возможных ходов на каждом этапе представляет собой коэффициент ветвления.

В шахматах средний коэффициент ветвления составляет около 30. В начале игры у каждого игрока есть 20 разрешенных вариантов первого хода, число которых увеличивается по мере освобождения доски. При таких масштабах вполне вероятно просчитывать каждый возможный ход на четыре или пять ходов вперед, а шахматный движок может сделать гораздо более долгосрочный прогноз многообещающих вариантов.

По сравнению с шахматами, коэффициент ветвления в го просто огромен. Существует 361 вариант первого хода, и это число уменьшается очень медленно. Средний коэффициент ветвления составляет около 250 на каждом ходу. Чтобы

просчитать игру всего на четыре хода вперед, нужно оценить почти 4 млрд позиций. Такое количество возможностей очень важно сократить. В табл. 2.1 на примере шахмат и го показано, как коэффициент ветвления влияет на количество возможных игровых состояний.

Таблица 2.1. Приблизительное количество возможных состояний доски в играх

	Коэффициент ветвления 30 (шахматы)	**Коэффициент ветвления 250 (го)**
Через два хода	900	62 500
Через три хода	27 000	15 млн
Через четыре хода	810 000	4 млрд
Через пять ходов	24 млн	1 трлн

В го основанные на правилах методы выбора хода оказываются не очень эффективными, поскольку крайне сложно прописать правила, которые позволяли бы надежно определять наиболее важную область доски. Однако глубокое обучение идеально подходит для решения этой задачи. Вы можете применить обучение с учителем, чтобы научить компьютер имитировать действия игрока в го.

Вы начинаете с большой коллекции записей партий, сыгранных сильными игроками-людьми. Отличным ресурсом в данном случае являются игровые серверы. Затем вы воспроизводите все эти игры на компьютере, извлекая каждое состояние доски и следующий ход. Это ваш набор обучающих данных. С помощью достаточно глубокой нейронной сети можно прогнозировать ход человека с точностью более 50 %. Вы можете создать бота, который просто совершает предсказанный человеческий ход, и он уже будет являться достойным противником. Однако настоящая мощь заключается в комбинировании этих предсказаний с поиском по дереву: предсказанные ходы предоставляют вам ранжированный список ветвей для исследования.

2.5.4. Оценка игровых состояний

Коэффициент ветвления ограничивает количество ходов, которые может просчитать ИИ. Если бы вы могли просчитать гипотетическую последовательность ходов до конца игры, вы бы знали, кто победит. Определить, насколько хороша эта последовательность, очень легко. Однако это не практично в любой игре сложнее, чем крестики-нолики, поскольку количество возможных вариантов слишком велико. В какой-то момент вам придется остановиться и выбрать одну из рассмотренных неполных последовательностей. Для этого вы берете просчитанное окончательное состояние доски и присваиваете ему числовую оценку. Из всех проанализированных вариантов вы выбираете ход, который приводит к лучшему состоянию доски. Самое сложное – это вычислить оценку состояния доски.

В ИИ для игры в шахматы оценка состояния основана на логике, которая имеет смысл для шахматистов. Вы можете начать с простых правил вроде этого: если ты съешь мою пешку, а я съем твою ладью, то это хорошо для меня. Лучшие шахматные движки выходят далеко за рамки этого, учитывая сложные правила для определения итогового положения фигур на доске и выявления других фигур, блокирующих их движение.

В случае с игрой в го оценка игрового состояния может оказаться более сложным делом, чем выбор хода. Цель игры состоит в том, чтобы захватить как можно больше территории, однако территорию на удивление сложно подсчитать: границы, как правило, остаются расплывчатыми вплоть до поздних этапов игры. Подсчет захваченных камней тоже мало помогает, иногда вы можете дойти до конца игры, захватив всего пару камней. Это еще одна область, где преимущество имеет человеческая интуиция.

Глубокое обучение также позволило совершить в этой области мощный прорыв. Виды нейронных сетей, подходящие для выбора хода, тоже можно обучить для оценки состояния доски. Вместо того чтобы учить нейронную сеть предсказывать следующий ход, вы учите ее предсказывать победителя. Вы можете спроектировать сеть так, чтобы она отражала свой прогноз в виде вероятности, что предоставляет вам числовую оценку для состояния доски.

2.6. Определение силы ИИ для игры в го

В процессе работы над ИИ для игры в го вам, естественно, захочется узнать, насколько он силен. Большинству игроков в го знакома традиционная японская система ранжирования, поэтому измерять силу своего бота вы будете именно по этой шкале. Единственным способом определения его уровня является игра с противниками – другими ИИ или игроками-людьми.

2.6.1. Традиционные ранги го

Игроки в го обычно используют традиционную японскую систему, в которой игрокам присуждаются ранги кю (начинающий) или дан (эксперт). Уровни дан, в свою очередь, делятся на любительские и профессиональные. Самым высоким рангом кю является 1 кю, большие числа соответствуют более низким рангам. Ранги дан идут в противоположном направлении: 1 дан на один уровень сильнее, чем 1 кю, и каждый последующий дан соответствует все более высокому рангу. Для любителей самым высоким рангом традиционно является 7 дан. Игроки-любители могут получить рейтинг в своей региональной ассоциации го, онлайн-серверы также отслеживают рейтинг своих игроков. Система ранжирования приведена в табл. 2.2.

Таблица 2.2. Традиционные ранги го

25 кю	Новички, только что ознакомленные с правилами
20 кю – 11 кю	Новички
10 кю – 1 кю	Игроки среднего уровня
1 дан и выше	Сильные игроки-любители
7 дан	Игроки-любители, сопоставимые по силе с профессионалами
Профессиональная шкала 1 дан – 9 дан	Самые сильные игроки в мире

Любительские ранги зависят от количества камней гандикапа, необходимых для компенсации разницы в силе между двумя игроками. Например, если ранг Алисы 2 кю, а ранг Боба 5 кю, Алиса предоставит Бобу фору в три камня, чтобы уравновесить их шансы на победу.

Профессиональная шкала устроена несколько иначе: ее ранги больше похожи на звания. Региональная ассоциация го присуждает лучшим игрокам профессиональное звание по результатам крупных турниров, которое сохраняется на всю жизнь. Любительские и профессиональные шкалы не имеют прямого соответствия, однако вы можете с уверенностью предположить, что любой игрок профессионального уровня, по крайней мере, так же силен, как любитель 7 дана. Лучшие профессионалы являются гораздо более сильными.

2.6.2. Сравнительный анализ вашего ИИ для игры в го

Простым способом оценки силы ваших ботов является игра против других ботов известного уровня. Для этого можно использовать такие движки го с открытым исходным кодом, как GNU Go и Pachi. GNU Go играет на уровне около 5 кю, а Pachi – около 1 дана (уровень Pachi немного варьируется в зависимости от количества предоставленной вычислительной мощности). Таким образом, если ваш бот играет с GNU Go 100 партий и выигрывает около 50 раз, вы можете сделать вывод о том, что уровень вашего бота также соответствует примерно 5 кю.

Для более точного определения ранга вы можете разместить своего бота на общедоступном сервере для игры в го с системой ранжирования. Для получения разумной оценки будет достаточно нескольких десятков партий.

2.7. Резюме

- Игры являются популярным предметом для исследований в области ИИ, поскольку позволяют создавать контролируемые среды с известными правилами.
- Сегодня самые сильные ИИ для игры в го полагаются на машинное обучение, а не на специфически игровые знания. Основанные на правилах ИИ для игры в го оказались неэффективными отчасти из-за огромного количества возможных вариантов, требующих рассмотрения.
- В случае с игрой го вы можете применить глубокое обучение для выбора хода и для оценки состояния.
- Выбор ходов предполагает сокращение количества ходов, которые необходимо рассмотреть при конкретном состоянии доски. Без эффективного метода выбора хода ваш ИИ для игры в го будет вынужден просчитывать слишком много веток.
- Оценка состояния сводится к проблеме определения того, какой игрок опережает соперника и на сколько. Без эффективного способа оценки состояния ваш ИИ для игры в го не сможет выбрать хороший вариант.
- Вы можете оценить мощность своего ИИ, сыграв против доступных ботов известного уровня, например GNU Go или Pachi.

Глава 3

Реализация первого бота для игры в го

В этой главе:

- реализация доски для игры в го средствами языка Python;
- размещение последовательностей камней и симуляция игры;
- кодирование правил игры го, гарантирующих совершение допустимых ходов;
- создание простого бота, способного играть против собственной копии;
- полноценная игра с созданным ботом.

В этой главе мы создадим гибкую библиотеку структур данных для представления игры го и алгоритмов, обеспечивающих соблюдение ее правил. Как было показано в предыдущей главе, эти правила довольно просты, однако для их реализации на компьютере необходимо тщательно рассмотреть все крайние случаи. Если вы новичок и незнакомы с правилами игры го, обязательно прочтите главу 2. Данная глава носит технический характер и требует хорошего практического знания правил игры для понимания всех нюансов.

Представление правил игры в го имеет большое значение, поскольку лежит в основе создания умных ботов. Ваш бот должен понимать разницу между допустимыми и недопустимыми ходами, прежде чем вы начнете учить его отличать хорошие ходы от плохих.

В этой главе вы реализуете своего первого бота для игры в го. Несмотря на изначальную примитивность, он будет обладать всеми знаниями об игре, которые ему потребуются для дальнейшего совершенствования.

Мы начнем с формального представления доски и таких основополагающих концепций игры в го, как игрок, камень и ход. Затем обсудим аспекты игрового процесса. Как компьютеру быстро выбрать камни для захвата и определить ситуацию, в которой применяется правило ко? Когда и как заканчивается игра? В данной главе приведены ответы на все эти вопросы.

3.1. Представление игры го средствами языка Python

Доска для игры в го имеет форму квадрата. Как правило, новички используют доски 9×9 или 13×13, а продвинутые и профессиональные игроки – доски 19×19. В принципе, в го можно играть на доске любого размера. Реализовать квадратную сетку для игры довольно просто, однако при этом потребуется учесть множество нюансов.

Для представления игры в го средствами языка Python мы шаг за шагом создадим модуль под названием *dlgo*. На протяжении всей этой главы вы будете создавать файлы, а также реализовывать классы и функции, результатом чего станет ваш первый бот. Весь код, приведенный в этой и следующих главах, можно найти на сайте GitHub по адресу **mng.bz/gYPe**.

Вам, безусловно, следует скопировать содержимое данного репозитория для справки, однако мы настоятельно рекомендуем создавать файлы самостоятельно, наблюдая за процессом постепенного роста библиотеки. Основная ветка нашего GitHub-репозитория содержит весь приведенный в этой книге код, и не только. Для каждой главы, начиная с этой, дополнительно предусмотрены отдельные ветки Git-репозитория, содержащие только код, имеющий отношение к соответствующей главе. Например, код для третьей главы можно найти в ветке *chapter_3*, код для четвертой – в ветке *chapter_4* и т. д. Обратите внимание на то, что GitHub-репозиторий также включает обширные тесты для большей части кода, приведенного в этой и следующих главах.

Чтобы создать библиотеку Python для представления игры го, потребуется достаточно гибкая модель данных, поддерживающая несколько вариантов действий:

- отслеживание хода игры против оппонента-человека;
- отслеживание хода игры против другого бота. Может показаться, что этот пункт совпадает с предыдущим, но на самом деле они слегка различаются. В частности, примитивные боты с трудом распознают момент окончания игры. Игра двух простых ботов друг против друга – это важная техника, применяемая в следующих главах, поэтому о ней стоит упомянуть здесь;
- сравнение множества возможных последовательностей ходов для одного и того же состояния доски;
- импорт записей партий и их преобразование в обучающие данные.

Мы начнем с нескольких простых концепций, таких как игрок и ход, которые позволят подойти к решению задач, приведенных в следующих главах.

Для начала создайте новую папку, *dlgo*, и поместите в нее пустой файл *__init__.py*, который будет выступать в качестве модуля Python. Также создайте два дополнительных файла с именами *gotypes.py* и *goboard_slow.py*, которые будут содержать всю функциональность, относящуюся к игровому процессу и доске. На данный момент содержимое папки должно выглядеть следующим образом:

```
dlgo
    __init__.py
    gotypes.py
    goboard_slow.py
```

В игре го игроки черными и белыми камнями ходят по очереди. Для представления разноцветных камней мы будем использовать модуль `enum`. Игрок (`Player`) может играть либо черными (`black`), либо белыми (`white`) камнями. После того как игрок поместил камень на доску, цвет можно переключить, вызвав метод `other` экземпляра `Player`. Добавьте класс `Player` в файл *gotypes.py*.

Листинг 3.1 ❖ Использование модуля `enum` для представления игроков

```
import enum

class Player(enum.Enum):
    black = 1
```

```
    white = 2

    @property
    def other(self):
        return Player.black if self == Player.white else Player.white
```

Как уже говорилось, в этой книге мы используем версию Python 3. Одна из причин этого заключается в том, что многие современные аспекты языка, такие как перечисления (`enum`) в файле *gotypes.py*, являются частью стандартной библиотеки Python 3.

Координаты доски будут представлены с помощью кортежей. Добавьте класс `Point` в файл *gotypes.py*.

Листинг 3.2 ❖ Использование кортежей для представления точек на доске для игры в го

```
from collections import namedtuple

class Point(namedtuple('Point', 'row col')):
    def neighbors(self):
        return [
            Point(self.row - 1, self.col),
            Point(self.row + 1, self.col),
            Point(self.row, self.col - 1),
            Point(self.row, self.col + 1),
        ]
```

Именованный кортеж `namedtuple` позволяет получить доступ к координатам в виде `point.row` и `point.col` вместо `point[0]` и `point[1]`, что облегчает чтение кода.

Нам также требуется структура для представления возможных действий игрока. Обычно ход предполагает размещение камня на доске, однако игрок на любом этапе может пропустить ход или выйти из игры. В соответствии с соглашениями, принятыми Американской ассоциацией го (American Go Association, AGA), под термином «ход» (move) мы будем подразумевать любое из этих трех действий, а под термином «игра» (play) – размещение камня на доске. Таким образом, в классе `Move` мы закодируем все три типа ходов – play (игра), pass (пас), resign (выход из игры), а также убедимся в том, что конкретный ход относится к одному из них. Для размещения камня на доске нужно передать координату `Point`. Добавьте класс `Move` в файл *goboard_slow.py*.

Листинг 3.3 ❖ Задание типов ходов: игра, пас и выход из игры

```
import copy
from dlgo.gotypes import Player

class Move():
    def __init__(self, point=None, is_pass=False, is_resign=False):
        assert (point is not None) ^ is_pass ^ is_resign
        self.point = point
        self.is_play = (self.point is not None)
        self.is_pass = is_pass
        self.is_resign = is_resign

    @classmethod
    def play(cls, point):
        return Move(point=point)
```

Задаются возможные действия игрока – `is_play`, `is_pass` или `is_resign`.

Этот ход предполагает размещение камня на доске.

```
@classmethod
def pass_turn(cls):                    <-- Этот ход предполагает пропуск хода.
    return Move(is_pass=True)

@classmethod
def resign(cls):                       <-- Этот ход предполагает выход из игры.
    return Move(is_resign=True)
```

Как правило, в дальнейшем клиенты не будут вызывать конструктор `Move` напрямую. Вместо этого обычно будет вызываться `Move.play`, `Move.pass_turn` или `Move.resign` с целью создания экземпляра соответствующего класса.

Обратите внимание на то, что до сих пор классы `Player`, `Point` и `Move` представляли собой простые типы данных. Несмотря на то что они имеют основополагающее значение для представления доски, они не содержат никакой игровой логики. Это сделано специально, и такое разделение аспектов игрового процесса имеет свои преимущества.

Далее мы реализуем классы, способные обновлять игровое состояние с помощью трех классов, созданных ранее:

- класс `Board` отвечает за логику размещения и захвата камней;
- класс `GameState` учитывает все камни на доске и предыдущее игровое состояние, а также отслеживает очередность ходов.

3.1.1. Реализация доски для игры в го

Прежде чем перейти к `GameState`, давайте реализуем класс `Board`. Для начала можно создать массив 19×19, отслеживающий состояние каждой точки доски. Теперь подумайте об алгоритме, позволяющем определить момент, когда камни необходимо убрать с доски. Напомним, что количество степеней свободы конкретного камня определяется количеством пустых точек в его непосредственной близости. Если все четыре соседние точки заняты вражескими камнями, то данный камень не имеет ни одной степени свободы и подлежит захвату. В случае больших групп связанных камней выполнить такую проверку сложнее. Например, после размещения черного камня нужно проверить все соседние белые камни на предмет их возможного захвата. В частности, необходимо проверить следующее:

1) остались ли у соседей степени свободы;
2) остались ли степени свободы у соседей соседей;
3) остались ли степени свободы у соседей соседей соседей и т. д.

Этот процесс может включать сотни этапов. Представьте длинную цепь камней на доске, протянувшуюся по территории противника в результате 200 сделанных ходов. Для ускорения проверки можно отслеживать группы связанных между собой камней.

3.1.2. Отслеживание связанных групп камней в игре го

В предыдущем разделе говорилось о том, что проверка отдельных камней может увеличить вычислительную сложность. Вместо этого можно одновременно отслеживать группы связанных камней одного цвета *и их степени свободы*. Этот подход оказывается намного более эффективным при реализации игровой логики.

Группа связанных камней одного цвета называется *цепочкой камней* или просто *цепочкой*, как показано на рис. 3.1. Эту структуру можно создать с помощью типа

set языка Python, как в следующей реализации класса GoString, которую также необходимо добавить в файл *goboard_slow.py*.

Листинг 3.4 ❖ Кодирование цепочек камней с помощью типа set

```
class GoString():
    def __init__(self, color, stones, liberties):
        self.color = color
        self.stones = set(stones)
        self.liberties = set(liberties)

    def remove_liberty(self, point):
        self.liberties.remove(point)

    def add_liberty(self, point):
        self.liberties.add(point)

    def merged_with(self, go_string):
        assert go_string.color == self.color
        combined_stones = self.stones | go_string.stones
        return GoString(
            self.color,
            combined_stones,
            (self.liberties | go_string.liberties) - combined_stones)

    @property
    def num_liberties(self):
        return len(self.liberties)

    def __eq__(self, other):
        return isinstance(other, GoString) and \
            self.color == other.color and \
            self.stones == other.stones and \
            self.liberties == other.liberties
```

GoString – это цепочка связанных камней одного цвета.

Возвращает новую цепочку, содержащую все камни обеих цепочек.

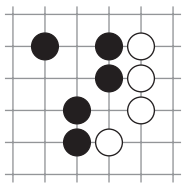


Рис. 3.1 ❖ В данном случае на доске находятся три цепочки черных камней и две цепочки белых. Большая цепочка белых камней имеет шесть степеней свободы, а один белый камень – только три

Обратите внимание на то, что GoString напрямую отслеживает свои собственные степени свободы, и мы в любой момент можем получить доступ к их количеству, вызвав num_liberties, что гораздо эффективнее по сравнению с описанным ранее примитивным подходом, предполагающим работу с отдельными камнями.

Также у нас есть возможность добавлять и удалять степени свободы конкретной цепочки с помощью remove_liberty и add_liberty. Количество степеней свободы цепочки обычно уменьшается, когда противник размещает камень рядом с ней,

и увеличивается, когда эта или другая группа захватывает камни противника, расположенные по соседству.

Кроме того, обратите внимание на метод `merged_with` класса `GoString`, который вызывается, когда игрок соединяет две свои группы путем размещения камня.

3.1.3. Размещение и захват камней на доске для игры в го

Теперь, когда мы обсудили камни и цепочки камней, пришло время поговорить об их размещении на доске. Алгоритм предполагает использование класса `GoString` из листинга 3.4 и состоит из трех этапов:

1) объединение всех смежных цепочек камней одного цвета;
2) уменьшение количества степеней свободы соседних цепочек камней противоположного цвета;
3) удаление с доски цепочек камней противоположного цвета с нулевой степенью свободы.

Кроме того, если вновь созданная цепочка имеет нулевую степень свободы, ход отклоняется. Это ведет к следующей реализации класса `Board`, который также необходимо добавить в файл *goboard_slow.py*. Мы можем получить доску, состоящую из любого количества строк и столбцов, создав ее экземпляр с помощью `num_rows` и `num_cols` соответственно. Для внутреннего отслеживания состояния доски применяется приватная переменная `_grid`, словарь, в котором хранятся цепочки камней. Для начала мы инициализируем экземпляр доски для игры в го, указав ее размер.

Листинг 3.5 ❖ Создание экземпляра доски для игры в го

```
class Board():
    def __init__(self, num_rows, num_cols):
        self.num_rows = num_rows
        self.num_cols = num_cols
        self._grid = {}
```

Доска инициализируется в виде пустой сетки, состоящей из заданного количества строк и столбцов.

Далее мы обсудим метод `Board` для размещения камней. В `place_stone` сначала определяется количество степеней свободы всех камней, соседствующих с данной точкой.

Листинг 3.6 ❖ Проверка количества степеней свободы соседних точек

```
    def place_stone(self, player, point):
        assert self.is_on_grid(point)
        assert self._grid.get(point) is None
        adjacent_same_color = []
        adjacent_opposite_color = []
        liberties = []
        for neighbor in point.neighbors():
            if not self.is_on_grid(neighbor):
                continue
            neighbor_string = self._grid.get(neighbor)
            if neighbor_string is None:
                liberties.append(neighbor)
            elif neighbor_string.color == player:
                if neighbor_string not in adjacent_same_color:
                    adjacent_same_color.append(neighbor_string)
```

Сначала исследуются непосредственные соседи конкретной точки.

```
        else:
            if neighbor_string not in adjacent_opposite_color:
                adjacent_opposite_color.append(neighbor_string)
    new_string = GoString(player, [point], liberties)
```

Обратите внимание на то, что в первых двух строках листинга 3.6 используются служебные методы, для того чтобы проверить, находится ли точка в пределах границ данной доски и не был ли на ней размещен камень. Эти два метода определены следующим образом.

Листинг 3.7 ❖ Служебные методы для размещения и удаления камней

```
def is_on_grid(self, point):
    return 1 <= point.row <= self.num_rows and \
        1 <= point.col <= self.num_cols

def get(self, point):
    string = self._grid.get(point)
    if string is None:
        return None
    return string.color

def get_go_string(self, point):
    string = self._grid.get(point)
    if string is None:
        return None
    return string
```

Возвращает содержимое точки на доске: сведения об игроке, если в этой точке находится камень, в противном случае – None.

Возвращает всю цепочку камней, если в этой точке находится камень, в противном случае – None.

Мы также определяем `get_go_string` для возврата цепочки камней, связанной с данной точкой. Эта функция полезна сама по себе, но представляет особую ценность для предотвращения *самозахвата*, о котором мы подробно поговорим в разделе 3.2.

Теперь продолжим определение `place_stone`, начатое в листинге 3.6. Сразу после определения `new_string` последует описанный далее трехэтапный процесс.

Листинг 3.8 ❖ Продолжение определения `place_stone`

```
    for same_color_string in adjacent_same_color:
        new_string = new_string.merged_with(same_color_string)
    for new_string_point in new_string.stones:
        self._grid[new_string_point] = new_string
    for other_color_string in adjacent_opposite_color:
        other_color_string.remove_liberty(point)
    for other_color_string in adjacent_opposite_color:
        if other_color_string.num_liberties == 0:
            self._remove_string(other_color_string)
```

Объединение всех смежных цепочек камней одного цвета.

Уменьшение количества степеней свободы соседних цепочек камней противоположного цвета.

Удаление с доски цепочек камней противоположного цвета с нулевой степенью свободы.

Теперь в определении игровой доски не хватает лишь способа удаления камней, как этого требует `remove_string` в последней строке листинга 3.8. Как видно из листинга 3.9, обеспечить это довольно просто, однако необходимо помнить о том, что при удалении вражеских камней другие камни могут *получить* дополнительные степени свободы. Например, на рис. 3.2 видно, что игрок черными может захватить белый камень, получив при этом дополнительную степень свободы для каждой цепочки черных камней.

Листинг 3.9 ❖ Продолжение определения `place_stone`

```
    def _remove_string(self, string):
        for point in string.stones:
            for neighbor in point.neighbors():
                neighbor_string = self._grid.get(neighbor)
                if neighbor_string is None:
                    continue
                if neighbor_string is not string:
                    neighbor_string.add_liberty(point)
            self._grid[point] = None
```

Удаление цепочки может привести к увеличению степеней свободы других цепочек.

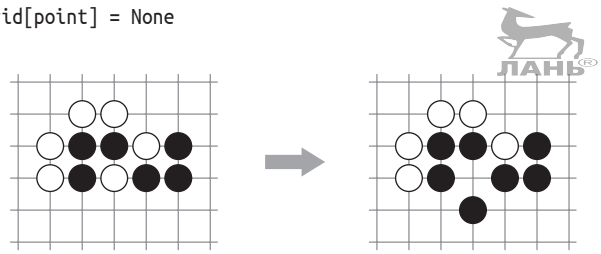


Рис. 3.2 ❖ Игрок черными может захватить белый камень, тем самым увеличив количество степеней свободы каждой из цепочек, прилегающих к захваченному камню

Это определение завершает реализацию игровой доски.

3.2. Фиксация игрового состояния и проверка допустимости ходов

Теперь, когда мы определили правила размещения и захвата камней, займемся фиксацией текущего игрового состояния с помощью класса `GameState`. Грубо говоря, *игровое состояние* учитывает позицию на доске, очередь игрока, предыдущее игровое состояние и последний сделанный ход. Далее приведено лишь начало определения класса `GameState`. По ходу чтения данного раздела вы будете расширять его функциональность, добавляя все новые фрагменты кода в файл *goboard_slow.py*.

Листинг 3.10 ❖ Кодирование игрового состояния

```
class GameState():
    def __init__(self, board, next_player, previous, move):
        self.board = board
        self.next_player = next_player
        self.previous_state = previous
        self.last_move = move

    def apply_move(self, move):
        if move.is_play:
            next_board = copy.deepcopy(self.board)
            next_board.place_stone(self.next_player, move.point)
        else:
            next_board = self.board
```

Возвращает новое игровое состояние после совершения хода.

```
        return GameState(next_board, self.next_player.other, self, move)

    @classmethod
    def new_game(cls, board_size):
        if isinstance(board_size, int):
            board_size = (board_size, board_size)
        board = Board(*board_size)
        return GameState(board, Player.black, None, None)
```

На данном этапе мы уже можем определить момент окончания игры, добавив следующий фрагмент кода в класс `GameState`.

Листинг 3.11 ❖ Определение момента окончания игры

```
    def is_over(self):
        if self.last_move is None:
            return False
        if self.last_move.is_resign:
            return True
        second_last_move = self.previous_state.last_move
        if second_last_move is None:
            return False
        return self.last_move.is_pass and second_last_move.is_pass
```

Теперь, когда мы обеспечили возможность применения хода к текущему игровому состоянию с помощью `apply_move`, необходимо написать код для проверки допустимости ходов. Люди могут попытаться совершить недопустимый ход случайно. Боты могут сделать это из-за отсутствия лучшей альтернативы. Чтобы это предотвратить, необходимо сделать следующее:

- убедиться в том, что точка, в которую вы собираетесь поместить камень, пуста;
- убедиться в том, что результатом хода не станет самозахват;
- убедиться в том, что ход не нарушает правило ко.

Первый пункт реализовать довольно легко, а два других заслуживают отдельного обсуждения.

3.2.1. Самозахват

Когда при размещении камня у цепочки ваших камней исчезает последняя степень свободы, это называется *самозахватом*. Например, черные камни, изображенные на рис. 3.3, обречены.

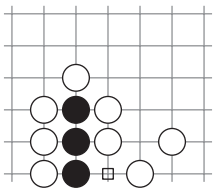


Рис. 3.3 ❖ При данном игровом состоянии у цепочки из трех черных камней есть лишь одна степень свободы, отмеченная квадратом. Применение правила самозахвата не позволяет черным поместить камень в эту точку. При этом игрок белыми может поместить в нее свой камень, захватив три черных камня

Игрок белыми может захватить их в любое время, поместив камень в отмеченную точку, и у его противника нет никакого способа это предотвратить. Но что, если игрок черными поместит в эту точку свой камень? Эта группа камней лишится последней степени свободы и будет захвачена. Большинство сводов правил запрещает такую игру, однако существует несколько исключений. В частности, самозахват разрешен правилами турнира «Кубок Инга», который проводится раз в четыре года и отличается одним из самых крупных призов среди международных соревнований по игре в го.

Мы обеспечим выполнение правила самозахвата в своем коде. Это согласуется с наиболее популярными сводами правил, а также уменьшает количество ходов, рассматриваемых ботами. Теоретически можно представить ситуацию, в которой самозахват будет являться наилучшим ходом, однако в серьезных играх такие случаи практически не встречаются.

Если слегка изменить расположение камней, изображенных на рис. 3.3, то ситуация станет совершенно иной (см. рис. 3.4).

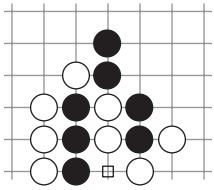


Рис. 3.4 ❖ В данном случае размещение черного камня в точке, отмеченной квадратом, приведет к захвату двух белых камней и приобретению двух степеней свободы

Учтите, что перед проверкой степеней свободы вновь размещенного на доске камня необходимо удалить камни противника. Во всех сводах правил следующий ход является допустимым захватом, а не самозахватом, поскольку захват белых камней ведет к увеличению количества степеней свободы черных камней.

Обратите внимание на то, что класс `Board` допускает самозахват, однако в `GameState` мы обеспечим выполнение этого правила, применив ход к копии доски и проверив получившееся в результате количество степеней свободы.

Листинг 3.12 ❖ Добавление правила самозахвата в определение класса `GameState`

```
def is_move_self_capture(self, player, move):
    if not move.is_play:
        return False
    next_board = copy.deepcopy(self.board)
    next_board.place_stone(player, move.point)
    new_string = next_board.get_go_string(move.point)
    return new_string.num_liberties == 0
```

3.2.2. Правило ко

После выполнения проверки на предмет самозахвата можно приступать к реализации правила ко, значение которого в игре го кратко описано в главе 2. Правило ко применяется в том случае, если ход может привести к восстановлению преды-

дущего состояния доски. Это не означает, что игрок не может сразу же нанести ответный удар, как показано далее. На рис. 3.5 игрок белыми только что разместил одиночный камень внизу. Теперь у двух черных камней есть только одна степень свободы, как и у белого камня.

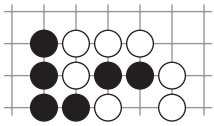


Рис. 3.5 ❖ В данной ситуации игрок белыми хочет захватить два черных камня, однако у белого камня осталась лишь одна степень свободы

Игрок черными может попытаться спасти два своих камня, захватив белый камень, как показано на рис. 3.6.

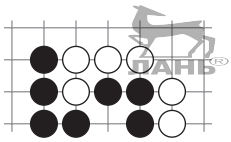


Рис. 3.6 ❖ Игрок черными пытается спасти два своих камня, захватив изолированный белый камень

Однако игрок белыми может сразу поместить камень *в ту же точку*, что и на рис. 3.5, как показано на рис. 3.7.

Как видите, белые могут захватить три черных камня, и правило ко не будет применено, поскольку состояния доски на рис. 3.5 и 3.7 не являются одинаковыми. Этот прием называется «защелка». В простых ситуациях правило ко сводится к запрещению немедленного повторного захвата камня. Однако прием «защелка» используется довольно часто, и существование таких позиций говорит о необходимости проявлять осторожность при реализации правила ко.

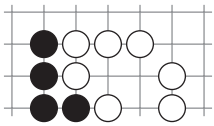


Рис. 3.7 ❖ В данной ситуации игрок белыми может воспользоваться приемом «защелка» (захватить три черных камня), не нарушая правило ко

Правило ко можно реализовать многими способами, которые, за редким исключением, являются практически эквивалентными. В своем коде мы реализуем правило, запрещающее игроку совершать ход, воссоздающий предыдущее игро-

вое состояние, при этом игровое состояние учитывает как расположение камней на доске, так и очередь игрока. Эта конкретная формулировка правила известна как *ситуационное суперко*.

Поскольку каждый экземпляр класса `GameState` хранит указатель на предыдущее игровое состояние, мы можем обеспечить выполнение правила ко, совершив возврат по дереву и сверив новое состояние со всей историей. Для этого необходимо добавить следующий метод в нашу реализацию `GameState`:

Листинг 3.13 ❖ Нарушает ли текущее состояние игры правило ко?

```
    @property
    def situation(self):
        return (self.next_player, self.board)

    def does_move_violate_ko(self, player, move):
        if not move.is_play:
            return False
        next_board = copy.deepcopy(self.board)
        next_board.place_stone(player, move.point)
        next_situation = (player.other, next_board)
        past_state = self.previous_state
        while past_state is not None:
            if past_state.situation == next_situation:
                return True
            past_state = past_state.previous_state
        return False
```

Эта реализация является простой и корректной, но довольно медленной. Для каждого хода необходимо создавать глубокую копию состояния доски и сравнивать это состояние со всеми предыдущими состояниями, которых со временем становится все больше. В разделе 3.5 мы поговорим об интересном способе ускорения этого процесса.

Для завершения определения класса `GameState` можно проверить ход на предмет его допустимости, используя сведения из раздела 3.2 о правиле ко и самозахвате.

Листинг 3.14 ❖ Является ли этот ход допустимым для данного игрового состояния?

```
    def is_valid_move(self, move):
        if self.is_over():
            return False
        if move.is_pass or move.is_resign:
            return True
        return (
            self.board.get(move.point) is None and
            not self.is_move_self_capture(self.next_player, move) and
            not self.does_move_violate_ko(self.next_player, move))
```

3.3. Завершение игры

Ключевым понятием в компьютерной игре го является *игра с самим собой*. Как правило, все начинается со слабого агента, играющего в го против самого себя, а полученные результаты используются для создания более сильного бота. В гла-

ве 4 игра с самим собой будет использована для оценки состояний доски. В главах с 9 по 12 с ее помощью будут оцениваться отдельные ходы и выбравшие их алгоритмы.

Чтобы получить пользу от применения этой техники, необходимо сделать так, чтобы игра с самим собой могла закончиться. Игры людей заканчиваются, когда у игроков не остается ходов, способных обеспечить их преимущество. Эта концепция может показаться сложной даже человеку. Новички часто заканчивают партии, делая безнадежные ходы на территории противника или наблюдая за тем, как их противник прорывается на территорию, казавшуюся надежно защищенной. С компьютерами дело обстоит еще сложнее. Если наш бот будет продолжать играть до тех пор, пока у него остаются допустимые ходы, он в конечном итоге заполнит свои собственные степени свободы и потеряет все свои камни.

Чтобы помочь боту завершить игру, можно применить эвристические методы. Например:

- не размещать камень в области, полностью окруженной камнями одного цвета;
- не размещать камень в точке, имеющей только одну степень свободы;
- всегда захватывать камень противника с единственной степенью свободы.

К сожалению, *все эти правила являются слишком строгими*. Если бы наши боты следовали им, то сильные противники могли бы убивать группы камней, которые должны жить, спасать группы, которые должны умереть, или просто получили бы преимущество. Вообще, наши эвристические методы должны как можно меньше ограничивать возможности бота, чтобы более сложные алгоритмы могли свободно осваивать продвинутые тактики.

Для решения этой проблемы можно обратиться к истории игры. В древние времена победителем считался игрок с наибольшим количеством камней на доске. Игроки старались заполнить все возможное пространство, оставляя пустыми только глаза групп своих камней. При таком подходе партия может затянуться, поэтому игроки придумали способ ее ускорения. Если игрок черными контролирует область доски (т. е. любой размещенный там белый камень в конечном итоге захватывается), то ему не нужно заполнять эту область камнями. Игроки просто соглашаются с тем, что эта область принадлежит черным. Отсюда и возникло понятие территории, и со временем правила игры начали предусматривать ее явный учет.

Такой способ подсчета очков позволяет избежать вопроса о том, что является территорией, а что – нет, однако нам по-прежнему необходимо каким-то образом помешать боту убивать собственные камни (см. рис. 3.8).

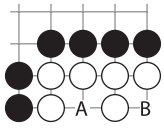


Рис. 3.8 ❖ Группа белых камней имеет два глаза в точках A и B, поэтому игроку белыми не нужно помещать камень ни в одну из них. В противном случае черные смогут захватить всю группу. Наивному боту нельзя разрешать заполнять глаза собственной группы камней

Мы жестко закодируем правило, строго запрещающее боту заполнять глаза собственных групп камней. Для наших целей мы будем считать *глазом* пустую точку, у которой все соседние точки и как минимум три из четырех точек, расположенных по диагонали, заполнены дружественными камнями.

Опытные игроки в го могут заметить, что приведенное выше определение *глаза* в некоторых случаях может не позволить учесть действительный глаз. Мы смиримся с этими ошибками для упрощения реализации.

Для глаз на краю доски необходимо предусмотреть специальный случай, при котором все соседние точки, расположенные по диагонали, должны содержать дружественные камни.

Создайте новый подмодуль *dlgo* под названием *agent* (т. е. создайте новую папку с именем *agent* и поместите в нее пустой файл *__init__.py*). Добавьте следующую функцию `is_point_an_eye` в файл с именем *helpers.py*.

Листинг 3.15 ❖ Является ли глазом данная точка доски?

```
from dlgo.gotypes import Point

def is_point_an_eye(board, point, color):
    if board.get(point) is not None:
        return False
    for neighbor in point.neighbors():
        if board.is_on_grid(neighbor):
            neighbor_color = board.get(neighbor)
            if neighbor_color != color:
                return False

    friendly_corners = 0
    off_board_corners = 0
    corners = [
        Point(point.row - 1, point.col - 1),
        Point(point.row - 1, point.col + 1),
        Point(point.row + 1, point.col - 1),
        Point(point.row + 1, point.col + 1),
    ]
    for corner in corners:
        if board.is_on_grid(corner):
            corner_color = board.get(corner)
            if corner_color == color:
                friendly_corners += 1
        else:
            off_board_corners += 1
    if off_board_corners > 0:
        return off_board_corners + friendly_corners == 4
    return friendly_corners >= 3
```

Глаз – это пустая точка.

Все смежные точки должны содержать дружественные камни.

Мы должны контролировать три из четырех углов, если точка находится в середине доски. Если она находится на краю доски, то мы должны контролировать все углы.

Точка находится на краю или в углу доски.

Точка находится в середине доски.

В этой главе не обсуждается способ определения результата игры, однако подсчет очков, безусловно, является важной темой. В разных турнирах и федерациях го применяются слегка различающиеся между собой своды правил. Боты, описанные в этой книге, будут следовать правилам AGA для выполнения *подсчета захваченной площади*, который также называется *китайским подсчетом*. Несмот-

ря на то что *японские правила* чаще используются в ходе обычных игр, правила AGA являются чуть более удобными в плане реализации с помощью компьютера, а различия в системах подсчета редко влияют на результат игры.

3.4. Создание первого слабого бота для игры в го

После реализации доски и кодирования игрового состояния можно приступать к созданию первого бота для игры в го. Он будет очень слабым игроком, но послужит в качестве основы для остальных более совершенных ботов. Сначала необходимо определить интерфейс, который будут использовать все наши боты. Следующее определение агента необходимо добавить в файл *base.py* в модуле *agent*.

Листинг 3.16 ❖ Центральный интерфейс для агентов го

```
class Agent:
    def __init__(self):
        pass

    def select_move(self, game_state):
        raise NotImplementedError()
```

Вот и все, достаточно лишь одного метода. Данный бот просто выбирает ход, исходя из текущего игрового состояния. Разумеется, внутренне это может требовать решения других сложных задач, например оценки текущей позиции, однако для игры боту этого достаточно.

Наша первая реализация будет максимально примитивной: она будет случайным образом выбирать любой допустимый ход, в результате которого не заполняется ни один из глаз собственных групп камней. При отсутствии таких ходов бот будет пасовать. Код этого бота следует поместить в файл *naive.py* в папку *agent*. Как говорилось в главе 2, ранги начинающих игроков в го обычно варьируются от 30 до 1 кю. Наш бот, выбирающий ходы случайным образом, будет играть на уровне абсолютного новичка – 30 кю.

Листинг 3.17 ❖ Бот, выбирающий ходы случайным образом, имеет ранг около 30 кю

```
import random
from dlgo.agent.base import Agent
from dlgo.agent.helpers import is_point_an_eye
from dlgo.goboard_slow import Move
from dlgo.gotypes import Point

class RandomBot(Agent):
    def select_move(self, game_state):
        """Choose a random valid move that preserves our own eyes."""
        candidates = []
        for r in range(1, game_state.board.num_rows + 1):
            for c in range(1, game_state.board.num_cols + 1):
                candidate = Point(row=r, col=c)
                if game_state.is_valid_move(Move.play(candidate)) and \
                        not is_point_an_eye(game_state.board,
                                            candidate,
                                            game_state.next_player):
                    candidates.append(candidate)
```

```
        if not candidates:
            return Move.pass_turn()
        return Move.play(random.choice(candidates))
```

На данном этапе структура вашего модуля должна выглядеть следующим образом (для инициализации подмодуля не забудьте поместить в папку пустой файл *__init__.py*):

```
dlgo
  ...
  agent
    __init__.py
    helpers.py
    base.py
    naive.py
```

Наконец, мы можем создать программу-драйвер, позволяющую двум экземплярам нашего бота сыграть целую партию. Сначала необходимо определить вспомогательные функции, такие как вывод на консоль всей доски или отдельного хода.

Координаты доски можно задавать разными способами, однако в Европе чаще всего столбцы обозначают буквами алфавита, начиная с A, а строки – цифрами, начиная с 1. При таком способе маркировки стандартной доски 19×19 нижний левый угол будет иметь координату A1, а верхний правый угол – T19. Обратите внимание на то, что буква I не используется, поскольку ее можно спутать с цифрой 1.

Определим строковую переменную `COLS = 'ABCDEFGHJKLMNOPQRST'`, символы которой будут обозначать столбцы доски для игры в го. Для отображения доски в командной строке мы кодируем пустое поле с помощью точки (`.`), черный камень – с помощью символа `x`, а белый – с помощью символа `o`. Следующий фрагмент кода нужно поместить в новый файл под названием *utils.py* в папке *dlgo*. Функция `print_move` выводит на командную строку следующий ход, а функция `print_board` – текущее состояние доски со всеми расположенными на ней камнями. Этот код нужно поместить в файл с именем *bot_v_bot.py* за пределами модуля *dlgo*.

Листинг 3.18 ❖ Служебные функции для игр «бот против бота»

```
from dlgo import gotypes

COLS = 'ABCDEFGHJKLMNOPQRST'
STONE_TO_CHAR = {
    None: ' . ',
    gotypes.Player.black: ' x ',
    gotypes.Player.white: ' o ',
}

def print_move(player, move):
    if move.is_pass:
        move_str = 'passes'
    elif move.is_resign:
        move_str = 'resigns'
    else:
        move_str = '%s%d' % (COLS[move.point.col - 1], move.point.row)
    print('%s %s' % (player, move_str))
```

```
def print_board(board):
    for row in range(board.num_rows, 0, -1):
        bump = " " if row <= 9 else ""
        line = []
        for col in range(1, board.num_cols + 1):
            stone = board.get(gotypes.Point(row=row, col=col))
            line.append(STONE_TO_CHAR[stone])
        print('%s%d %s' % (bump, row, ''.join(line)))
    print('    ' + '  '.join(COLS[:board.num_cols]))
```

Мы можем написать сценарий, запускающий двух ботов, выбирающих ходы случайным образом, которые играют друг с другом на доске 9×9 до тех пор, пока не решат, что игра окончена.

Листинг 3.19 ❖ Сценарий, позволяющий боту играть против самого себя

```
from dlgo import agent
from dlgo import goboard
from dlgo import gotypes
from dlgo.utils import print_board, print_move
import time

def main():
    board_size = 9
    game = goboard.GameState.new_game(board_size)
    bots = {
        gotypes.Player.black: agent.naive.RandomBot(),
        gotypes.Player.white: agent.naive.RandomBot(),
    }
    while not game.is_over():
        time.sleep(0.3)

        print(chr(27) + "[2J")
        print_board(game.board)
        bot_move = bots[game.next_player].select_move(game)
        print_move(game.next_player, bot_move)
        game = game.apply_move(bot_move)

if __name__ == '__main__':
    main()
```

Установка таймера на 0,3 секунды обеспечивает задержку между отображением ходов бота.

Перед выполнением каждого хода экран очищается, благодаря чему доска всегда отображается в одном и том же месте командной строки.

Чтобы запустить игру ботов на командной строке, выполните следующую команду:

```
python bot_v_bot.py
```

На экран будет выведено множество ходов, и игра закончится, когда оба игрока спасуют. Напомним, что черные камни кодируются с помощью символа x, белые камни – с помощью символа o, а пустые точки – с помощью символа (.). Вот пример последнего хода белых в сгенерированной игре:

```
9 o.ooooooo
8 ooooxxoxx
7 oooox.xxx
6 o.ooxxxxx
```

```
5 ooooxxxxx
4 ooooxxxxx
3 o.ooox.xx
2 ooooxxxxx
1 o.ooxxx.
  ABCDEFGHJ
Player.white passes
```

На данном этапе этот бот является не только слабым, но и весьма раздражающим противником: он упорно заполняет доску камнями, даже при очевидной безнадежности положения. Более того, вне зависимости от количества партий, сыгранных этими ботами, никакого *обучения* не происходит. Данный бот, выбирающий ходы случайным образом, навсегда останется на текущем уровне.

На протяжении оставшейся части книги мы будем постепенно решать обе эти проблемы, создавая все более интересный и мощный движок для игры в го.

3.5. Ускорение игрового процесса с помощью Zobrist-хеширования

Перед описанием процесса игры против бота нам следует решить проблему скорости его текущей реализации. Если вас не интересуют способы ускорения игрового процесса, можете сразу перейти к разделу 3.6.

Как говорилось в разделе 3.2, для выполнения проверки на предмет ситуационного суперко необходимо просмотреть всю историю игровых состояний, чтобы выяснить, встречалось ли текущее состояние раньше. Это требует большого количества вычислений. Для решения данной проблемы достаточно внести некоторые изменения: вместо сохранения прошлых состояний доски можно просто сохранить соответствующие *хеш-значения*.

Методы хеширования очень часто используются в сфере информатики. В таких играх, как шахматы, широко применяется *Zobrist-хеширование* (получившее свое название в честь ученого в области информатики Альберта Зобриста, создавшего в начале 1970-х годов одного из первых ботов для игры в го). При использовании Zobrist-хеширования каждому возможному ходу на доске присваивается хеш-значение. Для достижения наилучших результатов хеш-значения следует выбирать случайным образом. В игре го каждый ход совершается либо черными, либо белыми, поэтому хеш-таблица для доски 19×19 будет состоять из $2 \times 19 \times 19 = 722$ хеш-значений. Эти 722 хеша, соответствующих отдельным ходам, будут использоваться для кодирования самых сложных состояний доски. На рис. 3.9 показано, как это работает.

Самым интересным в представленной выше процедуре является то, что полное состояние доски может быть закодировано с помощью одного-единственного хеш-значения. Мы начинаем с хеш-значения пустой доски, которое для простоты можно задать равным 0. Первый ход имеет свое хеш-значение, и мы можем применить его к доске, выполнив операцию `XOR` над хеш-значениями доски и хода. Эта операция называется *применением хеш-значения*. Таким образом, для совершения каждого последующего хода мы применяем к доске соответствующий хеш. Это позволяет отслеживать текущее состояние доски с помощью одного хеш-значения.

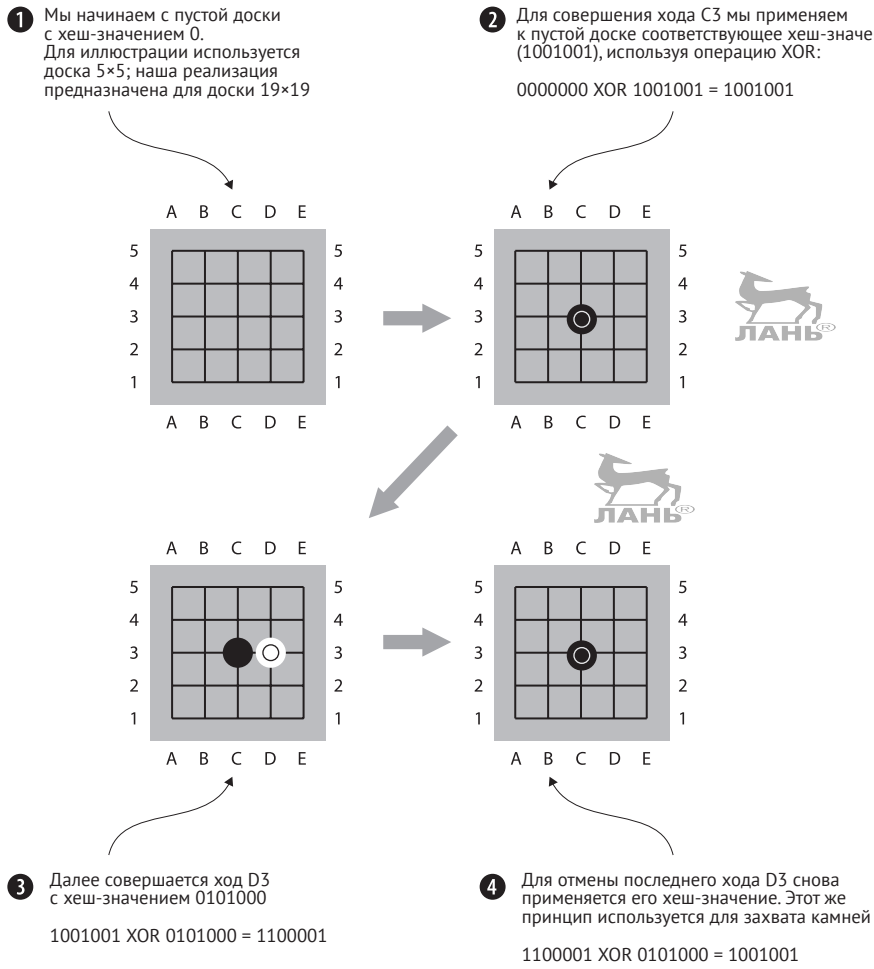


Рис. 3.9 ❖ Использование Zobrist-хеширования для кодирования ходов и эффективного хранения игрового состояния

Обратите внимание на то, что мы можем отменить любой ход путем повторного применения его хеша (в этом заключается удобство операции `XOR`). Этот процесс называется *отменой применения хеш-значения* и является очень важным, поскольку с его помощью можно легко удалять с доски захваченные камни. Например, при захвате черного камня в точке C3 его можно удалить из хеш-значения текущего состояния доски, применив хеш-значение C3. Разумеется, в этом случае также должно быть применено хеш-значение белого камня, размещение которого привело к захвату черного камня в точке C3. Если размещение белого камня привело к захвату нескольких черных камней, необходимо отменить применение всех соответствующих хеш-значений.

Таким способом можно закодировать любое состояние доски при условии выбора достаточно больших и общих хеш-значений, не допускающих возникновения коллизий хеш-функции (когда два разных игровых состояния имеют одинаковое

хеш-значение). На практике мы не выполняем проверку на предмет коллизии хеш-функции, а просто предполагаем, что они отсутствуют.

Перед применением Zobrist-хеширования к реализации доски для игры в го необходимо сгенерировать хеши. Для генерирования 64-битных случайных целых чисел мы используем библиотеку Python `random` для каждого из 3×19×19 возможных состояний точек доски. Учтите, что в Python операция `XOR` обозначается символом ^. Для пустой доски выбрано значение 0.

Листинг 3.20 ❖ Генерация Zobrist-хешей

```
import random

from dlgo.gotypes import Player, Point

def to_python(player_state):
    if player_state is None:
        return 'None'
    if player_state == Player.black:
        return Player.black
    return Player.white

MAX63 = 0x7fffffffffffffff

table = {}
empty_board = 0
for row in range(1, 20):
    for col in range(1, 20):
        for state in (Player.black, Player.white):
            code = random.randint(0, MAX63)
            table[Point(row, col), state] = code

print('from .gotypes import Player, Point')
print('')
print("__all__ = ['HASH_CODE', 'EMPTY_BOARD']")
print('')
print('HASH_CODE = {')
for (pt, state), hash_code in table.items():
    print('    (%r, %s): %r,' % (pt, to_python(state), hash_code))
print('}')
print('')
print('EMPTY_BOARD = %d' % (empty_board,))
```

После запуска данного сценария в командной строке отображаются нужные хеши. Выполнение представленного выше кода генерирует код Python, отображаемый в командной строке. Поместите эти выходные данные в файл *zobrist.py* в модуле *dlgo*.

Теперь, когда у нас есть необходимые хеши, пришло время заменить старый механизм отслеживания игровых состояний механизмом сохранения хешей. Создайте копию файла *goboard_slow.py* с именем *goboard.py*, в которую нужно будет внести все необходимые изменения в оставшейся части этого раздела. Вместо этого можно использовать код в файле *goboard.py* из нашего GitHub-репозитория. Для начала сделайте так, чтобы `GoString`, а также `stones` и `liberties` не могли быть изменены после создания. Для этого можно использовать тип данных Python `fro-`

zenset вместо set. Множество frozenset не предусматривает методов для добавления или удаления элементов, поэтому нам нужно создать новое множество вместо изменения существующего.

Листинг 3.21 ❖ Экземпляры GoString с неизменяемыми множествами камней и степеней свободы

```
class GoString:
    def __init__(self, color, stones, liberties):
        self.color = color
        self.stones = frozenset(stones)
        self.liberties = frozenset(liberties)

    def without_liberty(self, point):
        new_liberties = self.liberties - set([point])
        return GoString(self.color, self.stones, new_liberties)

    def with_liberty(self, point):
        new_liberties = self.liberties | set([point])
        return GoString(self.color, self.stones, new_liberties)
```

Множества камней и степеней свободы теперь являются неизменяемыми.

Метод without_liberty заменил предыдущий метод remove_liberty...

...а метод with_liberty заменил add_liberty.

В GoString мы заменяем два метода для учета неизменного состояния, а другие вспомогательные методы, такие как merged_with или num_liberties, оставляем без изменений.

Затем мы обновляем соответствующие части доски. Не забудьте поместить весь код, приведенный в оставшейся части этого раздела, в файл *goboard.py* – копию файла *goboard_slow.py*.

Листинг 3.22 ❖ Создание экземпляра доски для игры в го с использованием значения _hash для пустой доски

```
from dlgo import zobrist

class Board:
    def __init__(self, num_rows, num_cols):
        self.num_rows = num_rows
        self.num_cols = num_cols
        self._grid = {}
        self._hash = zobrist.EMPTY_BOARD
```

Далее, в методе place_stone при размещении каждого последующего камня мы применяем хеш соответствующего цвета. Не забудьте внести эти изменения в файл *goboard.py*, как и в случае со всеми остальными фрагментами кода, приведенными в этом разделе.

Листинг 3.23 ❖ Размещение камня означает применение соответствующего хеш-значения

```
        new_string = GoString(player, [point], liberties)

        for same_color_string in adjacent_same_color:
            new_string = new_string.merged_with(same_color_string)
        for new_string_point in new_string.stones:
            self._grid[new_string_point] = new_string

        self._hash ^= zobrist.HASH_CODE[point, player]

        for other_color_string in adjacent_opposite_color:
```

До этой строки метод place_stone остается прежним.

Объединение любых смежных цепочек камней одного цвета.

Применение хеш-кода для данной точки и игрока.

```
            replacement = other_color_string.without_liberty(point)
            if replacement.num_liberties:
 self._replace_string(other_color_string.without_liberty(point))
            else:
                self._remove_string(other_color_string)
```

Уменьшение количества степеней свободы любых смежных цепочек камней другого цвета.

Удаление цепочек камней другого цвета с нулевой степенью свободы.

Для удаления камня к доске еще раз применяется его хеш-значение.

Листинг 3.24 ❖ Удаление камня означает отмену применения его хеш-значения.

```
    def _replace_string(self, new_string):
        for point in new_string.stones:
            self._grid[point] = new_string

    def _remove_string(self, string):
        for point in string.stones:
            for neighbor in point.neighbors():
                neighbor_string = self._grid.get(neighbor)
                if neighbor_string is None:
                    continue
                if neighbor_string is not string:
                    self._replace_string(neighbor_string.with_liberty(point))
            self._grid[point] = None

            self._hash ^= zobrist.HASH_CODE[point, string.color]
```

Этот новый вспомогательный метод обновляет сетку доски.

Удаление цепочки камней может увеличить степени свободы других цепочек.

При использовании Zobrist-хеширования необходимо отменить применение хеш-значения для этого хода.

В класс `Board` осталось добавить лишь служебный метод для возврата текущего Zobrist-хеша.

Листинг 3.25 ❖ Возврат текущего Zobrist-хеша доски

```
    def zobrist_hash(self):
        return self._hash
```

Теперь, когда мы закодировали игровую доску с помощью хеш-значений Zobrist, давайте обсудим способ улучшения класса `GameState`.

Раньше предыдущее игровое состояние задавалось так: `self.previous_state = previous`, что было слишком неэффективным, поскольку проверка применимости правила ко предполагала перебор всех предыдущих игровых состояний. Вместо этого мы можем сохранить Zobrist-хеши, используя новую переменную `previous_states`, как показано в следующем листинге.

Листинг 3.26 ❖ Инициализация игрового состояния с помощью Zobrist-хешей

```
class GameState:
    def __init__(self, board, next_player, previous, move):
        self.board = board
        self.next_player = next_player
        self.previous_state = previous
        if self.previous_state is None:
            self.previous_states = frozenset()
        else:
            self.previous_states = frozenset(
```

```
                previous.previous_states |
                {(previous.next_player, previous.board.zobrist_hash())})
        self.last_move = move
```

При пустой доске `self.previous_states` представляет собой пустое неизменяемое множество `frozenset`. В противном случае мы дополняем состояния парой значений: цветом следующего игрока и Zobrist-хешем предыдущего игрового состояния.

После внесения всех этих изменений мы можем значительно улучшить реализацию `do_move_violate_ko`.

Листинг 3.27 ❖ Быстрая проверка игровых состояний на применимость правила ко с помощью Zobrist-хешей

```
def does_move_violate_ko(self, player, move):
    if not move.is_play:
        return False
    next_board = copy.deepcopy(self.board)
    next_board.place_stone(player, move.point)
    next_situation = (player.other, next_board.zobrist_hash())
    return next_situation in self.previous_states
```

Проверка предыдущих состояний доски с помощью одной строки `next_situation in self.previous_states` производится на порядок быстрее по сравнению с циклической обработкой состояний доски, используемой ранее.

Этот интересный прием с хешированием позволит значительно ускорить игру бота с самим собой в последующих главах, что будет способствовать более быстрому улучшению игрового процесса.

Дополнительное ускорение процесса реализации доски для игры в го

Мы подробно рассмотрели исходную реализацию доски для игры в го (см. файл *goboard_slow.py*) и показали, как ее можно ускорить с помощью Zobrist-хеширования (см. файл *goboard.py*). Репозиторий GitHub включает еще одну реализацию доски под названием *goboard_fast.py*, которая позволяет еще больше ускорить игровой процесс. Эти улучшения, сделанные в ущерб удобочитаемости кода, продемонстрируют свою чрезвычайную ценность в следующих главах.

Если вы хотите сделать доску для игры в го еще быстрее, обратитесь к файлу *goboard_fast.py* и содержащимся в нем комментариям. Большая часть оптимизации сводится к приемам, позволяющим избежать конструирования и копирования объектов Python.

3.6. Игра против собственного бота

Теперь, когда у вас есть слабый бот, способный играть с самим собой, вам может захотеться применить свои знания, полученные при изучении главы 2, и сыграть против него самостоятельно. Это возможно и не требует внесения большого количества изменений.

Для этого в файл *utils.py* необходимо поместить еще одну служебную функцию, преобразующую входные данные, полученные от пользователя, в координаты.

Листинг 3.28 ❖ Преобразование входных данных, полученных от пользователя, в координаты доски

```
def point_from_coords(coords):
    col = COLS.index(coords[0]) + 1
    row = int(coords[1:])
    return gotypes.Point(row=row, col=col)
```

Данная функция преобразует входные данные вида C3 или E7 в координаты доски для игры в го. С учетом этого вы можете настроить свою программу для игры на доске 9×9 в файле *human_v_bot.py*.

Листинг 3.29 ❖ Настройка сценария, позволяющего вам играть против собственного бота

```
from dlgo import agent
from dlgo import goboard_slow as goboard
from dlgo import gotypes
from dlgo.utils import print_board, print_move, point_from_coords
from six.moves import input


def main():
    board_size = 9
    game = goboard.GameState.new_game(board_size)
    bot = agent.RandomBot()

    while not game.is_over():
        print(chr(27) + "[2J")
        print_board(game.board)
        if game.next_player == gotypes.Player.black:
            human_move = input('-- ')
            point = point_from_coords(human_move.strip())
            move = goboard.Move.play(point)
        else:
            move = bot.select_move(game)
        print_move(game.next_player, move)
        game = game.apply_move(move)

if __name__ == '__main__':
    main()
```

Вы, игрок-человек, будете играть черными. Ваш бот – белыми. Запустите сценарий с помощью следующей команды:

```
python human_v_bot.py
```

Вам будет предложено ввести ход и подтвердить его с помощью клавиши **Enter**. Например, в случае хода G3 ответ бота может выглядеть следующим образом:

```
Player.white D8
9 .........
8 ...o.....
7 .........
6 .........
5 .........
4 .........
3 ......x..
```

```
2 .........
1 .........
  ABCDEFGHJ
```

При желании вы можете сыграть с ботом целую партию. Однако это не очень интересно, поскольку бот выбирает ходы случайным образом.

Обратите внимание на то, что в плане соблюдения правил игры го этот бот уже готов. Он знает об этой игре все, что ему необходимо. Это позволяет нам полностью сосредоточиться на алгоритмах, улучшающих игровой процесс. Данный бот представляет собой базу для создания более сильных ботов с помощью интересных методов, обсуждаемых в следующих главах.

3.7. Резюме

- Двух игроков го лучше всего кодировать с помощью перечислений (`enum`).
- Точка на игровой доске характеризуется ее непосредственным соседством.
- В го существует три варианта хода – размещение камня, пас и выход из игры.
- Цепочки камней – связанные группы камней одного цвета. Цепочки обеспечивают эффективность проверки захваченных камней после размещения камня.
- Класс `Board` отвечает за логику размещения и захвата камней.
- Класс `GameState` учитывает все камни на доске и историю предыдущих игровых состояний, а также отслеживает очередь игроков.
- Правило ко можно реализовать с помощью ситуационного правила суперко.
- Zobrist-хеширование является важным методом для эффективного кодирования истории игрового процесса и ускорения проверки применимости правила ко.
- Агент игры в го может быть определен одним методом `select_move`.
- Наш бот может играть с самим собой, другими ботами и людьми.

Часть II

МАШИННОЕ ОБУЧЕНИЕ И ИГРОВОЙ ИИ

Во второй части книги мы обсудим компоненты классического и современного игрового ИИ. Сначала изучим несколько алгоритмов поиска по дереву, которые представляют большую важность при создании игрового ИИ, а также при решении всевозможных задач оптимизации. Затем мы поговорим о глубоком обучении и нейронных сетях, начиная с математических основ и заканчивая различными практическими соображениями, касающимися дизайна. И наконец, мы затронем тему обучения с подкреплением, которое позволяет игровому ИИ совершенствоваться благодаря практике.

Разумеется, описанные техники используются не только в сфере игр. После освоения этих компонентов вы начнете замечать возможности для их применения повсюду.

Глава 4

Игры и поиск по дереву

В этой главе:

- выбор лучшего хода путем минимаксного перебора;
- ускорение поиска по дереву с помощью редукции;
- применение метода Монте-Карло в играх.

Предположим, перед нами стоят две задачи. Первая – написать компьютерную программу, которая играет в шахматы. Вторая – написать складскую программу для эффективного планирования заказов. Что общего может быть у этих программ? На первый взгляд, практически ничего. Но если абстрагироваться от деталей, то можно заметить несколько параллелей:

- **нам требуется принять последовательность решений:** в шахматах необходимо решить, какую фигуру следует передвинуть. В случае со складом решения касаются выбора товаров;
- **предыдущие решения могут повлиять на дальнейшие решения:** в шахматах перемещение пешки может позволить противнику атаковать вашу королеву несколько ходов спустя. На складе поход за какой-то вещью к полке 17 может позднее потребовать возврата к полке 99;
- **после принятия последовательности решений мы можем оценить эффективность достижения своей цели:** после завершения партии в шахматы мы знаем, кто победил. В случае со складом мы можем подсчитать, сколько времени требуется для сбора всех нужных вещей;
- **количество возможных последовательностей может оказаться огромным:** существует около 10^{100} вариантов шахматных партий. А для того чтобы добраться до 20 нужных вещей на складе, есть 2 млрд возможных путей.

Разумеется, эта аналогия имеет свои ограничения. В шахматах, например, вы ходите по очереди с противником, который активно пытается мешать реализации ваших намерений, чего не бывает ни на одном нормальном складе.

В информатике алгоритмы *поиска по дереву* представляют собой стратегии, позволяющие обойти множество возможных последовательностей решений и выбрать ту, которая дает наилучший результат. В этой главе обсуждаются алгоритмы поиска по дереву, используемые в играх. Однако многие из них можно применять и при решении других задач оптимизации. Мы начнем с *минимаксного* алгоритма поиска, который предполагает поочередное переключение между перспективами двух соперников. Минимаксный алгоритм позволяет находить идеальные последовательности ходов, но он является слишком медленным для сложных игр. Затем мы рассмотрим два метода получения полезного результата путем ис-

следования небольшой части дерева. Одним из них является *редукция*, позволяющая ускорить процесс поиска путем удаления некоторых разделов дерева. Для эффективной редукции в код необходимо внести реальные данные о проблеме. Когда это невозможно, поиск по дереву может выполняться *методом Монте-Карло* (ММК). ММК – это алгоритм рандомизированного поиска, который позволяет найти хороший результат без использования кода, являющегося специфическим для какой-то конкретной области.

С помощью этих приемов можно создать ИИ, способный играть в различные настольные и карточные игры.

4.1. Классификация игр

Алгоритмы поиска по дереву в основном используются в играх, где участники ходят по очереди и каждый ход предполагает отдельный набор вариантов. Под это описание подходят многие настольные и карточные игры. С другой стороны, поиск по дереву не поможет компьютеру играть в баскетбол, шарады или World of Warcraft. Кроме того, мы можем произвести дополнительную классификацию настольных и карточных игр с помощью следующих двух критериев:

- **детерминированная/недетерминированная игра:** ход *детерминированной игры* зависит только от решений игроков. В недетерминированной игре задействован элемент случайности, связанный, например, с бросанием костей или тасовкой карт;
- **игра с полной/неполной информацией:** в *играх с полной информацией* обоим игрокам на любом этапе доступно все игровое состояние – им видна вся доска или карты всех игроков на столе. В *играх с неполной информацией* каждому игроку доступна только часть игрового состояния. К играм с неполной информацией относятся карточные игры, где каждый игрок получает несколько карт и не знает, какие карты достались соперникам. Привлекательность таких игр отчасти связана с угадыванием того, что знают другие игроки, исходя из их игровых решений.

В табл. 4.1 приведена классификация нескольких известных игр.

Таблица 4.1. Классификация настольных и карточных игр

	Детерминированная игра	**Недетерминированная игра**
С полной информацией	Го, шахматы	Нарды
С неполной информацией	Морской бой, «Стратего»	Покер, «Эрудит»

В этой главе мы в первую очередь сосредоточимся на детерминированных играх с полной информацией. В таких играх каждый ход теоретически предусматривает один самый лучший вариант. В них нет места удаче и каким-либо секретам: перед выбором хода вы уже знаете, как ваш противник может на него отреагировать, и способны предвидеть развитие всей партии. Теоретически всю партию следует спланировать уже на первом ходу. Минимаксный алгоритм делает именно это с целью разработки идеальной партии.

На самом деле такие испытанные временем игры, как шахматы и го, предполагают огромное количество возможностей. Людям кажется, что каждая партия

живет своей собственной жизнью, и даже компьютеры не способны просчитать их до конца.

Все приведенные в этой главе примеры содержат небольшое количество игровой логики, что позволяет адаптировать их к любой детерминированной игре с полной информацией. Для этого можно взять за основу модуль *goboard* и реализовать новую игровую логику в таких классах, как `Player`, `Move` и `GameState`. Основными функциями для класса `GameState` являются `apply_move`, `legal_moves`, `is_over` и `winner`. Мы сделали это для игры крестики-нолики; результат можно найти в модуле *ttt* на GitHub (**mng.bz/gYPe**.)

Игры для экспериментов с ИИ

В качестве источника вдохновения можно использовать правила любой из перечисленных далее игр:

- шахматы;
- шашки;
- реверси;
- гекс;
- китайские шашки;
- манкала;
- мельница;
- гомоку.

4.2. Прогнозирование действий противника с помощью алгоритма минимаксного поиска

Как запрограммировать компьютер на выбор следующего хода в игре? Для начала следует подумать о том, как люди решают эту задачу. Начнем с самой простой детерминированной игры с полной информацией – крестики-нолики. Стратегия, которую мы опишем, называется *минимаксной*, сокращенно от *минимизации и максимизации*: вы пытаетесь максимизировать свой выигрыш, а ваш противник старается минимизировать его. Этот алгоритм предполагает, что ваш противник так же умен, как и вы.

Давайте посмотрим, как минимаксный поиск работает на практике. Взгляните на рис. 4.1. Какой ход следует сделать игроку X?

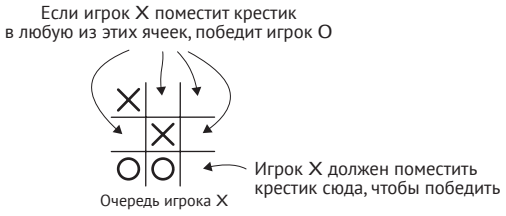


Рис. 4.1 ❖ Какой ход следует сделать игроку X?
Все просто: крестик в правом нижнем углу обеспечит этому игроку победу

Здесь нет никакого подвоха. Крестик в правом нижнем углу обеспечит победу игроку X.

Можно сформулировать общее правило: делать любой ход, который сразу же обеспечивает победу в игре. Этот план не может не сработать. Данное правило можно реализовать в коде следующим образом.

Листинг 4.1 ❖ Функция, определяющая ход, который сразу обеспечивает победу в игре

```
def find_winning_move(game_state, next_player):
  for candidate_move in game_state.legal_moves(next_player):
    next_state = game_state.apply_move(candidate_move)
    if next_state.is_over() and next_state.winner == next_player:
      return candidate_move
  return None
```

Циклическая обработка всех допустимых ходов.

Определение состояния доски при выборе конкретного хода.

Этот ход является выигрышным! Продолжать поиск не нужно.

Это ход не может обеспечить победу.

На рис. 4.2 показаны гипотетические состояния доски, которые эта функция будет исследовать. Структура, связывающая состояние доски с возможными последующими состояниями, называется *игровым деревом*.

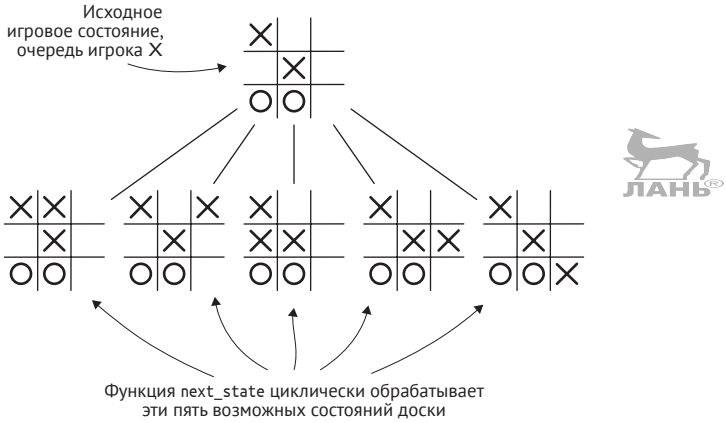


Рис. 4.2 ❖ Иллюстрация алгоритма поиска выигрышного хода. Мы начинаем с состояния доски, показанного вверху. Затем перебираем все возможные ходы и рассчитываем получающиеся в результате этих ходов игровые состояния. Потом мы проверяем, является ли это гипотетическое игровое состояние выигрышным для игрока X

Давайте вернемся немного назад. Как мы пришли к этому состоянию? Возможно, предыдущее состояние выглядело, как на рис. 4.3. Игрок O надеялся заполнить ноликами нижний ряд ячеек. Однако это предполагало сотрудничество со стороны игрока X. Так, из приведенного выше правила можно сделать вывод: не следует выбирать ход, предоставляющий вашему противнику возможность совершить выигрышный ход. Данная логика реализована в листинге 4.2.

Рис. 4.3 ❖ Какой ход следует совершить игроку O? Если он поместит нолик в левом нижнем углу, стоит предположить, что игрок X поместит крестик в нижнем правом углу, чтобы обеспечить себе победу. Игроку O следует найти единственный ход, позволяющий предотвратить это

Листинг 4.2 ❖ Функция, которая не дает противнику сделать выигрышный ход

```
def eliminate_losing_moves(game_state, next_player):
  opponent = next_player.other()
  possible_moves = []
  for candidate_move in game_state.legal_moves(next_player):
    next_state = game_state.apply_move(candidate_move)
    opponent_winning_move = find_winning_move(next_state, opponent)
    if opponent_winning_move is None:
      possible_moves.append(candidate_move)
  return possible_moves
```

`possible_moves` превращается в список всех ходов, достойных рассмотрения.

Циклическая обработка всех допустимых ходов.

Определение состояния доски при выборе конкретного хода.

Позволяет ли это противнику сделать выигрышный ход? Если нет, то этот вариант допустим.

Теперь вы знаете, что задача заключается в том, чтобы не позволить противнику оказаться в выигрышной позиции. Однако вам следует предположить, что ваш противник попытается сделать то же самое с вами. Как же выиграть с учетом этого? Рассмотрим состояние доски на рис. 4.4.

Рис. 4.4 ❖ Какой ход следует сделать игроку X? Если он поместит крестик в центральную ячейку, то у него будет две ячейки, позволяющие заполнить ряд: (1) верхняя средняя и (2) правая нижняя. Игрок O может заблокировать только один из этих вариантов, что гарантирует выигрыш игроку X

Если поместить крестик в центральную ячейку, у нас будет две ячейки, позволяющие заполнить ряд: верхняя средняя и правая нижняя. Противник не может заблокировать и ту, и другую. Мы можем описать этот общий принцип следующим образом: ищите ход, обеспечивающий следующий выигрышный ход, кото-

рый оппонент не может заблокировать. Звучит сложно, однако эту логику легко реализовать поверх уже написанных функций.

Листинг 4.3 ❖ Функция, которая находит последовательность из двух ходов, гарантирующую выигрыш

```
def find_two_step_win(game_state, next_player):
  opponent = next_player.other()
  for candidate_move in game_state.legal_moves(next_player):
    next_state = game_state.apply_move(candidate_move)
    good_responses = eliminate_losing_moves(next_state, opponent)
    if not good_responses:
      return candidate_move
  return None
```

Циклическая обработка всех допустимых ходов.

Определение состояния доски при выборе конкретного хода.

Имеет ли ваш противник хорошую защиту? Если нет, выбирайте этот ход.

Вне зависимости от выбранного вами хода противник может предотвратить вашу победу.

Ваш противник предвидит такое развитие событий и попытается помешать вам. Уже можно заметить формирование общей стратегии:

1) проверьте, сможете ли вы победить на следующем ходу. Если да, сделайте этот ход;
2) если нет, посмотрите, может ли ваш противник победить на следующем ходу. Если да, заблокируйте этот ход;
3) если нет, посмотрите, можете ли вы победить в два хода. Если да, выберите эту последовательность ходов;
4) если нет, посмотрите, может ли ваш противник победить в два хода.

Обратите внимание на то, что все три функции имеют похожую структуру. Каждая функция циклически обрабатывает все допустимые ходы и исследует состояние доски, являющееся следствием выбора конкретного хода. Кроме того, каждая функция учитывает предыдущую, чтобы спрогнозировать реакцию противника. Если обобщить эту концепцию, то мы получим алгоритм выбора наилучшего хода.

4.3. Крестики-нолики: пример использования минимаксного алгоритма

В предыдущем разделе мы рассмотрели способ прогнозирования действий противника на один или два хода вперед. В этом разделе мы обобщим стратегию для выбора идеальных ходов в игре крестики-нолики. Основная идея остается прежней, однако нам требуется определенная гибкость, для того чтобы просчитать произвольное количество ходов.

Сначала давайте определим перечисление, представляющее три возможных исхода игры: выигрыш, проигрыш или ничью. Эти возможности определяются относительно конкретного игрока: проигрыш для одного игрока является победой для другого.

Листинг 4.4 ❖ Перечисление для представления результата игры

```
class GameResult(enum.Enum):
    loss = 1
    draw = 2
    win = 3
```

Представим, что у нас есть функция best_result, которая принимает игровое состояние и сообщает наилучший результат, которого может добиться игрок из этого состояния. Если бы этот игрок мог гарантировать выигрыш с помощью последовательности ходов любой сложности, то функция best_result возвратила бы GameResult.win. Если бы у этого игрока была возможность обеспечить ничью, функция возвратила бы GameResult.draw. В противном случае функция возвратила бы GameResult.loss. Если мы предположим существование такой функции, то легко сможем написать функцию для выбора хода: мы перебираем все возможные ходы, вызываем best_result и выбираем ход, который приводит к наилучшему для нас результату. К одинаковым результатам может привести множество ходов. В этом случае мы можем выбирать их случайным образом. В следующем листинге показано, как можно это реализовать.

Листинг 4.5 ❖ Игровой агент, реализующий минимаксный алгоритм поиска

```
class MinimaxAgent(Agent):
    def select_move(self, game_state):
        winning_moves = []
        draw_moves = []
        losing_moves = []
        for possible_move in game_state.legal_moves():
            next_state = game_state.apply_move(possible_move)
            opponent_best_outcome = best_result(next_state)
            our_best_outcome = reverse_game_result(opponent_best_outcome)
            if our_best_outcome == GameResult.win:
                winning_moves.append(possible_move)
            elif our_best_outcome == GameResult.draw:
                draw_moves.append(possible_move)
            else:
                losing_moves.append(possible_move)
        if winning_moves:
            return random.choice(winning_moves)
        if draw_moves:
            return random.choice(draw_moves)
        return random.choice(losing_moves)
```

Циклическая обработка всех допустимых ходов.

Определение состояния доски при выборе конкретного хода.

Классификация хода в зависимости от его результата.

Поскольку следующий ход делает ваш противник, выясните его наилучший возможный результат. Ваш результат будет противоположным.

Выбор хода, обеспечивающего наилучший для вас результат.

Теперь вопрос в том, как реализовать функцию best_result. Как и в предыдущем разделе, мы можем начать с конца игры и двигаться в обратном направлении. Следующий листинг демонстрирует простой случай: если игра уже окончена, возможен только один результат, который функция и возвращает.

Листинг 4.6 ❖ Первый этап минимаксного алгоритма поиска

```
def best_result(game_state):
    if game_state.is_over():
        if game_state.winner() == game_state.next_player:
            return GameResult.win
        elif game_state.winner() is None:
            return GameResult.draw
        else:
            return GameResult.loss
```

Если мы находимся где-то в середине партии, нам требуется спрогнозировать ее возможное развитие. К настоящему времени алгоритм уже должен казаться знакомым. Сначала мы циклически обрабатываем все возможные ходы и определяем следующее игровое состояние. Затем предполагаем, что наш противник попытается помешать нам в осуществлении выбранного хода. Для этого мы можем вызвать функцию `best_result` из этого нового игрового состояния. Она покажет нам результат, который может получить наш *противник*, исходя из нового состояния, и мы инвертируем его, чтобы узнать свой результат. Из всех рассматриваемых ходов мы выбираем тот, который приводит к наилучшему для нас результату. В листинге 4.7 показан способ реализации этой логики, которая составляет вторую половину функции `best_result`. На рис. 4.5 показаны состояния доски, которые эта функция будет рассматривать для конкретной партии в крестики-нолики.

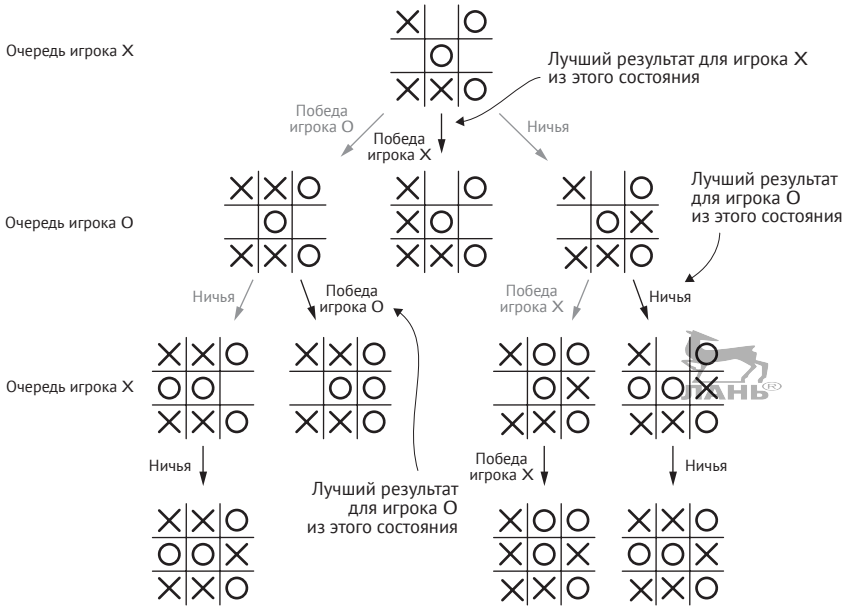


Рис. 4.5 ❖ Игровое дерево для игры в крестики-нолики. В верхнем случае очередь хода за игроком X. Помещение крестика в верхнюю ячейку центрального столбца гарантирует победу игроку O. Помещение крестика в среднюю ячейку левого столбца гарантирует победу игроку X. Помещение крестика в среднюю ячейку правого столбца позволяет игроку O обеспечить ничью. Поэтому игрок X поместит крестик в средней ячейке левого столбца

Листинг 4.7 ❖ Реализация минимаксного алгоритма поиска

```
best_result_so_far = GameResult.loss
for candidate_move in game_state.legal_moves():
    next_state = game_state.apply_move(candidate_move)
    opponent_best_result = best_result(next_state)
```

Посмотрите, как будет выглядеть доска в случае выбора данного хода.

Определите лучший ход вашего противника.

```
    our_result = reverse_game_result(opponent_best_result)
    if our_result.value > best_result_so_far.value:
        best_result_so_far = our_result
return best_result_so_far
```

Что бы ни было нужно вашему противнику, вам нужно противоположное.

Посмотрите, превосходит ли этот результат все рассмотренные до этого варианты.

Если применить этот алгоритм к такой простой игре, как крестики-нолики, мы получим непобедимого противника. Вы можете сыграть против него и убедиться в этом сами: используйте файл *play_ttt.py* на GitHub (**mng.bz/gYPe**). Теоретически этот алгоритм также подойдет для шахмат, го или любой другой детерминированной игры с полной информацией. На практике он является слишком медленным для любой из этих игр.

4.4. Сокращение пространства поиска путем редукции

В примере с игрой в крестики-нолики мы просчитали все возможные ходы для нахождения идеальной стратегии. Количество возможных партий в крестики-нолики не превышает 300 000 – ничего сложного для современного компьютера. Можно ли применить эту же технику к более интересным играм? Например, в шашках существует около 500 миллиардов миллиардов (5 с 20 нулями) возможных игровых состояний. Технически возможно просчитать их все с помощью группы современных компьютеров, но на это уйдут годы. Количество возможных состояний доски в шахматах и го превышает количество атомов во вселенной (на что часто указывают поклонники этих игр). О том, чтобы все их просчитать, не может быть и речи.

Чтобы использовать поиск по дереву для выбора ходов в сложной игре, требуется стратегия избавления от частей дерева. Определение того, какие части дерева можно пропустить, называется *редукцией*.

Игровые деревья двумерны: они имеют ширину и глубину. *Ширина* – это количество возможных ходов при данном состоянии доски. *Глубина* – это количество ходов от текущего состояния доски до финального игрового состояния, т. е. возможного окончания партии. В процессе игры обе эти величины варьируются от хода к ходу.

Обычно мы оцениваем размер дерева, исходя из типичной для конкретной игры ширины и глубины. Примерное количество состояний доски в игровом дереве определяется по формуле W^d, где W – средняя ширина, а d – средняя глубина. На рис. 4.6 и 4.7 продемонстрированы ширина и глубина дерева для игры в крестики-нолики. Например, в шахматах у игрока обычно есть около 30 вариантов каждого хода, а продолжительность партии составляет примерно 80 ходов, поэтому дерево предусматривает около $30^{80} \approx 10^{118}$ состояний. В го количество допустимых ходов составляет около 250, а партия может длиться до 150 ходов, поэтому дерево предусматривает примерно $250^{150} \approx 10^{359}$ состояний.

Формула W^d является примером описания экспоненциального роста: количество рассматриваемых позиций быстро увеличивается по мере увеличения глубины поиска. Представьте игровое дерево со средней шириной и глубиной около 10. Оно будет содержать 10^{10}, или 10 млрд, состояний доски.

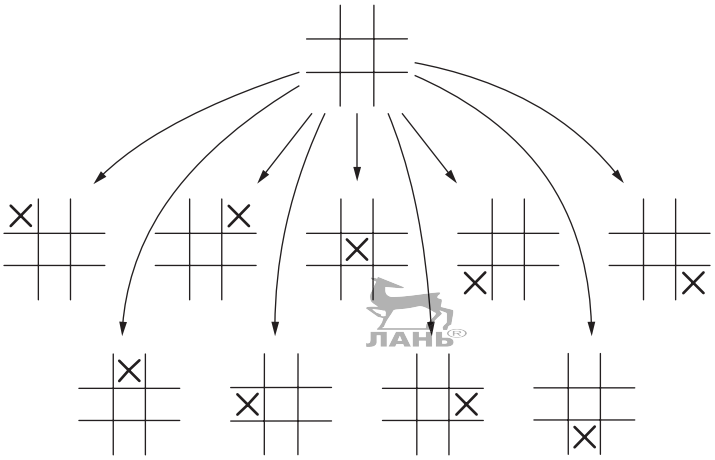


Рис. 4.6 ❖ Ширина дерева для игры в крестики-нолики: максимальная ширина равна 9, поскольку у нас есть 9 возможных вариантов первого хода. Однако количество допустимых ходов с каждым ходом уменьшается, поэтому средняя ширина составляет 4–5 ходов

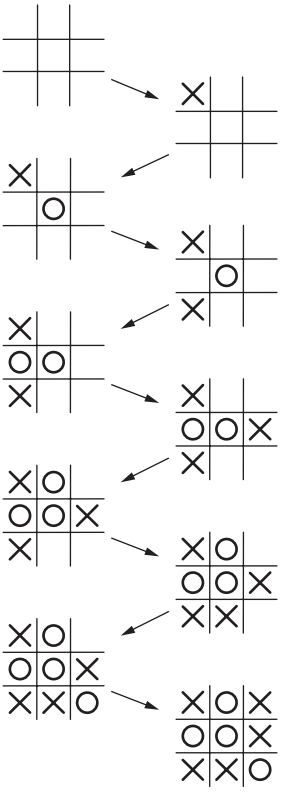


Рис. 4.7 ❖ Глубина дерева для игры в крестики-нолики: максимальная глубина составляет 9 ходов; после этого доска будет заполнена

Теперь предположим, что мы придумали несколько способов редукции. Во-первых, мы выяснили, как на каждом ходу можно быстро исключить из рассмотрения два хода, уменьшив ширину дерева до 8. Во-вторых, мы решили, что предсказать результат игры можно путем прогнозирования на 9 ходов вперед вместо 10. Это сокращает пространство поиска до 8^9, т. е. примерно до 130 млн состояний. Таким образом, мы исключили более 98 % вычислений! Ключевой вывод заключается в том, что даже незначительное уменьшение ширины или глубины дерева может значительно сократить время, необходимое для выбора хода. На рис. 4.8 проиллюстрирован результат редукции небольшого дерева.

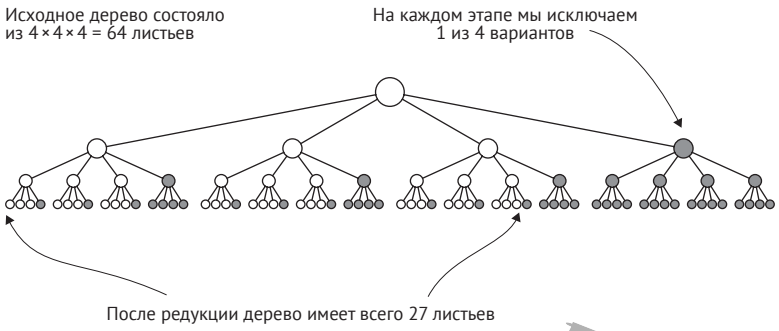


Рис. 4.8 ❖ Редукция позволяет быстро уменьшить игровое дерево. Данное дерево имеет ширину 4 и высоту 3, что в общей сложности дает 64 листа для рассмотрения. Предположим, мы нашли способ исключить 1 из 4 возможных вариантов на каждом ходу. Тогда у нас останется всего 27 листьев

В этом разделе мы рассмотрим два метода редукции: *функции оценки позиции* для уменьшения глубины поиска и *альфа-бета-отсечение* для уменьшения его ширины. Сочетание этих методов лежит в основе классического ИИ для настольной игры.

4.4.1. Сокращение глубины поиска с помощью оценки позиции

Если проанализировать все игровое дерево до самого конца партии, можно определить победителя. А как это сделать на более ранних этапах игры? Игроки-люди обычно чувствуют, кто из них лидирует на протяжении всей партии. Даже новички в го инстинктивно чувствуют, теснят ли они своего противника или сами пытаются отыграться. Реализация этого ощущения в компьютерной программе позволяет уменьшить глубину поиска. Для определения лидера в игре используется *функция оценки позиции*.

Для многих игр такую функцию можно создать на основании знаний об игре. Например:

- **шашки** – подсчитайте общее количество очков: одно очко за каждую обычную шашку на доске, плюс два очка за каждую дамку. Вычтите из своего количества очков количество очков противника;

- **шахматы** – подсчитайте общее количество очков: одно очко за каждую пешку, три очка за каждого слона или коня, пять очков за ладью и девять очков за ферзя. Вычтите из своего количества очков количество очков противника.

Эти оценочные функции являются очень упрощенными: лучшие шашечные и шахматные движки используют более сложные эвристические методы. Однако в обоих случаях ИИ будет стремиться захватить фигуры противника и сохранить собственные. Кроме того, он будет готов пожертвовать своими более слабыми фигурами, чтобы захватить более сильные фигуры противника.

В игре го эквивалентный эвристический способ предполагает подсчет количества захваченных игроком камней, из которого затем вычитается количество камней, захваченных противником. (Также можно посчитать разницу в количестве камней на доске.) Этот метод реализован в листинге 4.8. Однако данная функция не является особенно эффективной. В го *угроза* захвата камней оказывается гораздо более важной, чем их *фактический* захват. Очень часто первый захват камней происходит более чем через 100 ходов после начала партии. Создание функции оценки состояния доски, позволяющей точно отразить нюансы игрового состояния, оказывается чрезвычайно трудной задачей.

Тем не менее этот простой эвристический метод можно использовать для иллюстрации методов редукции. Он не позволит создать сильного бота, но это лучше, чем выбирать ходы случайным образом. В главах 11 и 12 мы поговорим об использовании глубокого обучения для создания более эффективной оценочной функции.

После выбора функции оценки можно приступить к сокращению *глубины*. Вместо рассмотрения всех ходов до конца партии с целью определения победителя можно исследовать фиксированное количество ходов и использовать функцию оценки для определения вероятного победителя.

Листинг 4.8 ❖ Очень упрощенный эвристический метод оценки доски для игры в го

```
def capture_diff(game_state):
    black_stones = 0
    white_stones = 0
    for r in range(1, game_state.board.num_rows + 1):
        for c in range(1, game_state.board.num_cols + 1):
            p = gotypes.Point(r, c)
            color = game_state.board.get(p)
            if color == gotypes.Player.black:
                black_stones += 1
            elif color == gotypes.Player.white:
                white_stones += 1
    diff = black_stones - white_stones
    if game_state.next_player == gotypes.Player.black:
        return diff
    return -1 * diff
```

Расчет разницы между количеством черных и белых камней на доске. Результат будет совпадать с разницей в количестве захваченных камней, если ни один из игроков не пасовал на более ранних этапах игры.

Если очередь хода за черными, возвращается результат вычитания: (черные камни) – (белые камни).

Если очередь хода за белыми, возвращается результат вычитания: (белые камни) – (черные камни).

На рис. 4.9 показана часть игрового дерева после сокращения глубины. (Мы исключили из диаграммы большую часть ветвей, чтобы сэкономить место, однако алгоритм предполагает их изучение.)

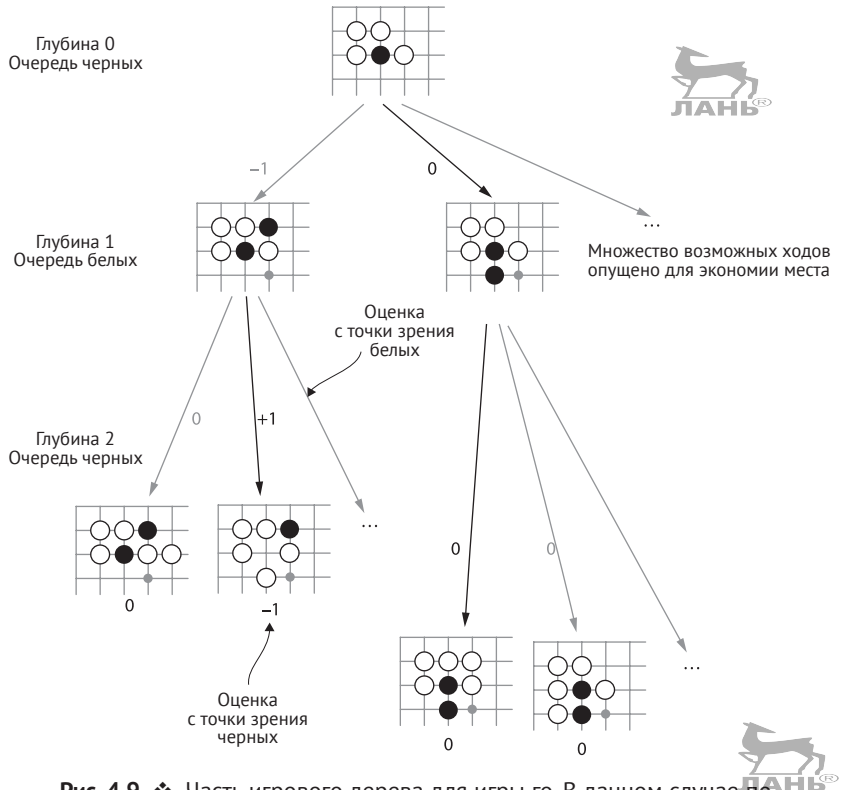


Рис. 4.9 ❖ Часть игрового дерева для игры го. В данном случае поиск по дереву осуществляется в глубину на 2 хода вперед. В этой точке мы проверяем количество захваченных камней для оценки состояния доски. Если игрок черными выберет крайнюю правую ветвь, игрок белыми сможет захватить камень, что дает оценку −1 для игрока черными. Если игрок черными выберет центральную ветвь, черный камень окажется в безопасности (на время). Эта ветвь оценивается в 0 баллов. Таким образом, игрок черными выберет центральную ветвь

В данном случае мы выполняем поиск по дереву в глубину на 2 хода вперед и используем количество захваченных камней в качестве функции оценки состояния доски. В исходной позиции очередь хода за черными. Черный камень имеет только одну степень свободы. Что делать игроку? Если он продолжит цепочку, поместив еще один черный камень непосредственно под имеющийся на доске (средняя ветвь), это обеспечит камню временную безопасность. Если игрок черными сделает любой другой ход, игрок белыми сможет захватить камень – левая ветвь отображает один из множества возможных вариантов этого развития событий.

Заглянув на два шага вперед, мы применяем к позиции свою функцию оценки. В данном случае любая ветвь, в которой игрок белыми захватывает камень, имеет оценку +1 для белых и –1 для черных. Все остальные ветви имеют оценку 0 (другого способа захватить камень за два хода нет). В данном случае игрок черными выбирает единственный ход, позволяющий защитить камень.

В листинге 4.9 показан способ реализации сокращения глубины. Этот код похож на код минимаксного алгоритма из листинга 4.7. При их сравнении можно обнаружить следующие различия:

- вместо возвращения перечисления выигрыш/проигрыш/ничья возвращается число, соответствующее значению оценочной функции. Мы договорились, что эта оценка определяется с точки зрения игрока, которому принадлежит право следующего хода: большое значение говорит о том, что этот игрок рассчитывает на победу. Для оценки доски с точки зрения противника требуется умножить полученное значение на –1;
- параметр `max_depth` контролирует глубину поиска, т. е. количество рассматриваемых ходов. На каждом ходу мы вычитаем из этого значения 1;
- когда значение `max_depth` становится равным 0, мы прекращаем поиск и вызываем функцию оценки позиции.

Листинг 4.9 ❖ Минимаксный алгоритм поиска с ограничением глубины

```
def best_result(game_state, max_depth, eval_fn):
    if game_state.is_over():
        if game_state.winner() == game_state.next_player:
            return MAX_SCORE
        else:
            return MIN_SCORE

    if max_depth == 0:
        return eval_fn(game_state)

    best_so_far = MIN_SCORE
    for candidate_move in game_state.legal_moves():
        next_state = game_state.apply_move(candidate_move)
        opponent_best_result = best_result(
            next_state, max_depth - 1, eval_fn)
        our_result = -1 * opponent_best_result
        if our_result > best_so_far:
            best_so_far = our_result

    return best_so_far
```

Если игра окончена, вам уже известен победитель.

Вы достигли максимальной глубины поиска. Используйте свой эвристический метод для оценки последовательности ходов.

Циклически обработайте все допустимые ходы.

Посмотрите, как будет выглядеть доска в случае выбора этого хода.

Определите лучший результат противника, исходя из этой позиции.

Что бы ни было нужно вашему противнику, вам нужно противоположное.

Посмотрите, превосходит ли этот результат все рассмотренные до этого варианты.

Не стесняйтесь экспериментировать со своими оценочными функциями и следить за тем, как они влияют на бота. Вы, безусловно, можете достичь гораздо большего по сравнению с описанным примером.

4.4.2. Сокращение ширины поиска путем альфа-бета-отсечения

Рассмотрим диаграмму на рис. 4.10. Очередь хода за игроком черными, и он рассматривает возможность помещения камня в точку, отмеченную квадратом. В этом случае игрок белыми сможет поместить камень в точку A, чтобы захва-

тить четыре черных камня. Очевидно, это катастрофа для черных! Что, если вместо этого игрок белыми поместит камень в точку B? Какая разница? Выбор точки A уже приводит к достаточно плохим последствиям. С точки зрения черных, не важно, является ли выбор точки A самым лучшим ходом для белых. Как только обнаруживается один мощный ответ, можно отказаться от помещения камня в точке, отмеченной квадратом, и перейти к рассмотрению следующего варианта. В этом и заключается принцип *альфа-бета-отсечения*.

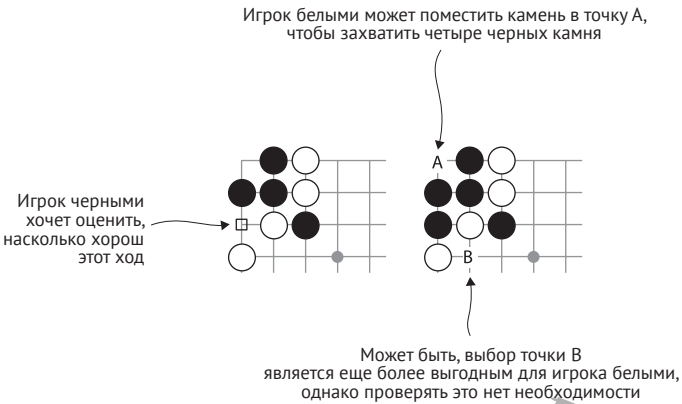


Рис. 4.10 ❖ Игрок черными рассматривает возможность помещения камня в точку, отмеченную квадратом. Если он это сделает, игрок белыми сможет поместить камень в точку A, чтобы захватить четыре черных камня. Этот результат настолько плох для игрока черными, что он может сразу же отказаться от этого варианта, и ему нет необходимости рассматривать другие ответы игрока белыми, вроде помещения камня в точку B

Давайте посмотрим, как алгоритм альфа-бета-отсечения применяется к данной позиции. Все начинается с обычного поиска по дереву с ограничением глубины. На рис. 4.11 показан первый шаг. Мы выбираем для оценки первый ход игрока черными, который отмечен на диаграмме буквой A. Затем оцениваем этот ход, выполняя поиск вглубь на 3 хода. Мы видим, что вне зависимости от ответа белых черные могут захватить как минимум два камня. Таким образом, для черных эта ветвь получает оценку +2.

Теперь мы рассмотрим следующий вариант хода черных, обозначенный буквой B на рис. 4.12. Так же, как и при выполнении поиска по дереву при ограничении глубины, мы поочередно исследуем и оцениваем все возможные ответы белых. Игрок белыми может поместить камень в верхний левый угол, чтобы захватить четыре камня, поэтому данная ветвь оценивается для игрока черными в –4. Мы уже знаем, что если игрок черными поместит камень в точку A, то он гарантированно получит как минимум +2 очка. Как было показано ранее, при помещении черного камня в точку B белые захватят четыре камня, что дает этому варианту оценку –4 для черных. Вероятно, игрок белыми мог бы получить еще больше очков. Однако поскольку оценка –4 гораздо хуже, чем +2, продолжать поиск нет необходимости. Мы можем пропустить оценку десятка других возможных ответов

игрока белыми, каждый из которых может предполагать множество вариантов дальнейшего развития событий. Это позволяет сэкономить большое количество вычислительных ресурсов, и при этом мы все равно выбираем тот же ход, который выбрали бы при выполнении полного поиска вглубь на 3 хода.

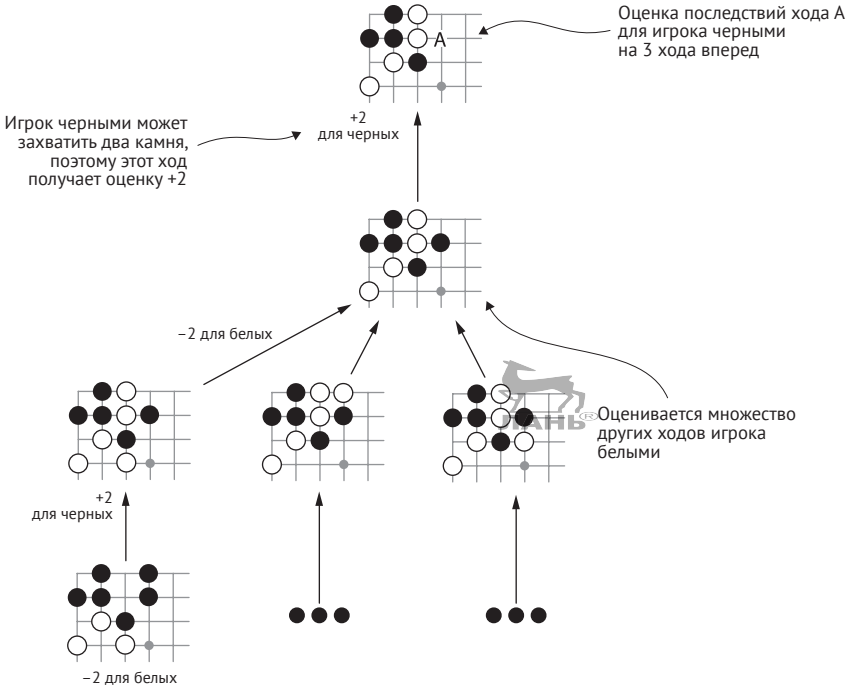


Рис. 4.11 ❖ Шаг 1 процесса альфа-бета-отсечения: мы полностью оцениваем первый из возможных ходов игрока черными, для которого он оценивается в +2 балла. Пока этот алгоритм ничем не отличается от описанного в предыдущем разделе поиска по дереву с ограничением глубины

В этом примере мы выбрали определенный порядок действий для оценки ходов, чтобы проиллюстрировать принцип редукции. Фактическая реализация этого алгоритма оценивает ходы в порядке, определяемом их координатами на доске. Экономия времени, обеспечиваемая альфа-бета-отсечением, зависит от скорости выявления хороших ветвей. Если нам удалось выявить лучшие ветви на раннем этапе, мы можем быстро отбросить другие. В худшем случае мы оцениваем лучшую ветвь в последнюю очередь, и тогда скорость альфа-бета-отсечения не превышает скорость выполнения поиска по дереву без ограничения глубины.

Для реализации этого алгоритма на протяжении всего процесса поиска необходимо отслеживать лучший результат для каждого игрока. Эти значения традиционно называются *альфа* и *бета*, отсюда и название алгоритма. В случае нашей реализации мы называем эти значения `best_black` и `best_white`.

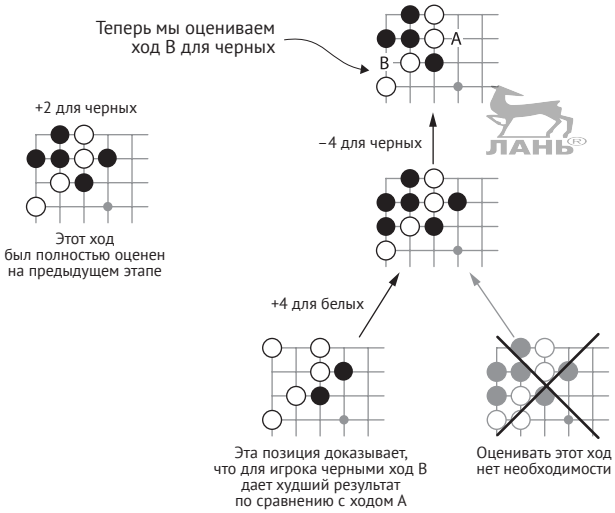


Рис. 4.12 ❖ Шаг 2 процесса альфа-бета-отсечения: теперь мы оцениваем второй вариант хода черных. В данном случае у игрока белыми есть возможность захватить четыре камня. Поэтому эта ветвь получает оценку –4 для черных. После оценки такого ответа игрока белыми мы можем полностью отказаться от этого варианта хода черных и не исследовать другие варианты ответов белых. Возможно, у белых есть лучший вариант, который мы не учли, однако нам достаточно того, что выбор точки B обеспечивает худший результат для игрока черными по сравнению с выбором точки A

Листинг 4.10 ❖ Проверка на предмет возможности прекращения оценки ветви

```
def alpha_beta_result(game_state, max_depth,
                      best_black, best_white, eval_fn):
...
        if game_state.next_player == Player.white:
            # Update our benchmark for white.
            if best_so_far > best_white:
                best_white = best_so_far
            outcome_for_black = -1 * best_so_far
            if outcome_for_black < best_black:
                return best_so_far
```

Обновление лучшего результата для игрока белыми.

Выбор хода для белых, который должен быть достаточно сильным, чтобы перебить предыдущий ход черных. После нахождения варианта, превосходящего лучший ход черных, поиск можно прекратить.

Мы можем объединить сокращение глубины поиска с альфа-бета-отсечением. В листинге 4.10 показаны ключевые дополнения. Этот блок реализован с точки зрения игрока белыми, поэтому нам потребуется аналогичный блок для игрока черными.

Сначала мы проверяем, требуется ли обновить оценку `best_white`. Затем проверяем, можно ли прекратить оценку ходов для игрока белыми. Для этого мы сравниваем текущий результат с лучшим результатом, обнаруженным для игрока черными в *любой* ветви. Если ход белых может привести к ухудшению положения

черных, то игрок черными не выберет эту ветвь. Нам нет необходимости искать самый лучший результат.

Полная реализация алгоритма альфа-бета-отсечения приведена в следующем листинге.

Листинг 4.11 ❖ Полная реализация алгоритма альфа-бета-отсечения

```
def alpha_beta_result(game_state, max_depth,
                      best_black, best_white, eval_fn):
    if game_state.is_over():
        if game_state.winner() == game_state.next_player:
            return MAX_SCORE
        else:
            return MIN_SCORE

    if max_depth == 0:
        return eval_fn(game_state)

    best_so_far = MIN_SCORE
    for candidate_move in game_state.legal_moves():
        next_state = game_state.apply_move(candidate_move)
        opponent_best_result = alpha_beta_result(
            next_state, max_depth - 1,
            best_black, best_white,
            eval_fn)
        our_result = -1 * opponent_best_result

        if our_result > best_so_far:
            best_so_far = our_result
        if game_state.next_player == Player.white:
            if best_so_far > best_white:
                best_white = best_so_far
            outcome_for_black = -1 * best_so_far
            if outcome_for_black < best_black:
                return best_so_far
        elif game_state.next_player == Player.black:
            if best_so_far > best_black:
                best_black = best_so_far
            outcome_for_white = -1 * best_so_far
            if outcome_for_white < best_white:
                return best_so_far

    return best_so_far
```

Выполните проверку на предмет окончания игры.

Вы достигли максимальной глубины поиска. Используйте свой эвристический метод для оценки последовательности ходов.

Циклически обработайте все допустимые ходы.

Посмотрите, как будет выглядеть доска в случае выбора этого хода.

Определите лучший результат противника, исходя из этой позиции.

Что бы ни было нужно вашему противнику, вам нужно противоположное.

Посмотрите, превосходит ли этот результат все рассмотренные до этого варианты.

Обновите лучший результат для игрока белыми.

Выберите ход для белых, который должен быть достаточно сильным, чтобы перебить предыдущий ход черных.

Обновите лучший результат для игрока черными.

Выберите ход для черных, который должен быть достаточно сильным, чтобы перебить предыдущий ход белых.

4.5. Оценка игрового состояния методом Монте-Карло

При использовании альфа-бета-отсечения мы применили функцию оценки позиции для сокращения количества рассматриваемых игровых состояний. Однако оценить позицию в игре го очень и очень трудно: наш простой эвристический метод, основанный на захвате камней, мало кого способен обмануть. *Поиск по дереву методом Монте-Карло* (ММК) позволяет оценить игровое состояние без каких-либо знаний о стратегии игры. Вместо применения специфического эвристического метода алгоритм ММК симулирует случайные игры для оценки конкретного

игрового состояния. Одна из этих случайных игр называется *развертыванием*, или *разыгрыванием*. В этой книге мы будем использовать термин *развертывание*.

Поиск по дереву методом Монте-Карло является частью большого семейства *алгоритмов Монте-Карло*, которые для анализа чрезвычайно сложных ситуаций используют элемент случайности, благодаря которому это семейство методов и получило свое название, намекающее на знаменитый район с казино в Монако.

То, что хорошую стратегию можно разработать на основе выбранных случайным образом ходов, может показаться невероятным. Разумеется, игровой ИИ, выбирающий совершенно случайные ходы, будет крайне слабым. Однако в случае игры таких ИИ друг с другом оба противника являются одинаково невежественными. Если черные выигрывают чаще, чем белые, это должно свидетельствовать об их изначальном преимуществе. Таким образом, мы можем выяснить, предоставляет ли позиция преимущество одному из игроков, начиная из нее случайные игры. И для этого нам необязательно понимать, *почему* эта позиция является хорошей.

Несбалансированные результаты можно получить совершенно случайно. Если в ходе 10 случайных игр белые победили семь раз, можно ли уверенно утверждать о том, что белые имели преимущество? Нет, белые выиграли всего на две игры больше, чем можно было бы ожидать. При идеальной сбалансированности черных и белых вероятность получения результата 7 из 10 составляет 30 %. С другой стороны, если бы белые победили в 70 случайных играх из 100, можно было бы практически с полной уверенностью предположить, что стартовая позиция предоставляла белым преимущество. Основная идея заключается в том, что точность оценки растет по мере увеличения количества развертываний.

Каждый раунд алгоритма ММК состоит из трех этапов:

1) добавление в дерево новой игровой позиции;
2) симуляция случайной игры из этой позиции;
3) обновление статистики дерева с учетом результатов этой случайной игры.

Мы повторяем данный процесс максимальное количество раз, исходя из доступного времени. Затем на основе статистики в вершине дерева выбираем предпочтительный ход.

Давайте рассмотрим один раунд алгоритма Монте-Карло. На рис. 4.13 показано дерево поиска. К этому моменту мы уже выполнили ряд развертываний и создали частичное дерево. Каждый узел отслеживает количество побед игроков в играх, начатых из любой игровой позиции после этого узла. Значение каждого узла включает сумму всех его дочерних элементов. (Обычно на данном этапе у дерева будет намного больше узлов; для экономии места мы исключили из диаграммы многие из них.)

В каждом раунде в дерево добавляется новая игровая позиция. Сначала мы выбираем узел в нижней части дерева (*лист*), куда хотим добавить дочерний узел. Данное дерево имеет пять листьев. Для получения наилучших результатов к выбору листа следует подходить осторожно (хорошая стратегия описана в разделе 4.5.2). Пока просто представим, что мы прошли весь путь по крайней левой ветви. Из этой точки мы случайным образом выбираем следующий ход, вычисляем новое состояние доски и добавляем в дерево соответствующий узел. Результат показан на рис. 4.14.

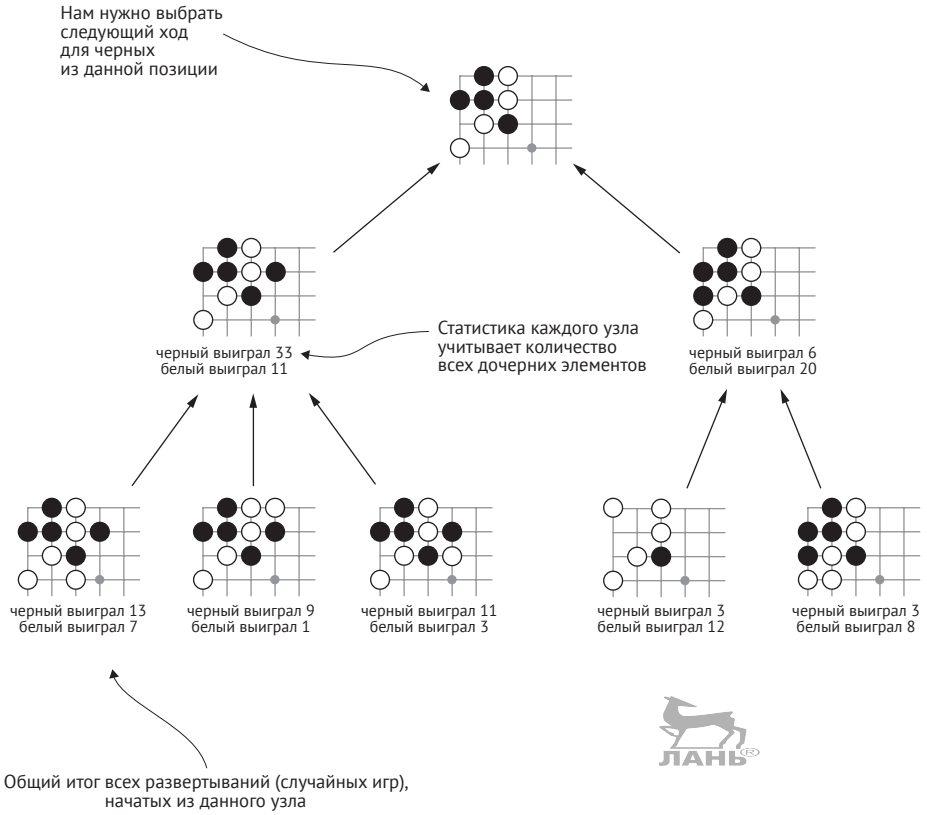


Рис. 4.13 ❖ Игровое дерево MMK. Вершина дерева соответствует текущему состоянию доски. Наша задача состоит в выборе следующего хода для черных. К данному моменту мы уже выполнили 70 случайных развертываний из различных возможных позиций. Каждый узел отслеживает статистику по всем развертываниям, начатым из любого дочернего элемента

Новый узел в дереве является отправной точкой для случайной игры. Мы симулируем остальную часть игры, просто выбирая любой допустимый вариант на каждом ходу вплоть до окончания игры. Затем подсчитываем количество очков и определяем победителя. Предположим, что в данном случае победителем является игрок белыми. Мы записываем результат этого развертывания в новом узле. Кроме того, проходим по всем предкам этого узла, добавляя результат нового развертывания в их значение. На рис. 4.15 показано, как выглядит дерево после выполнения этого шага.

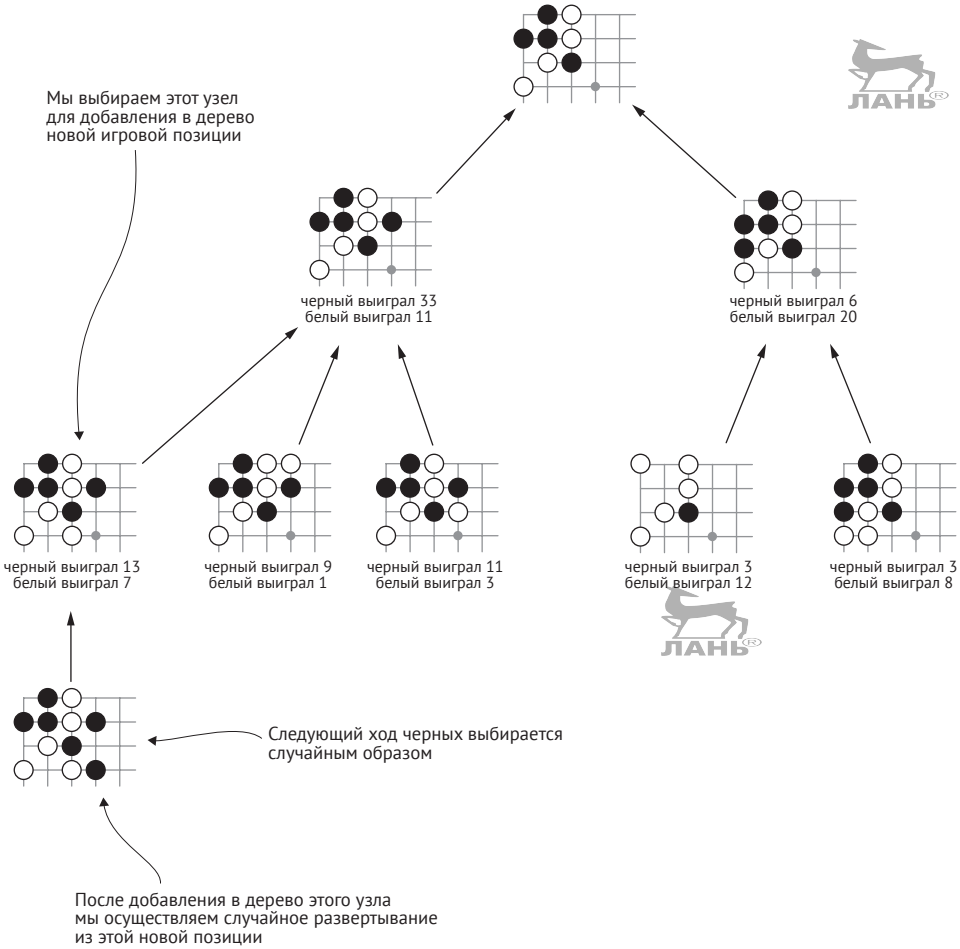


Рис. 4.14 ❖ Добавление нового узла в игровое дерево ММК. В данном случае для его вставки мы выбираем крайнюю левую ветвь. Затем из этой позиции мы случайным образом выбираем следующий ход, чтобы создать новый узел в дереве

Описанный процесс представляет собой один раунд алгоритма Монте-Карло. При каждом его повторе дерево увеличивается, а оценки в его вершине становятся все более точными. Обычно процесс прекращается после определенного количества раундов или по истечении определенного периода времени. В этот момент мы выбираем ход с наибольшим процентом выигрышей.

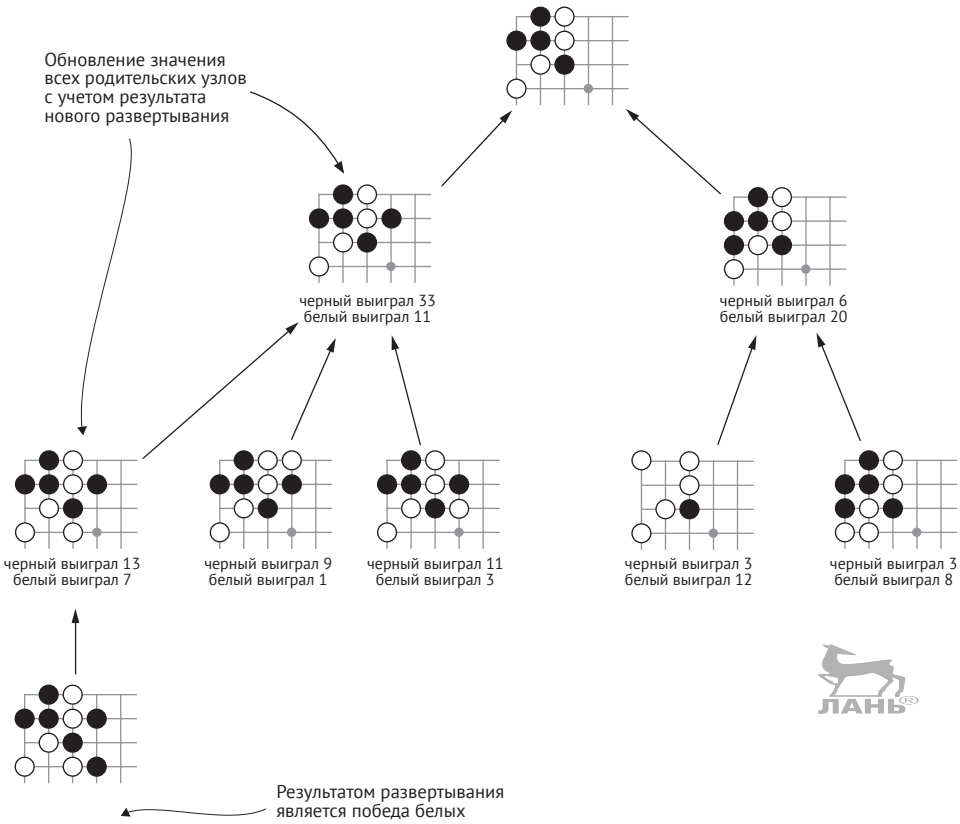


Рис. 4.15 ❖ Обновление игрового дерева ММК после нового развертывания. В данном сценарии результатом развертывания является победа белых. Мы добавляем эту победу в новый узел дерева и во все его родительские узлы

4.5.1. Реализация алгоритма Монте-Карло средствами языка Python

Теперь, когда мы разобрались с принципом работы алгоритма Монте-Карло, поговорим о его реализации. Для начала необходимо разработать структуру данных для представления дерева. Затем нужно будет написать функцию для выполнения развертываний ММК.

Как показано в листинге 4.12, мы начинаем с определения нового класса `MCTSNode`, который будет представлять любой узел нашего дерева. Каждый такой узел будет отслеживать следующие свойства:

- `game_state` – текущее игровое состояние (позиция на доске и очередь хода) в данном узле дерева;
- `parent` – непосредственный родитель данного узла. Чтобы указать на корень дерева, можно задать для данного свойства значение `None`;
- `move` – последний ход, непосредственно приведший к данному узлу;
- `children` – список всех дочерних узлов в дереве;

- `win_counts` и `num_rollouts` – статистика развертываний, выполненных из данного узла;
- `unvisited_moves` – список всех допустимых ходов из этой позиции, которые еще не являются частью дерева. Всякий раз при добавлении нового узла в дерево мы извлекаем один ход из `unvisited_moves`, генерируем для него новый `MCTSNode` и добавляем его в список `children`.

Листинг 4.12 ❖ Структура данных для представления дерева

```
class MCTSNode(object):
    def __init__(self, game_state, parent=None, move=None):
        self.game_state = game_state
        self.parent = parent
        self.move = move
        self.win_counts = {
            Player.black: 0,
            Player.white: 0,
        }
        self.num_rollouts = 0
        self.children = []
        self.unvisited_moves = game_state.legal_moves()
```

Узел `MCTSNode` можно изменить двумя способами – добавить в дерево новый дочерний элемент или обновить статистику развертываний. В следующем листинге продемонстрированы оба подхода.

Листинг 4.13 ❖ Методы обновления узла дерева

```
    def add_random_child(self):
        index = random.randint(0, len(self.unvisited_moves) - 1)
        new_move = self.unvisited_moves.pop(index)
        new_game_state = self.game_state.apply_move(new_move)
        new_node = MCTSNode(new_game_state, self, new_move)
        self.children.append(new_node)
        return new_node

    def record_win(self, winner):
        self.win_counts[winner] += 1
        self.num_rollouts += 1
```

Наконец, мы добавляем три вспомогательных метода для получения доступа к полезным свойствам узла дерева:

- `can_add_child` сообщает, предусматривает ли данная позиция допустимые ходы, которые еще не были добавлены в дерево;
- `is_terminal` сообщает, заканчивается ли игра в данном узле. Если это так, дальнейший поиск невозможен;
- `victory_frac` возвращает процент выигрышных развертываний для данного игрока.

Эти функции реализованы в следующем листинге.

Листинг 4.14 ❖ Вспомогательные методы для получения доступа к полезным свойствам дерева

```
    def can_add_child(self):
        return len(self.unvisited_moves) > 0
```

```
    def is_terminal(self):
        return self.game_state.is_over()

    def winning_frac(self, player):
        return float(self.win_counts[player]) / float(self.num_rollouts)
```

После определения структуры данных для дерева можно приступать к реализации алгоритма Монте-Карло. Мы начинаем с создания нового дерева. Корневой узел – это текущее игровое состояние. Затем мы генерируем несколько развертываний. В данной реализации обрабатываем каждый ход фиксированное количество раз с помощью цикла. В других реализациях вместо этого можно ограничить время, выделенное на обработку.

Каждый раунд начинается с обхода дерева вплоть до нахождения узла, к которому можно добавить дочерний элемент (это любая позиция на доске, предусматривающая допустимый ход, которого еще нет в дереве). Функция `select_move` отвечает за выбор лучшей ветви для исследования. Подробнее о том, как это происходит, мы поговорим в следующем разделе.

После нахождения подходящего узла мы вызываем `add_random_child` для выбора и включения в дерево любого из последующих ходов. На этом этапе `node` представляет собой вновь созданный `MCTSNode` с нулевым количеством развертываний.

Теперь мы осуществляем развертывание из этого узла путем вызова `simulate_random_game`. Реализация `simulate_random_game` идентична примеру `bot_v_bot`, описанному в главе 3.

Наконец, мы обновляем значение счетчика выигрышей вновь созданного узла и всех его предков. Весь этот процесс реализован в следующем листинге.

Листинг 4.15 ❖ Алгоритм Монте-Карло

```
class MCTSAgent(agent.Agent):
    def select_move(self, game_state):
        root = MCTSNode(game_state)

        for i in range(self.num_rounds):
            node = root
            while (not node.can_add_child()) and (not node.is_terminal()):
                node = self.select_child(node)

            if node.can_add_child():
                node = node.add_random_child()

            winner = self.simulate_random_game(node.game_state)

            while node is not None:
                node.record_win(winner)
                node = node.parent
```

Добавление в дерево нового дочернего узла.

Развертывание случайной игры из этого узла.

Обновление счета в предыдущих узлах дерева.

После завершения заданного количества раундов мы должны выбрать ход. Для этого мы просто перебираем все ветви верхнего уровня и выбираем ту, которая имеет наибольший процент выигрышей. В следующем листинге показано, как это можно реализовать.

Листинг 4.16 ❖ Выбор хода после завершения развертываний

```
class MCTSAgent:
...
    def select_move(self, game_state):
```

```
...
        best_move = None
        best_pct = -1.0
        for child in root.children:
            child_pct = child.winning_pct(game_state.next_player)
            if child_pct > best_pct:
                best_pct = child_pct
                best_move = child.move
        return best_move
```

4.5.2. Выбор ветви для исследования

Наш игровой ИИ может потратить на каждый ход ограниченное количество времени. Это означает, что мы можем осуществить лишь фиксированное количество развертываний. Каждое из них улучшает нашу оценку одного из возможных ходов. Думайте о развертываниях как об ограниченном ресурсе: если мы выполним лишнее развертывание при анализе хода A, то нам придется ограничить количество развертываний при анализе хода B. Поэтому нам нужна стратегия оптимального использования своего ограниченного ресурса. Стандартная стратегия называется *верхний предел доверительного интервала для деревьев*, или формула *UCT*. Формула UCT обеспечивает баланс между двумя конфликтующими целями.

Первая цель заключается в том, чтобы тратить время на изучение лучших ходов. Это называется *эксплуатацией*, поскольку мы хотим воспользоваться любым обнаруженным преимуществом. Мы готовы потратить больше развертываний на ходы с самым высоким предполагаемым процентом выигрышей. Некоторые из этих ходов могут иметь высокий процент выигрышей по чистой случайности. Однако по мере осуществления все большего количества развертываний в этих ветвях точность оценки увеличивается, а ложноположительные результаты смещаются вниз по списку.

С другой стороны, если вы посетили узел лишь несколько раз, то ваша оценка может оказаться слишком неточной. По чистой случайности очень хороший ход может получить низкую оценку. Несколько дополнительных развертываний могут выявить его истинную ценность. Таким образом, вашей второй целью является максимально точная оценка наименее посещаемых ветвей. Это называется *разведкой*.

На рис. 4.16 приведено сравнение поиска по дереву со смещением в сторону эксплуатации и со смещением в сторону разведки. Компромисс между эксплуатацией и разведкой свойствен алгоритмам, основанным на методе проб и ошибок. Мы вернемся к этой теме, когда будем говорить об обучении с подкреплением.

Для каждого рассматриваемого узла мы вычисляем процент выигрышей w, представляющий цель эксплуатации. Для представления цели разведки производим вычисления по нижеуказанной формуле, где N – общее количество развертываний, а n – количество развертываний, начатых с рассматриваемого узла. Данная формула имеет теоретическую основу, однако для наших целей достаточно учесть, что ее значение будет наибольшим для наименее посещаемых узлов.

$$\sqrt{\frac{\log N}{n}}.$$

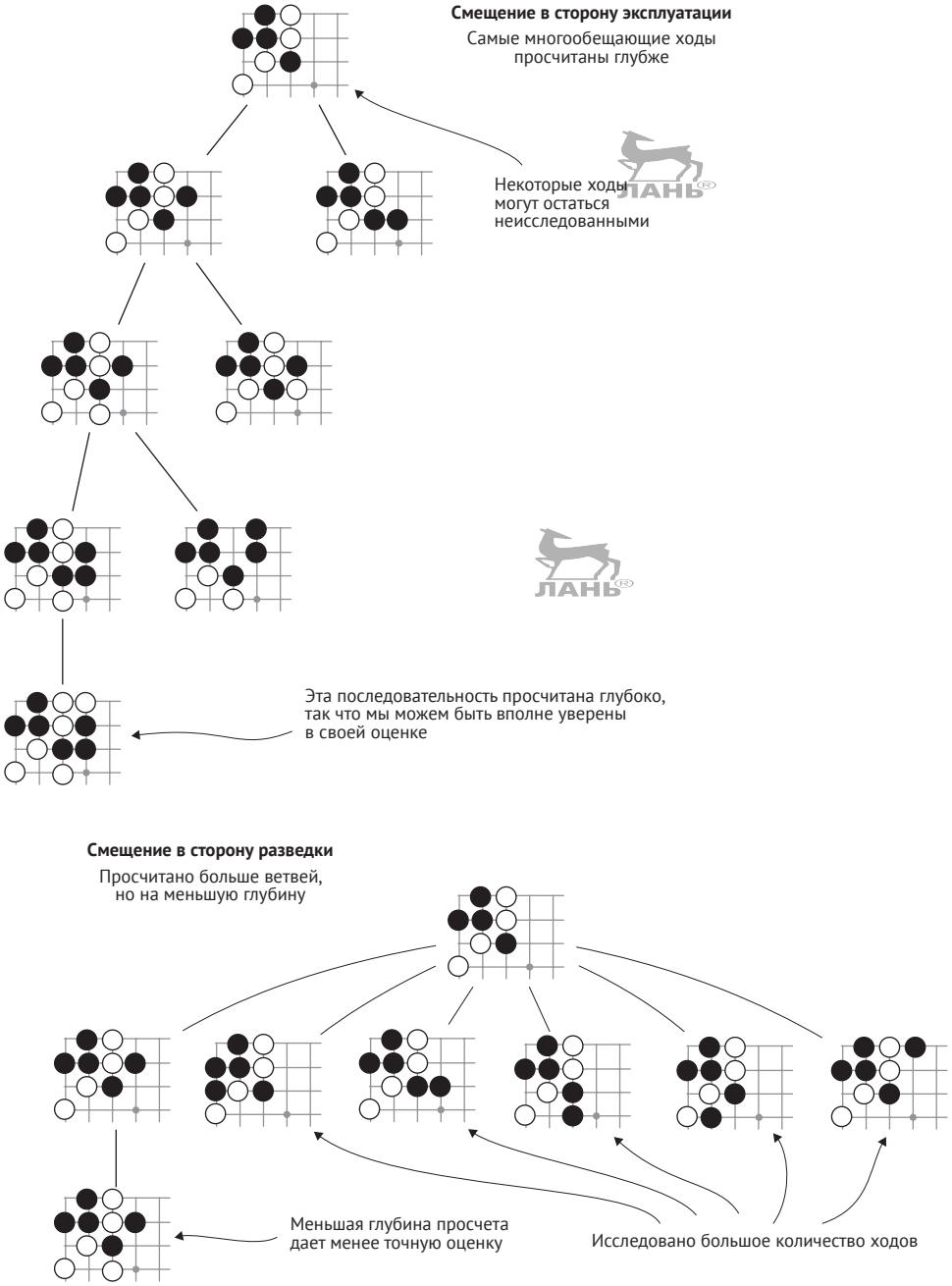


Рис. 4.16 ❖ Компромисс между эксплуатацией и разведкой. В обоих игровых деревьях мы исследовали семь состояний доски. В верхнем примере поиск смещен в сторону эксплуатации: самые многообещающие ходы просчитаны глубже. В нижнем примере поиск смещен в сторону разведки: исследовано больше ходов, но на меньшую глубину

Объединив эти два компонента, мы получим формулу UCT:

$$w + c\sqrt{\frac{\log N}{n}}.$$

Здесь c – это параметр, определяющий предпочтительный баланс между эксплуатацией и разведкой. Формула UCT позволяет дать оценку каждому узлу, а узел с наибольшей оценкой будет являться стартовой точкой для следующего развертывания.

При большем значении c мы будем тратить больше времени на посещение наименее исследованных узлов. При меньшем значении c мы сосредоточимся на получении точной оценки наиболее перспективного узла. Значение c, обеспечивающее наиболее эффективный результат, обычно определяется методом проб и ошибок. Мы предлагаем начать экспериментировать со значения 1,5. Иногда параметр c называется *температурой*. При «горячей» температуре поиск будет более разбросанным, а при «холодной» – более сфокусированным.

В листинге 4.17 продемонстрирована реализация данной политики. После определения подходящей метрики выбор дочернего элемента сводится к вычислению формулы для каждого узла и выбору узла с наибольшим значением. Как и в случае с минимаксным алгоритмом, нам необходимо переключать перспективу на каждом ходу. Процент выигрышей рассчитывается с точки зрения игрока, которому предстоит выбрать следующий ход, чтобы перспектива чередовалась между черными и белыми при продвижении по дереву.

Листинг 4.17 ❖ Выбор ветви для исследования с помощью формулы UCT

```
def uct_score(parent_rollouts, child_rollouts, win_pct, temperature):
    exploration = math.sqrt(math.log(parent_rollouts) / child_rollouts)
    return win_pct + temperature * exploration

class MCTSAgent:
...
    def select_child(self, node):
        total_rollouts = sum(child.num_rollouts for child in node.children)

        best_score = -1
        best_child = None
        for child in node.children:
            score = uct_score(
                total_rollouts,
                child.num_rollouts,
                child.winning_pct(node.game_state.next_player),
                self.temperature)
            if score > best_score:
                best_score = uct_score
                best_child = child
        return best_child
```

4.5.3. Применение алгоритма Монте-Карло к игре го

В предыдущем разделе мы рассмотрели реализацию общей формы алгоритма Монте-Карло. На основе таких простых реализаций можно создать бота для игры в го уровня сильного любителя (1 дан). Объединение алгоритма Монте-Карло

с другими методами позволяет создать гораздо более сильного бота. Многие из лучших современных ИИ для игры в го используют как метод Монте-Карло, так и глубокое обучение. В этом разделе мы обсудим некоторые практические детали создания конкурентоспособного бота для игры в го.

Быстрый код – сильный бот

Поиск по дереву методом Монте-Карло оправдывает себя при полноразмерной доске (19×19) и примерно 10 000 развертываний на ход. Описанная в этой главе реализация является недостаточно быстрой – на выбор каждого хода ей требуется несколько минут. Для выполнения такого количества развертываний в разумные сроки вам нужно будет немного оптимизировать свою реализацию. С другой стороны, при игре на маленьких досках даже бот, основанный на простейшей реализации, может оказаться достойным противником.

При прочих равных условиях большее количество развертываний позволяет принять лучшее решение. Вы всегда можете сделать своего бота сильнее, просто ускорив код, чтобы сократить количество развертываний, выполняемых за один и тот же период времени. И речь не только о коде алгоритма Монте-Карло. Например, код, вычисляющий захваты, вызывается сотни раз за одно развертывание. Оптимизации можно подвергнуть всю базовую игровую логику.

Лучшие политики развертывания обеспечивают лучшие оценки

Алгоритм выбора ходов в процессе случайного развертывания называется *политикой развертывания*. Чем более реалистичной является политика развертывания, тем точнее будут оценки. В главе 3 мы реализовали `RandomAgent` для игры в го. В этой главе мы использовали `RandomAgent` в качестве политики развертывания. Однако то, что `RandomAgent` выбирает ходы *совершенно* случайным образом без какого-либо знания об игре го, не совсем верно. Во-первых, он запрограммирован так, чтобы не пасовать и не выходить из игры вплоть до заполнения доски. Во-вторых, его программа не позволяет ему заполнять свои глаза, чтобы не убивать собственные камни в конце игры. Без этой логики результаты развертываний были бы менее точными.

Некоторые реализации алгоритма Монте-Карло идут дальше и учитывают в своей политике развертывания больше логики, относящейся к игре в го. Развертывания со специфической игровой логикой иногда называются *тяжелыми* развертываниями, в то время как развертывания, наиболее близкие к чисто случайным, иногда называются *легкими* развертываниями. Одним из способов реализации тяжелых развертываний является составление списка основных тактических форм, часто встречающихся в игре го наряду с известным ответным ходом. При обнаружении на доске известной формы мы смотрим известный ответ и повышаем вероятность его выбора. Мы не хотим *всегда* выбирать известный ответ, поскольку это устранило бы из алгоритма жизненно важный элемент случайности.

На рис. 4.17 приведен пример локальной схемы 3×3, в которой черный камень может быть захвачен при следующем ходе белых. Игрок черными может спасти его, хотя бы временно, путем наращивания цепочки. Этот ход не всегда является наилучшим, да и просто хорошим. Однако, скорее всего, он окажется предпочтительным по сравнению с выбором любой случайной точки на доске.

Составление хорошего списка форм требует знания тактических основ игры в го. Если вы хотите узнать о других тактических схемах, которые можно исполь-

зовать в рамках тяжелых развертываний, мы рекомендуем обратиться к исходному коду Fuego (**fuego.sourceforge.net**) или Pachi (**github.com/pasky/pachi**), двух движков для игры в го на основе ММК с открытым исходным кодом.

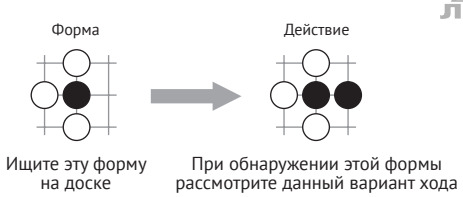


Рис. 4.17 ❖ Пример локальной схемы расположения камней. Когда вы видите форму слева, вам следует рассмотреть ответный ход, представленный справа. Политика, предполагающая подобный ответ, будет не самой сильной, но намного превзойдет политику совершенно случайного выбора ходов

Будьте осторожны при реализации тяжелых развертываний. Низкая скорость вычисления логики в вашей политике не позволит выполнить достаточное количество развертываний, что в конечном итоге может свести на нет все преимущества от использования более сложной политики.

Вежливый бот знает, когда выходить из игры

Создание игрового ИИ предполагает не только разработку лучшего алгоритма, но и обеспечение интересного опыта для оппонента-человека. Часть этого интереса заключается в переживании человеком радости от победы. Первый из описанных в этой книге бот для игры в го, `RandomAgent`, способен свести с ума. После того как игрок-человек неизбежно вырывается вперед, этот бот настаивает на продолжении игры вплоть до заполнения доски. Игроку ничто не мешает в любой момент закончить игру и мысленно приписать победу себе. Но это почему-то кажется проявлением неспортивного поведения. Будет намного лучше, если бот сможет с достоинством принять поражение.

Поверх реализации простого алгоритма Монте-Карло можно легко добавить дружественную к человеку логику выхода из игры. Алгоритм ММК вычисляет примерный процент выигрышей в процессе выбора хода. В процессе совершения одного хода мы сравниваем эти числа, чтобы определить предпочтительный ход. Однако мы также можем сравнить предполагаемый процент выигрышей на разных этапах одной и той же игры. Если значения уменьшаются, значит, человек лидирует. Когда лучшему варианту соответствует достаточно низкий процент выигрышей (скажем, 10 %) можно заставить бота выйти из игры.

4.6. Резюме

- Алгоритмы поиска по дереву позволяют оценить множество возможных последовательностей решений и найти лучшую из них. Поиск по дереву используется в играх, а также при решении общих проблем оптимизации.
- Вариантом поиска по дереву, применяющимся в играх, является *минимаксный* алгоритм, который предполагает переключение между двумя игроками, преследующими противоположные цели.

- Выполнение полного минимаксного перебора вариантов целесообразно только в очень простых играх (например, крестики-нолики). Для его применения к сложным играм (вроде шахмат или го) необходимо сократить размер дерева.
- *Функция оценки позиции* позволяет определить игрока, который с наибольшей вероятностью выиграет, исходя из данного состояния доски. При наличии хорошей функции оценки позиции вам необязательно просчитывать весь путь до конца игры, чтобы принять решение. Эта стратегия называется *сокращением глубины*.
- Альфа-бета-отсечение позволяет уменьшить количество рассматриваемых вариантов на каждом ходу, что делает этот алгоритм подходящим для использования в таких сложных играх, как шахматы. Принцип альфа-бета-отсечения интуитивно понятен: если при оценке возможного хода вы выявляете хотя бы один сильный ответный ход оппонента, то можете сразу исключить данный ход из дальнейшего рассмотрения.
- При отсутствии хорошего эвристического метода для оценки позиции иногда можно использовать *поиск по дереву методом Монте-Карло*. Этот алгоритм симулирует случайные игры из конкретной позиции и определяет, кто из игроков чаще выигрывает.

Глава 5

Знакомство с нейронными сетями

В этой главе:

- основы работы с искусственными нейронными сетями;
- обучение сети распознаванию рукописных цифр;
- создание нейронных сетей путем объединения слоев;
- принцип обучения нейронных сетей на основе данных;
- реализация простой нейронной сети с нуля.

В данной главе мы поговорим о базовых понятиях, связанных с искусственными нейронными сетями (ИНС), классом алгоритмов, лежащих в основе современных методов *глубокого обучения*. Первые искусственные нейронные сети появились довольно давно – в начале 40-х годов. На разработку эффективных способов их применения в различных областях ушло много десятилетий, однако основополагающие идеи остались прежними.

ИНС вдохновлены открытиями в области нейронауки и представляют собой попытку смоделировать класс алгоритмов, принцип работы которых аналогичен нашим представлениям о работе мозга. В частности, мы используем понятие *нейронов* – базовых элементов искусственной сети. Нейроны образуют группы, называемые *слоями*. А *связанные* определенным образом слои образуют *сеть*. После подачи входных данных нейроны передают информацию от слоя к слою через соединения, *активируясь*, когда сигнал оказывается достаточно сильным. Таким образом, данные распространяются по сети, пока не достигают последнего, выходного, слоя, от которого мы получаем *предсказания*. Затем эти предсказания можно сравнить с *ожидаемым выходным значением* для вычисления *ошибки*, которую сеть использует для дальнейшего обучения и улучшения будущих предсказаний.

Несмотря на то что аналогия с архитектурой мозга иногда бывает полезной, мы не хотим ее преувеличивать. Нам многое известно, в частности, о зрительной коре мозга, однако эта аналогия порой может вводить в заблуждение и даже вредить. Мы считаем, что ИНС следует воспринимать как попытку выявления *основных принципов обучения у организмов*, точно так же, как при создании самолета мы используем принципы аэродинамики, не пытаясь скопировать птицу.

Для большей наглядности в этой главе приведен пример реализации простой нейронной сети. Мы применим эту сеть для решения проблемы *оптического рас-*

познавания символов (optical character recognition, OCR), т. е. для того, чтобы позволить компьютеру определить, какая цифра показана на изображении.

Каждое изображение в нашем наборе данных OCR состоит из пикселов, размещенных в сетке, и нам необходимо проанализировать пространственные отношения между этими пикселами, чтобы распознать цифру, которую они составляют. Как и в случае с другими настольными играми, доска для игры в го представляет собой сетку, поэтому для выбора хорошего хода необходимо учитывать пространственные отношения между размещенными на доске камнями. Вы можете подумать, что методы машинного обучения для оптического распознавания символов могут применяться к играм вроде го, и окажетесь правы. Подробнее об этом мы поговорим в главах с 6 по 8.

В этой главе используется относительно простая математика. Если вы не знакомы с основами линейной алгебры, математического анализа и теории вероятностей или хотите освежить свои знания, рекомендуем сначала прочитать приложение А. Более сложные аспекты процесса обучения нейронной сети обсуждаются в приложении Б. Если вы уже знакомы с нейронными сетями, но никогда не создавали их, мы рекомендуем сразу перейти к разделу 5.5. Если вам уже известен процесс реализации нейронных сетей, можете перейти к главе 6, в которой обсуждается их применение для прогнозирования ходов в играх, сгенерированных в главе 4.

5.1. Простой пример использования: классификация рукописных цифр

Прежде чем углубляться в подробности, давайте рассмотрим конкретный вариант использования нейронных сетей. В данной главе мы создадим приложение, способное распознавать рукописные цифры на изображении с точностью до 95 %. Для этого мы предъявим нейронной сети только значения яркости пикселов изображений. Всю релевантную информацию о структуре цифр алгоритм научится извлекать самостоятельно.

Для этого мы используем базу данных образцов рукописного написания цифр MNIST (Modified National Institute of Standards and Technology), предложенную Национальным институтом стандартов и технологий и широко применяемую практиками машинного обучения.

В этой главе для выполнения низкоуровневых математических операций мы будем использовать библиотеку NumPy, которая является отраслевым стандартом для машинного обучения и математических вычислений средствами языка Python. Кроме того, она пригодится нам и в остальной части книги. Прежде чем опробовать любой из примеров кода в этой главе, вам необходимо установить библиотеку NumPy вместе с предпочтительным менеджером пакетов. Если вы используете менеджер pip, выполните команду `pip install numpy` в оболочке, чтобы его установить. Если вы используете менеджер Conda, выполните команду `conda install numpy`.

5.1.1. Набор данных MNIST

Набор данных MNIST включает 60 000 изображений размером 28×28 пикселов. Несколько примеров показано на рис. 5.1. В большинстве случаев люди не испы-

тывают проблем с распознаванием таких изображений, и вы легко можете прочитать цифры в первом ряду: 7, 5, 3, 9, 3, 0 и т. д. Однако в некоторых случаях даже людям бывает трудно понять, что изображено на картинке. Например, цифра на четвертом изображении в пятом ряду на рис. 5.1 легко может быть 4 или 9.

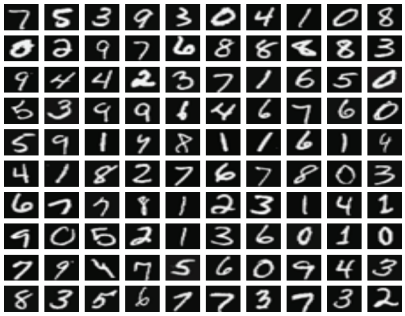


Рис. 5.1 ❖ Образцы рукописных цифр из набора данных MNIST, широко используемого в сфере оптического распознавания символов

Каждое изображение в наборе MNIST снабжено *меткой* – цифрой от 0 до 9, соответствующей изображенному значению.

Перед просмотром данных их необходимо загрузить. В папке *mng.bz/P8mn* репозитория GitHub для этой книги находится файл *mnist.pkl.gz*.

В этой папке также содержится весь код, который вам предстоит написать в ходе изучения этой главы. Как и прежде, мы рекомендуем создавать кодовую базу постепенно, однако вы также можете запустить уже готовый код из репозитория GitHub.

5.1.2. Предварительная обработка данных MNIST

Поскольку метки в этом наборе данных представляют собой целые числа от 0 до 9, мы используем так называемое *унитарное кодирование* (onehot encoding) для преобразования цифры 1 в вектор длиной 10, содержащий нули во всех позициях, за исключением первой, в которую мы поместим 1. Этот удобный способ представления широко используется в контексте машинного обучения. Резервирование первого слота вектора для метки 1 позволяет таким алгоритмам, как нейронные сети, легче различать метки. При использовании унитарного кодирования цифра 2, например, имеет следующее представление: `[0, 0, 1, 0, 0, 0, 0, 0, 0, 0]`.

Листинг 5.1 ❖ Унитарное кодирование меток MNIST

```
import six.moves.cPickle as pickle
import gzip
import numpy as np

def encode_label(j):
    e = np.zeros((10, 1))
    e[j] = 1.0
    return e
```

Унитарное кодирование индексов векторов длиной 10.

Преимущество унитарного кодирования заключается в том, что у каждой цифры есть свой «слот», и мы можем использовать нейронные сети, чтобы определить *вероятности* для входного изображения, которые нам очень пригодятся в дальнейшем.

Файл *mnist.pkl.gz* содержит три пула данных: обучающие, контрольные и тестовые данные. Как говорилось в главе 1, обучающие данные используются для обучения или подгонки алгоритма машинного обучения, а тестовые данные – для оценки качества этого обучения. Контрольные данные могут использоваться для настройки и проверки конфигурации алгоритма, однако при изучении этой главы их можно просто проигнорировать.

Набор данных MNIST содержит квадратные изображения, высота и ширина которых составляет 28 пикселов. Данные изображения загружаются в *векторы признаков* размером 784 = 28 × 28. Мы полностью отбрасываем структуру изображения и рассматриваем только пикселы, представленные в виде вектора. Каждое значение этого вектора представляет собой значение яркости серой шкалы от 0 до 1, где 0 соответствует белому цвету, а 1 – черному.

Листинг 5.2 ❖ Преобразование данных MNIST и загрузка обучающих и тестовых данных

```
def shape_data(data):
    features = [np.reshape(x, (784, 1)) for x in data[0]]
    labels = [encode_label(y) for y in data[1]]
    return zip(features, labels)

def load_data():
    with gzip.open('mnist.pkl.gz', 'rb') as f:
        train_data, validation_data, test_data = pickle.load(f)
    return shape_data(train_data), shape_data(test_data)
```

Преобразование входных изображений в векторы признаков длиной 784.

Унитарное кодирование всех меток.

Создание пар признаков и меток.

Получение трех наборов данных путем распаковки и загрузки данных MNIST.

Отбрасывание контрольного набора и преобразование двух других наборов данных.

Итак, мы получили простое представление набора данных MNIST. Как признаки, так и метки закодированы в виде векторов. Теперь нам необходимо разработать механизм, который способен научиться точно сопоставлять признаки и метки. А если конкретнее, то нам нужно разработать алгоритм, который будет использовать обучающие признаки и метки, для того чтобы научиться предсказывать метки для тестовых признаков.

Нейронные сети хорошо справляются с этой задачей, как будет показано в следующем разделе, однако давайте сначала обсудим наивный подход, демонстрирующий общие проблемы, которые нам предстоит решить. Распознавание цифр дается людям относительно легко, однако нам трудно объяснить, как именно мы это делаем и откуда мы знаем то, что знаем. Феномен обладания большим объемом знаний, чем можно облечь в слова, называется *парадоксом Поланьи*. Из-за этого нам особенно трудно предоставить машине *явный* способ решения данной задачи.

В данном случае одним из важнейших аспектов является *распознавание образов* – каждая рукописная цифра имеет определенные особенности, свойственные

ее прототипу или цифровой версии. Например, 0 представляет собой овал, а 1 – вертикальную линию. На основании этой эвристики можно попробовать классифицировать рукописные цифры с помощью наивного подхода, просто сравнивая их друг с другом: изображение цифры 8 должно быть ближе к среднему изображению восьмерки, чем к изображению любой другой цифры. Этот подход можно реализовать с помощью функции `average_digit`.

Листинг 5.3 ❖ Вычисление среднего значения для изображений одной и той же цифры

```
import numpy as np
from dlgo.nn.load_mnist import load_data
from dlgo.nn.layers import sigmoid_double

def average_digit(data, digit):
    filtered_data = [x[0] for x in data if np.argmax(x[1]) == digit]
    filtered_array = np.asarray(filtered_data)
    return np.average(filtered_array, axis=0)

train, test = load_data()
avg_eight = average_digit(train, 8)
```

Вычисление среднего значения для всех изображений конкретной цифры из набора данных.

Использование среднего изображения цифры 8 в качестве параметров простой модели для распознавания восьмерок.

Среднее изображение цифры 8 из обучающего набора представлено на рис. 5.2.

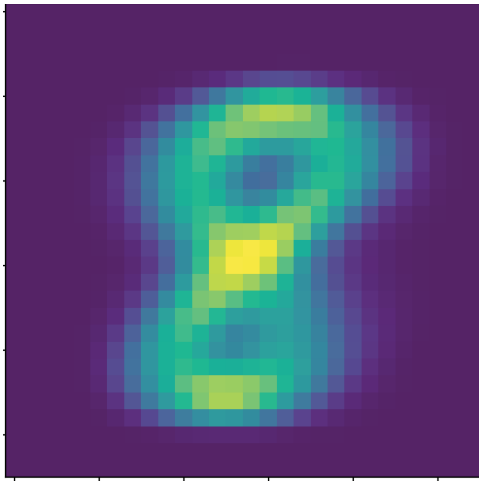


Рис. 5.2 ❖ Так выглядит среднее изображение рукописной цифры 8 из обучающего набора MNIST. Как правило, результатом усреднения многих сотен изображений бывает размытое пятно, однако на данном изображении довольно легко можно распознать цифру 8

Поскольку почерк сильно варьируется от человека к человеку, изображение восьмерки несколько размыто, но по-прежнему узнаваемо. Может быть, это представление позволит распознать и другие восьмерки в наборе данных? Для вычисления и отображения средней восьмерки, показанной на рис. 5.2, используется следующий код.

Листинг 5.4 ❖ Вычисление и отображение средней восьмерки из обучающего набора

```
from matplotlib import pyplot as plt

img = (np.reshape(avg_eight, (28, 28)))
plt.imshow(img)
plt.show()
```


Это среднее представление цифры 8, `avg_eight`, в обучающем наборе MNIST содержит большое количество информации о том, какими общими свойствами должны обладать изображения восьмерки. Мы используем `avg_eight` в качестве *параметров* простой модели, чтобы определить, является ли данный входной вектор `x`, представляющий цифру, изображением восьмерки. Когда речь идет о нейронных сетях, под параметрами, как правило, подразумеваются *веса*, и в данном случае `avg_eight` будет выступать именно в качестве веса.

Для удобства мы используем транспозицию и определим `W = np.transpose(avg_eight)`. Затем мы вычислим *скалярное произведение* `W` и `x`, т. е. попарно перемножим значения `W` и `x` и суммируем все 784 результирующих значения. Если наша гипотеза верна и `x` соответствует 8, то более темные пикселы должны располагаться примерно в тех же местах, что и в случае `W`, и наоборот. Если же `x` не соответствует 8, то совпадений должно быть меньше. Давайте проверим эту гипотезу на нескольких примерах.

Листинг 5.5 ❖ Вычисление степени соответствия цифры значениям весов с помощью скалярного произведения

```
x_3 = train[2][0]      <-- Обучающий образец с индексом 2 представляет цифру 4.
x_18 = train[17][0]    <-- Обучающий образец с индексом 17 представляет цифру 8.

W = np.transpose(avg_eight)
np.dot(W, x_3)         <-- Результат этого произведения составляет примерно 20,1.
np.dot(W, x_18)        <-- Результат этого произведения намного больше и составляет около 54,2.
```


Мы вычисляем скалярное произведение весов `W` и двух образцов из набора MNIST, один из которых представляет цифру 4, а другой – 8. Как видите, результат для цифры 8 (54,2) намного превышает результат для цифры 4 (20,1). Кажется, мы на верном пути. Теперь нам нужно определить границу, после которой результирующее значение будет считаться достаточно высоким, чтобы его можно было рассматривать в качестве предсказания цифры 8. В принципе, в результате скалярного произведения двух векторов может получиться любое действительное число. Полученный результат можно *преобразовать* в значение, принадлежащее диапазону [0, 1], чтобы определить точку отсечения, например 0,5, и объявить все, что превышает это значение, соответствующим цифре 8.

Среди прочего для этого используется *сигмоидальная функция*, или *сигмоида*, которая часто обозначается греческой буквой σ (сигма). Для действительного числа x сигмоидальная функция определяется следующим образом:

$$\sigma(x) = \frac{1}{1+e^{-x}}.$$

Рисунок 5.3 позволяет получить представление о том, как выглядит эта функция.

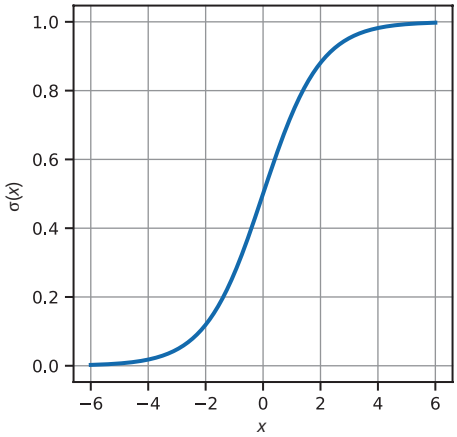


Рис. 5.3 ❖ График сигмоидальной функции. Сигмоида преобразует действительные значения в значения, принадлежащие диапазону [0, 1]. Около значения 0 наблюдается довольно крутой наклон графика, а ближе к минимальным и максимальным значениям кривая становится более пологой

Прежде чем применять сигмоидальную функцию к результату скалярного произведения, необходимо реализовать ее средствами языка Python.

Листинг 5.6 ❖ Простая реализация сигмоиды для значений типа double и векторов

```
def sigmoid_double(x):
    return 1.0 / (1.0 + np.exp(-x))

def sigmoid(z):
    return np.vectorize(sigmoid_double)(z)
```

Обратите внимание на то, что мы предоставляем как функцию `sigmoid_double`, работающую со значениями типа double, так и версию, которая вычисляет сигмоиду для векторов. Прежде чем применять эту функцию к результатам предыдущих вычислений, учтите, что значение сигмоиды 2 уже близко к единице, поэтому значения функции для двух ранее рассмотренных образцов, т. е. значения сигмоиды(54,2) и сигмоиды(20,1), будут практически одинаковыми. Данную проблему можно решить, *сдвинув* результат скалярного произведения к 0. Это называется применением *члена смещения*, который часто обозначается буквой *b*. Мы предполагаем, что подходящее смещение для наших образцов составляет $b = -45$. Используя весовые коэффициенты и смещение, мы можем вычислить *предсказания* нашей модели следующим образом.

Листинг 5.7 ❖ Вычисление предсказаний на основе весовых коэффициентов и смещения с помощью скалярного произведения и сигмоидальной функции

```
def predict(x, W, b):
    return sigmoid_double(np.dot(W, x) + b)
```

Простое предсказание определяется путем применения сигмоиды к результату выражения np.doc(W, x) + b.

```
b = -45
```

Исходя из предыдущих вычислений, мы задали смещение равным −45.

```
print(predict(x_3, W, b))
```

Предсказание для образца с цифрой 4 близко к 0.

```
print(predict(x_18, W, b))
```

Предсказание для образца с цифрой 8 в данном случае равно 0,96. Кажется, мы на верном пути.

Мы получили удовлетворительные результаты для двух примеров x_3 и x_18. Предсказание для второго близко к 1, а для первого – к 0. Эта процедура отображения входного вектора x в $\sigma(Wx + b)$ для W, вектора такого же размера, как и x, называется *логистической регрессией*. На рис. 5.4 схематически представлен этот алгоритм для вектора длиной 4.

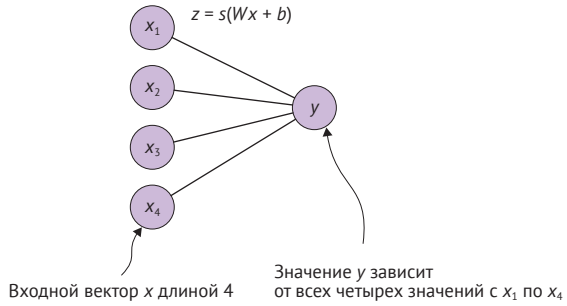


Рис. 5.4 ❖ Пример логистической регрессии, предполагающей отображение входного вектора x длиной 4 в выходное значение y, принадлежащее диапазону от 0 до 1. Представленная схема отражает зависимость выходного значения y от всех четырех значений входного вектора x

Чтобы лучше понять, как это работает, давайте вычислим предсказания для всех обучающих и тестовых образцов. Как говорилось ранее, мы определяем точку отсечения, или *порог принятия решения*, превышение которого будет говорить в пользу того, что входное изображение содержит цифру 8. В качестве метрики оценки в данном случае мы выбираем *верность*, т. е. вычисляем долю правильных предсказаний.

Листинг 5.8 ❖ Оценка предсказаний модели с помощью порога принятия решения

В качестве метрики оценки мы выбрали верность – долю правильных предсказаний.

```
def evaluate(data, digit, threshold, W, b):
    total_samples = 1.0 * len(data)
    correct_predictions = 0
    for x in data:
        if predict(x[0], W, b) > threshold and np.argmax(x[1]) == digit:
            correct_predictions += 1
        if predict(x[0], W, b) <= threshold and np.argmax(x[1]) != digit:
            correct_predictions += 1
    return correct_predictions / total_samples
```

Если предсказанием для образца с цифрой 8 является значение, которое соответствует цифре 8, то предсказание является верным.

Если значение предсказания не превышает порогового и образец не содержит изображения восьмерки, то предсказание также является верным.

Давайте применим эту оценочную функцию, чтобы оценить качество предсказаний на трех наборах данных: обучающего, тестового и набора, включающего все изображения восьмерки среди тестовых образцов. Как и прежде, для этого мы используем пороговое значение 0,5, веса W и член смещения b.

Листинг 5.9 ❖ Вычисление верности предсказания для трех наборов данных

```
evaluate(data=train, digit=8, threshold=0.5, W=W, b=b)

evaluate(data=test, digit=8, threshold=0.5, W=W, b=b)

eight_test = [x for x in test if np.argmax(x[1]) == 8]
evaluate(data=eight_test, digit=8, threshold=0.5, W=W, b=b)
```

Верность предсказаний нашей простой модели на обучающих данных составляет 78 % (0,7814).

В случае тестовых данных верность чуть ниже – 77 % (0,7749).

В случае набора, включающего только изображения цифры 8, верность составляет лишь 67 % (0,6663).

Как видите, верность предсказаний модели на наборе обучающих данных является самой высокой и составляет около 78 %. Это не удивительно, учитывая то, что наша модель была *откалибрована* с использованием этого обучающего набора. Вообще, проводить оценку верности предсказаний модели на наборе обучающих данных нет смысла, поскольку это не скажет нам, насколько хорошо алгоритм справляется с *обобщением* (т. е. насколько хорошо он работает на не виденных ранее данных). Верность предсказаний модели на наборе тестовых данных близка к аналогичному показателю для обучающего набора и составляет около 77 %. Однако наибольшего внимания заслуживает последний показатель. На наборе, включающем все изображения восьмерки среди тестовых образцов, мы получаем показатель 66 %, т. е. наша простая модель позволяет верно распознать не виденное ранее изображение цифры 8 только в двух случаях из трех. Этот результат может служить в качестве приемлемой отправной точки, однако это далеко не самое лучшее из того, чего мы можем достичь. Что же пошло не так, и что можно сделать?

- Наша модель способна проводить различия только между конкретной цифрой (в данном случае 8) и всеми остальными. Поскольку как в обучающем, так и в тестовом наборе количество изображений цифр *сбалансировано*, лишь около 10 % образцов содержат изображение цифры 8. Таким образом, верность модели, которая каждый раз предсказывает 0, будет составлять около 90 %. Учитывайте *несбалансированность классов* при анализе подобных проблем классификации. В свете этого верность в 77 % на тестовых данных уже не выглядит такой впечатляющей. *Нам необходимо определить модель, способную верно предсказывать все 10 цифр*.
- В наших моделях используется слишком мало параметров. Для коллекции, состоящей из многих тысяч различных изображений рукописных цифр, у нас есть лишь набор весов размером с одно такое изображение. Не стоит ожидать, что такая маленькая модель способна охватить всю изменчивость почерка на этих изображениях. *Нам нужен класс алгоритмов, способный эффективно использовать гораздо большее количество параметров для учета изменчивости в данных.*
- Для конкретного предсказания мы просто задали предельное значение, превышение которого позволяет нам объявить распознаваемую цифру восьмеркой. Мы не использовали фактическое значение предсказания для оценки качества модели. Например, правильное предсказание со значением 0,95 определенно является более весомым результатом, чем предсказание со значением 0,51. *Нам нужно формализовать представление о степени близости предсказанных и фактических значений*.

- Мы задавали параметры своей модели, руководствуясь интуицией. С этого можно начать, однако суть машинного обучения заключается в том, что вместо навязывания своего представления о данных мы можем позволить алгоритму учиться на их основе. Всякий раз, когда предсказание модели оказывается верным, мы можем подкреплять ее поведение, а когда модель выдает неправильный результат – вносить в нее соответствующие корректировки. Другими словами, *нам нужно разработать механизм, способный обновлять параметры модели в соответствии с качеством предсказаний, полученных на наборе обучающих данных.*

Простой пример использования и примитивная модель, описанные в этом разделе, могут не показаться особенно впечатляющими, однако благодаря им мы познакомились со многими компонентами нейронных сетей. В следующем разделе мы используем полученные знания для решения каждой из четырех приведенных выше задач.

5.2. Основы нейронных сетей

Как можно улучшить OCR-модель? Во введении уже говорилось о том, что нейронные сети могут справиться с такой задачей гораздо лучше, чем созданная нами модель. Однако модель, которую мы создали, позволяет проиллюстрировать ключевые концепции, необходимые для разработки нейронной сети. В этом разделе рассмотренная ранее модель будет описана в терминах нейронных сетей.

5.2.1. Логистическая регрессия как простая искусственная нейронная сеть

В разделе 5.1 мы обсуждали использование логистической регрессии для *бинарной классификации*. Как вы помните, мы взяли вектор признаков x, представляющий собой образец из набора данных, применили к нему алгоритм, сначала умножив его на весовую матрицу W, а затем прибавив смещение b. Затем для получения предсказания y в диапазоне от 0 до 1 мы применили сигмоидальную функцию: $y = \sigma(Wx + b)$.

Здесь следует обратить внимание на несколько моментов. Во-первых, вектор признаков x можно интерпретировать как совокупность нейронов, иногда называемых *блоками*, которые связаны со значением y посредством W и b, как было показано на рис. 5.4. Во-вторых, учтите, что сигмоиду можно рассматривать как функцию активации в плане того, что она принимает результат выражения $Wx + b$ и преобразует его в значение, принадлежащее диапазону [0, 1]. Если интерпретировать значение, близкое к 1, как активацию нейрона, а значение, близкое к 0, – наоборот, то эта структура уже может рассматриваться в качестве небольшой искусственной нейронной сети.

5.2.2. Сети с несколькими размерностями выходного сигнала

В примере, описанном в разделе 5.1, мы упростили задачу распознавания рукописных цифр до задачи бинарной классификации, т. е. до проведения различия между цифрой 8 и всеми остальными цифрами. Однако нас интересует предсказание 10 классов, по одному на каждую цифру. По крайней мере, формально мы

можем довольно легко этого достичь, изменив *y*, *W* и *b*, т. е. такие параметры модели, как выходные данные, веса и смещение.

Сначала мы преобразуем *y* в вектор длиной 10, который будет содержать по одному значению вероятности для каждой из 10 цифр:

$$y = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_8 \\ y_9 \end{bmatrix}.$$

Теперь давайте соответствующим образом скорректируем веса и смещение. Напомним, что до сих пор набор весов *W* представлял собой вектор длиной 784. Мы преобразуем его в матрицу размером (10, 784). Это позволит умножить матрицу *W* на входной вектор *x*, в результате чего получится вектор длиной 10. Затем если мы преобразуем член смещения в вектор длиной 10, то сможем прибавить его к результату матричного умножения *Wx*. Наконец, мы можем вычислить сигмоиду вектора z, применив эту функцию к каждому из его компонентов:

$$\sigma(z) = \begin{bmatrix} \sigma(z_0) \\ \sigma(z_1) \\ \vdots \\ \sigma(z_8) \\ \sigma(z_9) \end{bmatrix}.$$

На рис. 5.5 изображена слегка измененная схема, включающая четыре входных и два выходных нейрона.

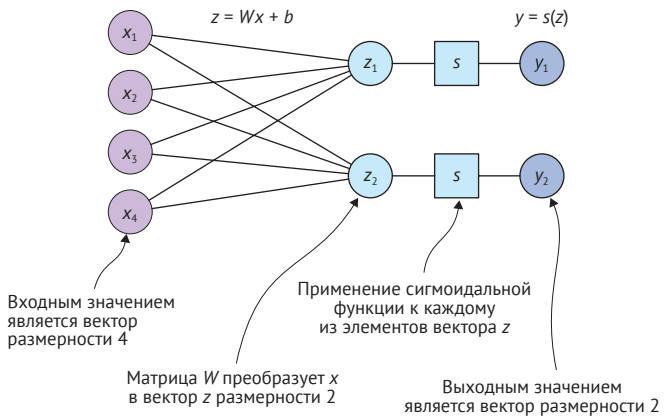


Рис. 5.5 ❖ В этой простой сети четыре входных нейрона соединены с двумя выходными нейронами путем выполнения матричного умножения, прибавления двумерного члена смещения и дальнейшего применения сигмоидальной функции к каждому из компонентов

Что мы получили в итоге? Теперь мы можем отобразить входной вектор x в выходной вектор y, который раньше представлял собой отдельное значение. Преимущество этого подхода состоит в том, что мы можем выполнять такое преобразование векторов многократно, создавая при этом так называемую *сеть прямого распространения*.

5.3. Сети прямого распространения

Давайте быстро вспомним, какие действия мы выполнили в предыдущем разделе.

1. Мы начали с вектора входных нейронов x и применили к нему простое преобразование, а именно $z = Wx + b$. На языке линейной алгебры эти преобразования называются *аффинно-линейными*. Здесь мы используем z в качестве промежуточной переменной для упрощения дальнейших обозначений.
2. Для получения выходных нейронов y мы применили функцию активации – сигмоиду $y = \sigma(z)$. Результат применения функции σ говорит об уровне активации нейронов y.

В основе сетей прямого распространения лежит идея о том, что этот процесс можно применять итеративно, т. е. многократно повторять два описанных выше простых действия. Эти строительные блоки образуют то, что мы называем *слоем*. Таким образом, можно сказать, что мы создаем *стек из множества слоев*, получая в итоге *многослойную нейронную сеть*. Давайте изменим последний пример, добавив еще один слой. Для этого нам нужно выполнить следующие действия:

1) начиная с входного вектора x, мы вычисляем выражение $z^1 = W^1x + b^1$;
2) из промежуточного результата z^1 мы получаем выходное значение y, вычислив выражение $y = W^2z^1 + b^2$.

Обратите внимание на то, что верхние индексы обозначают слой, в котором мы находимся, а нижние индексы – позицию в векторе или матрице. Способ работы с двумя слоями вместо одного представлен на рис. 5.6.

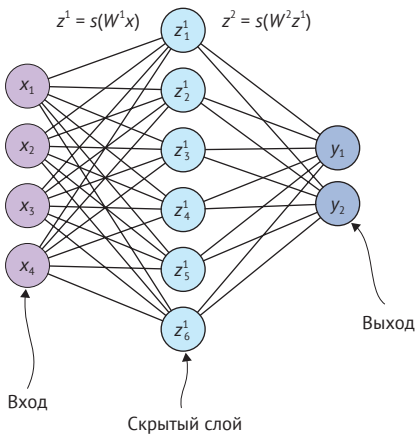


Рис. 5.6 ❖ Искусственная нейронная сеть с двумя слоями. Входные нейроны x связаны с промежуточным набором блоков z, которые, в свою очередь, связаны с выходными нейронами y

Вы, вероятно, уже догадались, что количество слоев в сети не ограничено. Кроме того, в качестве функции активации вовсе не обязательно постоянно использовать логистическую сигмоиду. Существует множество функций активации, и в следующей главе мы поговорим о некоторых из них. Последовательное применение функций активации всех слоев сети к одной или нескольким точкам данных обычно называется *прямым проходом*, поскольку данные всегда передаются только вперед, от входа к выходу (на рисунке слева направо), и никогда назад.

На рис. 5.7 представлена обычная сеть прямого распространения с тремя слоями.

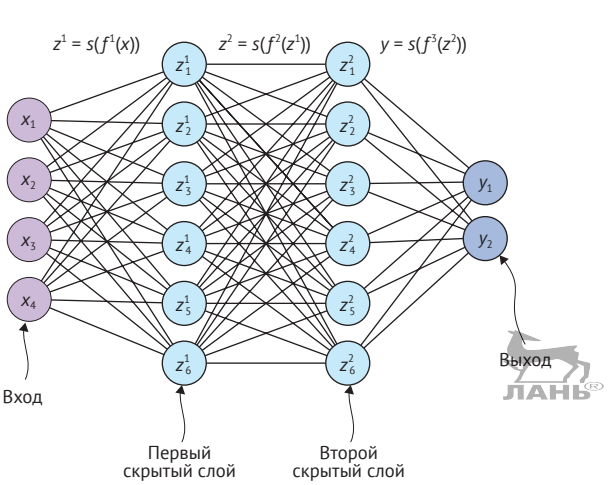


Рис. 5.7 ❖ Нейронная сеть с тремя слоями. При определении нейронной сети вы не ограничены ни количеством слоев, ни количеством нейронов в слое

Давайте кратко повторим пройденное.

- *Последовательная нейронная сеть* – это механизм для отображения признаков или входных нейронов x в предсказания или выходные нейроны y. Это достигается путем последовательного объединения слоев с простыми функциями активации.
- *Слой* – это рецепт для отображения конкретного входного значения в выходное. Вычисление выхода слоя для пакета данных называется *прямым проходом*. Прямой проход последовательной сети осуществляется путем последовательного вычисления выхода каждого слоя, начиная с входного.
- *Сигмоидальная функция* – это функция активации, которая принимает вектор действительнозначных нейронов и *активирует* их, преобразуя их выходные сигналы в значения, принадлежащие диапазону [0, 1]. Значения, близкие к 1, интерпретируются как активация нейрона.
- При заданной весовой матрице W и смещении b применение аффинно-линейного преобразования $Wx + b$ создает слой, который обычно называется *плотным*, или *полносвязным*. В дальнейшем мы будем называть такие слои *плотными слоями*.
- В зависимости от реализации плотные слои могут иметь или не иметь встроенных функций активации. В качестве слоя можно рассматривать

и $\sigma(Wx + b)$, а не только аффинно-линейное преобразование. С другой стороны, в качестве слоя принято рассматривать только функцию активации, и в своей реализации мы сделаем именно так. В конце концов, вопрос о целесообразности добавления функций активации в плотные слои – это всего лишь несколько иной взгляд на разделение и группировку частей функции в логический блок.

- *Нейронная сеть прямого распространения* – это последовательная сеть, состоящая из плотных слоев с функциями активации. По историческим причинам, обсуждение которых выходит за рамки данной книги, эта архитектура также часто называется *многослойным перцептроном*, или *MLP* (multilayer perceptron).
- Все нейроны, не являющиеся ни входными, ни выходными, называются *скрытыми блоками*. А входные и выходные нейроны иногда называются *видимыми блоками*. Это объясняется тем, что скрытые блоки находятся *внутри сети*, а видимые можно наблюдать непосредственно. И хотя это не совсем так, поскольку мы обычно имеем доступ к любой части системы, такое разделение является довольно удобным. Соответственно, слои, расположенные между входом и выходом, называются *скрытыми слоями*: любая последовательная сеть с двумя и более слоями имеет как минимум один скрытый слой.
- Если не указано иное, буквой x обозначаются входные данные для сети, а буквой y – выходные; иногда эти буквы сопровождаются нижними индексами, обозначающими рассматриваемый образец. Большая сеть, включающая множество скрытых слоев, называется *нейронной сетью глубокого обучения* (отсюда и название метода).

Непоследовательные нейронные сети

В данный момент нас интересуют только *последовательные* нейронные сети, слои которых образуют последовательность. Такая сеть начинается с входного слоя, а заканчивается выходным, при этом каждый скрытый слой имеет только одного предшественника и одного последователя. Этого достаточно для применения метода глубокого обучения к боту для игры в го.

Вообще, теория нейронных сетей допускает и произвольные непоследовательные архитектуры. Например, в некоторых приложениях имеет смысл объединить или сложить выходные значения двух слоев (путем объединения двух или более предыдущих слоев). В этом случае мы объединяем несколько входных значений в одно выходное.

В других приложениях полезным может оказаться разделение одного входного значения на несколько выходных. Как правило, слой может иметь несколько входов и выходов. Сети с несколькими входами и несколькими выходами описаны в главах 11 и 12 соответственно.

Структура многослойного перцептрона, включающего l слоев, полностью описывается набором весовых коэффициентов $W = W^1, \ldots, W^l$, набором смещений $b = b^1, \ldots, b^l$ и набором функций активации, выбранных для каждого из слоев.

Однако прежде чем приступать к обучению на основе данных и обновлению параметров, нам необходимо познакомиться с функциями потерь и способами их оптимизации.

5.4. Оценка предсказаний. Функции потерь и оптимизация

В разделе 5.3 мы говорили о том, как создать нейронную сеть прямого распространения и пропустить через нее входные данные, однако мы по-прежнему не знаем, как оценивать качество полученных предсказаний. Для этого нам потребуется способ определения степени близости предсказанных и фактических значений.

5.4.1. Что такое функция потерь?

Для количественной оценки степени близости предсказания и фактического значения нам потребуется *функция потерь*, которая также часто называется *целевой функцией*. Допустим, у нас есть сеть прямого распространения с весами W, смещениями b и сигмоидальными функциями активации. Для заданного набора входных признаков $X_1, \ldots, X_k$ и соответствующих меток $\hat{y}_1, \ldots, \hat{y}_k$ ($\hat{y}$ произносится как «у с крышкой») с помощью нашей сети можно вычислить предсказания $y_1, \ldots, y_k$. В данном случае функция потерь определяется следующим образом:

$$\sum_i \text{Loss}(W, b, X_i, \hat{y}_i) = \sum_i \text{Loss}(y_i, \hat{y}_i).$$

Здесь $\text{Loss}(y_i, \hat{y}_i) \geqslant 0$, а Loss представляет собой *дифференцируемую функцию*. Функция потерь – это гладкая функция, которая присваивает неотрицательное значение каждой паре (предсказание, метка). Потеря некоторого количества признаков и меток представляет собой сумму потерь для образцов. Функция потерь оценивает соответствие параметров нашего алгоритма предоставленным данным. Цель обучения состоит в *минимизации потерь* путем нахождения эффективных стратегий настройки параметров.

5.4.2. Среднеквадратическая ошибка

Одной из распространенных функций потерь является *среднеквадратическая ошибка* (СКО). Хотя СКО – это не оптимальный вариант для нашего случая, данная функция потерь является одной из самых интуитивно понятных. Мы определяем степень соответствия предсказания фактической метке путем вычисления квадрата расстояния и усреднения по всем рассмотренным примерам. Используя выражение $\hat{y} = \hat{y}_1, \ldots, \hat{y}_k$ для меток и $y = y_1, \ldots, y_k$ для предсказаний, мы можем определить среднеквадратическую ошибку следующим образом:

$$\text{MSE}(y, \hat{y}) = \frac{1}{2}\sum_{i=1}^{k}(y_i - \hat{y}_i)^2.$$

Мы обсудим преимущества и недостатки различных функций потерь после того, как разберемся с теорией. А пока давайте реализуем среднеквадратическую функцию потерь средствами языка Python.

Листинг 5.10 ❖ Среднеквадратическая функция потерь и ее производная

```
import random
import numpy as np

class MSE:

    def __init__(self):
        pass

    @staticmethod
    def loss_function(predictions, labels):
        diff = predictions - labels
        return 0.5 * sum(diff * diff)[0]

    @staticmethod
    def loss_derivative(predictions, labels):
        return predictions - labels
```

Использование среднеквадратической ошибки в качестве функции потерь.

Если определить СКО как половину квадрата разности между предсказаниями и метками...

...то производная функции потерь будет представлять собой просто разность между предсказаниями и метками.

Обратите внимание на то, что мы реализовали не только саму функцию потерь, но и ее производную относительно предсказаний: `loss_derivative`. Эта производная представляет собой вектор и вычисляется путем вычитания меток из предсказаний.

Далее мы поговорим о роли таких производных в обучении нейронных сетей.

5.4.3. Поиск минимумов функции потерь

Функция потерь для набора предсказаний и меток сообщает нам о том, насколько хорошо настроены параметры нашей модели. Чем меньше потери, тем вернее предсказания, и наоборот. Сама по себе функция потерь является функцией параметров сети. В нашей реализации СКО мы не предоставляем весовые коэффициенты напрямую. Вместо этого они передаются *неявно* через предсказания `predictions`, поскольку мы используем весовые коэффициенты для их вычисления.

Из математического анализа известно, что для минимизации функции потерь необходимо вычислить ее *производную* и приравнять ее к 0. Набор параметров в этой точке называется *решением*. Вычисление производной функции и ее оценивание в определенной точке называется *вычислением градиента*. Первый этап вычисления производной был выполнен в рамках реализации СКО, однако это еще не все. Наша цель заключается в явном вычислении градиентов для всех весовых коэффициентов и членов смещения в сети.

Если вам требуется освежить свои знания, касающиеся основ математического анализа, обязательно ознакомьтесь с приложением А. На рис. 5.8 изображена поверхность в трехмерном пространстве. Эту поверхность можно рассматривать как функцию потерь для двумерных входных данных. Первые две оси представляют весовые коэффициенты, а третья направленная вверх ось – величину потерь.

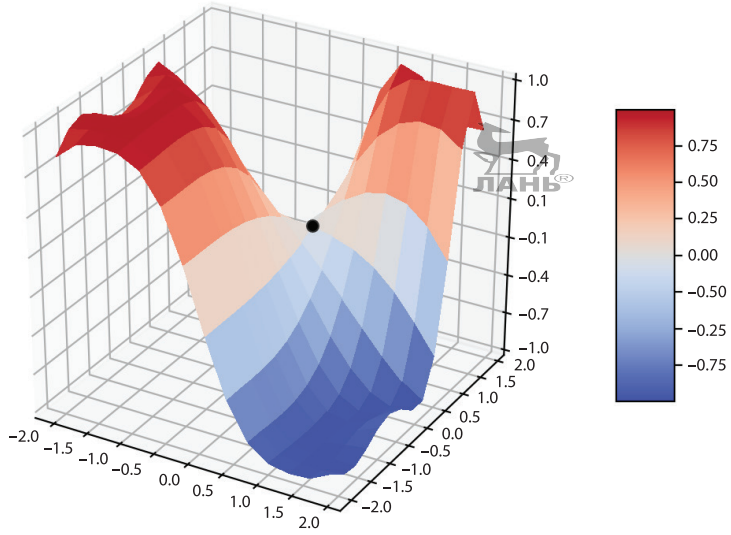


Рис. 5.8 ❖ Пример функции потерь для двумерных входных данных (поверхность потерь). Данная поверхность имеет минимум в темной области внизу справа, который можно вычислить путем решения производной функции потерь

5.4.4. Градиентный спуск для нахождения минимумов

Интуитивно понятно, что, когда мы вычисляем градиент функции в данной точке, этот градиент указывает в направлении самого крутого подъема. Алгоритм метода *градиентного спуска* для нахождения минимума функции потерь Loss с набором параметров W выглядит следующим образом:

1) вычислить градиент Δ функции Loss для текущего набора параметров W (вычислить производную функции Loss по каждому из весов W);
2) обновить значение W, вычтя из него Δ. Этот шаг называется *следованием направлению градиента*. Поскольку Δ указывает в направлении самого крутого подъема, при вычитании этого значения мы следуем в направлении самого крутого спуска;
3) повторять до тех пор, пока значение Δ не станет равным 0.

Поскольку наша функция потерь неотрицательна, мы точно знаем, что она имеет, по крайней мере, один минимум. Однако она может иметь и бесконечное множество минимумов. Например, в случае плоской поверхности *каждая* точка является минимумом.

Локальные и глобальные минимумы

Точка с нулевым градиентом, достигнутая в результате применения метода градиентного спуска, по определению является минимумом. Точное математическое определение минимума для дифференцируемых функций многих переменных предполагает использование информации о *кривизне* функции.

Градиентный спуск позволяет в конечном итоге обнаружить минимум. Мы можем следовать направлению градиента функции вплоть до обнаружения точки с нулевым градиентом. Единственная сложность заключается в том, что мы точно не знаем, является этот минимум *локальным* или *глобальным*. Мы можем застрять на плато, которое является локальным минимумом для данной функции, хотя другие точки при этом могут иметь меньшее абсолютное значение. Точка, отмеченная на рис. 5.8, является локальным минимумом, однако очевидно, что на этой поверхности существуют и меньшие значения.

Как ни странно, мы решаем данную проблему, просто игнорируя это. На практике градиентный спуск часто дает удовлетворительные результаты, поэтому в контексте функций потерь для нейронных сетей мы игнорируем вопрос о том, является минимум локальным или глобальным. Как правило, мы даже не используем алгоритм вплоть до достижения минимума, а останавливаемся после выполнения заданного количества итераций.

На рис. 5.9 показано, как работает градиентный спуск для поверхности потерь, изображенной на рис. 5.8, а выбор параметров обозначен точкой в правом верхнем углу.

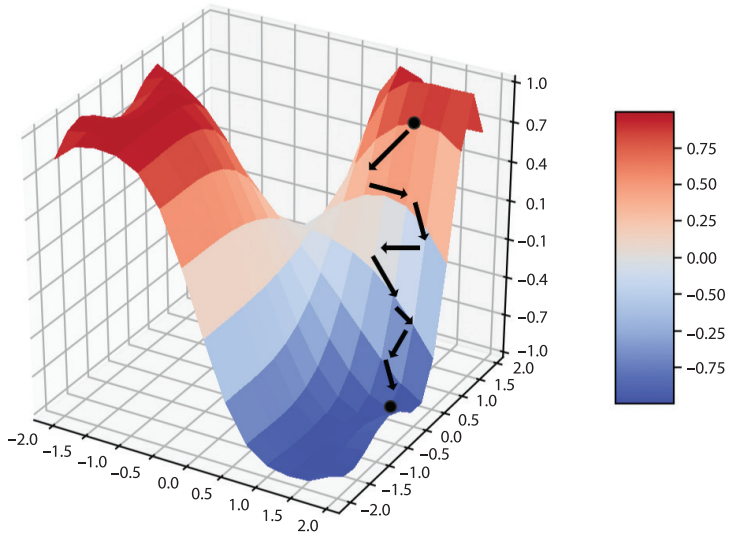


Рис. 5.9 ❖ Итеративное следование направлению градиента функции потерь в конечном итоге позволяет обнаружить ее минимум

При реализации СКО мы видели, что производную среднеквадратической функции потерь легко вычислить *формально*: это разница между метками и предсказаниями. Но чтобы *оценить такую производную*, сначала необходимо вычислить предсказания. Для получения представления о градиентах для всех параметров мы должны оценить и агрегировать производные для каждого образца из обучающего набора. Учитывая то, что мы обычно имеем дело со многими тысячами, а иногда и миллионами образцов, сделать это практически невозможно. Вместо этого мы аппроксимируем значение градиента с помощью метода под названием *стохастический градиентный спуск*.

5.4.5. Стохастический градиентный спуск для функций потерь

Чтобы вычислить градиенты и применить метод градиентного спуска для обучения нейронной сети, необходимо оценить функцию потерь и ее производную по параметрам сети в каждой отдельной точке обучающего набора, что в большинстве случаев является слишком дорогостоящим. Вместо этого мы можем использовать метод *стохастического градиентного спуска* (СГС). Для этого мы сначала формируем так называемые *мини-пакеты*, содержащие несколько образцов из обучающего набора. Все мини-пакеты имеют одинаковую длину, называемую *размером мини-пакета*. Для такой проблемы классификации, как рассматриваемая нами задача распознавания рукописных цифр, рекомендуется выбирать размер мини-пакета, сопоставимый с количеством меток. Это позволяет гарантировать то, что в мини-пакете будет представлена каждая из меток.

В случае данной нейронной сети прямого распространения с количеством слоев l и мини-пакетом входных данных x_1, ..., x_k размера k можно выполнить прямой проход и вычислить потери для этого мини-пакета. Затем для каждого образца x_j в этом пакете можно оценить градиент функции потерь по любому параметру сети. Градиенты веса и смещения для слоя i обозначим как $\Delta_j W^i$ и $\Delta_j b^i$ соответственно.

Для каждого слоя и каждого образца в пакете мы вычисляем соответствующие градиенты и используем следующие *правила обновления* параметров:

$$W^i \leftarrow W^i - \alpha \sum_{j=1}^{k} \Delta_j W^i;$$

$$b^i \leftarrow b^i - \alpha \sum_{j=1}^{k} \Delta_j b^i.$$

Обновление параметров предполагает вычитание накопленной ошибки для конкретного пакета. В данном случае $a > 0$ обозначает *скорость обучения*, заданную до начала обучения сети.

Мы получили бы гораздо более точную информацию о градиентах, если бы выполнили суммирование по всем обучающим образцам сразу. Использование мини-пакетов представляет собой компромисс между точностью значения градиента и вычислительной эффективностью. Мы называем этот метод *стохастическим* градиентным спуском, потому что образцы для мини-пакета выбираются случайным образом. Хотя градиентный спуск теоретически гарантирует возможность приближения к локальному минимуму, в случае СГС это не так. На рис. 5.10 показано типичное поведение СГС. Некоторые из приблизительных значений

стохастического градиента могут не указывать в направлении спуска, однако достаточное количество итераций, как правило, позволяет приблизиться к (локальному) минимуму.

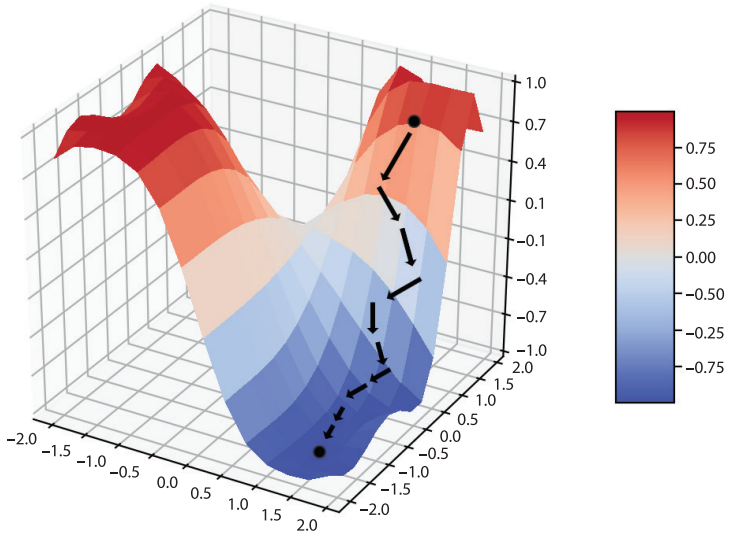


Рис. 5.10 ❖ Стохастические градиенты менее точны, поэтому при следовании в их направлении по поверхности потерь нам может потребоваться немного попетлять, прежде чем мы приблизимся к локальному минимуму

Оптимизаторы

Вычисление (стохастических) градиентов определяется фундаментальными принципами математического анализа, чего нельзя сказать о способе использования градиентов для обновления параметров. В случае СГС такие методы, как правило обновления весов, называются *оптимизаторами*.

Существует множество других оптимизаторов, а также более сложные версии стохастического градиентного спуска. Мы рассмотрим некоторые из расширений СГС в главе 7. Большая часть этих расширений сводится к адаптации скорости обучения с течением времени или предполагает более детальные обновления отдельных весовых коэффициентов.

5.4.6. Метод обратного распространения ошибки

Мы поговорили об обновлении параметров нейронной сети с помощью стохастического градиентного спуска, однако еще не обсудили метод вычисления градиентов. Для их вычисления используется так называемый алгоритм *обратного распространения ошибки*, который подробно описан в приложении Б. В этом разделе мы поговорим о данном методе, а также о необходимых компонентах, которые позволят вам самостоятельно реализовать нейронную сеть прямого распространения.

Напомним, что прямой проход по сети прямого распространения осуществляется путем последовательного вычисления простых строительных блоков. На основании выхода последнего слоя, предсказаний сети и меток мы определяем значение потери. Сама по себе функция потерь является *композицией* более простых функций. Для вычисления производной функции потерь мы можем использовать такое фундаментальное свойство математического анализа, как правило дифференцирования сложной функции. Это правило говорит о том, что производная сложной функции является производной композиции этих функций. Точно так же, как мы передавали входные данные вперед по сети слой за слоем, мы можем *передать производные назад по сети слой за слоем*. Мы осуществляем *обратное распространение* производных по сети, отсюда и название метода. Рисунок 5.11 иллюстрирует процесс обратного распространения ошибки по сети прямого распространения с двумя плотными слоями и сигмоидальными функциями активации.

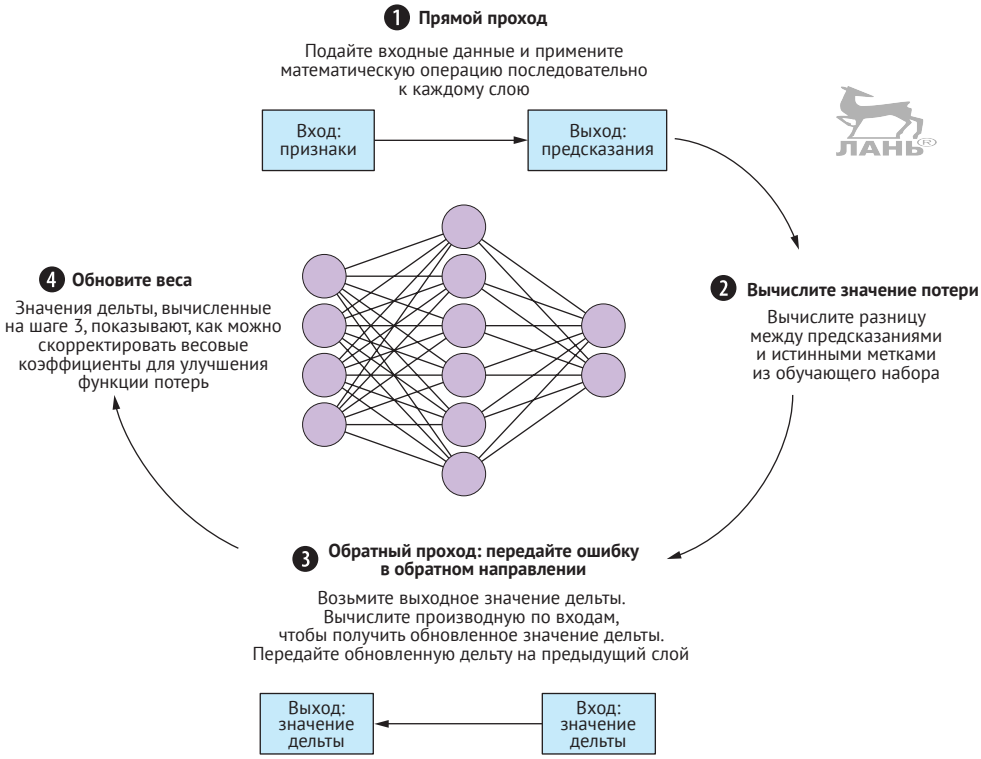


Рис. 5.11 ❖ Прямой и обратный проходы по двухслойной нейронной сети прямого распространения с сигмоидальными функциями активации и функцией потерь СКО

Давайте подробно рассмотрим каждый из этапов, представленных на рис. 5.11.

1. **Прямая передача обучающих данных**. На этом этапе мы берем образец входных данных x и передаем его по сети для получения предсказания следующим образом:

   а) выполняем аффинно-линейное преобразование: $Wx + b$;
   б) применяем к результату сигмоидальную функцию $\sigma(x)$. Обратите внимание на некоторое злоупотребление обозначениями – в данном случае под x подразумевается результат предыдущего этапа вычислений;
   в) повторяем эти два шага вплоть до достижения выходного слоя. В данном примере используется два слоя, однако количество слоев может быть любым.
2. **Оценивание функции потерь**. На этом этапе мы берем метки $\hat{y}$ для образца x и сравниваем их с предсказаниями y, вычисляя значение потери. В данном примере в качестве функции потерь используется среднеквадратическая ошибка.
3. **Обратное распространение ошибки**. На этом этапе мы берем значение потери и передаем его обратно по сети, вычисляя производные последовательно для каждого из слоев в соответствии с правилом дифференцирования сложной функции. При прямом проходе входные данные передаются вперед по сети, а при обратном проходе значения ошибки передаются назад.
   а) Мы распространяем ошибки, или значения дельты, обозначаемые символом Δ, назад по сети.
   б) Сначала мы вычисляем производную функции потерь, которая будет являться исходным значением Δ. Опять же, как и при прямом проходе сети, мы злоупотребляем обозначениями и на каждом этапе процесса обозначаем передаваемую ошибку символом Δ.
   в) Мы вычисляем производную сигмоиды по ее входу: $\sigma \cdot (1 - \sigma)$. Для передачи Δ на следующий слой можно выполнить компонентное умножение: $\sigma(1 - \sigma) \cdot \Delta$.
   г) Производная аффинно-линейного преобразования $Wx + b$ по x представляет собой просто W. Для передачи Δ мы вычисляем выражение $W^t \cdot \Delta$.
   д) Эти два этапа повторяются вплоть до достижения первого слоя сети.
4. **Обновление весовых коэффициентов с учетом значения градиента.** На последнем этапе мы используем вычисленные ранее значения дельты для обновления параметров сети (весовых коэффициентов и членов смещения).
   а) Сигмоидальная функция не имеет параметров, поэтому нам не нужно ничего с ней делать.
   б) Обновление Δb, применяемое к члену смещения на каждом слое, представляет собой просто Δ.
   в) Обновление ΔW для весов в слое задается в виде $\Delta \cdot x^T$ (нам необходимо транспонировать x, перед тем как умножать его на значение дельты).
   г) Вначале мы указали, что x представляет собой отдельный образец. Тем не менее все, что мы говорили, относится и к мини-пакетам. Если x обозначает мини-пакет образцов (x – это матрица, каждый столбец которой представляет собой входной вектор), прямой и обратный проходы вычисляются точно так же.

Теперь, когда мы обсудили все математические нюансы создания и использования сети прямого распространения, давайте применим полученные теоретические знания на практике и создадим нейронную сеть с нуля.

5.5. Обучение нейронной сети средствами языка Python

В предыдущем разделе было рассмотрено много теоретических основ, однако на концептуальном уровне нам удалось проработать лишь несколько базовых концепций. В ходе реализации нейронной сети вам следует беспокоиться только о трех классах: `Layer`, `SequentialNetwork`, который создается путем последовательного добавления нескольких объектов `Layer`, и `Loss`, который необходим сети для выполнения обратного распространения ошибки. Далее мы обсудим эти три класса, после чего вы загрузите и проверите образцы рукописных цифр и примените к ним свою реализацию нейронной сети. На рис. 5.12 показано, как эти классы Python сочетаются друг с другом при выполнении прямого и обратного проходов, описанных в предыдущем разделе.

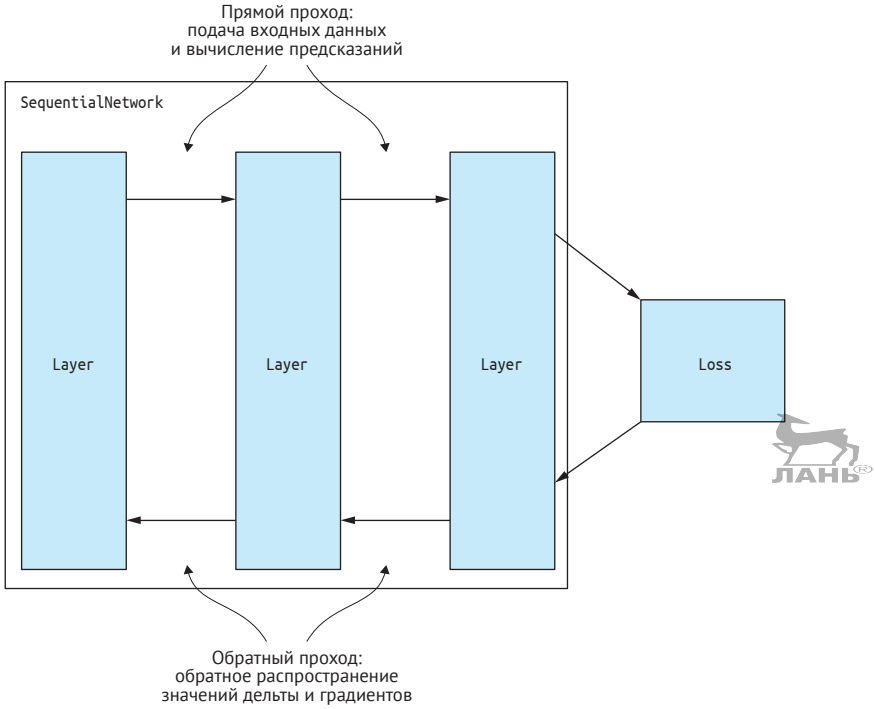


Рис. 5.12 ❖ Диаграмма классов для реализации сети прямого распространения на языке Python. Класс `SequentialNetwork` содержит несколько экземпляров класса `Layer`. Каждый экземпляр `Layer` реализует математическую функцию и ее производную. Методы Forward и Backward реализуют прямой и обратный проходы сети соответственно. Экземпляр `Loss` вычисляет функцию потерь – разницу между предсказаниями и обучающими данными

5.5.1. Слои нейронной сети в Python

Начиная работу с классом `Layer`, учтите, что слои предусматривают не только алгоритм обработки входных данных (прямой проход), но и механизм *обратного распространения* ошибки. Чтобы не пересчитывать значения активации при обрат-

ном проходе, целесообразно сохранять *состояние* данных, поступающих на слой и выходящих из слоя при выполнении обоих проходов. С учетом этого следующая инициализация `Layer` не должна вызвать сложностей. Мы начнем с создания модуля слоев, а позднее в этой главе используем компоненты данного модуля для создания нейронной сети.

Листинг 5.11 ❖ Базовая реализация слоя

```
import numpy as np

class Layer:                                  ← Объединение слоев в последовательную нейронную сеть.
    def __init__(self):
        self.params = []

        self.previous = None                  ← Слой знает своего предшественника (previous)...
        self.next = None                      ← ...и своего последователя (next).

        self.input_data = None
        self.output_data = None               ← Каждый слой может сохранять данные, поступающие и выходящие
                                                из него при выполнении прямого прохода сети.
        self.input_delta = None
        self.output_delta = None              ← Точно так же слой удерживает входные и выходные данные
                                                при обратном проходе сети.
```

Слой предусматривает список параметров и хранит как свои текущие входные и выходные данные, так и соответствующие входные и выходные значения дельты для выполнения обратного прохода сети.

Кроме того, поскольку мы хотим создать последовательную нейронную сеть, имеет смысл указать последователя и предшественника для каждого слоя. Добавьте следующий код в начатое ранее определение.

Листинг 5.12 ❖ Соединение слоев путем указания их последователей и предшественников

```
    def connect(self, layer):                 ← Этот метод позволяет соединить слой
        self.previous = layer                   с его соседями в последовательной сети.
        layer.next = self
```

Теперь нам нужно предусмотреть заглушки для прямого и обратного проходов в абстрактном классе `Layer`, которые предстоит реализовать подклассам.

Листинг 5.13 ❖ Прямой и обратный проходы в слое последовательной нейронной сети

```
    def forward(self):                        ← Каждая реализация слоя должна предусматривать функцию
        raise NotImplementedError               для передачи входных данных вперед по сети.

    def get_forward_input(self):              ← input_data зарезервированы для первого слоя.
        if self.previous is not None:           Для всех остальных входными данными являются
            return self.previous.output_data    выходы предыдущего слоя.
        else:
            return self.input_data

    def backward(self):                       ← Слои должны реализовать обратное распространение
        raise NotImplementedError               значений ошибки по сети.

    def get_backward_input(self):             ← input_delta зарезервирована для последнего слоя.
        if self.next is not None:               Все остальные слои получают значения ошибок
            return self.next.output_delta       от своих последователей.
        else:
```

```
        return self.input_delta
    def clear_deltas(self):
        pass
    def update_params(self, learning_rate):
        pass
    def describe(self):
        raise NotImplementedError
```

Мы вычисляем и аккумулируем значения дельты для каждого мини-пакета, после чего их необходимо сбросить.

Обновление параметров слоя в соответствии с текущими значениями дельты с помощью указанной скорости обучения (`learning_rate`).

Реализации слоя могут выводить на экран свои свойства.

В качестве вспомогательных функций мы предоставляем `get_forward_input` и `get_backward_input`, которые извлекают входные данные для соответствующего прохода, уделяя особое внимание входным и выходным нейронам. Кроме того, мы реализуем метод `clear_deltas` для периодического сброса значений дельты после их накопления для мини-пакетов, а также метод `update_params`, который обновляет параметры данного слоя после того, как сеть, использующая этот слой, подаст соответствующий сигнал.

Обратите внимание на то, что в качестве последнего фрагмента мы добавили метод, позволяющий слою вывести на экран собственное описание. Это удобный способ получения представления о том, как выглядит нейронная сеть.

5.5.2. Слои активации в нейронных сетях

Далее мы создадим первый слой `ActivationLayer`. Мы будем работать с реализованной ранее сигмоидальной функцией. Для осуществления обратного распространения ошибки нам также понадобится ее производная, реализовать которую не составит труда.

Листинг 5.14 ❖ Реализация производной сигмоидальной функции

```
def sigmoid_prime_double(x):
    return sigmoid_double(x) * (1 - sigmoid_double(x))

def sigmoid_prime(z):
    return np.vectorize(sigmoid_prime_double)(z)
```

Обратите внимание на то, что в случае сигмоиды мы предоставляем как скалярную, так и векторную версии производной. При определении слоя `ActivationLayer` с сигмоидой в качестве встроенной функции активации вы заметите, что сигмоида не имеет никаких параметров, поэтому на данном этапе нам не нужно беспокоиться о каком-либо обновлении.

Листинг 5.15 ❖ Сигмоидальный слой активации

```
class ActivationLayer(Layer):
    def __init__(self, input_dim):
        super(ActivationLayer, self).__init__()

        self.input_dim = input_dim
        self.output_dim = input_dim

    def forward(self):
        data = self.get_forward_input()
        self.output_data = sigmoid(data)

    def backward(self):
```

Данный слой активации использует сигмоидальную функцию для активации нейронов.

Прямой проход сети сводится к простому применению сигмоиды к входным данным.

```
        delta = self.get_backward_input()
        data = self.get_forward_input()
        self.output_delta = delta * sigmoid_prime(data)

    def describe(self):
        print("|-- " + self.__class__.__name__)
        print("  |-- dimensions: ({},{})"
              .format(self.input_dim, self.output_dim))
```

Обратный проход сети предполагает выполнение поэлементного умножения значения ошибки на производную сигмоиды, оцененную на входе в этот слой.

Тщательно изучите реализацию градиента, чтобы понять, как она вписывается в схему, представленную на рис. 5.11. Для этого слоя обратный проход сети предполагает простое поэлементное умножение текущей дельты слоя на производную сигмоиды, оцененную на входе в этот слой: $\sigma(x) \cdot (1 - \sigma(x)) \cdot \Delta$.

5.5.3. Плотные слои в Python как компоненты сетей прямого распространения

Теперь перейдем к реализации чуть более сложного слоя `DenseLayer`. Инициализация этого слоя предполагает использование нескольких дополнительных переменных, поскольку на этот раз нам также нужно позаботиться о матрице весов, смещении и соответствующих градиентах.

Листинг 5.16 ❖ Инициализация весов плотного слоя

```
class DenseLayer(Layer):

    def __init__(self, input_dim, output_dim):

        super(DenseLayer, self).__init__()

        self.input_dim = input_dim
        self.output_dim = output_dim

        self.weight = np.random.randn(output_dim, input_dim)
        self.bias = np.random.randn(output_dim, 1)

        self.params = [self.weight, self.bias]

        self.delta_w = np.zeros(self.weight.shape)
        self.delta_b = np.zeros(self.bias.shape)
```

Плотные слои имеют такие параметры, как размерность входных и выходных данных.

Произвольная инициализация матрицы весов и вектора смещения.

Параметрами слоя являются веса и члены смещения.

Значения дельты для весов и смещений приравнены к 0.

Обратите внимание на то, что инициализация W и b производится случайным образом. Существует множество способов инициализации весов нейронной сети. Случайная инициализация является вполне приемлемой, однако есть и более изощренные способы инициализации параметров, обеспечивающие более точное отражение структуры входных данных.

Прямой проход плотного слоя выполняется довольно просто.

Листинг 5.17 ❖ Прямой проход плотного слоя

```
    def forward(self):
        data = self.get_forward_input()
        self.output_data = np.dot(self.weight, data) + self.bias
```

Прямой проход плотного слоя сводится к аффинно-линейному преобразованию входных данных в соответствии со значениями весов и смещений.

Инициализация параметров как отправная точка процесса оптимизации

Инициализация параметров – это очень интересная тема, и мы обсудим некоторые дополнительные методы ее проведения в главе 6.

А пока просто имейте в виду, что инициализация влияет на процесс обучения. В случае с поверхностью потерь, изображенной на рис. 5.10, инициализация параметров означает *выбор начальной точки* для оптимизации. Легко понять, что выбор разных начальных точек для СГС на этой поверхности потерь может привести к разным результатам. По этой причине инициализация является важной темой при изучении нейронных сетей.

Что касается обратного прохода, помните о том, что для вычисления дельты для этого слоя достаточно транспонировать матрицу весов W и умножить ее на входящую дельту: $W^t\Delta$. Градиенты для W и b вычисляются тоже довольно легко: $\Delta W = \Delta y t$ и $\Delta b = \Delta$, где y – это вход данного слоя (оцененный с учетом используемых в настоящий момент данных).

Листинг 5.18 ❖ Обратный проход плотного слоя

```
def backward(self):
    data = self.get_forward_input()
    delta = self.get_backward_input()

    self.delta_b += delta

    self.delta_w += np.dot(delta, data.transpose())

    self.output_delta = np.dot(self.weight.transpose(), delta)
```

Для выполнения обратного прохода мы сначала получаем входные данные и значение дельты.

Текущая дельта прибавляется к дельте смещения.

Затем мы прибавляем это слагаемое к дельте веса.

Процесс обратного прохода завершается передачей выходной дельты на предыдущий слой.

Правило обновления для этого слоя задается накоплением значений дельты в соответствии со скоростью обучения, указанной для данной сети.

Листинг 5.19 ❖ Механизм обновления весов плотного слоя

```
def update_params(self, rate):
    self.weight -= rate * self.delta_w
    self.bias -= rate * self.delta_b

def clear_deltas(self):
    self.delta_w = np.zeros(self.weight.shape)
    self.delta_b = np.zeros(self.bias.shape)

def describe(self):
    print("|--- " + self.__class__.__name__)
    print("  |-- dimensions: ({},{})"
          .format(self.input_dim, self.output_dim))
```

Используя дельты веса и смещения, мы можем обновить параметры модели с помощью алгоритма градиентного спуска.

После обновления параметров необходимо сбросить все значения дельты.

Плотный слой может быть описан размерностью входных и выходных данных.

5.5.4. Создание последовательных нейронных сетей средствами языка Python

Теперь, когда мы разобрались со слоями – строительными блоками сети, пришло время заняться самими сетями. Для инициализации последовательной нейрон-

ной сети мы оснащаем ее пустым списком слоев и позволяем ей использовать СКО в качестве функции потерь, если не указано иное.

Листинг 5.20 ❖ Инициализация последовательной нейронной сети

```
class SequentialNetwork:
    def __init__(self, loss=None):
        print("Initialize Network...")
        self.layers = []
        if loss is None:
            self.loss = MSE()
```

В последовательной нейронной сети слои объединяются в стек последовательно.

Если функция потерь не предусмотрена, используется СКО.

Теперь нам нужно обеспечить функциональность для последовательного добавления слоев.

Листинг 5.21 ❖ Последовательное добавление слоев

```
    def add(self, layer):
        self.layers.append(layer)
        layer.describe()
        if len(self.layers) > 1:
            self.layers[-1].connect(self.layers[-2])
```

Каждый раз, когда мы добавляем слой, мы присоединяем его к предшественнику и позволяем ему описать себя.

В основе нашей реализации сети лежит метод `train`. В качестве входных данных используются мини-пакеты: мы перетасовываем обучающие данные и разбиваем их на пакеты размером `mini_batch_size`. Для обучения сети мы подаем ей один мини-пакет за другим. Чтобы повысить качество обучения, будем подавать обучающие данные на вход сети по нескольку раз, т. е. обучать ее в течение нескольких *эпох*. Для каждого мини-пакета мы вызываем метод `train_batch`. В случае предоставления тестовых данных `test_data` производительность сети оценивается после каждой эпохи обучения.

Листинг 5.22 ❖ Применение метода `train` для обучения последовательной сети

```
    def train(self, training_data, epochs, mini_batch_size,
              learning_rate, test_data=None):
        n = len(training_data)
        for epoch in range(epochs):
            random.shuffle(training_data)
            mini_batches = [
                training_data[k:k + mini_batch_size] for
                k in range(0, n, mini_batch_size)
            ]
            for mini_batch in mini_batches:
                self.train_batch(mini_batch, learning_rate)
            if test_data:
                n_test = len(test_data)
                print("Epoch {0}: {1} / {2}"
                      .format(epoch, self.evaluate(test_data), n_test))
            else:
                print("Epoch {0} complete".format(epoch))
```

Для обучения сети мы подаем данные на ее вход несколько раз в соответствии с заданным количеством эпох.

Перетасовка обучающих данных и создание мини-пакетов.

Обучение сети на каждом из мини-пакетов.

В случае предоставления тестовых данных производительность сети оценивается после каждой эпохи обучения.

Теперь метод `train_batch` выполняет вычисления для прямого и обратного проходов сети с использованием этого мини-пакета, а затем обновляет параметры.

Листинг 5.23 ❖ Обучение последовательной нейронной сети на основе пакета данных

```
def train_batch(self, mini_batch, learning_rate):
    self.forward_backward(mini_batch)
    self.update(mini_batch, learning_rate)
```

Чтобы обучить сеть на основе мини-пакета, мы выполняем вычисления для прямого и обратного проходов сети...

...а затем обновляем параметры модели.

Этапы `update` и `forward_backward` осуществляются следующим образом.

Листинг 5.24 ❖ Обновление параметров, а также прямой и обратный проходы сети

```
def update(self, mini_batch, learning_rate):
    learning_rate = learning_rate / len(mini_batch)
    for layer in self.layers:
        layer.update_params(learning_rate)
    for layer in self.layers:
        layer.clear_deltas()

def forward_backward(self, mini_batch):
    for x, y in mini_batch:
        self.layers[0].input_data = x
        for layer in self.layers:
            layer.forward()
        self.layers[-1].input_delta = \
            self.loss.loss_derivative(self.layers[-1].output_data, y)
        for layer in reversed(self.layers):
            layer.backward()
```

Распространенным методом является нормировка скорости обучения по размеру мини-пакета.

Обновление параметров всех слоев.

Сброс значений дельты во всех слоях.

Для каждого образца в мини-пакете мы передаем признаки вперед по сети.

Вычисление производной функции потерь для выходных данных.

Осуществление обратного распространения ошибки.

Данная реализация довольно проста, однако в ней есть несколько нюансов, требующих особого внимания. Во-первых, мы нормализуем скорость обучения по размеру мини-пакета, чтобы минимизировать обновления. Во-вторых, перед осуществлением полного обратного прохода сети путем обхода слоев в обратном порядке мы вычисляем производную функции потерь для выхода сети, которая послужит первой входной дельтой для выполнения обратного прохода.

Оставшаяся часть реализации `SequentialNetwork` касается производительности и оценки модели. Чтобы оценить сеть на тестовых данных, нужно передать эти данные вперед по сети. Именно за это отвечает метод `single_forward`. Метод `evaluate` осуществляет оценку и возвращает количество правильно предсказанных результатов.

Листинг 5.25 ❖ Оценка модели

```
def single_forward(self, x):
    self.layers[0].input_data = x
    for layer in self.layers:
            layer.forward()
    return self.layers[-1].output_data

def evaluate(self, test_data):
    test_results = [(
        np.argmax(self.single_forward(x)),
        np.argmax(y)
    ) for (x, y) in test_data]
    return sum(int(x == y) for (x, y) in test_results)
```

Передача одного образца вперед по сети и возврат результата.

Вычисление показателя верности предсказаний модели на тестовых данных.

5.5.5. Применение сети к задаче классификации рукописных цифр

Теперь, когда мы реализовали сеть прямого распространения, давайте вернемся к нашей исходной задаче распознавания рукописных цифр из набора данных MNIST. После импорта только что созданных классов мы загружаем данные MNIST, инициализируем сеть, добавляем в нее слои, обучаем ее на основе наших данных, а затем оцениваем ее производительность.

При создании сети помните о том, что размерность входного сигнала составляет 784, а размерность выходного сигнала – 10 (количество цифр). Мы выбираем три плотных слоя с размерностями выходных сигналов 392, 196 и 10 соответственно и добавляем после каждого из них сигмоидальные функции активации. Добавление каждого нового плотного слоя фактически вдвое уменьшает емкость слоя. Размеры и количество слоев являются *гиперпараметрами* этой сети. Эти значения были выбраны для настройки сетевой архитектуры. Мы рекомендуем вам поэкспериментировать со слоями других размеров, чтобы получить представление о зависимости процесса обучения сети от ее архитектуры.

Листинг 5.26 ❖ Создание экземпляра нейронной сети

```
from dlgo.nn import load_mnist
from dlgo.nn import network
from dlgo.nn.layers import DenseLayer, ActivationLayer

training_data, test_data = load_mnist.load_data()

net = network.SequentialNetwork()

net.add(DenseLayer(784, 392))
net.add(ActivationLayer(392))
net.add(DenseLayer(392, 196))
net.add(ActivationLayer(196))
net.add(DenseLayer(196, 10))
net.add(ActivationLayer(10))
```

Загрузка обучающих и тестовых данных.

Инициализация последовательной нейронной сети.

Поочередное добавление плотных слоев и слоев активации.

Размерность выходного слоя равна 10 в соответствии с количеством классов, которые требуется предсказать.

Для обучения сети на основе данных мы вызываем метод `train` со всеми необходимыми параметрами. Обучение длится на протяжении 10 эпох, в качестве скорости обучения задано значение 3,0, а в качестве размера мини-пакета – значение 10 в соответствии с количеством классов. В случае почти идеальной перетасовки обучающих данных в большинстве пакетов был бы представлен каждый из классов, что обеспечило бы хорошие результаты при применении стохастического градиентного спуска.

Листинг 5.27 ❖ Запуск экземпляра нейронной сети на обучающих данных

```
net.train(training_data, epochs=10, mini_batch_size=10,
          learning_rate=3.0, test_data=test_data)
```

Теперь мы можем легко обучить модель, предоставив обучающие и тестовые данные, задав количество эпох, размер мини-пакета и скорость обучения.

Выполните в командной строке команду:

```
python run_network.py
```

Вы увидите следующее:

```
Initialize Network...
|--- DenseLayer
  |-- dimensions: (784,392)
|-- ActivationLayer
  |-- dimensions: (392,192)
|--- DenseLayer
  |-- dimensions: (192,10)
|-- ActivationLayer
  |-- dimensions: (10,10)
Epoch 0: 6628 / 10000
Epoch 1: 7552 / 10000
...
```

Числа, полученные для каждой из эпох, в данном случае не имеют значения, важен лишь факт того, что результат сильно зависит от инициализации весов. Стоит отметить, что нам часто удается достичь более чем 95%-ной верности предсказаний менее чем за 10 эпох обучения. Это уже большое достижение, особенно если учесть, что данная сеть была создана с нуля. В частности, эта модель работает намного лучше, чем примитивный вариант, описанный в начале данной главы. Тем не менее этот результат можно улучшить.

Обратите внимание на то, что в рассмотренном случае мы полностью игнорировали пространственную структуру входных изображений и рассматривали их в качестве векторов. Однако совершенно очевидно, что пространство по соседству с данным пикселом содержит важную информацию, которую можно использовать. В конце концов, нам предстоит вернуться к игре в го, а в главах 2 и 3 мы говорили о том, насколько важным является учет непосредственного окружения (цепочки) камней.

В следующей главе вы узнаете о том, как создать нейронную сеть, позволяющую обнаруживать закономерности в таких пространственных данных, как изображения или состояния игровой доски. Это значительно поможет вам при разработке бота для игры в го, о котором мы поговорим в главе 7.

5.6. Резюме

- *Последовательная нейронная сеть* – это простая искусственная нейронная сеть, состоящая из набора расположенных друг за другом слоев. Нейронные сети можно применять для решения самых разнообразных задач машинного обучения, в том числе для распознавания образов.
- *Сеть прямого распространения* – это последовательная сеть, состоящая из плотных слоев с функцией активации.
- *Функции потерь* позволяют оценить качество предсказаний. Одной из наиболее часто используемых функций потерь является *среднеквадратическая ошибка*. Функция потерь позволяет количественно оценить верность предсказаний модели.
- *Градиентный спуск* – это алгоритм нахождения минимума функции. Градиентный спуск предполагает следование в направлении самого крутого спуска гра-

фика функции. В машинном обучении градиентный спуск используется для нахождения весов модели, позволяющих минимизировать потери.

- *Стохастический градиентный спуск* – это разновидность алгоритма градиентного спуска. При его использовании мы вычисляем градиент для небольшого подмножества обучающих данных, называемого *мини-пакетом*, а затем обновляем веса сети на основе каждого из них. При работе с большими обучающими наборами стохастический градиентный спуск, как правило, оказывается намного быстрее по сравнению с обычным градиентным спуском.
- При работе с последовательной нейронной сетью для эффективного расчета градиента можно использовать *алгоритм обратного распространения ошибки*. Комбинация обратного распространения ошибки и мини-пакетов позволяет достаточно сильно ускорить процесс обучения, чтобы сделать целесообразной работу с огромными наборами данных.

Глава 6

Создание нейронной сети для данных игры го

В этой главе:

- использование глубокого обучения для создания приложения, способного предсказывать следующий ход в игре го на основе имеющихся данных;
- знакомство с библиотекой глубокого обучения Keras;
- знакомство со сверточными нейронными сетями;
- создание нейронных сетей для анализа пространственных данных игры го.

В предыдущей главе мы познакомились с основными принципами работы нейронных сетей и реализовали сеть прямого распространения. В этой главе мы вернемся к игре го и поговорим о том, как использование методов глубокого обучения позволит предсказывать следующий ход для любого игрового состояния. В частности, мы сгенерируем игровые данные с помощью разработанных в главе 4 методов поиска по дереву, а затем применим их для обучения нейронной сети. Схема приложения, которое нам предстоит создать в этой главе, представлена на рис. 6.1.

Как видно на приведенной схеме, перед применением знаний о нейронных сетях, полученных в предыдущей главе, нам необходимо обсудить несколько важных нюансов.

1. В главе 3 мы обучали компьютер правилам игры, реализуя игровой процесс на доске для игры в го. В главе 4 рассматривались методы поиска по дереву. В главе 5 говорилось о том, что нейронные сети нуждаются в *числовых входных данных*, а для реализованной нами сети прямого распространения требовались *векторы*.
2. Для преобразования состояния доски для игры в го во входной вектор, который можно подать на вход нейронной сети, нам потребуется специальный *кодировщик*. На рис. 6.1 схематично представлен простой кодировщик, который нам предстоит реализовать в разделе 6.1. Состояние доски кодируется в виде матрицы размером с игровую доску. При этом черные камни кодируются значением 1, белые – значением –1, а пустые точки – значением 0. Эта матрица может быть преобразована в вектор, как в случае с данными MNIST, описанном в предыдущей главе. Несмотря на то что данное представление слишком примитивно для обеспечения выдающихся результатов в плане предсказания ходов, оно является первым шагом в правильном на-

правлении. В главе 7 мы обсудим более изощренные и эффективные способы кодирования состояния игровой доски.

3. Прежде чем приступать к обучению нейронной сети, мы должны подготовить для нее входные данные. В разделе 6.2 мы продолжим начатый в главе 4 разговор о методах создания записей игровых партий. Каждое закодированное состояние доски будет служить признаком, а следующий ход для каждого из состояний – меткой.
4. Помимо простой реализации нейронной сети, нам также важно повысить ее скорость и надежность за счет использования более продвинутой библиотеки глубокого обучения. С этой целью в разделе 6.3 будет представлена библиотека *Keras* – популярный инструмент глубокого обучения, написанный на языке Python. Мы используем Keras, чтобы смоделировать сеть для предсказания ходов.

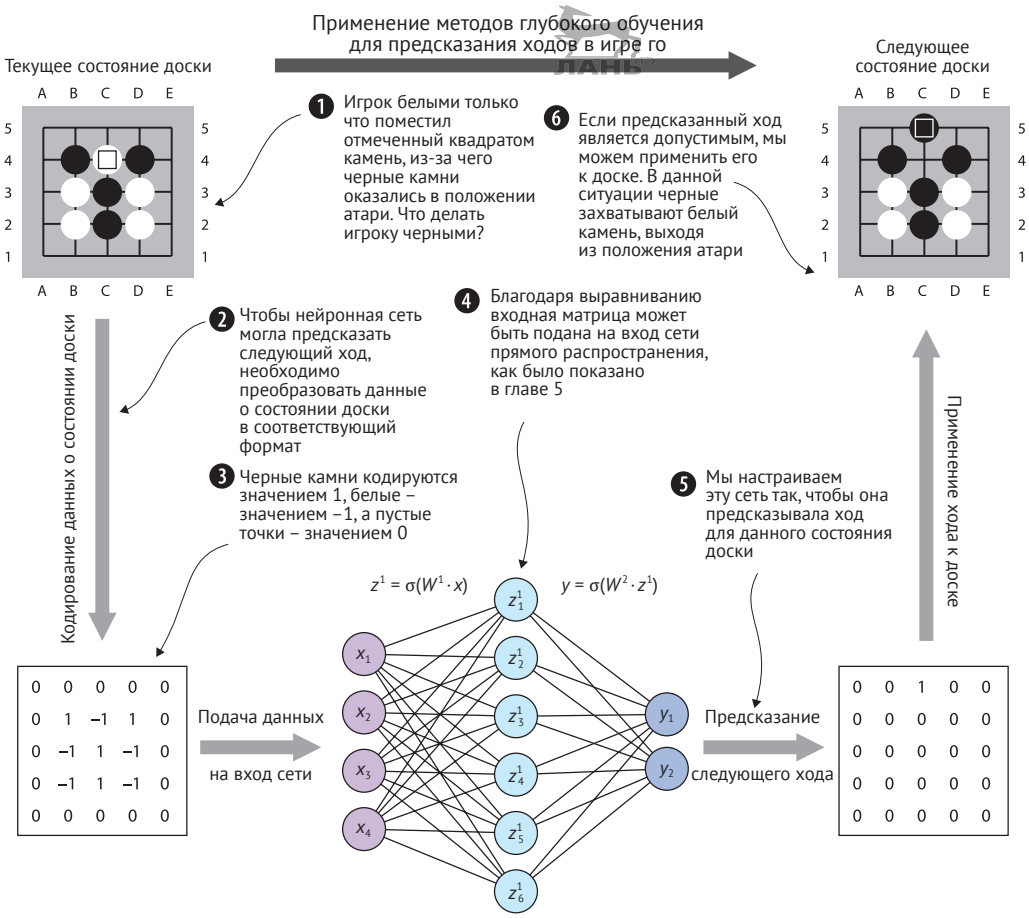


Рис. 6.1 ❖ Предсказание ходов в игре го с помощью методов глубокого изучения

5. На данном этапе вы можете задаться вопросом, почему мы полностью отбросили пространственную структуру доски для игры в го, преобразовав данные о ней в вектор. В разделе 6.4 вы узнаете о так называемом *сверточном слое*, который гораздо лучше подходит для решения стоящей перед нами задачи. Из этих слоев мы создадим *сверточную нейронную сеть*.
6. В конце этой главы мы поговорим о других ключевых концепциях глубокого обучения, позволяющих повысить верность предсказания ходов. В разделе 6.5 будет представлена функция *softmax* для эффективного предсказания вероятностей, а в разделе 6.6 мы обсудим создание более глубоких нейронных сетей с помощью интересной функции активации под названием *блок линейной ректификации* (ReLU).

6.1. Кодирование игрового состояния для подачи на вход нейронной сети

В главе 3 мы создали библиотеку классов Python, представляющих все сущности игры го: `Player`, `Board`, `GameState` и т. д. Теперь хотим применить методы машинного обучения для решения задач, связанных с этой игрой. Однако математические модели вроде нейронных сетей не способны работать с высокоуровневыми объектами наподобие класса `GameState` . Они могут иметь дело лишь с математическими объектами вроде векторов и матриц. В этом разделе мы создадим класс `Encoder` для преобразования объектов игры в математическую форму. В оставшейся части главы будем применять к полученному математическому представлению инструменты машинного обучения.

Первым этапом построения модели глубокого обучения для предсказания ходов в игре го является загрузка данных, которые можно подать на вход нейронной сети. Для этого мы определяем простой *кодировщик* для состояния доски, представленный на рис. 6.1. Кодировщик – это способ преобразования доски для игры в го, реализованной в главе 3, в подходящий формат. Многослойные перцептроны, о которых мы говорили до сих пор, принимают в качестве входных данных векторы, однако в разделе 6.4 вы узнаете о другой сетевой архитектуре, способной работать с данными более высокой размерности. На рис. 6.2 показан способ определения такого кодировщика.

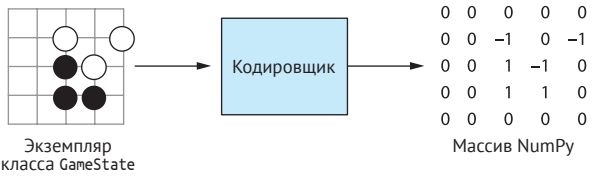


Рис. 6.2 ❖ Иллюстрация класса `Encoder`. Кодировщик принимает класс `GameState` и преобразует его в математическую форму – массив NumPy

Кодировщик должен уметь кодировать игровое состояние целиком. В частности, он должен определять способ кодирования отдельной точки доски. Иногда

интерес вызывает и обратная операция, когда закодированное предсказание сети требуется преобразовать в фактический ход на доске. Эта операция, называемая *декодированием*, является неотъемлемой частью процесса применения предсказанных ходов.

С учетом вышесказанного давайте определим класс `Encoder`, интерфейс для кодировщиков, которые мы создадим в этой и в следующей главах. Создайте новый модуль *encoders* в каталоге *dlgo*, который будет инициализироваться пустым файлом *__init__.py*, и поместите в него файл *base.py*. Затем добавьте в этот файл следующий код.

Листинг 6.1 ❖ Абстрактный класс `Encoder` для кодирования игрового состояния

```
class Encoder:
    def name(self):
        raise NotImplementedError()
    def encode(self, game_state):
        raise NotImplementedError()
    def encode_point(self, point):
        raise NotImplementedError()
    def decode_point_index(self, index):
        raise NotImplementedError()
    def num_points(self):
        raise NotImplementedError()
    def shape(self):
        raise NotImplementedError()
```

Позволяет регистрировать или сохранять имя кодировщика, используемого моделью.

Преобразует состояние доски для игры в го в числовые данные.

Преобразует точку доски в целочисленный индекс.

Преобразует целочисленный индекс обратно в точку доски для игры в го.

Количество точек на доске = ширина доски × высота доски.

Форма закодированной структуры доски.

Определение кодировщика является довольно простым, однако нам нужно добавить в файл *base.py* еще одну функцию, которая позволяет создать кодировщик путем указания его имени вместо создания объекта. Для этого необходимо добавить в определение кодировщиков функцию `get_encoder_by_name`.

Листинг 6.2 ❖ Создание кодировщика доски для игры в го путем указания его имени

```
import importlib

def get_encoder_by_name(name, board_size):
    if isinstance(board_size, int):
        board_size = (board_size, board_size)
    module = importlib.import_module('dlgo.encoders.' + name)
    constructor = getattr(module, 'create')
    return constructor(board_size)
```

Экземпляры кодировщика можно создавать путем указания их имени.

Если значением `board_size` является одно целое число, мы создаем на его основе квадратную доску.

Каждая реализация кодировщика должна предусматривать функцию `'create'` для создания экземпляра.

Теперь, когда мы разобрались с кодировщиками и способом их создания, давайте реализуем идею, представленную на рис. 6.2, и закодируем один цвет камней значением 1, другой – значением –1, а пустые точки – значением 0. Чтобы делать верные предсказания, модель также должна знать, за кем из игроков очередь хода. Поэтому, вместо того чтобы использовать значение 1 для черных и –1 для белых, мы будем использовать значение 1 для кодирования игрока, которому принадлежит право следующего хода, а –1 – для кодирования его противника. Мы

назовем этот кодировщик `OnePlaneEncoder`, поскольку будем кодировать игровую доску в виде матрицы или плоскости того же размера, что и доска. В главе 7 мы поговорим о кодировщиках с большим количеством *плоскостей признаков*. Например, мы реализуем кодировщик, который предусматривает по одной плоскости для черных и белых камней и одну плоскость для правила ко. А сейчас вам предстоит реализовать простой одноплоскостной кодировщик в файле *oneplane.py* из модуля *encoders*. Первая часть этой реализации приведена в следующем листинге.

Листинг 6.3 ❖ Кодирование игрового состояния с помощью простого одноплоскостного кодировщика доски для игры в го

```
import numpy as np

from dlgo.encoders.base import Encoder
from dlgo.goboard import Point

class OnePlaneEncoder(Encoder):
    def __init__(self, board_size):
        self.board_width, self.board_height = board_size
        self.num_planes = 1

    def name(self):                          <-- Мы можем сослаться на этот кодировщик, указав имя oneplane.
        return 'oneplane'

    def encode(self, game_state):            <-- Для выполнения кодирования мы помещаем в матрицу значение 1, если в точке доски находится камень игрока, которому принадлежит право следующего хода, значение -1, если в точке доски находится камень его противника, и значение 0, если точка доски пуста.
        board_matrix = np.zeros(self.shape())
        next_player = game_state.next_player
        for r in range(self.board_height):
            for c in range(self.board_width):
                p = Point(row=r + 1, col=c + 1)
                go_string = game_state.board.get_go_string(p)
                if go_string is None:
                    continue
                if go_string.color == next_player:
                    board_matrix[0, r, c] = 1
                else:
                    board_matrix[0, r, c] = -1
        return board_matrix
```

Во второй части определения мы позаботимся о кодировании и декодировании отдельных точек доски. Кодирование осуществляется путем отображения точки доски в вектор, длина которого равна произведению ширины доски на длину доски. Декодирование предполагает извлечение координат точки из такого вектора.

Листинг 6.4 ❖ Кодирование и декодирование точек с помощью одноплоскостного кодировщика доски для игры в го

```
    def encode_point(self, point):           <-- Преобразует точку доски в целочисленный индекс.
        return self.board_width * (point.row - 1) + (point.col - 1)

    def decode_point_index(self, index):     <-- Преобразует целочисленный индекс в точку доски.
        row = index // self.board_width
        col = index % self.board_width
        return Point(row=row + 1, col=col + 1)
```

```
    def num_points(self):
        return self.board_width * self.board_height

    def shape(self):
        return self.num_planes, self.board_height, self.board_width
```

Теперь, когда мы разобрались с кодировщиками доски для игры в го, давайте создадим данные, которые можно закодировать и подать на вход нейронной сети.

6.2. Генерирование обучающих игровых данных методом поиска по дереву

Перед применением методов машинного обучения к игре го необходимо подготовить набор обучающих данных. К счастью, сильные игроки очень часто играют на публичных го-серверах. В главе 7 мы поговорим о способах нахождения и обработки записей партий для создания обучающих данных. А пока мы можем создать собственные записи партий. В этом разделе показано, как это сделать с помощью созданных в главе 4 ботов, использующих поиск по дереву. В оставшейся части главы мы будем использовать полученные таким образом записи партий в качестве обучающих данных для проведения экспериментов в области глубокого обучения.

Не глупо ли использовать машинное обучение для имитации классического алгоритма? Нет, если традиционный алгоритм работает слишком медленно! Мы постараемся обеспечить более быструю альтернативу поиску по дереву с помощью машинного обучения. Эта концепция является ключевой для AlphaGo Zero, самой сильной версии алгоритма AlphaGo, принципы работы которой описаны в главе 14.

Создайте файл с именем *generate_mcts_games.py* за пределами модуля *dlgo*. Как следует из названия файла, в нем будет содержаться код, генерирующий игры методом Монте-Карло. Затем каждый ход созданных партий будет закодирован кодировщиком `OnePlaneEncoder`, описанным в разделе 6.1, и сохранен в массивах `numpy` для дальнейшего использования. Добавьте следующие операторы `import` в начало файла *generate_mcts_games.py*.

Листинг 6.5 ❖ Операторы `import` для создания закодированных игровых данных, сгенерированных методом Монте-Карло

```
import argparse
import numpy as np

from dlgo.encoders import get_encoder_by_name
from dlgo import goboard_fast as goboard
from dlgo import mcts
from dlgo.utils import print_board, print_move
```

Приведенные в листинге операторы `import` показывают, какие инструменты мы будем использовать для решения стоящей перед нами задачи: модуль *mct*, реализацию *goboard*, описанную в главе 3, и только что созданный модуль *encoders*. Теперь давайте создадим функцию для генерирования игровых данных. Функция `generate_game` запускает игру экземпляра `MCTSAgent` из главы 4 с самим собой (как

вы помните из главы 4, *температура* агента `MCTSAgent` регулирует волатильность поиска по дереву). Для каждого хода мы кодируем состояние доски перед его совершением, кодируем этот ход в виде унитарного вектора, а затем применяем ход к доске.

Листинг 6.6 ❖ Генерирование игровых данных для этой главы методом Монте-Карло

```
def generate_game(board_size, rounds, max_moves, temperature):
    boards, moves = [], []

    encoder = get_encoder_by_name('oneplane', board_size)

    game = goboard.GameState.new_game(board_size)

    bot = mcts.MCTSAgent(rounds, temperature)

    num_moves = 0
    while not game.is_over():
        print_board(game.board)
        move = bot.select_move(game)
        if move.is_play:
            boards.append(encoder.encode(game))

            move_one_hot = np.zeros(encoder.num_points())
            move_one_hot[encoder.encode_point(move.point)] = 1
            moves.append(move_one_hot)

        print_move(game.next_player, move)
        game = game.apply_move(move)
        num_moves += 1
        if num_moves > max_moves:
            break

    return np.array(boards), np.array(moves)
```

В `boards` будут храниться закодированные состояния доски, а в `moves` – закодированные ходы.

Инициализация кодировщика `OnePlaneEncoder` путем указания его имени и размера доски.

Создание новой игры на доске размером `board_size`.

В качестве бота будет использоваться агент поиска по дереву методом Монте-Карло с указанным количеством раундов и температурой.

Следующий ход выбирается ботом.

Закодированные данные о состоянии доски добавляются в `boards`.

Следующий ход, закодированный методом унитарного кодирования, добавляется в `moves`.

Ход бота применяется к доске.

Цикл повторяется, если не было достигнуто максимальное количество ходов.

Теперь, когда у нас есть средства для создания и кодирования игровых данных с помощью поиска по дереву методом Монте-Карло, мы определим в файле *generate_mcts_games.py* метод `main` для проведения и сохранения нескольких игр.

Листинг 6.7 ❖ Основное приложение для генерирования игр методом Монте-Карло

```
def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--board-size', '-b', type=int, default=9)
    parser.add_argument('--rounds', '-r', type=int, default=1000)
    parser.add_argument('--temperature', '-t', type=float, default=0.8)
    parser.add_argument('--max-moves', '-m', type=int, default=60,
                        help='Max moves per game.')
    parser.add_argument('--num-games', '-n', type=int, default=10)
    parser.add_argument('--board-out')
    parser.add_argument('--move-out')

    args = parser.parse_args()
    xs = []
    ys = []
```

Данное приложение позволяет производить настройку с помощью аргументов командной строки.

```
    for i in range(args.num_games):
        print('Generating game %d/%d...' % (i + 1, args.num_games))
        x, y = generate_game(args.board_size, args.rounds, args.max_moves,
    args.temperature)
        xs.append(x)
        ys.append(y)

    x = np.concatenate(xs)
    y = np.concatenate(ys)

    np.save(args.board_out, x)
    np.save(args.move_out, y)

if __name__ == '__main__':
    main()
```

Генерирование игровых данных для указанного количества игр.

После генерации всех игр производится конкатенация признаков и меток.

Данные о признаках и метках сохраняются в отдельных файлах в соответствии с параметрами командной строки.

С помощью этого инструмента мы можем легко генерировать игровые данные. Допустим, нам нужно сгенерировать данные для двадцати игр на доске 9×9, сохранить признаки в файле *features.npy*, а метки – в файле *label.npy*. Для этого можно использовать следующую команду:

```
python generate_mcts_games.py -n 20 --board-out features.npy --move-out labels.npy
```

Обратите внимание на то, что данный метод генерирования игр является довольно медленным, поэтому процесс создания большого количества игр может занять некоторое время. Мы могли бы уменьшить количество раундов для алгоритма Монте-Карло, но это снизило бы уровень игры бота. Поэтому мы сгенерировали для вас игровые данные и поместили их в папку *generated_games* в репозитории GitHub. Вы можете найти выходные данные в файлах *features40k.npy* и *label40k.npy*, которые содержат около 40 000 ходов, сгенерированных за несколько сотен игр. Каждый ход потребовал 5000 ММК-раундов. При таких параметрах ММК-движок генерирует довольно разумные ходы, поэтому можно надеяться, что нейронная сеть научится ему подражать.

Мы закончили с предварительной обработкой и теперь можем использовать сгенерированные данные для обучения нейронной сети. В качестве упражнения можно попробовать применить реализацию сети из главы 5, однако для работы с более сложными и глубокими нейронными сетями вам потребуется более мощный инструмент, например описанная далее библиотека Keras.

6.3. Использование библиотеки глубокого обучения Keras

Вычисление градиентов и обратного прохода нейронной сети постепенно отходит в прошлое благодаря появлению множества мощных библиотек глубокого обучения, скрывающих абстракции более низкого уровня. В прошлой главе вы приобрели полезный навык создания нейронных сетей с нуля, а сейчас пришло время познакомиться с более продвинутым и функциональным программным обеспечением.

Библиотека Keras – это популярный элегантный инструмент глубокого обучения, написанный на языке Python. Этот проект с открытым исходным кодом был создан в 2015 году и быстро привлек внимание большого количества пользователей. Код можно найти на странице **github.com/keras-team/keras**, а исчерпывающую документацию – на сайте **keras.io**.

6.3.1. Принципы проектирования с помощью библиотеки Keras

Одно из главных преимуществ библиотеки Keras заключается в том, что этот интуитивно понятный и простой в использовании API-интерфейс позволяет быстро создавать прототипы и проводить эксперименты. Из-за этого библиотека Keras часто используется участниками конкурсов, проводимых, например, на платформе **kaggle.com**. Библиотека Keras имеет модульную структуру и изначально была вдохновлена такими инструментами глубокого обучения, как Torch. Еще одним значительным преимуществом Keras является расширяемость. Добавление новых пользовательских слоев или расширение существующей функциональности не вызывает особых сложностей.

Библиотека Keras поставляется со всем необходимым, что позволяет быстро приступить к ее использованию. Например, непосредственно с библиотекой можно загрузить многие популярные наборы данных вроде MNIST, а в репозитории GitHub содержатся дополнительные полезные образцы. Кроме того, по адресу **github.com/fchollet/keras-resources** можно найти целую экосистему расширений Keras и множество независимых проектов.

Отличительной особенностью библиотеки Keras является концепция *бэкендов*: она работает с мощными движками, которые можно менять по требованию. Keras можно рассматривать как *фронтенд*, библиотеку, которая предоставляет удобный набор высокоуровневых абстракций и функций для работы с моделями и при этом поддерживается бэкендом, который выполняет всю тяжелую работу в фоновом режиме. На момент написания данной книги для библиотеки Keras было доступно три официальных бэкенда: Tensor Flow, Theano и Microsoft Cognitive Toolkit. В этой книге мы будем работать исключительно с библиотекой TensorFlow от компании Google, которая используется в Keras по умолчанию. Однако если вы предпочитаете другой вариант, смена бэкенда не составит особого труда.

В этом разделе мы установим библиотеку Keras. После этого изучим ее API на примере классификации рукописных цифр из главы 5, а затем перейдем к решению задачи предсказания ходов в игре го.

6.3.2. Установка библиотеки глубокого обучения Keras

Перед началом работы с Keras необходимо установить бэкенд. Вы можете начать с TensorFlow, который легко устанавливается с помощью следующей команды:

```
pip install tensorflow
```

При наличии графического процессора NVIDIA и актуальных драйверов CUDA вы можете попробовать установить версию TensorFlow с поддержкой GPU-ускорения:

```
pip install tensorflow-gpu
```

Если `tensorflowgpu` совместим с вашим оборудованием и драйверами, его установка позволит вам значительно увеличить скорость работы.

Для Keras можно установить несколько необязательных зависимостей, которые бывают полезны для сериализации и визуализации моделей, однако сейчас мы пропустим этот шаг и перейдем непосредственно к установке самой библиотеки:

```
pip install Keras
```

6.3.3. Применение библиотеки Keras к рассмотренному ранее примеру

В этом разделе мы рассмотрим процесс определения и запуска моделей Keras, который состоит из четырех этапов.

1. **Предварительная обработка данных** – загрузка и подготовка набора данных для подачи на вход нейронной сети.
2. **Определение модели** – создание экземпляра модели и добавление в нее слоев.
3. **Компиляция модели** – компиляция ранее определенной модели с оптимизатором, функцией потерь и необязательным списком оценочных метрик.
4. **Обучение и оценка модели** – подгонка модели глубокого обучения к данным и ее оценка.

Чтобы познакомиться с библиотекой Keras, мы используем уже знакомый пример предсказания рукописных цифр из набора данных MNIST. Как вы скоро убедитесь, наша простая модель из главы 5 уже очень близка к синтаксису Keras, что еще больше упрощает задачу применения этой библиотеки.

Keras позволяет определить два типа моделей: последовательные и более общие непоследовательные модели. Здесь мы будем использовать только последовательные. В *keras.models* можно найти модели обоих типов. Чтобы определить последовательную модель, мы должны добавить в нее слои, как это делалось в реализации, описанной в главе 5. Слои Keras доступны через модуль *keras.layers*. Загрузить набор данных MNIST с библиотекой Keras очень просто. Этот набор данных находится в модуле *keras.datasets*. Сначала импортируем все необходимое для решения поставленной задачи.

Листинг 6.8 ❖ Импорт моделей, слоев и наборов данных из библиотеки Keras

```
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
```

Теперь нам нужно загрузить и выполнить предварительную обработку данных MNIST, для чего потребуется лишь несколько строк кода. После загрузки мы выравниваем 60 000 обучающих и 10 000 тестовых образцов, преобразуем их в тип `float`, а затем нормализуем входные данные путем деления на 255. Это связано с тем, что значения пикселов варьируются от 0 до 255, поэтому мы преобразуем их в значения, принадлежащие диапазону [0, 1], чтобы повысить качество обучения сети. Кроме того, нам нужно выполнить унитарное кодирование меток, как это делалось в главе 5. Следующий листинг демонстрирует выполнение описанных выше действий с помощью библиотеки Keras.

Листинг 6.9 ❖ Загрузка и предварительная обработка данных MNIST с помощью Keras

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
```

```
x_train /= 255
x_test /= 255

y_train = keras.utils.to_categorical(y_train, 10)
y_test = keras.utils.to_categorical(y_test, 10)
```

После подготовки данных можно приступать к определению нейронной сети. В Keras мы инициализируем последовательную модель `Sequential`, а затем добавляем слои один за другим. В первом слое мы должны указать *форму* входных данных с помощью аргумента `input_shape`. В нашем случае входные данные представляют собой вектор длиной 784, поэтому мы сообщаем информацию о форме этих данных в виде `input_shape =(784,)`. Плотные слои `Dense` в Keras можно создать с помощью ключевого слова `activation`, которое позволяет добавить слой с функцией активации. В качестве функции активации мы указываем единственную известную нам на данный момент сигмоиду `sigmoid`. Библиотека Keras предусматривает гораздо больше функций активации. Некоторые из них мы обсудим далее.

Листинг 6.10 ❖ Создание простой последовательной модели с помощью библиотеки Keras

```
model = Sequential()
model.add(Dense(392, activation='sigmoid', input_shape=(784,)))
model.add(Dense(196, activation='sigmoid'))
model.add(Dense(10, activation='sigmoid'))
model.summary()
```

Следующим этапом процесса создания модели является ее *компиляция* с функцией потерь и оптимизатором. Для этого используются строки. В качестве оптимизатора мы укажем `sgd` (стохастический градиентный спуск), а в качестве функции потерь – `mean_squared_error` (среднеквадратическая ошибка). Опять же, Keras содержит гораздо больше функций потерь и оптимизаторов, однако в данном случае мы используем те, с которыми познакомились в главе 5. Еще одним аргументом, который можно передать на этапе компиляции моделей Keras, является список оценочных метрик. Для первого приложения мы используем единственную метрику `accuracy`, показывающую, насколько часто предсказание модели с самым высоким значением совпадает с истинной меткой.

Листинг 6.11 ❖ Компиляция модели глубокого обучения Keras

```
model.compile(loss='mean_squared_error',
              optimizer='sgd',
              metrics=['accuracy'])
```

Последним этапом является обучение сети и ее оценка на тестовых данных. Для этого мы вызываем метод `fit` модели `model`, предоставляя, помимо обучающих данных, размер мини-пакета, с которым предстоит работать, и количество эпох обучения.

Листинг 6.12 ❖ Обучение и оценка модели Keras

```
model.fit(x_train, y_train,
          batch_size=128,
          epochs=20)
score = model.evaluate(x_test, y_test)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

Подытожим. Процесс создания и запуска модели Keras включает четыре этапа: предварительная обработка данных, определение модели, ее компиляция, а также обучение и оценка. Одно из важнейших преимуществ библиотеки Keras заключается в том, что этот четырехэтапный процесс выполняется довольно быстро, что ускоряет экспериментальный цикл. Это очень важно, поскольку настройка параметров зачастую позволяет улучшить исходную версию модели.

6.3.4. Предсказание ходов в игре го с помощью нейронной сети прямого распространения и библиотеки Keras

Теперь, когда мы обсудили API Keras для последовательных нейронных сетей, давайте вернемся к задаче предсказания ходов в игре го. Этот этап процесса проиллюстрирован на рис. 6.3. Сначала мы загружаем игровые данные, сгенерированные в разделе 6.2, как показано в листинге 6.13. Обратите внимание на то, что, как и в случае с набором MNIST, нам необходимо преобразовать данные о доске в векторы.

Рис. 6.3 ❖ Нейронная сеть способна предсказывать игровые ходы. Мы можем подать закодированное в виде матрицы игровое состояние на вход модели, а на выходе получить вектор, содержащий значения вероятности для каждого из возможных ходов

Листинг 6.13 ❖ Загрузка и предварительная обработка ранее сохраненных игровых данных

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense

np.random.seed(123)
X = np.load('../generated_games/features-40k.npy')
Y = np.load('../generated_games/labels-40k.npy')
samples = X.shape[0]
board_size = 9 * 9

X = X.reshape(samples, board_size)
Y = Y.reshape(samples, board_size)

train_samples = int(0.9 * samples)
X_train, X_test = X[:train_samples], X[train_samples:]
Y_train, Y_test = Y[:train_samples], Y[train_samples:]
```

Использование функции `random.seed()` гарантирует воспроизводимость сценария.

Загрузка данных образца в массивы NumPy.

Преобразование входных данных в векторы длиной 81 вместо матриц 9×9.

Выделение 10 % данных в тестовый набор; обучение сети на оставшихся 90 % данных.

Теперь давайте определим и запустим модель для предсказания ходов с использованием только что определенных признаков *X* и меток *Y*. Для доски 9×9 количество допустимых ходов составляет 81, поэтому наша сеть должна уметь предсказывать 81 класс. Представьте, что вы закрыли глаза и указали на случайную

точку доски. С вероятностью 1 к 81, или 1,2 %, мы можем предсказать следующий ход по чистой случайности. Поэтому верность предсказаний нашей модели должна значительно превышать показатель 1,2 %.

Определим простой многослойный перцептрон Keras с тремя плотными слоями (`Dense`), каждый из которых предусматривает сигмоидальную функцию активации (`sigmoid`). При компиляции в качестве функции потерь используем среднеквадратическую ошибку, а в качестве оптимизатора – стохастический градиентный спуск. Затем мы обеспечим обучение этой сети в течение 15 эпох и оценим качество ее предсказаний на тестовых данных.

Листинг 6.14 ❖ Обучение многослойного перцептрона Keras на сгенерированных игровых данных


```
model = Sequential()
model.add(Dense(1000, activation='sigmoid', input_shape=(board_size,)))
model.add(Dense(500, activation='sigmoid'))
model.add(Dense(board_size, activation='sigmoid'))
model.summary()

model.compile(loss='mean_squared_error',
              optimizer='sgd',
              metrics=['accuracy'])

model.fit(X_train, Y_train,
          batch_size=64,
          epochs=15,
          verbose=1,
          validation_data=(X_test, Y_test))

score = model.evaluate(X_test, Y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

После выполнения этого кода на консоли появится следующая информация о модели:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 1000)              82000
_________________________________________________________________
dense_2 (Dense)              (None, 500)               500500
_________________________________________________________________
dense_3 (Dense)              (None, 81)                40581
=================================================================
Total params: 623,081
Trainable params: 623,081
Non-trainable params: 0
_________________________________________________________________

...

Test loss: 0.0129547887068
Test accuracy: 0.0236486486486
```

Обратите внимание на строку `Trainable params: 623,081` в выходных данных. Она означает, что в процессе обучения обновляется более 600 000 отдельных весовых коэффициентов. Это приблизительный показатель вычислительной интенсивности модели. Кроме того, это значение дает нам общее представление о *емкости* модели, т. е. о ее способности изучать сложные отношения. При сравнении различных сетевых архитектур общее количество параметров позволяет приблизительно сопоставить общий размер моделей.

Как видите, верность предсказаний в нашем эксперименте составляет около 2,3 %, что на первый взгляд кажется неудовлетворительным. Однако, учитывая, что случайное угадывание обеспечивает верность предсказания около 1,2 %, это значение говорит о том, что модель обучается, и ее применение позволяет получить более качественные предсказания по сравнению со случайным угадыванием.

Мы можем получить представление о модели, подав на ее вход образцы состояния доски. На рис. 6.4 показана доска, которую мы создали специально, чтобы сделать правильный ход очевидным. Тот, кому принадлежит право следующего хода, может захватить два камня противника, заполнив точку A или B. Данное состояние доски отсутствует в обучающем наборе.

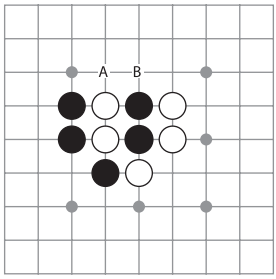


Рис. 6.4 ❖ Пример игрового состояния для тестирования модели. В данном случае черные могут захватить два камня, заполнив точку A, а белые могут захватить два камня, заполнив точку B. Тот, кто сделает такой ход первым, получит огромное преимущество в игре

Теперь мы можем подать это состояние доски на вход обученной модели и вывести на экран ее предсказания.

Листинг 6.15 ❖ Оценка модели с использованием известного состояния доски

```
test_board = np.array([[
    0, 0,  0,  0,  0, 0, 0, 0, 0,
    0, 0,  0,  0,  0, 0, 0, 0, 0,
    0, 0,  0,  0,  0, 0, 0, 0, 0,
    0, 1, -1,  1, -1, 0, 0, 0, 0,
    0, 1, -1,  1, -1, 0, 0, 0, 0,
    0, 0,  1, -1,  0, 0, 0, 0, 0,
    0, 0,  0,  0,  0, 0, 0, 0, 0,
    0, 0,  0,  0,  0, 0, 0, 0, 0,
    0, 0,  0,  0,  0, 0, 0, 0, 0,
]])
move_probs = model.predict(test_board)[0]
```

```
i = 0
for row in range(9):
    row_formatted = []
    for col in range(9):
        row_formatted.append('{:.3f}'.format(move_probs[i]))
        i += 1
    print(' '.join(row_formatted))
```

Выходные данные будут выглядеть примерно следующим образом:

```
0.037 0.037 0.038 0.037 0.040 0.038 0.039 0.038 0.036
0.036 0.040 0.040 0.043 0.043 0.041 0.042 0.039 0.037
0.039 0.042 0.034 0.046 0.042 0.044 0.039 0.041 0.038
0.039 0.041 0.044 0.046 0.046 0.044 0.042 0.041 0.038
0.042 0.044 0.047 0.041 0.045 0.042 0.045 0.042 0.040
0.038 0.042 0.045 0.045 0.045 0.042 0.045 0.041 0.039
0.036 0.040 0.037 0.045 0.042 0.045 0.037 0.040 0.037
0.039 0.040 0.041 0.041 0.043 0.043 0.041 0.038 0.037
0.036 0.037 0.038 0.037 0.040 0.039 0.037 0.039 0.037
```

Эта матрица соответствует исходной доске 9×9: каждое число означает степень уверенности модели в том, что следующий ход будет совершен в конкретной точке. Этот результат не является очень впечатляющим – модель пока даже не научилась игнорировать уже заполненные камнями точки доски. Однако обратите внимание на то, что оценки по краю доски ниже, чем оценки, находящиеся ближе к ее центру. В игре го принято помещать камни на самом краю доски только в конце игры и в каких-то исключительных ситуациях. Наша модель выявила это негласное правило не путем усвоения понятий стратегии или эффективности, а просто копируя действия MMK-бота. Эта модель вряд ли сможет предсказать множество удачных ходов, но она научилась избегать целого класса неудачных.

Это уже прогресс, однако мы можем добиться гораздо большего. Далее мы обсудим недостатки нашей экспериментальной версии и поработаем над повышением верности модели для предсказания ходов в игре го. При этом мы осудим следующие моменты:

- данные, которые используются для решения этой задачи предсказания, были *сгенерированы методом поиска по дереву*, который предполагает значительный элемент случайности. Иногда MMK-движки генерируют странные ходы, особенно когда они сильно лидируют или отстают в игре. В главе 7 мы создадим модель глубокого обучения на основе партий, сыгранных человеком. Разумеется, люди тоже непредсказуемы, однако они менее склонны делать бессмысленные ходы;
- используемую нами архитектуру нейронной сети можно значительно улучшить. Многослойные перцептроны не очень подходят для захвата данных о состоянии доски для игры в го. Мы вынуждены преобразовывать данные о двумерной доске в плоский вектор, теряя при этом всю пространственную информацию о ней. В разделе 6.4 мы поговорим о другом типе сети, которая гораздо лучше справляется с захватом структуры доски для игры в го;
- во всех рассмотренных ранее сетях мы использовали только сигмоидальную функцию активации. В разделах 6.5 и 6.6 мы обсудим две новые функции активации, которые часто обеспечивают более качественные результаты;

- до этого момента мы использовали в качестве функции потерь только СКО, которая является интуитивно понятной, но не вполне подходит для нашего случая использования. В разделе 6.5 мы используем функцию потерь, которая специально предназначена для решения задач классификации.

После обсуждения всех этих тем в конце главы мы создадим нейронную сеть, которая будет предсказывать ходы гораздо лучше, чем наша первая версия. А в главе 7 обсудим ключевые методы для создания значительно более мощного бота.

Помните о том, что нашей конечной целью является не максимально точное предсказание ходов, а создание бота, способного хорошо играть. Даже если ваши глубокие нейронные сети никогда не смогут делать выдающиеся предсказания на основе исторических данных, мощь глубокого обучения заключается в том, что эти сети способны неявно выявлять *структуру игры* и делать разумные, а иногда и весьма удачные ходы.

6.4. Анализ пространства с помощью сверточных сетей

В игре го часто встречаются особые локальные схемы расположения камней. Игроки-люди учатся распознавать эти формы и часто дают им различные названия. Чтобы сравняться с человеком в принятии решений, нашему ИИ для игры в го тоже придется научиться распознавать множество таких форм. Для подобных задач используются так называемые *сверточные нейронные сети*, или СНС. Сверточные нейронные сети применяются не только к играм, но и к изображениям, аудиозаписям и даже тексту. В этом разделе мы поговорим о том, как создавать СНС и применять их к данным игры го. Сначала введем понятие свертки. Далее речь пойдет о создании СНС с помощью библиотеки Keras. Наконец, мы обсудим способы обработки выходных данных сверточного слоя.

6.4.1. Назначение сверточных слоев

Сверточные слои и состоящие из них сети получили свое название от традиционной операции, применяемой в области компьютерного зрения. *Свертка* – это простой способ трансформации изображения или применения фильтра. Для двух матриц одного размера простая свертка вычисляется в два этапа:

1) поэлементное умножение двух матриц;
2) суммирование всех значений полученной матрицы.

Результатом такой простой свертки является скалярное значение. На рис. 6.5 показан пример свертки двух матриц 3×3 для вычисления скаляра.

Эти простые свертки не окажут нам непосредственной помощи, однако их можно использовать для вычисления более сложных сверток, которые могут оказаться полезными для решения стоящей перед нами задачи. Вместо того чтобы начинать с двух матриц одинакового размера, давайте зафиксируем размер второй матрицы и произвольно увеличим размер первой. В этом сценарии мы будем называть первую матрицу *входным изображением*, а вторую – *ядром свертки* или просто *ядром* (иногда ее также называют *фильтром*). Поскольку ядро меньше входного изображения, мы можем вычислить простые свертки для многих *окон* входного изображения. На рис. 6.6 продемонстрировано применение операции свертки к входному изображению 10×10 с использованием ядра 3×3.

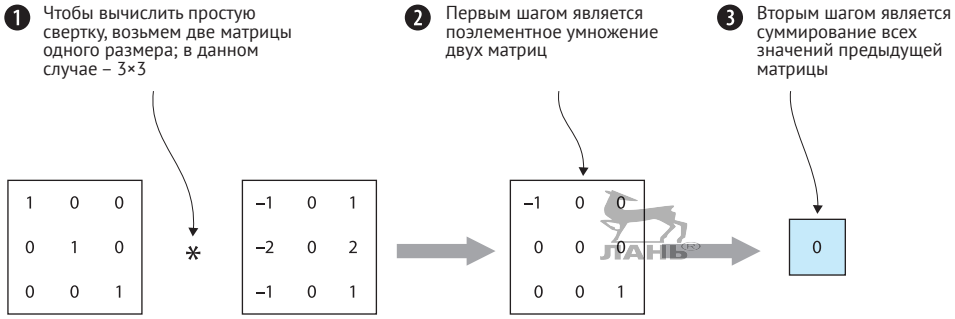


Рис. 6.5 ❖ Простая свертка предполагает поэлементное умножение двух матриц одного размера и последующее суммирование полученных значений

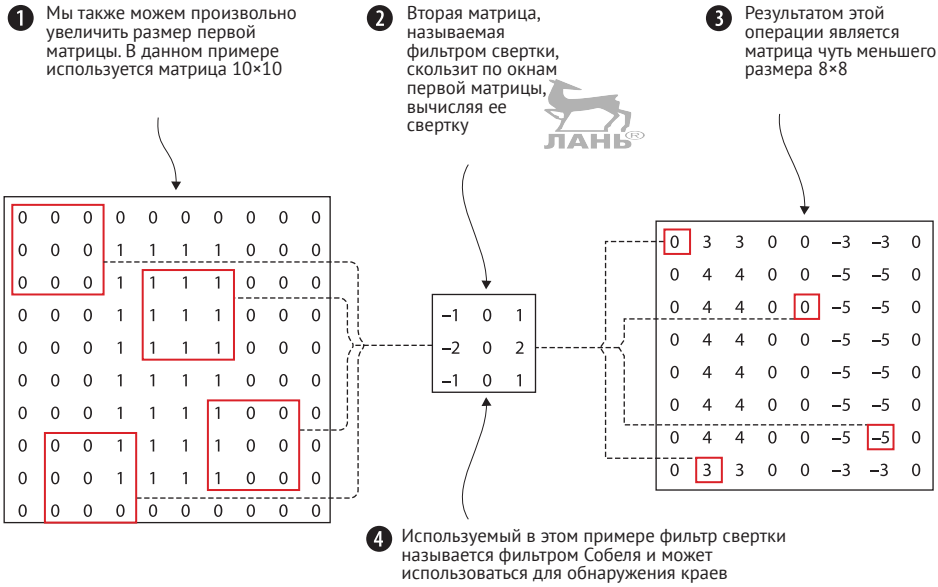


Рис. 6.6 ❖ Скольжение ядра свертки по окнам входного изображения позволяет вычислить свертку этого изображения. Используемое в этом примере ядро является детектором вертикальных краев

Пример, изображенный на рис. 6.6, позволяет получить некоторое представление о важности сверток. Входное изображение представляет собой матрицу 10×10, состоящую из центрального 4×8 блока единиц, окруженного нулями. Ядро выбрано так, чтобы первый столбец матрицы состоял из отрицательных значений (–1, –2, –1), третий столбец – из таких же значений, но противоположных по знаку (1, 2, 1), а средний столбец – из одних 0. Таким образом, справедливы следующие утверждения:

- всякий раз, когда это ядро применяется к окну 3×3 входного изображения, в котором все значения пикселов одинаковы, результатом вычисления свертки будет 0;

- когда мы применяем это ядро свертки к окну изображения, в котором левый столбец пикселов имеет более высокие значения, чем правый, результат вычисления свертки будет отрицательным;
- когда мы применяем это ядро свертки к окну изображения, в котором правый столбец пикселов имеет более высокие значения, чем левый, результат вычисления свертки будет положительным.

Ядро свертки используется для обнаружения *вертикальных краев во входном изображении*. Края слева от объекта будут иметь положительные значения, а края справа – отрицательные. Именно такой результат свертки виден на рис. 6.6.

На рис. 6.6 продемонстрировано классическое *ядро Собеля*, используемое во многих приложениях. Если перевернуть это ядро на 90 градусов, мы получим детектор горизонтальных краев. Таким же образом мы можем определить ядра свертки, способные размывать или повышать резкость изображения, обнаруживать углы и т. д. Многие из таких ядер можно найти в стандартных библиотеках для обработки изображений.

Важно то, что операцию свертки можно использовать для извлечения ценных данных из изображения. Именно это мы и будем делать для предсказания ходов в игре го. Несмотря на то что в предыдущем примере мы выбрали определенное ядро свертки, при использовании нейронной сети эти ядра формируются самостоятельно в ходе обучения сети методом обратного распространения ошибки.

До этого момента мы говорили о применении одного ядра свертки к одному входному изображению. Гораздо полезнее применить несколько ядер ко многим изображениям для получения множества выходных изображений. Как это сделать? Допустим, мы имеем четыре входных изображения и определяем четыре ядра. Затем мы можем суммировать результаты свертки всех входных изображений, чтобы получить одно выходное изображение. В дальнейшем мы будем называть выходные изображения, полученные в результате таких сверток, *картами признаков*. Если мы хотим получить не одну, а пять карт признаков, то должны определить пять ядер для каждого входного изображения вместо одного. Отображение n входных изображений в m карт признаков с помощью ядер свертки $n \times m$ называется *сверточным слоем* (см. рис. 6.7).

С этой точки зрения сверточный слой представляет собой способ преобразования множества входных изображений в выходные, позволяющий извлечь релевантную пространственную информацию из входных данных. Сверточные слои можно объединить в *цепочку*, создав из них сеть. Обычно сеть, состоящая только из сверточных и плотных слоев, называется *сверточной нейронной сетью* или просто *сверточной сетью*.

Обратите внимание на то, что до этого мы говорили лишь о передаче данных через сверточный слой, но не об обратном распространении ошибки. Обсуждение этой темы выходит за рамки данной книги, кроме того, библиотека Keras решает задачи, связанные с обратным проходом сети, за нас.

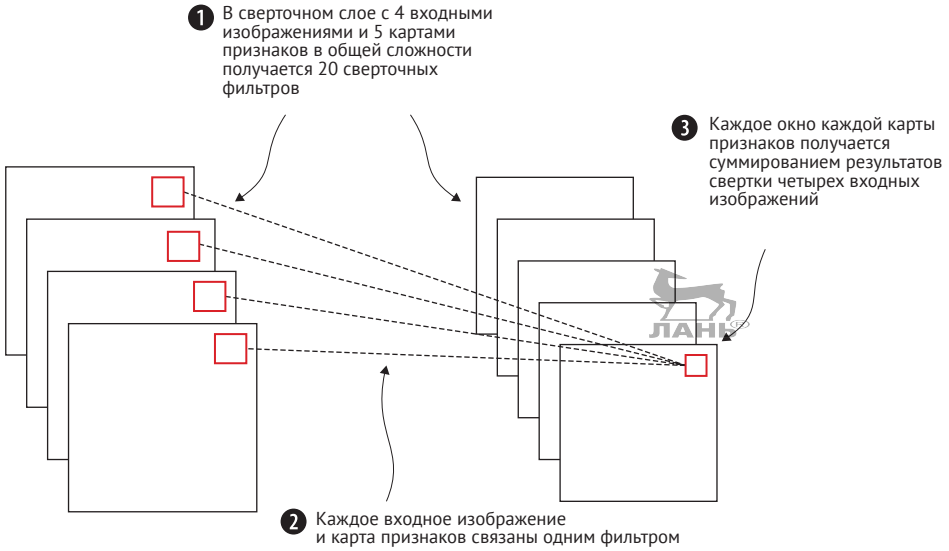


Рис. 6.7 ❖ В сверточном слое несколько входных изображений обрабатываются ядрами свертки, в результате чего получается заданное количество карт признаков

Использование тензоров в глубоком обучении

Мы уже сказали, что выходом сверточного слоя является набор изображений. Такое представление является полезным, но не исчерпывающим. Одномерные векторы состоят из отдельных записей, но они не являются просто набором чисел. Точно так же двумерные матрицы состоят из вектор-столбцов, но обладают двумерной структурой, которая используется при выполнении матричного умножения и других операций (вроде свертки). Выход сверточного слоя имеет трехмерную структуру. Фильтры в сверточном слое имеют на одно измерение больше и обладают 4-мерной структурой (2D-фильтр для каждой комбинации входного и выходного изображений). И это еще не все, продвинутые методы глубокого обучения предполагают использование структур данных еще большей размерности.

В линейной алгебре эквивалентом векторов и матриц большей размерности является *тензор*. Подробное определение тензора вы можете найти в приложении А, однако в остальной части книги оно вам не потребуется. Тем не менее концепция тензоров предоставляет удобную терминологию, которая пригодится нам в следующих главах. Например, набор изображений, являющийся выходом сверточного слоя, можно назвать 3-тензором. 4-мерные фильтры в сверточном слое образуют 4-тензор. Таким образом, можно сказать, что свертка – это операция, при которой 4-тензор (сверточные фильтры) преобразует 3-тензор (входные изображения) в другой 3-тензор.

В более общем смысле можно сказать, что последовательная нейронная сеть – это механизм, который шаг за шагом преобразует тензоры различной размерности. Идея «потока» входных данных через сеть при использовании тензоров вдохновила компанию Google назвать свою популярную библиотеку для машинного обучения TensorFlow. Именно эту библиотеку мы будем использовать для запуска моделей Keras.

Как правило, сверточный слой имеет гораздо меньше параметров, чем сопоставимый с ним плотный слой. Если бы мы определили сверточный слой с размером ядра (3, 3), применяемый к входному изображению 28×28 для получения выходного изображения 26×26, то этот сверточный слой имел бы всего 3 × 3 = 9 параметров. Как правило, сверточный слой также включает *член смещения*, прибавляемый к результату каждой свертки, что дает в общей сложности 10 параметров. Для сравнения, плотный слой, соединяющий входной вектор длиной 28×28 с выходным вектором длиной 26×26, будет иметь 28 × 28 × 26 × 26 = 529 984 параметра, не считая смещений. В то же время операции свертки являются более затратными в вычислительном отношении, чем обычное матричное умножение, применяемое в плотных слоях.

6.4.2. Создание сверточных сетей с помощью библиотеки Keras

Для создания и запуска сверточных нейронных сетей с помощью библиотеки Keras нам потребуется новый тип слоев `Conv2D`, который выполняет свертку двумерных данных вроде состояния доски для игры в го. Кроме того, нам понадобится слой выравнивания `Flatten`, который преобразует выход сверточного слоя в векторы, которые затем можно подать на вход плотного слоя.

Теперь этап предварительной обработки входных данных выглядит немного иначе. Вместо выравнивания данных о состоянии игровой доски мы сохраняем ее двумерную структуру.

Листинг 6.16 ❖ Загрузка и предварительная обработка игровых данных для сверточных нейронных сетей

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Conv2D, Flatten

np.random.seed(123)
X = np.load('../generated_games/features-40k.npy')
Y = np.load('../generated_games/labels-40k.npy')

samples = X.shape[0]
size = 9
input_shape = (size, size, 1)

X = X.reshape(samples, size, size, 1)

train_samples = int(0.9 * samples)
X_train, X_test = X[:train_samples], X[train_samples:]
Y_train, Y_test = Y[:train_samples], Y[train_samples:]
```

Импорт двух новых слоев: двумерного сверточного слоя и слоя, преобразующего входные данные в векторы.

Входные данные имеют трехмерную форму; доска представляется в виде плоскости 9×9.

Изменение формы входных данных.

Теперь можно создать сеть с помощью объекта `Conv2D` библиотеки Keras. Мы используем два сверточных слоя, а затем *выравниваем* выход второго из них, после чего применяем два плотных слоя, чтобы в итоге получить выход размером 9×9, как и раньше.

Листинг 6.17 ❖ Создание простой сверточной нейронной сети для данных игры го с помощью библиотеки Keras

```
model = Sequential()
model.add(Conv2D(filters=48,
                 kernel_size=(3, 3),
                 activation='sigmoid',
                 padding='same',
                 input_shape=input_shape))
model.add(Conv2D(48, (3, 3),
                 padding='same',
                 activation='sigmoid'))
model.add(Flatten())
model.add(Dense(512, activation='sigmoid'))
model.add(Dense(size * size, activation='sigmoid'))
model.summary()
```

Первым слоем сети является Conv2D с 48 выходными фильтрами.

Для этого слоя выбрано ядро свертки 3×3.

Обычно размер выхода свертки меньше, чем размер ее входа. С помощью фрагмента `padding='same'` мы даем библиотеке Keras команду добавить ряды нулей по краям матрицы, чтобы размеры входа и выхода были одинаковыми.

Второй слой представляет собой еще одну свертку. Фильтры и аргументы `kernel_size` опущены для краткости.

Затем мы выравниваем 3-мерный выход предыдущего сверточного слоя...

...и добавляем еще два плотных слоя, как мы это делали в примере с многослойным перцептроном.

Компиляция, запуск и оценка модели осуществляются так же, как в примере с многослойным перцептроном. Изменяется только форма входных данных и спецификация самой модели.

Если мы запустим описанную выше модель, то увидим, что степень верности ее предсказаний практически не изменилась и составляет около 2,3 %. Это нормально, у нас в запасе есть еще несколько хитростей, позволяющих раскрыть потенциал нашей сверточной модели. В оставшейся части этой главы мы обсудим более продвинутые техники глубокого обучения, помогающие повысить степень верности предсказания ходов.

6.4.3. Сокращение пространственной размерности с помощью слоев пулинга

В большинстве приложений, использующих глубокое обучение и сверточные слои, часто применяется метод *пулинга*. Пулинг позволяет уменьшить размер изображения и сократить количество нейронов в слое.

Концепция пулинга довольно проста: мы сокращаем изображение, группируя или уплотняя участки изображения до одного пиксела. На рис. 6.8 показано, как можно уменьшить изображение в 4 раза, оставив только максимальное значение в каждом непересекающемся участке изображения размером 2×2.

Этот метод называется *пулингом с функцией максимума*, а размер непересекающихся участков изображения – *размером пула*. Мы можем использовать для пулинга и другие функции, например среднего значения. Такая версия будет называться *пулингом с функцией среднего значения*.

Слой нейронной сети, предшествующий или следующий за сверточным слоем, можно определить так:

Листинг 6.18 ❖ Добавление слоя пулинга с функцией максимума и размером пула (2, 2) в модель Keras

```
model.add(MaxPooling2D(pool_size=(2, 2)))
```

Рис. 6.8 ❖ Уменьшение изображения 8×8 до изображения 4×4 с помощью пулинга с функцией максимума и фильтром 2×2

Вы также можете поэкспериментировать, заменив `MaxPooling2D` на `AveragePooling2D` в листинге 6.4. В случае с распознаванием изображений пулинг часто оказывается незаменимым методом для уменьшения размера выхода сверточных слоев. Несмотря на некоторую потерю информации при сокращении размера изображения, этот метод позволяет сохранить достаточно данных, для того чтобы делать качественные предсказания, и при этом значительно сократить объем необходимых вычислений.

Прежде чем рассмотреть слои пулинга в действии, давайте обсудим еще несколько инструментов, позволяющих повысить верность предсказания ходов в игре го.

6.5. Предсказание вероятностей для ходов в игре го

Поскольку мы познакомились с нейронными сетями лишь в главе 5, нам довелось использовать только одну функцию активации: логистическую сигмоиду. Кроме того, мы использовали среднеквадратическую ошибку в качестве функции потерь. Оба варианта являются приемлемыми отправными точками и имеют свое место в наборе инструментов для глубокого обучения, но они не особенно хорошо подходят для решения стоящей перед нами задачи.

В конце концов, когда мы предсказываем ходы в игре го, мы фактически пытаемся определить *вероятность* того, что тот или иной ход и является следующим ходом. В каждый момент времени состояние доски допускает совершение множества хороших ходов. Целью нашего эксперимента с глубоким обучением является определение следующего хода на основе данных, поданных на вход алгоритма. Однако в конечном итоге суть обучения представлениям и особенно глубокого обучения заключается в том, что эти методы позволяют узнать о структуре игры достаточно для предсказания вероятности хода. Мы стремимся предсказать

распределение вероятностей для всех возможных ходов. Использование сигмоидальных функций активации нам этого не гарантирует. Вместо нее мы применим функцию активации softmax, позволяющую предсказать вероятности в последнем слое.

6.5.1. Использование функции активации softmax в последнем слое

Функция активации softmax является простым обобщением логистической сигмоиды σ. Чтобы вычислить функцию softmax для вектора $x = (x_1, \ldots, x_l)$, мы сначала применяем экспоненциальную функцию к каждому компоненту, т. е. вычисляем e^{x_i}, а затем *нормализуем* каждое из этих значений по сумме всех значений:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{l} e^{x_j}}.$$

По определению компоненты функции softmax неотрицательны, а при их суммировании получается 1. Это означает, что результатом использования данной функции являются значения вероятности. Давайте посмотрим, как это работает.

Листинг 6.19 ❖ Определение функции активации softmax в Python

```
import numpy as np

def softmax(x):
    e_x = np.exp(x)
    e_x_sum = np.sum(e_x)
    return e_x / e_x_sum

x = np.array([100, 100])
print(softmax(x))
```


После определения функции softmax на языке Python мы вычисляем ее для вектора длиной 2, а именно $x = (100, 100)$. Если мы вычислим сигмоидальную функцию для x, то получим результат, близкий к (1, 1). Однако вычисление softmax для этого вектора дает результат (0.5, 0.5). Этого и следовало ожидать – поскольку значения функции softmax в сумме дают 1 и оба элемента вектора являются одинаковыми, функция softmax присваивает обоим компонентам равную вероятность.

Чаще всего softmax применяется в качестве функции активации в последнем слое нейронной сети, что гарантирует предсказание вероятностей на выходе сети.

Листинг 6.20 ❖ Добавление слоя пулинга с функцией максимума и фильтром (2, 2) в модель Keras

```
model.add(Dense(9*9, activation='softmax'))
```

6.5.2. Перекрестная энтропия как функция потерь для задач классификации

В предыдущей главе мы обсуждали среднеквадратическую ошибку и говорили о том, что она является не самой лучшей функцией потерь для нашего случая использования. Далее мы рассмотрим ее недостатки и предложим альтернативный вариант.

Напомним, что проблема предсказания ходов была сформулирована как *задача классификации*, где существует 9×9 возможных классов, только один из которых является верным. Верный класс обозначается 1, а остальные – 0. Предсказания для каждого класса всегда будут иметь значение в диапазоне от 0 до 1. Наша функция потерь должна отражать это сильное предположение относительно предсказаний. Функция СКО, предполагающая возведение разности между предсказанием и меткой в квадрат, не придает особого значения тому факту, что мы ограничены диапазоном от 0 до 1. На самом деле СКО лучше всего подходит для *задач регрессии*, в которых выходом является непрерывный диапазон значений. Например, в случае предсказания роста человека СКО будет штрафовать *большие* разности. В нашем сценарии наибольшая разность между предсказанием и фактическим результатом равна 1.

Другая проблема СКО заключается в том, что эта функция одинаково штрафует все предсказания, т. е. 81 значение. В конце концов, мы стремимся предсказать единственный верный класс, обозначенный 1. Предположим, у нас есть модель, которая присваивает правильному ходу значение вероятности 0,6, а всем остальным – 0, кроме одного, которому присваивается значение 0,4. В этом случае среднеквадратическая ошибка составляет $(1 - 0{,}6)^2 + (0 - 0{,}4)^2 = 2 \times 0{,}4^2$, или около 0,32. Наше предсказание верно, однако мы присваиваем обоим ненулевым предсказаниям одно и то же значение потерь, составляющее около 0,16. Стоит ли придавать такой же вес меньшему значению? Если сравнить этот случай с ситуацией, в которой правильному ходу снова присваивается значение 0,6, а еще двум ходам – по 0,2, то СКО составит $(0{,}4)^2 + 2 \times 0{,}2^2$, т. е. около 0,24, что значительно меньше, чем в предыдущем сценарии. Но что, если значение 0,4 на самом деле является более верным в том смысле, что оно соответствует удачному ходу, который *может оказаться кандидатом на звание следующего хода*? Должна ли наша функция потерь штрафовать и его?

Для решения всех этих проблем мы используем в качестве функции потерь *категорическую перекрестную энтропию*, или просто *перекрестную энтропию*. Для меток $\hat{y}$ и предсказаний модели y эта функция потерь определяется следующим образом:

$$-\sum_i \hat{y}_i \log(y_i).$$

Несмотря на то что эта формула может напоминать сумму множества слагаемых, предполагающую большое количество вычислений, в нашем случае все сводится к единственному слагаемому, для которого $\hat{y}_i$ равно 1. Перекрестная энтропия – это просто $-\log(\hat{y}_i)$ для индекса i, для которого $\hat{y}_i = 1$. Все достаточно просто, но что нам это дает?

- Поскольку перекрестная энтропия штрафует только значение с меткой 1, эта функция напрямую не влияет на распределение остальных значений. В частности, когда мы предсказываем правильный ход с вероятностью 0,6, нет никакой разницы между присвоением одному из других ходов вероятности 0,4 и присвоением двум другим ходам вероятности 0,2. В обоих случаях перекрестная энтропия составляет $-\log(0{,}6) = 0{,}51$.
- Перекрестная энтропия отлично подходит для диапазона значений [0, 1]. Если модель предсказывает вероятность 0 для хода, который на самом деле

был совершен, то такое предсказание никуда не годится. Мы знаем, что $\log(1) = 0$ и что $-\log(x)$ для x в диапазоне от 0 до 1 стремится к бесконечности по мере приближения значения x к 0. Это означает, что значение $-\log(x)$ становится произвольно большим (а не просто растет квадратично, как СКО).

- Кроме того, значение СКО уменьшается быстрее по мере приближения значения x к 1. Это означает, что мы получаем гораздо меньшие значения потери для менее уверенных предсказаний. На рис. 6.9 приведено сравнение графиков СКО и перекрестной энтропии.

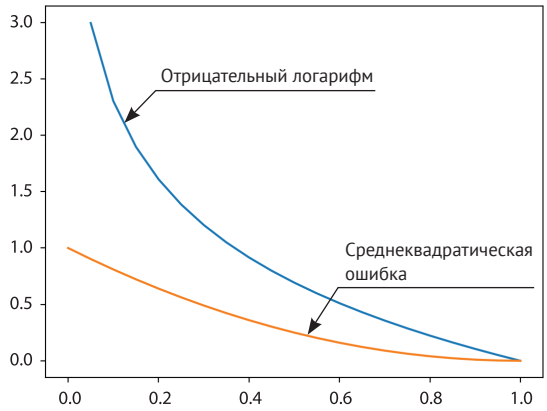


Рис. 6.9 ❖ Сравнение графиков функций потерь СКО и перекрестной энтропии для класса с меткой 1. В случае перекрестной энтропии всем значениям от 0 до 1 соответствует более высокое значение потери

Другим важным моментом, отличающим перекрестную энтропию от СКО, является поведение этой функции во время *обучения* методом стохастического градиентного спуска (СГС). Фактически в случае с СКО обновления градиента становятся все меньше по мере увеличения значения предсказания (т. е. по мере приближения значения y к 1), а значит, обучение, как правило, замедляется. При использовании перекрестной энтропии такого замедления не наблюдается, а обновление параметров пропорционально разнице между предсказанием и истинным значением. Мы не будем углубляться в данную тему, однако следует заметить, что это дает огромные преимущества в нашем случае с предсказанием ходов.

Компиляция модели Keras с категориальной перекрестной энтропией в качестве функции потери также не составляет особого труда.

Листинг 6.21 ❖ Компиляция модели Keras с категориальной перекрестной энтропией

```
model.compile(loss='categorical_crossentropy'...)
```

Теперь, когда вы познакомились с перекрестной энтропией и функцией активации softmax, вы гораздо лучше подготовлены к работе с категориальными метками и предсказанию вероятностей с помощью нейронной сети. В завершение этой главы давайте рассмотрим два метода, позволяющих создавать более глубокие сети, состоящие из большего количества слоев.

6.6. Создание более глубоких сетей с помощью прореживания и блоков линейной ректификации

До сих пор мы создавали нейронные сети, включающие не более двух-четырех слоев. Вам может показаться, что для улучшения результатов достаточно добавить несколько дополнительных слоев. Было бы замечательно, если бы все было так просто, однако на практике нам требуется учесть некоторые нюансы. Несмотря на то что создание все более глубоких нейронных сетей предполагает увеличение количества параметров модели, а значит, и ее способности адаптироваться к подающимся на ее вход данным, это влечет за собой и некоторые проблемы. Одной из главных причин, по которым это может не сработать, является *переобучение*, при котором модель работает все лучше и лучше на *обучающих* данных, но демонстрирует не самые оптимальные результаты на *тестовых* данных. Модель, которая способна практически идеально предсказывать или даже запоминать уже виденное, но не знает, что делать с новыми данными, практически бесполезна. Она должна быть способна к обобщению. Это особенно верно для предсказания следующего хода в такой сложной игре, как го. Независимо от того, сколько времени мы потратим на сбор обучающих данных, в процессе игры возникнут новые для модели ситуации. В любом случае, нам важно определить удачный следующий ход.

6.6.1. Исключение нейронов методом регуляризации

Предотвращение переобучения является очень распространенной проблемой в сфере машинного обучения. Существует большое количество литературы, посвященной *методам регуляризации*, предназначенным для решения этой проблемы. Для глубоких нейронных сетей есть очень простой, но эффективный метод под названием *прореживание*. При применении данного метода к слою сети на каждом этапе обучения мы *случайным образом* выбираем нейроны и задаем для них значение 0, *исключая* их из процесса обучения. Обычно для этого указывается *коэффициент исключения*, задающий процент нейронов, который требуется исключить из конкретного слоя. На рис. 6.10 показан пример слоя исключения, в котором для каждого мини-пакета исключается половина случайно выбранных нейронов (при прямом и обратном проходах сети).

Суть данного метода состоит в том, что исключение выбранных случайным образом нейронов предотвращает излишнюю специализацию отдельных слоев, а значит, и всей сети на конкретных данных. Слои должны быть достаточно гибкими и не полагаться слишком сильно на отдельные нейроны. Так можно предотвратить переучивание нейронной сети.

В библиотеке Keras слой `Dropout` с параметром `rate` определяется следующим образом.

Листинг 6.22 ❖ Импорт и добавление слоя `Dropout` в модель Keras

```
from keras.layers import Dropout

...
model.add(Dropout(rate=0.25))
```

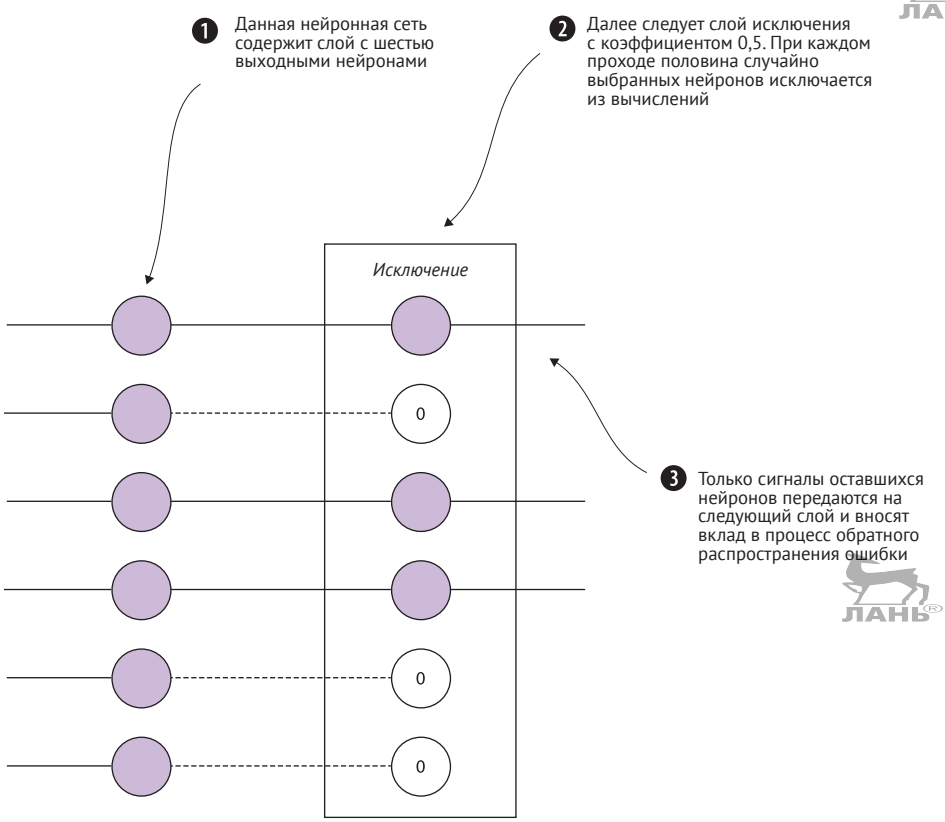


Рис. 6.10 ❖ Слой исключения с коэффициентом 0,5 будет исключать из вычислений половину случайным образом выбранных нейронов для каждого мини-пакета данных, подаваемых на вход сети

Слои исключения можно добавлять в последовательную сеть до или после любого другого слоя. Добавление таких слоев часто бывает необходимым, особенно в случае более глубокой архитектуры.

6.6.2. Функция активации ReLU

Наконец, давайте познакомимся с функцией активации *ReLU* (rectified linear unit, блок линейной ректификации), которая часто оказывается более эффективной для глубоких сетей по сравнению с сигмоидой и другими функциями активации. График данной функции изображен на рис. 6.11.

Функция ReLU игнорирует отрицательные входные значения, приравнивая их к 0, а положительные возвращает в неизменном виде. В случае с ReLU чем сильнее положительный сигнал, тем сильнее активация. Учитывая вышесказанное, блок линейной ректификации довольно сильно напоминает простую модель нейронной сети в мозге, в которой более слабые сигналы игнорируются, а более сильные приводят к возбуждению нейронов. Помимо этой базовой аналогии, мы не будем обсуждать какие-либо теоретические преимущества или недостатки функций ReLU, скажем лишь о том, что их использование часто позволяет получить удов-

летворительные результаты. Чтобы использовать функцию ReLU в Keras, достаточно заменить `sigmoid` на `relu` в аргументе `activation` слоев.

Листинг 6.23 ❖ Добавление функции активации ReLU в слой `Dense`

```
from keras.layers import Dense

...
model.add(Dense(activation='relu'))
```

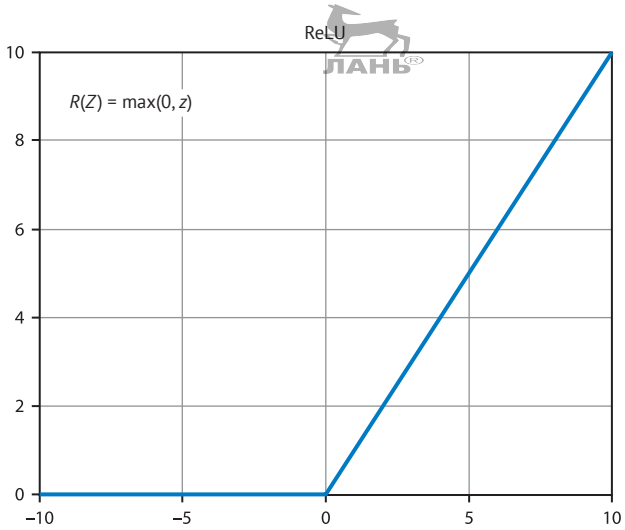


Рис. 6.11 ❖ Функция активации ReLU задает значение 0 для отрицательных входных значений, а положительные оставляет как есть

6.7. Объединяем все вместе и создаем более мощную сеть для предсказания ходов в игре го

В предыдущих разделах были описаны не только сверточные нейронные сети и слои пулинга с функцией максимума, но и такая функция потерь, как перекрестная энтропия, функция активации softmax для последних слоев, метод регуляризации под названием «прореживание», а также функция активации ReLU для повышения производительности сетей. В завершение этой главы мы внедрим новые инструменты в нашу нейронную сеть для предсказания ходов в игре го и посмотрим, что получится.

Для начала давайте вспомним, как загрузить игровые данные, закодированные с помощью простого одноплоскостного кодировщика, и преобразовать их для подачи на вход сверточной сети.

Листинг 6.24 ❖ Загрузка и предварительная обработка данных игры го для сверточных нейронных сетей

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
```

```
from keras.layers import Conv2D, MaxPooling2D

np.random.seed(123)
X = np.load('../generated_games/features-40k.npy')
Y = np.load('../generated_games/labels-40k.npy')

samples = X.shape[0]
size = 9
input_shape = (size, size, 1)

X = X.reshape(samples, size, size, 1)

train_samples = int(0.9 * samples)
X_train, X_test = X[:train_samples], X[train_samples:]
Y_train, Y_test = Y[:train_samples], Y[train_samples:]
```

Теперь давайте доработаем нашу сверточную сеть из листинга 6.3 следующим образом:

- оставьте без изменений базовую архитектуру, начинающуюся с двух сверточных слоев, после которых идет слой пулинга с функцией максимума, а затем – два плотных слоя;
- добавьте три слоя исключения для регуляризации – по одному после каждого сверточного слоя и один – после первого плотного слоя. Для параметра `dropout rate` задайте значение 50 %;
- замените выходной слой функцией активации softmax, а внутренние слои – функциями активации ReLU;
- в качестве функции потерь используйте перекрестную энтропию вместо среднеквадратической ошибки.

Давайте посмотрим, как эта модель выглядит в библиотеке Keras.

Листинг 6.25 ❖ Создание сверточной сети для данных игры го со слоями исключения и функциями ReLU

```
model = Sequential()
model.add(Conv2D(48, kernel_size=(3, 3),
                 activation='relu',
                 padding='same',
                 input_shape=input_shape))
model.add(Dropout(rate=0.5))
model.add(Conv2D(48, (3, 3),
                 padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(rate=0.5))
model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dropout(rate=0.5))
model.add(Dense(size * size, activation='softmax'))
model.summary()

model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])
```

Наконец, для оценки модели можно выполнить следующий код.

Листинг 6.26 ❖ Оценка усовершенствованной сверточной сети

```
model.fit(X_train, Y_train,
          batch_size=64,
          epochs=100,
          verbose=1,
          validation_data=(X_test, Y_test))
score = model.evaluate(X_test, Y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```


Обратите внимание на то, что в данном примере количество эпох увеличено до 100, хотя раньше мы использовали только 15. Выходные данные выглядят примерно так:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 9, 9, 48)          480
_________________________________________________________________
dropout_1 (Dropout)          (None, 9, 9, 48)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 9, 9, 48)          20784
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 4, 4, 48)          0
_________________________________________________________________
dropout_2 (Dropout)          (None, 4, 4, 48)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 768)               0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               393728
_________________________________________________________________
dropout_3 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_2 (Dense)              (None, 81)                41553
=================================================================
Total params: 456,545
Trainable params: 456,545
Non-trainable params: 0
_________________________________________________________________
...
Test loss: 3.81980572336
Test accuracy: 0.0834942084942
```


Верность предсказаний модели возросла до 8 %, что является существенным улучшением по сравнению с исходной версией. Также обратите внимание на запись `Trainable params: 456,545` в выходных данных. Напомним, что исходная модель предусматривала более 600 000 обучаемых параметров. Увеличив в три раза верность предсказаний, мы одновременно уменьшили количество весовых коэффициентов. Это означает, что заслугу в улучшении следует приписать *структуре* нашей новой модели, а не только ее размеру.

С другой стороны, обучение заняло намного больше времени. Во многом это связано с увеличением количества эпох. Данная модель изучает более сложные

концепции, поэтому ей требуется выполнить больше проходов. При наличии достаточного терпения для проведения еще большего количества эпох обучения вы сможете еще немного увеличить верность предсказаний этой модели. В главе 7 описаны продвинутые оптимизаторы, позволяющие ускорить этот процесс.

Теперь давайте подадим на вход модели образец состояния доски и посмотрим, какие ходы она порекомендует:

```
0.000 0.001 0.001 0.002 0.001 0.001 0.000 0.000 0.000
0.001 0.006 0.011 0.023 0.017 0.010 0.005 0.002 0.000
0.001 0.011 0.001 0.052 0.037 0.026 0.001 0.003 0.001
0.002 0.020 0.035 0.045 0.043 0.030 0.014 0.006 0.001
0.003 0.020 0.030 0.031 0.039 0.039 0.018 0.007 0.001
0.001 0.021 0.033 0.048 0.050 0.032 0.017 0.006 0.001
0.001 0.010 0.001 0.039 0.035 0.022 0.001 0.004 0.001
0.000 0.006 0.008 0.017 0.017 0.010 0.007 0.002 0.000
0.000 0.000 0.001 0.001 0.002 0.001 0.001 0.000 0.000
```

Ход с самой высокой оценкой 0,052 соответствует точке A на рис. 6.4 и позволяет черным захватить два белых камня. Возможно, наша модель еще не овладела тактикой игры, но она определенно кое-что узнала о захвате камней! Конечно, результаты все еще не совершенны – модель по-прежнему присваивает высокие оценки многим уже занятым камнями точкам доски.

Мы рекомендуем вам поэкспериментировать с моделью и посмотреть, что произойдет. Вот несколько идей для таких экспериментов:

- что является более эффективным: пулинг с функцией максимума, пулинг с функцией среднего значения или отсутствие пулинга? (Помните, что удаление слоя пулинга ведет к увеличению количества обучаемых параметров в модели. Имейте в виду, что за любое повышение верности предсказаний вы платите дополнительными вычислениями);
- что более эффективно: добавить третий сверточный слой или увеличить количество фильтров на двух уже имеющихся слоях?
- до какой степени можно уменьшить предпоследний слой `Dense` без значительного снижения результативности?
- можно ли улучшить результат, изменив значение коэффициента исключения?
- какой максимальной верности модели можно добиться без использования сверточных слоев? Сравните размер и время обучения этой модели с лучшими результатами, достигаемыми с помощью CHC.

В следующей главе мы применим все изученные методы глубокого обучения для создания бота, способного обучаться на основе *настоящих игровых данных*, а не просто на основе симулированных игр. Кроме того, мы обсудим новые способы кодирования входных данных, позволяющие повысить производительность модели. Сочетая эти приемы, вы сможете создать бота, способного совершать разумные ходы и обыгрывать, по крайней мере, начинающих игроков в го.

6.8. Резюме

- Кодировщики позволяют преобразовать состояния доски для игры в го во входные данные для нейронных сетей, что является важным первым шагом на пути к применению методов глубокого обучения к этой игре.

- Генерация игровых данных с помощью поиска по дереву позволяет получить первый набор данных для подачи на вход нейронной сети.
- Keras – это мощная библиотека для создания архитектуры сетей глубокого обучения.
- Использование сверточных нейронных сетей позволяет извлекать релевантные признаки из пространственной структуры входных данных.
- С помощью слоев пулинга можно уменьшить размеры изображения для снижения вычислительной сложности.
- Использование функции активации softmax в последнем слое сети позволяет получить предсказание вероятности на ее выходе.
- Применение категориальной перекрестной энтропии в качестве функции потерь является более естественным выбором при создании сетей для предсказания ходов в игре го по сравнению со среднеквадратической ошибкой. Среднеквадратическая ошибка более эффективна для предсказания значений в непрерывном диапазоне.
- Слои исключения позволяют предотвратить переобучение глубоких нейронных сетей.
- Использование блоков линейной ректификации вместо сигмоидальных функций активации позволяет значительно повысить производительность.

Глава 7

Глубокое обучение бота на основе данных

В этой главе:

- загрузка и обработка записей реальных партий игры в го;
- описание стандартного формата для хранения партий игры в го;
- обучение модели предсказанию ходов на основе таких данных;
- использование сложных кодировщиков состояния доски для создания мощных ботов;
- проведение собственных экспериментов и оценка их результатов.

В предыдущей главе мы рассмотрели множество основных компонентов для построения приложения глубокого обучения, а также создали несколько нейронных сетей для тестирования новых инструментов. Однако нам по-прежнему не хватает хороших обучающих данных. Качество предсказаний глубокой нейронной сети, обученной с учителем, напрямую зависит от данных, подаваемых на ее вход, а до этого момента в нашем распоряжении были лишь данные, сгенерированные нами самими.

В этой главе вы узнаете о наиболее распространенном формате для хранения записей партий в го, Smart Game Format (SGF). Записи реальных партий игры го в формате SGF можно найти практически на любом популярном го-сервере. Чтобы обучить глубокую нейронную сеть предсказывать ходы, мы загрузим с го-сервера несколько файлов в формате SGF, особым образом закодируем их и обучим нейронную сеть с ними работать. Полученная в итоге сеть будет гораздо мощнее по сравнению с любыми моделями, описанными в предыдущих главах.

На рис. 7.1 продемонстрирован бот, который мы создадим к концу данной главы.

В конце этой главы вы сможете провести собственные эксперименты со сложными нейронными сетями для создания мощного бота. Для начала вам нужно получить доступ к записям реальных партий.

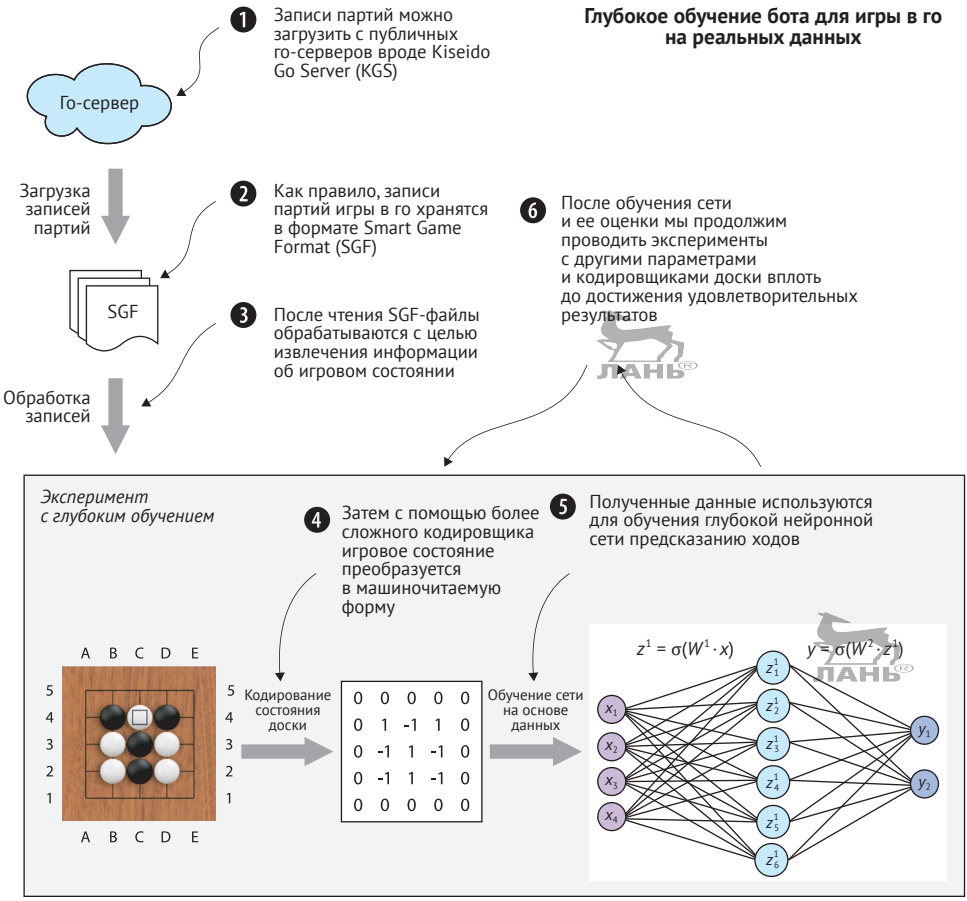


Рис. 7.1 ❖ Создание бота для игры в го с помощью методов глубокого обучения и записей реальных игровых партий. Данные для обучения бота можно найти на публичных серверах для игры в го. В этой главе вы узнаете, как найти эти записи, преобразовать их в обучающий набор и обучить модель Keras имитировать решения игроков

7.1. Импорт записей партий в го

До этого момента мы использовали только данные, сгенерированные нами самими. В предыдущей главе мы обучили глубокую нейронную сеть предсказывать ходы на основе этих сгенерированных данных. Лучшее, на что мы могли рассчитывать, – это то, что сеть сможет точно предсказать эти ходы. При этом сеть играла бы *на том же уровне, что и алгоритм поиска по дереву, сгенерировавший обучающие данные*. В некотором смысле данные, подаваемые на вход сети, определяют верхний предел результативности бота, который на них обучается. Этот бот не способен превзойти игроков, генерирующих данные.

Используя в качестве входных данных записи партий, сыгранных сильными игроками-людьми, мы можем существенно повысить мощность своих ботов. В этой главе мы будем использовать данные с сервера KGS Go Server (ранее из-

вестного как Kiseido Go Server), который является одной из самых популярных в мире платформ для игры в го. Прежде чем приступить к загрузке и обработке данных с сервера KGS, давайте обсудим формат, в котором хранятся эти данные.

7.1.1. Формат файла SGF

Формат Smart Game Format (SGF), первоначально называемый Smart Go Format, используется с конца 1980-х годов. Текущая четвертая версия (`FF[4]`) была выпущена в конце 1990-х годов. SGF представляет собой простой текстовый формат, который можно использовать для сохранения партий в го, их вариаций (например, партий, дополненных комментариями профессиональных игроков) и других настольных игр. Оставшаяся часть главы предполагает работу с файлами SGF, которые содержат только данные о партии в го без каких-либо дополнений. В этом разделе мы коснемся лишь основ данного формата. Узнать о нем более подробно вы можете на сайте **senseis.xmp.net/?SmartGameFormat**.

По сути, файл SGF содержит метаданные о партии и совершенных ходах. Метаданные обозначаются двумя заглавными буквами, кодирующими свойство, а соответствующее значение указывается в квадратных скобках. Например, при использовании формата SGF партия в го, сыгранная на доске размером (`SZ`) 9×9, будет закодирована в виде `SZ[9]`. Ходы кодируются следующим образом: размещение белого камня в точке на пересечении третьей строки и третьего столбца сетки доски кодируется в виде `W[cc]`, тогда как размещение черного камня в точке на пересечении седьмой строки и третьего столбца кодируется в виде `B[gc]`. Буквы `B` и `W` обозначают цвет камня, а строки и столбцы индексируются в алфавитном порядке. Пропуск хода черными и белыми кодируется в виде `B[]` и `W[]` соответственно.

Следующий файл SGF взят из примера, описанного в конце главы 2. Он содержит данные о партии в го (в формате SGF игра го обозначается цифрой 1 или `GM[1]`) в текущей версии формата SGF (`FF[4]`), сыгранной на доске 9×9 с нулевым гандикапом (`HA[0]`) и 6,5 балла коми, присужденного белым в качестве компенсации за право черных на совершение первого хода (`KM[6.5]`) . В этой игре используется японский набор правил (`RU[Japanese]`), а результатом (`RE`) является победа белых со счетом 9,5 очка (`RE[W+9.5]`):

```
(;FF[4] GM[1] SZ[9] HA[0] KM[6.5] RU[Japanese] RE[W+9.5]
;B[gc];W[cc];B[cg];W[gg];B[hf];W[gf];B[hg];W[hh];B[ge];W[df];B[dg]
;W[eh];B[cf];W[be];B[eg];W[fh];B[de];W[ec];B[fb];W[eb];B[ea];W[da]
;B[fa];W[cb];B[bf];W[fc];B[gb];W[fe];B[gd];W[ig];B[bd];W[he];B[ff]
;W[fg];B[ef];W[hd];B[fd];W[bi];B[bh];W[bc];B[cd];W[dc];B[ac];W[ab]
;B[ad];W[hc];B[ci];W[ed];B[ee];W[dh];B[ch];W[di];B[hb];W[ib];B[ha]
;W[ic];B[dd];W[ia];B[];
 TW[aa][ba][bb][ca][db][ei][fi][gh][gi][hf][hg][hi][id][ie][if]
  [ih][ii]
 TB[ae][af][ag][ah][ai][be][bg][bi][ce][df][fe][ga]
 W[])
```

Файл SGF организован в виде списка *узлов*, разделенных точками с запятой. Первый узел содержит метаданные об игре: размер доски, набор правил, результат и другую справочную информацию. Каждый следующий узел соответствует ходу. Пробелы не имеют никакого значения. Весь пример можно преобразовать

в одну строку, которая по-прежнему будет являться действительным файлом SGF. В конце также перечислены точки доски, находящиеся на территории белых `TW` и черных `TB`. Обратите внимание на то, что эти индикаторы захваченной территории относятся к тому же узлу, что и последний ход белых (`W[]` обозначает пропуск хода). Их можно рассматривать в качестве своего рода комментария относительно конкретной игровой позиции.

Этот пример иллюстрирует некоторые основные свойства файлов в формате SGF и демонстрирует все, что вам потребуется для воспроизведения игровых партий с целью генерации обучающих данных. Формат SGF поддерживает множество других функций, но они в основном используются для добавления комментариев и аннотаций к записям партий, поэтому не понадобятся нам в ходе изучения этой книги.

7.1.2. Загрузка и воспроизведение партий в го с сервера KGS

На странице **u-go.net/gamerecords** вы увидите таблицу с записями партий, доступными для скачивания в различных форматах (zip, tar.gz). Эти игровые данные собираются с сервера KGS Go Server, начиная с 2001 года, и содержат только те партии, в которых один из игроков имел 7 дан или выше, или оба игрока имели 6 дан. Как вы помните из главы 2, ранги от 1 до 9 дан присуждаются сильным игрокам. Также имейте в виду, что все эти партии были сыграны на доске 19×19, в то время как в главе 6 мы использовали данные, сгенерированные для доски 9×9, соответствующие менее сложной игровой ситуации.

Далее в главе мы используем этот набор данных, чтобы создать более мощного бота для предсказания ходов в игре го с помощью методов глубокого обучения. Мы загрузим эти данные автоматически, запросив HTML со ссылками на отдельные файлы, после чего распакуем и обработаем содержащиеся в них записи партий в формате SGF.

Прежде чем подавать эти данные на вход модели глубокого обучения, создайте в основном модуле *dlgo* новый подмодуль *data* с пустым файлом *__init__.py*. Этот подмодуль будет содержать все, что имеет отношение к обработке данных игры го.

Для загрузки игровых данных создайте класс `KGSIndex` в новом файле *index_processor.py* в подмодуле *data*. Поскольку этот шаг носит исключительно технический характер и не расширяет ваших знаний ни об игре го, ни о машинном обучении, мы не будем подробно останавливаться на его реализации. Если вас интересуют подробности, вы можете найти код в репозитории GitHub. Реализация класса `KGSIndex` содержит лишь один метод, который пригодится нам в дальнейшем: `download_files`. Этот метод будет создавать локальное зеркало страницы **u-go.net/gamerecords**, находить все релевантные ссылки и загружать соответствующие файлы *tar.gz* в отдельную папку *data*. Вот как он вызывается.

Листинг 7.1 ❖ Создание индекса zip-файлов, содержащих игровые данные с сервера KGS

```
from dlgo.data.index_processor import KGSIndex

index = KGSIndex()
index.download_files()
```

После выполнения этого кода в командной строке должно отобразиться следующее:

```
>>> Downloading index page
KGS-2017_12-19-1488-.tar.gz 1488
KGS-2017_11-19-945-.tar.gz 945

...

>>> Downloading data/KGS-2017_12-19-1488-.tar.gz
>>> Downloading data/KGS-2017_11-19-945-.tar.gz

...
```

Теперь, когда эти данные хранятся локально, давайте приступим к их обработке и использованию в нейронной сети.

7.2. Подготовка игровых данных для глубокого обучения

В главе 6 мы рассмотрели простой *кодировщик* для данных игры го, который был представлен в терминах классов `Board` и `GameState`, описанных в главе 3. При работе с файлами в формате SGF сначала необходимо извлечь их содержимое (т. е. *распаковать*) и воспроизвести партию на их основе для получения необходимой информации об игровых состояниях.

7.2.1. Воспроизведение партии в го на основе ее записи в формате SGF

Чтение данных об игровом состоянии из файла SGF предполагает понимание и реализацию спецификаций данного формата. Это не составляет особого труда и требует лишь применения определенного набора правил к строке текста, однако этот не самый интересный этап процесса создания игрового бота занимает довольно много времени. По данной причине мы добавим в модуль *dlgo* еще один подмодуль под названием *gosgf*, который будет отвечать за обработку файлов SGF. Мы не будем детально обсуждать этот подмодуль в данной главе. Подробную информацию о том, как считывать и интерпретировать данные в формате SGF средствами языка Python, можно найти в нашем репозитории GitHub.

Модуль *gosgf* представляет собой адаптацию библиотеки Python под названием Gomill, доступной по адресу: **mjw.woodcraft.me.uk/gomill**.

Для обработки всего необходимого нам будет достаточно одной сущности из модуля *gosgf*: `Sgf_game`. Давайте посмотрим, как с ее помощью можно загрузить образец партии в формате SGF, считать информацию о ходах и применить их к объекту `GameState`. На рис. 7.2 представлено начало партии в го с указанием команд SGF.

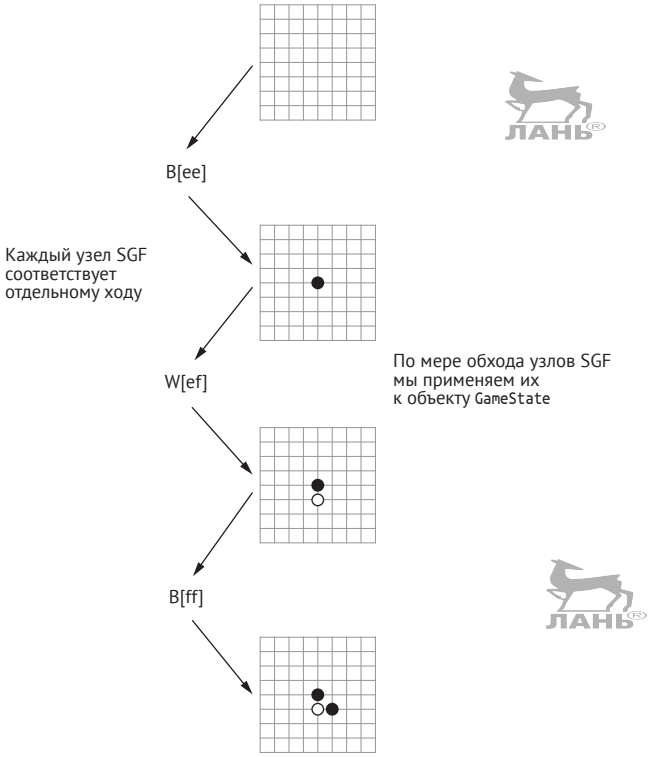


Рис. 7.2 ❖ Воспроизведение партии в го из файла SGF. Исходный файл SGF содержит ходы, закодированные в строки вида `B[ee]`. Класс `Sgf_game` декодирует эти строки и возвращает их в виде кортежей Python. Затем эти ходы можно применить к объекту `GameState`, чтобы воспроизвести партию (см. следующий листинг)

Листинг 7.2 ❖ Воспроизведение ходов из файла SGF с помощью фреймворка для игры в го

```
from dlgo.gosgf import Sgf_game
from dlgo.goboard_fast import GameState, Move
from dlgo.gotypes import Point
from dlgo.utils import print_board

sgf_content = "(;GM[1]FF[4]SZ[9];B[ee];W[ef];B[ff]" + \
              ";W[df];B[fe];W[fc];B[ec];W[gd];B[fb])"

sgf_game = Sgf_game.from_string(sgf_content)

game_state = GameState.new_game(19)

for item in sgf_game.main_sequence_iter():

    color, move_tuple = item.get_move()
    if color is not None and move_tuple is not None:
        row, col = move_tuple
        point = Point(row + 1, col + 1)
        move = Move.play(point)
```

Импорт класса `Sgf_game` из нового модуля *gosgf*.

Определение образца строки SGF. Соответствующее содержимое будет в дальнейшем получено из загруженных данных.

Для создания `Sgf_game` используется метод `from_string`.

Итеративная обработка главной последовательности игры с игнорированием вариаций и комментариев.

Элементами этой основной последовательности являются пары (`color`, `move`), при этом «move» представляет собой пару координат точки доски.

```
game_state = game_state.apply_move(move)
print_board(game_state.board)
```

Затем считанный ход применяется к текущему игровому состоянию.

По сути, после того как у вас появится действительная строка SGF, вы сможете создать из нее игру, которую можно будет итеративно обрабатывать любым способом. Листинг 7.2 является ключевым для данной главы и позволяет получить представление о подготовке игровых данных к использованию в процессе глубокого обучения:

1) загрузите и распакуйте сжатые файлы игры го;
2) итеративно обработайте каждый файл SGF, считайте эти файлы как строки Python и создайте на их основе `Sgf_game`;
3) считайте главную последовательность игры для каждой строки SGF, при этом позаботьтесь о таких важных деталях, как размещение камней гандикапа, а затем передайте полученные данные о ходах объекту `GameState`;
4) для каждого хода закодируйте текущее состояние доски в виде признака с помощью кодировщика `Encoder`, а информацию о ходе сохраните в качестве метки, прежде чем применить ход к доске. Так вы сможете создавать данные для глубокого обучения на лету;
5) сохраните полученные признаки и метки в подходящем формате для дальнейшей подачи на вход глубокой нейронной сети.

В следующих разделах мы подробно рассмотрим каждый из этих пяти этапов. После обработки данных вы сможете вернуться к своему приложению для предсказания ходов и посмотреть, как эти данные влияют на верность его предсказаний.

7.2.2. Создание обработчика данных игры го

В этом разделе мы создадим *обработчик данных* игры го, способный преобразовывать необработанные данные в формате SGF в признаки и метки для подачи на вход алгоритма машинного обучения. Его реализация является относительно громоздкой, поэтому мы разделим ее на несколько частей. По окончании у вас будет все, что нужно для глубокого обучения модели на реальных данных.

Для начала создайте новый файл с именем *processor.py* в новом подмодуле *data*. Как и прежде, вы можете взять копию файла *processor.py* из репозитория GitHub. Давайте импортируем в файл *processor.py* несколько основных библиотек Python, с которыми нам предстоит работать. Помимо NumPy для данных, нам понадобится несколько пакетов для обработки файлов.

Листинг 7.3 ❖ Библиотеки Python для обработки данных и файлов

```
import os.path
import tarfile
import gzip
import glob
import shutil

import numpy as np
from keras.utils import to_categorical
```

Из модуля *dlgo* нам потребуется импортировать многие из основных абстракций, созданных ранее.

Листинг 7.4 ❖ Импорт абстракций для обработки данных из модуля `dlgo`

```
from dlgo.gosgf import Sgf_game
from dlgo.goboard_fast import Board, GameState, Move
from dlgo.gotypes import Player, Point
from dlgo.encoders.base import get_encoder_by_name

from dlgo.data.index_processor import KGSIndex
from dlgo.data.sampling import Sampler
```

Для создания набора обучающих и тестовых данных будет использоваться сэмплер `Sampler`.

Мы еще не обсуждали последние два импортируемых компонента (`Sampler` и `DataGenerator`). Мы поговорим о них в процессе создания обработчика данных игры го. Вернемся к файлу *processor.py*. Для инициализации обработчика `GoDataProcessor` мы предоставляем кодировщик `Encoder` в виде строки и директорию `data_directory` для хранения данных SGF.

Листинг 7.5 ❖ Инициализация обработчика данных игры го путем указания кодировщика и локальной директории с данными

```
class GoDataProcessor:
    def __init__(self, encoder='oneplane', data_directory='data'):
        self.encoder = get_encoder_by_name(encoder, 19)
        self.data_dir = data_directory
```

Далее мы реализуем главный метод обработки данных `load_go_data`. В этом методе можно указать количество игр, которые требуется обработать, а также тип загружаемых данных (обучающие или тестовые). Метод `load_go_data` загрузит записи партий с го-сервера KGS, отберет указанное количество образцов игр, обработает их, создав признаки и метки, а затем сохранит результат локально в виде массивов NumPy.

Листинг 7.6 ❖ Метод `load_go_data` загружает, обрабатывает и сохраняет данные

```
    def load_go_data(self, data_type='train',
                     num_samples=1000):
        index = KGSIndex(data_directory=self.data_dir)
        index.download_files()

        sampler = Sampler(data_dir=self.data_dir)
        data = sampler.draw_data(data_type, num_samples)

        zip_names = set()
        indices_by_zip_name = {}
        for filename, index in data:
            zip_names.add(filename)
            if filename not in indices_by_zip_name:
                indices_by_zip_name[filename] = []
            indices_by_zip_name[filename].append(index)
        for zip_name in zip_names:
            base_name = zip_name.replace('.tar.gz', '')
            data_file_name = base_name + data_type
            if not os.path.isfile(self.data_dir + '/' + data_file_name):
                self.process_zip(zip_name, data_file_name,
                                 indices_by_zip_name[zip_name])
```

В качестве типа данных `data_type` можно указать `train` (обучающие) или `test` (тестовые).

Значение `num_samples` задает количество партий, данные которых требуется загрузить.

Загрузка игр с сервера KGS в локальную директорию с данными. Если данные уже доступны, они не будут загружены снова.

Экземпляр сэмплера `Sampler` отбирает указанное количество партий для формирования набора данных заданного типа.

Составление списка всех имен zip-файлов, содержащихся в данных.

Группировка всех индексов SGF-файлов по имени zip-файла.

Обработка отдельных zip-файлов.

```
    features_and_labels = self.consolidate_games(data_type, data)
    return features_and_labels
```

Объединение и возврат признаков и меток из каждого zip-файла.

Обратите внимание на то, что после загрузки данных мы разделили их с помощью экземпляра сэмплера `Sampler`, который случайным образом отбирает указанное количество партий так, чтобы *наборы обучающих и тестовых данных не перекрывались*. Сэмплер разделяет данные на уровне файлов, просто относя партии, сыгранные до 2014 года, к тестовым данным, а более новые партии – к обучающим. Благодаря этому мы можем быть уверены в том, что никакая информация из тестовых данных не будет (даже частично) включена в обучающие данные, что могло бы привести к переобучению наших моделей.

Разделение данных на обучающие и тестовые

Мы разделяем данные на обучающие и тестовые с целью получения надежных показателей производительности. Для обучения модели используются обучающие данные, а для оценки – тестовые. Это позволяет понять, как модель адаптируется к *не виденным ранее ситуациям*, насколько хорошо она применяет полученные на этапе обучения знания к реальному миру. Чтобы результатам, полученным от модели, можно было доверять, необходимо правильно собрать и разделить данные. Можно было бы просто загрузить все имеющиеся у нас данные, перетасовать их и случайным образом разделить на обучающий и тестовый наборы. Однако эффективность этого наивного подхода зависит от стоящей перед нами задачи. В случае с игрой го ходы, совершенные в рамках одной партии, зависят друг от друга. Обучение модели на серии ходов, которые также входят в набор тестовых данных, может создать иллюзию ее эффективности. Однако на практике ваш бот может оказаться гораздо менее мощным. Не жалейте времени на анализ данных и постарайтесь обеспечить их корректное разделение.

После загрузки и сэмплирования данных метод `load_go_data` использует для их обработки вспомогательные функции: `process_zip` для считывания отдельных zip-файлов и `consolidate_games` для группировки результатов из каждого zip-файла в один набор признаков и меток. Функция `process_zip` выполняет следующие действия:

1) разархивирует текущий файл с помощью `unzip_data`;
2) инициализирует экземпляр кодировщика `Encoder` для кодирования записей SGF;
3) инициализирует массивы NumPy правильной формы, содержащие признаки и метки;
4) итеративно обрабатывает список партий;
5) для каждой партии сначала применяет все камни гандикапа;
6) затем считывает каждый ход, обнаруженный в записи SGF;
7) кодирует каждый следующий ход как метку `label`;
8) кодирует текущее состояние доски как признак `feature`;
9) применяет следующий ход к доске;
10) сохраняет небольшие фрагменты признаков и меток в локальной файловой системе.

Вот как первые девять из перечисленных выше шагов реализуются в process_zip. Вспомогательный метод unzip_data был опущен для краткости, но его можно найти в нашем репозитории GitHub. На рис. 7.3 изображен процесс преобразования сжатых файлов SGF в закодированное игровое состояние.

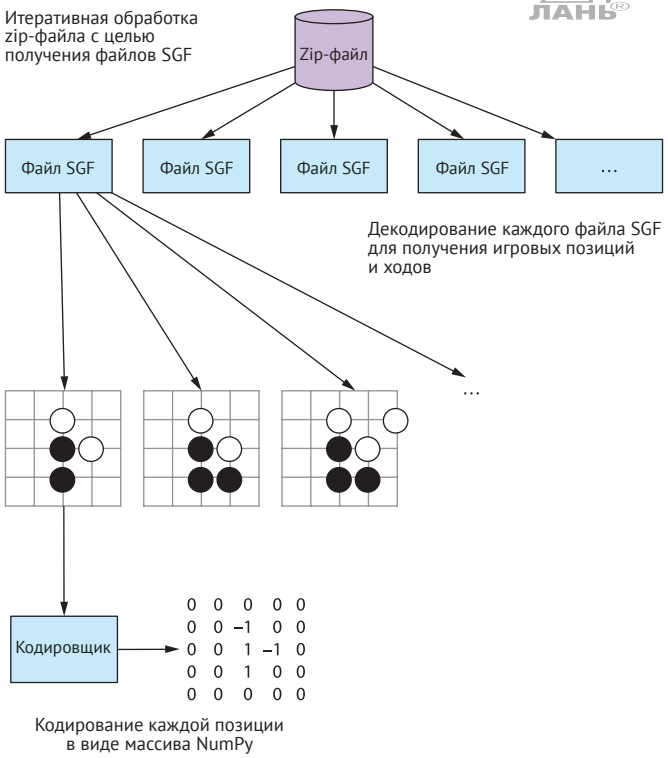


Рис. 7.3 ❖ Функция process_zip. Мы итеративно обрабатываем zip-файл, содержащий множество файлов SGF. Каждый файл SGF содержит последовательность ходов, которые используются для реконструкции объектов GameState. Затем используем объект Encoder для преобразования каждого игрового состояния в массив NumPy

Теперь мы можем определить функцию process_zip.

Листинг 7.7 ❖ Преобразование записей партий игры го, хранящихся в zip-файлах, в закодированные признаки и метки

```
def process_zip(self, zip_file_name, data_file_name, game_list):
    tar_file = self.unzip_data(zip_file_name)
    zip_file = tarfile.open(self.data_dir + '/' + tar_file)
    name_list = zip_file.getnames()
    total_examples = self.num_total_examples(zip_file, game_list,
                                             name_list)

    shape = self.encoder.shape()
    feature_shape = np.insert(shape, 0, np.asarray([total_examples]))
    features = np.zeros(feature_shape)
    labels = np.zeros((total_examples,))
```

Определение общего количества ходов во всех партиях, содержащихся в данном zip-файле.

Определение формы признаков и меток на основе используемого кодировщика.

```
counter = 0
for index in game_list:
    name = name_list[index + 1]
    if not name.endswith('.sgf'):
        raise ValueError(name + ' is not a valid sgf')
    sgf_content = zip_file.extractfile(name).read()
    sgf = Sgf_game.from_string(sgf_content)

    game_state, first_move_done = self.get_handicap(sgf)

    for item in sgf.main_sequence_iter():
        color, move_tuple = item.get_move()
        point = None
        if color is not None:
            if move_tuple is not None:
                row, col = move_tuple
                point = Point(row + 1, col + 1)
                move = Move.play(point)
            else:
                move = Move.pass_turn()
            if first_move_done and point is not None:
                features[counter] = self.encoder.encode(game_state)
                labels[counter] = self.encoder.encode_point(point)
                counter += 1
            game_state = game_state.apply_move(move)
            first_move_done = True
```

Чтение содержимого файла SGF в виде строки после распаковки zip-файла.

Определение начального игрового состояния путем применения всех камней гандикапа.

Итеративная обработка всех ходов в файле SGF.

Считывание координат камня, который предстоит поместить на доску...

...или пропуск хода при отсутствии такого камня.

Кодирование текущего игрового состояния в виде признаков...

...а следующего хода – в виде метки для этих признаков.

Применение хода к доске и повторение цикла для следующего хода.

Цикл for имеет много общего с процессом, описанным в листинге 7.2, поэтому этот код должен показаться вам знакомым. Функция process_zip использует два вспомогательных метода, которые мы реализуем далее. Первый из них, num_total_examples, предварительно вычисляет количество доступных ходов на zip-файл, что позволяет определить размер массивов признаков и меток.

Листинг 7.8 ❖ Вычисление общего количества ходов, доступных в текущем zip-файле

```
def num_total_examples(self, zip_file, game_list, name_list):
    total_examples = 0
    for index in game_list:
        name = name_list[index + 1]
        if name.endswith('.sgf'):
            sgf_content = zip_file.extractfile(name).read()
            sgf = Sgf_game.from_string(sgf_content)
            game_state, first_move_done = self.get_handicap(sgf)

            num_moves = 0
            for item in sgf.main_sequence_iter():
                color, move = item.get_move()
                if color is not None:
                    if first_move_done:
                        num_moves += 1
                    first_move_done = True
            total_examples = total_examples + num_moves
```

```
            else:
                raise ValueError(name + ' is not a valid sgf')
        return total_examples
```

Второй вспомогательный метод используется для определения количества камней гандикапа в текущей партии и применения соответствующих ходов к пустой доске.

Листинг 7.9 ❖ Определение количества камней гандикапа и их применение к пустой доске для игры в го

```
@staticmethod
def get_handicap(sgf):
    go_board = Board(19, 19)
    first_move_done = False
    move = None
    game_state = GameState.new_game(19)
    if sgf.get_handicap() is not None and sgf.get_handicap() != 0:
        for setup in sgf.get_root().get_setup_stones():
            for move in setup:
                row, col = move
                go_board.place_stone(Player.black,
                                     Point(row + 1, col + 1))
        first_move_done = True
        game_state = GameState(go_board, Player.white, None, move)
    return game_state, first_move_done
```

Для завершения реализации функции `process_zip` мы сохраняем фрагменты признаков и меток в отдельных файлах.

Листинг 7.10 ❖ Локальное хранение фрагментов признаков и меток

```
    feature_file_base = self.data_dir + '/' + data_file_name +
'_features_%d'
    label_file_base = self.data_dir + '/' + data_file_name + '_labels_%d'

    chunk = 0  # Due to files with large content, split up after chunksize
    chunksize = 1024
    while features.shape[0] >= chunksize:
        feature_file = feature_file_base % chunk
        label_file = label_file_base % chunk
        chunk += 1
        current_features, features = features[:chunksize], features[chunksize:]
        current_labels, labels = labels[:chunksize], labels[chunksize:]
        np.save(feature_file, current_features)
        np.save(label_file, current_labels)
```

Обработка признаков и меток в виде фрагментов размером 1024.

Текущий фрагмент отсекается от признаков и меток...

...и сохраняется в отдельном файле.

Мы сохраняем признаки и метки небольшими фрагментами из-за того, что размер массивов NumPy может быстро увеличиваться, а хранение данных в небольших файлах обеспечивает большую гибкость в долгосрочной перспективе. Например, эти фрагменты можно объединять или загружать в память по мере необходимости. Мы будем делать и то, и другое. Второй из этих подходов – динамическая загрузка пакетов данных в процессе работы – является немного более

сложным, в то время как консолидация данных затруднений не вызывает. Отметим, что в нашей реализации существует вероятность потери последней части фрагмента в цикле `while`, но это несущественно, поскольку в нашем распоряжении данных более чем достаточно.

Вернемся к определению обработчика `GoDataProcessor` в файле *processor.py* и просто объединим все массивы в один методом *конкатенации*.

Листинг 7.11 ❖ Объединение массивов признаков и меток NumPy в один набор

```
def consolidate_games(self, data_type, samples):
    files_needed = set(file_name for file_name, index in samples)
    file_names = []
    for zip_file_name in files_needed:
        file_name = zip_file_name.replace('.tar.gz', '') + data_type
        file_names.append(file_name)

    feature_list = []
    label_list = []
    for file_name in file_names:
        file_prefix = file_name.replace('.tar.gz', '')
        base = self.data_dir + '/' + file_prefix + '_features_*.npy'
        for feature_file in glob.glob(base):
            label_file = feature_file.replace('features', 'labels')
            x = np.load(feature_file)
            y = np.load(label_file)
            x = x.astype('float32')
            y = to_categorical(y.astype(int), 19 * 19)
            feature_list.append(x)
            label_list.append(y)
    features = np.concatenate(feature_list, axis=0)
    labels = np.concatenate(label_list, axis=0)
    np.save('{}/features_{}.npy'.format(self.data_dir, data_type), features)
    np.save('{}/labels_{}.npy'.format(self.data_dir, data_type), labels)

    return features, labels
```

Для проверки этой реализации загрузим признаки и метки для 100 партий следующим образом.

Листинг 7.12 ❖ Загрузка обучающих данных из 100 записей партий

```
from dlgo.data.processor import GoDataProcessor

processor = GoDataProcessor()
features, labels = processor.load_go_data('train', 100)
```

Эти признаки и метки были закодированы с помощью кодировщика `oneplane` из главы 6, а значит, они имеют ту же структуру. Вы можете попробовать обучить любую из сетей, созданных в главе 6, на основе только что подготовленных данных. При этом не рассчитывайте на особую эффективность. Несмотря на то что записи реальных партий значительно превосходят игры, сгенерированные в главе 6, теперь мы работаем с игровой доской 19×19, а не 9×9, что гораздо сложнее.

Процедура загрузки в память большого количества небольших файлов может привести к исключениям нехватки памяти. Мы постараемся решить эту пробле-

му в следующем разделе, используя *генератор данных* для предоставления только того мини-пакета, который необходим для обучения модели.

7.2.3. Создание генератора данных игры go для обеспечения их эффективной загрузки

Индекс KGS, загруженный с сайта **u-go.net/gamerecords**, содержит более 170 000 партий и миллионы ходов, которые можно использовать для обучения сети. Процесс добавления всех этих точек данных в единственную пару массивов NumPy будет усложняться по мере загрузки дополнительных записей партий. В долгосрочной перспективе такая попытка консолидации игровых данных обречена на провал.

Мы предлагаем разумную альтернативу функции `consolidate_games` обработчика `GoDataProcessor`. В конце концов, для обучения нейронной сети достаточно *по одному* подавать на ее вход мини-пакеты признаков и меток. Нет необходимости постоянно хранить данные в памяти. Поэтому далее мы создадим *генератор* данных игры go. Если вы уже сталкивались с концепцией генераторов Python, то этот процесс покажется вам знакомым. Если нет, то думайте о генераторе как о функции, которая по мере необходимости предоставляет вам только нужные пакеты.

Для начала инициализируем `DataGenerator`. Поместите следующий код в файл *generator.py*, находящийся в модуле *data*. Для инициализации генератора мы предоставляем локальную директорию `data_directory` и образцы, предусмотренные нашим сэмплером `Sampler` в `GoDataProcessor`.

Листинг 7.13 ❖ Сигнатура генератора игровых данных

```
import glob
import numpy as np
from keras.utils import to_categorical

class DataGenerator:
    def __init__(self, data_directory, samples):
        self.data_directory = data_directory
        self.samples = samples
        self.files = set(file_name for file_name, index in samples)
        self.num_samples = None

    def get_num_samples(self, batch_size=128, num_classes=19 * 19):
        if self.num_samples is not None:
            return self.num_samples
        else:
            self.num_samples = 0
            for X, y in self._generate(batch_size=batch_size,
                                       num_classes=num_classes):
                self.num_samples += X.shape[0]
            return self.num_samples
```

Генератор имеет доступ к созданной ранее выборке файлов.

В зависимости от приложения у нас может возникнуть потребность в выяснении количества имеющихся образцов.

Далее мы реализуем частный метод `_generate`, который создает и возвращает пакеты данных. Этот метод имеет ту же общую логику, что и `consolidate_games`, за одним важным исключением. Ранее мы создавали большой массив NumPy как для признаков, так и для меток, а теперь просто возвращаем (`yield`) следующий пакет данных.

Листинг 7.14 ❖ Частный метод для генерации и возврата следующего пакета игровых данных

```
def _generate(self, batch_size, num_classes):
    for zip_file_name in self.files:
        file_name = zip_file_name.replace('.tar.gz', '') + 'train'
        base = self.data_directory + '/' + file_name + '_features_*.npy'
        for feature_file in glob.glob(base):
            label_file = feature_file.replace('features', 'labels')
            x = np.load(feature_file)
            y = np.load(label_file)
            x = x.astype('float32')
            y = to_categorical(y.astype(int), num_classes)
            while x.shape[0] >= batch_size:
                x_batch, x = x[:batch_size], x[batch_size:]
                y_batch, y = y[:batch_size], y[batch_size:]
                yield x_batch, y_batch
```

Возврат пакетов данных по мере необходимости.

Теперь в нашем генераторе не хватает только метода для его возврата. При наличии генератора мы можем явно вызвать метод `next()`, чтобы сгенерировать пакеты данных. Это делается следующим образом.

Листинг 7.15 ❖ Вызов метода `generate` с целью получения генератора данных для обучения модели

```
def generate(self, batch_size=128, num_classes=19 * 19):
    while True:
        for item in self._generate(batch_size, num_classes):
            yield item
```

Прежде чем обсуждать способ применения генератора в процессе обучения нейронной сети, необходимо разобраться с тем, как эта концепция вписывается в структуру обработчика `GoDataProcessor`.

7.2.4. Параллельная обработка игровых данных и генераторы

Вы, вероятно, заметили, что на загрузку всего лишь 100 записей партий в листинге 7.3 уходит больше времени, чем можно было бы ожидать. Разумеется, сначала данные требуется загрузить, однако проблема в том, что их обработка происходит относительно медленно. Напомним, что zip-файлы обрабатываются *последовательно*. После завершения работы с одним файлом мы переходим к следующему. Но если внимательно присмотреться, то обработку данных игры го в том виде, в котором мы ее представили, можно назвать *чрезвычайно параллельной*. Обеспечить параллельную обработку zip-файлов совсем не трудно. Для этого достаточно распределить рабочую нагрузку между всеми процессорами компьютера, например с помощью многопроцессорной библиотеки Python.

Параллельную реализацию обработчика `GoDataProcessor` можно найти в файле *parallel_processor.py*, который содержится в модуле *data* репозитория GitHub. Если вы хотите разобраться в тонкостях ее работы, мы рекомендуем вам ознакомиться с этим файлом. Здесь мы опустили эти детали, чтобы не усложнять код.

Помимо ускорения, параллельная версия `GoDataProcessor` обеспечивает еще одно преимущество, связанное с возможностью ее использования совместно с `DataGenerator` для возврата генератора вместо данных.

Листинг 7.16 ❖ Параллельная версия `load_go_data` может опционально возвращать генератор

```
def load_go_data(self, data_type='train', num_samples=1000,
                 use_generator=False):
    index = KGSIndex(data_directory=self.data_dir)
    index.download_files()

    sampler = Sampler(data_dir=self.data_dir)
    data = sampler.draw_data(data_type, num_samples)

    self.map_to_workers(data_type, data)
    if use_generator:
        generator = DataGenerator(self.data_dir, data)
        return generator
    else:
        features_and_labels = self.consolidate_games(data_type, data)
        return features_and_labels
```

Распределение рабочей нагрузки между процессорами.

Возвращает генератор игровых данных или...

...консолидированные данные, как и прежде.

Обе версии обработчика `GoDataProcessor` имеют один и тот же интерфейс, за исключением того, что в параллельном расширении используется флаг `use_generator`. Теперь обработчик `GoDataProcessor` из файла *parallel_processor.py*, содержащегося в подмодуле *data* модуля *dlgo*, позволяет задействовать генератор для предоставления игровых данных.

Листинг 7.17 ❖ Загрузка обучающих данных из 100 записей партий

```
from dlgo.data.parallel_processor import GoDataProcessor

processor = GoDataProcessor()
generator = processor.load_go_data('train', 100, use_generator=True)

print(generator.get_num_samples())
generator = generator.generate(batch_size=10)
X, y = generator.next()
```

Поначалу загрузка данных все равно занимает довольно много времени, однако этот процесс должен ускориться пропорционально количеству процессоров, установленных в компьютере. После создания генератора вызов метода `next()` мгновенно возвращает пакеты данных. Это позволяет нам избежать проблем, связанных с превышением допустимого объема памяти.

7.3. Обучение глубокой сети на основе партий, сыгранных человеком

Теперь, когда мы получили качественные игровые данные и подготовили их для подачи на вход модели, пришло время создать для них глубокую нейронную сеть. В пакете *dlgo* репозитория GitHub находится модуль *networks* с образцами архитектур нейронных сетей, которые можно использовать в качестве основы для создания мощных моделей для предсказания ходов. Например, этот модуль предусматривает три сверточные нейронные сети разной степени сложности: *small.py*, *medium.py* и *large.py*. Каждый из этих файлов содержит функцию `layers`, возвращающую список слоев, которые можно добавить в последовательную модель Keras.

Мы создадим сверточную нейронную сеть, состоящую из четырех сверточных слоев и последнего плотного слоя, содержащих функцию активации ReLU. Кроме того, непосредственно перед каждым сверточным слоем мы используем вспомогательный слой `ZeroPadding2D` для *дополнения* входных признаков нулями. Допустим, мы используем одноплоскостной кодировщик из главы 6 для кодирования состояния доски в виде матрицы 19×19. Если мы зададим в качестве дополнения значение 2, то по краям этой матрицы будет добавлено два столбца нулей, что увеличит ее размер до 23×23. В данной ситуации мы используем дополнение нулями, чтобы искусственно увеличить размер входа сверточного слоя, дабы операция свертки не привела к чрезмерному уменьшению изображения.

Прежде чем переходить к коду, давайте обсудим некоторые технические нюансы. Напомним, что входы и выходы сверточных слоев являются четырехмерными: мы предоставляем мини-пакет с несколькими фильтрами, каждый из которых имеет по два измерения (ширину и высоту). *Порядок* указания этих четырех измерений (размер мини-пакета, количество фильтров, ширина и высота) является условным, и на практике в основном используется два его варианта. Учтите, что фильтры также часто называются каналами (channels) и обозначаются буквой `C`, а размер мини-пакета – количеством (number) образцов и обозначается буквой `N`. Кроме того, мы можем использовать сокращение `W` для обозначения ширины (width) и `H` для обозначения высоты (height). При использовании таких обозначений двумя основными способами упорядочивания являются NWHC и NCWH. В библиотеке Keras подобный способ упорядочивания называется `data_format`. При этом по очевидным причинам порядок NWHC называется `channels_last`, а NCWH – `channels_first`. При создании одноплоскостного кодировщика доски для игры в го использовался порядок `channels_first` (закодированная доска имеет форму 1,19,19, т. е. *первой* указана единственная закодированная плоскость). Это означает, что в качестве аргумента для всех сверточных слоев мы должны передать `data_format=channels_first`. Давайте рассмотрим модель из файла *small.py*.

Листинг 7.18 ❖ Задание слоев небольшой сверточной сети для предсказания ходов в игре го

```
from keras.layers.core import Dense, Activation, Flatten
from keras.layers.convolutional import Conv2D, ZeroPadding2D

def layers(input_shape):
    return [
        ZeroPadding2D(padding=3, input_shape=input_shape,
                      data_format='channels_first'),
        Conv2D(48, (7, 7), data_format='channels_first'),
        Activation('relu'),

        ZeroPadding2D(padding=2, data_format='channels_first'),
        Conv2D(32, (5, 5), data_format='channels_first'),
        Activation('relu'),

        ZeroPadding2D(padding=2, data_format='channels_first'),
        Conv2D(32, (5, 5), data_format='channels_first'),
        Activation('relu'),

        ZeroPadding2D(padding=2, data_format='channels_first'),
        Conv2D(32, (5, 5), data_format='channels_first'),
```

Дополнение нулями используется для увеличения размера входных изображений.

Порядок channel_first говорит о том, что первым будет указан размер входной плоскости признаков.

```
        Activation('relu'),
        Flatten(),
        Dense(512),
        Activation('relu'),
    ]
```

Функция `layers` возвращает список слоев Keras, которые можно по одному добавить в последовательную модель. Используя эти слои, мы можем создать приложение, отвечающее за первые пять этапов процесса, представленного на рис. 7.1, а именно за загрузку, извлечение и кодирование игровых данных, а также за обучение нейронной сети на их основе. Для обучения мы будем использовать созданный ранее *генератор данных*. Однако сначала давайте импортируем некоторые из важнейших компонентов нашей растущей библиотеки машинного обучения. Для создания приложения нам понадобится обработчик данных игры го, кодировщик и архитектура нейронной сети.

Листинг 7.19 ❖ Импорт основных компонентов нейронной сети для данных игры го

```
from dlgo.data.parallel_processor import GoDataProcessor
from dlgo.encoders.oneplane import OnePlaneEncoder

from dlgo.networks import small
from keras.models import Sequential
from keras.layers.core import Dense
from keras.callbacks import ModelCheckpoint
```

Контрольные точки позволяют сохранять промежуточное состояние модели в ходе длительных экспериментов.

Последним импортируется удобный инструмент Keras `ModelCheckpoint`. При наличии доступа к большому количеству обучающих данных полный цикл обучения модели, состоящий из нескольких эпох, может занять много часов или даже дней. На случай провала эксперимента следует предусмотреть резервную копию. Именно для этого и существуют контрольные точки: они фиксируют состояние модели после каждой эпохи обучения. Даже если что-то пойдет не так, мы сможем продолжить обучение с последней контрольной точки.

Теперь давайте определим обучающие и тестовые данные. Для этого мы сначала инициализируем кодировщик `OnePlaneEncoder`, который будет использоваться для создания обработчика `GoDataProcessor`. С помощью этого обработчика можно создать экземпляр генератора обучающих и тестовых данных, который будет использоваться с моделью Keras.

Листинг 7.20 ❖ Создание генераторов обучающих и тестовых данных

```
go_board_rows, go_board_cols = 19, 19
num_classes = go_board_rows * go_board_cols
num_games = 100

encoder = OnePlaneEncoder((go_board_rows, go_board_cols))

processor = GoDataProcessor(encoder=encoder.name())

generator = processor.load_go_data('train', num_games, use_generator=True)
test_generator = processor.load_go_data('test', num_games, use_generator=True)
```

Создание кодировщика, соответствующего размеру доски.

Инициализация обработчика игровых данных с его помощью.

С помощью обработчика создаются генераторы обучающих и тестовых данных.

Следующим этапом является определение нейронной сети с помощью библиотеки Keras. Для этого используется функция `layers` из файла *small.py* подмодуля *networks* модуля *dlgo*. Мы по одному добавляем слои этой небольшой сети в новую последовательную сеть и завершаем ее плотным слоем с функцией активации softmax. Затем эта модель компилируется с использованием категориальной перекрестной энтропии в качестве функции потерь и обучается методом стохастического градиентного спуска (СГС).

Листинг 7.21 ❖ Создание модели Keras на основе архитектуры из файла small.py

```
input_shape = (encoder.num_planes, go_board_rows, go_board_cols)
network_layers = small.layers(input_shape)
model = Sequential()
for layer in network_layers:
    model.add(layer)
model.add(Dense(num_classes, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='sgd',
     metrics=['accuracy'])
```

Процесс обучения модели Keras с помощью генераторов немного отличается от ее обучения с использованием наборов данных. Теперь вместо вызова метода `fit` модели нам нужно вызвать метод `fit_generator`, а также заменить метод `evaluate` на `evaluate_generator`. Кроме того, сигнатуры этих методов немного отличаются от рассматриваемых ранее. Использование метода `fit_generator` предполагает указание генератора (`generator`), количества эпох (`epochs`) и количества обучающих этапов на эпоху (`steps_per_epoch`). Эти три аргумента обеспечивают необходимый минимум для обучения модели. Контроль над процессом обучения осуществляется с помощью тестовых данных. Для этого мы предоставляем `validation_data` вместе с генератором тестовых данных и указываем количество этапов валидации на эпоху (`validation_steps`). Наконец, мы добавляем в модель функцию обратного вызова (`callback`), которая позволяет отслеживать и возвращать дополнительную информацию в процессе обучения. В данном случае мы используем функции обратного вызова, чтобы задействовать утилиту `ModelCheckpoint` для сохранения состояния модели Keras после каждой эпохи обучения. В качестве эксперимента мы обучим модель в течение пяти эпох на пакете данных размером 128.

Листинг 7.22 ❖ Подгонка и оценка моделей Keras с помощью генераторов

```
epochs = 5
batch_size = 128
model.fit_generator(
  generator=generator.generate(batch_size, num_classes),
  epochs=epochs,
  steps_per_epoch=generator.get_num_samples() / batch_size,
  validation_data=test_generator.generate(
    batch_size, num_classes),
  validation_steps=test_generator.get_num_samples() / batch_size,
  callbacks=[
    ModelCheckpoint('../checkpoints/small_model_epoch_{epoch}.h5')
  ])
model.evaluate_generator(
```

Указание генератора обучающих данных с учетом размера пакета...

...и количества этапов обучения на эпоху.

Дополнительный генератор используется для контроля...

...который также осуществляется в несколько этапов.

После каждой эпохи состояние модели сохраняется с помощью контрольной точки.

```
generator=test_generator.generate(batch_size, num_classes),
steps=test_generator.get_num_samples() / batch_size)
```

Для оценки также указывается генератор и количество этапов.

Если вы намерены запустить этот код самостоятельно, вам следует иметь представление о том, сколько времени может потребоваться для завершения данного эксперимента. При его запуске на ЦПУ одна эпоха обучения может занять несколько часов. Математика, используемая в машинном обучении, имеет много общего с математикой, используемой в компьютерной графике. Поэтому иногда выполнение вычислений на графическом процессоре позволяет значительно ускорить работу – в случае сверточных нейронных сетей обычно на один-два порядка. Библиотека TensorFlow позволяет перенести вычисления на определенные графические процессоры при условии наличия на компьютере соответствующих драйверов.

Если вы хотите использовать для машинного обучения графический процессор, то наилучшей комбинацией будет чип NVIDIA с ОС Windows или Linux. Возможны и другие комбинации, но при их использовании много времени может уйти на решение проблем с драйверами.

На случай, если вы не захотите запускать код самостоятельно или решите сделать это позднее, мы выполнили вычисления за вас. В папке *checkpoints* репозитория GitHub вы найдете пять контрольных точек, соответствующих состоянию модели после каждой эпохи обучения. Вот результат этого вычисления (мы выполнили его на старом ЦПУ ноутбука, чтобы подвигнуть вас как можно скорее обзавестись быстрым графическим процессором):

```
Epoch 1/5
12288/12288 [==============================] - 14053s 1s/step - loss: 3.5514
➥ - acc: 0.2834 - val_loss: 2.5023 - val_acc: 0.6669
Epoch 2/5
12288/12288 [==============================] - 15808s 1s/step - loss: 0.3028
➥ - acc: 0.9174 - val_loss: 2.2127 - val_acc: 0.8294
Epoch 3/5
12288/12288 [==============================] - 14410s 1s/step - loss: 0.0840
➥ - acc: 0.9791 - val_loss: 2.2512 - val_acc: 0.8413
Epoch 4/5
12288/12288 [==============================] - 14620s 1s/step - loss: 0.1113
➥ - acc: 0.9832 - val_loss: 2.2832 - val_acc: 0.8415
Epoch 5/5
12288/12288 [==============================] - 18688s 2s/step - loss: 0.1647
➥ - acc: 0.9816 - val_loss: 2.2928 - val_acc: 0.8461
```

Как видите, после трех эпох обучения мы достигли показателя верности предсказаний 98 % на обучающих данных и 84 % на тестовых данных. Это огромный прогресс по сравнению с моделями из предыдущей главы! Кажется, обучение более крупной сети на реальных данных себя оправдало: наша сеть научилась предсказывать ходы из 100 партий практически идеально и достаточно хорошо справляться с обобщением. Верность предсказаний в 84 % на тестовых данных является вполне удовлетворительным результатом. С другой стороны, количество ходов в 100 партиях – это слишком небольшой набор данных, и мы пока не знаем,

насколько результативной окажется модель на более крупной выборке. В конце концов, нам нужен мощный бот, способный конкурировать с сильными противниками, а не просто расправляться с крошечным набором данных.

Чтобы создать по-настоящему сильного противника, нам потребуются более качественные кодировщики игровых данных. Одноплоскостной кодировщик из главы 6 является хорошей отправной точкой, но он не отражает всей сложности, с которой мы имеем дело. В разделе 7.4 описано два более сложных кодировщика, позволяющих повысить результативность обучения.

7.4. Создание более реалистичных кодировщиков данных игры го

В главах 2 и 3 мы подробно разобрали правило ко. Напомним, что это правило существует для предотвращения бесконечных циклов в игре: нельзя помещать на доску камень, если это приведет к восстановлению уже существовавшего ранее игрового состояния. При выбранном случайным образом состоянии доски на вопрос о применимости правила ко невозможно ответить, не изучив последовательность ходов, приведшую к этому состоянию. В частности, наш одноплоскостной кодировщик, который кодирует черные камни значением –1, белые – значением 1, а пустые точки – значением 0, ничего не способен узнать о правиле ко. Это только один пример, но он показывает, что созданный в главе 6 кодировщик `OnePlaneEncoder` является слишком примитивным, чтобы охватить все необходимое для создания мощного бота для игры в го.

В этом разделе мы обсудим два более сложных кодировщика, способных повысить точность предсказания ходов. Первый из них называется `SevenPlaneEncoder` и состоит из следующих семи плоскостей признаков. Каждая плоскость представляет собой матрицу 19×19 и описывает различный набор признаков:

- первая плоскость кодирует значением 1 каждый *белый* камень, имеющий *одну* степень свободы, в противном случае используется значение 0;
- вторая и третья плоскости кодируют значением 1 белые камни с двумя или как минимум тремя степенями свободы соответственно;
- плоскости с четвертой по шестую делают то же самое для черных камней; Они кодируют черные камни с одной, двумя или как минимум тремя степенями свободы;
- последняя плоскость кодирует значением 1 точки доски, помещать камни на которые запрещено правилом ко.

Помимо явного кодирования концепций правила ко, этот набор признаков также позволяет нам моделировать степени свободы и различать черные и белые камни. Камни с одной степенью свободы имеют дополнительное тактическое значение, поскольку могут быть захвачены на следующем ходу. (В игре го о камнях с одной степенью свободы принято говорить как о находящихся в положении *атари*.) Возможность модели непосредственно «наблюдать» это свойство позволяет ей легче выяснить его значение для игрового процесса. Создавая плоскости для таких концепций, как правило ко и количество степеней свободы, мы намекаем модели о важности этих концепций, не объясняя при этом, в чем она заключается.

Давайте посмотрим, как можно реализовать этот кодировщик на основе базовой версии Encoder из модуля *encoders*. Поместите следующий код в файл *sevenplane.py*.

Листинг 7.23 ❖ Инициализация простого кодировщика, состоящего из семи плоскостей

```
import numpy as np

from dlgo.encoders.base import Encoder
from dlgo.goboard import Move, Point

class SevenPlaneEncoder(Encoder):
    def __init__(self, board_size):
        self.board_width, self.board_height = board_size
        self.num_planes = 7

    def name(self):
        return 'sevenplane'
```

В данном случае следует обратить внимание на кодирование состояния доски, которое выполняется следующим образом.

Листинг 7.24 ❖ Кодирование игрового состояния с помощью кодировщика SevenPlaneEncoder

```
    def encode(self, game_state):
        board_tensor = np.zeros(self.shape())
        base_plane = {game_state.next_player: 0,
                      game_state.next_player.other: 3}
        for row in range(self.board_height):
            for col in range(self.board_width):
                p = Point(row=row + 1, col=col + 1)
                go_string = game_state.board.get_go_string(p)
                if go_string is None:
                    if
     game_state.does_move_violate_ko(game_state.next_player,
                                                    Move.play(p)):
                        board_tensor[6][row][col] = 1
                else:
                    liberty_plane = min(3, go_string.num_liberties) - 1
                    liberty_plane += base_plane[go_string.color]
                    board_tensor[liberty_plane][row][col] = 1
        return board_tensor
```

Кодирование ходов, запрещенных правилом ко.

Кодирование черных и белых камней с 1, 2 или большим количеством степеней свободы.

Чтобы завершить это определение, нам также нужно реализовать несколько вспомогательных методов, составляющих интерфейс кодировщика Encoder.

Листинг 7.25 ❖ Реализация оставшихся методов для создания кодировщика SevenPlaneEncoder

```
    def encode_point(self, point):
        return self.board_width * (point.row - 1) + (point.col - 1)

    def decode_point_index(self, index):
        row = index // self.board_width
        col = index % self.board_width
        return Point(row=row + 1, col=col + 1)

    def num_points(self):
        return self.board_width * self.board_height
```

```
    def shape(self):
        return self.num_planes, self.board_height, self.board_width

def create(board_size):
    return SevenPlaneEncoder(board_size)
```

Нам осталось обсудить еще один кодировщик, напоминающий `SevenPlaneEncoder` и включающий 11 плоскостей признаков. В этом кодировщике под названием `SimpleEncoder`, который находится в файле *simple.py* в модуле *encoders* репозитория GitHub, используются следующие плоскости признаков:

- первые четыре плоскости признаков описывают черные камни с одной, двумя, тремя или четырьмя степенями свободы;
- следующие четыре плоскости признаков описывают белые камни с одной, двумя, тремя или четырьмя степенями свободы;
- девятая плоскость имеет значение 1, если право следующего хода принадлежит черным, а десятая – если белым;
- последняя плоскость признаков зарезервирована для правила ко.

Этот кодировщик с 11 плоскостями напоминает предыдущий, но является более конкретным в плане очередности ходов и количества степеней свободы камней. Оба варианта являются отличными кодировщиками и способны значительно повысить производительность модели.

В главах 5 и 6 мы обсуждали методы улучшения моделей глубокого обучения, однако во всех экспериментах кое-что оставалось неизменным: в качестве оптимизатора мы использовали стохастический градиентный спуск. Несмотря на то что СГС является отличной отправной точкой, существуют оптимизаторы, способные заметно улучшить процесс обучения, и в следующем разделе мы обсудим два из них – *Adagrad* и *Adadelta*.

7.5. Эффективное обучение с помощью адаптивных градиентов

В этом разделе мы обсудим несколько оптимизаторов, позволяющих еще больше повысить производительность моделей для предсказания ходов в игре го. Из главы 5 вы помните, что СГС предусматривает довольно примитивное правило обновления. Если для параметра W мы получаем ошибку ΔW при скорости обучения α, то обновление данного параметра с помощью СГС сводится к простому вычислению выражения $W - \alpha\Delta W$.

Во многих случаях это правило обновления может обеспечить хороший результат, однако у него есть и недостатки. Для их компенсации существует множество отличных расширений СГС.

7.5.1. Затухание и импульс в СГС

Например, широко используемым приемом является обеспечение постепенного *затухания* скорости обучения после очередного этапа обновления. Как правило, этот прием дает хорошие результаты, поскольку в самом начале обучения сети значительные обновления позволяют приблизиться к минимуму функции потерь. Однако по достижении определенного уровня обновления следует свести

лишь к необходимым уточнениям, которые не оказывают отрицательного воздействия на достигнутые результаты. Как правило, затухание скорости обучения задается с помощью *коэффициента затухания*, выраженного в процентах.

Другим популярным методом является метод *импульса*, предполагающий добавление части значения предыдущего обновления к текущему. Например, если W – это вектор параметров, который требуется обновить, ΔW – текущее значение градиента, вычисленное для вектора W, а U – величина предыдущего изменения веса, то следующий шаг обновления будет выглядеть следующим образом:

$$W \leftarrow W - \alpha(\gamma U + (1 + \gamma)\partial W).$$

Величина γ, оставшаяся с предыдущего шага обновления, называется *слагаемым импульса*. Если оба градиентных слагаемых указывают примерно в одном и том же направлении, то следующий шаг обновления «усиливается» (получает импульс). Если градиенты указывают в противоположных направлениях, они компенсируют друг друга, и градиент затухает. Эта техника называется методом *импульса* из-за сходства с одноименным физическим понятием. Если представить функцию потерь в виде поверхности, а лежащие на ней параметры – в виде шарика, то обновление параметра будет описывать движение этого шарика. Тогда о градиентном спуске можно думать как о скатывающемся вниз шарике. Если несколько последних (градиентных) шагов указывают в одном и том же направлении, шарик набирает скорость и быстрее достигнет своего пункта назначения – минимума функции или впадины на поверхности. Метод импульса работает аналогичным образом.

Для применения затухания скорости обучения, метода импульса или обоих этих методов с помощью библиотеки Keras достаточно передать значения соответствующих коэффициентов экземпляру `SGD`. Например, если мы хотим использовать СГС со скоростью обучения 0,1, коэффициентом затухания 1 % и импульсом 90 %, нам нужно сделать следующее.

Листинг 7.26 ❖ Инициализация СГС с использованием импульса и затухания скорости обучения в Keras

```
from keras.optimizers import SGD
sgd = SGD(lr=0.1, momentum=0.9, decay=0.01)
```

7.5.2. Оптимизация нейронных сетей с помощью метода Adagrad

Как затухание скорости обучения, так и метод импульса позволяют уточнить простой алгоритм СГС, однако они не устраняют всех его недостатков. Например, профессиональные игроки в го практически всегда делают первые несколько ходов на линиях доски с третьей по пятую и никогда на первой или второй. В финале партии ситуация меняется на противоположную, т. е. большое количество ходов делается на краях доски. Во всех моделях глубокого обучения, с которыми мы работали до сих пор, последним являлся плотный слой размером с доску (в данном случае 19×19). Каждый нейрон этого слоя соответствует позиции на доске. Если мы используем СГС, с затуханием или импульсом, либо без них, *к каждому из этих нейронов применяется одинаковая скорость обучения*. Это может приводить к проблемам. Если вы плохо перетасовали обучающие данные и скорость обучения снизилась настолько, что последние ходы партии, совершенные на первой

или второй линии, больше не получают значительных обновлений, то никакого обучения не происходит. В общем, нам нужно сделать так, чтобы редко наблюдаемые закономерности получали достаточно большие обновления, а часто наблюдаемые – постепенно уменьшающиеся.

Для решения этой проблемы, вызванной заданием глобальной скорости обучения, можно использовать методы *адаптивного* градиента. Далее мы рассмотрим два таких метода: *Adagrad* и *Adadelta*.

Метод Adagrad не предполагает использования глобальной скорости обучения. Вместо этого мы *адаптируем скорость обучения отдельных параметров*. Метод Adagrad работает довольно хорошо при наличии большого количества данных, закономерности в которых обнаруживаются относительно редко. Оба этих критерия применимы к нашему случаю: у нас есть очень много данных, а игра профессионалов настолько сложна, что конкретные комбинации ходов встречаются в наборе данных нечасто, хотя и считаются профессиональными игроками стандартной стратегией.

Допустим, у нас есть вектор весов W длиной l (в данном случае для простоты используется пример векторов, однако этот метод применим и к тензорам) с отдельными элементами W_i. Для данного градиента ΔW при использовании простого СГС со скоростью обучения α правило обновления каждого W_i выглядит следующим образом:

$$W_i \leftarrow W_i - \alpha \partial W_i.$$

При использовании оптимизатора Adagrad мы заменяем α слагаемым, которое динамически адаптируется для каждого индекса i, исходя из предыдущего обновления W_i. Фактически при использовании этого метода индивидуальная скорость обучения будет обратно пропорциональна предыдущим обновлениям. Итак, при использовании оптимизатора Adagrad параметры обновляются следующим образом:

$$W \leftarrow W - \frac{\alpha}{\sqrt{G + \varepsilon}} \cdot \partial W.$$

В данном уравнении ε – это небольшое положительное значение, предотвращающее деление на ноль, а Gi,i – это сумма квадратов градиентов параметра W_i, накопленная за предыдущие шаги. Мы используем запись Gi,i, поскольку это слагаемое можно рассматривать как часть квадратной матрицы G длиной l, в которой все диагональные элементы Gj,j имеют описанную выше форму, а элементы, стоящие вне главной диагонали, равны 0. Матрица такого вида называется *диагональной*. Мы обновляем G после каждого обновления параметра, прибавляя последние градиентные вклады к значениям диагональных элементов. Вот так выглядит краткая форма записи этого правила обновления, не зависящая от индекса i:

$$W \leftarrow W - \frac{\alpha}{\sqrt{G + \varepsilon} \cdot \partial W}.$$

Учтите, что, поскольку G представляет собой матрицу, нам требуется прибавить ε к каждому элементу Gi,j и разделить α на каждый такой элемент. Более того,

под $G \cdot \Delta W$ подразумевается матричное умножение G на ΔW. Компиляция модели Keras с оптимизатором Adagrad выглядит следующим образом.

Листинг 7.27 ❖ Использование оптимизатора Adagrad для компиляции модели Keras

```
from keras.optimizers import Adagrad
adagrad = Adagrad()
```

Главное преимущество метода Adagrad по сравнению с другими методами СГС заключается в отсутствии необходимости вручную задавать скорость обучения. Нахождение хорошей сетевой архитектуры и настройка параметров модели и без того требует усилий. На самом деле мы могли бы изменить начальную скорость обучения в Keras, используя фрагмент кода `Adagrad(lr = 0.02)`, однако делать это не рекомендуется.

7.5.3. Уточнение адаптивных градиентов с помощью Adadelta

Оптимизатор *Adadelta* напоминает Adagrad и является его расширением. В данном случае вместо накопления всех прошлых (квадратов) градиентов в матрице G мы используем ту же идею, что и в методе импульса, сохраняя лишь *часть последнего обновления* и прибавляя к ней текущий градиент:

$$G \leftarrow \gamma G + (1 - \gamma)\partial W.$$

Вкратце идея такова, однако детали, обеспечивающие эффективность оптимизатора Adadelta, довольно сложны, и их обсуждение выходит за рамки данной книги. Для получения более подробной информации мы рекомендуем изучить статью по адресу: **arxiv.org/abs/1212.5701**.

В модели Keras оптимизатор Adadelta используется следующим образом.

Листинг 7.28 ❖ Применение оптимизатора Adadelta к моделям Keras

```
from keras.optimizers import Adadelta
adadelta = Adadelta()
```

Оптимизаторы Adagrad и Adadelta оказываются более полезными для обучения глубоких нейронных сетей на данных игры го по сравнению со стохастическим градиентным спуском. Далее в книге мы будем часто использовать какой-то из этих оптимизаторов в более продвинутых моделях.

7.6. Проведение экспериментов и оценка эффективности

В этой и предыдущих двух главах мы рассмотрели множество методов глубокого обучения. Мы поделились с вами некоторыми советами и примерами архитектур, которые можно использовать в качестве отправной точки, однако теперь вам предстоит заняться обучением собственных моделей. В экспериментах по машинному обучению крайне важно попробовать разные комбинации таких *гиперпараметров*, как количество слоев, их типы, количество эпох обучения и т. д. В случае с глубокими нейронными сетями число доступных вариантов может оказаться огромным. Не всегда легко понять, как настройка того или иного параметра влияет на производительность модели. Специалисты по глубокому обучению могут опереться на большой массив экспериментальных данных и теоретических

аргументов, выведенных за десятилетия исследований. Эта книга не способна передать всю глубину знания, однако она может помочь вам приступить к наработке собственных навыков и развитию интуиции.

Для достижения наилучших результатов в экспериментах вроде нашего, а именно в обучении нейронной сети верному предсказанию ходов в игре го, ключевым фактором является *быстрый цикл экспериментов*. На построение архитектуры модели, ее обучение, наблюдение и оценку показателей производительности, а также коррекцию модели и повторный запуск должно уходить как можно меньше времени. Если посмотреть на конкурсы, проводящиеся на платформах вроде **kaggle.com**, можно заметить, что чаще всего побеждают команды, сделавшие *наибольшее количество попыток*. К счастью, библиотека Keras позволяет проводить эксперименты довольно быстро. Это одна из главных причин, по которой мы выбрали ее в качестве фреймворка для глубокого обучения в этой книге. Вы наверняка согласитесь с тем, что Keras позволяет быстро создавать нейронные сети и легко изменять условия эксперимента.

7.6.1. Руководство по тестированию архитектур и гиперпараметров

Давайте рассмотрим несколько практических соображений, которые следует иметь в виду при создании сети для предсказания ходов.

- При создании сети для предсказания ходов в игре го хорошим вариантом являются сверточные нейронные сети. Учтите, что работа исключительно с плотными слоями приведет к ухудшению качества предсказания. Создавайте сети, состоящие из нескольких сверточных слоев и одного или двух плотных слоев в конце. В следующих главах мы рассмотрим более сложные архитектуры, а до тех пор работайте со сверточными сетями.
- Варьируйте размер фильтра в сверточных слоях и смотрите, как это влияет на производительность модели. Как правило, размеры фильтра от 2 до 7 являются наиболее подходящими.
- При использовании слоев пулинга поэкспериментируйте как с функцией максимума, так и с функцией среднего значения, однако не выбирайте слишком большой размер пула. В вашем случае в качестве верхнего предела можно выбрать значение 3. Вы также можете попробовать создать сети без слоев пулинга, что является более дорогостоящим в плане вычислений, но дает довольно хорошие результаты.
- Для регуляризации используйте слои исключения. В главе 6 мы говорили о том, что исключение позволяет предотвратить переобучение модели. Как правило, добавление слоев исключения оказывается довольно полезным, при условии что вы не используете слишком много таких слоев и не задаете слишком большое значение коэффициентов исключения.
- В последнем слое используйте функцию активации softmax для получения распределения вероятностей и дополните ее категориальной перекрестной энтропией, которая очень хорошо подходит для решения вашей задачи.
- Поэкспериментируйте с различными функциями активации. Вы уже знакомы с функцией ReLU, которую вам пока следует использовать по умолчанию, а также с сигмоидами. Библиотека Keras предусматривает множество других функций активации, например elu, selu, PReLU и LeakyReLU. Их об-

суждение выходит за рамки данной книги, но вы можете найти подробную информацию об их использовании по адресу: **keras.io/activations**.

- Варьирование размера мини-пакета влияет на производительность модели. Для решения задач, связанных с предсказанием, как в примере с данными MNIST из главы 5, рекомендуется выбирать размер мини-пакета, сопоставимый с количеством классов. При работе с набором данных MNIST часто используются размеры мини-пакетов в диапазоне от 10 до 50. Если данные идеально рандомизированы, каждый градиент будет получать информацию от каждого класса, что обычно повышает результативность алгоритма СГС. В игре го некоторые ходы совершаются чаще, чем другие. Например, углы доски гораздо реже заполняются камнями, особенно по сравнению со звездными пунктами. Это называется *дисбалансом классов* в данных. В этом случае не стоит ожидать, что мини-пакет будет охватывать все классы. Вам следует работать с мини-пакетами, размеры которых находятся в диапазоне от 16 до 256 (что чаще всего встречается в литературе). Выбор оптимизатора также довольно сильно влияет на процесс обучения сети. Вы можете поэкспериментировать с алгоритмом СГС, затуханием скорости обучения, а также с оптимизаторами Adagrad и Adadelta. На странице **keras.io/optimizers** вы найдете дополнительные оптимизаторы, способные положительно повлиять на процесс обучения модели. Кроме того, вам следует обдуманно подойти к выбору количества эпох обучения. Использование контрольных точек и отслеживание показателей производительности после каждой эпохи позволяет определить момент, когда результаты обучения перестают улучшаться. В заключительном разделе этой главы мы поговорим об оценке производительности. Как правило, при наличии достаточной вычислительной мощности лучше задать слишком большое количество эпох, чем слишком маленькое. Если результаты обучения модели перестанут улучшаться или даже ухудшатся из-за переобучения, вы все равно сможете использовать одну из предыдущих контрольных точек для создания своего бота.

Инициализаторы весов

Еще одним важным аспектом настройки глубоких нейронных сетей является инициализация весов перед запуском процесса обучения. Поскольку оптимизация сети предполагает нахождение набора весов, соответствующих минимуму на поверхности потерь, начальные веса имеют большое значение. В реализации сети, описанной в главе 5, начальные веса были назначены *случайным образом*, что, как правило, является плохой идеей.

Инициализаторы весов являются интересной темой для исследования и заслуживают отдельной главы. Библиотека Keras предусматривает множество способов инициализации весов, и каждый слой с весами может быть соответствующим образом инициализирован. Причина, по которой мы не обсуждаем их здесь, заключается в том, что инициализаторы, выбранные Keras по умолчанию, как правило, настолько хороши, что не нуждаются в каких-либо изменениях. Обычно внимание требуется уделить другим аспектам сети. Однако знать о существующих различиях полезно, и опытные пользователи могут поэкспериментировать с инициализаторами Keras, которые можно найти по адресу **keras.io/initializers**.

7.6.2. Оценка показателей производительности для обучающих и тестовых данных

В разделе 7.3 были продемонстрированы результаты обучения на небольшом наборе данных. В приведенном примере использовалась относительно небольшая сверточная сеть, обученная в течение пяти эпох. В этом эксперименте мы отслеживали потери и верность предсказаний на обучающих данных, а тестовые использовали для контроля. В конце мы вычислили верность предсказаний на тестовых данных. Старайтесь придерживаться именно такой последовательности этапов, а для определения момента прекращения обучения и выявления проблем используйте следующие рекомендации:

- верность предсказаний и показатель потерь должны улучшаться после каждой эпохи обучения. На более поздних этапах эти показатели, скорее всего, будут демонстрировать лишь небольшие колебания. Если на протяжении нескольких эпох каких-либо улучшений не наблюдается, вероятно, имеет смысл прекратить обучение;
- в то же время необходимо отслеживать значения потери и верности предсказаний на тестовой выборке. На протяжении ранних эпох обучения значение потери на тестовой выборке постоянно снижается, а в более поздние эпохи часто можно видеть, как оно достигает плато и снова начинает расти, что является верным признаком переобучения сети;
- если вы используете контрольные точки, выбирайте состояние модели после эпохи обучения, характеризующейся высокой верностью предсказания на обучающих данных и низким значением ошибки на тестовых;
- при высоких потерях обучения и контроля попробуйте выбрать более глубокую сетевую архитектуру или скорректируйте гиперпараметры;
- низкая ошибка обучения и высокая ошибка контроля свидетельствуют о переобучении модели. Этого обычно не происходит при использовании по-настоящему большого набора обучающих данных. При наличии более 170 000 партий игры в го, включающих миллионы ходов, вам не о чем беспокоиться;
- при выборе размера обучающего набора учитывайте возможности своего аппаратного обеспечения. Процесс обучения, длящийся часами, может сильно утомить. Чтобы этого не произошло, попробуйте найти самую производительную модель, которая лучше всего работает на наборе данных среднего размера, а затем обучите ее еще раз на максимально возможном наборе данных;
- при отсутствии хорошего графического процессора вы можете использовать для обучения модели облачный сервис. В приложении Г мы покажем, как обучить модель на графическом процессоре с помощью Amazon Web Services (AWS);
- при сравнении прогонов не торопитесь останавливать прогон, который выглядит хуже предыдущего. Некоторые процессы обучения происходят медленнее, чем другие, и в конечном итоге могут обеспечить результаты, сопоставимые или даже превосходящие показатели других моделей.

Вас, вероятно, интересует, насколько мощным может быть бот, созданный с помощью методов, представленных в этой главе. Теоретически сеть никогда не смо-

жет играть в го лучше, чем это позволяют данные, на основе которых она обучена. В частности, использование только методов глубокого обучения с учителем, как это делалось в последних трех главах, не позволит игровому боту превзойти человека. На практике при наличии достаточной вычислительной мощности и времени определенно можно достичь уровня игры, примерно соответствующего 2-му дану.

Чтобы превзойти игру человека, необходимо применить методы *обучения с подкреплением*, описанные в главах с 9 по 12. После этого вы сможете объединить поиск по дереву из главы 4, обучение с подкреплением и методы глубокого обучения с учителем для создания еще более мощных ботов в главах 13 и 14.

Однако, прежде чем углубиться в методологию создания мощных ботов, мы поговорим о *развертывании* бота и об обеспечении его взаимодействия со средой путем игры против человека и других ботов.

7.7. Резюме

- Широко распространенный формат Smart Game Format (SGF) для данных игры го и других игр полезен при создании наборов данных для обучения нейронных сетей.
- Для ускорения процесса данные игры го можно обрабатывать параллельно и представлять с помощью генераторов.
- При наличии записей партий, сыгранных любителями и профессионалами, можно создавать модели глубокого обучения, способные достаточно хорошо предсказывать ходы в игре го.
- Важные свойства обучающих данных можно явно закодировать в виде *плоскостей признаков*. После этого модель сможет быстро изучить связи между плоскостями признаков и результатами, которые вы пытаетесь предсказать. В случае с ботом для игры в го вы можете добавить плоскости признаков, представляющие такие концепции, как количество степеней свободы (соседние пустые точки) цепочки камней.
- Эффективность обучения можно повысить с помощью таких методов адаптивного градиента, как Adagrad или Adadelta. Эти алгоритмы позволяют регулировать скорость обучения на лету.
- Сквозное обучение модели может обеспечиваться относительно небольшим сценарием, который можно использовать в качестве шаблона для проведения собственных экспериментов.

Глава 8

Развертывание ботов

В этой главе:

- создание комплексного приложения для обучения и запуска бота для игры в го;
- создание интерфейса для игры против собственного бота;
- организация локальной игры бота против других ботов;
- развертывание бота на сервере для игры го.

К этому моменту вы уже научились создавать и обучать модель глубокого обучения для предсказания ходов в игре го, однако вы пока не знаете, как объединить изученные компоненты в приложение, способное играть с противниками. Обучение нейронной сети – это только один из этапов создания комплексного приложения, вне зависимости от того, собираетесь ли вы играть против собственного бота сами или организовать его игру с другими ботами. Обученную модель необходимо интегрировать в движок, против которого можно играть.

В этой главе мы создадим простую модель го-сервера и два фронтенда. Сначала мы обсудим HTTP-фронтенд, который вы сможете использовать для игры против своего бота. Затем поговорим о протоколе Go Text Protocol (GTP), который позволяет ботам для игры в го обмениваться информацией. Благодаря этому широко используемому протоколу ваш бот может играть против других ботов вроде *GNU Go* или *Pachi*. Наконец, мы поговорим о развертывании бота для игры в го на платформе Amazon Web Services (AWS) и его подключении к серверу Online Go Server (OGS). После этого ваши боты смогут играть с другими ботами и игроками по всему миру и даже участвовать в турнирах. Для этого придется решить следующие задачи:

- **создание агента для предсказания ходов**: нейронные сети, обученные в главах 6 и 7, должны быть интегрированы в среду, позволяющую использовать их в игровом процессе. Для этого в разделе 8.1 мы снова обратимся к идее *агентов* из главы 3, в которой мы создавали агента, выбирающего ходы случайным образом;
- **создание графического интерфейса**: для более комфортной игры с ботом людям требуется графический интерфейс. И хотя до сих пор мы обходились командной строкой, в разделе 8.2 мы создадим удобный интерфейс для нашего бота;
- **развертывание бота на облачном сервисе**: если в вашем компьютере не установлен мощный графический процессор, вам не удастся создать по-настоящему сильного бота для игры в го. К счастью, большинство крупных облачных провайдеров предоставляет экземпляры графических процессо-

ров по требованию. Однако даже при наличии мощного графического процессора для обучения вы все равно можете рассмотреть возможность размещения своей обученной модели на сервере. В разделе 8.3 мы покажем, как это делается, а в приложении Г вы найдете более подробную информацию об использовании сервиса AWS;

- **взаимодействие с другими ботами**: для взаимодействия друг с другом люди используют графические и другие интерфейсы. Для обеспечения взаимодействия ботов применяется стандартный протокол. В разделе 8.4 будет представлен широко распространенный протокол Go Text Protocol (GTP), который позволяет:
  - **организовать игру бота против других ботов**: в разделе 8.5 мы создадим GTP-интерфейс, обеспечивающий локальную игру бота против двух других программ для игры в го с целью оценки его результативности;
  - **развернуть бота на онлайн-сервере для игры в го**: в разделе 8.6 мы поговорим о том, как разместить бота на онлайн-сервере для игры в го, чтобы он мог играть с зарегистрированными пользователями, другими ботами, а также участвовать в рейтинговых играх и турнирах. Большая часть этого материала носит технический характер, поэтому более подробную информацию вы найдете в приложении Д.

8.1. Создание агента для предсказания ходов на основе глубокой нейронной сети

Теперь, когда у нас есть все строительные блоки для создания мощных нейронных сетей для данных игры го, пришло время интегрировать эти сети в *агента*, который позволит их задействовать. В главе 3 мы определили понятие `Agent` как класс, способный выбирать следующий ход для текущего игрового состояния путем реализации метода `select_move`. Давайте создадим агента `DeepLearningAgent`, используя модели Keras и кодировщик доски для игры в го. Для этого поместите следующий код в файл *predict.py*, находящийся в подмодуле *agent* модуля *dlgo*.

Листинг 8.1 ❖ Инициализация агента с помощью модели Keras и кодировщика доски

```
import numpy as np

from dlgo.agent.base import Agent
from dlgo.agent.helpers import is_point_an_eye
from dlgo import encoders
from dlgo import goboard
from dlgo import kerasutil

class DeepLearningAgent(Agent):
    def __init__(self, model, encoder):
        Agent.__init__(self)
        self.model = model
        self.encoder = encoder
```

Мы будем использовать кодировщик для преобразования состояния доски в признаки, а модель – для предсказания следующего хода. Фактически мы используем модель для вычисления распределения вероятностей различных ходов, из которого впоследствии сделаем выборку.

Листинг 8.2 ❖ Кодирование состояния доски и предсказание вероятностей ходов с помощью модели

```
def predict(self, game_state):
    encoded_state = self.encoder.encode(game_state)
    input_tensor = np.array([encoded_state])
    return self.model.predict(input_tensor)[0]

def select_move(self, game_state):
    num_moves = self.encoder.board_width * self.encoder.board_height
    move_probs = self.predict(game_state)
```

Теперь мы немного изменим распределение вероятностей, хранящееся в `move_probs`. Сначала возведем в куб все значения, чтобы резко увеличить расстояние между наиболее вероятными и наименее вероятными ходами. Нам нужно, чтобы наилучшие из возможных ходов выбирались чаще. Затем мы используем прием под названием *обрезка*, чтобы не позволить значению вероятности хода слишком сильно приблизиться к 0 или 1. Для этого задается небольшое положительное значение, $\epsilon = 0{,}000001$, после чего значения, не превышающие ϵ, приравниваются к ϵ, а значения, превышающие $1 - \epsilon$, приравниваются к $1 - \epsilon$. Затем мы нормализуем полученные значения, чтобы снова получить распределение вероятностей.

Листинг 8.3 ❖ Возведение в куб, обрезка и перенормировка распределения вероятностей ходов

```
move_probs = move_probs ** 3
eps = 1e-6
move_probs = np.clip(move_probs, eps, 1 - eps)
move_probs = move_probs / np.sum(move_probs)
```

Увеличение расстояния между наиболее вероятными и наименее вероятными ходами.

Предотвращение слишком сильного приближения значения вероятности хода к 0 или 1.

Перенормировка с целью получения нового распределения вероятностей.

Мы выполняем это преобразование, потому что хотим сделать выборку ходов из полученного распределения на основе их вероятностей. Альтернативная стратегия состоит в том, чтобы всегда выбирать самый вероятный ход (соответствующий максимуму кривой распределения). Преимущество используемого нами подхода заключается в возможности выбора других ходов, что бывает особенно полезным при отсутствии единственного сильно выделяющегося хода.

Листинг 8.4 ❖ Выбор ходов из ранжированного списка

```
candidates = np.arange(num_moves)
ranked_moves = np.random.choice(
    candidates, num_moves, replace=False, p=move_probs)
for point_idx in ranked_moves:
    point = self.encoder.decode_point_index(point_idx)
    if game_state.is_valid_move(goboard.Move.play(point)) and \
            not is_point_an_eye(game_state.board, point,
game_state.next_player):
        return goboard.Move.play(point)
return goboard.Move.pass_turn()
```

Преобразование вероятностей в ранжированный список ходов.

Отбор кандидатов на звание следующего хода.

Последовательный перебор элементов списка с целью нахождения допустимого хода, не приводящего к уменьшению глазного пространства.

При отсутствии допустимых и несамоубийственных вариантов ход пропускается.

Для удобства мы также хотим сохранить DeepLearningAgent для дальнейшего использования. На практике обычно происходит следующее: мы обучаем модель глубокого обучения и создаем агента, которого затем сохраняем. В дальнейшем этот агент десериализуется и используется для игры с людьми или другими ботами. Для сериализации используется соответствующий формат Keras. Модель Keras сохраняется в формате сериализации HDF5. Файлы HDF5 содержат гибкие *группы*, которые используются для хранения *метаинформации* и *данных*. С помощью метода `model.save("model_path.h5")` можно сохранить всю модель Keras, т. е. архитектуру нейронной сети и все веса, в локальном файле *model_path.h5*. Единственное, что необходимо сделать перед сохранением модели Keras таким способом, – это установить библиотеку Python h5py, например с помощью команды `pip install h5py`.

Чтобы сохранить агент целиком, можно добавить дополнительную группу для хранения информации о кодировщике доски для игры в го.

Листинг 8.5 ❖ Сериализация агента DeepLearningAgent

```
def serialize(self, h5file):
    h5file.create_group('encoder')
    h5file['encoder'].attrs['name'] = self.encoder.name()
    h5file['encoder'].attrs['board_width'] = self.encoder.board_width
    h5file['encoder'].attrs['board_height'] = self.encoder.board_height
    h5file.create_group('model')
    kerasutil.save_model_to_hdf5_group(self.model, h5file['model'])
```

Теперь, когда мы разобрались с сериализацией модели, давайте рассмотрим способ ее загрузки из файла HDF5.

Листинг 8.6 ❖ Десериализация агента DeepLearningAgent из файла HDF5

```
def load_prediction_agent(h5file):
    model = kerasutil.load_model_from_hdf5_group(h5file['model'])
    encoder_name = h5file['encoder'].attrs['name']
    if not isinstance(encoder_name, str):
        encoder_name = encoder_name.decode('ascii')
    board_width = h5file['encoder'].attrs['board_width']
    board_height = h5file['encoder'].attrs['board_height']
    encoder = encoders.get_encoder_by_name(
        encoder_name, (board_width, board_height))
    return DeepLearningAgent(model, encoder)
```

Мы закончили с определением агента глубокого обучения. Теперь нам нужно обеспечить его подключение и взаимодействие со средой. Для этого мы внедрим DeepLearningAgent в веб-приложение, позволяющее людям играть с ботом в их собственном браузере.

8.2. Создание веб-интерфейса для бота

В главах 6 и 7 мы создали и обучили нейронную сеть, предсказывающую ходы, которые сделал бы человек в игре го. В разделе 8.1 мы преобразовали эту модель для *предсказания* ходов в агента DeepLearningAgent для *выбора* ходов. Теперь пришло время сыграть против этого бота! В главе 3 мы создали простой интерфейс,

позволяющий вводить ходы с клавиатуры, после чего наш примитивный RandomBot выводил на консоль свой ответ. Теперь у нас есть более изощренный бот, который заслуживает лучшего интерфейса для взаимодействия с человеком-игроком.

В этом разделе мы подключим агента DeepLearning Agent к веб-приложению Python, чтобы иметь возможность играть против него в веб-браузере. Для передачи этого агента через HTTP мы применим легковесную библиотеку Flask. На стороне браузера будем использовать библиотеку JavaScript под названием jgoboard, чтобы визуализировать доску для игры в го, которую могут использовать люди. Код находится в подмодуле *httpfrontend* модуля *dlgo* репозитория GitHub. Мы не будем обсуждать этот код здесь, чтобы не отвлекаться от главной темы – создания ИИ для игры в го – и не углубляться в методы веб-разработки на других языках вроде HTML или JavaScript. Вместо этого мы кратко рассмотрим принцип работы этого приложения и способ его использования. На рис. 8.1 представлен обзор приложения, которое нам предстоит создать в этой главе.

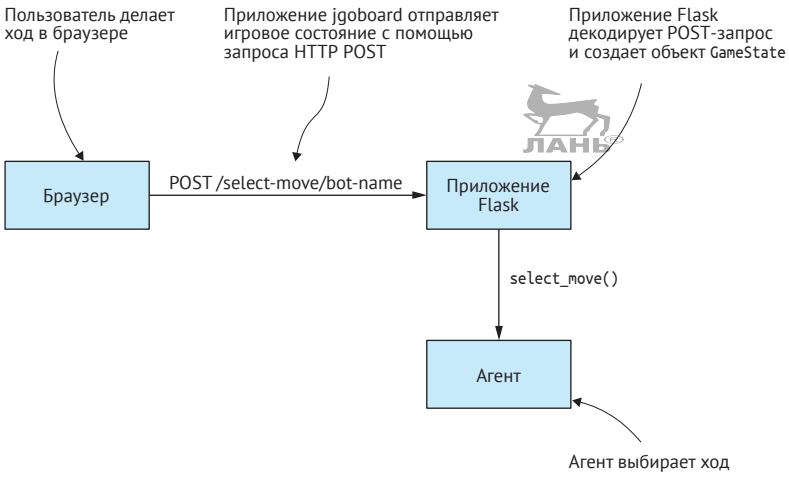


Рис. 8.1 ❖ Создание веб-интерфейса для игрового бота. Модуль *httpfrontend* запускает веб-сервер Flask, который декодирует HTTP-запросы и передает их одному или нескольким агентам для игры в го. В браузере клиент, созданный на основе библиотеки jgoboard, взаимодействует с сервером по протоколу HTTP

В подмодуле *httpfrontend* находится файл *server.py*, который содержит единственный хорошо документированный метод get_web_app, применяемый для возврата веб-приложения, которое требуется запустить. Далее приведен пример использования метода get_web_app для загрузки случайного бота в веб-приложение.

Листинг 8.7 ❖ Регистрация случайного агента и запуск веб-приложения

```
from dlgo.agent.naive import RandomBot
from dlgo.httpfrontend.server import get_web_app

random_agent = RandomBot()
web_app = get_web_app({'random': random_agent})
web_app.run()
```

При выполнении этого кода на локальном хосте (127.0.0.1) будет запущено веб-приложение, прослушивающее порт 5000, который используется в приложениях Flask по умолчанию. Бот `RandomBot`, который мы только что зарегистрировали как `random`, соответствует HTML-файлу *play_random_99.html* в папке *static* модуля *httpfrontend*. Этот файл отвечает за визуализацию доски для игры в го и содержит определения правил игры бота с человеком. Человек начинает игру и играет черными камнями, а бот – белыми. После каждого хода игрока-человека отправляется запрос (route/selectmove/random) на получение следующего хода бота. После получения хода бота он применяется к доске, и очередь снова переходит к человеку. Чтобы сыграть против данного бота, откройте страницу **http://127.0.0.1:5000/static/play_random_99.html** в своем браузере. На рис. 8.2 показано то, что вы должны увидеть.

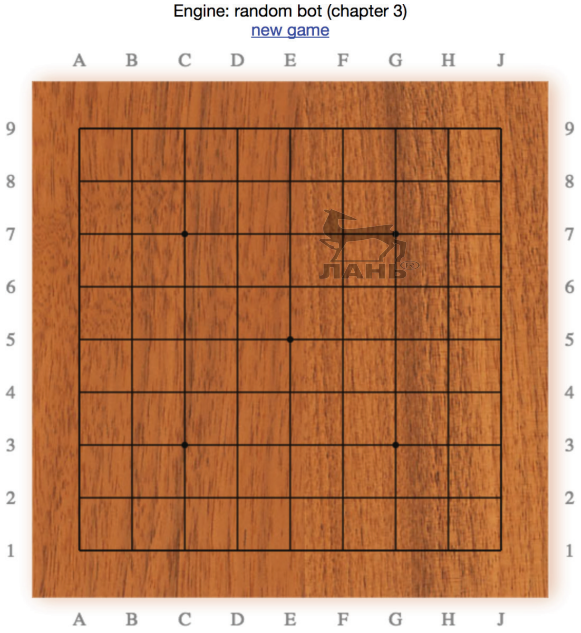


Рис. 8.2 ❖ Запуск веб-приложения Python для игры против бота в браузере

В следующих главах мы создадим множество других ботов, однако учтите, что в файле *play_predict_19.html* содержится еще один фронтенд, который взаимодействует с ботом под названием predict и может использоваться для игр на доске 19×19. Таким образом, при обучении модели Keras на основе данных игры го и использовании кодировщика доски `encoder` можно сначала создать экземпляр `agent = DeepLearningAgent(model, encoder)`, а затем зарегистрировать его в веб-приложении `web_app = get_web_app({'predict': agent})`, которое можно будет запустить с помощью метода `web_app.run()`.

8.2.1. Пример бота для игры в го

На рис. 8.3 проиллюстрирован весь процесс создания бота (этот же процесс был представлен в начале главы 7). Мы начинаем с импорта необходимых компонентов и загрузки признаков *X* и меток *y* с помощью кодировщика и обработчика данных игры го, как показано в листинге 8.8.

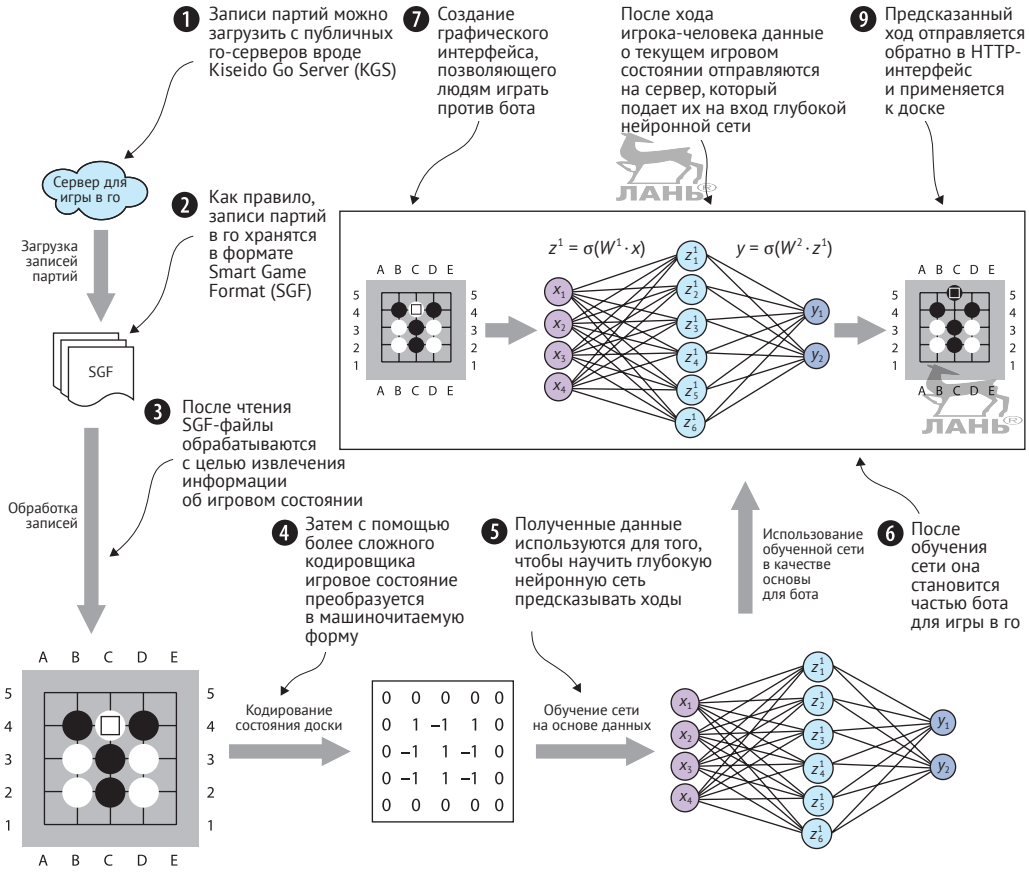


Рис. 8.3 ❖ Процесс обучения бота для игры в го

Листинг 8.8 ❖ Загрузка признаков и меток с помощью обработчика данных

```
import h5py

from keras.models import Sequential
from keras.layers import Dense

from dlgo.agent.predict import DeepLearningAgent, load_prediction_agent
from dlgo.data.parallel_processor import GoDataProcessor
from dlgo.encoders.sevenplane import SevenPlaneEncoder
from dlgo.httpfrontend import get_web_app
from dlgo.networks import large
```

```
go_board_rows, go_board_cols = 19, 19
nb_classes = go_board_rows * go_board_cols
encoder = SevenPlaneEncoder((go_board_rows, go_board_cols))
processor = GoDataProcessor(encoder=encoder.name())

X, y = processor.load_go_data(num_samples=100)
```

После подготовки признаков и меток мы можем создать глубокую сверточную нейронную сеть и обучить ее на их основе. На этот раз мы выберем большую сеть (*large.py*) из подмодуля *networks* модуля *dlgo* и используем оптимизатор Adadelta.

Листинг 8.9 ❖ Создание и запуск модели для предсказания ходов с использованием оптимизатора Adadelta

```
input_shape = (encoder.num_planes, go_board_rows, go_board_cols)
model = Sequential()
network_layers = large.layers(input_shape)
for layer in network_layers:
    model.add(layer)
model.add(Dense(nb_classes, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adadelta',
     metrics=['accuracy'])

model.fit(X, y, batch_size=128, epochs=20, verbose=1)
```

После обучения этой модели мы можем создать на ее основе бота для игры в го и сохранить его в формате HDF5.

Листинг 8.10 ❖ Создание и сохранение агента `DeepLearningAgent`

```
deep_learning_bot = DeepLearningAgent(model, encoder)
deep_learning_bot.serialize("../agents/deep_bot.h5")
```

Наконец, мы можем загрузить бота из файла и подключить его к веб-приложению.

Листинг 8.11 ❖ Загрузка бота обратно в память и его подключение к веб-приложению

```
model_file = h5py.File("../agents/deep_bot.h5", "r")
bot_from_file = load_prediction_agent(model_file)

web_app = get_web_app({'predict': bot_from_file})
web_app.run()
```

Если вы уже создали достаточно мощного бота, то можете пропустить все этапы, кроме последнего. Например, вы можете загрузить одну из моделей, состояние которой сохранили с помощью контрольных точек в главе 7, и проверить ее в действии, изменив соответствующим образом значение `model_file`.

8.3. Обучение и развертывание бота для игры в го в облаке

До этого момента вся разработка происходила на вашем локальном компьютере. Если на нем установлен современный графический процессор, то обучение глубоких нейронных сетей, созданных в главах с 5 по 7, не представляет для вас проблемы. В противном случае вам следует рассмотреть возможность *аренды графического процессора у облачного провайдера*.

Если вы не планируете проводить обучение в данный момент и считаете, что ваш бот уже является достаточно мощным, то можете воспользоваться услугами облачных провайдеров для публикации своего бота. В разделе 8.2 мы запускали бот через веб-приложение, размещенное на локальном хосте. Такое решение не является идеальным, если вы хотите поделиться своим ботом с другими людьми. Вам не следует ни предоставлять к своему компьютеру публичный доступ, ни заставлять его работать круглосуточно. Размещение бота на облачном сервисе позволяет отделить разработку от развертывания и просто поделиться URL-адресом со всеми, кто захочет сыграть против него.

Эта тема важна, но имеет очень малое отношение к машинному обучению, поэтому мы вынесли ее обсуждение в приложение Г. Ознакомление и применение методов, описанных в этом приложении, не является обязательным, но рекомендуется. В нем описан облачный провайдер Amazon Web Services (AWS) и освещаются следующие темы:

- создание учетной записи AWS;
- гибкая настройка, запуск и отключение экземпляров виртуальных серверов;
- создание экземпляра AWS для глубокого обучения модели на графическом процессоре, предоставляемом облачным провайдером за разумную цену;
- развертывание бота для игры в го на (практически) бесплатном сервере.

Помимо перечисленных выше тем, приложение Г содержит необходимую информацию, касающуюся развертывания полноценного игрового бота, взаимодействующего с онлайн-сервером для игры в го, о чем речь пойдет в разделе 8.6.

8.4. Взаимодействие с другими ботами по протоколу Go Text Protocol

В разделе 8.2 мы говорили об интеграции бота в веб-интерфейс. Для этого мы обеспечили взаимодействие между ботом и игроком-человеком с помощью протокола передачи гипертекста (HTTP), который является одним из главных протоколов, лежащих в основе Всемирной паутины. Для краткости мы не стали углубляться в детали, однако именно использование *стандартизированного протокола* позволило решить эту задачу. У людей и ботов нет общего языка для обмена ходами в игре го, но в качестве связующего звена между ними может выступать протокол.

Протокол Go Text Protocol (GTP) фактически представляет собой стандарт, используемый го-серверами по всему миру для обеспечения взаимодействия людей и ботов. Многие офлайн-программы для игры в го также основаны на данном протоколе. В этом разделе мы разберем принцип работы протокола GTP на примере. Мы реализуем его часть средствами языка Python, а затем используем эту реализацию для организации игры нашего бота с другими программами.

В приложении В описан процесс установки двух распространенных программ для игры в го GNU Go и Pachi, которые доступны практически для всех операционных систем. Мы настоятельно рекомендуем установить обе программы на свой компьютер. Вам не нужны никакие внешние интерфейсы, достаточно только инструментов командной строки. Если у вас установлена программа GNU Go, вы можете запустить ее в режиме GTP, выполнив следующую команду:

```
gnugo --mode gtp
```

Использование этого режима позволяет изучить принцип работы протокола GTP. GTP – это текстовый протокол, поэтому мы можем просто вводить команды в терминал и нажимать клавишу **Enter**. Например, для создания доски 9×9 достаточно ввести команду `boardsize 9`. GNU Go возвратит ответ и подтвердит правильность выполнения команды. Каждое успешное выполнение команды GTP вызывает ответ, начинающийся с символа `=`, а в случае неудачного выполнения используется символ `?`. Для проверки текущего состояния доски можно выполнить команду `showboard`, которая, как и следовало ожидать, выведет на экран пустую доску 9×9.

В реальном игровом процессе наиболее важными являются две команды: `genmove` и `play`. Команда `genmove` просит бота GTP сгенерировать следующий ход. Как правило, GTP-бот внутренне применяет этот ход к своему игровому состоянию. В качестве аргумента эта команда принимает только цвет игрока – (`black`) черный или (`white`) белый. Например, чтобы сгенерировать ход белых и применить его к доске GNU Go, введите команду `genmove white`. В результате будет получен ответ вида `= C4`, означающий, что GNU Go принимает эту команду (`=`) и помещает белый камень в точку C4. Как видите, бот GTP принимает стандартные координаты, представленные в главах 2 и 3.

Еще одной важной командой является `play`, которая дает боту GTP задание поместить камень на доску. Например, если вы хотите, чтобы бот поместил черный камень в точку D4, то можете сообщить ему об этом с помощью команды `play black D4`. Ответом будет символ `=`, свидетельствующий о том, что команда принята. Когда играют два бота, они по очереди просят друг друга сгенерировать ход с помощью команды `genmove`, а затем поместить на свою доску камень с помощью команды `play`. Для простоты мы опустили некоторые детали. Полноценный клиент GTP способен обрабатывать множество других команд, имеющих отношение к камням гандикапа, учету времени, подсчету очков и т. д. Если вас интересуют подробности, обратитесь к странице **mng.bz/MWNQ**. Однако имейте в виду, что для организации игры ваших ботов с программами GNU Go и Pachi на базовом уровне будет достаточно команд `genmove` и `play`.

Чтобы задействовать протокол GTP и позволить агенту использовать его для обмена игровыми данными, создайте новый подмодуль *dlgo* с именем *gtp*. Вы можете по-прежнему создавать свою реализацию, следуя приведенным в книге листингам, однако начиная с этой главы мы рекомендуем использовать готовый код из репозитория GitHub (**mng.bz/a4Wj**).

Для начала давайте разберемся, что собой представляет команда GTP. Следует отметить, что на многих го-серверах командам присваивается порядковый номер, позволяющий сопоставлять их с ответами. Эти номера не являются обязательными и могут иметь значение `None`. В нашем случае GTP-команда состоит из порядкового номера, команды и некоторого количества аргументов. Поместите следующее определение в файл *command.py* в подмодуле *gtp*.

Листинг 8.12 ❖ Реализация GTP-команды на языке Python

```
class Command:

    def __init__(self, sequence, name, args):
        self.sequence = sequence
        self.name = name
```

```
        self.args = tuple(args)

    def __eq__(self, other):
        return self.sequence == other.sequence and \
            self.name == other.name and \
            self.args == other.args

    def __repr__(self):
        return 'Command(%r, %r, %r)' % (self.sequence, self.name, self.args)

    def __str__(self):
        return repr(self)
```

Теперь нам нужно проанализировать или разобрать текст, введенный в командную строку, и преобразовать его в `Command`. Например, в результате разбора «999 play white D4» должно получиться `Command(999, 'play', ('white', 'D4'))`. Используемую для этого функцию `parse` также следует поместить в файл *command.py*.

Листинг 8.13 ❖ Преобразование простого текста в GTP-команду

```
def parse(command_string):
    pieces = command_string.split()
    try:
        sequence = int(pieces[0])
        pieces = pieces[1:]
    except ValueError:
        sequence = None
    name, args = pieces[0], pieces[1:]
    return Command(sequence, name, args)
```

GTP-команды могут начинаться с порядкового номера.

Если первый фрагмент не является числовым, порядковый номер отсутствует.

Выше было сказано, что для указания координат GTP используется стандартная запись, поэтому преобразование координат GTP в позиции на доске и наоборот не должно вызывать сложностей. Для этого определим две вспомогательные функции в файле *board.py* подмодуля *gtp*.

Листинг 8.14 ❖ Преобразование координат GTP в данные типа `Point`

```
from dlgo.gotypes import Point
from dlgo.goboard_fast import Move

def coords_to_gtp_position(move):
    point = move.point
    return COLS[point.col - 1] + str(point.row)

def gtp_position_to_coords(gtp_position):
    col_str, row_str = gtp_position[0], gtp_position[1:]
    point = Point(int(row_str), COLS.find(col_str.upper()) + 1)
    return Move(point)
```

8.5. Локальные игры против других ботов

Теперь, когда мы разобрались с основами протокола GTP, давайте создадим программу, которая загружает один из наших ботов и позволяет ему играть против GNU Go или Pachi. Однако прежде чем переходить к этой программе, нам нужно решить одну техническую задачу, чтобы наш бот знал, когда ему следует пропустить ход или выйти из игры.

8.5.1. Пропуск хода и выход из игры

На данном этапе наши боты не способны определить момент, когда игру стоит прекратить. Они созданы таким образом, чтобы всегда выбирать наилучший ход. Такая стратегия может оказаться бесполезной в конце игры, когда лучшим вариантом бывает пропуск хода или даже выход из игры, если ситуация оказывается безвыходной. Чтобы явно сообщить боту о необходимости остановиться, мы будем использовать *стратегии прекращения игры*. В главах 13 и 14 описаны мощные методы, благодаря которым от этих стратегий можно будет отказаться (наш бот научится оценивать текущую ситуацию на доске и определять момент, когда следует остановиться). Но сейчас эта концепция нам пригодится для организации игры бота против других ботов.

В файле *termination.py* в подмодуле *agent* модуля *dlgo* мы создадим стратегию `TerminationStrategy`, позволяющую боту определить момент, когда следует пропустить ход или выйти из игры (по умолчанию он никогда этого не делает).

Листинг 8.15 ❖ Стратегия прекращения игры сообщает боту, когда ему следует остановиться

```
from dlgo import goboard
from dlgo.agent.base import Agent
from dlgo import scoring

class TerminationStrategy:

    def __init__(self):
        pass

    def should_pass(self, game_state):
        return False

    def should_resign(self, game_state):
        return False
```


Простая эвристика для прекращения игры заключается в том, чтобы пропускать ход тогда, когда это делает противник. В этом случае нам приходится полагаться на то, что противник знает, когда следует пропустить ход, но этот прием является хорошей отправной точкой и работает в игре против ботов GNU Go и Pachi.

Листинг 8.16 ❖ Пропуск хода в ответ на пропуск хода противником

```
class PassWhenOpponentPasses(TerminationStrategy):

    def should_pass(self, game_state):
        if game_state.last_move is not None:
            return True if game_state.last_move.is_pass else False

def get(termination):
    if termination == 'opponent_passes':
        return PassWhenOpponentPasses()
    else:
        raise ValueError("Unsupported termination strategy: {}"
                         .format(termination))
```

Файл *termination.py* также содержит еще одну стратегию под названием `ResignLargeMargin`, которая заставляет бота выйти из игры, если противник сильно лидирует в счете. Вы можете придумать множество других стратегий, однако

помните о том, что машинное обучение, в конце концов, позволяет избавиться от этих костылей.

Последнее, что нам необходимо сделать для организации игры ботов друг против друга, – это снабдить агента стратегией прекращения игры, чтобы бот мог пропустить ход или выйти из игры, когда это уместно. Добавьте следующий класс `TerminationAgent` в файл *termination.py*.

Листинг 8.17 ❖ Снабжение агента стратегией завершения игры

```
class TerminationAgent(Agent):

    def __init__(self, agent, strategy=None):
        Agent.__init__(self)
        self.agent = agent
        self.strategy = strategy if strategy is not None \
            else TerminationStrategy()

    def select_move(self, game_state):
        if self.strategy.should_pass(game_state):
            return goboard.Move.pass_turn()
        elif self.strategy.should_resign(game_state):
            return goboard.Move.resign()
        else:
            return self.agent.select_move(game_state)
```

8.5.2. Запуск игры бота против других ботов

Теперь, когда мы обсудили стратегии прекращения игры, можно приступать к организации игры нашего бота с другими программами. В файле *play_local.py* в модуле *gtp* вы найдете сценарий, позволяющий вашему боту играть против GNU Go или Pachi. Давайте подробно разберем этот сценарий, начиная с импорта необходимых компонентов.

Листинг 8.18 ❖ Импорт компонентов, необходимых для запуска локальных игр

```
import subprocess
import re
import h5py

from dlgo.agent.predict import load_prediction_agent
from dlgo.agent.termination import PassWhenOpponentPasses, TerminationAgent
from dlgo.goboard_fast import GameState, Move
from dlgo.gotypes import Player
from dlgo.gtp.board import gtp_position_to_coords, coords_to_gtp_position
from dlgo.gtp.utils import SGFWriter
from dlgo.utils import print_board
from dlgo.scoring import compute_game_result
```

Вам уже должны быть знакомы все импортируемые компоненты, за исключением `SGFWriter`. Это небольшой служебный класс из файла *utils.py* (находится в подмодуле *gtp* модуля *dlgo*), который отслеживает ход игры и в конце записывает в файл SGF.

Чтобы инициализировать утилиту для запуска игры `LocalGtpBot`, необходимо предоставить агента глубокого изучения и стратегию прекращения игры (необя-

зательно). Кроме того, можно указать количество камней гандикапа и название бота-противника, например gnugo или pachi. Утилита LocalGtpBot инициализирует одну из этих программ в качестве подпроцессов, а обмен данными между ботами будет осуществляться по протоколу GTP.

Листинг 8.19 ❖ Инициализация утилиты для запуска игры двух ботов

```
class LocalGtpBot:

    def __init__(self, go_bot, termination=None, handicap=0,
                 opponent='gnugo', output_sgf="out.sgf",
                 our_color='b'):
        self.bot = TerminationAgent(go_bot, termination)
        self.handicap = handicap
        self._stopped = False
        self.game_state = GameState.new_game(19)
        self.sgf = SGFWriter(output_sgf)

        self.our_color = Player.black if our_color == 'b' else Player.white
        self.their_color = self.our_color.other

        cmd = self.opponent_cmd(opponent)
        pipe = subprocess.PIPE
        self.gtp_stream = subprocess.Popen(
            cmd, stdin=pipe, stdout=pipe
        )

    @staticmethod
    def opponent_cmd(opponent):
        if opponent == 'gnugo':
            return ["gnugo", "--mode", "gtp"]
        elif opponent == 'pachi':
            return ["pachi"]
        else:
            raise ValueError("Unknown bot name {}".format(opponent))
```

Инициализация бота с использованием агента и стратегии прекращения игры.

Игра продолжается до остановки одного из игроков.

В конце партия записывается в файл в формате SGF.

В качестве противника нашего бота может выступать GNU Go или Pachi.

Чтение и запись GTP-команд из командной строки.

Одним из главных методов, используемых описанным здесь инструментом, является command_and_response, который отправляет команду GTP и считывает ответ на нее.

Листинг 8.20 ❖ Отправка GTP-команды и получение ответа

```
    def send_command(self, cmd):
        self.gtp_stream.stdin.write(cmd.encode('utf-8'))

    def get_response(self):
        succeeded = False
        result = ''
        while not succeeded:
            line = self.gtp_stream.stdout.readline()
            if line[0] == '=':
                succeeded = True
                line = line.strip()
                result = re.sub('^= ?', '', line)
        return result
```

```
def command_and_response(self, cmd):
    self.send_command(cmd)
    return self.get_response()
```

Процесс игры состоит из следующих этапов:

1) настройка доски с помощью команды GTP `boardize`. В данном случае мы ограничиваемся доской 19×19, поскольку именно к такому размеру доски приспособлены наши боты;
2) гандикап задается с помощью метода `set_handicap`;
3) за саму игру отвечает метод `play`;
4) запись партии сохраняется в файле SGF.

Листинг 8.21 ❖ Настройка доски, игра и сохранение записи партии

```
def run(self):
    self.command_and_response("boardsize 19\n")
    self.set_handicap()
    self.play()
    self.sgf.write_sgf()

def set_handicap(self):
    if self.handicap == 0:
        self.command_and_response("komi 7.5\n")
        self.sgf.append("KM[7.5]\n")
    else:
        stones = self.command_and_response("fixed_handicap
 {}\n".format(self.handicap))
        sgf_handicap = "HA[{}]AB".format(self.handicap)
        for pos in stones.split(" "):
            move = gtp_position_to_coords(pos)
            self.game_state = self.game_state.apply_move(move)
            sgf_handicap = sgf_handicap + "[" +
 self.sgf.coordinates(move) + "]"
        self.sgf.append(sgf_handicap + "\n")
```

Логика данного игрового процесса довольно проста: пока один из противников не остановится, следует поочередно совершать ходы. Для этого боты используют методы `play_our_move` и `play_their_move` соответственно. Кроме того, мы очищаем экран, а затем отображаем текущее состояние доски и приблизительную оценку результата.

Листинг 8.22 ❖ Игра заканчивается, когда один из противников подает соответствующий сигнал

```
def play(self):
    while not self._stopped:
        if self.game_state.next_player == self.our_color:
            self.play_our_move()
        else:
            self.play_their_move()
        print(chr(27) + "[2J")
        print_board(self.game_state.board)
        print("Estimated result: ")
        print(compute_game_result(self.game_state))
```

Игра нашего бота сводится к генерированию хода с помощью метода select_move, применению его к доске, а также преобразованию и отправке этого хода по протоколу GTP. При этом пропуск хода и выход из игры должны обрабатываться особым образом.

Листинг 8.23 ❖ Команды, заставляющие бота сгенерировать ход, применить его к доске и передать по протоколу GTP

```
    def play_our_move(self):
        move = self.bot.select_move(self.game_state)
        self.game_state = self.game_state.apply_move(move)

        our_name = self.our_color.name
        our_letter = our_name[0].upper()
        sgf_move = ""
        if move.is_pass:
            self.command_and_response("play {} pass\n".format(our_name))
        elif move.is_resign:
            self.command_and_response("play {} resign\n".format(our_name))
        else:
            pos = coords_to_gtp_position(move)
            self.command_and_response("play {} {}\n".format(our_name, pos))
            sgf_move = self.sgf.coordinates(move)
        self.sgf.append(";{}[{}]\n".format(our_letter, sgf_move))
```

Игра бота-противника подчиняется той же логике. Мы просим бота GNU Go или Pachi сгенерировать ход (genmove), а затем преобразуем GTP-ответ в ход, понятный для нашего бота. Единственное, что нам нужно сделать, – это остановить игру в случае пропуска хода обоими ботами или выхода нашего противника из игры.

Листинг 8.24 ❖ Наш противник совершает ходы в ответ на команду genmove

```
    def play_their_move(self):
        their_name = self.their_color.name
        their_letter = their_name[0].upper()

        pos = self.command_and_response("genmove {}\n".format(their_name))
        if pos.lower() == 'resign':
            self.game_state = self.game_state.apply_move(Move.resign())
            self._stopped = True
        elif pos.lower() == 'pass':
            self.game_state = self.game_state.apply_move(Move.pass_turn())
            self.sgf.append(";{}[]\n".format(their_letter))
            if self.game_state.last_move.is_pass:
                self._stopped = True
        else:
            move = gtp_position_to_coords(pos)
            self.game_state = self.game_state.apply_move(move)
            self.sgf.append(";{}[{}]\n".format(their_letter, self.sgf.coordinates(move)))
```

Реализация *play_local.py* завершена, и теперь мы можем ее протестировать.

Листинг 8.25 ❖ Запуск игры нашего бота против программы Pachi

```
from dlgo.gtp.play_local import LocalGtpBot
from dlgo.agent.termination import PassWhenOpponentPasses
```

```
from dlgo.agent.predict import load_prediction_agent
import h5py

bot = load_prediction_agent(h5py.File("../agents/betago.hdf5", "r"))

gtp_bot = LocalGtpBot(go_bot=bot, termination=PassWhenOpponentPasses(),
                      handicap=0, opponent='pachi')
gtp_bot.run()
```

На рис. 8.4 показано то, что должно появиться на экране в процессе игры между двумя ботами.

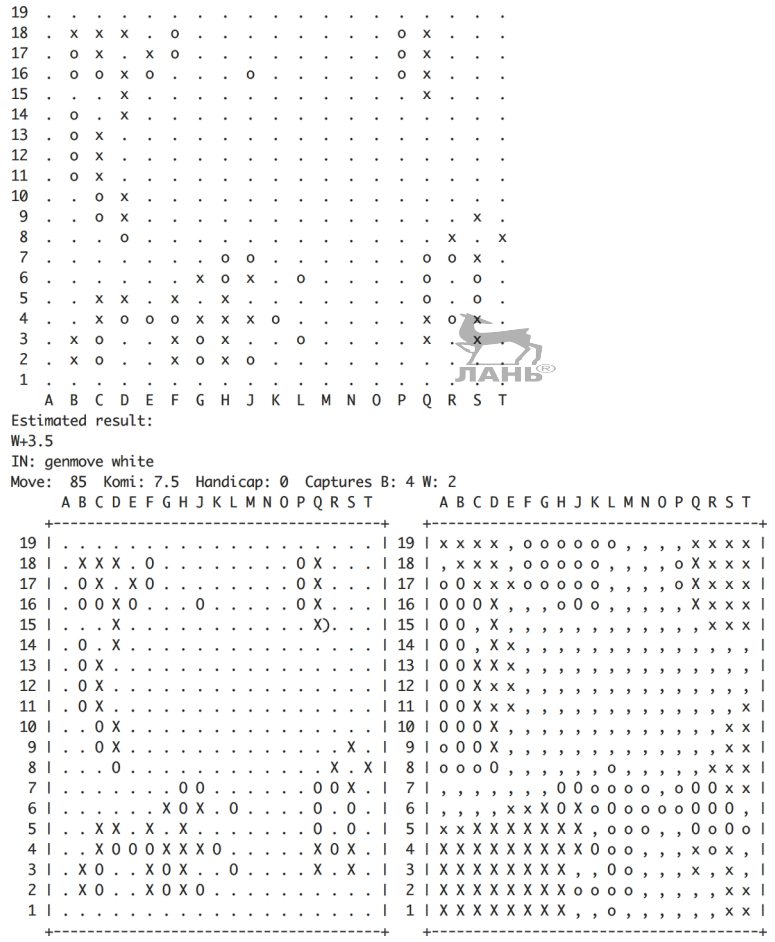


Рис. 8.4 ❖ Пример того, как Pachi и наш бот воспринимают и оценивают свою игру

В верхней части рисунка показано состояние нашей доски и наша текущая оценка результата. В нижней части слева показано игровое состояние бота Pachi (идентичное нашему), а справа – его оценка территории доски, принадлежащей каждому из игроков.

Данная демонстрация возможностей нашего бота является достаточно впечатляющей, однако это еще далеко не все, на что он способен. В следующем разделе мы сделаем еще один шаг и подключим нашего бота к реальному го-серверу.

8.6. Развертывание бота для игры в го на онлайн-сервере

Файл *play_local.py*, по сути, представляет собой небольшой го-сервер для игры двух ботов-противников. Он принимает и отправляет команды GTP и знает, когда следует начинать и заканчивать игру. Это создает дополнительную нагрузку, поскольку программа берет на себя роль рефери, который контролирует взаимодействие противников.

Если подключить бота к реальному го-серверу, он позаботится обо всей логике игрового процесса и позволит вам сосредоточиться на отправке и получении команд GTP. С одной стороны, ваша жизнь облегчится, поскольку сервер избавит вас от лишних забот. С другой – подключение к го-серверу потребует обеспечения поддержки всего диапазона GTP-команд, предусмотренных этим сервером. В противном случае ваш бот может перестать работать должным образом.

Чтобы этого не произошло, давайте сделаем процесс обработки GTP-команд еще более формальным. Сначала мы реализуем класс ответа GTP на случай удачного и неудачного выполнения команд.

Листинг 8.26 ❖ Кодирование и сериализация GTP-ответа

```
class Response:
    def __init__(self, status, body):
        self.success = status
        self.body = body

def success(body=''):
    return Response(status=True, body=body)

def error(body=''):
    return Response(status=False, body=body)

def bool_response(boolean):
    return success('true') if boolean is True else success('false')

def serialize(gtp_command, gtp_response):
    return '{}{} {}\n\n'.format(
        '=' if gtp_response.success else '?',
        '' if gtp_command.sequence is None else str(gtp_command.sequence),
        gtp_response.body
    )
```

Создание GTP-ответа с телом ответа на случай удачного выполнения команды.

Создание GTP-ответа на случай неудачного выполнения команды.

Преобразование логического значения Python в GTP.

Сериализация GTP-ответа в строку.

Теперь нам осталось реализовать класс `GTPFrontend`, который мы поместим в файл *frontend.py* в подмодуле *gtp*. Для этого необходимо импортировать из подмодуля *gtp* следующие компоненты, включая `command` и `response`.

Листинг 8.27 ❖ Импорт компонентов для GTP-интерфейса

```
import sys

from dlgo.gtp import command, response
from dlgo.gtp.board import gtp_position_to_coords, coords_to_gtp_position
from dlgo.goboard_fast import GameState, Move
from dlgo.agent.termination import TerminationAgent
from dlgo.utils import print_board
```

Для инициализации GTP-интерфейса нужно указать экземпляр агента `Agent` и (опционально) стратегию прекращения игры. Затем класс `GTPFrontend` создаст словарь GTP-событий, которые нам предстоит обрабатывать. Каждое из этих событий, включая общие команды вроде `play`, придется реализовать вам.

Листинг 8.28 ❖ Инициализация GTP-интерфейса, определяющего обработчики GTP-событий

```
HANDICAP_STONES = {
    2: ['D4', 'Q16'],
    3: ['D4', 'Q16', 'D16'],
    4: ['D4', 'Q16', 'D16', 'Q4'],
    5: ['D4', 'Q16', 'D16', 'Q4', 'K10'],
    6: ['D4', 'Q16', 'D16', 'Q4', 'D10', 'Q10'],
    7: ['D4', 'Q16', 'D16', 'Q4', 'D10', 'Q10', 'K10'],
    8: ['D4', 'Q16', 'D16', 'Q4', 'D10', 'Q10', 'K4', 'K16'],
    9: ['D4', 'Q16', 'D16', 'Q4', 'D10', 'Q10', 'K4', 'K16', 'K10'],
}

class GTPFrontend:

    def __init__(self, termination_agent, termination=None):
        self.agent = termination_agent
        self.game_state = GameState.new_game(19)
        self._input = sys.stdin
        self._output = sys.stdout
        self._stopped = False

        self.handlers = {
            'boardsize': self.handle_boardsize,
            'clear_board': self.handle_clear_board,
            'fixed_handicap': self.handle_fixed_handicap,
            'genmove': self.handle_genmove,
            'known_command': self.handle_known_command,
            'komi': self.ignore,
            'showboard': self.handle_showboard,
            'time_settings': self.ignore,
            'time_left': self.ignore,
            'play': self.handle_play,
            'protocol_version': self.handle_protocol_version,
            'quit': self.handle_quit,
        }
```

После запуска игры с помощью метода `run` мы последовательно считываем GTP-команды, которые пересылаются соответствующему обработчику событий с помощью метода `process`.

Листинг 8.29 ❖ Интерфейс разбирает поток входных данных вплоть до завершения игры

```
def run(self):
    while not self._stopped:
        input_line = self._input.readline().strip()
        cmd = command.parse(input_line)
        resp = self.process(cmd)
        self._output.write(response.serialize(cmd, resp))
        self._output.flush()

def process(self, cmd):
    handler = self.handlers.get(cmd.name, self.handle_unknown)
    return handler(*cmd.args)
```

Для завершения интерфейса осталось реализовать отдельные GTP-команды. Три самые важные из них показаны в следующем листинге, а остальные вы можете найти в репозитории GitHub.

Листинг 8.30 ❖ Некоторые из наиболее важных ответов на события для GTP-интерфейса

```
def handle_play(self, color, move):
    if move.lower() == 'pass':
        self.game_state = self.game_state.apply_move(Move.pass_turn())
    elif move.lower() == 'resign':
        self.game_state = self.game_state.apply_move(Move.resign())
    else:
        self.game_state =
 self.game_state.apply_move(gtp_position_to_coords(move))
    return response.success()

def handle_genmove(self, color):
    move = self.agent.select_move(self.game_state)
    self.game_state = self.game_state.apply_move(move)
    if move.is_pass:
        return response.success('pass')
    if move.is_resign:
        return response.success('resign')
    return response.success(coords_to_gtp_position(move))

def handle_fixed_handicap(self, nstones):
    nstones = int(nstones)
    for stone in HANDICAP_STONES[nstones]:
        self.game_state = self.game_state.apply_move(
            gtp_position_to_coords(stone))
    return response.success()
```

Теперь мы можем использовать этот GTP-интерфейс в рамках небольшого сценария, позволяющего запустить его из командной строки.

Листинг 8.31 ❖ Запуск GTP-интерфейса из командной строки

```
from dlgo.gtp import GTPFrontend
from dlgo.agent.predict import load_prediction_agent
from dlgo.agent import termination
import h5py
```

```
model_file = h5py.File("agents/betago.hdf5", "r")
agent = load_prediction_agent(model_file)
strategy = termination.get("opponent_passes")
termination_agent = termination.TerminationAgent(agent, strategy)

frontend = GTPFrontend(termination_agent)
frontend.run()
```

После запуска этой программы мы сможем использовать ее так же, как в случае тестирования бота GNU Go в разделе 8.4: мы будем передавать ей GTP-команды, а она будет их обрабатывать. Для ее тестирования попробуйте сгенерировать ход с помощью команды `genmove` или вывести на экран состояние доски с помощью метода `showboard`. Вы можете использовать любую команду, предусмотренную обработчиком событий в `GTPFrontend`.

8.6.1. Регистрация бота на онлайн-сервере для игры в го

Теперь, когда GTP-интерфейс готов и работает локально так же, как GNU Go и Pachi, вы можете зарегистрировать своего бота на онлайн-платформе, использующей протокол GTP. Большинство популярнейших го-серверов основано на этом протоколе, и в приложении В вы найдете описание трех таких платформ. Одним из самых популярных серверов в Европе и Северной Америке является Online Go Server (OGS). Мы выбрали его в качестве платформы, чтобы продемонстрировать процесс запуска бота, однако вы можете использовать для этого большинство других серверов.

Поскольку процесс регистрации бота на сервере OGS является довольно сложным, а программа, отвечающая за подключение бота к OGS, написана на языке JavaScript, мы вынесли обсуждение этой темы в приложение Д. Вы можете прочитать это приложение сейчас или пропустить его изучение, если не намерены запускать бота на онлайн-сервере. Изучение приложения Д позволит вам овладеть следующими навыками:

- создание двух учетных записей на сервере OGS. Одна из них предназначена для вашего бота, а другая – для администрирования его учетной записи;
- подключение бота к серверу OGS с локального компьютера с целью тестирования;
- развертывание бота на экземпляре AWS для обеспечения его длительного подключения к серверу OGS.

Это позволит вам участвовать в (ранжированной) игре против собственного бота онлайн. Любой обладатель учетной записи OGS тоже сможет сыграть с вашим ботом. Кроме того, ваш бот сможет принимать участие в турнирах, проводимых на сервере OGS!

8.7. Резюме

- Встраивание глубокой сети в структуру агента позволяет обеспечить взаимодействие моделей с их средой.
- Внедрение агента в веб-приложении путем создания HTTP-фронтенда позволяет вам играть против собственных ботов, используя графический интерфейс.

- Вы можете арендовать графический процессор у облачного провайдера вроде AWS для эффективного проведения экспериментов, связанных с глубоким обучением.
- Развертывание веб-приложения на платформе AWS позволяет вам легко предоставить доступ к своему боту другим людям.
- Позволив боту отправлять и принимать команды GTP, вы сможете запустить его локальную игру против других программ для игры в го.
- Создание GTP-интерфейса для бота является наиболее важным условием для его регистрации на онлайн-платформе для игры в го.
- Развертывание бота в облаке позволяет ему участвовать в обычных играх и турнирах, проводимых на го-сервере OGS. Кроме того, это дает вам возможность играть против него в любое время.

Глава 9

Обучение на практике: обучение с подкреплением

В этой главе:

- формулирование задачи обучения с подкреплением;
- создание обучающегося игрового агента;
- набор опыта игры против самого себя для дальнейшего обучения.

Я прочитал около десятка книг, посвященных игре в го и написанных профессиональными игроками из Китая, Кореи и Японии. Однако я всего лишь любитель среднего уровня. Почему мне не удалось сравняться с этими легендарными игроками? Я не думаю, что забыл их рекомендации. Книгу Тосиро Кагэямы «Lessons in the Fundamentals of Go» (Ishi Press, 1978) я знаю практически наизусть. Может быть, мне просто следует прочитать еще больше книг...

Я не знаю, как стать лучшим игроком в го, однако мне известно, по крайней мере, одно различие между мной и профессионалами: практика. Чтобы стать профессионалом, человеку требуется сыграть около пяти, а то и десяти тысяч партий. Практика порождает знание, которое иногда невозможно передать другим. Вы можете *обобщить* это знание, что и делается в книгах об игре го. Однако при этом упускаются многие нюансы. Если я хочу по-настоящему усвоить то, о чем прочитал, то мне придется как следует попрактиковаться.

Практика очень важна для людей, а как насчет компьютеров? Может ли компьютерная программа учиться на практике? Метод *обучения с подкреплением* обещает именно это. При обучении с подкреплением (ОП) программа совершенствуется, многократно решая одну и ту же задачу. Когда она достигает хороших результатов, мы модифицируем программу для воспроизведения принятых ею решений. А когда результаты оказываются плохими, мы меняем программу так, чтобы она избегала принятия таких решений в дальнейшем. Это не означает, что нам приходится писать новый код после каждого испытания: алгоритмы ОП предусматривают автоматизированные методы для внесения соответствующих изменений.

Обучение с подкреплением имеет и свои издержки. Во-первых, это медленный процесс: боту придется сыграть тысячи игр, чтобы продемонстрировать существенные улучшения. Кроме того, процесс обучения сложен для отладки. Однако если вы освоите эти методы, ваши вложения многократно окупятся. Вы сможете создавать программы, применяющие сложные стратегии для решения различных задач, даже если эти стратегии не будут понятны вам самим.

Эта глава начинается с общего обзора цикла обучения с подкреплением. Далее мы поговорим об организации игры бота с самим собой. А в главе 10 покажем, как использовать данные, полученные в результате игры бота с самим собой, для улучшения его производительности.

9.1. Цикл обучения с подкреплением

Механику обучения с подкреплением реализуют многие алгоритмы, однако все они работают в рамках стандартной структуры. В этом разделе описан цикл обучения с подкреплением, позволяющий компьютерной программе совершенствоваться с помощью многократных попыток решения одной и той же задачи. Данный цикл проиллюстрирован на рис. 9.1.

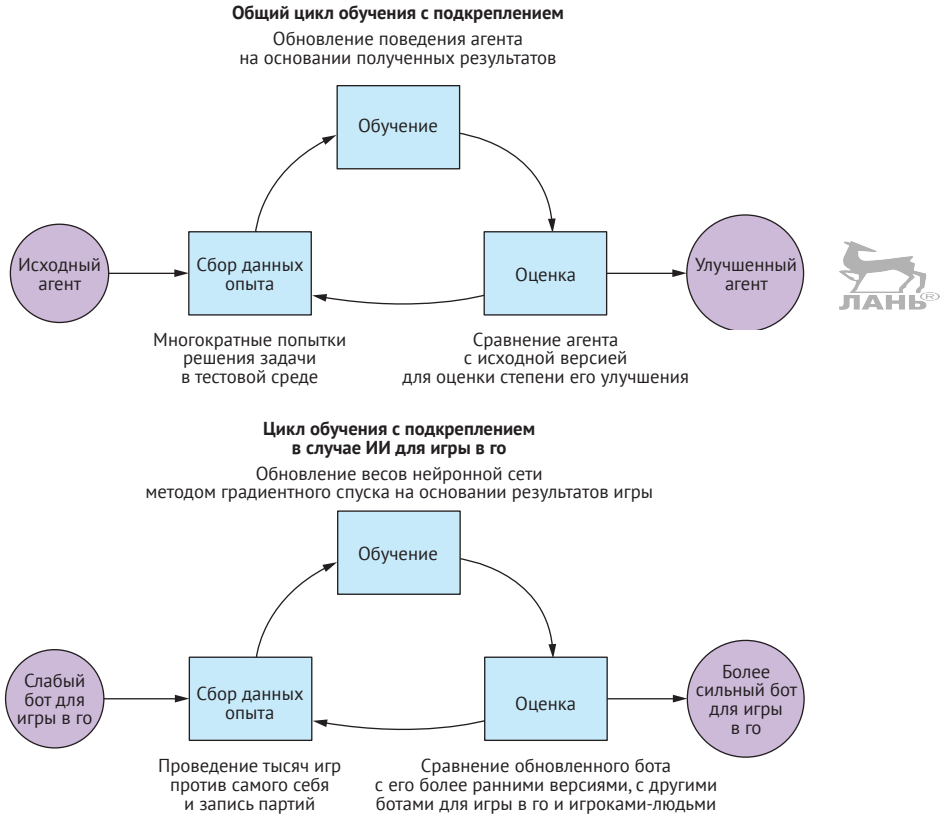


Рис. 9.1 ❖ Цикл обучения с подкреплением. Обучение с подкреплением можно реализовать разными способами, однако сам процесс имеет стандартную структуру. Сначала компьютерная программа предпринимает множество попыток решить задачу. Записи об этих попытках называются *данными опыта*. Затем мы изменяем поведение программы, чтобы имитировать наиболее успешные попытки; этот процесс называется *обучением*. Затем мы периодически оцениваем производительность программы, чтобы убедиться в том, что она повышается. Как правило, данный процесс многократно повторяется

В терминах обучения с подкреплением наш бот для игры в го называется *агентом* – программой, которая принимает решения для выполнения некоторой задачи. Ранее в книге мы реализовали несколько версий класса `Agent`, способного выбирать ходы в игре го. Мы предоставляли агенту данные о ситуации – объект `GameState`, а в ответ он предоставлял выбранный ход. И хотя в этих случаях мы не применяли обучение с подкреплением, в них использовалась та же самая концепция агента.

Цель обучения с подкреплением заключается в том, чтобы сделать агента максимально эффективным. В данном случае мы хотим, чтобы наш бот победил в игре го.

Сначала мы позволяем боту сыграть несколько игр с самим собой. В процессе каждой такой игры он должен записывать все ходы и итоговый результат. Эти записи партий называются *опытом* бота.

Затем мы *обучаем* нашего бота, обновляя его поведение на основании данных, полученных в ходе партий, сыгранных ботом против самого себя. Этот процесс напоминает обучение нейронных сетей, о котором говорилось в главах 6 и 7. Основная идея заключается в том, чтобы заставить бота повторять решения, которые он принимал в выигранных им партиях, и отказаться от решений, принятых им в играх, закончившихся его поражением. Алгоритм обучения связан со структурой агента: нам необходимо иметь возможность систематически изменять поведение агента, чтобы обеспечить его обучение. Для этого существует множество алгоритмов, и далее в книге мы обсудим три из них. В этой и следующей главах мы поговорим об алгоритме *градиента политики*. В главе 11 мы рассмотрим алгоритм *Q-обучения*, а в главе 12 обсудим алгоритм типа «*актор–критик*».

Ожидается, что после обучения наш бот станет немного сильнее. Однако существует множество причин, по которым учебный процесс может пойти не так, поэтому нам не помешает способ оценки прогресса нашего бота. Чтобы оценить производительность игрового агента, увеличьте количество сыгранных им партий. В качестве противника можно выставить предыдущие версии агента, чтобы оценить его прогресс. А в целях проверки его работоспособности вы можете периодически сравнивать бота с другими ботами или играть против него самостоятельно.

После этого можно будет до бесконечности повторять цикл «сбор данных опыта, обучение и оценка».

Мы разобьем этот цикл на несколько сценариев. В данной главе реализуем сценарий `self_play`, который будет симулировать игры бота с самим собой и сохранять данные опыта на диск. В следующей главе мы создадим сценарий `train`, который принимает полученный ботом опыт в качестве входных данных, обновляет агента и сохраняет его новую версию.

9.2. Данные опыта

В главе 3 мы разработали набор структур для представления данных игры го. Используя такие классы, как `Move`, `GoBoard` и `GameState`, можно сохранить запись целой игровой партии. Однако алгоритмы обучения с подкреплением имеют дело с высоко абстрактным представлением проблемы, поэтому одни и те же алгоритмы могут применяться для решения задач в различных областях. В этом разделе мы поговорим об описании партий в терминах обучения с подкреплением.

Игровой опыт можно разделить на отдельные партии, или *эпизоды*. Каждый эпизод имеет четкое окончание, а решения, принятые в ходе одного из эпизодов, никак не влияют на то, что происходит в следующем. В других областях возможности четкого деления опыта на эпизоды может не быть. Например, решения, принимаемые роботом, предназначенным для непрерывной работы, представляют собой бесконечную последовательность. Мы, конечно, можем применить к нему методы обучения с подкреплением, однако отсутствие четких границ между эпизодами усложняет задачу.

В рамках эпизода агент имеет дело с *состоянием* своей среды, исходя из которого он должен выбрать *действие*. После выбора действия агент сталкивается с новым состоянием, которое зависит от выбранного агентом действия и других событий, произошедших в этой среде. В случае игры го бот будет рассматривать позицию на доске (состояние), а затем выбирать допустимый ход (действие). Когда очередь хода опять перейдет к нему, бот будет рассматривать уже новое состояние доски.

Учтите, что после того, как агент выберет действие, следующее состояние также будет учитывать ход его противника. Вы не можете определить следующее состояние на основании текущего состояния и выбранного вами действия, вам необходимо дождаться хода противника. Поведение противника является фактором *среды*, в которой агент должен научиться ориентироваться.

Для совершенствования агенту требуется обратная связь, позволяющая ему судить о степени достижения цели. Мы предоставляем эту обратную связь с помощью выраженной численно *награды*. В случае ИИ для игры в го целью является победа в игре, поэтому мы присуждаем боту награду, равную 1, после каждой победы и –1 при каждом поражении. Алгоритмы обучения с подкреплением будут изменять поведение агента с целью увеличения совокупной награды. На рис. 9.2 показано, как можно описать игру в го с помощью состояний, действий и наград.

Игра го и другие подобные игры представляют собой особый случай, когда награда выдается агенту целиком в конце игры. Кроме того, есть только два возможных варианта награды: вы либо выигрываете, либо проигрываете, и вам нет дела до всех остальных событий в игре. В других областях награда может быть распределена. Представьте, что мы создаем бота для игры «Эрудит». На каждом ходу он будет добавлять слово и подсчитывать очки, после чего то же самое будет делать его противник. В этом случае мы можем вычислить награду бота путем вычитания из его счета количества очков, набранных противником. Тогда боту нет необходимости дожидаться конца эпизода для получения награды. Вместо этого он будет получать ее частями после каждого своего действия.

Ключевая идея обучения с подкреплением состоит в том, что выбранное действие может повлиять на полученную в дальнейшем награду. Представьте, что агент принял особенно удачное решение на 35-м ходу и выиграл партию на 200-м. Хороший ход на ранних этапах игры внес определенный вклад в победу. Нам нужен способ распределения награды среди всех ходов. Вознаграждение, получаемое агентом после совершения действия, называется *выгодой* от этого действия. Чтобы вычислить выгоду от действия, мы складываем все вознаграждения, полученные агентом после совершенного действия, вплоть до конца эпизода, как показано в листинге 9.1. Это говорит о том, что мы заранее не знаем, какие ходы ответственны за победу или поражение. Бремя разделения ответственности между отдельными ходами лежит на алгоритме обучения.

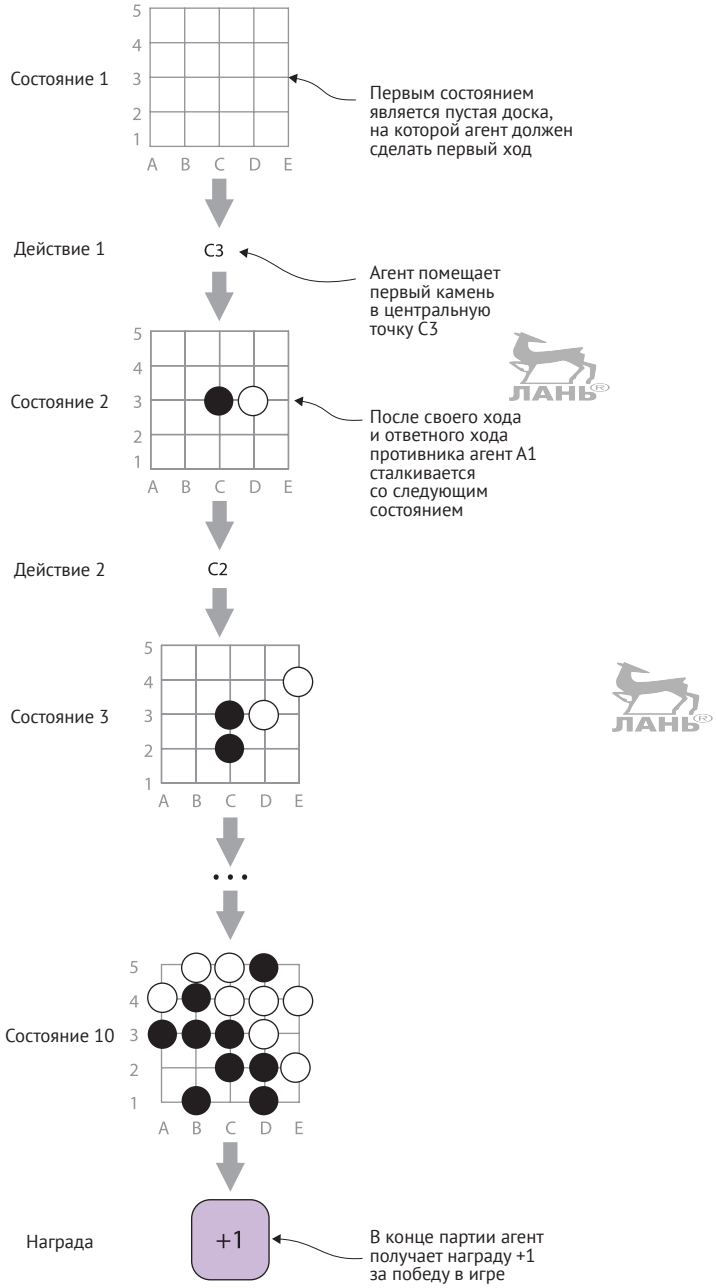


Рис. 9.2 ❖ Игра в го на доске 5×5, выраженная в терминах обучения с подкреплением. Обучающийся агент играет черными. Он видит последовательность состояний (позиций на доске) и выбирает действия (допустимые ходы). В конце эпизода (по завершении партии) он получает награду в соответствии с достигнутыми результатами. В случае победы он получает награду +1

Листинг 9.1 ❖ Вычисление выгоды от действия

```
for exp_idx in range(exp_length):
    total_return[exp_idx] = reward[exp_idx]
    for future_reward_idx in range(exp_idx + 1, exp_length):
            total_return[exp_idx] += reward[future_reward_idx]
```

reward[i] – это награда, полученная агентом сразу после совершения действия i.

Циклическая обработка всех будущих наград и их прибавление к значению выгоды (return).

Это актуально не для всех задач. Вернемся к примеру с игрой «Эрудит». Решения, принятые вами на 1-м ходу, могут повлиять на количество очков, заработанных на 3-м. Скажем, вы придерживаете дорогостоящую букву Щ, дожидаясь возможности разместить ее на бонусной клетке. Однако очень трудно понять, как решение, принятое на 3-м ходу, может повлиять на 20-й. Чтобы учесть это при расчете выгоды от действия, можно вычислить взвешенную сумму всех будущих наград. По мере удаления от действия значения весов должны уменьшаться, чтобы отдаленные награды оказывали на результат меньшее влияние по сравнению с более близкими.

Этот метод называется *дисконтированием* награды. Пример вычисления приведен в листинге 9.2. В этом примере каждому действию приписывается 100 % награды, полученной непосредственно после его совершения. За следующий ход засчитывается только 75 % награды, за третий – 75×75 = 56 % и т. д. Значение 75 % выбрано произвольно. Подходящая ставка дисконтирования будет зависеть от конкретной области, и для ее выбора может потребоваться проведение некоторых экспериментов.

Листинг 9.2 ❖ Определение выгоды от действия с учетом дисконтирования

```
for exp_idx in range(exp_length):
    discounted_return[exp_idx] = reward[exp_idx]
    discount_amount = 0.75
    for future_reward_idx in range(exp_idx + 1, exp_length):
        discounted_return[exp_idx] +=
            discount_amount * reward[future_reward_idx]
        discount_amount *= 0.75
```

Значение discount_amount уменьшается по мере удаления от исходного действия.

В случае ИИ для игры в го награда присуждается только за победу или поражение, что облегчает расчет выгоды. Когда агент выигрывает, выгода каждого действия приравнивается к 1, а когда проигрывает – к –1.

9.3. Создание обучающегося агента

Обучение с подкреплением не позволяет создать ИИ для игры в го или для решения любой другой задачи просто из воздуха. Оно позволяет лишь *улучшить* уже работающего бота. Для начала нам требуется агент, способный, как минимум, сыграть партию целиком. В этом разделе мы поговорим о создании бота для игры в го, использующего нейронную сеть для выбора ходов. Если мы начнем с необученной сети, то бот будет играть так же плохо, как исходный RandomAgent из главы 3. В дальнейшем мы сможем улучшить эту сеть путем обучения с подкреплением.

Политика – это функция, которая выбирает действие, исходя из данного состояния. В предыдущих главах мы обсуждали несколько реализаций класса Agent

с функцией `select_move`. Каждая из этих функций `select_move` представляет собой политику: входом является игровое состояние, а выходом – выбранный ход. Все реализованные до сих пор политики являются действительными, поскольку обеспечивают выбор допустимых ходов. Однако они не являются одинаково эффективными: `MCTSAgent` из главы 4 будет гораздо чаще выигрывать в игре против `RandomAgent` из главы 3, чем наоборот. Для улучшения одного из этих агентов нам необходимо подумать об усовершенствовании алгоритма, написать новый код и протестировать его, т. е. пройти стандартный процесс разработки программного обеспечения.

Для обучения с подкреплением нам нужна политика, которую можно обновлять автоматически с помощью другой компьютерной программы. В главе 6 вы познакомились со сверточными нейронными сетями, классом функций, позволяющих делать именно это. Глубокая нейронная сеть способна производить сложные логические вычисления, и ее поведение можно изменять с помощью алгоритма градиентного спуска.

Выходом нейронной сети для предсказания ходов, разработанной в главах 6 и 7, является вектор со значением для каждой точки доски. Это значение соответствует степени уверенности сети в том, что следующий ход будет совершен именно в этой точке. Как можно сформулировать политику на основе этих выходных данных? Одним из способов является выбор хода, соответствующего наибольшему значению. Это даст хороший результат, если наша сеть уже научилась выбирать хорошие ходы. Однако в этом случае сеть всегда будет выбирать один и тот же ход для конкретного состояния доски, что является проблемой при обучении с подкреплением. Чтобы добиться улучшения с помощью этого метода, нам нужно выбрать множество ходов. Какие-то из них будут более удачными, какие-то – менее. Мы сможем выявить хорошие ходы на основе результатов, к которым они приведут. Однако для улучшения бота нам потребуется множество разнообразных ходов.

Вместо того чтобы всегда выбирать ход с самым высоким рейтингом, нам нужна стохастическая политика. В данном случае слово *стохастическая* говорит о том, что при многократной подаче одного и того же положения доски наш агент сможет выбирать разные ходы. Это подразумевает элемент случайности, но не как в случае с `RandomAgent` из главы 3, который выбирал ходы, не обращая внимания на то, что происходит в игре. Стохастическая политика означает, что выбор хода будет зависеть от состояния доски, но не будет на 100 % предсказуемым.

9.3.1. Сэмплирование из распределения вероятностей

Для любого состояния доски нейронная сеть предоставляет вектор, содержащий по одному элементу на каждую позицию. Чтобы создать на его основе политику, мы можем рассматривать каждый элемент вектора как вероятность выбора того или иного хода. В этом разделе мы поговорим о выборе ходов в соответствии с этими вероятностями.

Например, в случае игры «Камень, ножницы, бумага» можно использовать следующую политику: выбирать камень в 50 % случаев, бумагу – в 30 %, а ножницы – в 20 %. 50 %–30 %–20 % – это *распределение вероятностей* для трех вариантов. Обратите внимание на то, что сумма вероятностей составляет ровно 100 %. Это связано с тем, что наша политика должна всегда выбирать из списка ровно

один элемент. Это необходимое свойство распределения вероятностей. Политика 50 %–30 %–10 % в 10 % случаев оставляла бы нас без какого-либо решения.

Процесс случайного выбора вариантов в соответствии с данной пропорцией называется *сэмплированием* из распределения вероятностей. В следующем листинге представлена функция Python, которая выбирает один из вариантов в соответствии с этой политикой.

Листинг 9.3 ❖ Пример сэмплирования из распределения вероятностей

```
import random

def rps():
    randval = random.random()
    if 0.0 <= randval < 0.5:
        return 'rock'
    elif 0.5 <= randval < 0.8:
        return 'paper'
    else:
        return 'scissors'
```

Выполните этот фрагмент кода несколько раз, чтобы увидеть, как он себя ведет. Вариант `rock` будет появляться чаще, чем `paper`, а `paper` – чаще, чем `scissors`. Однако все три варианта будут появляться регулярно.

Данная логика сэмплирования из распределения вероятностей встроена в библиотеку NumPy в виде функции `np.random.choice`. Следующий листинг демонстрирует реализацию этого же поведения с помощью NumPy.

Листинг 9.4 ❖ Сэмплирование из распределения вероятностей с помощью NumPy

```
import numpy as np

def rps():
    return np.random.choice(
        ['rock', 'paper', 'scissors'],
        p=[0.5, 0.3, 0.2])
```

Кроме того, функция `np.random.choice` может выполнить и *повторное* сэмплирование из того же распределения. После выполнения первого сэмплирования она удалит выбранный элемент из списка и произведет выборку из оставшихся элементов. В результате мы получим полуслучайный упорядоченный список. Элементы, имеющие высокую вероятность, скорее всего, окажутся в начале списка, однако некоторая доля неопределенности сохраняется. Следующий листинг демонстрирует процесс выполнения повторного сэмплирования с помощью функции `np.random.choice`. Фрагмент `size=3` говорит о том, что мы хотим получить три разных элемента, а `replace=False` – о том, что нам не нужны повторяющиеся результаты.

Листинг 9.5 ❖ Повторное сэмплирование из распределения вероятностей с помощью NumPy

```
import numpy as np

def repeated_rps():
    return np.random.choice(
        ['rock', 'paper', 'scissors'],
        size=3,
```

```
        replace=False,
        p=[0.5, 0.3, 0.2])
```

Повторное сэмплирование может пригодиться, если ваша политика для игры в го порекомендует недействительный ход. В этом случае вам нужно будет выбрать другой ход. Тогда вы сможете единожды вызвать функцию `np.random.choice`, а затем просто обработать список, который она сгенерирует.

9.3.2. Обрезка распределения вероятностей

Процесс обучения с подкреплением может протекать довольно нестабильно, особенно на ранних этапах. Агент может чрезмерно отреагировать на несколько случайных побед и временно присвоить высокую вероятность не самым удачным ходам. (В этом отношении он очень напоминает новичков-людей!) При этом вероятность для конкретного хода может достигнуть значения 1. Из-за этого возникнет проблема: если агент будет всегда выбирать один и тот же ход, он не сможет избавиться от этого навыка.

Чтобы предотвратить это, выполняется *обрезка* распределения вероятности, гарантирующая, что ни одно из значений не достигнет 0 или 1. В главе 8 мы делали то же самое с `DeepLearningAgent`. Большую часть работы в данном случае выполняет функция `np.clip` из библиотеки NumPy.

Листинг 9.6 ❖ Обрезка распределения вероятностей

```
def clip_probs(original_probs):
    min_p = 1e-5
    max_p = 1 - min_p
    clipped_probs = np.clip(original_probs, min_p, max_p)
    clipped_probs = clipped_probs / np.sum(clipped_probs)
    return clipped_probs
```

Гарантирует, что результатом по-прежнему является допустимое распределение вероятностей.

9.3.3. Инициализация агента

Теперь приступим к созданию нового типа агента, `PolicyAgent`, который выбирает ходы в соответствии со стохастической политикой и учится на основе полученного опыта. Эта модель может быть идентична модели для предсказания ходов, описанной в главах 6 и 7, единственное отличие будет заключаться в способе ее обучения. Мы добавим эту модель в файл *pg.py*, расположенный в подмодуле *agent* модуля *dlgo*.

В предыдущих главах мы говорили о том, что наша модель нуждается в соответствующей схеме кодирования состояния доски. Конструктор класса `PolicyAgent` может принимать модель и кодировщик доски. Это обеспечивает хорошее разделение задач. Класс `PolicyAgent` отвечает за выбор ходов в соответствии с моделью, а также за изменение ее поведения в соответствии с полученным опытом. При этом он может игнорировать детали структуры модели и схемы кодирования состояния доски.

Листинг 9.7 ❖ Конструктор класса `PolicyAgent`

```
class PolicyAgent(Agent):
    def __init__(self, model, encoder):
        self.model = model      <-- Экземпляр последовательной модели Keras.
        self.encoder = encoder  <-- Реализация интерфейса кодировщика.
```

Перед запуском процесса обучения с подкреплением необходимо создать кодировщик состояния доски, затем модель и, наконец, агента. Этот процесс продемонстрирован в следующем листинге.

Листинг 9.8 ❖ Создание нового обучающегося агента

```
encoder = encoders.simple.SimpleEncoder((board_size, board_size))
model = Sequential()
for layer in dlgo.networks.large.layers(encoder.shape()):
    model.add(layer)
model.add(Dense(encoder.num_points()))
model.add(Activation('softmax'))
new_agent = agent.PolicyAgent(model, encoder)
```

Создание последовательной модели из слоев, описанных в файле *large.py*, расположенном в подмодуле *networks* модуля *dlgo* (см. главу 6).

Добавление выходного слоя, возвращающего распределение вероятностей для точек доски.

При создании агента с использованием вновь созданной модели библиотека Keras инициализирует веса модели небольшими случайными значениями. На этом этапе политика агента обеспечивает практически *равномерное случайное распределение*, т. е. выбирает любые допустимые ходы с примерно одинаковой вероятностью. В процессе дальнейшего обучения решения модели приобретут определенную структуру.

9.3.4. Загрузка агента с диска и его сохранение на диск

Процесс обучения с подкреплением может продолжаться бесконечно. Вы можете потратить на обучение своего бота многие дни и даже недели. При этом вам нужно будет периодически сохранять своего бота на диск, чтобы иметь возможность запускать и останавливать процесс обучения, а также отслеживать динамику показателей производительности.

Для сохранения агента можно использовать описанный в главе 8 формат файла HDF5, который представляет собой удобный способ хранения числовых массивов и отлично интегрируется с библиотеками NumPy и Keras.

Метод `serialize` класса `PolicyAgent` позволяет сохранить его кодировщик и модель на диск, и этого будет достаточно для воссоздания агента.

Листинг 9.9 ❖ Сериализация `PolicyAgent` на диск

```
class PolicyAgent(Agent):
...
    def serialize(self, h5file):
        h5file.create_group('encoder')
        h5file['encoder'].attrs['name'] = self.encoder.name()
        h5file['encoder'].attrs['board_width'] = \
            self.encoder.board_width
        h5file['encoder'].attrs['board_height'] = \
            self.encoder.board_height
        h5file.create_group('model')
        kerasutil.save_model_to_hdf5_group(
            self._model, h5file['model'])
```

Сохранение достаточного количества информации для воссоздания кодировщика доски.

Использование встроенных функций Keras для сохранения модели и ее весов.

Аргументом `h5file` может быть объект `h5py.File` или группа внутри `h5py.File`. Это позволяет объединить агента с другими данными в одном файле HDF5.

Чтобы использовать метод serialize, мы сначала создаем новый файл HDF5, а затем передаем дескриптор файла.

Листинг 9.10 ❖ Пример использования функции serialize

```
import h5py

with h5py.File(output_file, 'w') as outf:
    agent.serialize(outf)
```

Затем соответствующая функция load_policy_agent выполняет обратную процедуру.

Листинг 9.11 ❖ Загрузка агента политики из файла

```
def load_policy_agent(h5file):
    model = kerasutil.load_model_from_hdf5_group(
        h5file['model'])
    encoder_name = h5file['encoder'].attrs['name']
    board_width = h5file['encoder'].attrs['board_width']
    board_height = h5file['encoder'].attrs['board_height']
    encoder = encoders.get_encoder_by_name(
        encoder_name,
        (board_width, board_height))
    return PolicyAgent(model, encoder)
```

Использование встроенных функций Keras для загрузки структуры и весов модели.

Восстановление кодировщика состояния доски.

Воссоздание агента.

9.3.5. Реализация функции выбора хода

Прежде чем мы сможем запустить игру бота с самим собой, необходимо реализовать функцию select_move, которая будет напоминать одноименную функцию, добавленную в DeepLearningAgent в главе 8. Первым делом необходимо закодировать состояние доски в виде тензора (стека матриц; см. приложение А). Затем мы подадим этот тензор на вход модели, чтобы получить распределение вероятностей для разных ходов. После этого выполняем обрезку распределения, чтобы ни одно из значений вероятности не достигло 1 или 0. Этот процесс проиллюстрирован на рис. 9.3, а в листинге 9.12 продемонстрирован способ его реализации.

Листинг 9.12 ❖ Выбор хода с помощью нейронной сети

```
class PolicyAgent(Agent):
...
    def select_move(self, game_state):
        board_tensor = self._encoder.encode(game_state)
        X = np.array([board_tensor])
        move_probs = self._model.predict(X)[0]

        move_probs = clip_probs(move_probs)

        num_moves = self._encoder.board_width * \
            self._encoder.board_height
        candidates = np.arange(num_moves)
        ranked_moves = np.random.choice(
            candidates, num_moves,
            replace=False, p=move_probs)
```

Вызов функции predict библиотеки Keras производит пакетные предсказания, поэтому мы преобразуем отдельное состояние доски в массив и извлекаем из него первый элемент.

Создание массива с индексами всех точек доски.

Сэмплирование из точек доски в соответствии с политикой, создание ранжированного списка точек для выполнения попыток выбора хода.

```
for point_idx in ranked_moves:
    point = self._encoder.decode_point_index(point_idx)
    move = goboard.Move.play(point)
    is_valid = game_state.is_valid_move(move)
    is_an_eye = is_point_an_eye(
        game_state.board,
        point,
        game_state.next_player)
    if is_valid and (not is_an_eye):
        return goboard.Move.play(point)
return goboard.Move.pass_turn()
```

Циклический перебор точек, проверка хода на допустимость и выбор первого из таких ходов.

Достижение этого этапа означает, что разумных ходов больше нет.

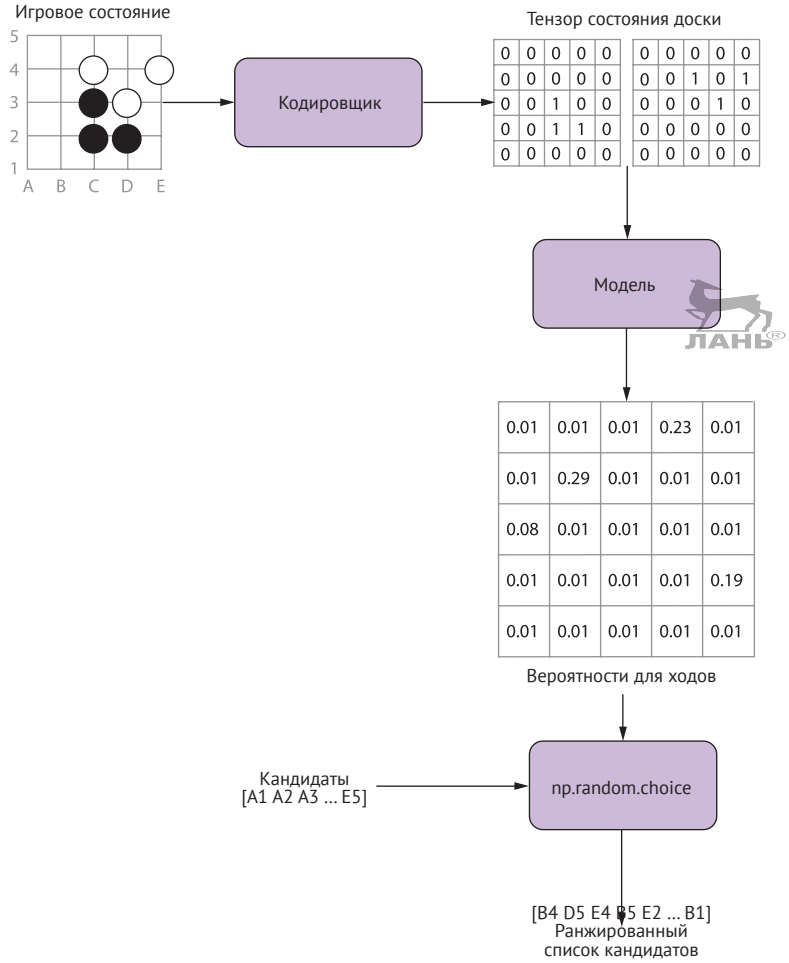


Рис. 9.3 ❖ Процесс выбора хода. Сначала мы кодируем игровое состояние в виде числового тензора, который затем подаем на вход модели для получения распределения вероятностей. После этого мы производим сэмплирование из точек доски в соответствии с вероятностями, чтобы определить порядок совершения попыток выбора хода

9.4. Игра бота с самим собой: практика компьютерной программы

Теперь, когда у нас есть обучающийся агент, способный сыграть партию от начала до конца, мы можем приступить к сбору данных опыта. В случае с ИИ для игры в го это означает проведение тысячи партий. В этом разделе мы поговорим о том, как это реализовать. Сначала обсудим некоторые структуры данных, делающие обработку данных опыта более удобной. А затем реализуем программу для запуска игры бота с самим собой.

9.4.1. Представление данных опыта

Данные опыта состоят из трех компонентов: состояний, действий и наград. Для их организации можно создать единую структуру данных.

Класс `ExperienceBuffer` представляет собой минимальный контейнер для хранения набора данных опыта. Он предусматривает три атрибута: `states`, `actions` и `rewards`. Все они представлены в виде массивов NumPy. Наш агент будет кодировать свои состояния и действия в виде числовых структур. `ExperienceBuffer` – это всего лишь контейнер для передачи набора данных. Ни одна из частей этой реализации не является специфической для методов градиента политики, поэтому вы можете использовать этот класс с другими алгоритмами обучения с подкреплением в следующих главах. Добавьте этот класс в файл *experience.py*, находящийся в подмодуле *rl* модуля *dlgo*.

Листинг 9.13 ❖ Конструктор класса `ExperienceBuffer`

```
class ExperienceBuffer:
    def __init__(self, states, actions, rewards):
        self.states = states
        self.actions = actions
        self.rewards = rewards
```

После сбора большого количества данных опыта нам потребуется способ их сохранения на диск. Для этого снова идеально подойдет формат HDF5. Добавьте метод `serialize` в класс `ExperienceBuffer`.

Листинг 9.14 ❖ Сохранение буфера опыта на диск

```
class ExperienceBuffer:
...
    def serialize(self, h5file):
        h5file.create_group('experience')
        h5file['experience'].create_dataset(
            'states', data=self.states)
        h5file['experience'].create_dataset(
            'actions', data=self.actions)
        h5file['experience'].create_dataset(
            'rewards', data=self.rewards)
```

Нам также понадобится функция `load_experience` для считывания данных опыта из файла. Учтите, что при этом каждый набор данных считывается в массив `np.array`, что позволяет загрузить в память весь набор данных.

Листинг 9.15 ❖ Восстановление ExperienceBuffer из файла HDF5

```
def load_experience(h5file):
    return ExperienceBuffer(
        states=np.array(h5file['experience']['states']),
        actions=np.array(h5file['experience']['actions']),
        rewards=np.array(h5file['experience']['rewards']))
```

Теперь у нас есть простой контейнер для передачи данных опыта. Однако у нас пока нет способа его заполнения решениями агента. Сложность в том, что агент принимает решения по одному, но не получает награды вплоть до окончания игры и определения победителя. Для решения этой проблемы нам необходимо отслеживать все решения, начиная с текущего эпизода. Один из вариантов предполагает встраивание этой логики непосредственно в агента, однако это может излишне усложнить реализацию PolicyAgent. Вместо этого можно создать дискретный объект Experience Collector, единственной обязанностью которого является ведение учета эпизодов.

Объект ExperienceCollector реализует четыре метода:

- begin_episode и complete_episode вызываются драйвером игры бота с самим собой для обозначения начала и окончания отдельной партии;
- record_decision вызывается агентом для обозначения одного выбранного им действия;
- to_buffer упаковывает все данные, собранные объектом ExperienceCollector, и возвращает ExperienceBuffer. Драйвер для игры бота с самим собой вызывает этот метод по завершении партии.

Полная реализация продемонстрирована в следующем листинге.

Листинг 9.16 ❖ Объект для отслеживания решений, принятых в ходе одного эпизода

```
class ExperienceCollector:
    def __init__(self):
        self.states = []
        self.actions = []
        self.rewards = []
        self.current_episode_states = []
        self.current_episode_actions = []

    def begin_episode(self):
        self.current_episode_states = []
        self.current_episode_actions = []

    def record_decision(self, state, action):
        self.current_episode_states.append(state)
        self.current_episode_actions.append(action)

    def complete_episode(self, reward):
        num_states = len(self.current_episode_states)
        self.states += self.current_episode_states
        self.actions += self.current_episode_actions
        self.rewards += [reward for _ in range(num_states)]

        self.current_episode_states = []
        self.current_episode_actions = []
```

Сохранение одного решения в текущем эпизоде; за кодирование состояния и действия отвечает агент.

Распределение итоговой награды среди всех действий, совершенных в ходе партии.

```
    def to_buffer(self):
            return ExperienceBuffer(
                    states=np.array(self.states),
                actions=np.array(self.actions),
                rewards=np.array(self.rewards)
            )
```

Объект ExperienceCollector накапливает списки Python; при этом они преобразуются в массивы NumPy.

Чтобы интегрировать объект ExperienceCollector с агентом, можно добавить метод set_collector, который будет сообщать агенту, куда отправлять накопленные данные опыта. Затем в рамках метода select_move агент будет уведомлять ExperienceCollector о каждом принятом решении.

Листинг 9.17 ❖ Интеграция ExperienceCollector с PolicyAgent

```
class PolicyAgent:
...
    def set_collector(self, collector):
        self.collector = collector
...
    def select_move(self, game_state):
...
        if self.collector is not None:
                self.collector.record_decision(
                    state=board_tensor,
                    action=point_idx
                )
        return goboard.Move.play(point)
```

Позволяет драйверу игры бота с самим собой прикрепить к агенту объект ExperienceCollector.

При выборе хода уведомляет ExperienceCollector о принятом решении.

9.4.2. Симуляция игр

Следующим этапом является симуляция игр. Мы уже дважды делали это ранее в книге: в главе 3, когда обсуждали программу bot_v_bot, и в рамках реализации поиска по дереву методом Монте-Карло в главе 4. Здесь мы можем использовать ту же самую реализацию simulate_game.

Листинг 9.18 ❖ Симуляция игры между двумя агентами

```
def simulate_game(black_player, white_player):
    game = GameState.new_game(BOARD_SIZE)
    agents = {
        Player.black: black_player,
        Player.white: white_player,
    }
    while not game.is_over():
        next_move = agents[game.next_player].select_move(game)
        game = game.apply_move(next_move)
    game_result = scoring.compute_game_result(game)
    return game_result.winner
```

В этой функции black_player и white_player могут соответствовать любому из экземпляров класса Agent. В качестве противника для обучаемого агента PolicyAgent может выступать кто угодно. Теоретически это может быть и человек, однако в этом случае набор необходимого опыта займет очень много времени. Можно

организовать игру со сторонним ботом, обеспечив их взаимодействие с помощью GTP-фреймворка из главы 8.

Кроме того, в качестве противника обучающегося агента можно выставить его собственную копию. Помимо простоты, это решение имеет еще два преимущества.

Во-первых, обучение с подкреплением предполагает наличие большого опыта, состоящего как из побед, так и из поражений. Представьте, что вы играете первую в своей жизни партию в шахматы или го против гроссмейстера. Будучи новичком, вы бы отстали настолько, что определить, где именно вы ошиблись, было бы невозможно. При этом опытный игрок, вероятно, мог бы уверенно победить, даже если бы допустил несколько ошибок. В результате никто из игроков не вынес бы из игры никакого полезного опыта. Вместо этого новички, как правило, начинают играть против других новичков, постепенно набирая опыт. Тот же принцип применяется в случае обучения с подкреплением. Играя с самим собой, бот всегда играет с равным по силе противником.

Во-вторых, когда наш агент играет с самим собой, мы получаем две партии по цене одной. Поскольку оба противника используют один и тот же алгоритм принятия решений, мы можем учиться как на победах, так и на поражениях. Для обучения с подкреплением требуется огромное количество игр, а данный способ позволяет получить их вдвое быстрее.

Чтобы запустить игру бота с самим собой, мы создаем две копии агента и назначаем каждому из них `ExperienceCollector`. Каждый агент нуждается в собственном объекте, поскольку по окончании игры они получат разные награды. В листинге 9.19 продемонстрирован этап инициализации.

Обучение с подкреплением за пределами сферы игр

Игра бота с самим собой – это отличный способ сбора данных опыта в случае настольных игр. В других сферах для запуска агента может потребоваться отдельно симулировать среду. Например, если вы хотите использовать обучение с подкреплением для создания системы управления роботом, вам потребуется детализированная симуляция физического окружения, в котором его планируется использовать.

Для проведения дальнейших экспериментов в области обучения с подкреплением мы рекомендуем использовать ресурс OpenAI Gym (**github.com/openai/gym**), который предоставляет среды для различных настольных игр и видеоигр, а также симуляции физического мира.

Листинг 9.19 ❖ Инициализация процесса генерации данных опыта

```
agent1 = agent.load_policy_agent(h5py.File(agent_filename))
agent2 = agent.load_policy_agent(h5py.File(agent_filename))
collector1 = rl.ExperienceCollector()
collector2 = rl.ExperienceCollector()
agent1.set_collector(collector1)
agent2.set_collector(collector2)
```

Теперь пришло время реализовать основной цикл для симуляции игры бота с самим собой. В этом цикле `agent1` всегда будет играть черными, а `agent2` – белыми.

Это нормально, если agent1 и agent2 абсолютно идентичны и вы намерены использовать их совокупный опыт в ходе дальнейшего обучения. Если ваш обучающийся агент играет против другого эталонного агента, сделайте так, чтобы он поочередно играл то черными, то белыми. В случае игры го черные ходят первыми, поэтому обучающий агент должен приобрести опыт игры за каждую из сторон.

Листинг 9.20 ❖ Проведение серии игр

```
for i in range(num_games):
    collector1.begin_episode()
    collector2.begin_episode()

    game_record = simulate_game(agent1, agent2)
    if game_record.winner == Player.black:
        collector1.complete_episode(reward=1)
        collector2.complete_episode(reward=-1)
    else:
        collector2.complete_episode(reward=1)
        collector1.complete_episode(reward=-1)
```

Агент agent1 победил, поэтому он получает положительную награду.

Агент agent2 победил.

По завершении игры бота с самим собой необходимо объединить полученные данные опыта и сохранить их в файл. Этот файл послужит источником входных данных для сценария обучения, который мы рассмотрим в следующей главе.

Листинг 9.21 ❖ Сохранение пакета данных опыта

```
experience = rl.combine_experience([
    collector1,
    collector2])
with h5py.File(experience_filename, 'w') as experience_outf:
    experience.serialize(experience_outf)
```

Объединение данных опыта обоих агентов в единый буфер.

Сохранение результата в файл HDF5.

Теперь все готово к запуску игр бота с самим собой. В следующей главе мы поговорим о том, как можно улучшить бота на основе данных, полученных в результате проведения таких игр.

9.5. Резюме

- *Агент* – это компьютерная программа, предназначенная для решения определенной задачи. Например, ИИ для игры в го является агентом, целью которого является победа над противниками.
- Цикл обучения с подкреплением предполагает сбор данных опыта, обучение агента на их основе и оценку обновленного агента. По окончании одного цикла мы ожидаем обнаружить небольшое повышение его производительности. В идеале цикл повторяется многократно с целью непрерывного улучшения агента.
- Чтобы применить алгоритм обучения с подкреплением к задаче, ее необходимо описать в терминах *состояний*, *действий* и *наград*.
- *Награды* – это способ контроля поведения обучающегося агента. Мы можем использовать положительную награду в случае достижения агентом желаемых результатов и отрицательную – в случае получения результатов, которых стремимся избегать.

- *Политика* – это правило принятия решений, учитывающее текущее игровое состояние. В случае ИИ для игры в го политикой является алгоритм, который выбирает ход с учетом состояния доски.
- Чтобы создать политику из нейронной сети, можно рассмотреть ее выходной вектор в качестве *распределения вероятностей* для возможных действий и произвести из него *сэмплирование*.
- При использовании алгоритма обучения с подкреплением в сфере игр мы можем собрать данные опыта, запустив *игру бота с самим собой*.

Глава 10

Обучение с подкреплением и градиенты политики

В этой главе:

- улучшение игрового процесса с помощью алгоритма градиента политики;
- реализация обучения методом градиента политики в Keras;
- настройка оптимизаторов для обучения методом градиента политики.

В главе 9 мы говорили о том, как организовать игру бота с самим собой, а затем сохранить полученные данные опыта. Это первая часть процесса обучения с подкреплением. Следующим шагом является использование данных опыта для улучшения агента и повышения частоты его выигрышей. Агент из предыдущей главы использовал нейронную сеть для выбора ходов. В качестве мысленного эксперимента представьте, что мы сместили каждый вес сети на случайную величину. После этого агент будет выбирать другие ходы. По счастливой случайности некоторые из этих новых ходов окажутся лучше прежних, другие – хуже. В конечном итоге обновленный агент может оказаться немного сильнее или слабее предыдущей версии. Но каким именно он окажется, зависит от случая.

Можно ли как-то повлиять на это? В данной главе мы поговорим об *обучении методом градиента политики*. Методы градиента политики помогают приблизительно определить направление смещения весов, позволяющее агенту лучше справляться со своей задачей. Вместо того чтобы смещать каждый вес случайным образом, мы можем проанализировать данные опыта, чтобы попытаться угадать направление, в котором это следует делать. Элемент случайности в этом случае по-прежнему присутствует, однако применение алгоритма градиента политики повышает наши шансы на успех.

В главе 9 мы говорили о принятии решений с помощью стохастической политики – функции, которая определяет вероятность для каждого возможного хода агента. Метод градиента политики, рассматриваемый в этой главе, работает следующим образом:

1) когда агент побеждает, вероятность каждого выбранного им хода повышается;
2) когда агент проигрывает, вероятность каждого выбранного им хода понижается.

Сначала на простом примере мы рассмотрим, как использование этого метода позволяет увеличить частоту выигрышей. Затем мы поговорим о том, как градиентный спуск позволяет произвести желаемое изменение, т. е. увеличить или

уменьшить вероятность конкретного хода в нейронной сети. В конце главы будут приведены некоторые практические советы по управлению процессом обучения.

10.1. Выявление хороших решений с помощью случайных игр

Знакомство с темой обучения на основе политики следует начать с примера гораздо более простой игры, чем го. Назовем эту игру «Определи сумму». Вот ее правила:

- игроки по очереди выбирают число от 1 до 5;
- после 100 ходов каждый игрок складывает все выбранные им числа;
- побеждает игрок с наибольшей итоговой суммой.

Да, это означает, что оптимальной стратегией является выбор числа 5 на каждом ходу. И нет, это не очень хорошая игра. Мы будем использовать ее, чтобы проиллюстрировать метод *обучения на основе политики*, предполагающий постепенное улучшение стохастической политики в зависимости от результатов игры. Поскольку нам известна правильная стратегия для этой игры, мы знаем, как обеспечить наилучший результат с помощью обучения на основе политики.

Игра «Определи сумму» является очень примитивной, однако мы можем использовать ее как метафору для более серьезной игры вроде го. Как и в случае с го, игра «Определи сумму» предполагает проведение длительных партий, в ходе которых у игроков есть множество возможностей, как для принятия удачных решений, так и для совершения ошибок. Чтобы обновить политику на основе результатов игры, нам требуется определить вклад каждого хода в победу или поражение в конкретной игре. Эта проблема *присвоения заслуги* является одной из основных в сфере обучения с подкреплением. В этом разделе мы поговорим об усреднении результатов множества партий с целью присвоения заслуг отдельным решениям. В главе 12 мы доработаем этот метод, чтобы создать более сложный и надежный алгоритм присвоения заслуги.

Начнем с совершенно случайной политики, предполагающей равную вероятность выбора любого из пяти вариантов. (Такая политика называется *равномерно случайной*.) Ожидается, что в ходе одной партии каждое из чисел с 1 по 5 будет выбрано около 20 раз. Однако мы не ожидаем, что каждый из вариантов будет выбран *ровно* 20 раз. Конкретное количество будет варьироваться от игры к игре. В следующем листинге показана функция Python, которая имитирует решения, принимаемые таким агентом в ходе одной партии.

На рис. 10.1 представлены результаты нескольких прогонов. Вы можете попробовать запустить этот фрагмент кода несколько раз и посмотреть, что произойдет.

Листинг 10.1 ❖ Случайный выбор числа от 1 до 5

```
import numpy as np

counts = {1: 0, 2: 0, 3: 0, 4: 0, 5: 0}
for i in range(100):
    choice = np.random.choice([1, 2, 3, 4, 5],
                              p=[0.2, 0.2, 0.2, 0.2, 0.2])
    counts[choice] += 1
print(counts)
```

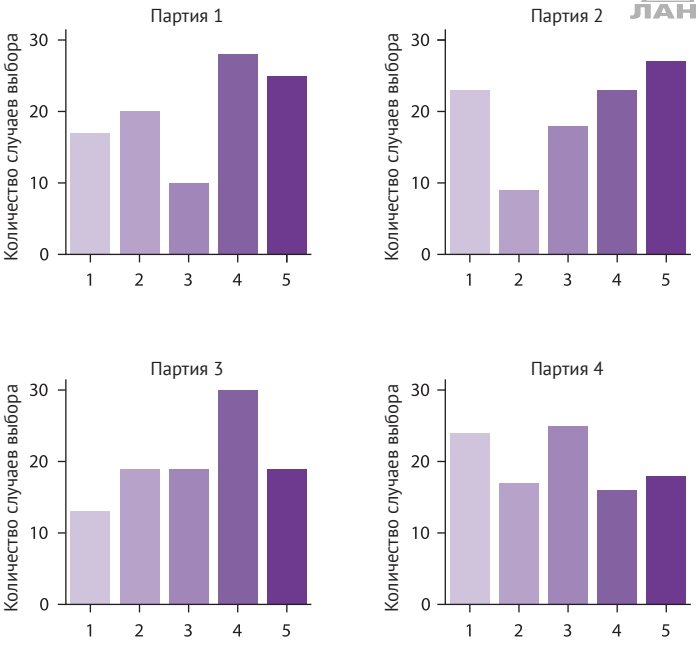


Рис. 10.1 ❖ На данной иллюстрации представлены результаты четырех партий, сыгранных агентом, случайным образом выбирающим число от 1 до 5. Столбцы показывают частоту выбора того или иного варианта. Несмотря на то что во всех играх агент использует одну и ту же политику, количество случаев выбора конкретного варианта может значительно варьироваться от игры к игре

Несмотря на то что в каждой игре агент придерживается одной и той же политику, из-за ее стохастической природы количество случаев выбора конкретного варианта варьируется от игры к игре. И мы можем использовать этот факт для улучшения политики.

В следующем листинге представлена функция, которая симулирует полную партию игры «Определи сумму», отслеживает решения каждого игрока и определяет победителя.

Листинг 10.2 ❖ Симуляция игры «Определи сумму»

```
def simulate_game(policy):
    """Returns a tuple of (winning choices, losing choices)"""
    player_1_choices = {1: 0, 2: 0, 3: 0, 4: 0, 5: 0}
    player_1_total = 0
    player_2_choices = {1: 0, 2: 0, 3: 0, 4: 0, 5: 0}
    player_2_total = 0
    for i in range(100):
        player_1_choice = np.random.choice([1, 2, 3, 4, 5],
                                           p=policy)
        player_1_choices[player_1_choice] += 1
        player_1_total += player_1_choice
```

```
        player_2_choice = np.random.choice([1, 2, 3, 4, 5],
                                           p=policy)
        player_2_choices[player_2_choice] += 1
        player_2_total += player_2_choice
    if player_1_total > player_2_total:
        winner_choices = player_1_choices
        loser_choices = player_2_choices
    else:
        winner_choices = player_2_choices
        loser_choices = player_1_choices
    return (winner_choices, loser_choices)
```

Запустите несколько игр и проанализируйте результаты. Листинг 10.3 иллюстрирует несколько таких запусков. Обычно победитель реже выбирает 1, но не всегда. Иногда победитель чаще выбирает 5, но тоже не всегда.

Листинг 10.3 ❖ Примеры результатов запуска кода из листинга 10.2

```
>>> policy = [0.2, 0.2, 0.2, 0.2, 0.2]
>>> simulate_game(policy)
({1: 20, 2: 23, 3: 15, 4: 25, 5: 17},  <-- Выбор победителя.
 {1: 21, 2: 20, 3: 24, 4: 16, 5: 19})  <-- Выбор проигравшего.
>>> simulate_game(policy)
({1: 22, 2: 22, 3: 19, 4: 20, 5: 17},
 {1: 28, 2: 23, 3: 17, 4: 13, 5: 19})
>>> simulate_game(policy)
({1: 13, 2: 21, 3: 19, 4: 23, 5: 24},
 {1: 22, 2: 20, 3: 19, 4: 19, 5: 20})
>>> simulate_game(policy)
({1: 20, 2: 19, 3: 15, 4: 21, 5: 25},
 {1: 19, 2: 23, 3: 20, 4: 17, 5: 21})
```

Если усреднить результаты четырех партий из листинга 10.3, мы увидим, что победители выбирали вариант 1 в среднем 18,75 раза за игру, а проигравшие – в среднем 22,5 раза. Это имеет смысл, поскольку выбор 1 является плохим ходом. Несмотря на то что в ходе всех этих игр использовалась одна и та же политика, победители и проигравшие демонстрируют разное распределение, поскольку выбор варианта 1 чаще приводил к проигрышу агента.

Разница между ходами, выбранными агентом в случае выигрыша, и ходами, выбранными им в случае проигрыша, позволяет выявить лучшие ходы. Чтобы улучшить политику, можно обновить значения вероятности в соответствии с этой разницей. Для этого можно прибавлять небольшое фиксированное значение всякий раз, когда ход совершается в процессе выигранной партии, и вычитать небольшое фиксированное значение всякий раз, когда ход совершается в процессе проигранной партии. В этом случае распределение вероятностей будет постепенно смещаться в сторону ходов, которые чаще совершаются в выигранных партиях и, соответственно, считаются более предпочтительными. Для игры «Определи сумму» этот алгоритм подходит отлично. Однако в случае такой сложной игры, как го, нам понадобится более изощренная схема обновления значений вероятности. Речь о ней пойдет в разделе 10.2.

Листинг 10.4 ❖ Реализация процесса обучения на основе политики в случае простой игры «Определи сумму»

```
def normalize(policy):
    policy = np.clip(policy, 0, 1)
    return policy / np.sum(policy)
```

Гарантирует то, что сумма вероятностей равна 1.

```
choices = [1, 2, 3, 4, 5]
policy = np.array([0.2, 0.2, 0.2, 0.2, 0.2])
learning_rate = 0.0001
for i in range(num_games):
    win_counts, lose_counts = simulate_game(policy)
    for i, choice in enumerate(choices):
        net_wins = win_counts[choice] - lose_counts[choice]
        policy[i] += learning_rate * net_wins
    policy = normalize(policy)
    print('%d: %s' % (i, policy))
```

Параметр, определяющий скорость обновления политики.

Значение `net_wins` будет положительным, если выбранный вариант чаще встречается в выигранных партиях, чем в проигранных. В противном случае это значение будет отрицательным.

На рис. 10.2 показано, как развивается политика на протяжении всего процесса. Примерно после тысячи сыгранных партий алгоритм перестает выбирать самый худший ход. В результате следующей тысячи партий он вырабатывает практически идеальную стратегию, предполагающую выбор числа 5 на каждом ходу. Кривые не являются абсолютно плавными. Иногда агент многократно выбирает 1 в ходе партии и все равно выигрывает. При этом происходит сдвиг политики в сторону (неправильного) варианта 1. Мы рассчитываем на то, что эти ошибки будут скорректированы в ходе большого количества игр.

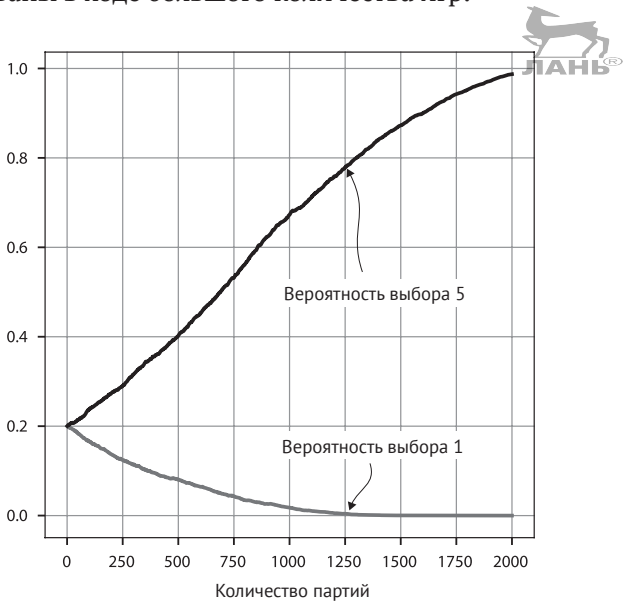


Рис. 10.2 ❖ Этот график демонстрирует процесс развития политики в соответствии с нашей упрощенной схемой. На протяжении сотен партий агент постепенно отказывается от выбора наихудшего хода (1) и все чаще начинает выбирать наилучший ход (5). Обе кривые колеблются из-за периодического смещения политики в неправильном направлении

10.2. Изменение политик нейронной сети методом градиентного спуска

Между процессом обучения агента игре «Определи сумму» и процессом его обучения игре в го существует огромная разница. Политика, используемая в случае игры «Определи сумму», никак не зависит от игрового состояния. Выбор числа 5 – это всегда хороший ход, а выбор числа 1 – всегда плохой. В случае с го, когда мы говорим, что хотим увеличить вероятность хода, мы на самом деле хотим увеличить вероятность выбора этого хода в *подобных ситуациях*. Однако «подобные ситуации» – это очень расплывчатое понятие, и в данном случае мы полагаемся на способность нейронных сетей самостоятельно определить, что под ним подразумевается.

При создании политики нейронной сети в главе 9 мы реализовали функцию, которая принимала в качестве входных данных состояние доски, а на выходе выдавала распределение вероятностей. Для каждого состояния доски в данных опыта мы хотим либо повышать вероятность хода (если он способствовал победе), либо уменьшать ее (если ход привел к проигрышу). Однако мы не можем принудительно изменить вероятности в политике, как это было показано в разделе 9.1. Вместо этого нам следует изменить веса нейронной сети, чтобы добиться желаемого результата. Это можно сделать методом градиентного спуска. Изменение политики с помощью градиентного спуска называется *обучением методом градиента политики*. Существует несколько вариантов данного подхода. В этой главе мы поговорим об алгоритме, который иногда называют *градиентом политики Монте-Карло*, или методом *REINFORCE*. На рис. 10.3 продемонстрировано применение этого алгоритма к играм.

Название метода *REINFORCE* расшифровывается как *Reward Increment = Nonnegative Factor times Offset Reinforcement times Characteristic Eligibility* (Приращение награды = Неотрицательный коэффициент × Подкрепление смещения × Приемлемость характеристики). В этой фразе заключена формула обновления с помощью градиента.

Давайте вспомним, как происходит обучение с учителем методом градиентного спуска, о котором мы говорили в главе 5. Мы выбрали функцию потерь, которая определяет расстояние между нашей функцией и обучающими данными, и вычислили ее градиент. Цель состояла в уменьшении значения функции потерь, т. е. в том, чтобы добиться лучшего ее соответствия обучающим данным. Градиентный спуск – постепенное обновление весов в направлении градиента функции потерь, обеспечивает механизм уменьшения значения функции потерь. Градиент сообщает нам направление, в котором следует сместить каждый из весовых коэффициентов, чтобы уменьшить значение функции потерь.

В случае обучения на основе политики нам необходимо определить направление смещения каждого веса, позволяющее повысить (или понизить) предпочтение, отдаваемое политикой тому или иному ходу. Мы можем создать функцию потерь, градиент которой обладает таким свойством. При ее наличии можно будет воспользоваться быстрой и гибкой инфраструктурой Keras для изменения весов нейронной сети.

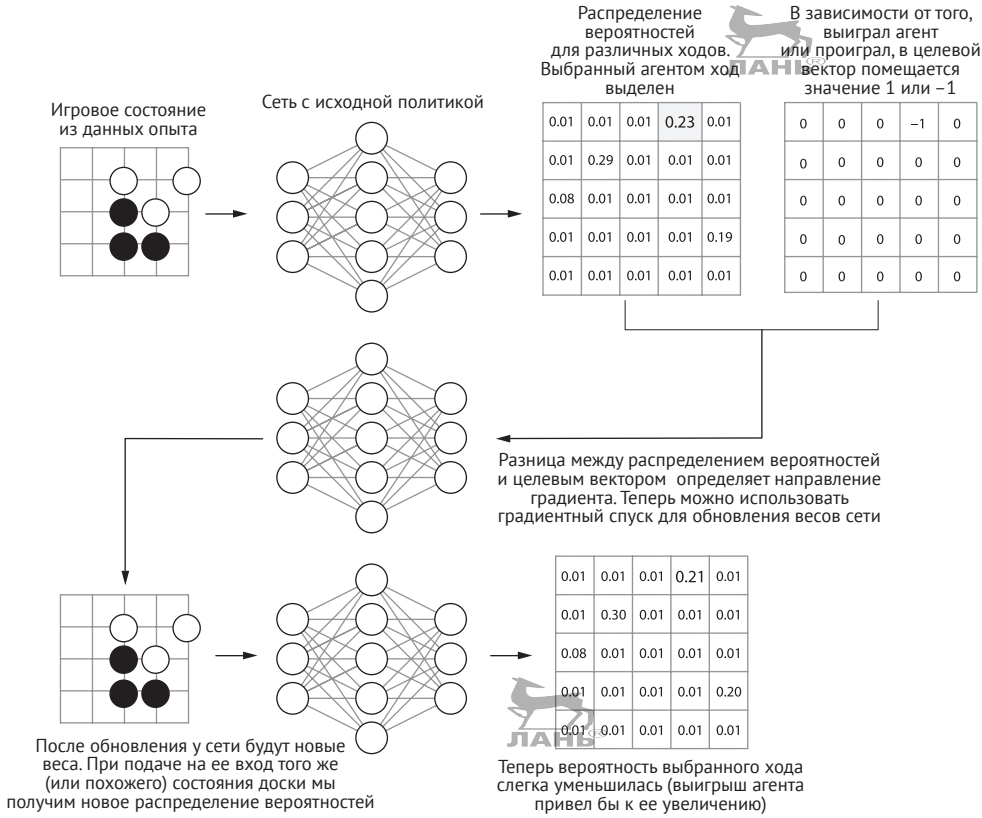


Рис. 10.3 ❖ Процесс обучения методом градиента политики. Мы начинаем с набора записей игровых партий и их результатов. Для каждого выбранного агентом хода мы хотим либо увеличить вероятность выбора (в случае победы агента), либо уменьшить ее (в случае проигрыша агента). Обновление весов политики осуществляется методом градиентного спуска. После прохода градиентного спуска значения вероятности будут смещены в нужном направлении

Вспомните процесс обучения с учителем, который обсуждался в главе 7. Для каждого игрового состояния нам был известен ход игрока-человека. Мы создали целевой вектор, который содержал значение 0 для каждой позиции на доске и значение 1 для обозначения хода игрока-человека. Функция потерь измеряла разницу между предсказанным и истинным распределениями вероятностей, а ее градиент указывал в направлении уменьшения этой разницы. После завершения пакетного градиентного спуска прогнозируемая вероятность хода игрока-человека немного увеличивается.

Это именно тот эффект, которого мы хотим достичь: повысить вероятность совершения конкретного хода. В случае с партиями, выигранными нашим агентом, мы можем создать точно такой же целевой вектор для хода агента, как если бы это был ход человека-игрока из записи реальной игры. Затем функция библиотеки Keras `fit` обновит политику в правильном направлении.

А как насчет проигранных агентом партий? В этом случае мы хотим уменьшить вероятность совершения выбранного хода, однако фактический лучший ход нам неизвестен. В идеале, обновление должно обеспечить эффект, противоположный тому, который имел место в случае выигранной партии.

Оказывается, что при использовании в процессе обучения перекрестной энтропии в качестве функции потерь мы можем просто поместить в целевой вектор значение –1 вместо 1. Это изменит знак градиента функции потерь. При этом веса будут смещаться в противоположном направлении, что приведет к уменьшению вероятности.

Чтобы этого добиться, мы должны использовать перекрестную энтропию. Другие функции потерь вроде среднеквадратической ошибки не позволят добиться такого эффекта. В главе 7 мы выбрали перекрестную энтропию по причине того, что это самый эффективный способ обучения сети выбору одного из фиксированного количества вариантов. В данном случае выбор этой функции обусловлен другим свойством: замена значения –1 на 1 в целевом векторе меняет направление градиента.

Напомним, что данные опыта состоят из трех параллельных массивов:

- `states[i]` представляет определенное состояние доски, с которым столкнулся агент в ходе игры с самим собой;
- `actions[i]` представляет ход, выбранный агентом с учетом этого состояния;
- `rewards[i]` содержит значение 1 в случае победы агента и –1 – в случае его поражения.

Следующий листинг демонстрирует реализацию функции `prepare_experience_data`, которая помещает буфер данных опыта в целевой вектор, подходящий для функции `fit` библиотеки Keras.

Листинг 10.5 ❖ Кодирование данных опыта в виде целевого вектора

```
def prepare_experience_data(experience, board_width, board_height):
    experience_size = experience.actions.shape[0]
    target_vectors = np.zeros((experience_size, board_width * board_height))
    for i in range(experience_size):
        action = experience.actions[i]
        reward = experience.rewards[i]
        target_vectors[i][action] = reward
    return target_vectors
```

В следующем листинге показан процесс реализации функции `train` для класса `PolicyAgent`.

Листинг 10.6 ❖ Обучение агента на основе данных опыта методом градиента политики

```
class PolicyAgent(Agent):
...
    def train(self, experience, lr, clipnorm, batch_size):
        self._model.compile(
            loss='categorical_crossentropy',
            optimizer=SGD(lr=lr, clipnorm=clipnorm))

        target_vectors = prepare_experience_data(
            experience,
```

Параметры `lr` (скорость обучения), `clipnorm` и `batch_size` позволяют точно настроить процесс обучения. Далее мы рассмотрим их более подробно.

Метод `compile` назначает модели оптимизатор. В данном случае в качестве оптимизатора используется стохастический градиентный спуск (СГС).

```
            self._encoder.board_width,
            self._encoder.board_height)
        self._model.fit(
            experience.states, target_vectors,
            batch_size=batch_size,
            epochs=1)
```

Помимо буфера данных опыта, функция `train` принимает три параметра, влияющих на поведение оптимизатора:

- `lr` – это *скорость обучения*, которая определяет степень смещения весов на каждом этапе;
- `clipnorm` жестко ограничивает максимальное смещение весов на каждом отдельном этапе;
- `batch_size` определяет количество ходов из данных опыта, учитываемых при обновлении отдельного веса.

Для получения хорошего результата при обучении методом градиента политики может потребоваться точная настройка этих параметров. В разделе 10.3 вы найдете советы по определению их оптимальных значений.

В главе 7 мы использовали оптимизаторы Adadelta и Adagrad, которые автоматически регулируют скорость обучения на протяжении всего процесса. К сожалению, они оба делают предположения, которые не всегда применимы к процессу обучения методом градиента политики. Вместо них нам следует использовать такой простой оптимизационный алгоритм, как стохастический градиентный спуск, а скорость обучения задавать вручную. Подчеркнем, что в 95 % случаев адаптивные оптимизаторы вроде Adadelta или Adagrad являются наилучшим вариантом, позволяющим ускорить процесс обучения. Однако иногда имеет смысл воспользоваться простым алгоритмом СГС. Кроме того, полезно получить некоторое представление о задании скорости обучения вручную.

Также обратите внимание на то, что буфер данных опыта используется для проведения только одной эпохи обучения. Это отличается от описанного в главе 7 процесса обучения, в котором несколько эпох проводилось на одном и том же наборе обучающих данных. Ключевое различие заключается в том, что в главе 7 использовались заведомо качественные обучающие данные. Каждый ход в этом наборе представлял собой ход опытного игрока-человека, совершенный в реальной партии. В данных, полученных в ходе игры бота с самим собой, результаты игр являются частично рандомизированными, и мы не знаем, какие ходы ответственны за победы. Мы рассчитываем на то, что проведение огромного количества игр позволит сгладить ошибки. Поэтому не хотим повторно использовать отдельные записи партий, чтобы не усиливать влияние содержащихся в них некачественных данных.

К счастью, обучение с подкреплением предоставляет нам неограниченный объем обучающих данных. Вместо проведения многочисленных эпох на одном и том же обучающем наборе нам следует организовать еще одну серию игр бота с самим собой, чтобы сгенерировать новый обучающий набор.

В следующем листинге приведен сценарий для обучения с применением функции `train`. Полная версия этого сценария находится в файле *train_pg.py* на GitHub. Он использует файлы с данными опыта, сгенерированные сценарием `self_play` из главы 9.

Листинг 10.7 ❖ Обучение на основе сохраненных ранее данных опыта

```
learning_agent = agent.load_policy_agent(h5py.File(learning_agent_filename))
for exp_filename in experience_files:
    exp_buffer = rl.load_experience(h5py.File(exp_filename))
    learning_agent.train(
        exp_buffer,
        lr=learning_rate,
        clipnorm=clipnorm,
        batch_size=batch_size)
with h5py.File(updated_agent_filename, 'w') as updated_agent_outf:
    learning_agent.serialize(updated_agent_outf)
```

Количество имеющихся у вас данных может превышать доступный объем памяти. Эта реализация будет считывать их из нескольких файлов по одному фрагменту за раз.

10.3. Советы по обучению бота на основе его игры с самим собой

Задача настройки параметров процесса обучения может оказаться довольно сложной. Кроме того, обучение большой нейронной сети происходит медленно, а значит, вам, скорее всего, придется подождать, прежде чем вы сможете проверить результаты. Вы должны быть готовы к некоторому количеству проб и ошибок, а также фальстартов. В этом разделе вы найдете советы по управлению длительным процессом обучения. Во-первых, мы подробно поговорим об отслеживании прогресса нашего бота. Затем обсудим некоторые параметры настройки, которые влияют на процесс обучения.

Процесс обучения с подкреплением занимает довольно много времени: в случае обучения ИИ для игры в го вам может потребоваться запустить более 10 000 игр бота с самим собой, прежде чем вы заметите какое-либо улучшение. Мы рекомендуем начинать эксперименты с маленькой доски размером 9×9 или даже 5×5. В этом случае партия завершается быстрее, что ускоряет процесс генерации данных. Кроме того, меньшая сложность игры позволяет добиваться прогресса при использовании меньшего количества обучающих данных. Благодаря этому тестирование кода и настройка параметров процесса обучения происходит быстрее. Когда вы будете уверены в своем коде, то сможете перейти к использованию доски большего размера.

10.3.1. Оценка прогресса

В случае такой сложной игры, как го, процесс обучения с подкреплением может занять много времени, особенно при отсутствии доступа к специализированному оборудованию. Ничто так не раздражает, как потеря впустую многих дней работы из-за не выявленной вовремя проблемы. Мы рекомендуем регулярно отслеживать прогресс обучающегося агента путем симуляции большего количества игр. Сценарий *eval_pg_bot.py* организует игру бота против другой своей версии. В следующем листинге показано, как это работает.

Листинг 10.8 ❖ Сценарий для сравнения силы двух агентов

```
wins = 0
losses = 0
color1 = Player.black
for i in range(num_games):
```

Этот сценарий отслеживает победы и поражения с точки зрения агента `agent1`.

`color1` – это цвет камней, которыми играет `agent1`, `agent2` играет камнями противоположного цвета.

```
    print('Simulating game %d/%d...' % (i + 1, num_games))
    if color1 == Player.black:
        black_player, white_player = agent1, agent2
    else:
        white_player, black_player = agent1, agent2
    game_record = simulate_game(black_player, white_player)
    if game_record.winner == color1:
        wins += 1
    else:
        losses += 1
    color1 = color1.other
print('Agent 1 record: %d/%d' % (wins, wins + losses))
```

После каждой игры цвета меняются на случай появления у одного из агентов преимущества при игре камнями определенного цвета.

После каждой серии обучающих игр вы можете выставить обновленного агента в качестве противника исходного агента, чтобы удостовериться в том, что обновленная версия демонстрирует прогресс или, по крайней мере, не ухудшается.

10.3.2. Измерение небольших различий в силе

Результатом проведения множества тысяч игр бота с самим собой может оказаться лишь незначительное улучшение по сравнению с его предыдущей версией. Выявление разницы в несколько процентов – это довольно сложная задача. Допустим, мы завершили раунд обучения. Чтобы оценить прогресс бота, мы позволяем ему сыграть 100 партий против его предыдущей версии. Обновленный бот выигрывает 53 раза. Действительно ли новый бот на 3 % сильнее предыдущего? Или ему просто повезло? Нам нужен способ определения необходимого объема данных для точной оценки силы бота.

Представьте, что процесс обучения ни к чему не привел, и обновленный бот ничем не отличается от своей предыдущей версии. Какова вероятность того, что идентичный бот победит в 53 играх? Для вычисления этой вероятности в статистике используется так называемый *биномиальный тест*. Для его реализации в Python используется библиотека scipy:

```
>>> from scipy.stats import binom_test
>>> binom_test(53, 100, 0.5)
0.61729941358925255
```

В этом фрагменте кода:

- 53 – это фактическое количество выигранных партий;
- 100 – количество симулированных игр;
- 0,5 – вероятность победы бота, идентичного противнику, в одной игре.

В результате биноминального теста было получено значение 61,7 %. Если ваш бот идентичен своему противнику, его шанс на победу в 53 или более партиях составляет 61,7 %. Эта вероятность иногда называется *p-значением* (p-value). Это *не* значит, что с вероятностью 61,7 % ваш бот ничему не научился. Это просто говорит об отсутствии достаточного количества данных для оценки его прогресса. Чтобы удостовериться в улучшении бота, необходимо выполнить большее количество тестов.

Выходит, что нам требуется провести немало испытаний для оценки разницы в силе ботов. Если в серии из 1000 игр агент победит 530 раз, то в результате биноминального теста будет получено p-значение около 6 %. Для принятия окончательного решения, как правило, используется значение, не превышающее 5 %.

Однако в 5%-ном пороге нет ничего магического. Воспринимайте p-значения как спектр, показывающий, насколько скептически следует относиться к победе бота.

10.3.3. Настройка алгоритма стохастического градиентного спуска (СГС)

Оптимизатор СГС предусматривает несколько параметров, способных повлиять на его производительность. Обычно они предполагают достижение компромисса между скоростью и точностью. Обучение на основе градиента политики, как правило, более чувствительно к точности, чем обучение с учителем, что следует учитывать при задании параметров.

Первым параметром, который нужно задать, является скорость обучения. Чтобы правильно выбрать скорость обучения, необходимо представлять, к каким проблемам может привести некорректное задание этого параметра. В этом разделе мы будем обращаться к рис. 10.4 с воображаемой целевой функцией, которую стремимся минимизировать. Концептуально данная диаграмма не отличается от диаграмм, обсуждавшихся в разделе 5.4. Однако в данном случае мы ограничили ее одним измерением, чтобы проиллюстрировать несколько конкретных точек. В действительности при оптимизации функции, как правило, используются многие тысячи измерений.

Рис. 10.4 ❖ Этот график иллюстрирует то, как может варьироваться гипотетическая целевая функция в зависимости от изучаемого веса. Нам нужно переместить значение θ (греческая буква тета) из текущего местоположения в точку минимума. Можно сказать, что алгоритм градиентного спуска как бы заставляет вес скатываться вниз

В главе 5 мы пытались оптимизировать функцию потерь, которая измеряла ошибку между предсказаниями и известными правильными образцами. В данном случае целевая функция описывает процент выигрышей нашего бота. (Технически, когда речь идет о проценте выигрышей, целевую функцию следует максимизировать. В обоих случаях используются одни и те же принципы, только зеркально отраженные.) В отличие от функции потерь, процент выигрышей нельзя вычислить напрямую, однако мы можем оценить градиент функции на основе данных, полученных в процессе игры бота с самим собой. На рис. 10.4 ось абсцисс представляет некоторый вес сети, а ось ординат показывает, как изменяется зна-

чение целевой функции в зависимости от этого веса. Точкой отмечено текущее состояние сети. В идеале можно представить, что градиентный спуск заставляет эту точку скатиться вниз и остановиться в долине.

Если скорость обучения слишком мала, как на рис. 10.5, оптимизатор сдвинет вес в правильном направлении, однако для достижения минимума потребуется очень много раундов обучения. Исходя из соображений эффективности, мы хотим максимизировать скорость обучения, не вызвав при этом проблем.

Рис. 10.5 ❖ В данном примере скорость обучения слишком мала, и нам требуется очень много обновлений, чтобы вес достиг минимума

Если мы немного промахнемся, значение целевой функции может улучшиться не так сильно, как могло бы. Однако следующий градиент будет указывать в правильном направлении. Таким образом, значение функции может колебаться в течение некоторого времени, как показано на рис. 10.6.

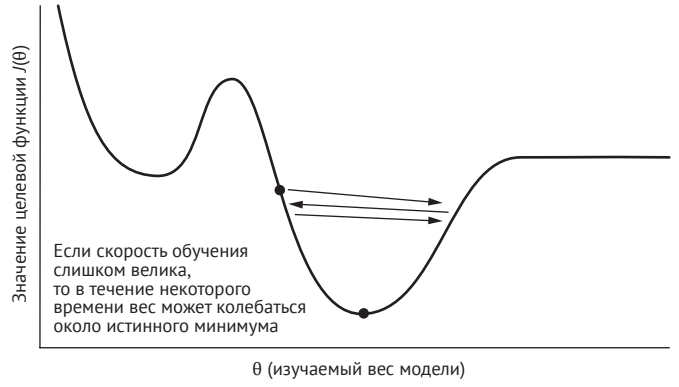


Рис. 10.6 ❖ В данном случае скорость обучения слишком велика. Вес промахивается мимо цели. На следующем этапе обучения градиент будет указывать в противоположном направлении, однако существует вероятность очередного промаха. Это может привести к тому, что в течение некоторого времени вес будет колебаться около истинного минимума

Если мы промахнемся очень сильно, вес окажется на плоском участке справа (см. рис. 10.7). На этом участке градиент близок к нулю, а значит, градиентный спуск больше не подсказывает нам, в каком направлении следует смещать вес. Это может привести к застреванию целевой функции. Данная проблема существует не только в теории. Такие плоские участки нередко имеют место в сетях, использующих блоки линейной ректификации, о которых мы говорили в главе 6. Специалисты по глубокому обучению иногда называют это *проблемой мертвых ReLU*: они считаются «мертвыми», поскольку постоянно возвращают значение 0 и прекращают делать вклад в общий процесс обучения.

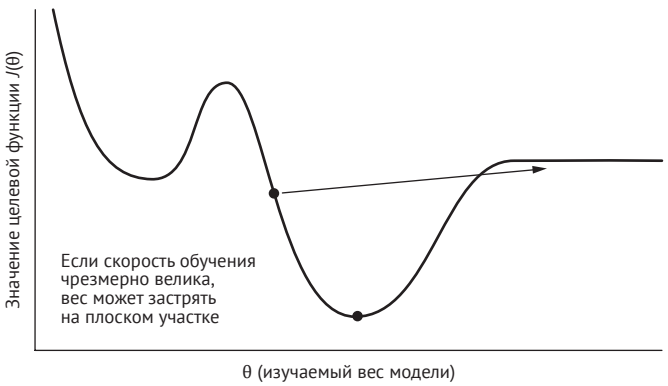


Рис. 10.7 ❖ В данном случае скорость обучения настолько велика, что вес оказывается на плоском участке справа. На этом участке градиент равен 0, поэтому оптимизатор больше не знает, в каком направлении ему двигаться. Вес может застрять здесь навсегда. Эта проблема часто имеет место в нейронных сетях, в которых используются блоки линейной ректификации

Это те проблемы, которые могут возникнуть в случае промаха в правильном направлении. При обучении методом градиента политики проблема является еще более серьезной, поскольку нам неизвестен истинный градиент, в направлении которого мы пытаемся следовать. Где-то во Вселенной существует теоретическая функция, связывающая уровень игры агента с весами его сети. Однако у нас нет никакой возможности записать ее. Максимум, на что мы способны, – это оценить градиент на основании обучающих данных. Эта оценка является далеко не точной и иногда указывает в неправильном направлении. (Вспомните рис. 10.2 из раздела 10.1; вероятность выбора наилучшего хода часто немного смещалась в неправильном направлении. В случае го или другой сложной игры данные, полученные в ходе игры бота с самим собой, окажутся еще более зашумленными.)

Если вы сдвинетесь слишком далеко в неправильном направлении, вес может остановиться в другой долине слева (см. рис. 10.8). Это называется *забыванием*: сеть научилась обрабатывать определенное свойство набора данных, а затем внезапно разучилась это делать.

Мы можем предпринять некоторые шаги для повышения точности оценок градиента. Напомним, что стохастический градиентный спуск работает на мини-пакетах: этот оптимизатор берет небольшое подмножество из набора обучающих

данных, вычисляет на его основе значение градиента, а затем обновляет все веса. Большой размер пакетов способствует сглаживанию ошибки. В библиотеке Keras в качестве размера пакета по умолчанию используется значение 32, которое хорошо подходит для решения многих задач при обучении с учителем. В случае обучения на основе политики мы рекомендуем использовать гораздо большие значения, например 1024 или даже 2048.

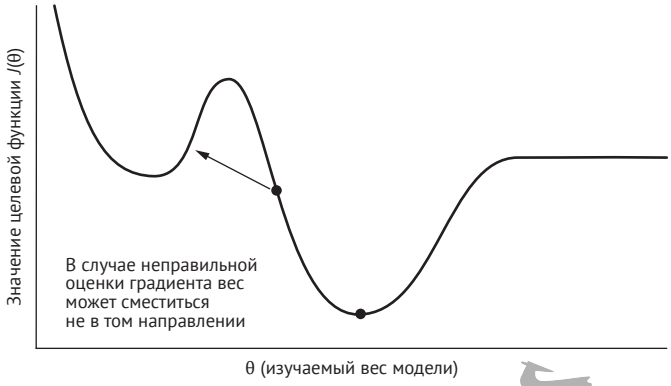


Рис. 10.8 ❖ В случае обучения методом градиента политики мы пытаемся оценить истинный градиент на основании очень зашумленного сигнала. Иногда отдельная оценка может указывать в неправильном направлении. Если сместить вес слишком далеко в неправильном направлении, он может переместиться из истинного минимума в середине графика в локальный минимум слева и на некоторое время там застрять

Наконец, при обучении методом градиента политики может произойти застревание в *локальных максимумах*, при котором любое постепенное изменение политики делает бота слабее. Иногда этого удается избежать путем введения дополнительного элемента случайности в игру бота с самим собой. В небольшом количестве случаев (скажем, 1 % или 0,5 % ходов) агент может игнорировать политику и выбирать ход совершенно случайным образом.

На практике процесс обучения методом градиента политики происходит следующим образом:

1) генерация большого количества игр бота с самим собой (в зависимости от доступного объема памяти);
2) обучение;
3) тестирование обновленного бота в игре против его предыдущей версии;
4) если бот стал заметно сильнее, переключитесь на новую версию;
5) если уровень игры бота не изменился, сгенерируйте больше игр и обучите его снова;
6) если бот стал значительно слабее, настройте параметры оптимизатора и проведите переобучение.

Настройка параметров оптимизатора может показаться сложной задачей, однако, потренировавшись и поэкспериментировав, вы сможете развить необходимую интуицию. В табл. 10.1 кратко представлены рассмотренные в этом разделе советы.

Таблица 10.1. Устранение проблем, возникающих при обучении сети на основе политики

Проблема	Возможные причины	Решение
Процент выигрышей застрял у отметки 50 %	Скорость обучения слишком мала	Увеличьте скорость обучения
	Застревание на локальном максимуме	Добавьте дополнительный элемент случайности в игру бота с самим собой
Процент выигрышей значительно снизился	Промах	Уменьшите скорость обучения
	Неверные оценки градиента	Увеличьте размер пакета
		Соберите большее количество данных в играх бота с самим собой

10.4. Резюме

- Обучение на основе политики представляет собой один из методов обучения с подкреплением, предполагающий обновление политики на основании данных опыта. В случае игры это означает, что бот учится выбирать более удачные ходы, исходя из результатов игры агента.
- Одна из форм обучения на основе политики предполагает увеличение вероятности каждого хода, совершенного в выигранной партии, и уменьшение вероятности ходов, совершенных в проигранной партии. На протяжении тысячи игр этот алгоритм постепенно обновит политику, что приведет к повышению частоты выигрышей. Этот алгоритм называется *обучением с помощью градиента политики*.
- *Перекрестная энтропия* – это функция потерь, предназначенная для ситуаций, когда требуется выбрать один параметр из фиксированного набора вариантов. В главе 7 мы использовали эту функцию потерь, когда пытались предсказать, какой ход выберет человек в данной игровой ситуации. Перекрестную энтропию также можно адаптировать для проведения обучения на основе градиента политики.
- Обучение методом градиента политики можно реализовать в рамках фреймворка Keras путем корректного кодирования данных опыта и дальнейшего обучения с использованием перекрестной энтропии.
- Обучение методом градиента политики может потребовать ручной настройки параметров оптимизатора. Такой способ обучения может потребовать использования меньшей скорости обучения и большего размера мини-пакета, чем в случае обучения с учителем.

Глава 11

Обучение с подкреплением и методы на основе ценности действий

В этой главе:

- создание самосовершенствующегося игрового бота с помощью алгоритма Q-обучения;
- определение и обучение нейронных сетей с несколькими входами в Keras;
- создание и обучение агента Q-обучения средствами Keras.

Когда эксперты комментируют турниры по шахматам или го, они часто используют выражения вроде «На данном этапе черные сильно отстают» или «Пока все складывается в пользу белых». Что значит «лидировать» и «отставать», когда речь идет о середине партии в такие стратегические игры? Это не баскетбол, где для определения лидера достаточно посмотреть на текущий счет. На самом деле данные комментарии означают, что позиция на доске благоприятна для того или другого игрока. Прояснить это можно с помощью мысленного эксперимента. Найдите сто пар игроков примерно одинакового уровня. Предоставьте каждой паре состояние доски из середины партии, чтобы они продолжили играть с этой позиции. Если игроки черными выиграют с небольшим перевесом, скажем, в 55 партиях из 100, можно сказать, что начальная позиция была чуть более благоприятной для черных.

Разумеется, комментаторы такого не делают. Вместо этого при формулировании своих прогнозов они полагаются на собственную интуицию, натренированную на тысячах партий. В этой главе мы покажем, как обучить компьютер делать аналогичные прогнозы. И для этого компьютеру придется сделать то же, что и человеку, – сыграть множество партий.

В данной главе представлен алгоритм *Q-обучения*, представляющего собой один из методов обучения с подкреплением, позволяющий научить агента прогнозировать будущую награду. (В контексте игр *награда* соответствует *выигрышу*.) Сначала мы поговорим о том, как агент Q-обучения принимает решения и совершенствуется с течением времени. Затем рассмотрим процесс реализации алгоритма Q-обучения в рамках Keras. После этого можно будет приступать к обучению самосовершенствующегося бота, слегка отличающегося от агента из главы 10, обученного на основе политики.

11.1. Игры и алгоритм Q-обучения

Представьте, что у вас есть функция, которая определяет ваши шансы на победу после совершения конкретного хода. Эта так называемая *функция ценности действия* (action-value) позволяет определить, насколько ценным является конкретное действие. Она значительно упрощает игру, позволяя просто выбирать действие с наибольшей ценностью на каждом ходу. Осталось лишь понять, как реализовать данную функцию.

Этот раздел посвящен *Q-обучению*, разновидности обучения с подкреплением, позволяющей реализовать функцию *ценности действия*. Конечно, в случае игры го просчитать истинное значение этой функции никогда не удастся, поскольку это потребовало бы анализа всего дерева игры со всеми его триллионами триллионов вариантов. Однако мы можем итеративно улучшать *оценку* функции ценности действия в ходе игры бота с самим собой. По мере повышения точности этой оценки использующий ее бот будет становиться все более сильным.

Название данного метода происходит от стандартных математических обозначений. Традиционно запись $Q(s,a)$ используется для представления функции ценности действия. Это функция двух переменных: s – это состояние, с которым столкнулся агент (например, позиция на доске), a – это рассматриваемое агентом действие (возможный следующий ход). На рис. 11.1 представлены входные данные для функции ценности действия. Эта глава посвящена теме *глубокого Q-обучения*, предполагающего применение нейронной сети для оценки функции Q. Однако большинство описанных здесь принципов также применимо к классическому алгоритму Q-обучения, в котором функция Q аппроксимируется с помощью простой таблицы, содержащей строку для каждого возможного состояния и столбец для каждого возможного действия.

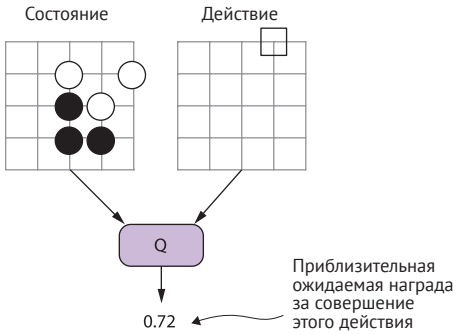


Рис. 11.1 ❖ *Функция ценности действия* принимает два значения: состояние (позиция на доске) и действие (предлагаемый ход), а возвращает оценку ожидаемой награды (шанс на победу в игре) в случае выбора агентом этого действия. В математике функция ценности действия традиционно обозначается буквой *Q*

Ранее процесс обучения с подкреплением рассматривался как непосредственное изучение политики – правила выбора ходов, поэтому структура алгоритма Q-обучения должна показаться вам знакомой. Сначала мы организуем игру аген-

та с самим собой и запишем все принятые им решения и результаты партий, позволяющие оценить качество этих решений. Затем на основании собранных данных мы обновим поведение агента. Алгоритм Q-обучения отличается от обучения на основе политики в плане того, как агент принимает решения в игре и как он обновляет свое поведение в зависимости от результатов.

Для создания игрового агента на основе функции Q нужно преобразовать эту функцию в политику. Один из способов предполагает передачу функции Q всех возможных ходов и выбор варианта с наибольшей ожидаемой наградой, как показано на рис. 11.2. Такая политика называется *жадной*.

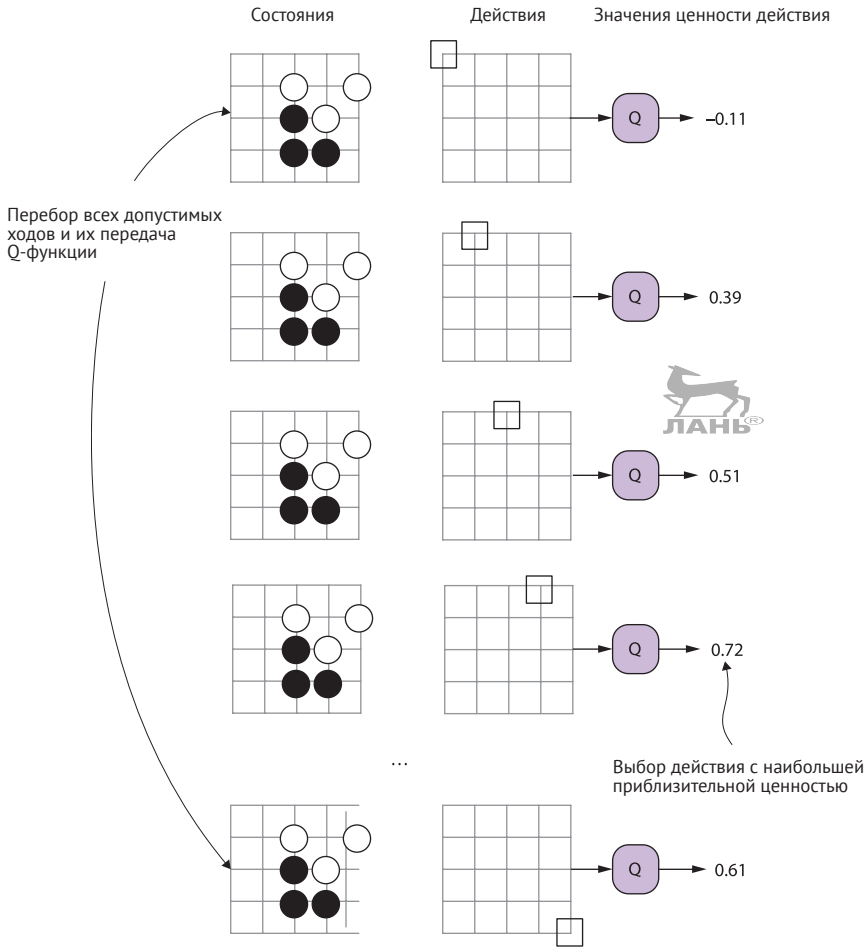


Рис. 11.2 ❖ *Жадная* политика предполагает перебор всех возможных ходов и оценку ценности действия, после чего выбирается действие с наибольшим значением этой оценки. (Многие допустимые ходы были опущены для экономии места)

Если вы уверены в своих оценках ценности действия, то жадная политика является самым подходящим вариантом. Однако для *улучшения* оценок следует перио-

дически позволять боту исследовать неизвестные ходы. Такая политика называется ε-*жадной* и предполагает некоторую долю (ε) случайных ходов, а в остальных случаях – применение обычной жадной политики. На рис. 11.3 эта политика представлена в виде блок-схемы.

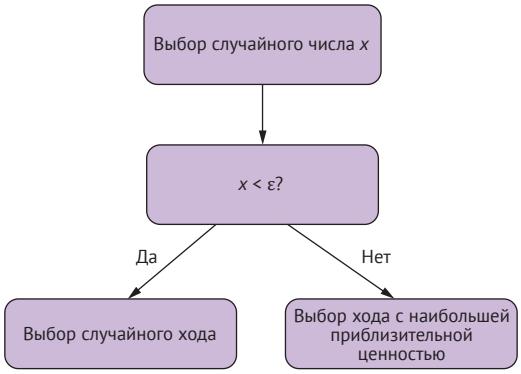


Рис. 11.3 ❖ Блок-схема применения ε-*жадной* политики для оценки ценности действия. Данная политика пытается найти баланс между выбором лучшего хода и изучением неизвестных ходов, используя для этого значение ε

Греческая буква ε (эпсилон) часто используется для обозначения небольшого дробного значения.

Листинг 11.1 ❖ Псевдокод для реализации ε-жадной политики

```
def select_action(state, epsilon):
    possible_actions = get_possible_actions(state)
    if random.random() < epsilon:
        return random.choice(possible_actions)
    best_action = None
    best_value = MIN_VALUE
    for action in get_possible_actions(state):
        action_value = self.estimate_action_value(state, action)
        if action_value > best_value:
            best_action = action
            best_value = action_value
    return best_action
```

Исследование случайных ходов.

Выбор наилучшего из известных ходов.

Выбор значения ε предполагает достижение некоторого компромисса. Когда оно близко к 0, агент выбирает лучшие ходы в соответствии со своей текущей оценкой ценности действия. Однако при этом агент не имеет возможности опробовать новые ходы, а значит, повысить точность своих оценок. При слишком высоком значении ε агент проиграет больше игр, но при этом исследует множество новых ходов.

Здесь можно провести аналогию с тем, как люди приобретают новый навык, будь то игра в го или игра на фортепиано. Как правило, они достигают некоторого плато – точки, в которой они чувствуют себя комфортно с определенным набором навыков, и перестают совершенствоваться. Чтобы преодолеть его, нужно выйти из зоны комфорта и поэкспериментировать с чем-то новым. В случае игры на

фортепиано это могут быть новые аппликатуры и ритмы, а в случае игры в го – новые дебюты и тактические приемы. В незнакомой ситуации ваша производительность может ухудшиться, однако освоение новых навыков позволит вам стать сильнее, чем раньше.

При использовании алгоритма Q-обучения мы обычно начинаем с довольно высокого значения ε, например 0,5. По мере совершенствования агента мы постепенно уменьшаем это значение. Учтите, что при достижении значения 0 агент перестанет обучаться и будет просто выбирать одни и те же ходы снова и снова.

После генерации большого количества игр алгоритм Q-обучения напоминает процесс обучения с учителем. Действия, предпринятые агентом, входят в обучающий набор, а результаты игр можно рассматривать в качестве хороших меток. Разумеется, обучающий набор будет содержать некоторое количество случайных побед, однако в долгосрочной перспективе можно рассчитывать на такое же количество компенсирующих их поражений.

Так же, как описанная в главе 7 модель училась предсказывать ходы человека в не виденных ею ранее партиях, модель для оценки ценности действий может научиться предсказывать ценность ходов, которые она раньше никогда не совершала. Мы можем использовать результаты партий в качестве целей в процессе обучения, как показано на рис. 11.4. Для подобного обобщения нам необходима нейронная сеть, обладающая соответствующей архитектурой, и большое количество обучающих данных.

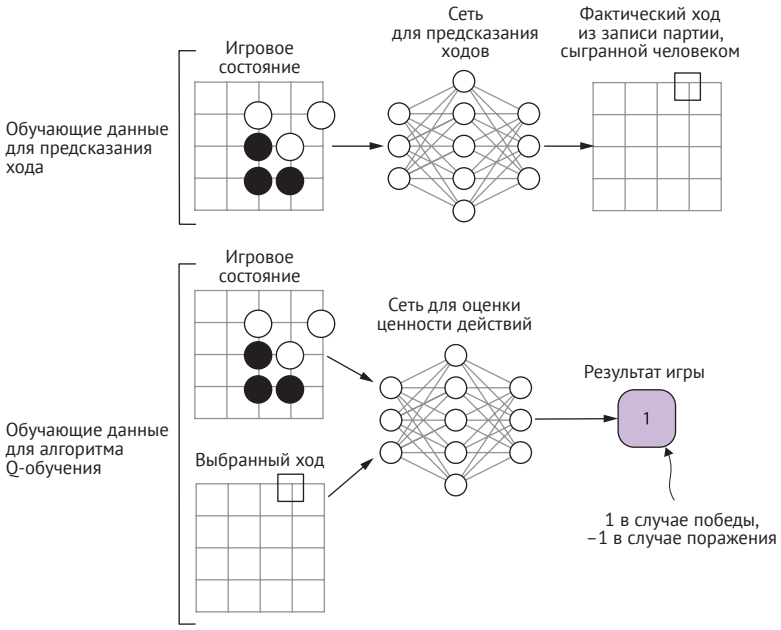


Рис. 11.4 ❖ Подготовка обучающих данных для глубокого Q-обучения. Вверху показано, как были созданы обучающие данные для описанных в главах 6 и 7 сетей для предсказания ходов. На вход подавалось состояние доски, а выходом был выбранный ход. Внизу показана структура обучающих данных для алгоритма Q-обучения. Состояние доски и выбранный ход – это входные данные, а выходом является результат игры: 1 в случае победы и –1 в случае поражения

11.2. Реализация алгоритма Q-обучения в Keras

В этом разделе мы поговорим о реализации алгоритма Q-обучения в среде Keras. До сих пор мы использовали Keras для реализации функций, имеющих один вход и один выход. Поскольку функция ценности действия предусматривает два входа, нам потребуются новые возможности Keras для создания соответствующей сети. Далее мы поговорим о сетях с двумя входами, а затем обсудим процесс оценки ходов, подготовки обучающих данных и обучения агента.

11.2.1. Создание сетей с двумя входами с помощью Keras

В предыдущих главах для определения нейронных сетей использовалась последовательная модель Keras `Sequential`. Следующий листинг демонстрирует пример модели, определенной с помощью последовательного API.

Листинг 11.2 ❖ Определение модели с помощью последовательного API библиотеки Keras

```
from keras.models import Sequential
from keras.layers import Dense

model = Sequential()
model.add(Dense(32, input_shape=(19, 19)))
model.add(Dense(24))
```

Для определения нейронной сети библиотека Keras предусматривает второй *функциональный* API, расширяющий возможности последовательного API. Любую последовательную сеть можно переписать в функциональном стиле, кроме того, можно создавать сложные сети, которые невозможно описать в последовательном стиле.

Основное различие заключается в способе задания связей между слоями. Для соединения слоев в последовательной модели мы многократно вызываем метод `add` объекта модели, что автоматически соединяет выход последнего слоя со входом нового. Для соединения слоев в функциональной модели мы передаем сигнал от входного слоя на следующий с помощью синтаксиса, напоминающего вызов функции. Явное создание каждой связи позволяет нам описывать более сложные сети. В следующем листинге продемонстрирована сеть из листинга 11.2, созданная с использованием функционального стиля.

Листинг 11.3 ❖ Определение идентичной модели с помощью функционального API Keras

```
from keras.models import Model
from keras.layers import Dense, Input

model_input = Input(shape=(19, 19))
hidden_layer = Dense(32)(model_input)
output_layer = Dense(24)(hidden_layer)

model = Model(inputs=[model_input], outputs=[output_layer])
```

Соединяет `model_input` со входом слоя `Dense` и присваивает этому слою имя `hidden_layer`.

Соединяет `hidden_layer` со входом нового слоя `Dense` и присваивает этому слою имя `output_layer`.

Эти две модели являются идентичными. Последовательный API представляет собой удобный способ описания наиболее распространенных нейронных сетей, а функциональный API обеспечивает гибкость, позволяя задавать многочисленные входы и выходы, а также сложные соединения.

Поскольку наша сеть для оценки ценности действия имеет два входа и один выход, на каком-то этапе нам потребуется объединить две цепочки входов. Для этого можно использовать слой Keras `Concatenate`. Слой конкатенации не выполняет никаких вычислений, он просто склеивает два вектора или тензора в один, как показано на рис. 11.5. Он принимает необязательный аргумент `axis`, указывающий измерение, подлежащее конкатенации. По умолчанию это последнее измерение, что нам и нужно. Все остальные измерения должны быть одинакового размера.

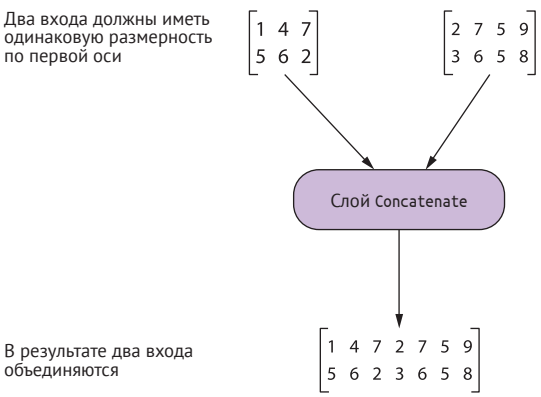


Рис. 11.5 ❖ Слой конкатенации Keras объединяет два тензора в один

Теперь мы можем создать сеть для реализации функции ценности действия. Вспомните сверточные сети, которые использовались для предсказания ходов в главах 6 и 7. Концептуально работу сети можно разделить на два этапа: сначала сверточные слои выявляют значимые формы расположения камней на доске, на основе которых плотный слой принимает решение. На рис. 11.6 показано, как слои сети для предсказания ходов решают эти две задачи.

В случае сети для реализации функции ценности действия мы по-прежнему хотим выявить важные формы и группы камней. Любая форма, представляющая важность для предсказания хода, вероятно, будет представлять важность и для оценки ценности действия, поэтому эта часть сети может иметь ту же структуру. Разница проявляется на этапе принятия решения. Вместо того чтобы принимать решение только на основе выявленных групп камней, мы хотим оценить ценность действия, исходя из обработанного состояния доски *и* предложенного действия. Поэтому мы можем подать вектор предложенного хода после сверточных слоев. Такая сеть представлена на рис. 11.8.

Поскольку для обозначения проигрыша мы используем значение –1, а для обозначения выигрыша – 1, значение функции ценности действия должно принадлежать диапазону от –1 до 1. Для этого мы добавляем слой `Dense` размером 1 с функцией активации `tanh`, которая может быть вам известна из тригонометрии, в которой она называется гиперболическим тангенсом. В случае глубокого обучения нам нет дела до ее тригонометрических свойств. Она представляет для нас интерес как гладкая функция, ограниченная значениями –1 и 1. Вне зависимости от

вычислений, выполняемых на предыдущих слоях сети, выходное значение будет принадлежать желаемому диапазону. График функции tanh показан на рис. 11.7.

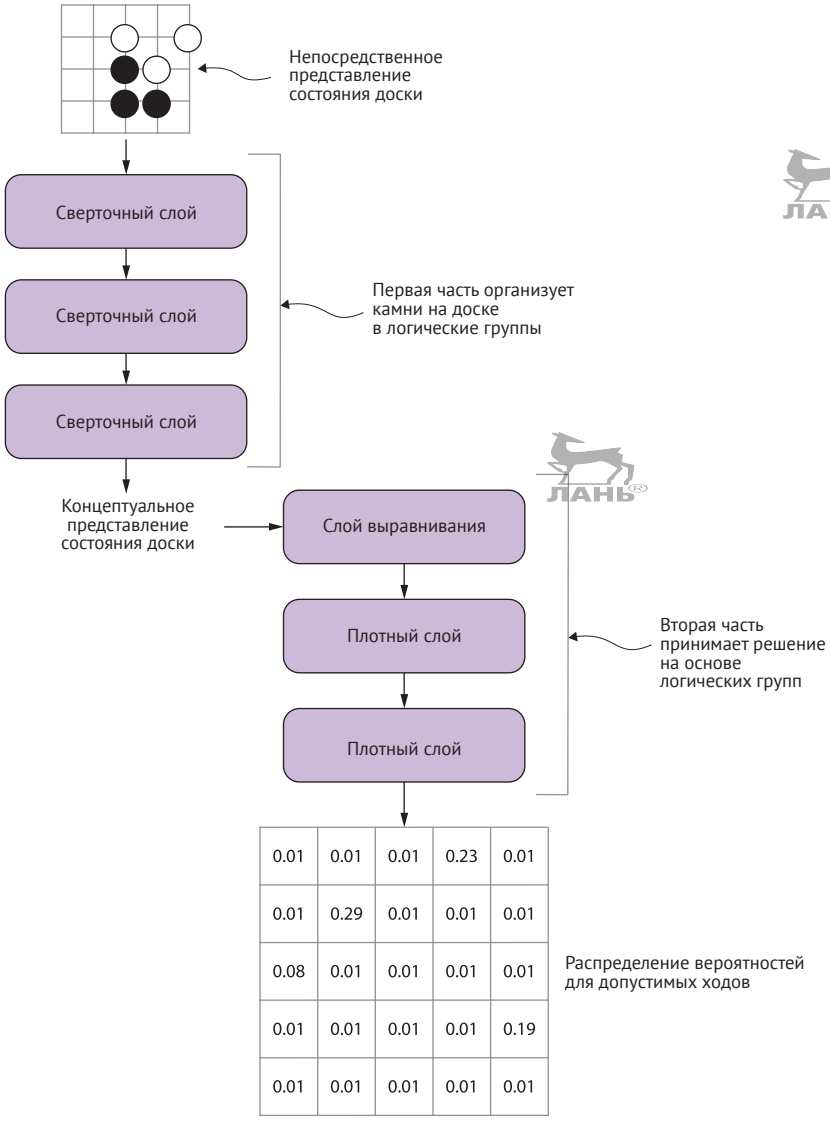


Рис. 11.6 ❖ Сеть для предсказания хода, описанная в главах 6 и 7. Несмотря на большое количество слоев, концептуально работу сети можно разделить на два этапа. Сверточные слои обрабатывают данные о расположении камней и организуют их в логические группы и тактические формы. На основании этого представления плотные слои выбирают действие

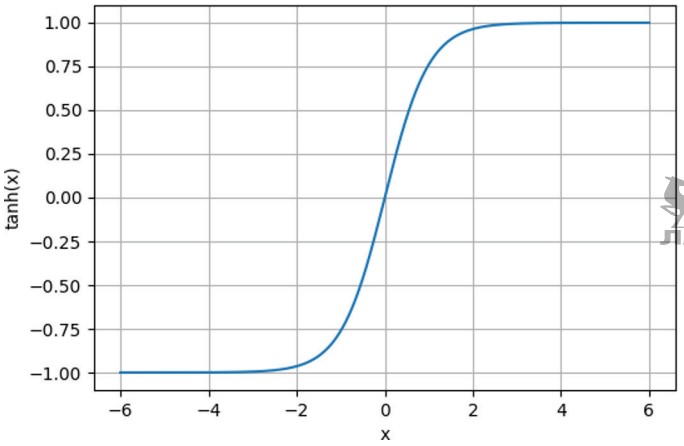


Рис. 11.7 ❖ Функция tanh (гиперболический тангенс), значения которой принадлежат диапазону –1 до 1

Полная спецификация сети для оценки ценности действия приведена в следующем листинге.

Листинг 11.4 ❖ Сеть с двумя входами для реализации функции ценности действия

```
from keras.models import Model
from keras.layers import Conv2D, Dense, Flatten, Input
from keras.layers import ZeroPadding2D, concatenate

board_input = Input(shape=encoder.shape(), name='board_input')
action_input = Input(shape=(encoder.num_points(),),
                     name='action_input')

conv1a = ZeroPadding2D((2, 2))(board_input)
conv1b = Conv2D(64, (5, 5), activation='relu')(conv1a)

conv2a = ZeroPadding2D((1, 1))(conv1b)
conv2b = Conv2D(64, (3, 3), actionvation='relu')(conv2a)

flat = Flatten()(conv2b)
processed_board = Dense(512)(flat)

board_and_action = concatenate([action_input, processed_board])
hidden_layer = Dense(256, activation='relu')(board_and_action)
value_output = Dense(1, activation='tanh')(hidden_layer)

model = Model(inputs=[board_input, action_input],
              outputs=value_output)
```

Добавьте любое количество сверточных слоев. Все, что хорошо работало для предсказания хода, должно сработать и здесь.

Вы можете поэкспериментировать с размером этого скрытого слоя.

Слой активации с функцией tanh ограничивает выходное значение диапазоном от –1 до 1.

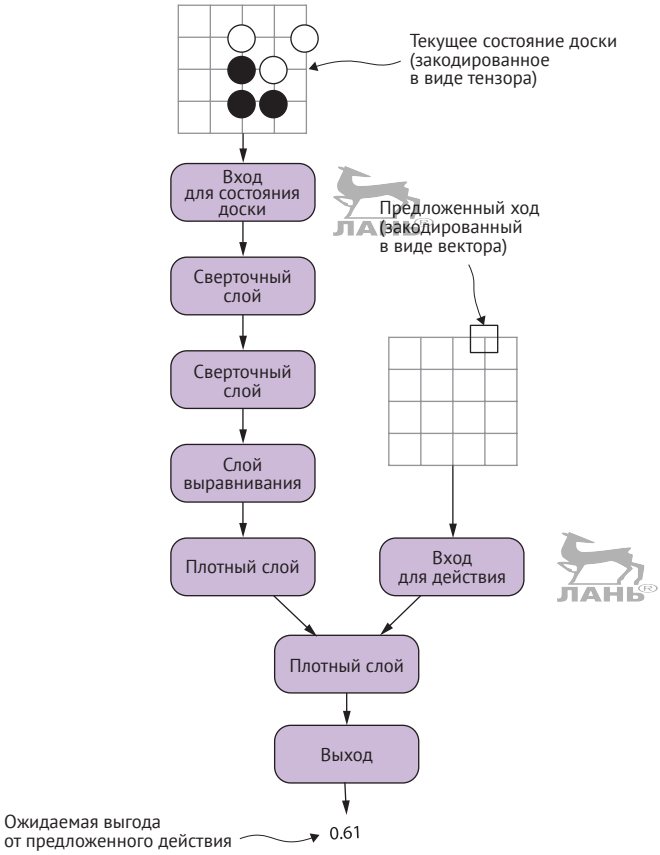


Рис. 11.8 ❖ Нейронная сеть с двумя входами описана в листинге 11.4. Состояние доски проходит через несколько сверточных слоев, как в случае с сетью для предсказания ходов из главы 7. Предложенный ход подается на отдельный вход, объединяется с выходом сверточных слоев и пропускается через другой плотный слой

11.2.2. Реализация ε-жадной политики с помощью Keras

Давайте приступим к созданию агента `QAgent`, к которому будет применен алгоритм Q-обучения. Этот код следует поместить в файл *q.py* в подмодуле *rl* модуля *dlgo*. В листинге 11.5 продемонстрирован конструктор, который так же, как и в случае с обучением на основе политики, принимает модель и кодировщик доски. Кроме того, мы определяем два служебных метода. Метод `set_temperature` позволяет варьировать значение ε в процессе обучения. Как и в главе 9, метод `set_collector` позволяет присоединить объект `ExperienceCollector`, чтобы сохранить данные опыта для дальнейшего обучения.

Листинг 11.5 ❖ Конструктор и служебные методы для агента Q-обучения

```
class QAgent(Agent):
    def __init__(self, model, encoder):
        self.model = model
```

```
        self.encoder = encoder
        self.collector = None
        self.temperature = 0.0

    def set_temperature(self, temperature):
        self.temperature = temperature

    def set_collector(self, collector):
        self.collector = collector
```

Параметр `temperature` – это значение ε, определяющее степень случайности политики.

Подробную информацию об использовании объекта `collector` для записи данных опыта агента можно найти в главе 9.

Теперь пришло время реализовать ε-жадную политику. Вместо того чтобы просто выбирать ход с самой высокой оценкой, мы сортируем все ходы и пробуем их по порядку. Как и в главе 9, это предотвращает самоуничтожение агента в конце выигранной партии.

Листинг 11.6 ❖ Выбор ходов для агента Q-обучения

```
class QAgent(Agent):
    ...
    def select_move(self, game_state):
        board_tensor = self.encoder.encode(game_state)

        moves = []
        board_tensors = []
        for move in game_state.legal_moves():
            if not move.is_play:
                continue
            moves.append(self.encoder.encode_point(move.point))
            board_tensors.append(board_tensor)
        if not moves:
            return goboard.Move.pass_turn()

        num_moves = len(moves)
        board_tensors = np.array(board_tensors)
        move_vectors = np.zeros(
            (num_moves, self.encoder.num_points()))
        for i, move in enumerate(moves):
            move_vectors[i][move] = 1

        values = self.model.predict(
            [board_tensors, move_vectors])
        values = values.reshape(len(moves))

        ranked_moves = self.rank_moves_eps_greedy(values)

        for move_idx in ranked_moves:
            point = self.encoder.decode_point_index(
                moves[move_idx])
            if not is_point_an_eye(game_state.board,
                                   point,
                                   game_state.next_player):
                if self.collector is not None:
                    self.collector.record_decision(
                        state=board_tensor,
                        action=moves[move_idx],
                    )
```

Генерация списка всех допустимых ходов.

Если допустимых ходов не осталось, агент может пропустить ход.

Унитарное кодирование всех допустимых ходов (более подробную информацию об унитарном кодировании можно найти в главе 5).

Это форма функции `predict` с двумя входами: мы передаем эти два входа в виде списка.

Значения будут представлены в виде матрицы $N \times 1$, где N – количество допустимых ходов. Вызов функции `reshape` преобразует ее в вектор размером N.

Ранжирование ходов в соответствии с ε-жадной политикой.

Выбор первого несамоубийственного хода из списка, как в случае игры агента с самим собой из главы 9.

Запись решения в буфер данных опыта (см. главу 9).

```
            return goboard.Move.play(point)
        return goboard.Move.pass_turn()
```

Этого этапа вы достигнете в том случае, если все допустимые ходы будут признаны самоубийственными.

Q-обучение и поиск по дереву

Структура реализации `select_move` напоминает некоторые алгоритмы поиска по дереву, описанные в главе 4. Например, алгоритм альфа-бета-отсечения опирается на функцию оценки состояния доски, которая определяет, кто из игроков лидирует и насколько. Это аналогично, но не идентично функции ценности действия, рассмотренной в этой главе. Представьте, что агент играет черными, оценивает некоторый ход *X* и получает приблизительное значение функции ценности действия, равное 0,65. Теперь мы точно знаем, как будет выглядеть доска после совершения хода X. Также мы знаем, что выигрыш для черных означает проигрыш для белых. Таким образом, мы можем сказать, что следующее состояние доски оценивается в –0,65 для белых. Математически это отношение можно описать следующим образом:

$$Q(s, a) = -V(s'),$$

где s' – это состояние доски, которое увидит игрок белыми после того, как игрок черными выберет ход a.

Несмотря на то что в целом алгоритм Q-обучения может применяться к любой среде, эта равнозначность между значением функции ценности действия для одного состояния и ценностью следующего состояния имеет место только в детерминированных играх.

Глава 12 посвящена третьему методу обучения с подкреплением, который предполагает использование функции ценности вместо функции ценности действия. В главах 13 и 14 представлены способы интеграции такой функции ценности с алгоритмом поиска по дереву.

Теперь нам осталось лишь реализовать код для сортировки ходов в порядке уменьшения ценности. Сложность заключается в том, что у нас есть два параллельных массива: `values` и `moves`. Чтобы с ней справиться, мы используем функцию NumPy `argsort`. Вместо сортировки массива эта функция возвращает список индексов, в соответствии с которыми можно считать элементы параллельного массива. Принцип работы функции `argsort` представлен на рис. 11.9. А в листинге 11.7 показано, как с ее помощью можно ранжировать ходы.

Листинг 11.7 ❖ Выбор ходов для агента Q-обучения

```
class QAgent(Agent):
    ...
    def rank_moves_eps_greedy(self, values):
        if np.random.random() < self.temperature:
            values = np.random.random(values.shape)
        ranked_moves = np.argsort(values)
        return ranked_moves[::-1]
```

В случае исследования ходы упорядочиваются случайным образом, а не в соответствии с реальными значениями.

Получение индексов ходов в порядке возрастания ценности.

Синтаксис `[::1]` – это самый эффективный способ обратить порядок элементов вектора в NumPy. В результате ходы будут ранжированы в порядке уменьшения ценности.

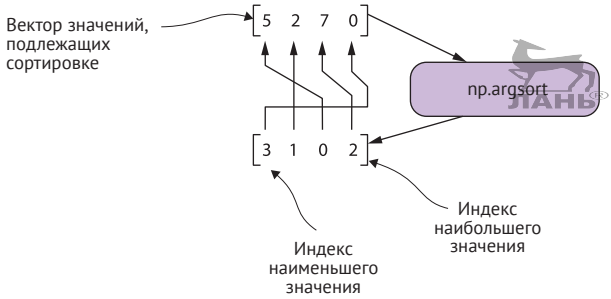


Рис. 11.9 ❖ Иллюстрация принципа работы функции `argsort` из библиотеки NumPy. Эта функция принимает вектор значений, которые требуется отсортировать. Вместо непосредственной сортировки значений она возвращает вектор отсортированных индексов. Первым значением выходного вектора является индекс наименьшего значения из входного вектора, а последним – индекс наибольшего значения из входного вектора

Теперь мы готовы реализовать игры агента с самим собой. Далее обсудим способ обучения сети для реализации функции ценности действия.

11.2.3. Обучение сети, реализующей функцию ценности действия

После сбора данных опыта можно будет приступить к обновлению агента. В случае с обучением методом градиента политики нам было приблизительно известно нужное значение градиента, однако нам требовалось разработать сложную схему для обновления градиента в среде Keras. Алгоритм Q-обучения, напротив, предполагает простое применение функции `fit` библиотеки Keras. Мы можем поместить результаты игры прямо в целевой вектор.

В главе 6 были рассмотрены две функции потерь: среднеквадратическая ошибка и перекрестная энтропия. Мы использовали перекрестную энтропию, когда хотели выбрать один вариант из дискретного набора элементов – тогда речь шла об одной из точек доски для игры в го. С другой стороны, функция Q является непрерывной и может принимать любые значения в диапазоне от –1 до 1. Для решения этой задачи предпочтительной функцией потерь является среднеквадратическая ошибка.

В следующем листинге представлена реализация функции `train` для класса `QAgent`.

Листинг 11.8 ❖ Обучение агента Q-обучения на основе набранного им опыта

```
class QAgent(Agent):
...
    def train(self, experience, lr=0.1, batch_size=128):
        opt = SGD(lr=lr)
        self.model.compile(loss='mse', optimizer=opt)

        n = experience.states.shape[0]
        num_moves = self.encoder.num_points()
        y = np.zeros((n,))
        actions = np.zeros((n, num_moves))
        for i in range(n):
            action = experience.actions[i]
```

`lr` и `batch_size` – это параметры, позволяющие точно настроить процесс обучения (более подробную информацию можно найти в главе 10).

`mse` – это среднеквадратическая ошибка. Мы используем ее вместо `categorical_crossentropy`, потому что имеем дело с непрерывной функцией.

```
            reward = experience.rewards[i]
            actions[i][action] = 1
            y[i] = reward
        self.model.fit(
            [experience.states, actions], y,
            batch_size=batch_size,
            epochs=1)
```

Передает два разных входа в виде списка.

11.3. Резюме

- *Функция ценности действия* позволяет агенту оценить ожидаемую награду после совершения определенного действия. В случае с играми речь идет об ожидаемом шансе на победу.
- *Q-обучение* – это метод обучения с подкреплением, предполагающий оценку функции ценности действия (традиционно обозначаемой буквой Q).
- При обучении агента Q-обучения, как правило, используется *ε-жадная политика*, которая предполагает, что в некоторых случаях агент будет выбирать наиболее ценные ходы, а в остальное время – делать выбор случайным образом. Параметр ε контролирует процесс исследования агентом неизвестных ходов.
- Функциональный API Keras позволяет проектировать нейронные сети с несколькими входами, несколькими выходами или сложными внутренними связями. В случае с Q-обучением функциональный API может использоваться для создания сети с отдельными входами для игрового состояния и предложенного хода.

Глава 12

Обучение с подкреплением и методы типа «актор–критик»

В этой главе:

- использование преимущества для повышения эффективности обучения с подкреплением;
- создание самосовершенствующегося игрового бота с помощью методов типа «актор–критик»;
- разработка и обучение нейронных сетей с несколькими выходами в Keras.

Когда вы учитесь играть в го, один из лучших способов повышения уровня заключается в получении комментария со стороны более сильного игрока. Иногда самым полезным оказывается простое указание на тот момент, в котором был совершен выигрышный или проигрышный ход. Например, более сильный игрок может сказать что-то вроде: «К 30-му ходу вы уже сильно отставали» или «На 110-м ходу у вас была выигрышная позиция, но ваш противник обошел вас к 130-му ходу».

В чем польза таких комментариев? Возможно, у вас нет времени на тщательное исследование всех 300 ходов партии, однако вы вполне можете сосредоточить все свое внимание на последовательности, состоящей из 10 или 20 ходов. Более опытный игрок может указать на самые важные этапы партии.

Этот принцип применяется в алгоритме обучения типа *«актор–критик»*, который сочетает в себе обучение на основе политики (см. главу 10) и обучение на основе оценки ценности действия (см. главу 11). Функция политики играет роль *актора*, который выбирает ходы. Функция ценности является *критиком*, который определяет, лидирует агент в игре или отстает. Эта обратная связь направляет процесс обучения так же, как обзор партии направляет процесс обучения человека.

В данной главе мы поговорим о создании самосовершенствующегося игрового бота с помощью методов обучения типа «актор–критик». Ключевым понятием данного метода является *преимущество* – разница между фактическим и ожидаемым результатом игры. Мы начнем с иллюстрации того, как преимущество может улучшить процесс обучения. После этого приступим к созданию игрового агента, обучающегося с помощью алгоритма «актор–критик». Первым делом мы обсудим процесс выбора ходов, а затем реализуем новый процесс обучения. Код обеих функций будет в значительной степени основан на примерах, приведенных в главах 10 и 11. Конечным результатом станет агент, сочетающий в себе преимущества обучения на основе политики и Q-обучения.

12.1. Преимущество позволяет выявить важные решения

В главе 10 мы кратко упомянули проблему присвоения заслуги. Допустим, обучающийся агент выиграл партию, состоящую из 200 ходов. Его победа дает основания полагать, что он совершил, по крайней мере, несколько хороших ходов, однако он, вероятно, допустил и некоторое количество ошибок. *Присвоение заслуги* – это проблема отделения хороших ходов, выбор которых требуется поощрить, от плохих ходов, которые агенту следует игнорировать. В этом разделе речь пойдет о *преимуществе*, формуле для оценки вклада конкретного решения в итоговый результат. Сначала мы поговорим о том, как концепция преимущества помогает с присвоением заслуги, а затем разберем фрагменты кода для его вычисления.

12.1.1. Что такое преимущество?

Представьте, что вы смотрите баскетбольный матч, и на исходе четвертой четверти ваш любимый игрок совершает трехочковый бросок. Ваша реакция на это будет зависеть от состояния игры. При счете 80:78 вы, вероятно, вскочете с места, а при счете 110:80 – останетесь равнодушны. Чем объясняется эта разница? При почти равном счете трехочковый бросок сильно влияет на ожидаемый исход игры. С другой стороны, при огромной разнице в счете один бросок не может сильно повлиять на результат. Самыми важными являются броски, совершаемые тогда, когда результат игры все еще не ясен. В обучении с подкреплением данная концепция количественно выражена в формуле преимущества.

Для вычисления преимущества сначала нужно оценить ценность состояния, которое мы обозначим как $V(s)$. Это ожидаемая выгода, оцененная агентом по достижении конкретного состояния s. В случае с играми выражением $V(s)$ можно обозначить преимущество, предоставляемое состоянием доски тому или иному игроку. Если значение $V(s)$ близко к 1, значит, наш агент находится в выигрышной позиции, а если значение $V(s)$ близко к –1, значит, агент проигрывает.

Данная концепция аналогична функции ценности действия $Q(s,a)$ из предыдущей главы. Разница лишь в том, что $V(s)$ оценивает преимущества состояния доски *до* совершения хода, а $Q(s, a)$ – *после*.

Как правило, преимущество определяется следующим образом:

$$A = Q(s, a) - V(s).$$

Если вы, находясь в благоприятном положении (высокое значение $V(s)$), совершаете ужасный ход (низкое значение $Q(s, a)$), то теряете свое преимущество, поэтому результат вычисления отрицателен. Однако проблема в том, что мы не знаем, как рассчитать $Q(s, a)$. Тем не менее мы можем рассматривать награду, получаемую в конце игры, как объективную оценку значения Q. Таким образом, мы можем подождать до получения награды R, а затем оценить преимущество следующим образом:

$$A = R - V(s).$$

Именно эту формулу мы будем использовать для оценки преимущества в данной главе. Давайте посмотрим, чем нам может быть полезно это значение.

В целях иллюстрации представим, что у нас уже есть точный способ оценки значения $V(s)$. В реальности наш агент осваивает функции ценности и политики одновременно. В следующем разделе мы поговорим о том, как это происходит.

Рассмотрим несколько примеров:

- в начале игры $V(s) = 0$: шансы обоих игроков примерно равны. Предположим, что наш агент выигрывает игру, тогда его награда составляет 1, а преимущество первого хода составляет $1 - 0 = 1$;
- представьте, что игра почти закончена, и наш агент сильно лидирует: $V(s) = 0{,}95$. Если наш агент действительно победит, то преимущество данного состояния будет оценено как $1 - 0{,}95 = 0{,}05$;
- теперь представьте, что у нашего агента есть еще одна выигрышная позиция с оценкой $V(s) = 0{,}95$. Но в этом случае бот допускает ошибку и проигрывает, получая награду -1. Тогда преимущество составит: $-1 - 1{,}95 = -1{,}95$.

На рис. 12.1 и 12.2 проиллюстрирован процесс расчета преимущества для гипотетической игры. Первые несколько ходов позволили нашему обучающемуся агенту вырваться вперед, затем он совершил несколько серьезных ошибок и оказался в проигрышной позиции. Ближе к 150-му ходу ему удалось переломить ситуацию и выиграть партию. В соответствии с методом на основе градиента политики (см. глава 10) мы бы одинаково оценили все ходы в этой игре. В случае использования метода обучения типа «актор–критик» нам необходимо выявить наиболее важные ходы и придать им больший вес. Вычисление преимущества позволяет сделать именно это.

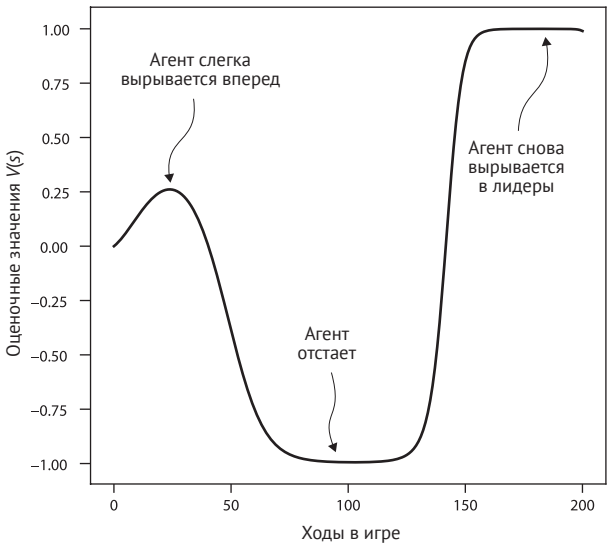


Рис. 12.1 ❖ Примерные оценки ходов гипотетической партии, состоящей из 200 ходов. Вначале обучающийся агент немного вырвался вперед, затем сильно отстал, после чего внезапно переломил ситуацию и вышел победителем

Поскольку обучающийся агент выиграл, преимущество вычисляется по формуле: $A(s) = 1 - V(s)$. На рис. 12.2 видно, что кривая преимущества идентична кривой

оценочного значения, только отражена сверху вниз. Наибольшее преимущество наблюдается, пока агент сильно отстает. Поскольку большинство игроков проигрывает в такой плохой ситуации, агент, должно быть, совершил на каком-то этапе отличный ход.

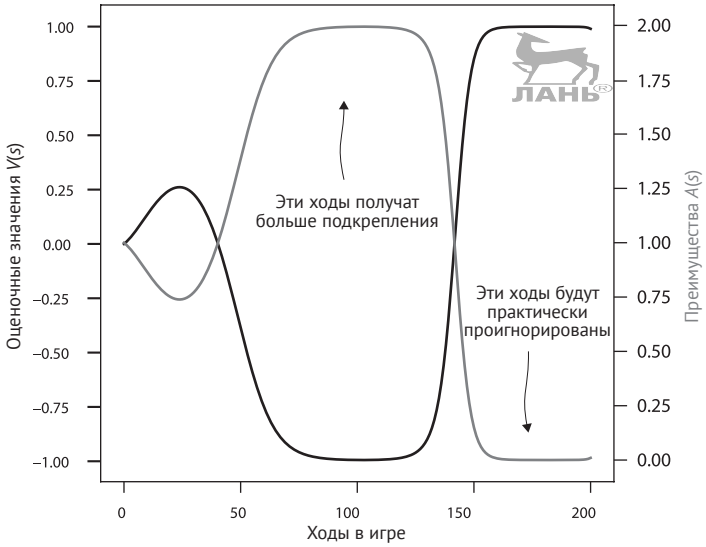


Рис. 12.2 ❖ Преимущество для каждого хода гипотетической партии. Обучающийся агент выиграл игру, поэтому его итоговая награда составляет 1. Ходы, которые позволили ему переломить ситуацию, имеют значение преимущества около 2, поэтому они получат больше подкрепления в процессе обучения. Ходы, совершенные ближе к концу партии, когда ее исход уже ясен, имеют значение преимущества около 0, поэтому в процессе обучения они будут практически проигнорированы

После того как агент вырвался вперед примерно на 160-м ходу, его решения больше не являются интересными, поскольку исход игры уже предопределен. Значение преимущества на этом участке близко к 0.

Далее в этой главе мы поговорим о настройке процесса обучения на основании значений преимущества. Но перед этим порассуждаем о том, как вычислять и сохранять это значение в процессе игры бота с самим собой.

12.1.2. Вычисление преимущества в процессе игры бота с самим собой

Для вычисления преимущества мы обновим объект `ExperienceCollector`, созданный в главе 9. Изначально буфер данных опыта отслеживал три параллельных массива, содержащих состояния, действия и награды. Для отслеживания значений преимущества можно добавить четвертый параллельный массив. Чтобы заполнить этот массив, нам потребуется как оценочное значение для каждого состояния, так и итоговый результат игры. Итоговый результат мы получим только в конце игры,

поэтому в середине эпизода можем накапливать оценочные значения, а по завершении партии преобразовать их в преимущества.

Листинг 12.1 ❖ Обновление объекта `ExperienceCollector` для отслеживания значений преимущества

```
class ExperienceCollector:
    def __init__(self):
        self.states = []
        self.actions = []
        self.rewards = []
        self.advantages = []
        self._current_episode_states = []
        self._current_episode_actions = []
        self._current_episode_estimated_values = []
```

Могут охватывать множество эпизодов.

Сбрасываются в конце каждого эпизода.

Точно так же нам нужно обновить метод `record_decision` для принятия оценочного значения наряду с состоянием и действием.

Листинг 12.2 ❖ Обновление `ExperienceCollector` для сохранения оценочных значений

```
class ExperienceCollector:
...
    def record_decision(self, state, action,
            estimated_value=0):
        self._current_episode_states.append(state)
        self._current_episode_actions.append(action)
        self._current_episode_estimated_values.append(
            estimated_value)
```

Теперь в методе `complete_episode` мы можем вычислить значение преимущества для каждого решения, принятого агентом.

Листинг 12.3 ❖ Вычисление преимущества в конце эпизода

```
class ExperienceCollector:
...
    def complete_episode(self, reward):
        num_states = len(self._current_episode_states)
        self.states += self._current_episode_states
        self.actions += self._current_episode_actions
        self.rewards += [reward for _ in range(num_states)]

        for i in range(num_states):
            advantage = reward - \
                self._current_episode_estimated_values[i]
            self.advantages.append(advantage)

        self._current_episode_states = []
        self._current_episode_actions = []
        self._current_episode_estimated_values = []
```

Вычисление преимущества каждого решения.

Сброс буферов для эпизодов.

Нам также требуется обновить класс `ExperienceBuffer` и вспомогательный метод `combine_experience` для обработки значений преимущества.

Листинг 12.4 ❖ Добавление преимущества в структуру `ExperienceBuffer`

```
class ExperienceBuffer:
    def __init__(self, states, actions, rewards, advantages):
        self.states = states
        self.actions = actions
        self.rewards = rewards
        self.advantages = advantages

    def serialize(self, h5file):
        h5file.create_group('experience')
        h5file['experience'].create_dataset('states',
data=self.states)
        h5file['experience'].create_dataset('actions',
data=self.actions)
        h5file['experience'].create_dataset('rewards',
data=self.rewards)
        h5file['experience'].create_dataset('advantages',
data=self.advantages)

def combine_experience(collectors):
    combined_states = np.concatenate(
       [np.array(c.states) for c in collectors])
    combined_actions = np.concatenate(
       [np.array(c.actions) for c in collectors])
    combined_rewards = np.concatenate(
       [np.array(c.rewards) for c in collectors])
    combined_advantages = np.concatenate([
        np.array(c.advantages) for c in collectors])

    return ExperienceBuffer(
        combined_states,
        combined_actions,
        combined_rewards,
        combined_advantages)
```

Теперь наши классы `experience` готовы к отслеживанию значений преимущества. Эти классы по-прежнему можно применять с методами, не учитывающими преимущества. При этом просто проигнорируйте содержимое буфера `advantages` в процессе обучения.

12.2. Создание нейронной сети для обучения методом «актор–критик»

В главе 11 мы говорили о том, как определить нейронную сеть с двумя входами средствами Keras. Сеть, предназначенная для применения алгоритма Q-обучения, предусматривала один вход для состояния доски и один вход для предложенного хода. Для применения алгоритма типа «актор–критик» нам требуется сеть с одним входом и двумя выходами. Входом будет являться представление состояния доски. Одним из выходов будет распределение вероятностей для ходов – актор. Другим выходом будет ожидаемая выгода от текущего положения – критик.

Создание сети с двумя выходами обеспечивает неожиданный бонус: каждый выход служит своего рода регуляризатором для другого. (В главе 6 мы говорили о том, что *регуляризация* предотвращает *переобучение* модели, при котором она специализируется на наборе обучающих данных.) Представьте, что для группы камней на доске существует угроза захвата. Этот факт имеет значение для выхода ценности, поскольку игрок со слабыми камнями, вероятно, является отстающим. Это также имеет значение для выхода действия, поскольку вы, вероятно, хотите атаковать или защищать слабые камни. Если ваша сеть сформирует детектор «слабых камней» в начальных слоях, это будет иметь значение для обоих выходов. Обучение на основе обоих выходов заставляет сеть изучить представление, которое оказывается полезным для достижения обеих целей. Это часто помогает улучшить качество обобщения, а иногда даже ускорить процесс обучения.

В главе 11 был представлен функциональный API Keras, который позволяет соединять слои сети любым способом. Мы используем его в этой главе для создания сети, представленной на рис. 12.3. Поместите соответствующий код в файл *init_ac_agent.py*.

Листинг 12.5 ❖ Сеть с двумя выходами – `policy` и `value`

```
from keras.models import Model
from keras.layers import Conv2D, Dense, Flatten, Input

board_input = Input(shape=encoder.shape(), name='board_input')

conv1 = Conv2D(64, (3, 3),
               padding='same',
               activation='relu')(board_input)
conv2 = Conv2D(64, (3, 3),
               padding='same',
               activation='relu')(conv1)
conv3 = Conv2D(64, (3, 3),
               padding='same',
               activation='relu')(conv2)

flat = Flatten()(conv3)
processed_board = Dense(512)(flat)

policy_hidden_layer = Dense(
    512, activation='relu')(processed_board)
policy_output = Dense(
    encoder.num_points(), activation='softmax')(
    policy_hidden_layer)

value_hidden_layer = Dense(
    512, activation='relu')(
    processed_board)
value_output = Dense(1, activation='tanh')(
    value_hidden_layer)

model = Model(inputs=board_input,
  outputs=[policy_output, value_output])
```

Добавьте нужное количество сверточных слоев.

В этом примере используются скрытые слои размером 512. Поэкспериментируйте с размерами, чтобы выбрать подходящий. Три скрытых слоя не обязательно должны быть одного размера.

Этот выход соответствует функции политики.

Этот выход соответствует функции ценности.

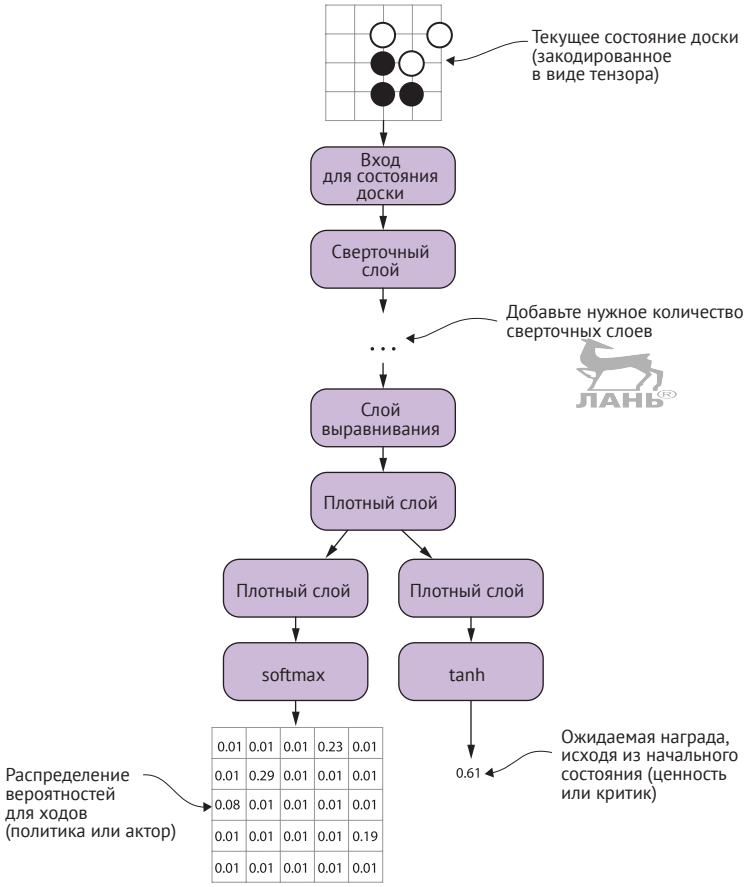


Рис. 12.3 ❖ Нейронная сеть для игры в го, обучаемая методом типа «актор–критик». Эта сеть имеет один вход, на который подается представление текущего состояния доски, и два выхода. Один выход показывает, какой ход сеть должна совершить, – это выход функции политики, или актор. Другой выход определяет лидирующего игрока – это выход функции ценности, или критик. Критик не используется в процессе игры, но оказывается полезным в процессе обучения

Эта сеть содержит три сверточных слоя с 64 фильтрами в каждом. Это мало для сети, предназначенной для игры в го, но позволяет ускорить процесс обучения. Как обычно, мы рекомендуем вам поэкспериментировать с различными сетевыми структурами.

Выход функции политики представляет собой распределение вероятностей для возможных ходов. Размерность соответствует количеству точек доски, а функция активации softmax гарантирует, что сумма всех выходных сигналов равна 1.

Выход функции ценности представляет собой единственное значение в диапазоне от –1 до 1. Этот выход имеет размерность 1, а в качестве функции активации используется `tanh`.

12.3. Игра с агентом типа «актор–критик»

В данном случае процесс выбора ходов практически тот же, что и в случае с обучением агента на основе политики из главы 10. Нам требуется внести лишь два изменения. Во-первых, поскольку теперь модель предусматривает два выхода, вам нужен дополнительный код для распаковки результатов. Во-вторых, вам необходимо передать объекту `ExperienceCollector` оценочное значение, а также данные о состоянии и действии. Процесс выбора хода из распределения вероятностей остается прежним. Следующий листинг содержит обновленную реализацию функции `select_move`. Мы отметили те места, в которых данная реализация отличается от кода, приведенного в главе 10.

Листинг 12.6 ❖ Процесс выбора хода агентом типа «актор–критик»

```
class ACAgent(Agent):
...
    def select_move(self, game_state):
        num_moves = self.encoder.board_width * \
self.encoder.board_height

        board_tensor = self.encoder.encode(game_state)
        X = np.array([board_tensor])

        actions, values = self.model.predict(X)
        move_probs = actions[0]
        estimated_value = values[0][0]

        eps = 1e-6
        move_probs = np.clip(move_probs, eps, 1 - eps)
        move_probs = move_probs / np.sum(move_probs)

        candidates = np.arange(num_moves)
        ranked_moves = np.random.choice(
            candidates, num_moves, replace=False, p=move_probs)
        for point_idx in ranked_moves:
            point = self.encoder.decode_point_index(point_idx)
            move = goboard.Move.play(point)
            move_is_valid = game_state.is_valid_move(move)
            fills_own_eye = is_point_an_eye(
                game_state.board, point,
game_state.next_player)
            if move_is_valid and (not fills_own_eye):
                if self.collector is not None:
                    self.collector.record_decision(
                        state=board_tensor,
                        action=point_idx,
                        estimated_value=estimated_value
                    )
                return goboard.Move.play(point)
        return goboard.Move.pass_turn()
```

Поскольку эта модель предусматривает два выхода, функция `predict` возвращает кортеж с двумя массивами NumPy.

Вызов функции `predict` – это вызов пакета, позволяющий одновременно обрабатывать несколько состояний доски, поэтому для получения желаемого распределения вероятностей вам необходимо выбрать первый элемент массива.

Значения представлены в виде одномерного вектора, поэтому для получения значения в виде простого числа с плавающей запятой вы должны извлечь первый элемент.

Включение оценочного значения в буфер с данными опыта.

12.4. Обучение агента типа «актор–критик» на данных опыта

Обучение сети с помощью алгоритма типа «актор–критик» напоминает сочетание методов обучения на основе политики (см. главу 10) и на основе ценности действия (см. главу 11). Для обучения сети с двумя выходами мы создадим для каждого выхода отдельные цели обучения и применим к ним разные функции потерь. В этом разделе мы поговорим о преобразовании данных опыта в цели обучения, а также о применении функции `fit` библиотеки Keras к модели с несколькими выходами.

Вспомните, как выполнялось кодирование обучающих данных при использовании метода градиента политики. Для любой игровой позиции целью обучения был вектор размером с доску, содержащий значение 1 или –1 в ячейке, соответствующей выбранному ходу (значение 1 соответствовало выигрышу, а –1 – проигрышу). При обучении методом типа «актор–критик» используется та же схема кодирования обучающих данных, только вместо 1 и –1 применяются значения преимущества хода. Преимущество будет иметь такой же знак, как и итоговая награда, поэтому вероятность того или иного решения будет смещаться в том же направлении, что и при использовании алгоритма обучения на основе политики. Однако в данном случае она сместится сильнее для более важных действий и лишь немного для действий с преимуществом, близким к 0.

Для выхода функции ценности целью обучения является совокупная награда, как и в случае с алгоритмом Q-обучения. Весь процесс представлен на рис. 12.4.

При наличии нескольких выходов в сети для каждого из них можно выбрать разные функции потерь. Для выхода функции политики мы будем использовать категориальную перекрестную энтропию, а для выхода функции ценности – среднеквадратическую ошибку. (Объяснение целесообразности применения этих функций потерь для решения данных задач можно найти в главах 10 и 11.)

Пришло время познакомиться с новой функцией Keras под названием *вес потерь*. По умолчанию Keras суммирует значения функции потерь для всех выходов, чтобы получить общее значение потери. При указании весов потерь Keras отмасштабирует значения отдельных функций потерь перед их суммированием. Это позволяет варьировать относительную важность каждого выхода. В ходе наших экспериментов мы обнаружили, что значение потери для выхода функции ценности значительно превышает значение потери для выхода функции политики, поэтому мы уменьшили его вдвое. В зависимости от структуры вашей сети и обучающих данных вам может потребоваться немного скорректировать веса потери.

При каждом вызове функции `fit` Keras будет выводить на экран вычисленные значения потери. В случае сети с двумя выходами Keras выведет два отдельных значения потери. Это позволяет сопоставить их величины. Если одно значение потери сильно превышает другое, рассмотрите возможность коррекции весов. Не стремитесь к слишком большой точности.

В следующем листинге показано, как закодировать данные опыта в виде обучающих данных, а затем применить функцию `fit` к целям обучения. Представленная структура аналогична реализациям метода `train` из глав 10 и 11.

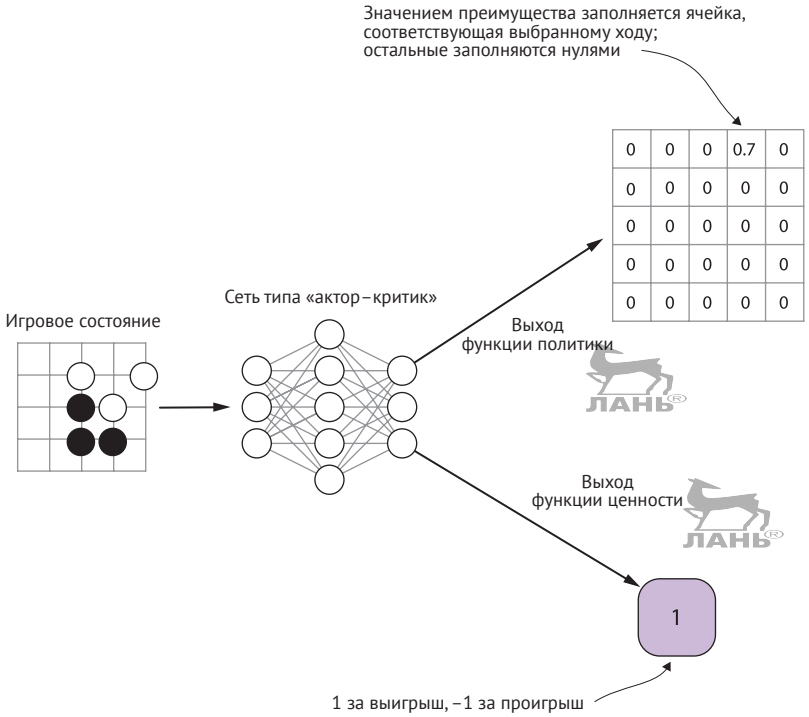


Рис. 12.4 ❖ Процесс обучения с использованием алгоритма типа «актор–критик». Нейронная сеть имеет два выхода: один для функции политики и один для функции ценности. Каждый из них предусматривает собственную цель обучения. Для выхода политики это вектор размером с доску. Ячейка вектора, соответствующая выбранному ходу, содержит значение его преимущества, остальные заполнены нулями. Для выхода ценности целью является итоговый результат игры

Листинг 12.7 ❖ Процесс выбора хода агентом типа «актор–критик»

```
class ACAgent(Agent):
...
    def train(self, experience, lr=0.1, batch_size=128):
        opt = SGD(lr=lr)
        self.model.compile(
            optimizer=opt,
            loss=['categorical_crossentropy', 'mse'],
    loss_weights=[1.0, 0.5])

        n = experience.states.shape[0]
        num_moves = self.encoder.num_points()
        policy_target = np.zeros((n, num_moves))
```

`categorical_crossentropy` – функция потери для выхода политики, как в главе 10. `mse` (среднеквадратичная ошибка) – функция потери для выхода ценности, как в главе 11. В данном случае порядок соответствует порядку в конструкторе Model из листинга 12.5.

`lr` (скорость обучения) и `batch_size` – это параметры для настройки оптимизатора (см. главу 10).

К выходу политики применен вес 1.0, а к выходу ценности – 0.5.

```
value_target = np.zeros((n,))
for i in range(n):
    action = experience.actions[i]
    policy_target[i][action] = experience.advantages[i]
    reward = experience.rewards[i]
    value_target[i] = reward

self.model.fit(
    experience.states,
    [policy_target, value_target],
    batch_size=batch_size,
    epochs=1)
```

Эта схема кодирования аналогична схеме, использованной в главе 10, но взвешена с учетом преимущества.

Эта схема кодирования аналогична схеме, использованной в главе 11.

Теперь, когда мы подготовили все необходимые компоненты, давайте попробуем применить алгоритм обучения типа «актор–критик». Чтобы ускорить получение результатов, мы начнем с доски размером 9×9. Цикл обучения будет включать следующие этапы:

1) генерация игр бота с самим собой партиями по 5000 игр;
2) после генерации каждой партии игр обучайте агента и сравнивайте его с предыдущей версией;
3) если новый бот сумеет обыграть предыдущего в 60 играх из 100, можете считать, что агент улучшен! Начните процесс заново с использованием нового бота;
4) если обновленный бот победит менее чем в 60 играх из 100, сгенерируйте еще одну партию игр бота с самим собой и переобучите агента. Продолжайте процесс обучения до тех пор, пока новый бот не станет достаточно сильным.

Контрольный показатель «60 побед из 100» выбран произвольно; это хорошее круглое число, позволяющее с достаточной степенью уверенности судить о прогрессе бота.

Начните с инициализации бота с помощью сценария *init_ac_agent.py* (как показано в листинге 12.5):

```
python init_ac_agent.py --board-size 9 ac_v1.hdf5
```

После этого у вас появится новый файл *ac_v1.hdf5*, содержащий веса для нового бота. На данном этапе бот будет выбирать ходы и определять их ценность практически случайным образом. Теперь вы можете приступить к генерации игр бота с самим собой:

```
python self_play_ac.py \
--board-size 9 \
--learning-agent ac_v1.hdf5 \
--num-games 5000 \
--experience-out exp_0001.hdf5
```

Если вы не являетесь счастливым обладателем быстрого графического процессора, сейчас самое время пойти выпить кофе или выгулять собаку. Результат выполнения сценария `self_play` будет выглядеть примерно так:

```
Simulating game 1/5000...
 9 ooxxxxxxx
 8 ooox.xx.x
 7 oxxxxooxx
 6 oxxxxxox.
 5 ooooooxxx
 4 ooo.oooxo
 3 ooooooooo
 2 .oo.ooo.o
 1 ooooooooo
   ABCDEFGHJ
W+28.5
...
Simulating game 5000/5000...
 9 x.x.xxxxx
 8 xxxxx.xxx
 7 .x.xxxxoo
 6 xxxx.xo.o
 5 xxxxxxooo
 4 xooooooxo
 3 xoooxxxxo
 2 o.o.oxxxx
 1 ooooox.x.
   ABCDEFGHJ
B+15.5
```

После этого у вас должен появиться файл *exp_0001.hdf5*, содержащий значительное количество записей партий. Следующим шагом является обучение:

```
python train_ac.py \
--learning-agent bots/ac_v1.hdf5 \
--agent-out bots/ac_v2.hdf5 \
--lr 0.01 --bs 1024 \
exp_0001.hdf5
```

Этот код задействует нейронную сеть из файла *ac_v1.hdf5*, проведет одну эпоху обучения на основе данных из файла *exp_0001.hdf5* и сохранит обновленную версию агента в файле *ac_v2.hdf5*. Оптимизатор будет использовать скорость обучения 0,01 и размер пакета 1024. На выходе должно получиться что-то вроде этого:

```
Epoch 1/1
574234/574234 [==============================] - 15s 26us/step - loss:
➥ 1.0277 - dense_3_loss: 0.6403 - dense_5_loss: 0.7750
```

Обратите внимание на то, что теперь потеря разбита на два значения: `dense_3_loss` и `dense_5_loss` для выходов политики и ценности соответственно.

Теперь мы можем сравнить обновленного бота с его предыдущей версией с помощью сценария *eval_ac_bot.py*:

```
python eval_ac_bot.py \
--agent1 bots/ac_v2.hdf5 \
--agent2 bots/ac_v1.hdf5 \
--num-games 100
```

На выходе должно получиться что-то вроде этого:

```
...
Simulating game 100/100...
 9 oooxxxxx.
 8 .oox.xxxx
 7 ooxxxxxxx
 6 .oxx.xxxx
 5 oooxxx.xx
 4 o.ox.xx.x
 3 ooxxxxxxx
 2 ooxx.xxxx
 1 oxxxxxxx.
   ABCDEFGHJ
B+31.5
Agent 1 record: 60/100
```

Данный результат точно соответствует пороговому значению: 60 побед из 100, поэтому мы можем быть уверены в том, что наш бот научился чему-то полезному. (Разумеется, это всего лишь пример. Ваш фактический результат будет иным, и это нормально.) Поскольку бот ac_v2 заметно сильнее, чем ac_v1, вы можете приступить к генерации игр, используя версию ac_v2:

```
python self_play_ac.py \
--board-size 9 \
--learning-agent ac_v2.hdf5 \
--num-games 5000 \
--experience-out exp_0002.hdf5
```

После этого вы сможете снова приступить к процессу обучения и оценки:

```
python train_ac.py \
--learning-agent bots/ac_v2.hdf5 \
--agent-out bots/ac_v3.hdf5 \
--lr 0.01 --bs 1024 \
exp_0002.hdf5
python eval_ac_bot.py \
--agent1 bots/ac_v3.hdf5 \
--agent2 bots/ac_v2.hdf5 \
--num-games 100
```

На этот раз результат оказался менее впечатляющим:

```
Agent 1 record: 51/100
```

Бот ac_v3 обыграл бота ac_v2 только в 51 партии из 100. Этот результат не позволяет точно оценить его прогресс. Мы можем сделать вывод, что силы обоих ботов примерно равны. Однако не отчаивайтесь. Сгенерируйте новые обучающие данные и повторите попытку:

```
python self_play_ac.py \
--board-size 9 \
--learning-agent ac_v2.hdf5 \
--num-games 5000 \
--experience-out exp_0002a.hdf5
```

Сценарий *train_ac.py* примет несколько файлов с обучающими данными в командной строке:


```
python train_ac.py \
--learning-agent ac_v2.hdf5 \
--agent-out ac_v3.hdf5 \
--lr 0.01 --bs 1024 \
exp_0002.hdf5 exp_0002a.hdf5
```

После генерации каждой следующей партии игр вы можете снова сравнить бота с версией ac_v2. В ходе нашего эксперимента для достижения удовлетворительного результата нам потребовалось три партии по 5000 игр, т. е. всего 15 000 игр:

```
Agent 1 record: 62/100
```

Успех! Бот ac_v3 победил ac_v2 в 62 играх из 100, что является достаточным доказательством его превосходства. Теперь мы можем сгенерировать партию игр бота ac_v3 с самим собой и повторить цикл снова.

Не вполне ясно, какого уровня может достичь бот при использовании только этой реализации алгоритма «актор–критик». Мы показали, что благодаря ей бот может освоить базовые тактические приемы, однако на каком-то этапе его прогресс остановится. Для создания бота, превосходящего человека, необходима глубокая интеграция методов обучения с подкреплением и алгоритма поиска по дереву, речь о которой пойдет в главе 14.

12.5. Резюме

- Обучение с помощью *алгоритма типа «актор–критик»* – это метод обучения с подкреплением, предполагающий одновременную реализацию как функции политики, так и функции ценности. Функция политики отвечает за принятие решений, а функция ценности помогает улучшить процесс обучения. Алгоритм типа «актор–критик» применяется к тем же задачам, что и обучение на основе градиента политики, но часто обеспечивает большую стабильность.
- *Преимущество* – это разница между фактической наградой, полученной агентом, и наградой, которую он ожидал получить в какой-то момент эпизода. В случае игр это разница между фактическим результатом партии (выигрыш или проигрыш) и ожидаемым значением (оцененным моделью агента).
- Преимущество позволяет выявить важные решения в игре. Если обучающийся агент выиграет игру, наибольшим преимуществом будут обладать ходы, совершенные из ничейной или проигрышной позиции. Ходы, совершенные агентом после того, как исход игры стал ясен, будут обладать практически нулевым преимуществом.
- Последовательная сеть Keras может иметь несколько выходов. При использовании алгоритма типа «актор–критик» можно создать одну сеть для моделирования как функции политики, так и функции ценности.

Часть III

БОЛЬШЕ, ЧЕМ СУММА ВСЕХ ЧАСТЕЙ

К этому моменту вы познакомились с рядом методов разработки ИИ, основанных на классическом алгоритме поиска по дереву, машинном обучении и обучении с подкреплением. Каждый из них имеет свои преимущества, но у каждого есть и ограничения. Для создания по-настоящему мощного ИИ для игры в го необходимо объединить полученные знания. Интеграция всех этих компонентов – это серьезная инженерная задача. В этой части описана архитектура программы AlphaGo, которая потрясла мир искусственного интеллекта и игры в го! В заключение вы узнаете об элегантном и простом дизайне программы AlphaGo Zero, которая является самой сильной версией AlphaGo на сегодняшний день.

Глава 13

AlphaGo: собираем все вместе

В этой главе:

- изучение принципов, позволивших ботам для игры в го превзойти человека;
- создание такого бота с помощью алгоритмов поиска по дереву, глубокого обучения с учителем и обучения с подкреплением;
- реализация собственной версии движка AlphaGo компании DeepMind.

Когда в 2016 году бот AlphaGo компании DeepMind совершил 37-й ход во второй партии против Ли Седоля, мир го был поражен. Комментатор Майкл Редмонд, профессиональный игрок, сыгравший тысячи игр на высшем уровне, задержал дыхание. Он даже сначала снял камень с демонстрационной доски и оглянулся по сторонам, пытаясь убедиться в том, что AlphaGo сделал правильный ход. («Я до сих пор не понимаю, как это было сделано», – заявил Редмонд на следующий день в интервью ресурсу American Go EJournal.) Ли, мировой лидер прошлого десятилетия, потратил 12 минут на изучение доски, прежде чем ответить. Этот легендарный ход показан на рис. 13.1.

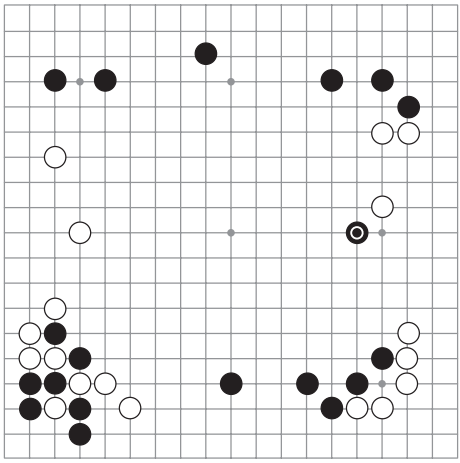


Рис. 13.1 ❖ Легендарный «удар в плечо», совершенный ботом AlphaGo во второй игре серии против Ли Седоля, ошеломил многих профессиональных игроков

Этот ход противоречит общепринятой теории го. Диагональный ход под названием «удар в плечо» побуждает игрока белыми к выстраиванию сплошной стены из камней. Если белый камень находится на третьей линии, а черный – на четвертой, обмен считается равноценным: белые зарабатывают очки у края доски, а черные расширяют свое влияние в центре. Но когда белый камень находится на четвертой линии, стена блокирует слишком большую территорию. (Мы приносим извинения опытным игрокам за столь сильное упрощение.) Удар в плечо, совершенный на пятой линии, выглядит по-дилетантски, по крайней мере так было до тех пор, пока «Профессор Альфа» не обыграл легенду в четырех играх из пяти. Удар в плечо стал одним из многих сюрпризов, преподнесенных ботом AlphaGo. Спустя год все игроки, начиная с профессионалов и заканчивая любителями, уже экспериментировали с ходами, совершенными AlphaGo.

В этой главе вы узнаете о том, как работает AlphaGo, в ходе реализации всех его компонентов. AlphaGo предполагает использование комбинации алгоритмов глубокого обучения с учителем на записях партий, сыгранных профессионалами (см. главы 5–8), глубокого обучения с подкреплением на данных партий, сыгранных ботом с самим собой (см. главы 9–12), а также определенного способа применения этих глубоких сетей для совершенствования алгоритма поиска по дереву. Вы, вероятно, удивитесь, когда поймете, как много вы уже знаете о компонентах AlphaGo. Если конкретнее, то система AlphaGo, с которой нам предстоит познакомиться, работает следующим образом:

- мы начинаем с обучения *двух* глубоких сверточных нейронных сетей (*сетей политики*) для предсказания ходов. Одна из этих сетей будет обладать более глубокой архитектурой и давать *более точные результаты*, архитектура другой сети будет менее глубокой, но ее *оценка будет проходить быстрее*. Они будут называться *сильной* и *быстрой* сетями политики соответственно;
- сильная и быстрая сети политики используют несколько более изощренный кодировщик доски с 48 плоскостями признаков. Кроме того, они характеризуются более глубокой архитектурой по сравнению с сетями, которые мы обсуждали в главах 6 и 7. Все остальное, скорее всего, покажется вам знакомым. Архитектура сетей политики AlphaGo описана в разделе 13.1;
- после завершения первого этапа обучения этих сетей мы используем сильную сеть политики в качестве отправной точки для обучения бота на основе игры с самим собой (см. раздел 13.2). При наличии достаточного объема вычислительных мощностей это позволит достичь значительного прогресса;
- далее на основе сильной сети, обученной на предыдущем этапе, мы создадим *сеть ценности* (см. раздел 13.3). На этом мы закончим процесс глубокого обучения сетей;
- для игры в го мы применим алгоритм поиска по дереву, однако вместо простых развертываний Монте-Карло (см. главу 4) будем использовать быструю сеть политики для исследования дальнейших ходов. Кроме того, мы сбалансируем результат поиска по дереву значением, предоставленным функцией ценности. Это нововведение будет подробно описано в разделе 13.4;

- реализация всего этого процесса, начиная с обучения сетей политики и игр бота с самим собой и заканчивая проведением игр на сверхчеловеческом уровне, требует огромных вычислительных ресурсов и времени. В разделе 13.5 вы получите некоторое представление о том, чем обусловлена мощь алгоритма AlphaGo и чего можно ожидать от проведения собственных экспериментов.

На рис. 13.2 описанный выше процесс представлен в виде схемы, компоненты которой мы подробно обсудим далее в главе.

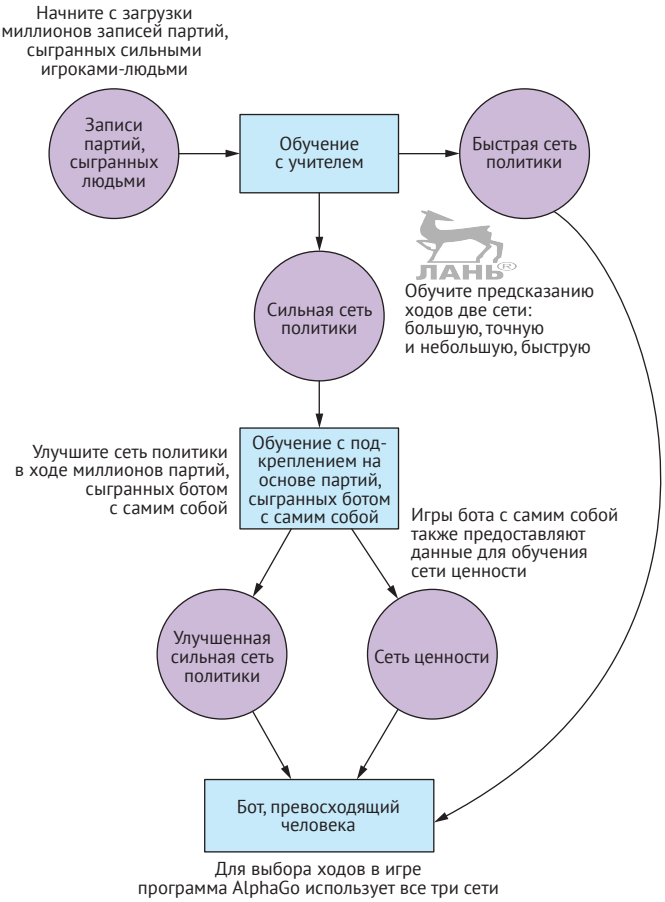


Рис. 13.2 ❖ Процесс обучения трех нейронных сетей, лежащих в основе алгоритма AlphaGo. После сбора записей партий, сыгранных людьми, мы можем обучить предсказанию ходов две нейронные сети: небольшую, быструю и большую, сильную. Затем с помощью обучения с подкреплением можно дополнительно повысить уровень игры большой сети. Игры, сыгранные ботом с самим собой, также предоставляют данные для дальнейшего обучения. Затем программа AlphaGo задействует все три сети для реализации алгоритма поиска по дереву, который обеспечивает невероятно высокий уровень игры бота

13.1. Обучение глубоких нейронных сетей для создания бота AlphaGo

Выше мы говорили о том, что программа AlphaGo использует три нейронные сети: две сети политики и одну сеть ценности. Несмотря на кажущуюся сложность, эти сети и подаваемые на их вход признаки концептуально близки. Вы, вероятно, удивитесь, когда поймете, чему уже научились в ходе чтения глав с 5 по 12. Прежде чем мы перейдем к обсуждению процесса создания и обучения этих нейронных сетей, давайте кратко рассмотрим их роль в системе AlphaGo.

- **Быстрая сеть политики.** Эта сеть для предсказания ходов в игре го сопоставима по размеру с сетями, описанными в главах 7 и 8. Она предназначена не для максимально точного, а, скорее, для максимально быстрого и достаточно точного предсказания ходов. Эта сеть используется в разделе 13.4 для выполнения развертываний с использованием поиска по дереву, и в главе 4 мы говорили о том, что нам нужно осуществить множество таких развертываний, чтобы применение данного алгоритма стало целесообразным. Мы уделим немного меньше внимания этой сети и сосредоточимся на следующих двух.
- **Сильная сеть политики.** Данная сеть больше ориентирована не на скорость, а на верность предсказания ходов. Эта сверточная сеть имеет более глубокую архитектуру по сравнению со своей быстрой версией и более чем в два раза превосходит ее в плане верности предсказания ходов. Как и быстрая версия, эта сеть обучается на записях партий, сыгранных человеком, как это делалось в главе 7. После завершения обучения эта сильная сеть политики будет использована в качестве отправной точки для генерации игр бота с самим собой с применением методов обучения с подкреплением, описанных в главах 9 и 10. Этот шаг позволит сделать данную сеть политики еще более сильной.
- **Сеть ценности.** Игры сильной сети политики с самой собой позволяют сгенерировать новый набор данных для дальнейшего обучения сети ценности. В частности, мы используем результаты этих игр и приемы, описанные в главах 11 и 12, для реализации функции ценности. Важная роль полученной сети ценности будет описана в разделе 13.4.

13.1.1. Сетевые архитектуры, используемые в программе AlphaGo

Теперь, когда вы имеете некоторое представление о том, как каждая из трех глубоких нейронных сетей используется в AlphaGo, мы поговорим об их создании средствами языка Python и библиотеки Keras. Далее приведен краткий обзор сетевых архитектур, после ознакомления с которым мы перейдем к обсуждению кода. Терминологию, имеющую отношение к сверточным сетям, можно найти в главе 7.

- Сильная сеть политики представляет собой 13-слойную сверточную сеть. Все эти слои предусматривают фильтры размером 19×19, что позволяет сохранить исходный размер доски во всей сети. Для этого необходимо *дополнить входные данные нулями*, как это делалось в главе 7. Первый слой имеет ядро размером 5, а все следующие слои работают с ядром размером 3. Последний слой использует функцию активации softmax и имеет один выход-

ной фильтр, а первые 12 слоев используют функцию активации ReLU и имеют по 192 выходных фильтра каждый.

- Сеть ценности представляет собой сверточную сеть, состоящую из 16 слоев, первые 12 из которых являются *точной копией слоев сильной сети политики*. Слой 13 – это дополнительный сверточный слой, структурно идентичный слоям с 2 по 12. Слой 14 представляет собой сверточный слой с размером ядра 1 и одним выходным фильтром. Сеть завершается двумя плотными слоями: один имеет 256 выходов и функцию активации ReLU, а последний – один выход и функцию активации tahn.

Как видите, сеть политики и сеть ценности программы AlphaGo ничем не отличаются от глубоких сверточных нейронных сетей, описанных в главе 6. Схожесть этих двух сетей позволяет определить их в одной функции Python. Прежде чем это сделать, мы обсудим особенность Keras, которая значительно упрощает процесс определения сети. В главе 7 мы говорили о возможности дополнения входных изображений нулями с помощью вспомогательного слоя `ZeroPadding2D`. Можно сделать и так, однако для упрощения определения модели это дополнение лучше поручить слою `Conv2D`. Как в сети ценности, так и в сети политики мы хотим дополнить нулями входное изображение, подаваемое на каждый сверточный слой, чтобы размер выходных фильтров *совпадал* с размером входа (19×19). Например, вместо явного дополнения входных изображений 19×19, подаваемых на первый сверточный слой, до размера 23×23, чтобы следующий сверточный слой с размером ядра 5 создавал выходные фильтры 19×19, мы даем сверточному слою команду сохранить размер входного изображения. Для этого мы передаем аргумент `padding='same'` сверточному слою, который и позаботится о дополнении. С помощью этого удобного способа давайте определим первые 11 слоев, которые являются общими для сети политики и сети ценности AlphaGo. Это определение можно найти в файле *alphago.py*, находящемся в подмодуле *networks* модуля *dlgo* нашего репозитория GitHub.

Листинг 13.1 ❖ Инициализация нейронной сети для сетей политики и ценности AlphaGo

```
from keras.models import Sequential
from keras.layers.core import Dense, Flatten
from keras.layers.convolutional import Conv2D

def alphago_model(input_shape, is_policy_net=False,
                  num_filters=192,
                  first_kernel_size=5,
                  other_kernel_size=3):
    model = Sequential()
    model.add(
        Conv2D(num_filters, first_kernel_size, input_shape=input_shape,
     padding='same',
               data_format='channels_first', activation='relu'))

    for i in range(2, 12):
        model.add(
            Conv2D(num_filters, other_kernel_size, padding='same',
                   data_format='channels_first', activation='relu'))
```

Этот логический флаг позволяет указать тип нужной сети (сеть политики или сеть ценности).

Все сверточные слои, кроме последнего, имеют одинаковое количество фильтров.

Первый слой имеет ядро размером 5, все остальные – размером 3.

Первые 12 слоев сети политики и сети ценности программы AlphaGo являются одинаковыми.

Обратите внимание на то, что мы пока не указали форму входных данных для первого слоя. Дело в том, что у сетей политики и ценности эта форма будет разной. Вы поймете, в чем заключается эта разница, после знакомства с кодировщиком доски AlphaGo, который описан в следующем разделе. Для создания сильной сети политики в определение модели осталось добавить всего один сверточный слой.

Листинг 13.2 ❖ Создание сильной сети политики AlphaGo в Keras

```
if is_policy_net:
    model.add(
        Conv2D(filters=1, kernel_size=1, padding='same',
               data_format='channels_first', activation='softmax'))
    model.add(Flatten())
    return model
```

Как видите, мы добавляем последний слой `Flatten` для выравнивания предсказаний и обеспечения согласованности с предыдущим определением модели из глав 5–8.

Если вместо этого вы хотите возвратить сеть ценности AlphaGo, добавьте еще два слоя `Conv2D`, два слоя `Dense` и один слой `Flatten` для их соединения.

Листинг 13.3 ❖ Построение сети ценности AlphaGo в Keras

```
else:
    model.add(
        Conv2D(num_filters, other_kernel_size, padding='same',
               data_format='channels_first', activation='relu'))
    model.add(
        Conv2D(filters=1, kernel_size=1, padding='same',
               data_format='channels_first', activation='relu'))
    model.add(Flatten())
    model.add(Dense(256, activation='relu'))
    model.add(Dense(1, activation='tanh'))
    return model
```

Мы не обсуждаем здесь быструю сеть политики, поскольку ее архитектура и определение входных признаков технически сложны и мало что дают для понимания системы AlphaGo. Для проведения экспериментов вы можете смело использовать одну из сетей `small`, `medium` или `large` из подмодуля *networks* модуля *dlgo*. Основная идея состоит в том, что быстрая сеть политики имеет меньший размер по сравнению с сильной сетью, но обеспечивает более быструю оценку. Процесс обучения будет подробно описан в следующих разделах.

13.1.2. Кодировщик доски AlphaGo

Теперь, когда мы разобрались с сетевыми архитектурами, которые используются в системе AlphaGo, давайте обсудим соответствующий им способ кодирования данных доски. В главах 6 и 7 мы уже реализовали несколько кодировщиков вроде `oneplane`, `sevenplane` и `simple` и сохранили их в подмодуле *encoders* модуля *dlgo*. Плоскости признаков, используемые в AlphaGo, являются чуть более сложными по сравнению с виденными ранее, но они представляют собой естественное продолжение рассматриваемых до сих пор кодировщиков.

Кодировщик доски AlphaGo для сетей политики предусматривает 48 плоскостей признаков. Для создания сетей ценности мы добавляем еще одну дополнительную плоскость. Эти 48 плоскостей состоят из 11 концепций. Некоторые из них мы уже использовали, другие являются новыми. Далее мы обсудим каждую из концепций более подробно. В общем система AlphaGo чуть лучше учитывает специфические для игры в го тактические ситуации по сравнению с рассмотренными ранее версиями кодировщиков доски. Ярким примером этого является такая часть набора признаков, как концепция «лесенки» и ее «избегания» (см. рис. 13.3).

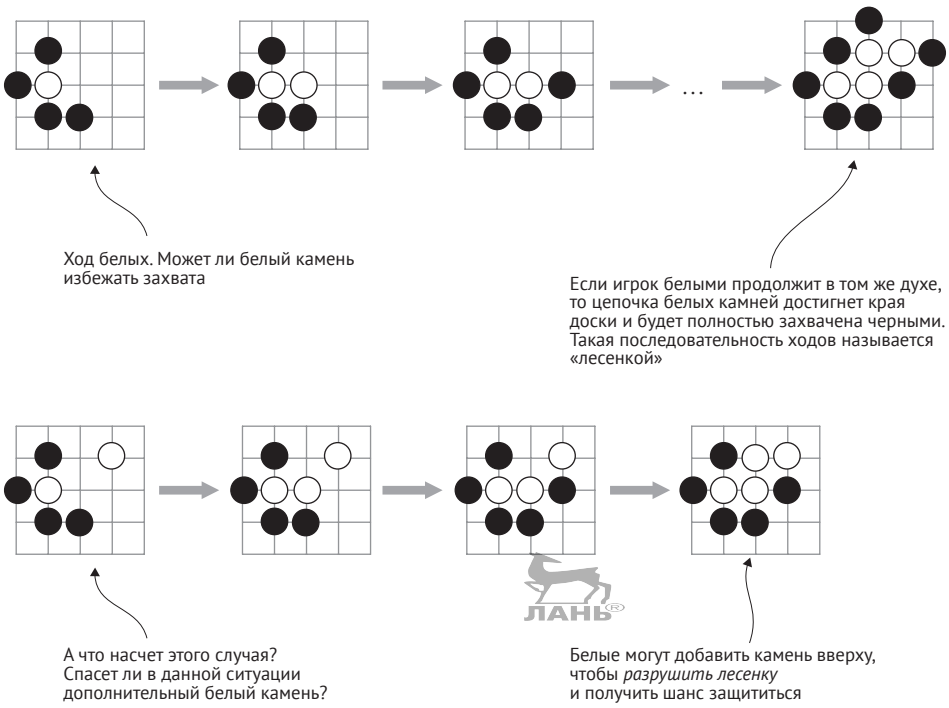


Рис. 13.3 ❖ В плоскостях признаков AlphaGo непосредственно закодировано множество тактических концепций игры го, в том числе «лесенка». В первом примере белый камень имеет только одну степень свободы, т. е. черные могут захватить его на следующем ходу. Игрок белыми продлевает цепочку белых камней, чтобы получить дополнительную свободу. Однако черные снова могут уменьшить количество степеней свободы белых камней до одной. Такая последовательность ходов продолжается вплоть до достижения края доски, в результате чего захватывается вся цепочка белых камней. С другой стороны, если на пути построения этой лесенки находится белый камень, то белые могут избежать захвата. Система AlphaGo предусматривает плоскость признаков, которая позволяет определить, окажется ли построение лесенки успешным

Во всех кодировщиках доски, в том числе в AlphaGo, используются так называемые *бинарные признаки*. Например, при захвате точек свободы (пустые соседние точки на доске) мы использовали не просто плоскость признаков, учитывающую количество степеней свободы каждого камня на доске, а бинарное представление с плоскостями, позволяющими указать на то, что камень имеет 1, 2, 3 или более степеней свободы. В AlphaGo реализована точно такая же идея, но

с восемью плоскостями признаков. В случае со степенями свободы это означает восемь плоскостей для указания 1, 2, 3, 4, 5, 6, 7 или, как минимум, 8 степеней свободы для конкретного камня.

Единственное принципиальное отличие от того, что делалось в главах 6–8, заключается в том, что AlphaGo явно кодирует цвет камня в *отдельных* плоскостях признаков. Напомним, что кодировщик `sevenplane` из главы 7 предусматривал плоскости для степеней свободы как черных, так и белых камней. В AlphaGo есть только один набор признаков для подсчета степеней свободы. Кроме того, все признаки выражены с точки зрения игрока, имеющего право следующего хода. Например, набор признаков «Количество захваченных камней», определяющий количество камней, которые можно захватить путем совершения конкретного хода, подсчитывает камни, которые мог бы захватить *текущий* игрок.

В табл. 13.1 представлены все признаки, используемые в AlphaGo. Первые 48 плоскостей предназначены для сетей политики, а последняя – только для сетей ценности.

Таблица 13.1. Плоскости признаков, используемые в AlphaGo

Название признака	Количество плоскостей	Описание
Цвет камней (Stone color)	3	Три плоскости признаков, соответствующие цвету камней, – по одной для текущего игрока, для его противника и для пустых точек на доске
Единицы (Ones)	1	Плоскость признаков, полностью заполненная единицами
Нули (Zeros)	1	Плоскость признаков, полностью заполненная нулями
Допустимость (Sensibleness)	1	Если ход является допустимым и не приводит к заполнению «глаз» текущего игрока, то он кодируется на этой плоскости значением 1, в противном случае – значением 0
Ходов с тех пор (Turns since)	8	Этот набор из восьми бинарных плоскостей показывает, сколькими ходами ранее был совершен тот или иной ход
Степени свободы (Liberties)	8	Количество степеней свободы цепочки камней, к которым относится этот ход, разделенное на восемь бинарных плоскостей
Степени свободы после совершения хода (Liberties after move)	8	Сколько степеней свободы останется в результате совершения данного хода?
Количество захваченных камней (Capture size)	8	Сколько камней противника можно захватить, совершив этот ход?
Количество камней в положении «атари» (Selfatari size)	8	Сколько из ваших камней в случае совершения этого хода окажутся в положении «атари» и смогут быть захвачены противником на следующем ходу?
Захват «лесенка» (Ladder capture)	1	Может ли этот камень быть захвачен с помощью «лесенки»?
Защита от захвата «лесенка» (Ladder escape)	1	Может ли этот камень избежать возможных «лесенок»?
Цвет камней текущего игрока (Current player color)	1	Эта плоскость заполняется единицами, если текущий игрок играет черными, и нулями – если белыми

Реализацию этих признаков можно найти в файле *alphago.py* в подмодуле *encoders* модуля *dlgo* нашего репозитория GitHub. Несмотря на то что реализация каждого набора признаков из табл. 13.1 не является особенно сложной, она не является и особенно интересной по сравнению с другими аспектами AlphaGo, которые нам предстоит обсудить. Однако реализация захватов «лесенка» имеет некоторые сложности, а кодирование количества ходов с момента совершения конкретного кода требует модификации определения доски для игры в го. Поэтому если вас интересуют подробности, обязательно ознакомьтесь с нашей реализацией на GitHub.

Давайте кратко рассмотрим способ инициализации кодировщика `AlphaGoEncoder` с целью его дальнейшего использования для обучения глубоких нейронных сетей. Мы передаем размер доски и логическое значение `use_player_plane`, которое указывает, нужно ли использовать 49-ю плоскость признаков (см. листинг 13.4).

Листинг 13.4 ❖ Сигнатура и инициализация кодировщика доски AlphaGo

```
class AlphaGoEncoder(Encoder):
    def __init__(self, board_size, use_player_plane=False):
        self.board_width, self.board_height = board_size
        self.use_player_plane = use_player_plane
        self.num_planes = 48 + use_player_plane
```

13.1.3. Обучение сетей политики в стиле AlphaGo

После подготовки сетевых архитектур и входных признаков можно приступать к обучению сетей политики для AlphaGo, процесс которого точно соответствует процедуре, описанной в главе 7, и включает такие этапы, как указание кодировщика доски и агента, загрузка данных игры го и обучение агентов на их основе (см. рис. 13.4). Тот факт, что мы используем чуть более сложные признаки и сети, ничего не меняет.

Чтобы инициализировать и обучить сильную сеть политики AlphaGo, сначала нужно создать экземпляр `AlphaGoEncoder` и два генератора данных игры го для обучения и тестирования, как это делалось в главе 7. Соответствующий код можно найти в файле *examples/alphago/alphago_policy_sl.py* репозитория GitHub.

Листинг 13.5 ❖ Загрузка данных для первого этапа обучения сети политики AlphaGo

```
from dlgo.data.parallel_processor import GoDataProcessor
from dlgo.encoders.alphago import AlphaGoEncoder
from dlgo.agent.predict import DeepLearningAgent
from dlgo.networks.alphago import alphago_model

from keras.callbacks import ModelCheckpoint
import h5py

rows, cols = 19, 19
num_classes = rows * cols
num_games = 10000

encoder = AlphaGoEncoder()
processor = GoDataProcessor(encoder=encoder.name())
generator = processor.load_go_data('train', num_games, use_generator=True)
test_generator = processor.load_go_data('test', num_games, use_generator=True)
```

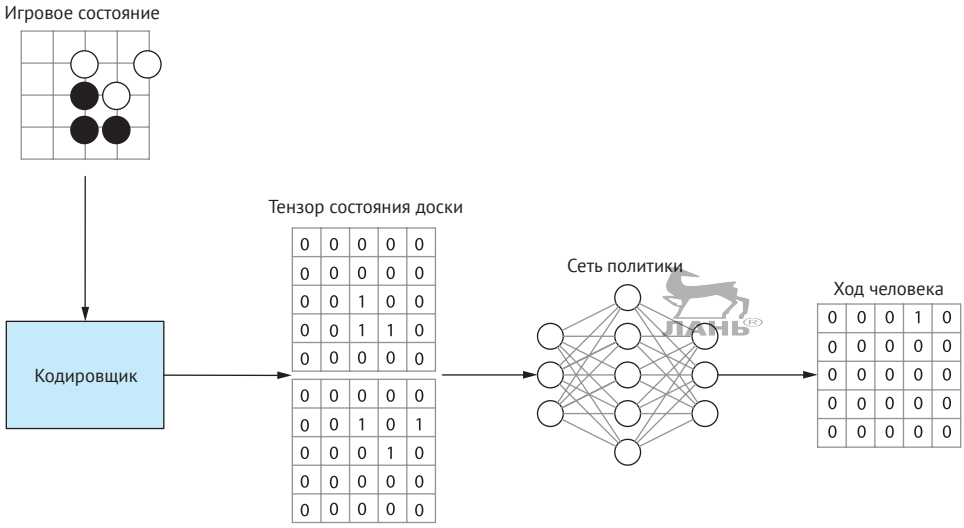


Рис. 13.4 ❖ Алгоритм обучения с учителем, применяемый к сетям политики AlphaGo, в точности соответствует процессу, описанному в главах 6 и 7. Мы воспроизводим партии, сыгранные человеком, восстанавливая игровые состояния. Каждое игровое состояние кодируется в виде тензора (на данной диаграмме показан тензор лишь с двумя плоскостями; AlphaGo использует 48 плоскостей). Целью обучения является вектор того же размера, что и доска, со значением 1 в точке, соответствующей фактическому ходу игрока-человека

Теперь можно загрузить сеть политики AlphaGo с помощью функции `alphago_model`, определенной ранее в этом разделе, и скомпилировать эту модель Keras с использованием категориальной перекрестной энтропии и стохастического градиентного спуска. Мы назовем эту модель *alphago_sl_policy* для указания на то, что к данной сети политики был применен алгоритм обучения с учителем (*sl*).

Листинг 13.6 ❖ Создание сети политики AlphaGo с помощью Keras

```
input_shape = (encoder.num_planes, rows, cols)
alphago_sl_policy = alphago_model(input_shape, is_policy_net=True)

alphago_sl_policy.compile('sgd', 'categorical_crossentropy', metrics=['accuracy'])
```

Для завершения первого этапа обучения осталось лишь вызвать функцию `fit_generator` этой сети политики, используя генераторы обучающих и тестовых данных, как это делалось в главе 7. Помимо применения более крупной сети и более изощренного кодировщика, это ничем не отличается от процесса, описанного в главах с 6 по 8.

После завершения обучения мы можем создать агента `DeepLearningAgent` на основе `model` и `encoder` и сохранить его для двух следующих этапов обучения, которые обсудим далее.

Листинг 13.7 ❖ Обучение и сохранение сети политики

```
epochs = 200
batch_size = 128
```

```
alphago_sl_policy.fit_generator(
    generator=generator.generate(batch_size, num_classes),
    epochs=epochs,
    steps_per_epoch=generator.get_num_samples() / batch_size,
    validation_data=test_generator.generate(batch_size, num_classes),
    validation_steps=test_generator.get_num_samples() / batch_size,
    callbacks=[ModelCheckpoint('alphago_sl_policy_{epoch}.h5')]
)

alphago_sl_agent = DeepLearningAgent(alphago_sl_policy, encoder)

with h5py.File('alphago_sl_policy.h5', 'w') as sl_agent_out:
    alphago_sl_agent.serialize(sl_agent_out)
```

Из соображений простоты в этой главе мы не будем отдельно обучать быструю и сильную сети политики AlphaGo. В качестве быстрой сети политики можно использовать *alphago_sl_agent*. В следующем разделе мы продемонстрируем способ применения этого агента в качестве отправной точки для обучения с подкреплением с целью улучшения сети политики.

13.2. Бутстрэппинг игр бота с самим собой из сетей политики

После обучения относительно сильного агента политики с помощью *alphago_sl_agent* мы можем провести серию его игр с самим собой, используя алгоритм градиента политики, описанный в главе 10. Как будет показано в разделе 13.5, программа AlphaGo компании DeepMind предполагает игру *разных версий сильной сети политики* против самой сильной на данный момент версии. Это предотвращает переобучение и обеспечивает более высокий уровень производительности, однако предложенный нами простой подход вполне передает идею использования игры агента с самим собой для улучшения сети политики.

Для проведения следующего этапа обучения мы сначала дважды загружаем сеть политики *alphago_sl_agent*: одна версия будет служить в качестве нового агента обучения с подкреплением под названием *alphago_rl_agent*, а другая – в качестве его противника. Соответствующий код можно найти в файле *examples/alphago/alphago_policy_sl.py* репозитория GitHub.

Листинг 13.8 ❖ Загрузка двух версий обученной сети политики для создания двух противников

```
from dlgo.agent.pg import PolicyAgent
from dlgo.agent.predict import load_prediction_agent
from dlgo.encoders.alphago import AlphaGoEncoder
from dlgo.rl.simulate import experience_simulation
import h5py

encoder = AlphaGoEncoder()

sl_agent = load_prediction_agent(h5py.File('alphago_sl_policy.h5'))
sl_opponent = load_prediction_agent(h5py.File('alphago_sl_policy.h5'))

alphago_rl_agent = PolicyAgent(sl_agent.model, encoder)
opponent = PolicyAgent(sl_opponent.model, encoder)
```

Теперь мы можем запустить игру этих двух агентов друг против друга и сохранить полученные данные опыта для дальнейшего обучения. На этих данных опыта мы обучим сеть *alphago_rl_agent*. Затем мы сохраним агента, к которому был применен алгоритм обучения с подкреплением, а также данные опыта, полученные в ходе игр агента с самим собой, для дальнейшего обучения сети ценности AlphaGo.

Листинг 13.9 ❖ Генерирование данных игры агента с самим собой для обучения агента `PolicyAgent`

```
num_games = 1000
experience = experience_simulation(num_games, alphago_rl_agent, opponent)

alphago_rl_agent.train(experience)

with h5py.File('alphago_rl_policy.h5', 'w') as rl_agent_out:
    alphago_rl_agent.serialize(rl_agent_out)

with h5py.File('alphago_rl_experience.h5', 'w') as exp_out:
    experience.serialize(exp_out)
```

Обратите внимание на то, что в этом примере используется служебная функция под названием `experience_simulation` из файла *simulate.py*, который содержится в подмодуле *rl* модуля *dlgo*. Ее реализацию можно найти на GitHub. Данная функция организует определенное количество игр (`num_games`) двух агентов друг против друга и возвращает данные опыта в виде `ExperienceCollector`, концепция которого обсуждалась в главе 9.

В 2016 году, когда была создана программа AlphaGo, самым сильным ботом для игры в го с открытым исходным кодом был *Pachi* (см. приложение B), уровень которого примерно соответствовал 2-му дану. Простое использование агента *alphago_rl_agent* для выбора ходов позволило AlphaGo одержать победу в 85 % игр против Pachi. Сверточные нейронные сети и раньше использовались для предсказания ходов в игре го, однако ни одному боту не удавалось победить более чем в 10 % игр против Pachi. Это иллюстрирует относительный прирост производительности глубоких нейронных сетей благодаря игре бота с самим собой по сравнению с использованием только алгоритма обучения с подкреплением. При проведении собственных экспериментов не ожидайте от своих ботов таких стартовых показателей. Маловероятно, что имеющиеся у вас вычислительные мощности позволят обеспечить подобные результаты.

13.3. Создание сети ценности на основе данных, полученных в ходе игры бота с самим собой

Третий и последний этап процесса подготовки сетей AlphaGo предполагает обучение сети ценности на *данных опыта, полученных в ходе игры бота с самим собой*, которые также использовались для обучения агента *alphago_rl_agent*. Структурно этот этап аналогичен предыдущему. Сначала мы инициализируем сеть ценности AlphaGo и создаем `ValueAgent` с кодировщиком доски AlphaGo. Код, соответствующий этому этапу обучения, можно найти в файле *examples/alphago/alphago_value.py* репозитория GitHub.

Листинг 13.10 ❖ Инициализация сети ценности AlphaGo

```
from dlgo.networks.alphago import alphago_model
from dlgo.encoders.alphago import AlphaGoEncoder
from dlgo.rl import ValueAgent, load_experience
import h5py

rows, cols = 19, 19
encoder = AlphaGoEncoder()
input_shape = (encoder.num_planes, rows, cols)
alphago_value_network = alphago_model(input_shape)

alphago_value = ValueAgent(alphago_value_network, encoder)
```

Теперь мы можем взять данные, полученные в ходе игры бота с самим собой, и обучить на их основе агента ценности, а затем сохранить его так же, как это делалось с двумя другими агентами до этого.

Листинг 13.11 ❖ Обучение сети ценности на данных опыта

```
experience = load_experience(h5py.File('alphago_rl_experience.h5', 'r'))

alphago_value.train(experience)

with h5py.File('alphago_value.h5', 'w') as value_agent_out:
    alphago_value.serialize(value_agent_out)
```

Если бы вы пробрались в помещение, где сотрудники DeepMind работали над AlphaGo (не стоит этого делать), предполагая, что они использовали библиотеку Keras для обучения своего бота так же, как и вы (а это не так), то, завладев параметрами обеих сетей политики и сети ценности, вы получили бы бота, играющего в го на сверхчеловеческом уровне. Разумеется, при условии что вы знаете о том, как правильно использовать эти три глубокие сети для реализации алгоритма поиска по дереву. В следующем разделе речь пойдет именно об этом.

13.4. Повышение эффективности поиска с помощью сетей политики и ценности

В главе 4 мы говорили о том, что применение поиска по дереву методом Монте-Карло к игре го предполагает создание дерева игровых состояний. Этот процесс состоит из четырех этапов:

1) **выбор:** обход дерева игровых состояний и выбор случайных *дочерних узлов*;
2) **расширение:** добавление в дерево нового *узла* (нового игрового состояния);
3) **оценка:** из этого состояния, которое иногда называется *листом*, симулируется случайная игра;
4) **обновление:** после симуляции игры статистика дерева соответствующим образом обновляется.

Симулирование большого количества игр позволяет получать все более точную статистику, которую затем можно будет использовать для выбора следующего хода.

Система AlphaGo использует более сложный алгоритм поиска по дереву, однако многие его аспекты все равно покажутся вам знакомыми. Предыдущие четыре

этапа по-прежнему являются неотъемлемой частью алгоритма MMK AlphaGo, однако мы особым образом используем глубокие нейронные сети для оценки позиций, добавления узлов и отслеживания статистики. В оставшейся части главы мы рассмотрим, как именно это делается, и реализуем версию поиска по дереву AlphaGo.

13.4.1. Нейронные сети и развертывания MMK

В разделах 13.1, 13.2 и 13.3 был подробно описан процесс обучения трех сетей для системы AlphaGo: быстрой и сильной сетей политики и сети ценности. Как можно их использовать для повышения эффективности поиска по дереву методом Монте-Карло? Во-первых, можно осуществлять развертывания не случайным образом, а руководствоваться в ходе этого процесса сетью политики. Именно для этого и предназначена быстрая сеть политики – чтобы оправдать большое количество развертываний, они должны осуществляться *быстро*.

Следующий листинг демонстрирует процесс выбора ходов для данного игрового состояния с помощью сети политики. Мы выбираем лучшие ходы вплоть до окончания игры и, если победит текущий игрок, возвращаем значение 1, в противном случае – возвращаем значение –1.

Листинг 13.12 ❖ Осуществление развертываний с помощью быстрой сети политики

```
def policy_rollout(game_state, fast_policy):
    next_player = game_state.next_player()
    while not game_state.is_over():
        move_probabilities = fast_policy.predict(game_state)
        greedy_move = max(move_probabilities)
        game_state = game_state.apply_move(greedy_move)

    winner = game_state.winner()
    return 1 if winner == next_player else -1
```

Использование такой стратегии развертывания выгодно само по себе, поскольку сети политики позволяют выбирать ходы гораздо эффективнее, чем подбрасывание монетки. Однако возможности для улучшения еще далеко не исчерпаны.

Например, когда мы достигаем конечного узла дерева и хотим добавить после него новый элемент, вместо того чтобы выбирать новый узел случайным образом, мы можем *попросить сильную сеть политики предоставить нам варианты хороших ходов*. Сеть политики выдает распределение вероятностей по всем следующим ходам, и каждый узел может отслеживать эту вероятность так, чтобы сильные (с точки зрения политики) ходы выбирались чаще, чем остальные. Такие вероятности называются *априорными*, поскольку позволяют оценить силу хода еще до выполнения поиска по дереву.

Наконец, давайте посмотрим, какую роль играет сеть ценности. Мы уже улучшили механизм развертывания, заменив случайные угадывания сетью политики. Тем не менее в каждом листе дерева мы вычисляем результат только одной игры для определения ценности этого листа. Однако определение ценности позиции – это как раз та задача, для решения которой мы создавали сеть ценности, так что у нас уже есть довольно изощренный способ судить о ней. Система AlphaGo *взвешивает* результаты развертываний относительно выходных данных сети ценно-

сти. Это похоже на то, как вы принимаете решения в процессе игры: вы пытаетесь просчитать как можно больше ходов, но при этом учитываете полученный ранее игровой опыт. Если вам удастся просчитать выгодную для себя последовательность ходов, вы можете переменить свое мнение о текущей позиции как о проигрышной, и наоборот.

Теперь, когда вы примерно представляете себе роль каждой из трех глубоких нейронных сетей в системе AlphaGo, пришло время разобраться с деталями.

13.4.2. Поиск по дереву с помощью комбинированной функции ценности

В главе 11 мы говорили о применении концепции ценности действия (*Q-значений*) к игре го. Напомним, что для текущего состояния доски s и потенциального следующего хода a ценность действия $Q(s,a)$ определяет примерную выгоду от совершения хода a в ситуации s. О том, как определить $Q(s,a)$, мы поговорим чуть позже, а пока просто учтите, что каждый узел в дереве поиска AlphaGo хранит Q-значения. Кроме того, каждый узел отслеживает *количество посещений*, показывающее, как часто в процессе поиска совершался обход этого узла, а также априорные вероятности $P(s,a)$ или оценку действия a в ситуации s со стороны сильной сети политики.

Каждый узел дерева имеет только одного родителя, но может иметь множество *дочерних элементов*, которые можно закодировать в виде словаря Python, сопоставив ходы в другие узлы. С учетом этого мы можем определить узел `AlphaGoNode` следующим образом.

Листинг 13.13 ❖ Простое представление узла в дереве AlphaGo

```
class AlphaGoNode:
    def __init__(self, parent, probability):
        self.parent = parent
        self.children = {}

        self.visit_count = 0
        self.q_value = 0
        self.prior_value = probability
```

Допустим, вы находитесь в середине партии, уже построили большое дерево, подсчитали количество посещений узлов и получили приблизительную оценку ценности действий. Теперь вам нужно симулировать некоторое количество игр и отследить статистику, чтобы после развертывания можно было выбрать лучший из найденных ходов. Как выполнить обход дерева для симуляции игры? Если вы находитесь в игровом состоянии s и обозначаете количество посещений как $N(s)$, то можете выбрать действие следующим образом:

$$a' = \text{argmax}_a Q(s,a) + \frac{P(s,a)}{1+N(s,a)}.$$

Это уравнение может показаться несколько сложным, поэтому давайте подробно рассмотрим его компоненты.

- Запись argmax означает, что мы берем аргумент a, для которого максимизируется формула $Q(s, a) + P(s, a)/(1 + N(s, a))$.

- Слагаемое, которое мы максимизируем, состоит из двух частей: Q-значения и априорных вероятностей, *нормализованных* по количеству посещений.
- Изначально значения всех счетчиков посещений равны нулю. Это значит, что мы придаем одинаковый вес Q-значению и априорной вероятности, максимизируя $Q(s, a) + P(s, a)$.
- При очень больших значениях счетчиков посещений слагаемым $P(s, a)/(1 + N(s, a))$ можно пренебречь, что фактически оставляет нас с $Q(s, a)$.
- Назовем эту служебную функцию $u(s, a) = P(s, a)/(1 + N(s, a))$. В следующем разделе мы немного модифицируем ее, однако данная версия уже содержит все необходимые компоненты. Для выбора хода также можно использовать запись $a' = \text{argmax}_a Q(s, a) + u(s, a)$.

Таким образом, мы выбираем действия, взвешивая априорные вероятности относительно Q-значений. По мере обхода дерева, увеличения количества посещений и получения все более точных оценок Q мы постепенно отказываемся от своих *априорных оценок* и все больше доверяем Q-значениям. Можно сказать, что мы все меньше полагаемся на априорные знания и все больше исследуем. Здесь можно провести аналогию с вашим собственным игровым опытом. Допустим, вы всю ночь играете в свою любимую стратегическую настольную игру. В начале игры вы полагаетесь на свой предыдущий опыт, однако в процессе игры вы, скорее всего, пробуете что-то новое и обновляете свои представления о том, что работает, а что – нет.

Так AlphaGo *выбирает* ходы из существующего дерева, а как насчет *расширения* дерева при достижении листа l? Рассмотрим рис. 13.5. Сначала мы вычисляем предсказания сильной сети политики $P(l)$ и сохраняем их в качестве априорных вероятностей для каждого дочернего элемента l. Затем мы *оцениваем* конечный узел, *комбинируя результат развертывания сети политики с выходом сети ценности* следующим образом:

$$V(l) = \lambda \cdot \text{value}(l) + (1 - \lambda) \cdot \text{rollout}(l).$$

В этом уравнении value(l) представляет собой выход сети ценности для l, rollout(l) – результат развертывания быстрой сети политики из l, а λ – это значение в диапазоне от 0 до 1, которое по умолчанию задано равным 0,5.

Имейте в виду, что нам требуется симулировать с использованием поиска по дереву n игр, а затем выбрать ход. Чтобы это сработало, по окончании симуляции нужно обновить значения счетчиков посещений и Q-значения. Со счетчиками посещений все просто: мы увеличиваем значение счетчика данного узла на 1 в случае его обхода в процессе поиска. Чтобы обновить Q-значения, мы суммируем $V(l)$ для всех посещенных конечных узлов l и делим на количество посещений:

$$Q(s,a) = \sum_{i=1}^{n} \frac{V(l_i)}{N(s,a)}.$$

Здесь мы суммируем все n симуляций и прибавляем значение концевого узла i-й симуляции, если в ее процессе был выполнен обход узла, соответствующего (s, a). Чтобы подвести итог, давайте посмотрим, как изменился четырехэтапный процесс поиска дерева из главы 4.

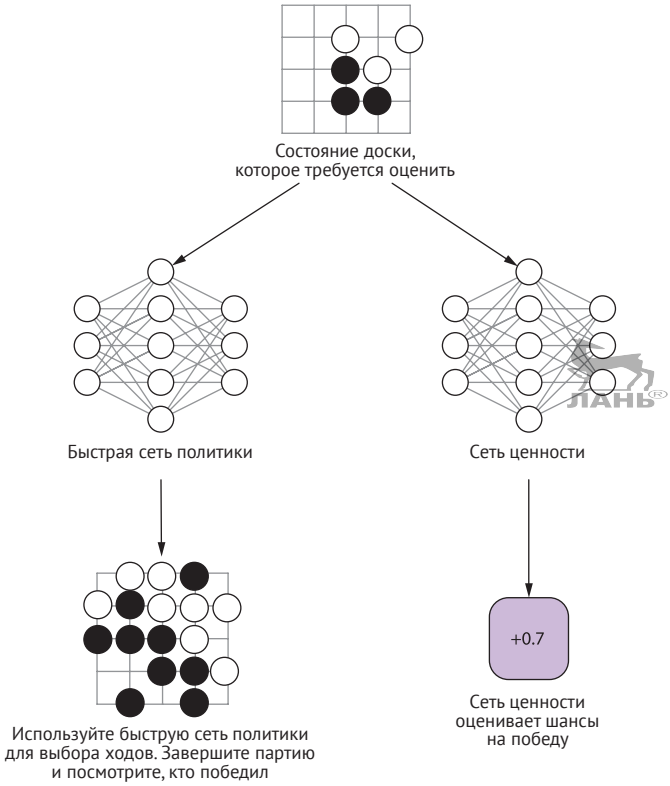


Рис. 13.5 ❖ Чтобы оценить возможные состояния доски, система AlphaGo объединяет две независимые оценки. Во-первых, она подает состояние доски на вход сети ценности, которая возвращает приблизительный шанс на выигрыш. Во-вторых, она использует быструю сеть политики для развертывания игры из этой позиции и определения победителя. Оценка, используемая в ходе поиска по дереву, представляет собой взвешенную сумму этих двух компонентов

1. **Выбор.** Мы обходим дерево игровых состояний, выбирая действия, которые позволяют максимизировать $Q(s, a) + u(s, a)$.
2. **Расширение.** При добавлении нового узла мы единожды просим сильную сеть политики сохранить априорные вероятности для каждого дочернего элемента.
3. **Оценка.** По завершении симуляции выполняется оценка концевого узла путем усреднения выхода сети ценности с результатом развертывания быстрой сети политики.
4. **Обновление.** После завершения всех симуляций мы обновляем счетчики посещений и Q-значения посещенных в процессе этих симуляций узлов.

Единственное, что мы еще не обсудили, – это способ выбора хода после завершения симуляций. Здесь все просто: мы выбираем самый посещаемый узел! Это может показаться слишком упрощенным, однако помните о том, что узлы обходятся все чаще по мере улучшения их Q-значений. После проведения доста-

точного количества симуляций значение счетчика посещений узла позволит нам получить представление об относительной ценности хода.

13.4.3. Реализация алгоритма поиска AlphaGo

Теперь, когда мы обсудили, как система AlphaGo использует нейронные сети в сочетании с поиском по дереву, давайте реализуем этот алгоритм средствами языка Python. Наша цель заключается в создании агента с методом `select_move` в соответствии с методологией AlphaGo. Код, используемый в этом разделе, можно найти в файле *alphago.py* из подмодуля *agent* модуля *dlgo* репозитория GitHub.

Мы начинаем с полного определения узла дерева AlphaGo, процесс которого подробно обсуждался в предыдущем разделе. У узла `AlphaGoNode` есть родитель и дочерние элементы, представленные в виде словаря ходов, сопоставленных в другие узлы. Узел также предусматривает значения `visit_count`, `q_value` и `prior_value`. Кроме того, мы сохраняем *служебную функцию* этого узла `u_value`.

Листинг 13.14 ❖ Определение узла дерева AlphaGo средствами языка Python

```
import numpy as np
from dlgo.agent.base import Agent
from dlgo.goboard_fast import Move
from dlgo import kerasutil
import operator

class AlphaGoNode:
    def __init__(self, parent=None, probability=1.0):
        self.parent = parent
        self.children = {}

        self.visit_count = 0
        self.q_value = 0
        self.prior_value = probability
        self.u_value = probability
```

Узлы дерева имеют один родительский и в потенциале множество дочерних элементов.

Узел инициализируется значением априорной вероятности.

Служебная функция будет обновлена во время поиска.

Такой узел будет использоваться алгоритмом поиска по дереву в трех местах:

1) **`select_child`**: обходя дерево в процессе симуляции, мы выбираем дочерние элементы узла в соответствии с $\text{argmax}_a Q(s, a) + u(s, a)$; выбираем действие, которое максимизирует сумму Q-значения и значения служебной функции;
2) **`expand_children`**: в концевом узле мы просим сильную сеть политики оценить все допустимые ходы из этой позиции и добавить новые экземпляры `AlphaGoNode` для каждого из них;
3) **`update_values`**: наконец, после завершения всех симуляций мы обновляем значения `visit_count`, `q_value` и `u_value`.

Первые два метода являются довольно простыми, как видно в следующем листинге.

Листинг 13.15 ❖ Выбор дочернего элемента `AlphaGoNode` путем максимизации Q-значения

```
class AlphaGoNode():
...

    def select_child(self):
        return max(self.children.items(),
```

```
                key=lambda child: child[1].q_value + \
                child[1].u_value)

    def expand_children(self, moves, probabilities):
        for move, prob in zip(moves, probabilities):
            if move not in self.children:
                self.children[move] = AlphaGoNode(probability=prob)
```

Третий способ обновления статистики узла AlphaGo является чуть более сложным. Во-первых, мы используем более сложную версию служебной функции:

$$u(s,a) = c_u \sqrt{N_p(s,a)} \frac{P(s,a)}{1+N(s,a)}.$$

По сравнению с версией из предыдущего раздела эта функция имеет два дополнительных члена. Первый c_u (в коде `c_u`) масштабирует значение служебной функции для всех узлов на фиксированную *константу*, которая по умолчанию задается равной 5. Второй отвечает за дальнейшее масштабирование значения служебной функции на квадратный корень из числа посещений родительского узла (родитель рассматриваемого узла обозначается *Np*). Благодаря этому узлы, родители которых посещались чаще, отличаются большим значением служебной функции.

Листинг 13.16 ❖ Обновление количества посещений, Q-значения и значения служебной функции узла AlphaGo

```
class AlphaGoNode():
...
    def update_values(self, leaf_value):
        if self.parent is not None:
            self.parent.update_values(leaf_value)

        self.visit_count += 1

        self.q_value += leaf_value / self.visit_count

        if self.parent is not None:
            c_u = 5
            self.u_value = c_u * np.sqrt(self.parent.visit_count) \
                * self.prior_value / (1 + self.visit_count)
```

Сначала обновляются родительские узлы для гарантии того, что обход дерева осуществляется сверху вниз.

Увеличение количества посещений данного узла на 1.

Прибавление ценности указанного листа к Q-значению, нормализованному по количеству посещений.

Обновление значения служебной функции с учетом текущего количества посещений.

На этом определение `AlphaGoNode` завершено. Теперь мы можем использовать эту древовидную структуру в алгоритме поиска, применяемом в системе AlphaGo. Класс `AlphaGoMCTS`, который нам предстоит реализовать, является агентом и инициализируется несколькими аргументами. Сначала мы предоставляем этому агенту быструю и сильную сети политики, а также сеть ценности. Затем нам необходимо указать специфические для AlphaGo параметры развертываний и оценки:

- **`lambda_value`**: это значение λ, используемое для взвешивания результатов развертываний относительно значений функции ценности: $V(l) = \lambda \cdot \text{value}(l) + (1 - \lambda) \cdot \text{rollout}(l)$;
- **`num_simulations`**: это значение показывает, сколько симуляций будет выполнено в процессе выбора хода;

- **depth**: с помощью этого параметра мы сообщаем алгоритму, сколько ходов он должен просчитать в процессе симуляции (т. е. указываем глубину поиска);
- **rollout_limit**: при определении ценности листа мы выполняем развертывание политики rollout(*l*). С помощью параметра rollout_limit сообщаем системе AlphaGo, сколько ходов требуется развернуть, прежде чем судить о результате.

Листинг 13.17 ❖ Инициализация игрового агента AlphaGoMCTS

```
class AlphaGoMCTS(Agent):
    def __init__(self, policy_agent, fast_policy_agent, value_agent,
                 lambda_value=0.5, num_simulations=1000,
                 depth=50, rollout_limit=100):
        self.policy = policy_agent
        self.rollout_policy = fast_policy_agent
        self.value = value_agent

        self.lambda_value = lambda_value
        self.num_simulations = num_simulations
        self.depth = depth
        self.rollout_limit = rollout_limit
        self.root = AlphaGoNode()
```

Теперь пришло время реализовать метод select_move нового агента, который выполняет практически всю тяжелую работу в этом алгоритме. Процедура поиска по дереву AlphaGo обсуждалась в предыдущем разделе, однако давайте еще раз рассмотрим все его этапы.

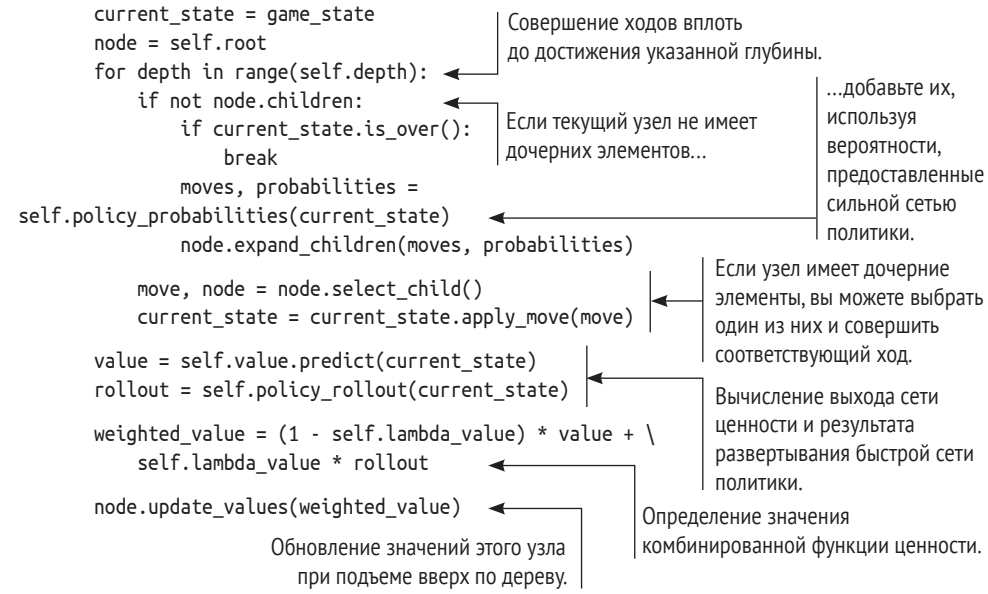
- Когда мы хотим совершить ход, первое, что нам нужно сделать, – это выполнить симуляции в игровом дереве в количестве num_simulations.
- В процессе каждой симуляции мы просчитываем ходы вплоть до достижения указанной глубины (depth).
- Если узел не имеет дочерних элементов, мы *расширяем* дерево, добавляя новый AlphaGoNode для каждого допустимого хода, используя сильную сеть политики для получения априорных вероятностей.
- Если у узла есть дочерние элементы, мы *выбираем* один тот, который максимизирует сумму Q-значения и значения служебной функции.
- Ход, выбранный в процессе симуляции, применяется к игровой доске.
- По достижении предельной глубины поиска мы *оцениваем* концевой узел дерева, вычисляя значение комбинированной функции ценности на основании выхода сети ценности и результата развертывания сети политики.
- Мы обновляем все узлы AlphaGo с учетом ценности листьев, определенной в процессе выполнения симуляций.

Именно этот процесс будет реализован в методе select_move. Обратите внимание на то, что данный метод предполагает использование двух других служебных методов policy_probabilities и policy_rollout, о которых мы поговорим далее.

Листинг 13.18 ❖ Основной метод алгоритма поиска по дереву в системе AlphaGo

```
class AlphaGoMCTS(Agent):
...
    def select_move(self, game_state):
        for simulation in range(self.num_simulations):
```

Симуляция заданного количества игр из текущего игрового состояния.



Как вы могли заметить, несмотря на осуществление всех симуляций, мы так и не совершили ни одного хода. Для этого мы выбираем самый посещаемый узел, после чего нам остается лишь задать новый корневой узел (`root`) и возвратить предложенный ход.

Листинг 13.19 ❖ Выбор наиболее посещаемого узла и обновление корневого узла дерева

```
class AlphaGoMCTS(Agent):
...
    def select_move(self, game_state):
    ...
        move = max(self.root.children, key=lambda move:
                   self.root.children.get(move).visit_count)

        self.root = AlphaGoNode()
        if move in self.root.children:
            self.root = self.root.children[move]
            self.root.parent = None

        return move
```


Это завершает основной процесс поиска по дереву AlphaGo, поэтому давайте рассмотрим два упомянутых ранее служебных метода. Метод `policy_probabilities`, используемый при добавлении узлов, вычисляет предсказания сильной сети политики, ограничивает их допустимыми ходами, а затем нормализует оставшиеся предсказания. Этот метод возвращает как допустимые ходы, так и соответствующие им нормализованные предсказания сети политики.

Листинг 13.20 ❖ Вычисление нормализованных значений сильной сети политики для допустимых ходов

```
class AlphaGoMCTS(Agent):
...
```

```
    def policy_probabilities(self, game_state):
        encoder = self.policy._encoder
        outputs = self.policy.predict(game_state)
        legal_moves = game_state.legal_moves()
        if not legal_moves:
            return [], []
        encoded_points = [encoder.encode_point(move.point) for move in
➥ legal_moves if move.point]
        legal_outputs = outputs[encoded_points]
        normalized_outputs = legal_outputs / np.sum(legal_outputs)
        return legal_moves, normalized_outputs
```

Последний вспомогательный метод, `policy_rollout`, потребуется нам для вычисления результата развертывания с использованием быстрой политики. Этот метод *жадно* выбирает самый сильный ход в соответствии с быстрой политикой до достижения предела развертывания, а затем определяет победителя. Он возвращает значение 1, если победил игрок, обладающий правом следующего хода, –1, если победил другой игрок, и 0, если не был достигнут ни один из результатов.

Листинг 13.21 ❖ Продолжение игры вплоть до достижения `rollout_limit`

```
class AlphaGoMCTS(Agent):
...
    def policy_rollout(self, game_state):
        for step in range(self.rollout_limit):
            if game_state.is_over():
                break
            move_probabilities = self.rollout_policy.predict(game_state)
            encoder = self.rollout_policy.encoder
            valid_moves = [m for idx, m in enumerate(move_probabilities)
                           if Move(encoder.decode_point_index(idx)) in
➥ game_state.legal_moves()]
            max_index, max_value = max(enumerate(valid_moves),
➥ key=operator.itemgetter(1))
            max_point = encoder.decode_point_index(max_index)
            greedy_move = Move(max_point)
            if greedy_move in game_state.legal_moves():
                game_state = game_state.apply_move(greedy_move)

        next_player = game_state.next_player
        winner = game_state.winner()
        if winner is not None:
            return 1 if winner == next_player else -1
        else:
            return 0
```

После создания фреймворка `Agent` и реализации агента AlphaGo мы можем использовать для игры экземпляр `AlphaGoMCTS`.

Листинг 13.22 ❖ Инициализация агента AlphaGo с тремя глубокими нейронными сетями

```
from dlgo.agent import load_prediction_agent, load_policy_agent, AlphaGoMCTS
from dlgo.rl import load_value_agent
import h5py
```

```
fast_policy = load_prediction_agent(h5py.File('alphago_sl_policy.h5', 'r'))
strong_policy = load_policy_agent(h5py.File('alphago_rl_policy.h5', 'r'))
value = load_value_agent(h5py.File('alphago_value.h5', 'r'))

alphago = AlphaGoMCTS(strong_policy, fast_policy, value)
```

Этот агент может использоваться точно так же, как и другие агенты, созданные в главах с 7 по 12. В частности, для него можно зарегистрировать HTTP- и GTP-фронтенды, как это делалось в главе 8. Благодаря этому вы сможете играть против своего бота AlphaGo, организовать его игру с другими ботами и даже запустить его на онлайн-сервере для игры в го (например, OGS, как показано в приложении Д).

13.5. Практические советы, касающиеся обучения бота AlphaGo

В предыдущем разделе мы разработали примитивную версию алгоритма поиска по дереву, используемого в системе AlphaGo. Данный алгоритм *может* достичь сверхчеловеческого уровня игры в го, однако для этого вам потребуется учесть множество нюансов. Помимо обучения всех трех глубоких нейронных сетей, используемых в системе AlphaGo, вам нужно будет обеспечить достаточно высокую скорость выполнения симуляций, чтобы не пришлось часами ждать, пока AlphaGo предложит следующий ход. Вот несколько практических советов:

- на первом этапе для обучения сетей политики с учителем были использованы данные 160 000 игровых партий с сервера KGS, содержащих около 30 млн игровых состояний. В общей сложности команда DeepMind, работающая над системой AlphaGo, насчитала 340 млн этапов обучения;
- хорошая новость заключается в том, что вы тоже можете получить доступ к этому набору данных. Компания DeepMind использовала обучающий набор KGS, представленный в главе 7. В принципе, ничто не мешает вам выполнить то же количество этапов обучения. Плохая новость заключается в том, что даже при наличии современного графического процессора этот процесс может занять много месяцев, а то и лет;
- команда AlphaGo решила эту проблему, *распределив* нагрузку между 50 графическими процессорами, сократив время обучения до трех недель. Маловероятно, что этот вариант вам подойдет, особенно если учесть, что мы не обсуждали тему распределенного обучения глубоких сетей;
- для получения удовлетворительных результатов вы можете, например, применить кодировщики доски из глав 7 или 8, а также использовать сети гораздо меньшего размера по сравнению с сетями политики и ценности AlphaGo, представленными в этой главе. Кроме того, попробуйте сначала использовать небольшой обучающий набор, чтобы наработать некоторую интуицию;
- в ходе игры с самим собой система AlphaGo сгенерировала 30 млн различных позиций. Это значительно превышает количество игровых состояний, которое можете сгенерировать вы. Ориентируйтесь на количество позиций, соответствующее числу игровых состояний из партий, сыгранных человеком, использовавшихся на этапе обучения с учителем;
- если вы просто возьмете большие сети, описанные в этой главе, и обучите их на очень небольших наборах данных, результат, вероятно, будет гораздо

хуже по сравнению с обучением сетей меньшего размера на больших наборах данных;

- быстрая сеть политики часто используется при осуществлении развертываний, поэтому для ускорения процесса поиска по дереву убедитесь в том, что ваша быстрая сеть политики поначалу является относительно небольшой, как сети, использовавшиеся в главе 6;
- алгоритм поиска по дереву, реализованный в предыдущем разделе, выполняет симуляции *последовательно*. Чтобы ускорить этот процесс, команда DeepMind *распараллелила* процесс поиска на 40 потоков. При этом несколько графических процессоров использовались для параллельной оценки глубоких сетей и еще несколько процессоров отвечали за другие части алгоритма поиска по дереву;
- выполнение поиска по дереву на нескольких процессорах, в принципе, осуществимо (в главе 7 мы уже использовали многопоточность для подготовки данных), однако эта тема является слишком сложной, чтобы обсуждать ее в этой книге;
- для улучшения игрового процесса вы можете пожертвовать скоростью ради большей мощности, уменьшив количество симуляций и глубину поиска. Это не позволит достичь сверхчеловеческого уровня, но, по крайней мере, система станет пригодной для игры.

Как видите, помимо нового способа комбинирования различных методов обучения и поиска по дереву, создание первого бота, сумевшего обыграть в го лучших профессионалов, потребовало и инженерных усилий, направленных на оптимизацию этих процессов.

В последней главе мы поговорим о следующем этапе разработки системы AlphaGo, который позволяет не только обойтись без обучения на основе партий, сыгранных человеком, но и достичь *более высокого уровня игры* по сравнению с исходной системой AlphaGo, реализованной в этой главе.

13.6. Резюме

- Для обеспечения работы системы AlphaGo необходимо обучить три глубокие нейронные сети: две сети политики и сеть ценности.
- Быстрая сеть политики обучается на данных партий, сыгранных человеком, и должна быть достаточно быстрой для осуществления большого количества развертываний алгоритма поиска по дереву в системе AlphaGo. Результат развертываний используется для оценки позиций в концевых узлах дерева.
- Сильная сеть политики сначала обучается на данных партий, сыгранных человеком, а затем улучшается в игре с самой собой с применением алгоритма градиента политики. Эта сеть используется в AlphaGo при вычислении априорных вероятностей для выбора узла.
- Сеть ценности обучается на данных опыта, сгенерированных в ходе игры с самой собой, и используется для оценки позиции в концевых узлах наряду с развертыванием политики.
- Выбор хода с помощью AlphaGo означает генерирование множества симуляций, предполагающих обход игрового дерева. После завершения симуляций выбирается самый посещаемый узел.

- В процессе симуляции *выбор* узлов осуществляется путем максимизации суммы Q-значения и значения служебной функции.
- По достижении концевого узла дерево *расширяется* с учетом априорных вероятностей, предоставленных сильной сетью политики.
- Концевой узел оценивается с помощью комбинированной функции ценности, объединяющей выходное значение сети ценности с результатом развертывания быстрой политики.
- При подъеме вверх по дереву счетчики посещений, Q-значения, а также значения служебных функций обновляются с учетом выбранных действий.

Глава 14

AlphaGo Zero: интеграция поиска по дереву и обучения с подкреплением

В этой главе:

- игры с использованием особой версии алгоритма Монте-Карло;
- интеграция поиска по дереву и игры бота с самим собой для реализации обучения с подкреплением;
- обучение нейронной сети для повышения эффективности алгоритма поиска по дереву.

После того как компания DeepMind представила вторую версию AlphaGo под кодовым названием *Master*, поклонники игры го по всему миру начали тщательно изучать ее шокирующий стиль игры. Партии этой системы содержали множество удивительных новых ходов. Несмотря на то что версия Master была обучена на партиях, сыгранных человеком, благодаря обучению с подкреплением она сумела обнаружить новые ходы, еще не совершавшиеся людьми.

Это вызвало очевидный вопрос: что, если бы система AlphaGo совершенно не полагалась на человеческие партии и полностью зависела от обучения с подкреплением? Смогла бы она превзойти игру человека или застряла бы на уровне новичка? Выявила бы она приемы, используемые мастерами, или начала бы демонстрировать совершенно непонятный стиль игры? Ответы на все эти вопросы были получены в 2017 году, когда был представлен алгоритм AlphaGo Zero (AGZ).

Алгоритм AlphaGo Zero основан на улучшенной системе обучения с подкреплением, при этом он обучил сам себя с нуля без использования записей партий, сыгранных человеком. Несмотря на то что поначалу уровень его игры не превышал уровень новичка, алгоритм AGZ неуклонно прогрессировал, что позволило ему быстро превзойти все предыдущие версии AlphaGo.

Самое удивительное в AlphaGo Zero то, что этот алгоритм достигает большего меньшими средствами. Во многих отношениях AGZ гораздо проще оригинальной версии AlphaGo. Этот алгоритм не использует плоскости признаков, записи человеческих партий и развертывания Монте-Карло. Вместо двух нейронных сетей и трех алгоритмов обучения AlphaGo Zero использует одну нейронную сеть и один алгоритм обучения.

И тем не менее алгоритм AlphaGo Zero сильнее исходной версии AlphaGo! Чем это объясняется? Во-первых, AGZ использует очень массивную нейронную сеть. Работа самой сильной версии этого алгоритма обеспечивается сетью, емкость которой примерно соответствует 80 сверточным слоям, что в четыре раза превышает размер исходной сети AlphaGo.

Во-вторых, AGZ использует инновационную методику обучения с подкреплением. Исходная версия AlphaGo предполагала обучение сети политики методом, примерно соответствующим тому, который был описан в главе 10, после чего эта сеть политики использовалась для улучшения поиска по дереву. Напротив, AGZ предполагает изначальную интеграцию поиска по дереву и обучения с подкреплением. Именно об этом пойдет речь в этой главе.

Сначала мы обсудим структуру нейронной сети, обучаемой AGZ. Затем подробно рассмотрим алгоритм поиска по дереву. AGZ использует один и тот же алгоритм поиска по дереву как в играх с самим собой, так и в играх с другими противниками. После этого мы поговорим о том, как AGZ обучает свою сеть на данных опыта. В конце главы приведены практические приемы, используемые программой AGZ для обеспечения стабильности и эффективности процесса обучения.

14.1. Создание нейронной сети для поиска по дереву

Алгоритм AlphaGo Zero использует единственную нейронную сеть с одним входом и двумя выходами: один выход предоставляет распределение вероятностей для ходов, а другой – значение, говорящее о том, для кого из игроков текущее игровое состояние является более выгодным. Такую же структуру мы использовали для обучения методом типа «актор–критик» в главе 12.

Между выходом сети AGZ и сети, использованной в главе 12, существует небольшая разница, связанная с пропуском хода в игре. В предыдущих реализациях игры бота с самим собой мы жестко закодировали логику пропуска хода. Например, бот `PolicyAgent` из главы 9 предусматривал логику, запрещающую ему заполнять собственные глаза и тем самым уничтожать свои камни. Когда все допустимые ходы являются самоубийственными, агент `PolicyAgent` должен пропустить ход. Это позволяет боту разумно завершить игру с самим собой.

Поскольку при игре с самим собой алгоритм AGZ использует поиск по дереву, нам не требуется эта логика. Мы можем рассматривать пропуск хода так же, как и любой другой ход, и рассчитывать на то, что бот научится самостоятельно определять момент, когда пропуск хода является целесообразным. Если в результате поиска по дереву выяснится, что размещение камня приведет к проигрышу, бот пропустит ход. Это означает, что на выходе мы должны получать вероятности не только для каждой точки доски, но и для пропуска хода. То есть вместо того, чтобы возвращать вектор $19 \times 19 = 361$, представляющий каждую точку на доске, наша сеть должна выдавать вектор $19 \times 19 + 1 = 362$ для представления каждой точки на доске *и* пропуска хода. Такой способ кодирования ходов проиллюстрирован на рис. 14.1.

Это означает, что нам требуется немного изменить кодировщик доски. В предыдущих версиях кодировщика мы реализовали методы `encode_point` и `decode_point_index`, которые отвечали за преобразование между элементами вектора и точками доски. В случае с ботом AGZ мы заменим их новыми функциями, `encode_move` и `decode_move_index`. Процесс кодирования хода остается прежним, а для представления пропуска хода используется индекс следующего элемента.

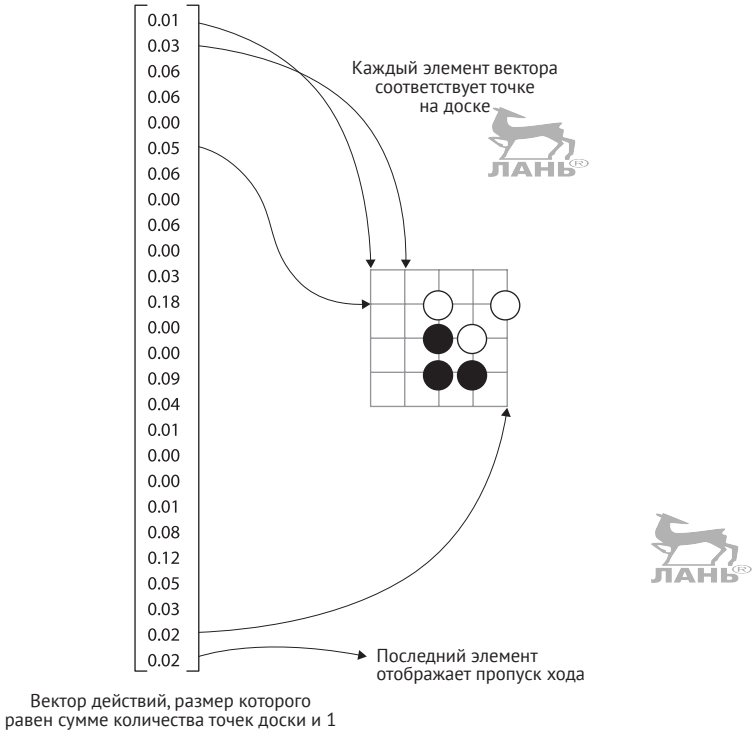


Рис. 14.1 ❖ Кодирование возможных ходов в виде вектора. Как и в предыдущих главах, AGZ использует вектор, каждый элемент которого соответствует точке игровой доски. Кроме того, AGZ добавляет дополнительный элемент, соответствующий пропуску хода. В этом примере используется доска 5×5, поэтому вектор имеет размерность 26:25 для точек на доске и 1 для пропуска хода

Листинг 14.1 ❖ Изменение кодировщика доски для учета пропуска хода

```
class ZeroEncoder(Encoder):
...
    def encode_move(self, move):
        if move.is_play:
            return (self.board_size * (move.point.row - 1) +
                (move.point.col - 1))
        elif move.is_pass:
            return self.board_size * self.board_size
        raise ValueError('Cannot encode resign move')

    def decode_move_index(self, index):
        if index == self.board_size * self.board_size:
            return Move.pass_turn()
        row = index // self.board_size
        col = index % self.board_size
        return Move.play(Point(row=row + 1, col=col + 1))

    def num_moves(self):
        return self.board_size * self.board_size + 1
```

Точки доски кодируются так же, как в предыдущих кодировщиках.

Пропуск хода кодируется с помощью индекса следующего элемента.

Сообщение, говорящее о том, что нейронная сеть не научилась выходить из игры.

Помимо метода обработки пропуска хода, вход и выход сети AGZ ничем не отличаются от версии, описанной в главе 12. В качестве внутренних слоев сети AGZ использует чрезвычайно глубокий стек сверточных слоев с некоторыми нововведениями, позволяющими сделать процесс обучения более плавным (мы кратко рассмотрим их в конце этой главы). Большая сеть отличается большей мощностью, однако она предполагает и большее количество вычислений, как в ходе обучения, так и в ходе игры сети с самой собой. При отсутствии доступа к оборудованию наподобие того, которое использовалось компанией DeepMind, рассмотрите возможность использования сети меньшего размера. Поэкспериментируйте, чтобы найти правильный баланс между мощностью и скоростью.

Для кодирования доски вы можете использовать любую схему, описанную в этой книге, начиная с простейшего кодировщика из главы 6 и заканчивая 48-плоскостным кодировщиком из главы 13. Алгоритм AlphaGo Zero использует самый простой из возможных кодировщиков, учитывающий только расположение черных и белых камней на доске и очередь хода. (Чтобы реализовать правило ко, AGZ также предусматривает плоскости для предыдущих семи состояний доски.) Однако нет никаких технических причин, по которым вы не можете использовать специфические для игры в го плоскости признаков в целях ускорения процесса обучения. Исследователи старались применять минимальное количество человеческих знаний просто для того, чтобы доказать, что без него вполне можно обойтись. В ходе собственных экспериментов вы можете свободно использовать различные комбинации плоскостей признаков при реализации обучения с подкреплением.

14.2. Управление процессом поиска по дереву с помощью нейронной сети

В обучении с подкреплением *политика* представляет собой алгоритм, который говорит агенту, как ему следует принимать решения. В предыдущих примерах обучения с подкреплением использовалась относительно простая политика. При обучении на основе градиента политики (глава 10) и метода типа «актор–критик» (глава 12) нейронная сеть напрямую говорила нам о том, какой ход следует выбрать. При использовании алгоритма Q-обучения (глава 11) политика предполагала вычисление Q-значения для каждого возможного хода, после чего выбирался ход с самым высоким Q-значением.

Политика AGZ предполагает применение определенной формы алгоритма поиска по дереву. Мы по-прежнему используем нейронную сеть, однако ее цель заключается в том, чтобы направлять процесс поиска по дереву, а не в непосредственном выборе или оценке ходов. Интеграция алгоритма поиска по дереву в игру сети с самой собой позволяет сделать такие игры более реалистичными, что, в свою очередь, повышает стабильность процесса обучения.

В основе алгоритма поиска по дереву лежат идеи, которые были рассмотрены ранее. Если вы изучили алгоритм поиска по дереву методом Монте-Карло (глава 4) и исходную версию AlphaGo (глава 13), то алгоритм поиска по дереву системы AlphaGo Zero покажется вам знакомым. Сравнение этих трех алгоритмов приведено в табл. 14.1. Сначала мы обсудим структуру данных, которую AGZ использует для представления игрового дерева. Затем рассмотрим алгоритм, применяемый AGZ для добавления в это дерево новой игровой позиции.

Таблица 14.1. Сравнение алгоритмов поиска по дереву

	Монте-Карло	**AlphaGo**	**AlphaGo Zero**
Выбор ветвей	Оценка UCT	Оценка UCT + априорная вероятность, полученная от сети политики	Оценка UCT + априорная вероятность, полученная от комбинированной сети
Оценка ветвей	Рандомизированные разыгрывания	Сеть ценности + рандомизированные разыгрывания	Значение, полученное от комбинированной сети

Основная идея алгоритмов поиска по дереву, применяемых к настольным играм, заключается в нахождении хода, обеспечивающего наилучший результат. Мы определяем это, изучая возможные последовательности ходов. Однако количество таких последовательностей огромно, поэтому нам нужно принять решение, изучив лишь малую их часть. Все искусство поиска по дереву заключается в выборе ветвей для исследования с целью нахождения наилучшего результата за минимальное количество времени.

Как и в случае с методом Монте-Карло, алгоритм поиска по дереву AGZ выполняется в течение определенного количества раундов, в ходе которых в дерево добавляется очередное состояние доски. По мере выполнения все большего количества раундов дерево увеличивается, и оценки алгоритма становятся все более точными. Чтобы лучше в этом разобраться, представьте, что вы применяете этот алгоритм в течение некоторого времени: вы уже частично создали дерево и хотите добавить в него новое состояние доски. Пример игрового дерева представлен на рис. 14.2.

Каждый узел в игровом дереве представляет собой возможное состояние доски, которое также определяет допустимые варианты следующих ходов. Алгоритм уже посетил некоторые из этих следующих ходов. Мы создаем *ветвь* для каждого из возможных следующих ходов вне зависимости от того, посетили мы его или нет. Каждая ветвь отслеживает:

- *априорную вероятность* хода, соответствующую примерной оценке хода еще до его посещения;
- количество посещений ветви в процессе выполнения поиска по дереву, которое может быть равно нулю;
- *математическое ожидание* всех посещений этой ветви, которая представляет собой среднее значение, полученное в ходе всех эпизодов обхода дерева. Для облегчения процесса обновления этого среднего значения мы сохраняем сумму всех значений, которая затем делится на количество посещений.

Для каждой *посещенной* ветви узел также содержит указатель на *дочерний узел*. В следующем листинге мы определяем минимальную структуру `Branch`, которая будет содержать статистику ветви.

Листинг 14.2 ❖ Структура для отслеживания статистики ветви

```
class Branch:
    def __init__(self, prior):
        self.prior = prior
        self.visit_count = 0
        self.total_value = 0.0
```

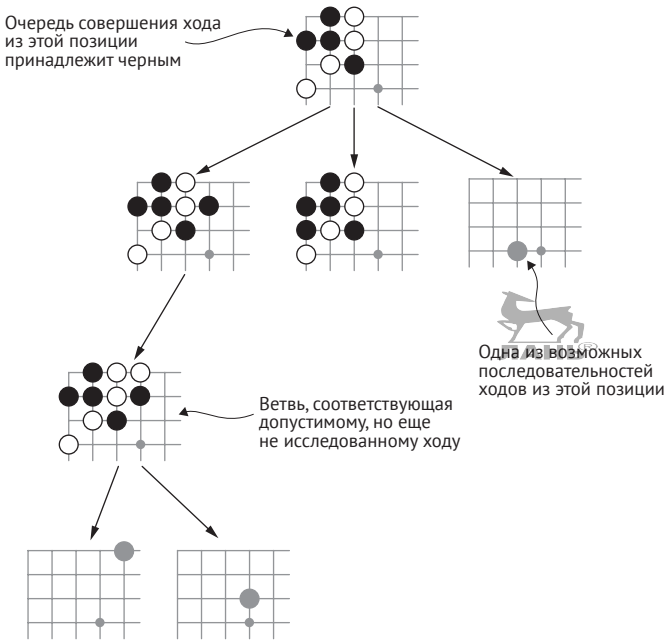


Рис. 14.2 ❖ Частичное дерево поиска в стиле AGZ. В данном случае очередь хода принадлежит черным, а алгоритм поиска по дереву изучил только три возможных игровых состояния. Дерево также содержит ветви, соответствующие еще не исследованным ходам. Большинство из них было опущено в целях экономии места

Теперь пришло время создать структуру для представления дерева поиска. В следующем листинге мы определяем класс `ZeroTreeNode`.

Листинг 14.3 ❖ Узел дерева поиска в стиле AGZ

```
class ZeroTreeNode:
    def __init__(self, state, value, priors, parent, last_move):
        self.state = state
        self.value = value
        self.parent = parent                    # В корне дерева параметры parent и last_move
        self.last_move = last_move              # будут иметь значение None.
        self.total_visit_count = 1
        self.branches = {}
        for move, p in priors.items():
            if state.is_valid_move(move):
                self.branches[move] = Branch(p)
        self.children = {}                      # Позднее дочерние элементы будут отображены
                                                # из move в другой ZeroTreeNode.
    def moves(self):                            # Возвращает список всех возможных ходов
        return self.branches.keys()             # из данного узла.

    def add_child(self, move, child_node):      # Позволяет добавить в дерево новые узлы.
        self.children[move] = child_node

    def has_child(self, move):                  # Проверяет наличие дочернего узла для конкретного хода.
        return move in self.children            # Возвращает определенный дочерний узел.
```

Класс ZeroTreeNode также содержит несколько вспомогательных функций для считывания статистики из дочерних элементов.

Листинг 14.4 ❖ Вспомогательные функции для чтения информации о ветви из узла дерева

```
class ZeroTreeNode:
...
    def expected_value(self, move):
        branch = self.branches[move]
        if branch.visit_count == 0:
            return 0.0
        return branch.total_value / branch.visit_count

    def prior(self, move):
        return self.branches[move].prior

    def visit_count(self, move):
        if move in self.branches:
            return self.branches[move].visit_count
        return 0
```

14.2.1. Спуск по дереву

Каждый раунд поиска начинается со спуска по дереву. Целью является проверка и оценка возможного будущего состояния доски. Чтобы оценка была точной, мы должны предположить, что наш противник совершит максимально сильный ответный ход. Разумеется, этот ход нам пока неизвестен, поэтому нам требуется опробовать различные варианты. В этом разделе мы обсудим алгоритм выбора сильных ходов в условиях неопределенности.

Математическое ожидание – это примерная оценка того, насколько хорош каждый из возможных ходов. Однако не все оценки одинаково точны. Чем больше времени мы потратим на исследование конкретной ветви, тем лучше будет оценка.

Более подробное исследование одного из лучших вариантов позволяет повысить точность его оценки. С этой же целью можно просчитать наименее исследованную ветвь. Вполне вероятно, что этот ход лучше, чем могло показаться изначально. Единственный способ выяснить это заключается в более подробном исследовании. В данном случае речь снова идет о нахождении компромисса между *эксплуатацией* и *разведкой*.

Исходный алгоритм поиска по дереву методом Монте-Карло использовал для этого формулу UCT (верхний предел доверительного интервала для деревьев; см. главу 4). Формула UCT позволяет сбалансировать две эти цели:

- если ветвь посещалась много раз, мы доверяем значению математического ожидания и отдаем предпочтение ветвям с более высокими оценками;
- если ветвь посещалась лишь несколько раз, значение ее математического ожидания может оказаться очень не точным. Вне зависимости от математического ожидания вам следует несколько раз посетить эту ветвь, чтобы повысить точность ее оценки.

В системе AGZ используется третий фактор:

- среди всех ветвей с небольшим количеством посещений предпочтение следует отдавать ветвям с высокой априорной вероятностью. Это ходы, которые еще до учета всех деталей игры интуитивно кажутся хорошими.

Математически функция оценки AGZ выражается следующим образом:

$$Q + cP\frac{\sqrt{N}}{1+n},$$

где Q – это математическое ожидание, усредненное по количеству посещений ветви (оно равно нулю, если эта ветвь еще не посещалась); P – это априорная вероятность для рассматриваемого хода; N – количество посещений *родительского* узла; n – количество посещений *дочерней* ветви; c – это коэффициент, который позволяет сбалансировать разведку и эксплуатацию (как правило, он выбирается методом проб и ошибок).

Рассмотрим пример, приведенный на рис. 14.3. Ветвь A посещалась дважды и имеет хорошую оценку Q = 0,1. Ветвь B посещалась один раз и имеет плохую оценку: Q = –0,5. Ветвь C не посещалась, но имеет априорную вероятность P = 0,038.

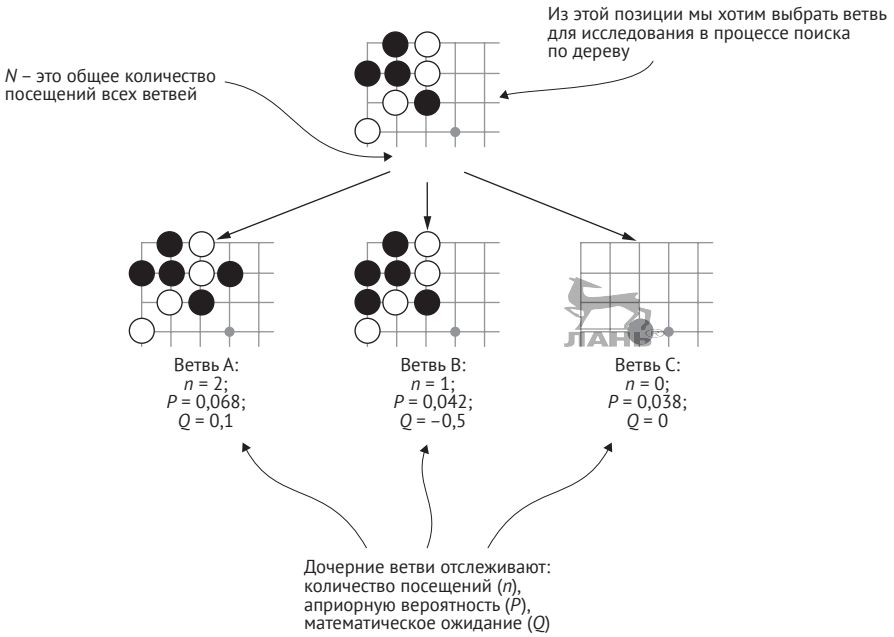


Рис. 14.3 ❖ Выбор ветви для исследования в процессе поиска по дереву AGZ. В данном примере мы рассматриваем три ветви из стартовой позиции. (На практике существует гораздо больше возможных ходов, но мы опустили их в целях экономии места.) Для выбора ветви учитываем количество посещений ветви, ее оценочное значение, а также априорную вероятность хода

В табл. 14.2 показано, как рассчитывается компонент неопределенности. Ветвь A отличается наибольшим значением Q, что указывает на наличие в ней нескольких хороших состояний доски. Ветвь C имеет самое большое значение функции UCT: мы ни разу ее не посещали, поэтому эта ветвь характеризуется наивысшей

неопределенностью. Ветвь B имеет более низкую оценку, чем ветвь A, и большее количество посещений, чем ветвь C, поэтому она вряд ли является хорошим вариантом в данный момент.

Таблица 14.2. Выбор ветви для исследования

	Q	n	N	P	$P\sqrt{N}/(n+1)$
Ветвь A	0,1	2	3	0,068	0,039
Ветвь B	−0,5	1	3	0,042	0,036
Ветвь C	0	0	3	0,038	0,065

Предположим, мы отбросили ветвь B, как нам сделать выбор между вариантами A и C? Этот выбор будет зависеть от значения параметра c. Небольшое значение c благоприятствует ветви с более высокой оценкой (в данном случае это ветвь A). Большое значение c благоприятствует ветви с наибольшей неопределенностью (в данном случае это ветвь C). Например, при c = 1,0 мы бы выбрали вариант A (0,139 против 0,065). При c = 4,0 мы бы выбрали вариант C (0,260 против 0,256). Ни одна из этих оценок не является объективно правильной, это просто компромисс. Следующий листинг демонстрирует вычисление этой оценки средствами языка Python.

Листинг 14.5 ❖ Выбор дочерней ветви


```
class ZeroAgent(Agent):
...
    def select_branch(self, node):
        total_n = node.total_visit_count

        def score_branch(move):
            q = node.expected_value(move)
            p = node.prior(move)
            n = node.visit_count(move)
            return q + self.c * p * np.sqrt(total_n) / (n + 1)

        return max(node.moves(), key=score_branch)
```

`node.moves()` – это список ходов. При передаче `key=score_branch` функция `max` возвращает ход с наибольшим значением функции `score_branch`.

После выбора ветви мы производим те же вычисления для ее дочерних элементов с целью выбора следующей ветви. Этот процесс продолжается вплоть до достижения ветви, не имеющей дочерних элементов.

Листинг 14.6 ❖ Спуск по дереву поиска

```
class ZeroAgent(Agent):
...
    def select_move(self, game_state):
        root = self.create_node(game_state)

        for i in range(self.num_rounds):
            node = root
            next_move = self.select_branch(node)
            while node.has_child(next_move):
                node = node.get_child(next_move)
                next_move = self.select_branch(node)
```

В следующем разделе будет представлена реализация функции `create_node`.

Это первый этап процесса, который повторяется много раз для каждого хода. `self.num_moves` определяет количество повторений цикла поиска.

Если функция `has_child` возвращает значение `False`, значит, мы достигли концевого узла дерева.

14.2.2. Расширение дерева

По достижении концевого узла дерева поиск не может продолжаться, поскольку в дереве больше нет узлов, соответствующих текущему ходу. Поэтому следующим шагом является создание нового узла и его добавление в дерево.

Чтобы создать новый узел, мы берем предыдущее игровое состояние и применяем текущий ход для получения нового игрового состояния. Теперь мы можем подать это новое состояние на вход нейронной сети, которая предоставит нам два ценных значения. Во-первых, мы получим априорные вероятности для всех возможных следующих ходов из нового игрового состояния. Во-вторых, мы получим оценочное значение нового игрового состояния. Эта информация будет использована для инициализации статистики ветвей из этого нового узла.

Листинг 14.7 ❖ Создание нового узла в дереве поиска

```
class ZeroAgent(Agent):
...
    def create_node(self, game_state, move=None, parent=None):
        state_tensor = self.encoder.encode(game_state)
        model_input = np.array([state_tensor])
        priors, values = self.model.predict(model_input)
        priors = priors[0]
        value = values[0][0]
        move_priors = {
            self.encoder.decode_move_index(idx): p
            for idx, p in enumerate(priors)
        }
        new_node = ZeroTreeNode(
            game_state, value,
            move_priors,
            parent, move)
        if parent is not None:
            parent.add_child(move, new_node)
        return new_node
```

Функция Keras `predict` – это функция пакетной обработки, принимающая массив примеров. Поэтому мы должны обернуть `board_tensor` в массив.

Функция `predict` возвращает массивы со множеством результатов, из которых мы извлекаем первый элемент.

Распаковка вектора априорных вероятностей в словарь, отображающий объекты `move` в соответствующие априорные вероятности.

Наконец, мы поднимаемся вверх по дереву и обновляем статистику каждого из предков данного узла, как показано на рис. 14.4. Для каждого узла на этом пути мы инкрементируем количество посещений и обновляем итоговое значение математического ожидания. В каждом узле перспектива поочередно переключается между черными и белыми, что указывает на необходимость изменения знака.

Листинг 14.8 ❖ Расширение дерева поиска и обновление статистики всех узлов

```
class ZeroTreeNode:
...
    def record_visit(self, move, value):
        self.total_visit_count += 1
        self.branches[move].visit_count += 1
        self.branches[move].total_value += value

class ZeroAgent(Agent):
...
    def select_move(self, game_state):
...
            new_state = node.state.apply_move(next_move)
```

```
child_node = self.create_node(
    new_state, parent=node)

move = next_move
value = -1 * child_node.value
while node is not None:
    node.record_visit(move, value)
    move = node.last_move
    node = node.parent
    value = -1 * value
```

На каждом уровне дерева мы переключаем перспективу между двумя игроками, что требует умножения значения на –1: то, что хорошо для черных, плохо для белых, и наоборот.

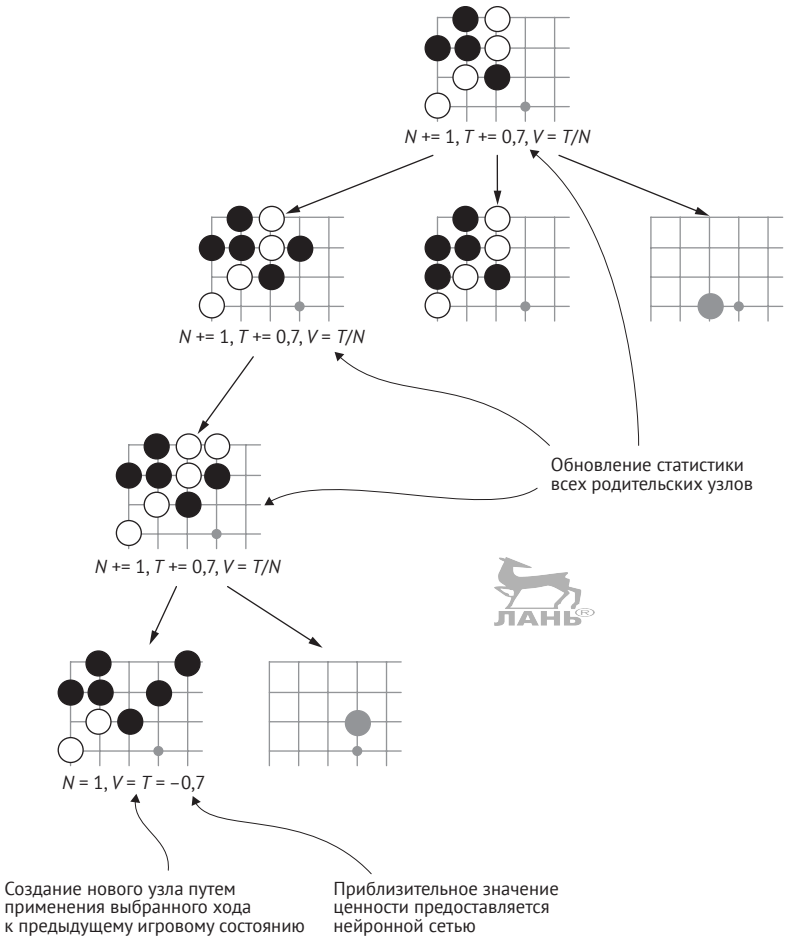


Рис. 14.4 ❖ Расширение дерева поиска AGZ. Сначала мы вычисляем новое игровое состояние. Из этого состояния мы создаем новый узел и добавляем его в дерево. Затем нейронная сеть предоставляет нам приблизительное значение ценности этого игрового состояния. Наконец, мы обновляем статистику всех предков нового узла. Мы увеличиваем количество посещений *N* на 1 и обновляем среднее значение *V*. В данном случае *T* – это общее значение ценности, исходя из всех посещений, произведенных через этот узел. Этот простой прием позволяет упростить процесс пересчета среднего значения

Этот процесс повторяется снова и снова, и каждый раз дерево расширяется. Система AGZ выполняет 1600 раундов на ход в процессе игры с самой собой. Чтобы бот выбирал все более удачные ходы в соревновательных играх, алгоритм следует запускать максимальное количество раз.

14.2.3. Выбор хода

Теперь, когда мы создали максимально глубокое дерево поиска, пришло время выбрать ход. Самое простое правило предполагает выбор хода с наибольшим количеством посещений.

Почему мы используем количество посещений, а не математическое ожидание? Может показаться, что самая посещаемая ветвь отличается высоким значением математического ожидания. Чтобы ответить на этот вопрос, вернемся к предыдущей формуле выбора ветви. По мере увеличения числа посещений ветви коэффициент $1/(n + 1)$ постепенно уменьшается. Поэтому функция выбора ветви делает выбор, основываясь только на значении Q. Ветви с более высоким значением Q получают наибольшее количество посещений.

При небольшом количестве посещений ветви возможно все. Она может иметь как низкое, так и высокое значение Q. Однако мы не можем доверять ее оценке, основываясь на небольшом количестве посещений. Если вы делаете выбор на основании самого высокого значения Q, то можете выбрать ветвь с одним-единственным посещением и гораздо более низкой истинной ценностью. Вот почему при выборе следует опираться на количество посещений. Это гарантирует выбор ветви с высоким приблизительным значением ценности *и* надежной оценкой.

Листинг 14.9 ❖ Выбор хода с наибольшим количеством посещений

```
class ZeroAgent(Agent):
...
    def select_move(self, game_state):
...
        return max(root.moves(), key=root.visit_count)
```

В отличие от других агентов для игры с самим собой, описанных в этой книге, `ZeroAgent` не предусматривает особой логики пропуска хода, поскольку мы включили этот вариант в дерево поиска. Таким образом, мы можем обращаться с ним так же, как с любым другим ходом.

После реализации `ZeroAgent` можно реализовать функцию `simulate_game` для запуска игры бота с самим собой.

Листинг 14.10 ❖ Симуляция игры бота с самим собой

```
def simulate_game(
        board_size,
        black_agent, black_collector,
        white_agent, white_collector):
    print('Starting the game!')
    game = GameState.new_game(board_size)
    agents = {
        Player.black: black_agent,
        Player.white: white_agent,
    }
```

```
black_collector.begin_episode()
white_collector.begin_episode()
while not game.is_over():
    next_move = agents[game.next_player].select_move(game)
    game = game.apply_move(next_move)

game_result = scoring.compute_game_result(game)
if game_result.winner == Player.black:
    black_collector.complete_episode(1)
    white_collector.complete_episode(-1)
else:
    black_collector.complete_episode(-1)
    white_collector.complete_episode(1)
```


14.3. Обучение

Целевым значением для выхода функции ценности является 1 в случае победы агента и –1 в случае его поражения. Усредняя результаты множества игр, мы получаем значение, находящееся между этими экстремумами, которое отражает вероятность выигрыша нашего бота. Такая же схема использовалась в случае с Q-обучением (глава 11) и обучением методом типа «актор–критик» (глава 12).

Выход функции политики имеет некоторые отличия. Так же, как в случае с обучением на основе политики (глава 10) и обучением методом типа «актор–критик» (глава 12), нейронная сеть имеет выход, предоставляющий распределение вероятностей для допустимых ходов. При использовании алгоритма обучения на основе политики мы учили сеть повторять ходы, выбранные агентом (в выигранных им партиях). Система AGZ работает немного иначе. Обучение ее сети основано на количестве посещений каждого из ходов в процессе поиска по дереву.

Чтобы понять, как это может повысить уровень игры бота, подумайте о принципе работы алгоритма поиска по дереву методом Монте-Карло. Предположим, у нас есть функция ценности, которая хотя бы приблизительно может отличить выигрышную позицию от проигрышной. Теперь представьте, что мы полностью отбрасываем априорные вероятности и запускаем алгоритм поиска. Алгоритм поиска задуман так, чтобы самым перспективным ветвям уделялось больше времени. Логика выбора ветвей делает это возможным: благодаря компоненту Q формулы UCT ветви с высокой ценностью выбираются чаще. При наличии неограниченного времени на поиск мы сумели бы в конечном итоге выбрать самые лучшие ходы.

После проведения достаточного количества раундов поиска по дереву мы уже можем полагаться на значения счетчика посещений. Мы знаем, какие из ходов являются хорошими, а какие плохими, потому что уже проверили, что произойдет в случае их совершения. Таким образом, значения счетчиков посещений становятся целевыми значениями для обучения функции априорной вероятности.

Функция априорной вероятности пытается предсказать, где алгоритм поиска по дереву проводил бы большую часть своего времени, если бы это время не было ограничено. Вооружившись функцией, обученной на предыдущих прогонах, алгоритм поиска по дереву может экономить время и сразу переходить к исследованию наиболее важных ветвей. При наличии точной функции априорной вероятности наш алгоритм поиска в ходе небольшого количества развертываний

может получать результаты, сопоставимые с результатами более медленного алгоритма поиска, предусматривающего проведение гораздо большего количества развертываний. В некотором смысле можно сказать, что сеть «запоминает» то, что произошло в ходе предыдущих эпизодов поиска, и использует эти знания для пропуска бесперспективных ветвей.

Чтобы провести обучение таким способом, после каждого хода нам потребуется сохранять количество прогонов алгоритма поиска. В предыдущих главах мы использовали универсальный `ExperienceCollector`, который можно применять к различным реализациям алгоритма обучения с подкреплением. Однако в случае AGZ нам придется создать специальный коллектор, структура которого слегка отличается от универсального.

Листинг 14.11 ❖ Специальный коллектор данных опыта для обучения в стиле AGZ

```
class ZeroExperienceCollector:
    def __init__(self):
        self.states = []
        self.visit_counts = []
        self.rewards = []
        self._current_episode_states = []
        self._current_episode_visit_counts = []

    def begin_episode(self):
        self._current_episode_states = []
        self._current_episode_visit_counts = []

    def record_decision(self, state, visit_counts):
        self._current_episode_states.append(state)
        self._current_episode_visit_counts.append(visit_counts)

    def complete_episode(self, reward):
        num_states = len(self._current_episode_states)
        self.states += self._current_episode_states
        self.visit_counts += self._current_episode_visit_counts
        self.rewards += [reward for _ in range(num_states)]

        self._current_episode_states = []
        self._current_episode_visit_counts = []
```

Листинг 14.12 ❖ Передача решения в коллектор данных опыта

```
class ZeroAgent(Agent):
...
    def select_move(self, game_state):
...
        if self.collector is not None:
            root_state_tensor = self.encoder.encode(game_state)
            visit_counts = np.array([
                root.visit_count(
                    self.encoder.decode_move_index(idx))
                for idx in range(self.encoder.num_moves())
            ])
            self.collector.record_decision(
                root_state_tensor, visit_counts)
```

Выход функции политики нашей нейронной сети использует функцию активации softmax. Как говорилось ранее, данная функция гарантирует, что сумма ее значений равна 1. Для обучения мы также должны гарантировать то, что сумма целевых значений равна 1. Для этого нужно разделить все значения счетчиков посещений на их общую сумму. Пример такой операции, называемой *нормализацией*, представлен на рис. 14.5.

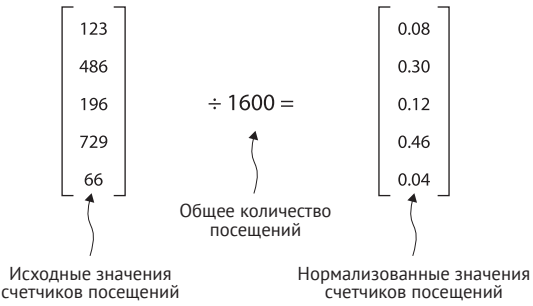


Рис. 14.5 ❖ Нормализация вектора. В ходе игры бота с самим собой мы отслеживаем количество посещений каждого хода. Для обучения мы должны нормализовать вектор так, чтобы сумма его значений была равна 1

В остальном процесс обучения аналогичен алгоритму типа «актор–критик» из главы 12. Следующий листинг демонстрирует его реализацию.

Листинг 14.13 ❖ Обучение комбинированной сети

```
class ZeroAgent(Agent):
...
    def train(self, experience, learning_rate, batch_size):
        num_examples = experience.states.shape[0]

        model_input = experience.states

        visit_sums = np.sum(
            experience.visit_counts, axis=1).reshape(
            (num_examples, 1))
        action_target = experience.visit_counts / visit_sums

        value_target = experience.rewards

        self.model.compile(
            SGD(lr=learning_rate),
            loss=['categorical_crossentropy', 'mse'])
        self.model.fit(
            model_input, [action_target, value_target],
            batch_size=batch_size)
```

Параметры `learning_rate` и `batch_size` обсуждались в главе 10.

Нормализация значений счетчиков посещений. Вызов функции `np.sum` с использованием `axis=1` выполняет построчное суммирование элементов матрицы. Вызов функции `reshape` реорганизует эти суммы в соответствующие строки. После этого мы можем разделить исходные значения счетчиков посещений на их общее количество.

Общий цикл обучения с подкреплением аналогичен описанному в главах с 9 по 12.

1. Сгенерируйте большое количество игр бота с самим собой.
2. Обучите модель на данных опыта.

3. Протестируйте обновленную модель в игре против ее предыдущей версии.
4. Если новая версия заметно сильнее, переключитесь на новую версию.
5. Если нет, сгенерируйте еще одну партию игр бота с самим собой и попробуйте снова.
6. Повторите этот цикл нужное количество раз.

В листинге 14.14 представлена реализация одного цикла данного процесса. Учтите, что вам потребуется сгенерировать очень большое количество игр бота с самим собой, чтобы с нуля создать сильный ИИ для игры в го. Для достижения сверхчеловеческого уровня игры системе AlphaGo Zero потребовалось сыграть около 5 млн игр с самой собой.

Листинг 14.14 ❖ Один цикл обучения с подкреплением

```
board_size = 9
encoder = zero.ZeroEncoder(board_size)

board_input = Input(shape=encoder.shape(), name='board_input')
pb = board_input
for i in range(4):
    pb = Conv2D(64, (3, 3),
        padding='same',
        data_format='channels_first',
        activation='relu')(pb)

policy_conv = \
    Conv2D(2, (1, 1),
        data_format='channels_first',
        activation='relu')(pb)
policy_flat = Flatten()(policy_conv)
policy_output = \
    Dense(encoder.num_moves(), activation='softmax')(
        policy_flat)

value_conv = \
    Conv2D(1, (1, 1),
        data_format='channels_first',
        activation='relu')(pb)
value_flat = Flatten()(value_conv)
value_hidden = Dense(256, activation='relu')(value_flat)
value_output = Dense(1, activation='tanh')(value_hidden)

model = Model(
    inputs=[board_input],
    outputs=[policy_output, value_output])

black_agent = zero.ZeroAgent(
    model, encoder, rounds_per_move=10, c=2.0)
white_agent = zero.ZeroAgent(
    model, encoder, rounds_per_move=10, c=2.0)
c1 = zero.ZeroExperienceCollector()
c2 = zero.ZeroExperienceCollector()
black_agent.set_collector(c1)
white_agent.set_collector(c2)

for i in range(5):
```

Создайте сеть с четырьмя сверточными слоями. Для создания сильного бота можно добавить множество дополнительных слоев.

Добавьте в сеть выход функции политики.

Добавьте в сеть выход функции ценности.

Здесь мы используем 10 раундов на ход для ускорения работы демонстрационной версии. В ходе реального обучения потребуется гораздо большее их количество. Система AGZ использовала 1600.

Сгенерируйте пять игр перед началом обучения. В ходе реального обучения вам потребуются партии, состоящие из тысяч игр.

```
    simulate_game(board_size, black_agent, c1, white_agent, c2)

exp = zero.combine_experience([c1, c2])
black_agent.train(exp, 0.01, 2048)
```

14.4. Повышение эффективности разведки с помощью шума Дирихле

Процесс обучения с подкреплением на основе игры бота с самим собой предполагает значительный элемент случайности. Ваш бот может выбрать довольно странное направление развития, особенно на ранних этапах процесса обучения. Для предотвращения застревания бота важно обеспечить некоторую случайность. Тогда, даже если бот зациклится на бесперспективном ходе, у него будет шанс найти более выигрышный вариант. В этом разделе мы рассмотрим один из приемов, использованных системой AGZ для обеспечения качественной разведки.

В предыдущих главах мы применяли несколько различных методов для внесения разнообразия в выбор бота. Например, в главе 9 мы создавали случайную выборку из выходных данных политики нашего бота, а в главе 11 использовали ε-жадную политику, позволяющую боту в некотором количестве случаев (ε) полностью игнорировать свою модель и выбирать ход случайным образом. В обоих случаях мы добавляли элемент случайности в момент принятия ботом решения. Система AGZ использует другой метод, чтобы внести элемент случайности на более раннем этапе поиска.

Представьте, что на каждом этапе мы искусственно увеличиваем значение априорной вероятности одного или двух случайно выбранных ходов. На начальных стадиях процесса поиска априорная вероятность определяет ветви для исследования, поэтому эти ходы получат дополнительные посещения. Если они окажутся плохими, алгоритм поиска быстро переключится на другие ветви, и никакого вреда не будет. Однако этот подход гарантирует, что каждый ход время от времени получает несколько посещений, что позволяет алгоритму поиска учесть все варианты.

Система AGZ достигает аналогичного эффекта, добавляя шум, т. е. прибавляя небольшие случайные числа к значениям априорной вероятности в корне каждого дерева поиска. Шум, создаваемый на основе *распределения Дирихле*, позволяет достичь описанного ранее результата: значение априорной вероятности некоторых ходов искусственно увеличивается, а другие остаются без изменений. В этом разделе мы обсудим свойства распределения Дирихле и посмотрим, как генерировать на его основе шум с помощью библиотеки NumPy.

В данной книге мы неоднократно использовали распределение вероятностей для игровых ходов. При создании выборки из такого распределения мы получаем определенный ход. Распределение Дирихле – это распределение вероятностей для распределения вероятностей: при создании выборки из распределения Дирихле вы получаете другое распределение вероятностей. Функция NumPy `np.random.dirichlet` генерирует выборки из распределения Дирихле. Она принимает в качестве аргумента вектор и возвращает вектор той же размерности. В следующем листинге показано несколько примеров: результатом является вектор, сумма значений элементов которого всегда равна 1, т. е. результат является действительным распределением вероятности.

Листинг 14.15 ❖ Использование функции `np.random.dirichlet` для создания выборки из распределения Дирихле

```
>>> import numpy as np
>>> np.random.dirichlet([1, 1, 1])
array([0.1146, 0.2526, 0.6328])
>>> np.random.dirichlet([1, 1, 1])
array([0.1671, 0.5378, 0.2951])
>>> np.random.dirichlet([1, 1, 1])
array([0.4098, 0.1587, 0.4315])
```

Мы можем влиять на результат распределения Дирихле с помощью параметра *концентрации*, который обычно обозначается буквой α. Когда значение α близко к 0, распределение Дирихле будет генерировать «комковатые» векторы, в которых большинство значений близко к 0 и лишь несколько элементов отличаются большими значениями. При больших значениях α выборка будет «гладкой», т. е. значения будут различаться в гораздо меньшей степени. Следующий листинг демонстрирует эффект изменения параметра концентрации.

Листинг 14.16 ❖ Создание выборки из распределения Дирихле при значении α, близком к нулю

```
>>> import numpy as np

>>> np.random.dirichlet([0.1, 0.1, 0.1, 0.1])
array([0.    , 0.044 , 0.7196, 0.2364])
>>> np.random.dirichlet([0.1, 0.1, 0.1, 0.1])
array([0.0015, 0.0028, 0.9957, 0.    ])
>>> np.random.dirichlet([0.1, 0.1, 0.1, 0.1])
array([0.    , 0.9236, 0.0002, 0.0763])

>>> np.random.dirichlet([10, 10, 10, 10])
array([0.3479, 0.1569, 0.3109, 0.1842])
>>> np.random.dirichlet([10, 10, 10, 10])
array([0.3731, 0.2048, 0.0715, 0.3507])
>>> np.random.dirichlet([10, 10, 10, 10])
array([0.2119, 0.2174, 0.3042, 0.2665])
```

Создание выборки из распределения Дирихле при малом значении параметра концентрации дает «комковатые» результаты: большая часть массы сосредоточена в одном или двух элементах.

Создание выборки из распределения Дирихле при большом значении параметра концентрации. Масса равномерно распределена по всем элементам полученного в результате вектора.

В этом и заключается рецепт изменения значений априорных вероятностей. Выбрав малое значение параметра α, вы получите распределение, в котором несколько ходов имеют высокое значение вероятности, а остальные – близкое к нулю. Затем вы можете вычислить средневзвешенное значение истинных априорных вероятностей с помощью шума Дирихле. Система AGZ использовала значение параметра концентрации, равное 0,03.

14.5. Современные методы создания более глубоких нейронных сетей

Проектирование нейронных сетей является чрезвычайно актуальной темой исследований. Одна из важнейших проблем заключается в обеспечении стабильности процесса обучения при работе со все более глубокими сетями. В системе AlphaGo Zero была применена пара передовых методов, которые быстро набирают популярность. Их подробное обсуждение выходит за рамки данной книги, поэтому мы ограничимся лишь общим описанием.

14.5.1. Пакетная нормализация

В основе глубоких нейронных сетей лежит идея о том, что в процессе обучения каждый слой может приобретать все более высокоуровневое представление исходных данных. Но чем именно являются эти представления? Под ними мы подразумеваем то, что некоторое важное свойство исходных данных проявится в виде конкретного числового значения при активации конкретного нейрона в слое. Однако отображение фактических чисел является абсолютно произвольным. Например, если умножить каждое значение функции активации в слое на 2, это не приведет к потере какой-либо информации, поскольку мы просто изменили масштаб. В принципе, такое преобразование не влияет на способность сети к обучению.

Однако абсолютное значение функций активации может повлиять на практическую эффективность обучения. Идея *пакетной нормализации* заключается в том, чтобы сдвинуть значения функции активации каждого слоя и центрировать их около 0, а затем отмасштабировать их так, чтобы дисперсия была равна 1. На начальном этапе обучения мы не знаем, как будут выглядеть значения функции активации. Пакетная нормализация предоставляет схему для определения правильных параметров сдвига и масштабирования прямо в процессе обучения.

Вопрос о том, как пакетная нормализация улучшает процесс обучения, еще продолжает исследоваться. Изначально пакетная нормализация разрабатывалась с целью уменьшения *ковариантного сдвига*. В процессе обучения значения активации в слоях имеют тенденцию дрейфовать. Пакетная нормализация корректирует этот дрейф, уменьшая нагрузку на следующие слои. Однако последние исследования показывают, что ковариантный сдвиг может оказаться не таким важным, как считалось ранее, а ценность пакетной нормализации может заключаться в сглаживании функции потерь.

Несмотря на то что исследователи пока не вполне понимают, *почему* пакетная нормализация работает, в том, что она *работает*, они не сомневаются. Библиотека Keras предусматривает слой `BatchNormalization`, который можно добавить в сеть. Следующий листинг демонстрирует пример добавления пакетной нормализации к сверточному слою сети Keras.

Листинг 14.17 ❖ Добавление пакетной нормализации в сеть Keras

```
from keras.models import Sequential
from keras.layers import Activation, BatchNormalization, Conv2D

model = Sequential()
model.add(Conv2D(64, (3, 3), data_format='channels_first'))
model.add(BatchNormalization(axis=1))
model.add(Activation('relu'))
```

Ось должна соответствовать способу упорядочивания `data_format`. Для `channel_first` используйте `axis=1` (первая ось). Для `channel_last` используйте `axis=-/1` (последняя ось).

Нормализация происходит между сверткой и функцией активации relu.

14.5.2. Остаточные сети

Представьте, что вы успешно обучили нейронную сеть, содержащую три скрытых слоя. Что произойдет, если вы добавите в нее четвертый слой? Теоретически это должно привести к увеличению емкости сети. В худшем случае при работе с че-

тырехслойной сетью первые три слоя обучатся тому же самому, что и при работе с трехслойной сетью, а четвертый слой просто передаст входные данные в неизменном виде. Вы рассчитываете на то, что эта сеть узнает больше, а не меньше. По крайней мере, вы могли бы ожидать, что более глубокая сеть способна на переобучение (при котором она запоминает обучающий набор, но относительно плохо работает на новых примерах).

На самом деле так происходит не всегда. При обучении четырехслойной сети существует больше способов организации данных, чем в случае с трехслойной сетью. Иногда из-за странностей стохастического градиентного спуска на сложных поверхностях потерь вы можете добавить больше слоев и обнаружить, что сеть не способна даже переобучиться. Идея *остаточной сети* заключается в упрощении того, что пытается изучить дополнительный слой. Если три слоя хорошо справляются с изучением некоторой проблемы, мы можем заставить четвертый слой сосредоточиться на заполнении пробела между тем, что изучили первые три слоя, и целевым значением. (Этот пробел представляет собой *остаток*, отсюда и название сети.)

Чтобы это реализовать, мы суммируем *входные данные* дополнительных слоев с их *выходными данными*, как показано на рис. 14.6. Соединение предыдущего слоя со слоем, выполняющим суммирование, называется *прямой связью* (skip connection). Обычно остаточные сети организуются в небольшие блоки, каждый из которых содержит два или три слоя и прямую связь. Затем необходимое количество таких блоков объединяется в стек.

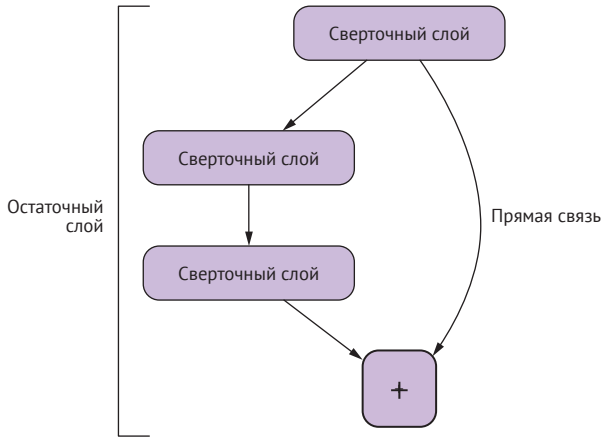


Рис. 14.6 ❖ Остаточный блок. Выход двух внутренних слоев прибавляется к выходу предыдущего слоя. Получается, что внутренние слои изучают разницу, или остаток, между целевым значением и тем, что изучил предыдущий слой

14.6. Дополнительные ресурсы

Если вы хотите поэкспериментировать с ботами в стиле AlphaGo Zero, к вашим услугам множество проектов с открытым исходным кодом, вдохновленных оригинальной статьей об алгоритме AGZ. Если вы хотите сыграть в го с ИИ сверх-

человеческого уровня или изучить его исходный код, то можете воспользоваться следующими ресурсами.

- Leela Zero – это реализация бота в стиле AGZ с открытым исходным кодом. Процесс игры бота с самим собой распределен: при наличии свободных вычислительных мощностей вы можете сгенерировать игры и загрузить их в качестве обучающих данных. На момент написания этой книги участники сообщества предоставили более 8 млн партий, благодаря чему бот Leela Zero уже достаточно силен, для того чтобы побеждать профессиональных игроков в го. Веб-сайт: **zero.sjeng.org**.
- Minigo – это еще одна реализация с открытым исходным кодом, написанная на языке Python с использованием библиотеки TensorFlow. Она полностью интегрирована с Google Cloud Platform, поэтому для проведения экспериментов вы можете использовать публичное облако Google. Веб-сайт: **github.com/tensorflow/minigo**.
- Группа Facebook AI Research реализовала алгоритм AGZ поверх своей платформы для обучения с подкреплением ELF. Результат, ELF OpenGo, теперь находится в свободном доступе, и это один из самых сильных ИИ для игры в го на сегодняшний день. Веб-сайт: **facebook.ai/developers/tools/elf**.
- Компания Tencent также реализовала и обучила бот в стиле AGZ, который получил название PhoenixGo. Этот бот еще известен как BensonDarr на сервере Fox Go, на котором он обыграл множество ведущих игроков мира. Веб-сайт: **github.com/Tencent/PhoenixGo**.
- Если вы предпочитаете шахматы, обратите внимание на движок Leela Chess Zero, который представляет собой адаптацию движка Leela Zero. Он уже достиг уровня человека-гроссмейстера, и поклонники шахмат оценили его захватывающую и нестандартную игру. Веб-сайт: **github.com/LeelaChessZero/lczero**.

14.7. Заключение

На этом мы заканчиваем наше знакомство с передовыми методами, лежащими в основе современных ИИ для игры в го. Теперь дело за вами. Поэкспериментируйте с собственным ботом для игры в го или попробуйте применить описанные методы к другим играм.

Кроме того, постарайтесь не ограничиваться сферой игр. Читая о новейшем способе применения машинного обучения, подумайте над следующими вопросами:

- Какова структура модели или нейронной сети?
- Какова функция потери или цель обучения?
- Как организован процесс обучения?
- Как кодируются входные и выходные данные?
- Как модель сочетается с традиционными алгоритмами или программными приложениями?

Мы надеемся, что нам удалось вдохновить вас на проведение собственных экспериментов с глубоким обучением, будь то в сфере игр или в любой другой области.

14.8. Резюме

- Система AlphaGo Zero использует одну нейронную сеть с двумя выходами. Один выход показывает важные ходы, а другой – лидирующего игрока.
- Алгоритм поиска по дереву AlphaGo Zero похож на алгоритм поиска по дереву методом Монте-Карло, но имеет два основных отличия. Вместо использования случайных игр для оценки позиции он опирается исключительно на нейронную сеть. Кроме того, он использует нейронную сеть для направления алгоритма поиска к новым ветвям.
- Нейронная сеть AlphaGo Zero обучается на основе количества посещений конкретных ходов в процессе поиска по дереву. Таким образом, ее обучение направлено на улучшение поиска по дереву, а не на выбор ходов.
- *Распределение Дирихле* – это распределение вероятностей для распределения вероятностей. Параметр концентрации влияет на степень «комковатости» получающегося в результате распределения вероятностей. AlphaGo Zero использует шум Дирихле для добавления элемента случайности в процесс поиска, чтобы гарантировать периодическое исследование всех ходов.
- Пакетная нормализация и остаточные сети – это два современных метода, помогающих обучать очень глубокие нейронные сети.

Приложение A

Математические основы

Машинное обучение невозможно без использования математики, в частности без линейной алгебры и математического анализа. Цель данного приложения состоит в том, чтобы познакомить вас с математическими основами, необходимыми для понимания приведенных в этой книге примеров кода. Полностью осветить эти масштабные темы не представляется возможным, поэтому далее перечислены дополнительные ресурсы на случай, если вы захотите разобраться в них подробнее.

Если вы уже знакомы с передовыми методами машинного обучения, то можете смело пропустить это приложение.

Дополнительные ресурсы

В этой книге мы можем осветить лишь некоторые математические основы. Тем, кого интересует математика, применяемая в машинном обучении, мы рекомендуем следующие книги:

- для знакомства с линейной алгеброй мы рекомендуем книгу Шелдона Экслера «Linear Algebra Done Right» (Springer, 2015);
- исчерпывающим практическим руководством по математическому анализу, в том числе по векторному исчислению, является книга Джеймса Стюарта «Calculus: Early Transcendentals» (Cengage Learning, 2015);
- если вы хотите досконально разобраться в теории того, как и почему работает математический анализ, трудно найти что-то лучше классической работы Уолтера Рудина «Основы математического анализа» (СПб.: Лань, 2004).

Векторы, матрицы, и не только: основные конструкции линейной алгебры

Линейная алгебра предусматривает такие инструменты для обработки массивов данных, как *векторы*, *матрицы* и *тензоры*. Все эти объекты можно представить в Python с помощью типа данных NumPy `array`.

Линейная алгебра имеет фундаментальное значение для машинного обучения. В этом разделе рассматриваются самые основные операции с акцентом на их реализацию в NumPy.

Векторы: одномерные данные

Вектор представляет собой одномерный массив чисел. Размер массива является размерностью вектора. Для представления векторов в коде Python используются массивы NumPy.

Приведенное определение вектора не является математически точным, однако для целей данной книги оно вполне пригодно.

Список чисел можно преобразовать в массив NumPy с помощью функции `np.array`. Атрибут `shape` позволяет проверить размерность:

```
>>> import numpy as np
>>> x = np.array([1, 2])
>>> x
array([1, 2])
>>> x.shape
(2,)
>>> y = np.array([3, 3.1, 3.2, 3.3])
>>> y
array([3. , 3.1, 3.2, 3.3])
>>> y.shape
(4,)
```

Обратите внимание на то, что атрибут `shape` всегда является кортежем, поскольку массивы могут быть многомерными, как будет показано в следующем разделе.

Доступ к отдельным элементам вектора можно получить, как в случае со списком Python:

```
>>> x = np.array([5, 6, 7, 8])
>>> x[0]
5
>>> x[1]
6
```

Над векторами можно производить некоторые основные алгебраические операции. Мы можем сложить два вектора одной размерности. Результатом будет третий вектор той же размерности. Каждый элемент итогового вектора представляет собой сумму соответствующих элементов исходных векторов:

```
>>> x = np.array([1, 2, 3, 4])
>>> y = np.array([5, 6, 7, 8])
>>> x + y
array([ 6,  8, 10, 12])
```

Также можно выполнить поэлементное умножение двух векторов с помощью оператора *. (В данном случае слово «поэлементное» означает отдельное перемножение пар соответствующих элементов.)

```
>>> x = np.array([1, 2, 3, 4])
>>> y = np.array([5, 6, 7, 8])
>>> x * y
array([ 5, 12, 21, 32])
```

Поэлементное произведение еще называется *произведением Адамара*.

Кроме того, можно умножить вектор на значение с плавающей запятой (или *скаляр*). В данном случае мы умножаем каждое значение в векторе на этот скаляр:

```
>>> x = np.array([1, 2, 3, 4])
>>> 0.5 * x
array([0.5, 1. , 1.5, 2. ])
```

Также векторы поддерживают третью операцию, называемую *скалярным*, или *внутренним*, *произведением*. При выполнении скалярного произведения мы перемножаем пары соответствующих элементов и суммируем результаты. Таким образом, скалярное произведение двух векторов представляет собой одно число с плавающей запятой. Скалярное произведение вычисляется с помощью функции NumPy `np.dot`. В Python 3.5 и более поздних версиях оператор `@` делает то же самое. (В этой книге мы используем функцию `np.dot`.)

```
>>> x = np.array([1, 2, 3, 4])
>>> y = np.array([4, 5, 6, 7])
>>> np.dot(x, y)
60
>>> x @ y
60
```

Матрицы: двумерные данные

Двумерный массив чисел называется *матрицей*. Матрицы также можно представить с помощью массивов NumPy. В этом случае если вы передадите функции `np.array` список списков, то получите двумерную матрицу:

```
>>> x = np.array([
  [1, 2, 3],
  [4, 5, 6]
 ])
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x.shape
(2, 3)
```

Обратите внимание на то, что формой матрицы является двухэлементный кортеж: первый элемент соответствует количеству строк, а второй – количеству столбцов. Доступ к отдельным элементам можно получить с помощью двойного индекса: первый будет обозначать строку, второй – столбец. Кроме того, NumPy позволяет передать эти индексы в формате `[строка, столбец]`. Оба варианта эквивалентны друг другу:

```
>>> x = np.array([
  [1, 2, 3],
  [4, 5, 6]
 ])
>>> x[0][1]
2
>>> x[0, 1]
2
>>> x[1][0]
```

```
4
>>> x[1, 0]
4
```

Мы также можем извлечь целую строку из матрицы и получить вектор:

```
>>> x = np.array([
  [1, 2, 3],
  [4, 5, 6]
 ])
>>> y = x[0]
>>> y
array([1, 2, 3])
>>> y.shape
(3,)
```

Для извлечения столбца можно использовать нотацию `[:, n]`. Если это поможет, думайте о : как об операторе Python для разрезания списков. Таким образом, нотация `[:, n]` означает «извлеки все строки, но только в столбце *n*». Вот пример:

```
>>> x = np.array([
  [1, 2, 3],
  [4, 5, 6]
 ])
>>> z = x[:, 1]
>>> z
array([2, 5])
```

Как и векторы, матрицы поддерживают поэлементное сложение, поэлементное умножение и скалярное произведение:

```
>>> x = np.array([
  [1, 2, 3],
  [4, 5, 6]
 ])
>>> y = np.array([
  [3, 4, 5],
  [6, 7, 8]
 ])
>>> x + y
array([[ 4,  6,  8],
       [10, 12, 14]])
>>> x * y
array([[ 3,  8, 15],
       [24, 35, 48]])
>>> 0.5 * x
array([[0.5, 1. , 1.5],
       [2. , 2.5, 3. ]])
```

Тензоры 3-го ранга

Игра в го предполагает использование сетки. Это же касается шахмат, шашек и множества других классических игр. Любая точка на этой сетке может содержать одну из множества различных игровых фигур. Как представить содержимое доски

в качестве математического объекта? Одно из решений заключается в представлении доски в виде стека матриц, размер которых соответствует размеру игровой доски.

Каждая отдельная матрица в стеке называется *плоскостью*, или *каналом*. Каждый канал представляет один из типов фигур, которые могут находиться на игровой доске. В случае с го мы можем иметь один канал для черных камней и второй – для белых (см. рис. А.1). В случае с шахматами у нас может быть канал для пешек, канал для слонов, канал для коней и т. д. Весь стек матриц можно представить в виде одного трехмерного массива под названием *тензор 3-го ранга*.

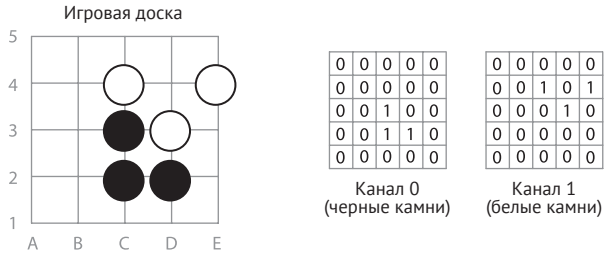


Рис. А.1 ❖ Представление доски для игры в го размером 5×5 с помощью двухплоскостного тензора. Мы используем один канал для черных камней и отдельный канал – для белых. Таким образом, для представления доски используется тензор 2×5×5

Другим распространенным случаем является представление изображения. Допустим, нам требуется представить изображение размером 128×64 пикселов с помощью массива NumPy. В этом случае мы начинаем с сетки, соответствующей пикселам изображения. В компьютерной графике цвет, как правило, разбивается на красный, зеленый и синий компоненты. Таким образом, мы можем представить это изображение с помощью тензора 3×128×64: у нас есть красный, зеленый и синий каналы.

Опять же, для создания тензора можно использовать `np.array`. Формой будет кортеж из трех компонентов, а для извлечения отдельных каналов можно использовать индексы:

```
>>> x = np.array([
 [[1, 2, 3],
  [2, 3, 4]],
 [[3, 4, 5],
  [4, 5, 6]]
])
>>> x.shape
(2, 2, 3)
>>> x[0]
array([[1, 2, 3],
       [2, 3, 4]])
>>> x[1]
array([[3, 4, 5],
       [4, 5, 6]])
```

Подобно векторам и матрицам, тензоры поддерживают поэлементное сложение, поэлементное умножение и скалярное произведение.

Сетку 8×8 с тремя каналами можно представить с помощью тензора 3×8×8 или тензора 8×8×3. Единственная разница заключается в индексации. При обработке тензоров с помощью библиотечных функций необходимо удостовериться в том, что функции знают, какую схему индексации вы используете. Библиотека Keras, применяемая для проектирования нейронных сетей, предусматривает для этого два параметра – `channels_first` и `channels_last`. В большинстве случаев разница не имеет значения: вам просто нужно выбрать один из вариантов и последовательно его использовать. В этой книге мы используем формат `channels_first`.

Если вам нужны доводы в пользу того или иного формата, имейте в виду, что некоторые графические процессоры NVIDIA особым образом оптимизированы под формат `channels_first`.

Тензоры 4-го ранга

Во многих местах книги мы используем тензор 3-го ранга для представления игровой доски. Для большей эффективности можно передать функции сразу несколько игровых досок. Для этого можно упаковать тензоры доски в четырехмерный массив NumPy, который представляет собой тензор 4-го ранга. Вы можете представить этот четырехмерный массив в виде списка тензоров 3-го ранга, каждый из которых представляет одну доску.

Матрицы и векторы – это просто частные случаи тензоров: матрица представляет собой тензор 2-го ранга, а вектор – тензор 1-го ранга. Тензор 0-го ранга – это простое число.

Несмотря на то что в этой книге не упоминаются тензоры выше 4-го ранга, NumPy может обрабатывать тензоры любого ранга. Несмотря на сложность визуализации многомерных тензоров, к ним применяется та же алгебра.

Математический анализ за пять минут: производные и нахождение максимума

В математическом анализе скорость изменения функции называется ее *производной*. В табл. А.1 приведено несколько примеров из реальной жизни.

Таблица А.1. Примеры производных

Количественный показатель	**Производная**
Пройденное расстояние	Скорость передвижения
Объем воды в ванне	Скорость спуска воды
Количество клиентов	Количество новых (или потерянных) клиентов

Производная не является фиксированным количественным значением: это еще одна функция, меняющаяся во времени или пространстве. При путешествии на автомобиле в разное время вы едете с разной скоростью. Однако ваша скорость всегда связана с преодолеваемым расстоянием. Если у вас есть точные данные о том, где вы были, то вы можете выяснить скорость, с которой проехали тот или иной отрезок пути. Это производная.

Когда значение функции увеличивается, ее производная положительна. Когда значение функции уменьшается, ее производная отрицательна. Эта закономерность продемонстрирована на рис. А.2. Учитывая это, мы можем использовать производную для нахождения *локального максимума* или *локального минимума*. На участке, где производная положительна, вы можете сдвинуться вправо и обнаружить большее значение функции. После преодоления максимума значение функции должно начать уменьшаться, при этом ее производная станет отрицательной. В этом случае вы можете немного сдвинуться влево. На локальном максимуме производная будет равна нулю. Для поиска минимума используется такая же логика, только движение происходит в противоположном направлении.

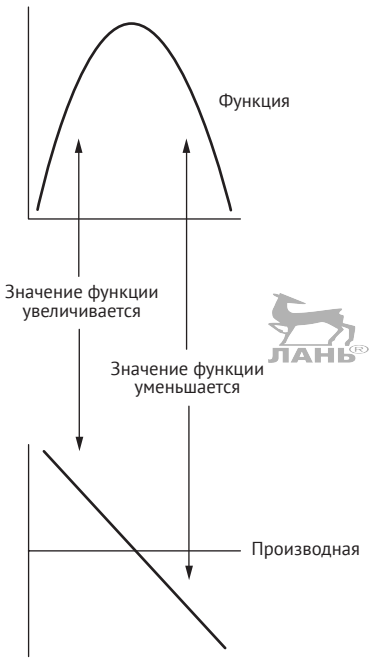


Рис. А.2 ❖ Функция и ее производная. Там, где производная положительна, значение функции увеличивается. Там, где производная отрицательна, значение функции уменьшается. Если производная равна нулю, функция находится в локальном минимуме или максимуме. Эта логика позволяет использовать производную для поиска локальных минимумов или максимумов

Многие функции, используемые в машинном обучении, принимают в качестве входных данных многомерный вектор и выдают одно число. Мы можем применить эту же идею для максимизации или минимизации такой функции. Производная такой функции представляет собой вектор той же размерности, что и ее вход, и называется *градиентом*. Для каждого элемента градиента знак сообщает о том, в каком направлении следует сдвигать соответствующую координату. Следование направлению градиента с целью максимизации функции называется

градиентным подъемом; если целью является минимизация функции, метод называется *градиентным спуском*.

В данном случае можно представить функцию в виде контурной поверхности. В любой точке градиент указывает в сторону самого крутого склона поверхности.

Для использования градиентного подъема требуется формула производной функции, которую необходимо максимизировать. Большинство простых алгебраических функций предусматривает производную, которую можно найти в любом учебнике по математическому анализу. Для вычисления производной сложной функции, созданной путем объединения множества простых функций, используется *правило дифференцирования сложной функции*. Такие библиотеки, как TensorFlow и Theano, задействуют это правило для автоматического вычисления производной сложных функций. При определении сложной функции в библиотеке Keras вам не придется самостоятельно выводить формулу градиента, Keras поручит эту задачу библиотеке TensorFlow или Theano.

Приложение Б

Алгоритм обратного распространения ошибки

В главе 5 были представлены последовательные нейронные сети, в частности сети прямого распространения. Мы кратко поговорили об *алгоритме обратного распространения ошибки*, который используется для обучения нейронных сетей. В этом приложении будут подробно описаны градиенты и процесс обновления параметров.

Сначала мы обсудим алгоритм обратного распространения ошибки для нейронных сетей прямого распространения, а затем поговорим о том, как применить его к более общим последовательным и непоследовательным сетям. Прежде чем углубляться в математические нюансы, давайте обсудим структуру и обозначения, которые мы будем использовать.

Пара слов о нотации

В этом разделе мы будем работать с нейронной сетью прямого распространения, содержащей l слоев. Каждый из этих слоев имеет сигмоидальную функцию активации. Веса i-го слоя обозначаются как W^i, а члены смещения – b^i. Буквой x обозначается мини-пакет входных данных размера k, подаваемый на вход сети, а буквой y – выходные данные сети. В данном случае x и y можно рассматривать в качестве векторов, однако все операции применимы и к мини-пакетам. Кроме того, мы будем использовать следующие обозначения:

- выход i-го слоя с функцией активации мы обозначим как y^{i+1}; т. е. $y^{i+1} = \sigma(W^i y^i + b^i)$. Обратите внимание на то, что y^{i+1} также является *входом* для слоя $i + 1$;
- выход i-го плотного слоя без функции активации мы обозначим как z^i; т. е. $z^i = W^i \cdot y^i + b^i$;
- с помощью этого удобного способа обозначения промежуточного выхода мы можем записать $z^i = W^i \cdot y^i + b^i$ и $y^{i+1} = \sigma(z^i)$. Обратите внимание, что эта нотация позволяет представить выход как $y = y^l$, а вход как $x = y^0$, однако здесь мы не будем этого делать;
- наконец, иногда мы будем использовать запись $f^i(y^i)$ для обозначения $\sigma(W^i y^i + b^i)$.

Алгоритм обратного распространения ошибки для сетей прямого распространения

С помощью перечисленных выше нотаций прямой проход i-го слоя нейронной сети можно описать следующим образом:

$$y^{i+1} = \sigma(W^i y^i + b^i) = f^i \circ y^i.$$

Мы можем использовать это определение рекурсивно для каждого слоя, чтобы записать предсказания так:

$$y = f^n \circ \ldots \circ f^1(x).$$

Поскольку мы вычисляем функцию потерь *Loss* на основании предсказаний y и меток $\hat{y}$, то можем разделить эту функцию похожим образом:

$$\text{Loss}(y, \hat{y}) = \text{Loss} \circ f^n \circ \ldots \circ f^1(x).$$

Вычисление и использование производной функции потерь, как показано здесь, выполняется с помощью *правила дифференцирования сложной функции*. Результат непосредственного применения этого правила к предыдущей формуле выглядит так:

$$\frac{d\text{Loss}}{dx} = \frac{d\text{Loss}}{df^n} \cdot \frac{df^n}{df^{n-1}} \cdots \frac{df^2}{df^1} \cdot \frac{df^1}{dx}.$$

Затем мы определяем *дельту* i-го слоя:

$$\Delta^i = \frac{d\text{Loss}}{df^n} \cdots \frac{df^{i+1}}{df^i}.$$

Теперь выразим дельты для описания *обратного прохода*:

$$\Delta^i = \Delta^{i+1} \frac{df^{i+1}}{df^i}.$$

Обратите внимание на то, что при вычислении обратного прохода индексы дельт уменьшаются. Формально вычисление обратного прохода структурно эквивалентно вычислению простого прямого прохода. Теперь перейдем к явному вычислению производных. Производные сигмоидальных и аффинно-линейных функций по входу выводятся следующим образом:

$$\sigma'(x) = \frac{d\sigma}{dx} = \sigma(x)(1 - \sigma(x));$$
$$\frac{d(Wx + b)}{dx} = W.$$

С помощью последних двух уравнений мы можем описать процесс обратного распространения ошибки Δ^{i+1} от (i+1)-го слоя к i-му слою:

$$\Delta^i = (W^i)^\top \cdot (\Delta^{i+1} \odot \sigma'(z^i)).$$

В этой формуле верхний индекс T обозначает транспонирование матрицы. Символ $\odot$ обозначает *произведение Адамара*, или поэлементное умножение двух векторов. Предыдущее вычисление состоит из двух частей: одна относится к плотному слою, а другая – к функции активации:

$$\Delta^\sigma = \Delta^{i+1} \odot \sigma'(z^i)\Delta^i = (W^i)^\top \cdot \Delta^\sigma.$$

Последним шагом является вычисление градиентов параметров W^i и b^i для каждого из слоев. После вычисления Δ^i мы можем считать градиенты параметров:

$$\Delta W^i = \frac{d\text{Loss}}{dW^i} = \Delta^i \cdot (y^i)^\top;$$
$$\Delta b^i = \frac{d\text{Loss}}{db^i} = \Delta^i.$$

Теперь, когда у нас есть значения ошибки, мы можем обновить параметры сети с помощью любого оптимизатора или правила обновления.

Обратное распространение ошибки для последовательных нейронных сетей

Вообще, последовательные нейронные сети могут содержать более интересные слои, чем те, которые мы обсуждали до сих пор. Например, вас могут заинтересовать сверточные слои, описанные в главе 6, или другие функции активации вроде softmax, также обсуждаемой в главе 6. Вне зависимости от того, какие слои используются в последовательной сети, процесс обратного распространения ошибки имеет одну и ту же общую логику. Если g^i обозначает прямой проход без функции активации, а Act^i – соответствующую функцию активации, то для передачи Δ^{i+1} на i-й слой требуется вычислить следующий переход:

$$\Delta^i = \frac{d\text{Act}^i}{dg^i}(z^i)\frac{dg^i}{dz^i}(y^i)\Delta^{i+1}.$$

Мы должны вычислить производную функции активации, оцененную по промежуточному выходу z^i, а также производную функции слоя g^i по входу i-го слоя. Зная все дельты, мы обычно можем легко определить градиенты для всех параметров слоя, как это делалось для весов и смещений в сети прямого распространения. Таким образом, каждый слой может передавать данные вперед, а значение ошибки назад, не имея представления о структуре окружающих слоев.

Обратное распространение ошибки для всех нейронных сетей в целом

В этой книге мы имели дело только с последовательными нейронными сетями, однако нелишним будет разобраться с тем, что происходит, когда мы не ограничены этими рамками. В непоследовательной сети слой имеет несколько выходов, несколько входов или и то, и другое.

Скажем, слой имеет m выходов. Прототипный подход может заключаться в разделении вектора на m частей. Локально для этого слоя прямой проход может быть разделен на k *отдельных* функций. При обратном проходе производная каждой из этих функций также может вычисляться отдельно и вносить равный вклад в значение дельты, передаваемой на предыдущей слой.

В ситуации, когда мы имеем дело с n входами и одним выходом, применяется несколько иной подход. Прямой проход вычисляется на основе n входных компонентов с помощью одной функции, выдающей одно значение. При обратном проходе мы получаем одну дельту от следующего слоя и должны вычислить n выходных дельт для передачи каждому из предыдущих n слоев. Эти производные могут быть вычислены независимо друг от друга и оценены по каждому из соответствующих входов.

Общий случай n входов и m выходов предполагает объединение двух предыдущих этапов. Каждая нейронная сеть, вне зависимости от сложности структуры и количества слоев, локально выглядит именно так.

Вычислительные сложности, связанные с обратным распространением ошибки

Теоретически обратное распространение ошибки можно рассматривать как применение правила дифференцирования сложной функции к определенному классу алгоритмов машинного обучения. Однако на практике существует множество нюансов, которые необходимо учесть при реализации этого алгоритма. Прежде всего, чтобы вычислить значения дельт и обновить градиенты того или иного слоя, нам потребуются соответствующие входные данные прямого прохода для оценки. Если мы просто отбросим результаты прямого прохода, нам придется снова вычислить их при осуществлении обратного прохода. Таким образом, будет лучше, если мы кешируем эти значения. В реализации, описанной в главе 5, каждый слой сохранял собственное состояние как для входных и выходных данных, так и для входных и выходных дельт. При создании сетей, обрабатывающих огромные объемы данных, важно, чтобы реализация была вычислительно-эффективной и не занимала слишком много памяти.

Еще один интересный нюанс заключается в повторном использовании промежуточных значений. Например, мы говорили о том, что в случае простой сети прямого распространения мы можем рассматривать аффинно-линейное преоб-

разование и сигмоидальную функцию активации как единое целое или разделить их на два слоя. Результат аффинно-линейного преобразования необходим для вычисления обратного прохода функции активации, поэтому мы должны сохранить эту информацию после выполнения прямого прохода. С другой стороны, поскольку сигмоидальная функция не имеет параметров, мы вычисляем обратный проход за один этап:

$$\Delta^i = (W^i)^\top(\Delta^{i+1} \odot \sigma'(z^i)).$$

В плане вычислительной эффективности этот подход может оказаться более предпочтительным по сравнению с двухэтапным. Автоматическое выявление операций, которые можно выполнять одновременно, позволяет значительно увеличить скорость. В более сложных ситуациях (например, при работе с рекуррентными нейронными сетями, слои которых, по сути, запускают *циклы* с использованием входных данных из последнего этапа) управление промежуточным состоянием становится еще более важным.

Приложение B

Программы и серверы для игры го

В этом приложении рассматриваются различные способы игры в го, как в режиме онлайн, так и в режиме офлайн. Сначала мы обсудим процесс установки программ GNU Go и Pachi и запуска локальных игр против этих двух ботов. Затем поговорим о нескольких популярных го-серверах, на которых можно найти противников (людей и ботов) различного уровня.

Программы для игры в го

Начнем с установки программ для игры в го на компьютер. Мы познакомим вас с двумя классическими бесплатными программами, которые существуют уже много лет. И GNU Go, и Pachi основаны на классических методах разработки игрового ИИ, которые были частично рассмотрены в главе 4. Наша цель состоит не в том, чтобы разобраться в методологии этих инструментов, а в том, чтобы предоставить вам двух противников, которых можно использовать локально для проведения тестов и игры.

Подобно большинству других программ для игры в го, Pachi и GNU Go используют протокол Go Text Protocol (GTP), о котором мы говорили в главе 8. Обе эти программы можно запускать разными удобными для вас способами:

- их можно запустить из командной строки и играть в игры, обмениваясь GTP-командами. Этот режим использовался в главе 8 для организации игры наших ботов против программ GNU Go и Pachi;
- обе программы могут использовать *GTP-фронтенды*, графические пользовательские интерфейсы, которые делают игру с этими движками намного более увлекательной для человека.

GNU Go

Движок GNU Go, разработанный в 1989 году, является одной из старейших программ для игры в го. Последняя его версия была выпущена в 2009 году. Несмотря на то что в последнее время движок GNU Go практически не дорабатывался, он не теряет своей популярности в качестве противника для новичков на многих го-серверах. Кроме того, это один из самых мощных движков, основанных на прописанных правилах, что выгодно отличает его от ботов, созданных на базе алгоритма MMK и методов глубокого обучения. Установочный файл GNU Go для ОС

Windows, Linux и macOS можно найти на странице **www.gnu.org/software/gnugo/download.html**. Там вы найдете инструкции по установке GNU Go в качестве инструмента интерфейса командной строки (CLI) и ссылки на различные графические интерфейсы. Для установки инструмента CLI загрузите последние версии бинарных файлов GNU Go с сайта **ftp.gnu.org/gnu/gnugo**, распакуйте соответствующий tar-архив и следуйте инструкциям для вашей платформы из файла INSTALL и README. В качестве графических интерфейсов мы рекомендуем установить JagoClient для Windows и Linux с сайта **www.rene-grothmann.de/jago** и FreeGoban для macOS с сайта **www.sente.ch/software/goban/freegoban.html**. Для проверки результатов установки вы можете запустить следующую команду:

```
gnugo mode gtp
```

Движок GNU Go будет запущен в режиме GTP. Программа начнет новую игру на доске 19×19 и будет готова принимать входные данные из командной строки. Например, вы можете попросить GNU Go сгенерировать ход белых, введя `genmove white` и нажав клавишу **Enter**. Программа возвратит символ `=`, обозначающий допустимость команды, и координаты точки доски. Например, ответ может выглядеть так: `= C3`. В главе 8 мы использовали GNU Go в режиме GTP в качестве противника для наших ботов, основанных на методах глубокого обучения.

Если вы решите установить графический интерфейс, то сможете сразу проверить свои навыки в игре против GNU Go.

Pachi

Программу Pachi, которая является гораздо более сильной по сравнению с GNU Go, можно загрузить с сайта **pachi.or.cz**. Кроме того, исходный код Pachi и подробные инструкции по установке можно найти на GitHub по адресу **github.com/pasky/pachi**. Для проверки Pachi в действии выполните команду `pachi` в командной строке и введите `genmove black`, чтобы программа сгенерировала ход черных на доске 9×9.

Серверы для игры в го

Играть против го-ботов на компьютере интересно и полезно, однако онлайн-серверы для игры в го обеспечивают намного более богатый выбор противников, среди которых есть как люди, так и боты. Люди и боты могут создать аккаунт на этих платформах и участвовать в рейтинговых играх с целью повышения своего уровня игры и, как следствие, своего рейтинга. Людям это предоставляет доступ к более конкурентной и интерактивной игровой среде, а также возможность протестировать своих ботов в игре против оппонентов со всего мира. Список го-серверов можно найти на сайте **senseis.xmp.net/?GoServers**. Крупнейшими на данный момент го-серверами являются китайский, корейский и японский, но они не поддерживают англоязычную версию, поэтому далее мы перечислим три сервера с поддержкой англоязычных клиентов.

OGS

Online Go Server (OGS) – это прекрасно разработанная веб-платформа для игры в го, доступная по адресу: **online-go.com**. Именно этот сервер мы использовали в главе 8 и приложении Д для демонстрации процесса подключения бота. Сервер

OGS предусматривает множество функций, часто обновляется, поддерживается группой администраторов и является одним из самых популярных го-серверов в Западном полушарии. Кроме того, он нам очень нравится.

IGS

Internet Go Server (IGS), доступный по адресу: **pandanet-igs.com/communities/pandanet**, был создан в 1992 году и является одним из старейших го-серверов. Он не теряет своей популярности и с 2013 года имеет новый интерфейс. Это один из немногих го-серверов с собственным клиентом для Mac. IGS является одним из наиболее популярных турнирных го-серверов, имеющих глобальную пользовательскую базу.

Tygem

Корейский сервер *Tygem*, доступный по адресу **www.tygemgo.com**, вероятно, имеет самую большую базу пользователей из трех представленных здесь платформ. В любое время суток на нем можно найти тысячи игроков разного уровня. На нем также проводятся турниры, в которых принимают участие многие из сильнейших профессионалов мира (иногда анонимно).

Приложение Г

Обучение и развертывание ботов с помощью Amazon Web Services

В этом приложении мы поговорим об использовании облачного сервиса Amazon Web Services (AWS) для создания и развертывания моделей глубокого обучения. Полученные навыки пригодятся вам не только в случае с ботами для игры в го, но и будут полезны сами по себе. Далее вы узнаете о том, как:

- настроить виртуальный сервер с помощью AWS для обучения моделей глубокого обучения;
- проводить эксперименты с глубоким обучением в облаке;
- развернуть бота для игры в го с веб-интерфейсом на сервере для предоставления доступа к нему другим людям.

Несмотря на то что на момент написания этой книги AWS являлся самым крупным из облачных сервисов и обладал многими преимуществами, для этого приложения мы могли бы выбрать и другие сервисы. Поскольку крупные облачные провайдеры, как правило, предлагают похожие услуги, знакомство с одним из них облегчает дальнейшее освоение других.

Начните работу с AWS, перейдя на сайт **aws.amazon.com**, чтобы ознакомиться с широким ассортиментом предлагаемых им продуктов. Облачный сервис Amazon предоставляет доступ к огромному количеству продуктов, однако для целей данной книги нам будет достаточно лишь одного сервиса – Amazon Elastic Compute Cloud (EC2). EC2 предоставляет легкий доступ к виртуальным серверам в облаке. В зависимости от ваших потребностей вы можете указать для этих серверов или инстансов различные спецификации оборудования. Для эффективного обучения глубоких нейронных сетей вам потребуется доступ к мощным графическим процессорам. Несмотря на то что сервис AWS не всегда может предоставить графические процессоры последнего поколения, гибкая система покупки рабочего времени облачных графических процессоров является хорошим способом приступить к работе без существенных инвестиций в оборудование.

Первым делом вам нужно зарегистрировать аккаунт AWS на сайте **portal.aws.amazon.com/billing/signup**, заполните форму, показанную на рис. Г.1.

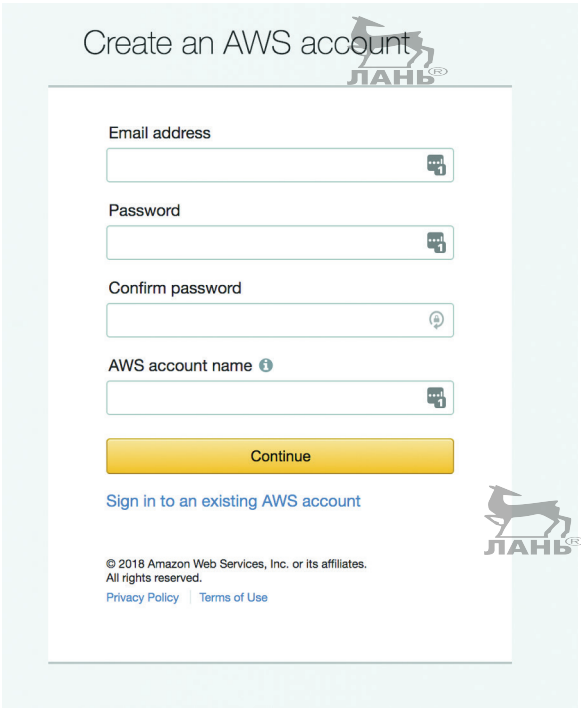


Рис. Г.1 ❖ Создание учетной записи на сервисе AWS

После регистрации щелкните по кнопке **Sign in to the Console** (Войти в консоль) в правом верхнем углу страницы **aws.amazon.com** и введите свои учетные данные. Это перенаправит вас на главную панель управления аккаунтом. В верхней строке меню выберите пункт **Services** (Сервисы), чтобы открыть панель с основными продуктами AWS. Выберите вариант **EC2** в категории **Compute** (Вычисления), как показано на рис. Г.2.

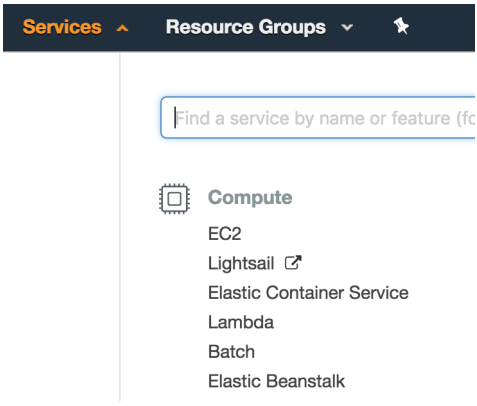


Рис. Г.2 ❖ Выбор сервиса **Elastic Cloud Compute (EC2)** в меню **Services** (Сервисы)

После этого откроется панель EC2 с обзором запущенных в настоящий момент инстансов и их статусов. Учитывая, что вы зарегистрировались только сейчас, вы должны увидеть 0 запущенных инстансов. Чтобы запустить новый инстанс, щелкните по кнопке **Launch Instance** (Запустить инстанс), как показано на рис. Г.3.

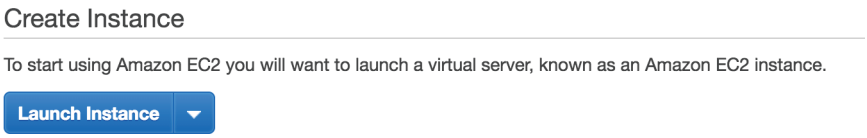


Рис. Г.3 ❖ Запуск нового инстанса AWS

На этом этапе вам будет предложено выбрать Amazon Machine Image (AMI) – образ программного обеспечения, которое будет доступно на запущенном инстансе. Чтобы быстро начать работу, можно выбрать AMI, предназначенный специально для приложений глубокого обучения. В левой боковой панели вы найдете каталог AWS Marketplace (см. рис. Г.4), содержащий множество полезных сторонних AMI.

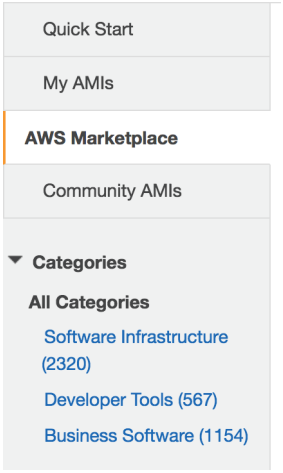


Рис. Г.4 ❖ Выбор каталога AWS Marketplace

В каталоге Marketplace выберите вариант Deep Learning AMI Ubuntu (см. рис. Г.5). Как следует из названия, этот инстанс работает на базе ОС Ubuntu Linux и уже содержит множество предустановленных полезных компонентов. Например, на этом AMI уже установлены библиотеки TensorFlow и Keras, а также все необходимые драйверы графического процессора. Таким образом, когда инстанс будет готов, вы сможете сразу приступить к использованию приложения глубокого обучения, не тратя времени и сил на установку программного обеспечения.

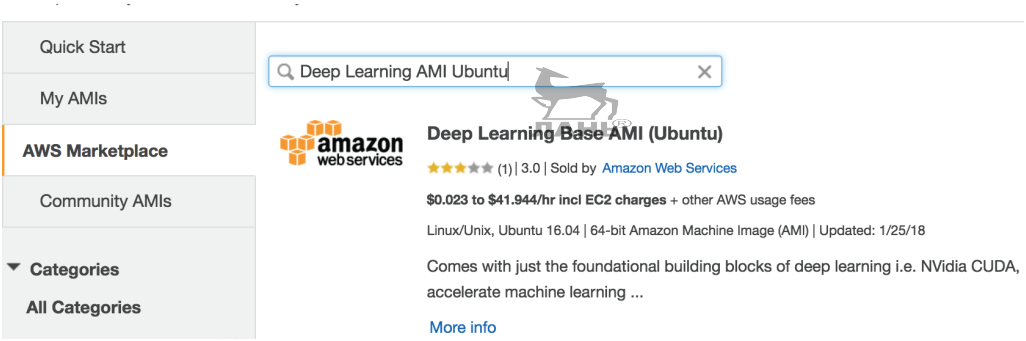


Рис. Г.5 ❖ Выбор AMI для глубокого обучения

Этот образ AMI стоит дешево, но не предоставляется бесплатно. Если вам нужен бесплатный инстанс, ищите образ с пометкой *free tier eligible*. Например, большинство образов AMI, представленных в разделе **Quick Start** (Быстрый старт) (см. рис. Г.4), *можно* использовать бесплатно.

После щелчка по кнопке **Select** (Выбрать) рядом с образом AMI отобразятся цены, соответствующие типу выбранного инстанса (см. рис. Г.6).

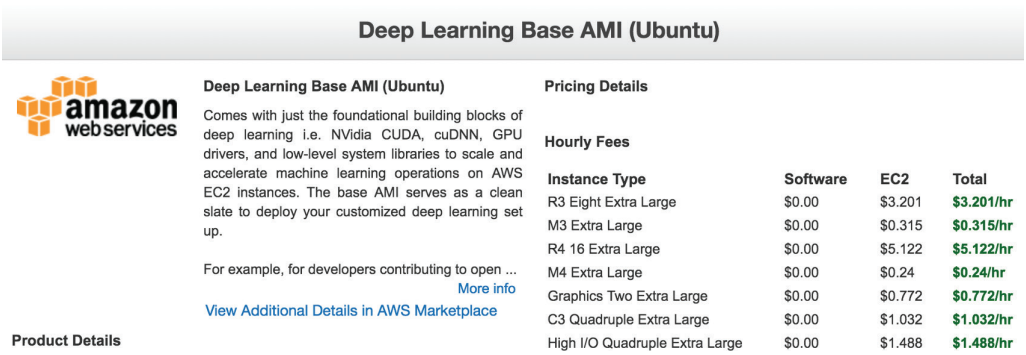


Рис. Г.6 ❖ Стоимость использования образа AMI для глубокого обучения в зависимости от выбранного инстанса

Теперь можно выбрать тип инстанса. На рис. Г.7 представлены все типы инстансов, оптимизированных для графических процессоров. Для начала можно выбрать вариант **p2.xlarge**, однако имейте в виду, что все инстансы графических процессоров являются сравнительно дорогими. Если вы просто хотите познакомиться с AWS и представленными здесь функциями, выберите для начала недорогой инстанс **t2.small**. Если вас интересует только развертывание и размещение моделей, вам вполне подойдет инстанс **t2.small**. Более дорогие инстансы графических процессоров потребуются только для обучения модели.

☐	GPU graphics	g3.4xlarge	16	122	EBS only
☐	GPU graphics	g3.8xlarge	32	244	EBS only
☐	GPU graphics	g3.16xlarge	64	488	EBS only
☐	GPU instances	g2.2xlarge	8	15	1 x 60 (SSD)
☐	GPU instances	g2.8xlarge	32	60	2 x 120 (SSD)
☐	GPU compute	p2.xlarge	4	61	EBS only
☐	GPU compute	p2.8xlarge	32	488	EBS only
☐	GPU compute	p2.16xlarge	64	732	EBS only

Рис. Г.7 ❖ Выбор нужного типа инстанса

После выбора типа инстанса вы можете запустить его, просто щелкнув по кнопке **Review and Launch** (Просмотр и запуск) в правом нижнем углу. Однако поскольку вам еще предстоит настроить некоторые параметры, выберите **Next: Configure Instance Details** (Далее: Настроить параметры инстанса). Шаги с 3 по 5 в открывшемся диалоговом окне можно пока пропустить, однако шаг 6 **Configure Security Group** (Настройка группы безопасности) заслуживает особого внимания. В AWS *группа безопасности* задает права доступа к инстансу, определяя *правила*. Вы предоставите следующие права доступа:

- прежде всего доступ к своему инстансу вы будете получать, входя в систему через SSH. SSH-порт 22 уже должен быть открыт (это единственное заданное правило для новых инстансов), однако вам нужно ограничить доступ и разрешить подключения только с локального компьютера. Это делается по соображениям безопасности. Чтобы никто другой не мог подключиться к вашему инстансу AWS, доступ предоставляется только по вашему IP. Для этого нужно выбрать вариант **My IP** (Мой IP) в меню **Source** (Источник);
- поскольку вам предстоит развернуть веб-приложение, а в дальнейшем и бота, способного подключаться к другим го-серверам, вам также нужно открыть HTTP-порт 80. Для этого сначала щелкните по кнопке **Add Rule** (Добавить правило) и задайте тип HTTP. При этом порт 80 будет выбран автоматически. Чтобы разрешить людям подключаться к вашему боту из любого места, выберите пункт **Anywhere** (Любой) в меню **Source** (Источник);
- HTTP-бот для игры в го из главы 8 использует порт 5000, поэтому вам нужно открыть и его. В ситуации, связанной с производством, вы развернули бы подходящий веб-сервер, прослушивающий порт 80 (который был настроен на предыдущем шаге), что перенаправило бы трафик на порт 5000. Чтобы не усложнять, мы жертвуем безопасностью ради удобства и открываем порт 5000 напрямую. Для этого можно добавить другое правило и выбрать вариант **Custom TCP Rule** (Собственное правило TCP) в меню **Type** (Тип) и **5000** в меню **Port Range** (Диапазон портов). Для HTTP-порта выберите вариант **Anywhere** (Любой) в меню **Source** (Источник). После этого появится предупреждение системы безопасности, которое вы можете проигнорировать, поскольку не работаете с конфиденциальными или личными данными и приложениями.

После выполнения перечисленных выше действий страница с настройками правил доступа должна выглядеть, как на рис. Г.8.

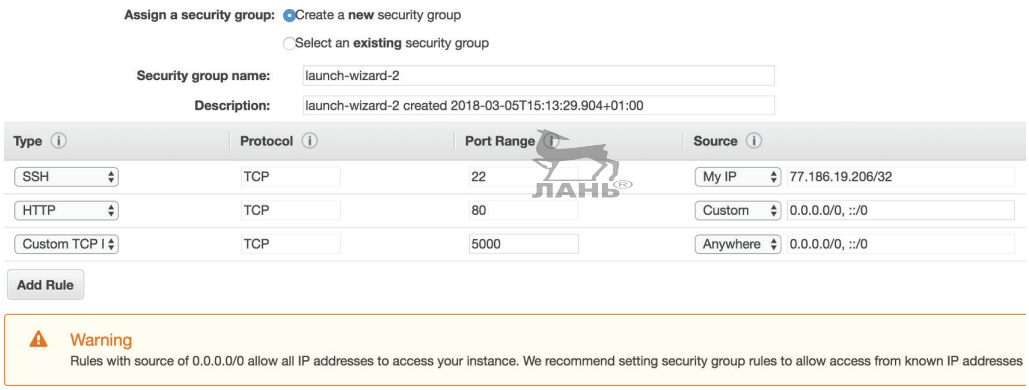


Рис. Г.8 ❖ Настройка групп безопасности для инстанса AWS

После настройки параметров безопасности щелкните по кнопке **Review and Launch** (Просмотр и запуск), а затем **Launch** (Запуск). Откроется диалоговое окно, предлагающее создать новую пару ключей или выбрать существующую. Выберите пункт **Create a new key pair** (Создать новую пару ключей) в раскрывающемся меню. Все, что вам нужно сделать, – это выбрать имя пары ключей, а затем скачать секретный ключ, щелкнув по кнопке **Download Key Pair** (Загрузить пару ключей). Загруженный файл ключа будет иметь имя, которое вы ему назначили, и расширение .pem. Сохраните этот закрытый ключ в надежном месте. Открытый ключ находится в ведении AWS и будет скопирован на инстанс, который вам предстоит запустить. С помощью закрытого ключа вы сможете подключиться к инстансу. После создания ключа вы сможете использовать его в будущем, выбрав пункт **Choose an Existing Key Pair** (Выбрать существующую пару ключей). На рис. Г.9 показано, как мы создали пару ключей с именем *maxpumperla_aws.pem*.

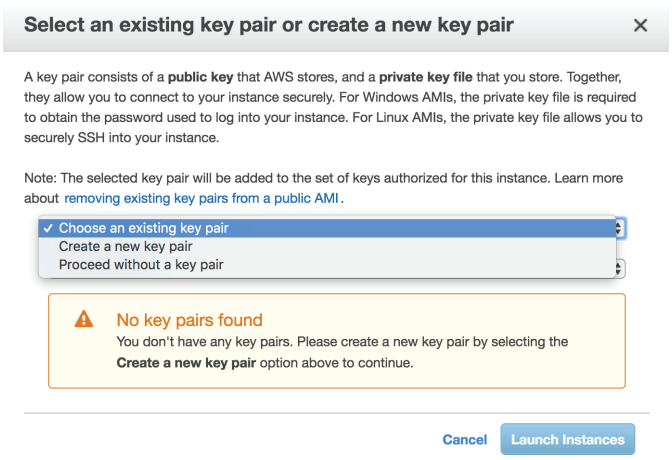


Рис. Г.9 ❖ Создание новой пары ключей для доступа к инстансу AWS

Это был последний этап настройки, и теперь можно запустить инстанс, щелкнув по кнопке **Launch Instance** (Запустить инстанс). В открывшемся окне **Launch Status** (Статус запуска) выберите **View Instances** (Показать инстансы) в правом нижнем углу. Откроется главная панель управления EC2, с которой мы начали (щелкнув по кнопке **Launch Instance** (Запустить инстанс)). В приведенном там списке вы должны увидеть свой инстанс. Через некоторое время состояние инстанса изменится на «running» («работает»), и рядом с ним появится зеленая точка. Это означает, что инстанс готов, и теперь к нему можно подключиться. Установите флажок слева от инстанса, чтобы активировать кнопку **Connect** (Подключиться) в верхней части. После щелчка по этой кнопке откроется окно, изображенное на рис. Г.10.

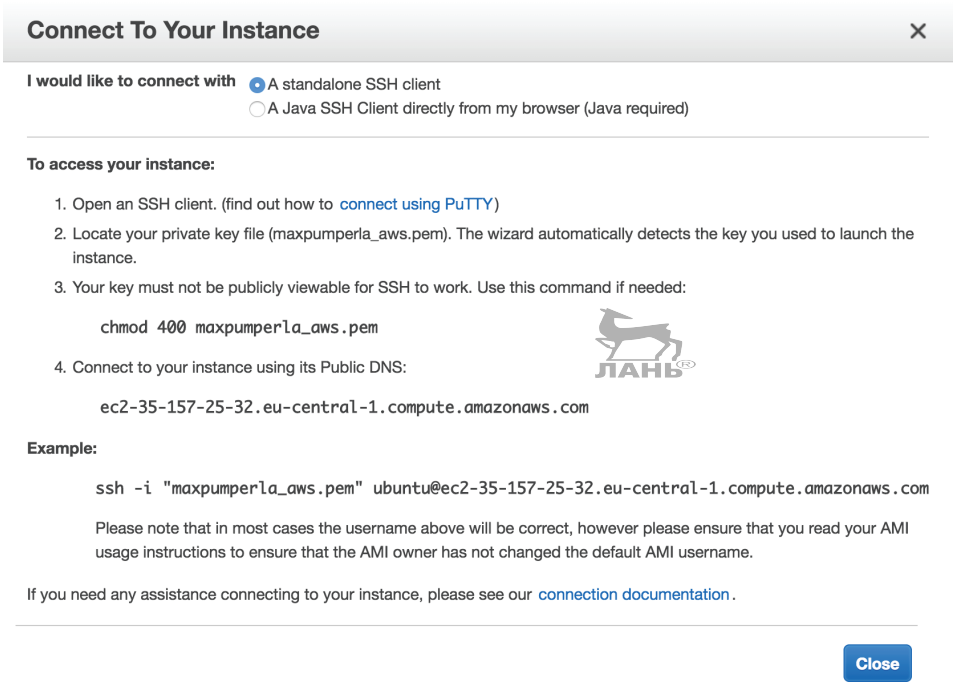


Рис. Г.10 ❖ Подключение к инстансу AWS

Это окно содержит большое количество полезной информации для подключения к инстансу, с которой следует внимательно ознакомиться. В частности, оно предоставляет инструкцию для подключения к инстансу с помощью протокола `ssh`. Если вы откроете терминал, а затем скопируете и вставите команду `ssh`, приведенную под словом **Example** (Пример), это позволит установить соединение с инстансом AWS. Данная команда выглядит следующим образом:

```
ssh -i "<full-path-to-secret-key-pem>" <username>@<public-dns-of-your-instance>
```

Работать с этой длинной командой может быть неудобно, особенно когда вам приходится одновременно управлять множеством инстансов или SSH-подключений к другим машинам. Чтобы облегчить себе жизнь, мы будем работать с фай-

лом конфигурации SSH. В средах UNIX этот файл конфигурации обычно хранится по адресу **~/.ssh/config**. В других системах путь к этому файлу может быть другим. При необходимости создайте этот файл в папке *.ssh*, а затем поместите в него следующий код:

```
Host aws
  HostName <public-dns-of-your-instance>
  User ubuntu
  Port 22
  IdentityFile <full-path-to-secret-key-pem>
```

После сохранения этого файла вы сможете подключиться к инстансу, введя команду `ssh aws` в терминал. При первом подключении система попросит вас подтвердить свое намерение. Введите `yes` и отправьте эту команду, нажав клавишу **Enter**. Ваш ключ будет навсегда добавлен в инстанс (что можно проверить с помощью команды `cat ~/.ssh/authorized_keys` для возврата безопасного хеша вашей пары ключей), и этот вопрос вам больше не зададут.

При первом успешном подключении к инстансу Deep Learning AMI Ubuntu AMI (в случае если вы выбрали именно его) вам будет предложено несколько сред Python на выбор. Вариантом, предусматривающим полную установку библиотек Keras и TensorFlow для Python 3.6, является `source activate tensorflow_p36` или `source activate tensorflow_p27`, если вы предпочитаете использовать Python 2.7. В оставшейся части этого приложения мы будем предполагать, что вы пропустили этот шаг и работаете с базовой версией Python, уже предусмотренной для выбранного инстанса.

Прежде чем приступать к запуску приложений на инстансе, давайте кратко обсудим процесс его отключения. Это важно, ведь если вы забудете отключить дорогостоящий инстанс, то в конце месяца можете получить счет на несколько сотен долларов. Чтобы отключить инстанс, выберите его (установив флажок рядом с его именем), щелкните по кнопке **Actions** (Действия) в верхней части страницы, а затем **Instance State** (Состояние инстанса) → **Terminate** (Прекратить работу). Прекращение работы инстанса ведет к его удалению вместе со всеми сохраненными в нем данными. Перед прекращением работы обязательно скопируйте все, что вам нужно, например обученную модель (далее мы покажем, как это делается). Другим вариантом является остановка работы инстанса с помощью кнопки **Stop** (Остановить), что позволяет в дальнейшем возобновить работу с помощью кнопки **Start** (Начать). Однако учтите, что в зависимости от объема хранилища, предусмотренного вашим инстансом, это может привести к потере данных. В этом случае вы увидите соответствующее предупреждение.

Обучение моделей на сервисе AWS

После всех приготовлений процесс запуска модели глубокого обучения на сервисе AWS ничем не отличается от процесса ее локального запуска. Сначала убедитесь в том, что инстанс содержит все необходимые данные и код. Самый простой способ это сделать – скопировать его туда безопасным способом с помощью команды `scp`. Например, на локальном компьютере вы можете выполнить следующие команды для запуска файла *endtoend.py*:

```
git clone https://github.com/maxpumperla/deep_learning_and_the_game_of_go
cd deep_learning_and_the_game_of_go
scp -r ./code aws:~/code
ssh aws
cd ~/code
python setup.py develop
cd examples
python end_to_end.py
```

Скопируйте код с локального компьютера на удаленный инстанс AWS.

Подключитесь к инстансу с помощью протокола ssh.

Установите библиотеку Python dlgo.

Запустите файл endtoend.py.

В этом примере мы предполагаем, что вы начинаете с нуля, клонируя наш репозиторий GitHub. На практике вы, скорее всего, решите провести собственный эксперимент. Для этого вам нужно будет создать глубокие нейронные сети для обучения и запустить нужные примеры. Приведенный выше пример *end_to_end.py* создает сериализованного бота глубокого обучения по адресу **../agents/deep_bot.h5** относительно папки *examples*. После запуска примера вы можете либо оставить модель на месте (для хостинга или дельнейшей доработки), либо извлечь ее из инстанса AWS и скопировать обратно на свой компьютер. Например, чтобы скопировать бота deep_bot.h5 с сервиса AWS на локальный компьютер, введите в терминал:

```
cd deep_learning_and_the_game_of_go/code
scp aws:~/code/agents/deep_bot.h5 ./agents
```

Таким образом, процесс обучения модели состоит из следующих этапов:

1) локальная подготовка и тестирование моделей глубокого обучения с помощью фреймворка *dlgo*;
2) безопасное копирование внесенных изменений на инстанс AWS;
3) подключение к удаленному компьютеру и проведение эксперимента;
4) после обучения оцените результаты, внесите корректировки и запустите новый экспериментальный цикл, начиная с этапа 1;
5) при желании скопируйте обученную модель на локальный компьютер для дальнейшего использования или обработайте ее иным способом.

Размещение бота на сервисе AWS с помощью протокола HTTP

В главе 8 мы говорили о подключении бота к веб-интерфейсу через протокол HTTP для предоставления доступа к нему другим людям. Недостаток этого подхода заключается в том, что он предполагает запуск веб-сервера Python на локальном компьютере. При этом для тестирования вашего бота другие люди должны иметь прямой доступ к вашему компьютеру. После развертывания веб-приложения на сервисе AWS и открытия необходимых портов (как это делалось при настройке инстанса) вы сможете предоставить другим людям доступ к боту, просто поделившись с ними соответствующим URL-адресом.

Запуск HTTP-фронтенда осуществляется так же, как и раньше. Для этого достаточно сделать следующее:

```
ssh aws
cd ~/code
python web_demo.py \
```

```
--bind-address 0.0.0.0 \
--pg-agent agents/9x9_from_nothing/round_007.hdf5 \
--predict-agent  agents/betago.hdf5
```

Этот код позволяет разместить рабочую демоверсию бота на сервисе AWS и сделать его доступным по адресу:

```
http://<public-dns-of-your-instance>:5000/static/play_predict_19.html
```

Вот и все! В приложении Д мы поговорим о том, как использовать описанные здесь основы работы с сервисом AWS для развертывания полномасштабного бота, способного подключаться к серверу Online Go Server (OGS) по протоколу Go Text Protocol (GTP).

Приложение Д

Отправка бота на онлайн-сервер для игры в го

В этом приложении мы поговорим о развертывании бота на популярном онлайн-сервере для игры в го. Для этого мы будем использовать фреймворк, описанный в первых восьми главах этой книги, для размещения бота на облачном сервисе Amazon Web Services (AWS), который будет обмениваться данными через протокол Go Text Protocol (GTP). Обязательно изучите основы работы с этим фреймворком в первых восьми главах, а также основы работы с сервисом AWS в приложении Г.

Регистрация и активация бота на сервере OGS

Online Go Server (OGS) – это популярная платформа, на которой вы можете играть в го против других игроков и ботов. В приложении В было представлено несколько го-серверов, однако для демонстрации процесса развертывания ботов в этом приложении мы выбрали именно OGS. Сервер OGS представляет собой современную веб-платформу, доступную по адресу: **online-go.com**. Для регистрации учетной записи OGS перейдите на страницу **online-go.com/register**. Если вы хотите развернуть бота на сервере OGS, вам нужно создать *две* учетные записи.

1. Зарегистрируйте учетную запись для себя как для игрока. Выберите доступное имя пользователя, придумайте пароль и при желании укажите адрес электронной почты. Также вы можете зарегистрироваться в системе, используя аккаунт Google, Facebook или Twitter. Эту учетную запись мы обозначим словом `<human>` (человек).
2. Вернитесь на страницу регистрации и создайте еще одну учетную запись для своего бота, выбрав для него соответствующее имя. Эту учетную запись мы обозначим словом `<bot>` (бот).

На данный момент у нас есть две обычные учетные записи. Нам нужно превратить вторую в *учетную запись бота*, которая принадлежит и управляется *учетной записью пользователя*. Для этого сначала нужно войти в аккаунт `<human>` на сервере OGS и связаться с модератором для активации аккаунта `<bot>`. В верхнем левом

углу рядом с логотипом OGS можно открыть меню и осуществить поиск пользователей по имени. При работе над этой книгой с регистрацией нам помогли модераторы OGS `crocrobot` и `anoek`. Если вы введете в строку поиска любое из этих имен, а затем щелкните по нему в результатах поиска, появится окно, изображенное на рис. Д.1.

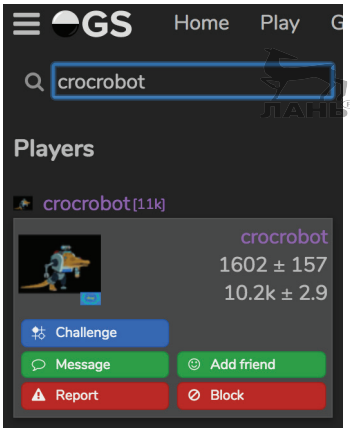


Рис. Д.1 ❖ Отправка модератору OGS сообщения для активации учетной записи бота

В открывшемся окне щелкните по кнопке **Message** (Сообщение), чтобы связаться с модератором. После этого в правом нижнем углу должно открыться окно для ввода сообщения. Вам нужно сообщить модератору о том, что вы хотите активировать аккаунт бота `<bot>`, принадлежащего аккаунту `<human>`, в котором вы в настоящее время находитесь. Как правило, модераторы OGS реагируют на сообщения в течение 24 часов, однако будьте готовы при необходимости немного подождать. Список модераторов можно найти в разделе **Chat** (Чат) в верхнем меню OGS. Их имена отмечены значком в виде молотка. Если модератор отсутствует или занят, вы можете обратиться за помощью к кому-нибудь другому.

Если по какой-то причине вам не удалось связаться с модератором напрямую, то можете оставить сообщение на форуме OGS (**forums.online-go.com**) в разделе OGS Development. Помните о том, что модераторы являются добровольцами, которые оказывают помощь в свободное время, так что наберитесь терпения!

После получения ответа от модератора войдите в свою учетную запись `<bot>`. Щелкните по значку меню в левом верхнем углу страницы OGS и выберите пункт **Profile** (Профиль) для отображения профиля вашего бота. Если все прошло удачно, ваша учетная запись `<bot>` должна быть обозначена как **Artificial Intelligence** (Искусственный интеллект), а в строке **Administrator** (Администратор) должна быть указана ваша учетная запись `<human>`. Короче говоря, профиль вашего бота должен выглядеть примерно так, как профиль бота `BetagoBot` на рис. Д.2, который администрируется учетной записью Макса `DoubleGotePanda`.

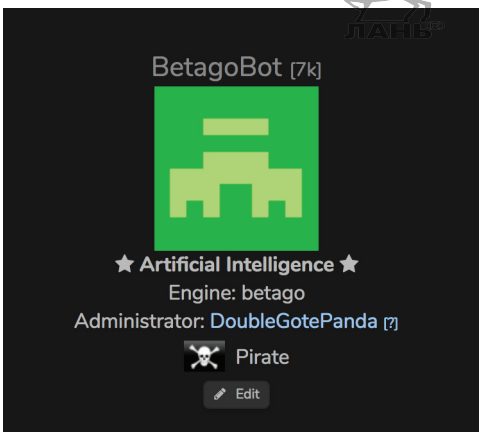


Рис. Д.2 ❖ Проверка профиля бота на предмет его активации

Теперь выйдите из своей учетной записи <bot> и вернитесь в учетную запись <human>. Это необходимо для того, чтобы сгенерировать API-ключ для бота, что можно сделать только через аккаунт администратора. После входа в учетную запись <human> перейдите на страницу профиля <bot> (для этого выполните поиск по имени <bot> и щелкните по соответствующей строке в результатах). Прокрутив вниз, вы увидите окно **Bot Controls** (Настройки бота) с кнопкой **Generate API Key** (Сгенерировать API-ключ). Щелкните по этой кнопке, чтобы сгенерировать API-ключ, а затем щелкните по кнопке **Save** (Сохранить), чтобы сохранить его. В оставшейся части этого приложения мы будем предполагать, что ваш API-ключ имеет имя <api-key>.

После настройки параметров OGS мы будем использовать имя бота и API-ключ для подключения GTP-бота к серверу OGS.

Локальное тестирование OGS-бота

В главе 8 мы создали бота, способного принимать и отправлять GTP-команды. Теперь у нас также есть аккаунт бота на сервере OGS. Недостающим звеном для их соединения является инструмент `gtp2ogs`, который принимает имя бота и API-ключ, а затем устанавливает соединение между компьютером, на котором установлен бот, и сервером OGS. Инструмент `gtp2ogs` представляет собой библиотеку с открытым исходным ходом, созданную на платформе Node.js и доступную в официальном GitHub-репозитории OGS по адресу: **github.com/online-go/gtp2ogs**. Вам не нужно специально загружать или устанавливать этот инструмент, поскольку мы уже включили его копию в наш репозиторий GitHub. В своей локальной копии **mng.bz/gYPe** вы найдете файл с именем *gtp2ogs.js* и файл JSON с именем *package.json*. Первый файл представляет собой сам инструмент, а второй будет использоваться для установки зависимостей.

При развертывании бота на сервере OGS мы хотим сделать его доступным для всех на протяжении длительного времени. Эта задача относится к задачам развертывания *долго работающих процессов*. По этой причине имеет смысл запустить

бота на (удаленном) сервере. О том, как это сделать, мы поговорим в следующем разделе, а сейчас можно быстро проверить работоспособность бота на локальном компьютере. Для этого убедитесь в том, что на нем установлена платформа Node.js и соответствующий менеджер пакетов (npm). Большинство систем позволяет получить оба инструмента с помощью выбранного вами менеджера пакетов (например, в ОС Mac можно выполнить команду `brew install node npm`, а в ОС Ubuntu – `sudo aptget install npm nodejslegacy`), однако эти инструменты также можно загрузить и установить с сайта **nodejs.org/en/download**.

Теперь нужно указать *системный путь* для сценария Python *run_gtp.py*, который находится на верхнем уровне нашего GitHub-репозитория. В средах Unix это можно сделать с помощью командной строки, выполнив следующую команду:

```
export PATH=/path/to/deep_learning_and_the_game_of_go/code:$PATH
```

Это позволит вызывать сценарий *run_gtp.py* из любого места с помощью командной строки. В частности, он будет доступен для инструмента gtp2ogs, который будет запускать нового бота с помощью *run_gtp.py* всякий раз, когда на сервере OGS будет запрошена новая игра. Теперь осталось лишь установить необходимые пакеты Node.js и запустить приложение. Мы используем пакет Node.js *forever*, чтобы приложение продолжало работать и перезапускалось в случае какого-либо сбоя.

```
cd deep_learning_and_the_game_of_go/code
npm install

forever start gtp2ogs.js \
  --username <bot> \
  --apikey <api-key> \
  --hidden \
  --persist \
  --boardsize 19 \
  --debug -- run_gtp.py
```

Давайте разберем фрагменты приведенного выше кода:

- `username` и `apikey` указывают, как подключаться к серверу;
- `hidden` предотвращает попадание бота в списки публичных ботов, что дает возможность протестировать его до того, как с ним начнут играть другие люди;
- `persist` поддерживает работу бота в перерыве между совершением ходов (в противном случае инструменту `gtp2ogs` придется перезапускать бота каждый раз, когда ему нужно будет сделать ход);
- `boardsize 19` разрешает боту участвовать только в играх на доске 19×19. Если при обучении бота вы использовали доску 9×9 (или доску другого размера), внесите в код соответствующие корректировки;
- `debug` выводит на экран дополнительные записи, сообщающие о том, что делает наш бот.

После запуска бота перейдите на сайт OGS, войдите в свою учетную запись `<human>` и щелкните по меню слева. Введите имя своего бота в поле поиска, щелкните по нему, а затем по кнопке **Challenge** (Играть), чтобы начать игру против своего бота.

Если вам удалось выбрать своего бота, скорее всего, все было сделано правильно, и теперь вы можете сыграть первую партию с собственным творением. После

успешного тестирования подключения остановите приложение Node.js, введя команду `forever stopall`.

Развертывание OGS-бота на сервисе AWS

Далее мы поговорим о бесплатном развертывании бота на сервисе AWS, чтобы вы и многие другие игроки по всему миру могли играть против него в любое время (без запуска приложения Node.js на локальном компьютере).

Мы предполагаем, что вы уже изучили приложение Г и сконфигурировали SSH-подключение для получения доступа к своему инстансу AWS с помощью `ssh aws`. Вам не обязательно выбирать очень мощный инстанс, поскольку генерирование предсказаний с использованием уже обученной модели не требует больших вычислительных ресурсов. На самом деле вы можете обойтись одним из бесплатных инстансов AWS вроде t2.micro. Если вы будете следовать приложению Г буквально и выберете образ Deep Learning AMI для ОС Ubuntu, работающий на инстансе t2.small, то поддержание работы бота на сервере OGS обойдется вам всего в несколько долларов в месяц.

Наш GitHub-репозиторий содержит сценарий *run_gtp_aws.py*, представленный в листинге Д.1. Первая строка, начинающаяся с `#!`, сообщает процессу Node.js, какую установку Python следует использовать для запуска бота. Базовая установка Python на инстансе AWS должна выглядеть примерно так: /usr/bin/python, что можно проверить, введя в терминал команду `which python`. Убедитесь в том, что в этой первой строке указана версия Python, используемая при установке dlgo.

Листинг Д.1 ❖ Сценарий *run_gtp_aws.py* для запуска бота на сервисе AWS и подключения к серверу OGS

```
#!/usr/bin/python
from dlgo.gtp import GTPFrontend
from dlgo.agent.predict import load_prediction_agent
from dlgo.agent import termination
import h5py

model_file = h5py.File("agents/betago.hdf5", "r")
agent = load_prediction_agent(model_file)
strategy = termination.get("opponent_passes")
termination_agent = termination.TerminationAgent(agent, strategy)

frontend = GTPFrontend(termination_agent)
frontend.run()
```

Убедитесь в том, что это совпадает с результатом выполнения команды `which python` в вашем инстансе.

Этот сценарий загружает агента из файла, инициализирует стратегию прекращения игры и запускает экземпляр `GTPFrontend`, как описано в главе 8. Агент и стратегия прекращения игры приведены в качестве иллюстрации. Вы можете изменить их в соответствии со своими потребностями и использовать собственные модели и стратегии. Однако в целях ознакомления с процессом развертывания бота вы можете оставить сценарий без изменений.

Теперь вам нужно удостовериться в том, что на вашем инстансе AWS установлено все необходимое для запуска бота. Давайте начнем с нуля. Клонируйте GitHub-репозиторий на локальный компьютер, скопируйте его в инстанс AWS, войдите в систему и установите пакет dlgo:

```
git clone https://github.com/maxpumperla/deep_learning_and_the_game_of_go
cd deep_learning_and_the_game_of_go
scp -r ./code aws:~/code
ssh aws
cd ~/code
python setup.py develop
```

По сути, этот процесс аналогичен тому, который использовался в приложении Г для запуска файла *endtoend.py*. Чтобы запустить *forever* и *gtp2ogs*, вам также необходимо убедиться в доступности Node.js и npm. После установки этих программ на инстанс AWS с помощью команды `apt` вы можете установить gtp2ogs так же, как вы это делали на локальном компьютере:

```
sudo apt install npm
sudo apt install nodejs-legacy
npm install
sudo npm install forever -g
```

Последним шагом является запуск GTP-бота с помощью gtp2ogs. Для этого необходимо указать системный путь для текущей рабочей директории и использовать файл *run_gtp_aws.py* в качестве утилиты для запуска бота:

```
PATH=/home/ubuntu/code:$PATH forever start gtp2ogs.js \
  --username <bot> \
  --apikey <api-key> \
  --persist \
  --boardsize 19 \
  --debug -- run_gtp_aws.py > log 2>&1 &
```

Обратите внимание на то, что стандартные потоки вывода и ошибок перенаправляются в файл журнала с именем *log*, а программа запускается в качестве фонового процесса с помощью оператора `&`. Благодаря этому командная строка инстанса не загромождается журналами сервера и не мешает вам работать на этом компьютере. Как и в случае локального тестирования OGS-бота, теперь вы можете подключиться к серверу OGS и сыграть против своего бота. При возникновении каких-либо проблем проверьте последние журналы бота в `tail log`.

Вот и все. Несмотря на то что на настройку этого конвейера уходит некоторое время (в частности, на создание инстанса AWS и регистрацию двух учетных записей OGS), после решения основных задач процесс развертывания не вызывает особых сложностей. Для того чтобы развернуть бот, разработанный на локальном компьютере, сделайте следующее:

```
scp -r ./code aws:~/code
ssh aws
cd ~/code
PATH=/home/ubuntu/code:$PATH node gtp2ogs.js \
  --username <bot> \
  --apikey <api-key> \
  --persist \
  --boardsize 19 \
  --debug -- run_gtp_aws.py > log 2>&1 &
```

Теперь, когда к боту не применяется параметр `hidden`, против него может сыграть любой пользователь сервера. Чтобы найти своего бота, войдите в учет-

ную запись <human> и щелкните по кнопке **Play** (Играть) в главном меню. В окне **Quick Match Finder** (Быстрый поиск противника) щелкните по кнопке **Computer** (Компьютер), чтобы выбрать бота для игры. Имя вашего бота, <bot>, должно отобразиться в раскрывающемся списке как AI Player. На рис. Д.3 показан бот BetagoBot, разработанный Максом и Кевином. В настоящий момент на сервере OGS размещено лишь несколько ботов. Может быть, вы добавите своего?

На этом приложение завершено. Теперь вы можете развернуть полноценный конвейер машинного обучения для размещения бота на онлайн-платформе для игры в го.

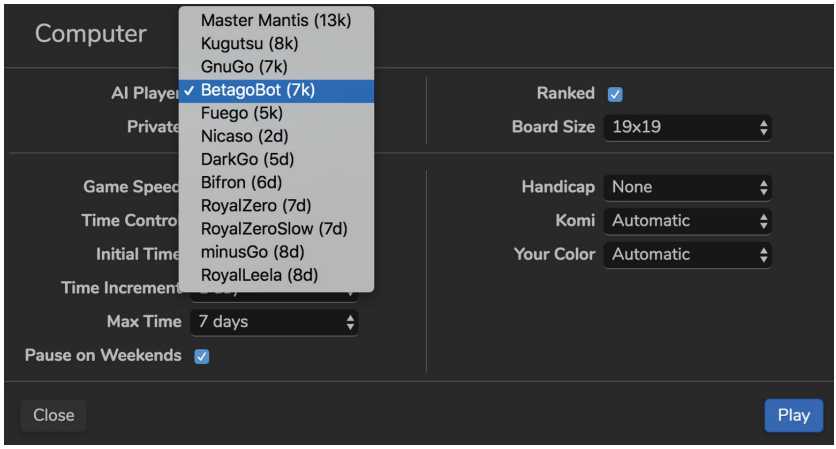


Рис. Д.3 ❖ Теперь имя вашего бота должно отображаться на сервере OGS в списке противников-компьютеров

Предметный указатель

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
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: **www.a-planeta.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Макс Памперла и Кевин Фергюсон

Глубокое обучение и игра в го

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Райтман М. А.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать офсетная.
Усл. печ. л. 30,23. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**