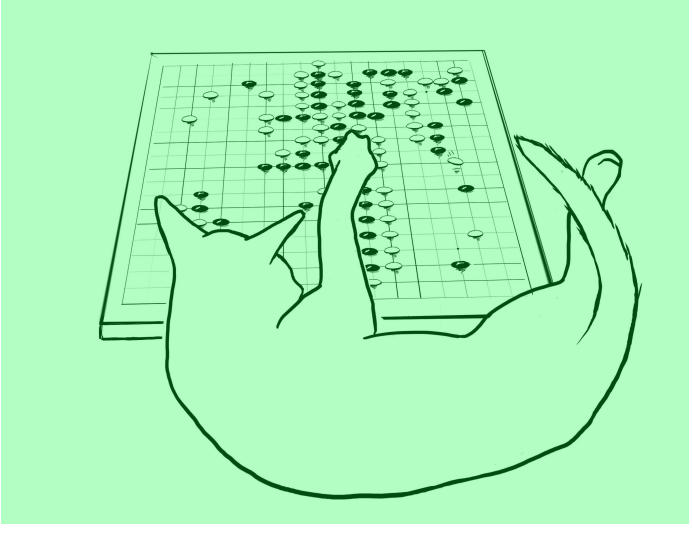

Глава 9
Глубокое обучение с подкреплением,

или Удивительная история о том, как корейский чемпион сумел победить глубокую сеть в одной партии из пяти

TL;DR

В этой главе мы поговорим о задаче обучения, которое, наверное, больше всего похоже на человеческое, — обучении с подкреплением. Мы:

- познакомимся с мотивацией и общей целью обучения с подкреплением;
- рассмотрим марковские процессы принятия решений и TD-обучение;
- поймем, где здесь нейронные сети, и разберем архитектуру DQN;
- подробно поговорим об одной из самых ярких недавних демонстраций искусственного интеллекта — победе AlphaGo над Ли Седолем;
- изучим методы градиентного спуска по стратегиям и их приложения в робототике и других областях.

9.1. Обучение с подкреплением

> Винни-Пух был всегда не прочь немного подкрепиться, в особенности часов в одиннадцать утра, потому что в это время завтрак уже давно окончился, а обед еще и не думал начинаться.
>
> *А. А. Милн. Винни-Пух и все-все-все*
> *(перевод Б. Заходера)*

Задачи машинного обучения, о которых мы говорили до сих пор, можно было разделить на две основные категории:

1) в задачах *обучения с учителем (supervised learning)* есть набор «правильных ответов», и нужно его продолжить на все пространство или, точнее говоря, на те примеры, которые могут встретиться в жизни, но которые еще не попадались в тренировочной выборке; к этому классу относятся задачи регрессии и классификации, и этим мы занимаемся в этой книге все время, будь то классификация рукописных цифр по набору данных MNIST или машинный перевод на основе параллельных текстов на разных языках;
2) в задачах *обучения без учителя (unsupervised learning)* есть просто набор примеров без дополнительной информации, и задача состоит в том, чтобы понять его структуру, выделить признаки, хорошо ее описывающие; мы подробно рассматривали некоторые задачи обучения без учителя в части III, когда говорили об автокодировщиках и предобучении нейронных сетей, да и вообще, как мы уже много раз видели, одно из главных достоинств глубокого обучения — это именно способность очень умело выделять сложные признаки, раскрывать непростую внутреннюю структуру окружающих нас объектов.

Это весьма классическое разделение, вы встретите его в любом учебнике по машинному обучению. Но разве так учимся мы с вами? Разве так работает обучение в реальной жизни?

Представьте, к примеру, как маленький ребенок учится ходить. Как могло бы это выглядеть в парадигмах обучения с учителем и без него:

1) *обучение с учителем* — родители, которые уже понимают, как ходят люди, выдают ребенку набор данных о том, какие мышцы и в какой последовательности нужно напрячь для того, чтобы сделать шаг; набор должен быть достаточно полным и покрывать все или почти все разные ситуации, в которых ребенок может захотеть сделать шаг, а также, разумеется, должен содержать и отрицательные примеры: как именно он упадет, если будет напрягать мышцы неправильно; ну или, чтобы не доводить до такого уж абсурда, родители просто показывают ребенку несколько десятков тысяч фотографий, на которых изображены разные фазы разных шагов, а дальше его волшебная глубокая сеть выделяет из них признаки и передает ребенку нужную информацию;

2) *обучение без учителя* — ребенок просто наблюдает, как ходят его папа и мама; и потом, постепенно, из сотен тысяч записанных в мозг картинок складываются нужные признаки: если вот так слегка напряглась левая икроножная мышца, а вот этак — правая, в том же кластере будут видео папы и мамы, которые идут вперед ровно и уверенно; ну а если напрячь иначе, то вот, давно уже наготове тысячи картинок того, как папа и мама неуклюже падают, не сумев сделать шаг.

Правда же, совсем как в жизни? Как сейчас помню, мама приходила к моей кроватке и начинала показывать, как надо ходить и как не надо...

На самом деле мы далеко не всегда знаем набор правильных ответов. Часто мы просто делаем то или иное действие и получаем результат. Когда ребенок учится ходить, он сначала делает что-то отдаленно похожее на правду, что подсказывает ему инстинкт[1], и получает результат, поначалу скорее всего отрицательный. Тогда он немного меняет свое поведение, возможно, довольно случайным образом, и смотрит, не увеличилась ли, так сказать, целевая функция; а когда что-то начинает получаться, ребенок запоминает, как это произошло, и затем повторяет то же самое, пытаясь развить успех дальше.

Отсюда и основная идея *обучения с подкреплением (reinforcement learning)*. В этой постановке задачи агент взаимодействует с окружающей средой, предпринимая действия; окружающая среда его поощряет за эти действия, а агент продолжает их предпринимать, пытаясь максимизировать свою «награду» за это; его награда тоже приходит из окружающей среды.

Историю обучения с подкреплением[2] можно смело начинать от работ Павлова[3] об условных и безусловных рефлексах, хотя психологические подходы были известны и ранее, например в работах Александра Бэна середины XIX века [24]. Его основная идея состояла в том, что мы обучаемся методом проб и ошибок: когда какое-нибудь спонтанное (то есть по сути случайное) движение совпадает с состоянием удовольствия, «удерживающая сила духа» устанавливает между ними ассоциацию. Ту же линию продолжили и теории Ллойда Моргана в психологии и Эдварда Ли Торндайка в его трудах об интеллекте животных [530]: полезное действие, вызывающее удовольствие, закрепляется и усиливает связь между ситуацией и реакцией, а вредное, вызывающее неудовольствие, ослабляет связь и исчезает.

---

[1] Впрочем, соотношение между инстинктами и собственно прижизненным обучением — это очень тонкий вопрос; мы не будем на нем подробно останавливаться, но последнего слова здесь наука еще не сказала.

[2] Эту историю мы излагаем по эссе Валентина Малых [593].

[3] *Иван Петрович Павлов* (1849–1936) — русский ученый, первый русский нобелевский лауреат, фактический создатель науки о высшей нервной деятельности. Работа Павлова чем-то похожа на то, как в обучении глубоких сетей сходятся теоретические и практические новшества: его знаменитые опыты с желудочным соком, имеющие огромное теоретическое значение и ставшие основой всей современной физиологии, были бы невозможны без виртуозной техники вивисекции, как это тогда называлось. А из других трудов Павлова возьмем на себя смелость порекомендовать прочесть две лекции с заманчивым названием «Об уме вообще, о русском уме в частности»: сказано в 1918 году, но до сих пор чрезвычайно актуально.

А после того как в психологию пришли бихевиористы, и особенно Б. Ф. Скиннер[1], эта идея стала абсолютным и не вызывающим сомнения мейнстримом.

В целом это и есть идея обучения с подкреплением, и в машинном обучении она тоже появилась очень давно. Еще Тьюринг в своих эссе об искусственном интеллекте описывал архитектуру системы «боли и наслаждения», которая должна управлять тем, что именно сохраняется в памяти искусственного интеллекта [542]. Ранние работы информатиков на эту тему похожи на работы Павлова и Скиннера: например, в 1952 году Клод Шеннон продемонстрировал лабиринт, по которому бегал мышонок по имени Тесей, исследуя его тем самым методом проб и ошибок и запоминая свой путь [490].

В своей диссертации Марвин Минский [362] представил вычислительные модели обучения с подкреплением, а также описал аналоговую вычислительную машину, построенную на элементах, которые он назвал SNARC — стохастический вычислитель, обучаемый через подкрепление и по сути аналогичный нейрону. Эти элементы должны были соответствовать изменяемым семантическим связям в человеческом мозге; обратите внимание, что это совсем не то же самое, что перцептрон Розенблатта, который мы обсуждали в разделе 3.2 и который явным образом обучается с учителем.

Если говорить чуть более формально, на каждом шаге агент может находиться в некотором состоянии $s \in S$, где S — множество всех состояний, и выбирает некоторое действие $a \in A$ из имеющегося набора действий A.

После этого окружающая среда сообщает агенту, какую награду r (от слова *reward*) он за это получил и в каком состоянии s' оказался в результате своих действий. Задача агента — заработать как можно большую награду:

- либо за отведенное время h (от слова horizon, «горизонт»); такая постановка задачи называется *моделью с конечным горизонтом*, и целевую функцию (доход, revenue) можно представить так:

$$R = \mathbb{E}\left[r_0 + r_1 + r_2 + \ldots + r_h\right] = \mathbb{E}\left[\sum_{t=0}^{h} r_t\right],$$

  где r_t — награда агента на шаге t;

[1] *Беррес Фредерик Скиннер* (Burrhus Frederic Skinner, 1904–1990) — американский психолог, бихевиорист и социальный философ. Скиннер довел идеи бихевиоризма до их наивысшего развития; он считал, что все многообразие поведения людей и животных так или иначе сводится к принципам позитивного и негативного подкрепления. Для развития своих идей он изобрел пресловутый Skinner box, в котором мышка может нажать на специальный рычажок и получить за это действие ту или иную награду. Во время Второй мировой войны Скиннер предложил конструкцию ракеты, управляемой специально натренированным голубем (проект был многообещающий, но, к сожалению, так и не реализовался до конца), а затем обнаружил у голубей даже суеверия: если давать голубю еду в случайные моменты времени, голубь очень постарается заметить случайные совпадения того, что он делал в эти моменты, с поступлением еды, и будет пытаться повторять эти случайные действия.

- либо за все бесконечное предстоящее время; в таком случае, конечно, просто суммировать не получится, но легко понять, что в таких задачах почти всегда получить награду раньше выгоднее, чем позже[1], поэтому обычно в целевую функцию модели с *бесконечным горизонтом* вводят некоторую константу (discount factor), на которую вознаграждение уменьшается с каждым последующим шагом:

$$R = \mathbb{E}\left[r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots + \gamma^k r_k + \ldots\right] = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right].$$

Есть и третий вариант, *модель среднего вознаграждения (average-reward model)*, когда оптимизируется $\lim_{h\to\infty} \mathbb{E}\left[\frac{1}{h}\sum_{t=0}^{h} r_t\right]$, но мы ее использовать практически не будем. А конечный горизонт легко свести к бесконечному: достаточно поставить $\gamma = 1$ и считать, что за горизонтом, после времени h, все награды r_t, $t > h$, равны нулю независимо от действий агента.

Самая простая постановка задачи обучения с подкреплением — это так называемая задача о *многоруких бандитах (multiarmed bandits)*. Формально здесь все точно так же, но $|S| = 1$, то есть состояние агента не меняется. У него просто есть некий фиксированный набор действий A и возможность выбирать из этого набора действий. Такая модель называется задачей о многоруких бандитах, потому что ее легко представить себе так: агент находится в комнате с несколькими игровыми автоматами, у каждого автомата свое ожидание выигрыша, а агенту нужно выиграть как можно больше денег, бросая монетки то в один автомат, то в другой.

Получается, что агент сам платит за свое обучение, и ему нужно суметь вовремя понять, что обучение (исследование, exploration) можно заканчивать и переходить к использованию полученных знаний (exploitation). *Exploration vs. exploitation* — это главная проблема, основная дилемма задачи о многоруких бандитах.

Мы не будем подробно рассказывать о многоруких бандитах, потому что глубокое обучение с подкреплением обычно начинается все-таки там, где состояний несколько, но пару важных замечаний сделать нужно. Начнем с простейшего *жадного алгоритма*, который всегда выбирает стратегию, максимизирующую прибыль. Прибыль можно оценить как среднее вознаграждение, полученное от того или иного действия:

$$Q_t(a) = \frac{r_1 + r_2 + \ldots + r_{k_a}}{k_a}.$$

Здесь и далее мы будем предполагать, что мы сначала дергаем по разу за каждую ручку, чтобы нигде не делить на ноль, а потом уже начинаем запускать алгоритмы.

[1] Представьте себе, например, награду в виде денег: сто рублей сегодня точно лучше, чем сто рублей через год, хотя бы потому, что их можно положить в банк и через год получить больше ста рублей, не говоря уж о более выгодных вложениях.

Жадный алгоритм кажется достаточно простым и логичным. Что же с ним не так? Дело в том, что оптимум легко проглядеть, если на начальной выборке нам немножко не повезет, что более чем возможно. Представьте себе, например, *бинарных* бандитов, у которых выплаты всегда равны нулю или единице. Тогда жадный алгоритм буквально *никогда* не вернется к тем ручкам, которые на первом обходе выдали ноль, ведь их среднее будет нулевым, а у тех ручек, которые хоть раз выдали единицу, нуля уже никогда не получится.

Поэтому полезная эвристика в задаче о многоруких бандитах, да и во всем обучении с подкреплением — *оптимизм при неопределенности*. Это значит, что выбирать нужно жадно, но при этом прибыль оценивать оптимистично, и всегда требовать серьезные свидетельства, чтобы перестать проверять ту или иную стратегию.

Одним из ключевых теоретических инструментов в обучении с подкреплением является *ϵ-жадная стратегия* (ϵ-greedy): выбрать действие с наилучшей ожидаемой прибылью с вероятностью $1 - \epsilon$, а с вероятностью ϵ выбрать случайное действие. Такая стратегия приводит к тому, что алгоритм не отличает хорошую альтернативу от бесполезной, выделяя только лучшую, но все равно процесс исследования получается разумным, и про него обычно можно доказать разные полезные теоретические свойства, ведь ϵ-жадная стратегия означает, что мы всегда можем дернуть за каждую ручку с положительной константной вероятностью. На практике обычно начинают с больших ϵ, а затем уменьшают; выбор стратегии этого уменьшения — важный параметр алгоритма.

Другой естественный способ применить оптимистично–жадный метод — это известные из статистики *доверительные интервалы*. Давайте для каждого действия хранить статистику числа таких действий n и числа успешных действий w (или среднего вознаграждения), а потом для выбора ручки, за которую хочется дернуть, использовать *верхнюю границу* доверительного интервала вероятности успеха (или ожидания вознаграждения). Тем самым мы достигнем как раз нужного эффекта: сначала все доверительные интервалы будут очень широкими, и их верхние границы будут очень высоко, а потом, по мере накопления опыта, интервалы начнут сужаться, причем менее исследованные альтернативы будут получать преимущество в выборе. Такая стратегия сойдется к оптимальной ручке, когда доверительные интервалы всех остальных ручек будут полностью лежать ниже ее среднего. Например, для бинарных бандитов, то есть фактически для подбрасывания монетки, с вероятностью 0,95 среднее лежит в интервале $\left(\bar{x} - 1{,}96\frac{s}{\sqrt{n}}, \bar{x} + 1{,}96\frac{s}{\sqrt{n}}\right)$, где 1,96 берется из распределения Стьюдента, n — число испытаний, $s = \sqrt{\frac{\sum(x-\bar{x})^2}{n-1}}$. Брать верхнюю границу доверительного интервала — отличный метод, если вероятностные предположения соответствуют действительности, что не всегда очевидно.

Другой, более простой вариант оптимизма при неопределенности — начать с оптимистичных значений средних: давайте выставим $Q_0(a)$ такими большими, что любое реальное вознаграждение будет «разочаровывать» нас, но не слишком

большими – нам нужно, чтобы достаточно быстро Q_0 усреднилось с реальными оценками средних. Тогда можно использовать тривиальную жадную стратегию, и она даст тот же эффект: сначала средние у всех ручек будут заведомо слишком высокими, а потом по мере накопления опыта уменьшатся и будут постепенно сходиться к истинным средним, причем чем меньше было экспериментов, тем большее влияние оказывает Q_0 на среднее. Сами значения Q_0 можно менять и тем самым управлять балансом между exploration и exploitation.

Современные алгоритмы действуют примерно по той же общей схеме: они присваивают приоритет каждой ручке i и доказывают оценки непосредственно на цену обучения (regret). Так, стратегия UCB1 [13] учитывает неопределенность, «оставшуюся» в той или иной ручке, и старается ограничить цену обучения так: если из n экспериментов n_i раз дернули за i-ю ручку и получили среднюю награду $\hat{\mu}_i$, алгоритм UCB1 присваивает ей приоритет

$$\text{Priority}_i = \hat{\mu}_i + \sqrt{\frac{2\log n}{n_i}}.$$

Дергать дальше надо за ручку с наивысшим приоритетом. В [13] доказано, что при таком подходе субоптимальные ручки будут дергать $O(\log n)$ раз, и цена обучения будет составлять $O(\log n)$. А меньше $O(\log n)$ и не получится — есть соответствующая нижняя оценка; впрочем, константы тут тоже важны, так что на UCB1 человеческая мысль не остановилась, но мы в это уже вдаваться не будем.

И последнее общее замечание, которое нам пригодится ниже. Давайте посмотрим на то, как пересчитывать оценки среднего $Q_t(a) = \frac{r_1+\ldots+r_{k_a}}{k_a}$ при поступлении новой информации. В этом, конечно, ничего сложного нет — будем хранить число попыток k и пересчитывать как

$$Q_{k+1} = \frac{1}{k+1}\sum_{i=1}^{k+1} r_i = \frac{1}{k+1}\left[r_{k+1} + \sum_{i=1}^{k} r_i\right] = \\ = \frac{1}{k+1}\left(r_{k+1} + kQ_k\right) = Q_k + \frac{1}{k+1}\left(r_{k+1} - Q_k\right).$$

У нас получилась очень важная формула, частный случай общего правила — сдвигаем оценку так, чтобы уменьшалась ошибка:

$$\text{НоваяОценка} := \text{СтараяОценка} + \text{Шаг}\left[\text{Цель} - \text{СтараяОценка}\right].$$

почему r_k это цель?

Именно так выглядит, например, общее правило градиентного спуска: производная указывает направление на цель в текущей точке, а шаг — это скорость обучения.

Заметим теперь, что шаг у среднего не постоянный, а уменьшающийся со временем: из формулы $Q_{k+1} = Q_k + \frac{1}{k+1}\left(r_{k+1} - Q_k\right)$ получается, что шаг ручки a

в момент времени k (после k экспериментов) $\alpha_k(a)$ равен $\frac{1}{ka+1}$. Изменяя последовательность шагов, можно добиться других эффектов. Например, часто бывает, что выплаты от разных ручек на самом деле нестационарны, то есть меняются со временем. В такой ситуации имеет смысл давать большие веса свежей информации, а далекому прошлому — маленькие. Как это сделать? Можно просто поставить вместо затухающих весов постоянные: у правила обновления $Q_{k+1} = Q_k + \alpha\,[r_{k+1} - Q_k]$ с постоянным $\alpha_k(a) = \alpha$ веса фактически затухают экспоненциально:

$$Q_k = Q_{k-1} + \alpha\,[r_k - Q_{k-1}] = \alpha r_k + (1-\alpha)Q_{k-1} =$$
$$= \alpha r_k + (1-\alpha)\alpha r_{k-1} + (1-\alpha)^2 Q_{k-2} = (1-\alpha)^k Q_0 + \sum_{i=1}^{k} \alpha(1-\alpha)^{k-i} r_i.$$

Такое правило обновления не сходится (разности между соседними Q_k и Q_{k-1} не обязательно стремятся к нулю с ростом k), но это и хорошо – мы хотим следовать за целью. Есть и общий результат: правило обновления сходится, если последовательность весов не сходится сама, $\sum_{k=1}^{\infty} \alpha_k(a) = \infty$, но сходится в квадрате, $\sum_{k=1}^{\infty} \alpha_k^2(a) < \infty$. И в целом, такой взгляд на вещи сразу объединяет и упорядочивает массу разных методов.

Сами по себе многорукие бандиты используются в задачах, где нужно сравнить несколько альтернатив между собой при помощи экспериментов. Например, несколько лет назад в IT-сообществе были очень популярны статьи, пропагандирующие использование многоруких бандитов для A/B тестирования: каждая альтернатива становится ручкой, и выходит, что размер выборки подбирается автоматически. Можно аналогично использовать бандитов и для оптимизации гиперпараметров, например в тех же глубоких нейронных сетях (или где угодно). Но для нас сейчас это скорее просто первый шаг, упрощенная постановка общей задачи обучения с подкреплением... так давайте же скорее избавимся от упрощений.

9.2. Марковские процессы принятия решений

> Но как я могу принять решение, – спрашивал себя этот рассудительный государь, – если мне совершенно неизвестны те доводы, которые могут привести обе стороны?
>
> *Стендаль. Чрезмерная благосклонность губительна*

Теперь, когда мы поняли основную суть обучения с подкреплением и разобрались с самой простой ситуацией, многорукими бандитами, пора обобщать дальше. В реальности, конечно, часто бывает, что агент не возвращается в точно такое же состояние для нового «хода»: например, играющая в го или шахматы программа должна,

пожалуй, учитывать, что позиция на доске после ее хода и ответа противника слегка изменится. Поэтому теперь нам нужно определить некий процесс, в котором агент последовательно переходит из состояния в состояние в зависимости от своих действий, иногда, в некоторых состояниях, получая награды; все зависимости здесь, конечно, будут не прямыми, а стохастическими, вероятностными.

Здесь возникает вторая центральная дилемма обучения с подкреплением, задача *распределения вознаграждения* (credit assignment): пусть даже мы знаем, выиграли мы партию или проиграли, но какой именно ход привел к победе или поражению? Эта проблема тоже была очевидна с первых шагов теории обучения с подкреплением, ее рассматривал еще Марвин Минский в начале 1960-х годов [363], но успешно решить ее бывает сложно до сих пор.

Итак, пора ввести основное определение данной главы. *Марковский процесс принятия решений* (Markov decision process) [233, 519] состоит из:

- множества состояний S;
- множества действий A;
- функции вознаграждения $R : S \times A \to \mathbb{R}$; это значит, что ожидаемое вознаграждение при переходе из s в s' после действия a составляет $R^a_{ss'}$;
- функции перехода между состояниями $p^a_{ss'} : S \times A \to \Pi(S)$, где $\Pi(S)$ — множество распределений вероятностей над S; это значит, что вероятность попасть из состояния s в состояние s' после действия a равна $P^a_{ss'}$.

Мы проиллюстрировали марковский процесс принятия решений на рис. 9.1: слева, на рис. 9.1, *а*, вы видите самую общую, классическую схему обучения с подкреплением. А на рис. 9.1, *б* показана более подробная схема марковского процесса принятия решений, где видно, какие его части от каких зависят.

Процесс называется *марковским*, потому что вероятности переходов между состояниями не зависят от истории предыдущих переходов; вообще, слово «марковский» в математике и информатике всегда обозначает именно это: отсутствие памяти, независимость от того, что было в прошлом. Это кажется очень сильным предположением, но на самом деле оно довольно часто выполняется. Например, в го или шахматах неважно, какими ходами мы пришли к текущей позиции, важна лишь позиция сама по себе.

А если марковское свойство существенно нарушено, то часто можно просто записать то, что нужно «помнить» из прошлого, в качестве части определения состояния, и тогда марковское свойство будет восстановлено. Например, состояние при игре в покер должно включать в себя не только текущий размер ставок, но и историю ставок в текущей раздаче, а возможно, и более долгую историю взаимодействия между игроками[1].

[1] Кстати, покер — это отдельная и очень сложная история, во многие его разновидности программы как раз пока что играют довольно плохо, и важная часть проблемы именно в том, чтобы правильно помнить и понимать историю взаимодействий. Однако стоит отметить модель DeepStack [113], которая научилась хорошо играть в heads-up no-limit Texas hold'em, то есть в безлимитный покер против одного оппонента.

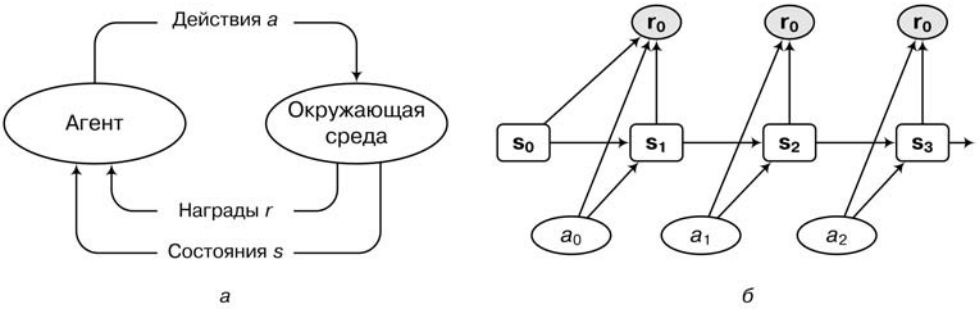

Рис. 9.1. Марковский процесс принятия решений: *а* — общая схема обучения с подкреплением, *б* — подробная схема марковского процесса принятия решений

Здесь стоит сделать небольшое отступление. Общая постановка задачи обучения с подкреплением похожа на постановку задачи *теории оптимального управления*, в котором известна некоторая модель объекта управления, на объект есть некоторые воздействия, и задача состоит в том, чтобы найти оптимальные воздействия, которые максимизируют нужную целевую функцию. Теория оптимального управления — область старая и заслуженная, в ней появились и уравнения Беллмана, о которых речь пойдет ниже, и принцип максимума Понтрягина[1]. И объекты в ней рассматриваются куда сложнее, чем обычно в обучении с подкреплением. Действительно, многое из того, о чем мы будем говорить в этой главе, можно рассматривать как часть теории управления. Однако акценты расставлены немного иначе: в обучении с подкреплением основная сложность не в том, чтобы найти оптимальное управление, а в том, чтобы изучить окружающую среду — если мы среду и объект управления уже знаем (в теории оптимального управления это называется *идентификацией системы*), обычно найти оптимальное действие не так уж сложно. Но области действительно взаимопроникающие, и изучить методы теории управления будет очень полезно, если вы всерьез решите заняться обучением с подкреплением.

[1] *Лев Семенович Понтрягин* (1908–1988) — советский математик, один из лучших математиков XX века. Когда Льву было 14 лет, возле него взорвался примус, и после неудачной операции Понтрягин полностью потерял зрение. Это предопределило судьбу семьи Понтрягиных: отец в результате потерял трудоспособность и вскоре умер от инсульта, а мать посвятила себя сыну и его математическому образованию. Она даже выучила немецкий язык, чтобы читать Льву статьи на языке тогдашней науки. Однако слепота не помешала ни активной жизни, ни тем более научной карьере Понтрягина. Он начал с топологии, получил ряд блестящих результатов, но позже вспоминал: «Я не мог ответить на вопрос, для чего нужно все это, все то, что я делаю? Самая пылкая фантазия не могла привести меня к мысли, что гомологическая теория размерности может понадобиться для каких-нибудь практических целей». Поэтому Понтрягин занялся прикладной математикой: его четырехстраничная статья 1937 года «Грубые системы» (с А. А. Андроновым) заложила основы теории динамических систем, позже развил теорию дифференциальных игр, а принцип максимума Понтрягина остается основополагающим принципом теории оптимального управления.

Подробный пример марковского процесса принятия решений, к которому мы будем постоянно возвращаться в этом разделе, приведен на рис. 9.2. На рисунке прямоугольники соответствуют состояниям, белые овалы — возможным действиям, а овалы со штриховкой — вознаграждениям, получаемым на этом переходе между состояниями. Сам процесс соответствует простой игре «чет-нечет», проходящей в два круга. Правила игры таковы:

- в каждом круге и мы, и противник выбираем число; если оба числа четные или оба нечетные, мы выигрываем; если четность разная (одно число четное, другое нечетное), мы проигрываем;
- в первом круге выигрыш и проигрыш равен $+1$ и -1 соответственно, а во втором круге ставки повышаются, и выигрыш и проигрыш начинают стоить $+5$ и -5 соответственно;
- после второго круга игра заканчивается (поэтому мы не стали выделять много состояний после второго круга, ограничившись двумя для удобства);
- поскольку игра конечная, процесс получается эпизодический, и $\gamma = 1$.

Но это еще не все; чтобы полностью задать марковский процесс принятия решений, нужно еще определить вероятности переходов между состояниями; они показаны на стрелках, соответствующих переходам. Здесь будет небольшое усложнение, без которого игра была бы совсем скучной. противник будет не просто подбрасывать монетку, а действовать таким образом:

- в первом круге противнику больше нравятся нечетные числа: вероятность четного числа от него равна $\frac{1}{3}$, а нечетного — $\frac{2}{3}$;
- а во втором круге он смотрит на то, к чему привела его игра в первый раз: если он в первом круге выиграл, то он считает, что сыграл правильно, и повторяет свой выбор, а если проиграл, возвращается к «стратегии по умолчанию» и во втором круге просто подбрасывает монетку.

Не поленитесь и проследите все стрелочки на рис. 9.2: пример еще пригодится.

Теперь, когда у нас есть состояния и переходы между ними, придется научиться различать *функцию вознаграждения* (reward function, непосредственное подкрепление, то, что мы обозначили за R) и то, что мы назовем *функцией значения состояния* (value function, $V(s)$); это будет общее ожидаемое подкрепление, которое можно получить, если начать с этого состояния. Суть многих методов обучения с подкреплением – в том, чтобы оценивать и оптимизировать функцию значений; по сути задача наша сводится к тому, чтобы выбирать ходы, которые приводят к состоянию с максимальным значением $V(s)$. Для марковских процессов можно формально определить:

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[R_t \mid s_t = s\right] = \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right],$$

где π — это стратегия, которой следует агент.

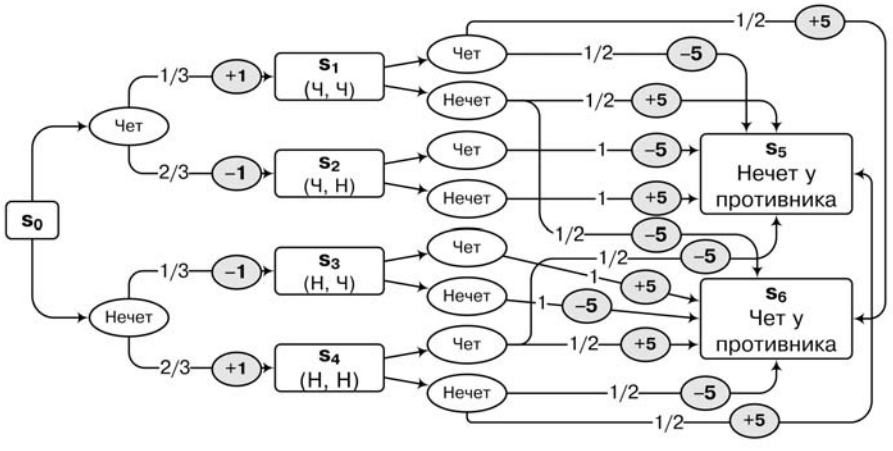

Рис. 9.2. Пример марковского процесса принятия решений: игра «чет-нечет»

Обратите внимание, что слово *стратегия* (policy) здесь понимается в достаточно строгом смысле. Поскольку, вообще говоря, агент может подбрасывать монетки, чтобы выбирать очередные действия (и многие стратегии из раздела 9.1 так и делали, например часто встречающаяся ϵ-жадная стратегия), стратегия π — это функция, которая для данного состояния s выдает распределение вероятностей на множестве действий A. Мы также будем обозначать через $\pi(a,s)$ вероятность выбрать действие $a \in A$ в состоянии s, а если захотим подчеркнуть, что стратегия задана параметрически с вектором параметров $\boldsymbol{\theta}$, будем писать $\pi(a,s;\boldsymbol{\theta})$ (это нам особенно пригодится в разделе 9.5). Если же стратегия детерминированная, это просто значит, что для каждого s все вероятности $\pi(a,s)$ равны нулю, кроме одной, которая равна единице.

Впрочем, функция значений состояния все еще удалена от непосредственных решений агента, ведь он не может просто взять и выбрать следующее состояние. Игрок в го не может сам определить позицию перед своим следующим ходом, она будет зависеть и от хода противника. Поэтому ожидаемое в будущем подкрепление часто рассматривают более детально: функция Q выражает общее подкрепление, ожидаемое, если агент начнет в состоянии s и сделает там действие a:

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[R_t \mid s_t = s, a_t = a\right] = \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a\right].$$

Функции V и Q — это как раз то, что нам нужно оценить; если бы мы их знали, можно было бы просто выбирать то действие a, которое максимизирует $Q(s,a)$.

Давайте попробуем подсчитать функцию значений состояния $V^\pi(s)$, последовательно разворачивая ее определение. В цепочке равенств ниже мы сначала выписываем определение ожидаемого суммарного вознаграждения с бесконечным горизонтом, затем отделяем от него первый шаг и замечаем, что после него остается точно такое же выражение, просто умноженное на γ:

$$
\begin{aligned}
V^\pi(s) = \mathbb{E}_\pi\left[R_t \mid s_t = s\right] &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k r_{t+k+1} \mid s_t = s\right] = \\
&= \mathbb{E}_\pi\left[r_{t+1} + \gamma\sum_{k=0}^{\infty}\gamma^k r_{t+k+2} \mid s_t = s\right] = \\
&= \sum_a \pi(s,a)\sum_{s'} P^a_{ss'}\left(R^a_{ss'} + \gamma\mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k r_{t+k+2} \mid s_{t+1} = s'\right]\right) = \\
&= \sum_a \pi(s,a)\sum_{s'} P^a_{ss'}\left(R^a_{ss'} + \gamma V^\pi(s')\right).
\end{aligned}
$$

У нас получилось, что для известной стратегии π значения V^π удовлетворяют так называемым *уравнениям Беллмана*[1].

Уравнения Беллмана — это по сути математическое выражение принципа динамического программирования. Такие уравнения и их аналоги появляются во множестве разных приложений. А в нашем случае получается, что, теоретически говоря, для того чтобы найти значения $V^\pi(s)$, можно просто взять и решить систему линейных уравнений, неизвестными в которых являются $V^\pi(s)$ для разных состояний $s \in S$.

Правда, для этого нужно знать все параметры марковского процесса, то есть функции R и P, а с этим в реальных ситуациях плохо. Все, что у нас обычно есть на входе, — это окружающая среда, выдающая награды как черный ящик.

Так что в реальных задачах функции R и P тоже приходится обучать по ходу дела. На самом деле методы обучения с подкреплением делятся на те, которые обучают функции R и P в явном виде, и те, которые обходятся без этого и сразу обучают V и/или Q.

[1] *Ричард Беллман* (Richard Ernest Bellman, 1920–1984) — американский математик и инженер. Его жизненный путь был очень интересным, но достаточно обычным для ученого того времени: родился в Бруклине (кстати, у Беллмана российские и польские корни), во время Второй мировой войны работал над Манхэттенским проектом как теоретический физик, а с 1949 года поступил на работу в RAND Corporation, знаменитый аналитический центр (think tank), изначально организованный Douglas Aircraft Company для нужд армии, но быстро превратившийся в некоммерческую научную организацию, занимающуюся самыми разными научными и инженерными вопросами. Именно там Беллман разработал основы теории марковских процессов принятия решений и динамического программирования в целом — кстати, он сам придумал это название, мотивировав его тем, что «прилагательное dynamic было невозможно использовать в негативном смысле... это было такое название, что даже конгрессмен не возразил бы».

Но для простых примеров воспользоваться уравнениями Беллмана вполне возможно. Давайте попробуем подсчитать функцию значений состояния для какой-нибудь стратегии в примере на рис. 9.2. Например, пусть стратегия π — это просто равновероятный выбор на каждом шаге, подбрасывание честной монетки. Тогда:

$$V^\pi(s_0) = \frac{1}{2}\left(\frac{1}{3}\left(1 + V^\pi(s_1)\right) + \frac{2}{3}\left(-1 + V^\pi(s_2)\right)\right) + \\ + \frac{1}{2}\left(\frac{1}{3}\left(-1 + V^\pi(s_3)\right) + \frac{2}{3}\left(1 + V^\pi(s_4)\right)\right).$$

Считаем дальше. Теперь рассмотрим, к примеру, состояние s_1:

$$V^\pi(s_1) = \frac{1}{2}\left(\frac{1}{2}\left(-5 + V^\pi(s_5)\right) + \frac{1}{2}\left(5 + V^\pi(s_6)\right)\right) + \\ + \frac{1}{2}\left(\frac{1}{2}\left(5 + V^\pi(s_5)\right) + \frac{1}{2}\left(-5 + V^\pi(s_6)\right)\right).$$

Поскольку после второго круга игра заканчивается, $V^\pi(s_5) = V^\pi(s_6) = 0$, а значит, $V^\pi(s_1) = 0$ тоже. Легко видеть, что для остальных состояний это тоже верно: $V^\pi(s_2) = V^\pi(s_3) = V^\pi(s_4) = 0$, а значит, и $V^\pi(s_0) = 0$.

Итак, подбрасывая монетку на каждом круге, мы ничего не выиграем и ничего не проиграем. Оптимальная ли это стратегия или можно все-таки что-то выиграть? Давайте попробуем ответить на этот вопрос.

Но давайте сначала для простоты предположим, что мы уже точно знаем нашу модель. Задача — найти оптимальную стратегию поведения для агента в этой модели. В такой постановке мы уже умеем искать $V^\pi(s)$, решая уравнения Беллмана, но разных стратегий еще больше, чем состояний, и перебрать их мы обычно, простите за каламбур, не в состоянии.

К счастью, это и не обязательно: нам нужно только научиться подсчитывать *оптимальное значение состояния*, то есть искать ожидаемую суммарную прибыль, которую получит агент, если начнет с этого состояния и будет следовать оптимальной стратегии:

$$V^*(s) = \max_\pi E\left[\sum_{t=0}^{\infty} \gamma^t r_t\right].$$

Эту функцию можно по тем же причинам определить как решение уравнений:

$$V^*(s) = \max_a \sum_{s' \in S} P^a_{ss'}\left(R^a_{ss'} + \gamma V^*(s')\right)$$

(они тоже называются уравнениями Беллмана), а затем выбрать оптимальную стратегию исходя из подсчитанных значений V^*:

$$\pi^*(s) = \arg\max_a \sum_{s' \in S} P^a_{ss'} \left(R^a_{ss'} + \gamma V^*(s') \right).$$

Как решать уравнения? Они уже не линейные, и решить их точно и эффективно не получится, но наше дело все равно отнюдь не безнадежно. Как известно, если сложное уравнение представлено в виде $x = f(x)$, то его можно решать итеративно методом Ньютона: начать с какого-нибудь x_0 и последовательно пересчитывать $x_{k+1} = f(x_k)$, пока процесс не сойдется, то есть пока изменения $|x_{k+1} - x_k|$ не станут совсем маленькими. Здесь эта идея великолепно работает, ведь, как легко заметить, уравнения уже представлены в нужном виде!

Это можно делать и для исходных линейных уравнений, получится быстрее, чем решать систему «по-честному». Естественно, в результате получается приближенный, численный метод решения уравнений Беллмана, но в машинном обучении нам ничего другого и не требуется.

Так что, чтобы подсчитать функции значений состояний для данной стратегии π, можно просто итеративно пересчитывать их по уравнениям Беллмана:

$$V^\pi(s) := \sum_a \pi(s,a) \sum_{s' \in S} P^a_{ss'} \left(R^a_{ss'} + \gamma V^\pi(s') \right),$$

пока процесс не сойдется.

А для оптимальных значений мы будем пересчитывать уравнения с максимумами вместо математических ожиданий:

$$V^*(s) := \max_a \sum_{s' \in S} P^a_{ss'} \left(R^a_{ss'} + \gamma V^*(s') \right).$$

Ровно то же самое можно сделать и для функции $Q(s,a)$, последовательно повторяя вычисления по следующей формуле:

$$Q(s,a) := \sum_{s' \in S} P^a_{ss'} \left(R^a_{ss'} + \gamma \sum_{a'} \pi(s,a') Q(s,a') \right),$$

до сходимости. И с оптимальным $Q^*(s,a)$ все обстоит точно так же: повторяем

$$Q^*(s,a) := \sum_{s' \in S} P^a_{ss'} \left(R^a_{ss'} + \gamma \max_{a'} Q^*(s,a') \right),$$

пока не сойдется; кстати, потом можно вычислить оптимальную функцию значений как $V^*(s) := \max_a Q^*(s,a)$.

Давайте подсчитаем оптимальные значения $V^*(s)$ и $Q^*(s,a)$ в примере, изображенном на рис. 9.2. Снова начинаем сначала:

$$V^{\pi}(s_0) = \max\left\{\frac{1}{3}\left(1 + V^{\pi}(s_1)\right) + \frac{2}{3}\left(-1 + V^{\pi}(s_2)\right), \right.$$
$$\left. \frac{1}{3}\left(-1 + V^{\pi}(s_3)\right) + \frac{2}{3}\left(1 + V^{\pi}(s_4)\right)\right\}.$$

Игра по-прежнему заканчивается в s_5 и s_6, так что $V^*(s_5) = V^*(s_6) = 0$. Воспользуемся этим, чтобы подсчитать V^* от состояний после первого круга игры:

$$V^*(s_1) = \max\left\{\frac{1}{2}(-5) + \frac{1}{2}(5), \frac{1}{2}(5) + \frac{1}{2}(-5)\right\} = 0.$$

Аналогично и $V^*(s_4) = 0$. А вот для s_2 и s_3 теперь ситуация более оптимистическая:

$$V^*(s_2) = \max\{-5, 5\} = 5, \quad V^*(s_3) = \max\{5, -5\} = 5.$$

Подставляя это в выражение для $V^*(s_0)$, получим:

$$V^*(s_0) = \max\left\{\frac{1}{3} + \frac{2}{3}(-1 + 5), \frac{1}{3}(-1 + 5) + \frac{2}{3}\right\} = 3.$$

Смотрите-ка — мы можем выиграть, причем с немалым в среднем счетом! Чтобы узнать, как выиграть, нужно подсчитать функцию Q^*; сделаем это для начального состояния s_0, подставляя остальные значения сразу (они считаются очевидным образом):

$$Q^*(s_0,\text{Чет}) = \frac{1}{3}(1 + \max_a Q^*(s_1,a)) + \frac{2}{3}(-1 + \max_a Q^*(s_2,a)) = \frac{1}{3}(1+0) + \frac{2}{3}(-1+5) = 3,$$
$$Q^*(s_0,\text{Нечет}) = \frac{1}{3}(-1 + \max_a Q^*(s_3,a)) + \frac{2}{3}(1 + \max_a Q^*(s_4,a)) = \frac{1}{3}(-1+5) + \frac{2}{3}(1+0) = 2.$$

Это значит, что на первом круге нужно выбирать не имеющее большую вероятность немедленной победы действие Нечет (мы же знаем, что противник любит нечетные числа), а наоборот: нужно сначала поддаться противнику, чтобы успокоить его и заставить повторить свое действие — тогда-то мы его и достанем, а второй круг куда важнее первого. И весь этот хитрый план получился естественным образом из уравнений Беллмана.

Мы проиллюстрировали некоторые результаты наших вычислений на рис. 9.3: серым на нем показаны те состояния и действия, в которые мы никогда не попадем, если будем следовать оптимальной стратегии, задаваемой Q^*, а черным — те, в которые попасть можем (в состоянии s_1 было все равно, какое действие выбирать, так что выбрали Чет).

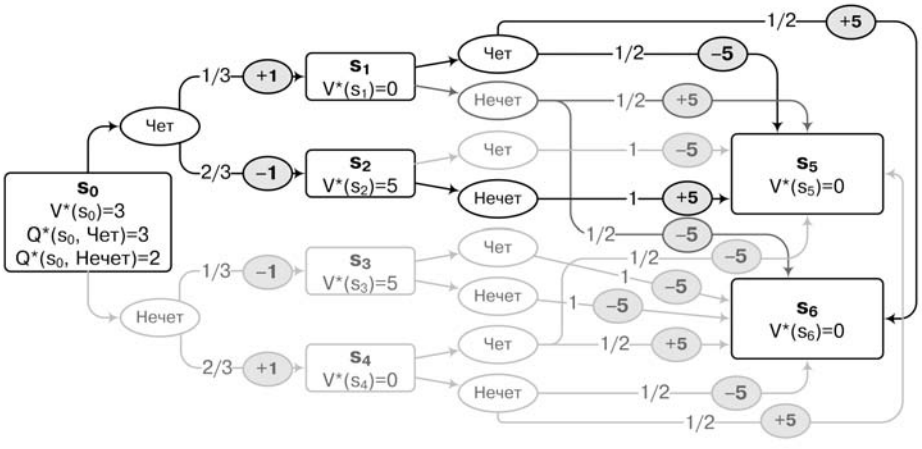

Рис. 9.3. Игра «чет-нечет»: значения функции состояний и оптимальная стратегия

Заметим, что пересчет в нашем алгоритме использует информацию от всех возможных состояний-предшественников; поскольку состояний обычно очень много, все эти формулы пока что фактически неприменимы на практике. Но можно сформулировать аналогичное правило и для одного «тренировочного примера», состоящего из текущего состояния s, произведенного действия a, состояния s', в которое мы после этого перешли, и непосредственной награды r:

$$Q(s,a) := Q(s,a) + \alpha\left(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right).$$

Теоретические гарантии у этого метода появляются, если каждая пара (s,a) встречается в процессе обучения бесконечное число раз, s' выбирают из распределения $P^a_{ss'}$, а r сэмплируется со средним $R(s,a)$ и ограниченной дисперсией.

Впрочем, на самом деле наша цель — не функции V и Q, а оптимальная стратегия π^*, и ищем мы V^π только затем, чтобы улучшить π. Как это можно сделать?

Итак, мы достаточно долго уже рассуждали о функциях значений состояний и пар состояние–действие, многое поняли, а теперь пора переходить к самому вкусному. Практически все современные подходы к обучению основаны на очень простом, но очень мощном принципе, который называется *TD-обучение* (TD-learning), от слов *temporal difference* (временные разности). Общий принцип TD-обучения таков: давайте обучать состояния на основе уже обученных нами оценок для последующих состояний. Каждый раз, когда мы делаем очередной переход, мы немножко «подтягиваем» функцию V для того состояния (или функцию Q для пары состояние-действие), из которого мы вышли, к значению функции V для того состояния (или функции Q для пары состояние-действие), в которое мы попали.

Простейший алгоритм TD-обучения, так называемое $TD(0)$-обучение, выглядит так. Сначала нужно инициализировать функцию $V(s)$ и стратегию π произвольно (обычно как-то случайно), а затем на каждом эпизоде обучения:

- инициализировать s;
- для каждого шага в эпизоде:
  - выбрать a по стратегии π;
  - сделать a, пронаблюдать результат r и следующее состояние s;
  - обновить функцию V на состоянии s по формуле:

$$V(s) := V(s) + \alpha\left(r + \gamma V(s') - V(s)\right);$$

  - перейти к следующему шагу, присвоив $s := s'$.

Вот и все! На первый взгляд кажется, что здесь происходит какая-то черная магия: мы обучаем $V(s)$ на основе других значений $V(s')$... но их мы тоже инициализировали случайным образом! Однако все работает: смысл в том, чтобы использовать уже обученные закономерности для поиска более глубоких закономерностей. Сначала обучатся значения $V(s)$ на состояниях, которые непосредственно ведут к настоящим наградам r, а потом эти значения будут постепенно передавать накопленные в них «знания» дальше, к предыдущим состояниям. В результате обучение получится целенаправленным, TD-обучение гораздо быстрее и эффективнее, чем другие стратегии.

Впрочем, и его можно улучшить. $TD(0)$ смотрит на один шаг вперед; можно рассмотреть алгоритм, который будет обновлять состояния сразу на много шагов назад. Он называется $TD(\lambda)$, и λ здесь играет ту же роль, что и раньше: мы обновляем каждое состояние u по формуле:

$$V(u) := V(u) + \alpha\left(r + \gamma V(s') - V(s)\right)\epsilon(u)$$

на основе значения $\epsilon(u)$ (от слова *eligibility*), которое показывает, насколько часто это состояние посещалось в прошлом. Значения $\epsilon(u)$ точно так же экспоненциально затухают с показателем λ:

$$\epsilon(s) = \sum_{k=1}^{t} \lambda^{t-k}[s = s_k],$$

где $[s = s_k]$ равно единице, если $s = s_k$, и нулю в противном случае.

Если $\lambda = 0$, $TD(\lambda)$ превращается в уже знакомый нам $TD(0)$. А значения $\epsilon(u)$ можно тоже хранить и пересчитывать в реальном времени после каждого нового перехода:

$$\epsilon(u) := \begin{cases} \lambda\epsilon(u) + 1, & \text{если текущее состояние — это } u, \\ \lambda\epsilon(u) & \text{в противном случае.} \end{cases}$$

Конечно, на практике обновляют не все состояния, а несколько с самыми большими значениями $\epsilon(u)$, которые в реальной реализации обычно хранятся в приоритетной очереди.

В реальный алгоритм принцип TD-обучения можно превратить разными способами. Если реализовать его для функции Q совсем в лоб, получится алгоритм SARSA, который представляет собой on-policy TD-обучение, после каждого очередного перехода $(s_t,a_t,r_{t+1},s_{t+1},a_{t+1})$[1] делает следующее обновление:

$$Q(s_t,a_t) := Q(s_t,a_t) + \alpha\left(r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)\right).$$

Аналогично, SARSA(λ) обновляет значения на всех парах (s,a) с учетом $\epsilon(s,a)$:

$$\begin{aligned}
Q(s,a) &:= Q(s,a) + \alpha\left[r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)\right]\epsilon_t(s,a),\\
\epsilon_t(s,a) &:= \gamma\lambda\epsilon_{t-1}(s,a) + [s = s_t, a = a_t].
\end{aligned}$$

При этом, правда, стратегия должна быть мягкой, например ϵ-жадной с уменьшающимся ϵ, чтобы алгоритм мог исследовать новые возможные действия, но со временем она должна как-то плавно сходиться; это вносит дополнительные инженерные сложности в реализацию, потому что от выбора характера затухания ϵ или другого параметра «нежадности» может многое зависеть.

Поэтому более популярно off-policy TD-обучение функции Q, которое обычно так и называется *Q-обучением* [559]. Здесь мы сразу решаем уравнения Беллмана относительно максимумов:

$$Q(s_t,a_t) := Q(s_t,a_t) + \alpha\left(r_{t+1} + \gamma\max_a Q(s_{t+1},a) - Q(s_t,a_t)\right).$$

Теперь Q напрямую аппроксимирует оптимальную функцию Q^*, независимо от стратегии; это значит, что мы можем придерживаться абсолютно любой стратегии, а обучаться все равно будут правильные оптимальные значения Q^*. Аналогично можно определить и Q(λ):

$$\begin{aligned}
Q(s,a) &:= Q(s,a) + \alpha\left(r_{t+1} + \gamma\max_{a'} Q(s_{t+1},a') - Q(s_t,a_t)\right)\epsilon_t(s,a),\\
\epsilon_t(s,a) &:= \gamma\lambda\epsilon_{t-1}(s,a) + [s = s_t, a = a_t],
\end{aligned}$$

только теперь еще надо забывать следы, если мы не следуем стратегии: $\epsilon_t(s,a) := 0$, если $Q(s_t,a_t) \neq \max_a Q(s_t,a)$.

О классическом обучении с подкреплением можно говорить еще долго, но в этой главе нам будет достаточно основных принципов TD- и Q-обучения. Напоследок же — небольшое замечание по поводу источников для интересующихся.

[1] Догадайтесь, почему алгоритм называется SARSA.

В обучении с подкреплением сложилась ситуация, совершенно уникальная для информатики: лучшей книгой по обучению с подкреплением до сих пор остается книга Ричарда Саттона и Эндрю Барто [519], изданная еще в 1998 году[1]! Тем, кто хочет подробнее разобраться в обучении с подкреплением, рекомендуем просто прочесть эту книгу от начала до конца (она доступна свободно), она небольшая и несложная, но в ней очень ясно и подробно изложены и все основные идеи этого раздела, и многие их расширения.

9.3. От TDGammon к DQN

> Его взор, прежде взор простого наблюдателя, впивается теперь глубже, сумрачнее, упорнее, неотступнее...
>
> *С. Цвейг. Зигмунд Фрейд*

Все те алгоритмы обучения с подкреплением, о которых мы до сих пор говорили, использовали значения вида $Q(s,a)$ или $V(s)$, причем они пытались получить эти значения в явном виде, например, обучить функцию Q^* на всех возможных входах (s,a), чтобы потом максимизировать ожидаемый результат, найдя оптимальную стратегию. Но ведь этих самых состояний s обычно астрономическое число — представьте, сколько возможных позиций в игре го! А если число состояний умножить на число возможных действий в них, получится еще больше. Поэтому в реальности, конечно, никто не пытается перечислить все возможные состояния, построив и обучив огромную таблицу $Q(s,a)$. Подход обычно такой:

- давайте представим входы, то есть состояния $s \in S$ и действия $a \in A$, в виде каких-то характерных признаков, так, чтобы размерность входа перестала быть астрономической;
- а функцию $Q(s,a)$, в которой раньше значения на разных входах были независимы друг от друга, представим как какую-то параметрическую модель машинного обучения $Q(s,a;\boldsymbol{\theta})$, на вход которой подаются признаки, описывающие s и a;
- тогда функция $Q(s,a;\boldsymbol{\theta})$ — это просто сложная функция из признаков в одно вещественное число (ожидаемый выигрыш, или его вероятность в случае

[1] Небольшое лирическое отступление: в информатике действительно обычно свежие книги лучше старых просто потому, что информатика развивается очень быстро, и новые книги передают новые результаты, которые часто поглощают старые, дают свежий взгляд и более глубокое понимание. Но в математике в целом ситуация зачастую обратная. Мы неоднократно убеждались, что, например, лучшие учебники по математическому анализу и дифференциальным уравнениям писали в начале и середине XX века, с расчетом на тогдашних инженеров; в этих книгах очень хорошо дается именно понимание происходящего, упор делается на связь математики с реальным миром и решение содержательных задач, а не только механическое переписывание формул. Так что если у вас на антресолях пылятся старые книги по математике, откройте их, возможно, будете приятно удивлены.

бинарного исхода), и ее параметры $\boldsymbol{\theta}$ можно пытаться обучать методами машинного обучения;

- входами для обучения будет, согласно идее TD-обучения, каждый очередной переход $(s_t, a_t, r_{t+1}, s_{t+1})$;
- и каждый шаг обучения выглядит так: агент делает ход a из состояния s, переходит в новое состояние s', получая за это непосредственную награду r, а затем делает один шаг обучения функции $Q(s,a;\boldsymbol{\theta})$ со входом (s,a) и выходом $\max_{a'} Q(s',a';\boldsymbol{\theta})+r$ (напомним, что в подавляющем большинстве случаев $r = 0$, награду обычно дают только в самом конце эпизода обучения); кроме того, агент может также сделать такие шаги по отношению к предыдущим позициям, обновив веса не только для последнего входа, но и для нескольких предыдущих.

В этой схеме нейронные сети, как обычно, отлично работают в качестве универсального черного ящика, который может приблизить любую функцию, в том числе и функцию $Q(s,a)$. Например, если под обучением понимать обычный градиентный спуск, то мы обновляем веса нейронной сети по следующему правилу:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha\,(y_{t+1} - y_t)\sum_{k=1}^{t}\lambda^{t-k}\nabla_{\boldsymbol{w}} y_k,$$

где y_k — выход нейронной сети на шаге k, а λ — это параметр затухания, который показывает, насколько сильно нужно учитывать результаты последнего хода для пред-предыдущего, пред-пред-предыдущего и так далее. Получается, что при $\lambda = 0$ мы обновляем веса только на основании последнего хода, а при $\lambda = 1$ рассматриваем всю историю от начала времен (то есть от начала эпизода обучения, одной «партии»).

Первые успехи нейронных сетей на поприще обучения с подкреплением относятся к временам весьма древним. В 1992 году Джеральд Тезауро (Gerald Tesauro) разработал программу, получившую название TD-Gammon [525, 526]; название вполне логичное, ведь играла эта программа в нарды (backgammon), и делала это при помощи TD-обучения. TD-Gammon работает в точности как написано выше, используя обычную неглубокую нейронную сеть с одним скрытым слоем для обучения по правилу выше. Нарды оказались очень благодатной почвой для такого подхода, потому что ход игры зависит от бросков кубика, а это значит, что можно исследовать значительную часть пространства поиска, просто играя с самим собой, кубики позаботятся о всех необходимых случайностях. И это именно то, что делала TD-Gammon для обучения: просто играла сама с собой миллионы партий, постепенно обучаясь все лучше и лучше; волшебство здесь в том, что никакого обучающего набора оказывается не нужно.

TD-Gammon ждал оглушительный успех. С точки зрения обучения добрым знаком было то, что модель хорошо масштабировалась: при увеличении размера сети и времени, выделенном на обучение, сеть начинала играть все лучше. По мере

обучения TD-Gammon сначала обучалась простейшим элементам стратегии и тактики нард, а потом постепенно начинала выделять более сложные признаки. В результате программа даже на простом представлении позиции без всяких хитростей начинала играть очень сильно, а при добавлении к представлению нескольких вручную порожденных признаков из более ранней программы Neurogammon [524] TD-Gammon начинала успешно соперничать с людьми-чемпионами.

Единственными слабостями TD-Gammon были ошибки в использовании удвоения ставок и плохая игра в эндшпиле: TD-Gammon смотрела только на два хода вперед, а эндшпили даже в нардах требуют более глубокого расчета. Поэтому в выставочных играх TD-Gammon немного уступила тогдашним чемпионам. Тем не менее, TD-Gammon оказала серьезное влияние на развитие нард: некоторые классические позиции были полностью переоценены игроками-людьми, потому что оценка TD-Gammon противоречила человеческой интуиции и практике игр; тот же эффект можно было наблюдать позднее и в шахматах, а теперь и в го.

Однако TD-Gammon надолго осталась практически единственной успешной программой, основанной на идеях обучения нейронной сети с подкреплением. Разумеется, исследователи тут же попытались применить аналогичных подход к шахматам и го, но у них мало что получилось. Последовали даже относительно пессимистические работы, которые показывали, что Q-обучение с нелинейными параметрическими приближениями (а нейронная сеть — это самое нелинейное приближение, которое только бывает) часто расходится [541], а успех TD-Gammon объяснялся исключительно упомянутым выше эффектом равномерного распределения обучения по пространству поиска за счет кубиков [428].

К счастью, эти пессимистические прогнозы не оправдались; по мере того, как развивалась революция глубокого обучения, начали появляться и попытки моделировать функции V и Q, а также окружающую агента среду, при помощи глубоких сетей. После ранних работ [210, 469] прорыв, окончательно определивший направление для современных успехов, был достигнут в работе Володимира Мниха (Volodymyr Mnih) с соавторами из компании Google DeepMind, в которой они применили идеи обучения с подкреплением к ранним, но от этого не менее привлекательным играм для приставок и аркадных автоматов Atari; да-да, тем самым *Pong*, *Space Invaders*, *Breakout* и другим классическим хитам 1980-х годов.

Первая работа была выложена на arXiv в 2013 году [426], а немного улучшенная и примененная к большему числу разных игр модель была описана в статье 2015 года, вышедшей в одном из главных научных журналов мира, *Nature* [238][1]. Этот подход получил название *глубокого обучения с подкреплением* (deep reinforcement learning), а сети, обученные таким способом, называются *глубокими Q-сетями* (deep Q-networks, DQN).

[1] Кстати, статья про AlphaGo тоже появилась в Nature [355]; результаты, безусловно, блестящие, но сама идея того, что статьи о компьютерных программых, играющих в настольные и компьютерные игры, печатают в Nature, для нас всегда была немного контринтуитивной.

У DQN в варианте [238] есть некоторые небольшие, но важные усовершенствования по отношению к базовой архитектуре, описанной выше. Во-первых, практика показывает, что обучаться непосредственно на последовательных кадрах игры — плохая идея: соседние кадры слишком похожи друг на друга, сильно коррелируют, причем со временем их распределение, естественно, сдвигается в зависимости от хода игры, но остается локализованным.

Это мешает эффективному обучению, ведь в обычной постановке задачи обучения мы предполагаем, что тренировочные данные независимы, а распределение данных со временем не меняется. Поэтому по мере обучения DQN сначала накапливает некоторый опыт, сохраняя свои действия и их результаты на протяжении какого-то времени, а потом выбирает из этого опыта случайный мини-батч отдельных примеров для обучения, взятых в случайном порядке; для накопления опыта при обучении использовалась ϵ-жадная стратегия. Формально говоря, на каждом шаге обучения t мы:

- выбираем следущее действие a_t (в ϵ-жадной стратегии мы выбираем случайное действие с вероятностью ϵ и $a_t = \arg\max Q(s_t, a; \boldsymbol{\theta})$ в противном случае;
- делаем это действие, получая награду r_t и следующее состояние s_{t+1}; новая единица опыта (s_t,a_t,r_t,s_{t+1}) записывается в память;
- затем выбираем из памяти случайный мини-батч таких «единиц опыта» для обучения; для простоты пусть это будет одна единица (s_j,a_j,r_j,s_{j+1});
- подсчитываем выход сети y_j (об этом чуть ниже) и делаем один шаг градиентного спуска для функции ошибки $L = \big(y_j - Q(s_j,a_j;\boldsymbol{\theta})\big)^2$; это значит, что мы сдвигаем веса сети на

$$\nabla_{\boldsymbol{\theta}} L = 2\big(y_j - Q(s_j,a_j;\boldsymbol{\theta})\big)\nabla_{\boldsymbol{\theta}} Q(s,a;\boldsymbol{\theta}).$$

Во-вторых, важную роль для успеха сыграло то, что при обучении DQN сеть, которая отвечала за целевую функцию (target network), была отделена от сети, которая собственно обучается. Практика показывает, что если применять TD-обучение напрямую, слишком мощные аппроксиматоры (например, нейронные сети) обнаруживают несомненные склонности к галлюцинациям: они быстро заходят в какие-то ими самими придуманные локальные экстремумы и начинают очень глубоко исследовать совершенно бессмысленные части пространства поиска, бесцельно тратя ресурсы и фактически не обучаясь.

К счастью, этой беде относительно легко помочь: достаточно сделать так, чтобы сеть не сразу использовала обновленную версию в целевой функции, а обучалась достаточно долгое время по старым образцам, прежде чем сделать уже полноценный глобальный апдейт. Иными словами, в приведенном выше алгоритме мы считаем

$$y_j = \begin{cases} r_j, & \text{если эпизод обучения закончился,} \\ r_j + \gamma \max_{a'} Q(s_j,a_j;\boldsymbol{\theta}_0), & \text{если нет,} \end{cases}$$

где $\boldsymbol{\theta}_0$ — это некие зафиксированные веса, которые не меняются после каждого тестового примера, меняются только $\boldsymbol{\theta}$. В какой-то момент, обычно один раз за один или несколько эпизодов обучения, нужно возвращаться к этим весам целевой функции и присваивать $\boldsymbol{\theta}_0 := \boldsymbol{\theta}$.

Де-факто у нас параллельно обучаются две сети: одна определяет поведение, а другая — целевую функцию; структура у них одна и та же, и обучаются они одинаково на одних и тех же данных, но одна сеть постепенно отстает от другой, время от времени скачком «догоняя» первую.[1]

В-третьих, стоит отметить, что на практике обычно применяется архитектура, в которой возможное действие агента $a \in A$ не подается на вход, а просто у сети столько выходов, сколько возможных действий, и на входе $s \in S$ сеть пытается предсказать результаты каждого действия (после чего, естественно, выбирает максимальный). Это важное улучшение, потому что оно позволяет получить ответы для сразу всех действий за один проход по сети, что ускоряет происходящее в разы, а сеть от этого сильно сложнее не становится, ведь основная часть ее «логики» остается прежней и используется заново.

И наконец, еще одно важное улучшение состоит в переходе к так называемой *dueling network*: мы разделяем Q-сеть на два канала, один из которых вычисляет оценку позиции $V(s;\boldsymbol{\theta})$, которая является функцией позиции и не зависит от текущего действия a, а другой — зависящую от действия *advantage function* (функцию преимущества) $A(s,a;\boldsymbol{\theta})$. На последнем этапе они просто складываются:

$$Q(s,a;\boldsymbol{\theta}) = V(s;\boldsymbol{\theta}) + A(s,a;\boldsymbol{\theta}).$$

Таким образом, одна часть сети обучается оценивать позицию как таковую, а другая — предсказывать, насколько полезны будут разные наши действия в этой позиции. Кстати, обратите внимание, что мы, конечно же, не можем отличить одно от другого в обучающей выборке, это всего лишь слегка измененная архитектура сети, а обучаем мы по-прежнему функцию $Q(s,a;\boldsymbol{\theta})$, но это изменение в архитектуре дает нужный «намек» и часто существенно улучшает результаты.

В работах [238, 426] этот подход применили к тому, чтобы играть в игры Atari, причем входом служило не состояние игры (это потребовало бы отдельных инженерных решений для каждой игры, а именно этого и хотелось бы избежать), а собственно игровое поле в виде картинки из пикселей, той самой, которую видит на экране игрок-человек. Точнее говоря, на вход подавалась конкатенация последних четырех кадров игры, то есть можно сказать, что долгой памятью агента не снабдили, он едва-едва мог оценить скорости разных объектов на экране.

[1] В алгоритмических разделах информатики в подобных ситуациях часто используются звериные метафоры; например, в «алгоритме зайца и черепахи» при поиске цикла в графе указатель-заяц постепенно обгоняет указатель-черепаху «на круг», и мы ждем, когда они снова встретятся. А здесь у нас скорее «алгоритм зайца и кенгуру»: сеть, определяющая поведение, обучается быстро и понемногу, маленькими шажками, а сеть, определяющая целевую функцию, время от времени величавым прыжком нагоняет зайца в очередном чекпойнте.

Единственное послабление, которое сделали агенту, состояло в том, что его не заставляли учить читать цифры очков[1], а просто давали число очков из игры в явном виде (замазывая его при этом на картинках); в итоге пришли даже к еще более простой схеме: выдавали награду +1 за каждое позитивное достижение в игре и −1 за каждое негативное событие (в Atari это обычно потеря очередной «жизни»).

В результате получилось, что такая сеть, ничего не зная собственно о «правилах игры», просто по входной картинке и целевой функции обучилась играть во многие игры Atari лучше человека. Любопытно, однако, что не во все. Игры без долгосрочной стратегии вроде *Breakout* или *Video Pinball* покорились DQN без вопросов, результаты были вдесятеро выше человеческих (напомним, что речь идет о людях–экспертах, лучших игроках мира в соответствующей игре). Результаты на уровне человеческих или немного лучше получились в играх вроде *Pong* и *Space Invaders*, где стратегия есть, но она не очень обязательна и не слишком долгосрочна. А вот в играх, где нужно думать надолго вперед, ничего не получилось; например, в *Montezuma's Revenge* DQN играть совершенно не обучилась, устойчиво получая нулевые результаты. Похожий подход привел к успеху и в игре го, но здесь, чтобы та самая «долгосрочная стратегия» все же сработала, добавятся еще несколько важных идей, поэтому об AlphaGo мы поговорим отдельно в разделе 9.4.

И еще одно замечание: обучение с подкреплением для игр и других подобных отличается от большинства других задач машинного обучения тем, что здесь у нас есть фактически *неограниченный* источник новых тренировочных примеров. В случае игр *Atari* мы можем запускать симулятор игры сколько угодно раз, пробуя самые разные стратегии и обучая модель хорошо играть. В случае игры в нарды или го модель может сколько угодно играть сама с собой, тоже получая практически неограниченную последовательность примеров для обучения. И хотя примеры эти будут в некотором смысле зависеть от текущей версии модели — в конце концов, именно она (обычно с добавленным случайным шумом для исследования) играет, когда создаются новые тренировочные примеры, — это все равно не идет ни в какое сравнение с обычными ситуациями, когда есть некий фиксированный датасет, и максимум, что вы можете сделать, — добавить в него немного случайного шума, как в шумоподавляющем автокодировщике (см. раздел 5.5). С другой стороны, обучение хорошей стратегии методами обучения с подкреплением в любой достаточно сложной ситуации — это процесс долгий и сложный: те самые модели DQN для игр *Atari* даже на самых современных видеокартах обучаются по несколько дней, прежде чем могут показать сколь-нибудь разумные результаты.

Поэтому вполне логично, что в глубоком обучении с подкреплением основной упор дальнейших исследований пока что был сделан не на максимально эффективное использование каждого тренировочного примера (их легко нагенерировать еще), а на том, чтобы максимально быстро обучаться.

[1] Да и с чего бы агент взял, что эти цифры нужно увеличивать и что на них вообще есть какой-то порядок... Вообще, целеполагание — это еще слабое место искусственного интеллекта. У нейронных сетей пока явные проблемы с мотивацией: может быть, искусственный интеллект уже давно поработил бы мир, да как-то не хочется...

Следующий прорыв в скорости обучения был сделан на почве *асинхронного* обучения с подкреплением, использующего две особенности обучения DQN:

- во-первых, обучение происходит не после каждого хода, а путем случайного сэмплирования из накопленной памяти, и для обучения нужно сначала собрать некоторое количество тестовых примеров, а только потом обновлять веса модели;
- во-вторых, сеть, которая генерирует целевые значения для функции потерь, — это не та же самая сеть, которая обучается после каждого мини-батча, она может и даже должна существенно отставать от сети, генерирующей поведение агента.

Поэтому оказалось, что обучение с подкреплением можно разбить на несколько практически независимых частей, которые должны делиться друг с другом новостями только в определенные достаточно далеко друг от друга отстоящие моменты времени, а между ними могут работать совершенно параллельно и независимо:

- должен все-таки быть некий центральный процесс, сервер, который хранит текущие значения параметров, обновляет их по мере надобности и раздает всем остальным;
- первый вид процессов — это собственно «игроки», которые взаимодействуют с окружающим миром и нарабатывают новый опыт; им нужно время от времени получать от сервера обновления параметров модели (они используются при выборе действий), а сами они просто накапливают единицы опыта в виде тех самых четверок (s_t,a_t,r_t,s_{t+1}) и передают накопленный опыт в общее хранилище памяти;
- второй вид процессов, «обучатели», получают из хранилища памяти опыт в виде мини-батчей единиц опыта и считают градиенты функции ошибки; им нужна для этого сеть, генерирующая целевые значения, и текущая, так что «обучатели» находятся в более тесном контакте с сервером; но заметим, что они по-прежнему совершенно независимы, каждый из них считает свое собственное значение градиента и свои собственные обновления для весов модели;
- наконец, собственно сервер собирает все эти обновления, применяет их к хранящейся у него модели, и в какой-то момент (обычно регулярный, но достаточно редкий) раздает обновленную модель обратно «игрокам» и «обучателям», а также обновляет модель, генерирующую целевые значения; получается, что синхронизация в такой архитектуре, конечно, нужна, но ее можно делать относительно редко.

В работе [354] этот подход был применен для того, чтобы распараллелить обучение с подкреплением на кластер из нескольких компьютеров: в разработанной авторами архитектуре с характерным названием *Gorila* (от слов General Reinforcement Learning Architecture) каждый из процессов, описанных выше, может быть реализован на отдельном компьютере.

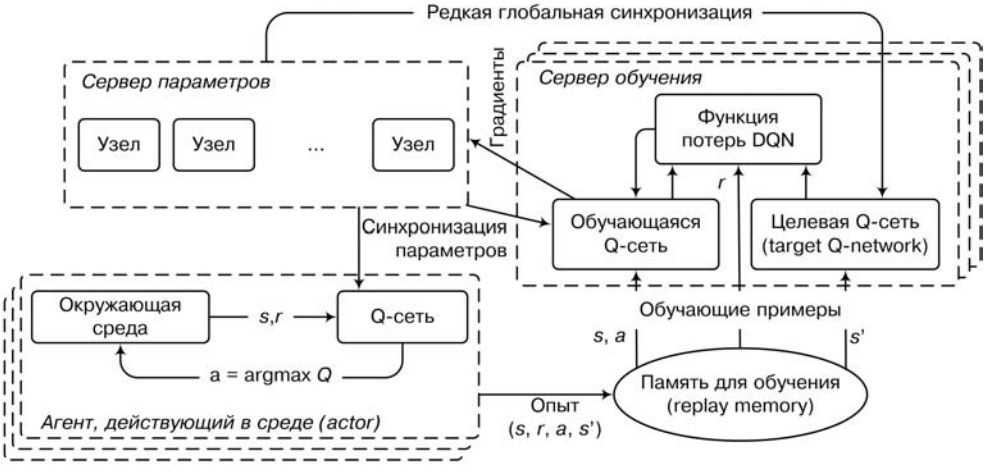

Рис. 9.4. Распределенная архитектура Gorila [354]

А потом выяснилось, что можно распараллелить все это еще эффективнее, если в качестве отдельных агентов использовать потоки одного и того же процессора; тогда они могут просто обращаться к одной и той же модели в памяти, но при этом все равно делать свое дело относительно независимо.

Мы изобразили архитектуру Gorila на рис. 9.4. Обратите внимание, что параллелизовано здесь буквально *все*. Во-первых, несколько независимых «игроков» (actors) действуют каждый в своей копии среды, порождая новые единицы опыта и передавая их в глобальную память (replay memory). Во-вторых, «обучателей» (learners) тоже несколько: каждый содержит копию Q-сети и вычисляет градиенты по своему очередному мини-батчу тренировочных примеров, взятому из памяти; для подсчета градиентов используется целевая Q-сеть (target network). В-третьих, градиенты эти отправляются на сервер параметров, который тоже содержит несколько параллельных узлов (shards), каждый из которых хранит и обновляет свою отдельную часть вектора параметров сети $\boldsymbol{\theta}$.

Любопытно, что асинхронное обучение оказывается не просто быстрее, но и существенно лучше. Причины такого эффекта точно не известны, но, по всей видимости, дело в том же эффекте, который помогает стохастическому градиентному спуску: распределенное асинхронное обучение приводит к тому, что модель не пытается слишком детально исследовать одну и ту же часть пространства поиска. Разные «игроки» ведут себя по-разному, и их «предвзятые» отношения к различным частям пространства поиска усредняются. В работе [11] первые результаты DQN на играх Atari были превышены в разы при том, что общее время обучения сократилось; любопытно, что для обучения при этом не использовались видеокарты, и все делалось в 16 потоков на обычном современном процессоре.

9.4. Бамбуковая хлопушка

> Рэндзю — удел простолюдинов, в шахматы играют герои, го — игра богов.
>
> *Японская пословица*

> Мэйдзину уже не раз приходилось вести матчи, где на карту ставилась его судьба, и в такого рода матчах он не проиграл ни разу. Его игра была мощной и до того, как он стал Мэйдзином, а игры, которые он вел, уже получив титул, уверили весь мир в его непобедимости. То, что он сам в нее верил, мало того, хотел верить, — лишь усугубляло трагедию.
>
> *Я. Кавабата. Мэйдзин*

29 января 2016 года. Google торжественно объявляет о том, что достигнута важна договоренность: через месяц состоится матч между Ли Седолем (Lee Sedol), национальным героем Кореи, великим чемпионом по игре го, который и сейчас остается одним из лучших игроков мира, и недавно появившейся программой AlphaGo, разработанной в Google DeepMind и несколько месяцев назад победившей чемпиона Европы Фань Хуэя[1].

Кажется, что для Ли Седоля появление программы, за победу в матче с которой Google предлагает миллион долларов, — всего лишь способ заработать: он абсолютно уверен в себе и не допускает даже мысли о поражении. На пресс-конференции Ли Седоль отдает должное разработчикам программы и признает, что AlphaGo, возможно, представляет собой новый шаг в развитии компьютерного го, но о результате матча говорит однозначно: «В этот раз я, конечно, выиграю со счетом 4:1 или 5:0, но я уверен, что через 2–3 года создатели AlphaGo захотят взять у меня реванш. Вот тогда мне будет действительно непросто победить» [479].

8 марта 2016 года. На пресс-конференции накануне матча Ли Седоль попрежнему не сомневается в том, что он сильнее компьютера, но заявления чемпиона становятся более осторожными: «Теперь мне кажется, что AlphaGo может до некоторой степени имитировать человеческую интуицию. Мне, наверное, стоит немножко волноваться за исход матча... Но я все равно уверен в своей победе».

12 марта 2016 года. Ли Седоль пожимает руку Адже Хуаню (Aja Huang), делавшему ходы за AlphaGo, и сдается в третьей подряд партии матча. Счет становится 3:0 в пользу AlphaGo; Ли Седоль сможет в блестящем стиле победить

[1] Целомудренная транскрипция; а сам факт того, что переехавший в Европу из Китая второй профессиональный дан Фань Хуэй стал чемпионом Европы, к сожалению, неплохо характеризовал уровень европейского го. Но сейчас уже намечается несомненный прогресс, и здесь Россия идет впереди остальной Европы: в России появились сразу несколько игроков профессионального уровня, а в 2020 году во Владивостоке пройдет чемпионат мира среди любителей — до этого этот турнир почти всегда проходил в Японии, несколько раз в Китае и буквально по одному разу в Корее и Таиланде.

в четвертой партии, но через три дня матч закончится со счетом 4:1, а сама программа AlphaGo получила почетный сертификат девятого профессионального дана как выполнившая требования профессиональной федерации. Подводя итоги матча, Ли Седоль написал: «Неделя с AlphaGo была для меня как бамбуковая хлопушка[1]. Слабости "го Ли Седоля" обнажились полностью. Старые, косные взгляды профессионального сообщества го разбились вдребезги. Нам нужно снова обдумать и пересмотреть ходы, которые мы принимали как должное» [478].

Скажете, это история безмерной гордыни все еще молодого корейского профессионала? Ведь, казалось бы, давно понятно, что с компьютерами в настольных играх, тем более в играх с полной информацией, шутки плохи: Гарри Каспаров проиграл DeepBlue еще за двадцать лет до описываемых событий, а современные шахматные программы играют на полтысячи очков Эло сильнее даже самых лучших гроссмейстеров. Неужели Ли Седоль был настолько неосведомлен о вычислительной мощи современных компьютеров?

Вовсе нет. AlphaGo действительно стала сенсацией; практически ни один эксперт не предсказывал перед матчем ни столь уверенной победы программы, ни ее победы вообще[2]. В этой главе мы поговорим о том, почему компьютерное го — это так сложно, опишем, как устроена AlphaGo и что она добавила к уже существовавшим программам для игры в го, подробнее рассмотрим собственно глубокие модели, использующиеся при этом, и обсудим еще несколько интересных применений глубокого обучения с подкреплением. Но сначала нам нужно поговорить о том, что это, собственно, такое.

Спустя год после матча с Ли Седолем, на специально собранном конгрессе *Future of Go Summit*, прошедшем в конце мая 2017 года в Вужене (Китай), состоялся матч между AlphaGo и номером 1 мирового рейтинга, китайцем Кэ Цзе (Ke Jie). Любопытно, что против Кэ Цзэ играла так называемая версия AlphaGo Master, которая обучалась больше на играх с самой собой, чем на играх профессионалов го, и работала всего лишь с одного компьютера на Google Cloud с TPU, а не на распределенной сети из 1920 CPU и 280 GPU, как игравшая с Ли Седолем программа. Тем не менее, больших проблем у AlphaGo не возникло: хотя первая партия была очень напряженной, и оценка позиции самой AlphaGo очень долго оставалась равной, в итоге Кэ Цзе был повержен со счетом 3:0. Кроме того, AlphaGo выиграла партию против команды из пяти ведущих профессионалов.

Что же произошло? Как устроена AlphaGo и почему ей удалось обыграть человека в самую «человеческую» из дискретных игр с полной информацией?

---

[1] Бамбуковые хлопушки используются в практике дзен-медитации: в большую хлопушку время от времени бьют для того, чтобы медитирующие не засыпали; отсюда и образ, использованный Ли Седолем — AlphaGo «открыла ему глаза».

[2] Прогнозы российских любителей и профессионалов можно прочесть в журнале российской гофедерации «оГо»: http://rusgo.org/magazine13/. Один из авторов книги тоже принял участие в опросе, и явно переосторожничал, предсказав, что Ли Седоль может поначалу недооценить AlphaGo и проиграть одну партию...

Сначала — немного истории. Первые программы для игры в го, как и для шахмат, появились еще в конце шестидесятых; самая известная из них связана с именем Альберта Зобриста. Однако быстро стало понятно, что научиться хорошо играть пока невозможно, и про го на некоторое время забыли. Вернулся интерес в середине восьмидесятых, когда у многих появились персональные компьютеры, а также появились шахматные программы, играющие достаточно сильно (но пока еще не обыгрывающие чемпионов). С 1987 года чемпионат по го среди компьютерных программ проводился под эгидой Фонда Ина (Ин Чанци, Ing Changqi — тайваньский мультимиллионер, любитель и популяризатор игры го); фонд учредил и премию в 40 миллионов тайваньских долларов (около полутора миллионов американских) для создателей программы, которая смогла бы обыграть одного из китайских или тайваньских профессионалов игры го (любой силы, по сути нужно было обыграть игрока первого профессионального дана). Приз поддерживался до 2000 года, в девяностые появились такие известные и важные программы, как Handtalk и Many Faces of Go... и как же они играли?

В 1997 году *Deep Blue* обыграл в официальном матче Гарри Каспарова. В том же 1997 году программа Handtalk получила часть приза фонда Ина, победив со счетом 2:1 трех любителей игры го уровнем от второго до шестого дана[1]. Звучит не так плохо, это примерно аналогично кандидату в мастера спорта по шахматам... вот только любителям этим было от 11 до 13 лет, а играли они *на 11 камнях форы*. В го есть очень логичная и удобная система гандикапа, которая часто применяется в турнирах, но в партиях между людьми больше девяти камней форы практически никогда не встречаются; обыграть любителя среднего дана на 11 камнях форы — это примерно как обыграть кандидата в мастера по шахматам, который дал вам ферзя и ладью вперед.

Почему же го — это так сложно? Во многих других играх (например, в шахматах) отлично работает так называемый *поиск с альфа-бета-отсечением* (alpha-beta search): мы строим минимакс-дерево ходов, выбрасываем ходы, которые заведомо хуже других, тем самым обрезая бесперспективные ветви дерева, и ведем поиск в глубину. Примерно так работал и *Deep Blue*. Но го — это совсем другое дело: в обычной шахматной позиции 30–40 возможных ходов, из которых многие сразу приводят к значительным материальным потерям, и их можно тут же перестать рассматривать, а в го около 200 возможных ходов (любое поле на доске 19×19, кроме занятых), и из них «вполне разумных» тоже около сотни. С оценкой позиции тоже сложно: в шахматах можно посчитать материал и какие-то простые позиционные эвристики, и если разрыв большой, то оценка позиции, скорее всего, очевидна; в го тоже может упасть большая группа, но это редкость, обычно потери будут территориальными и очень трудными для оценки. Кстати, вот вам очень сложная для

[1] В го есть существенная разница между профессионалами (это официальный титул) и любителями — любительские даны считаются (и являются) значительно слабее профессиональных, то есть сила эти игроков не достигала требуемого уровня первого профессионального дана.

автоматизации задача: по данной *финальной* позиции в го, когда люди уже закончили игру, определить, кто же, собственно, выиграл. С тактикой, где компьютеры должны, по идее, «обсчитывать» людей, тоже в го не все так просто: тактика там всегда завязана на стратегию и взаимодействие камней по всей доске, а кроме того, человеку в го считать значительно проще, чем в шахматах (потому что «фигуры» почти никогда не двигаются). И это мы еще не говорим о ко-борьбе[1], которую программы всегда вели просто отвратительно.

Первая революция в компьютерном го произошла в 2006 году, когда для этой игры начали использовать *поиск по дереву Монте-Карло* (Monte-Carlo tree search, MCTS) [190, 515]. Впервые реализованный в MoGo, а затем и во всех других программах, MCTS работает на удивление просто: чтобы оценить позицию, мы запускаем *случайные симуляции*, начинающиеся с этой позиции, смотрим, в каких ветках черные или белые выиграли больше *случайных* партий, а затем рекурсивно повторяем этот поиск в самых перспективных узлах получающегося дерева. Основная часть MCTS – формула, по которой выбирают узел для дальнейшего анализа. Все это основано на тех же многоруких бандитах, о которых мы говорили в разделе 9.1, и алгоритм UCT (upper confidence bound UCB1 applied to trees) выбирает узел с максимальным приоритетом

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln t}{n_i}},$$

где w_i — число побед в узле i, n_i — число симуляций в нем, c — параметр, часто $\sqrt{2}$, а $t = \sum_i n_i$ — общее число симуляций; сравните с UCB1.[2]

Идея доигрывать позицию до конца случайными ходами выглядит очень странно, но тут же пошли более разумные результаты: в 2008 году MoGo достиг уровня дана на доске 9×9, а в 2009 году Fuego победила одного из топ-игроков на доске 9×9, но с «настоящим» го, на доске 19×19, все шло медленнее: в 2009 году Zen достиг уровня первого любительского дана, в 2012 году новая версия Zen выиграла матч у любителя второго дана на полноразмерной доске со счетом 3:1, а в 2013 году CrazyStone выиграл у профессионала девятого дана (наивысший ранг в го), получив четыре камня форы. Но весь этот прогресс был довольно медленным и постепенным, в игре компьютерных программ было много очевидных пробелов, и ничто не предвещало внезапный успех AlphaGo.

---

[1] Ко-борьба — это специальный раздел тактики го, который происходит из запрета на повторение позиции. Мы не будем вдаваться в детали, хотя всецело рекомендуем читателям попробовать го на практике: правил там очень мало (куда меньше, чем в шахматах), но из них происходит одна из самых глубоких и интересных существующих игр.

[2] Идея алгоритма MCTS, конечно, применима не только к го. Упомянем самый общий вариант, General Game Playing. Это соревнование между программами, которым на вход подаются правила новой, ранее не известной им игры (естественно, не текстом, а в унифицированном формате), и они должны тут же начать в нее играть и выигрывать. В 2007 году программа CadiaPlayer, основанная на MCTS, стала чемпионом по General Game Playing, и с тех пор этот подход стал общепринятым [159, 160].

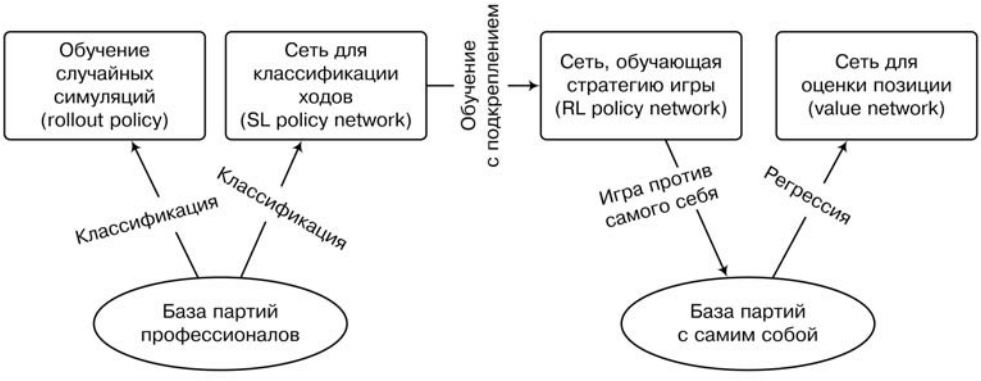

Рис. 9.5. Архитектура AlphaGo [355]

Давайте более подробно посмотрим на то, как именно работает AlphaGo [355]; общая ее архитектура изображена на рис. 9.5. В AlphaGo несколько сетей, которые обучались последовательно. Сначала SL policy network (от слов supervised learning) обучалась предсказывать ходы: 13-слойная сверточная сеть выделяет признаки по всей доске и обучается на профессиональных играх (30 миллионов позиций с сервера KGS), на выходе у сети получается распределение вероятных ходов $p_\sigma(a \mid s)$.

Сеть SL policy network в составе AlphaGo была способна правильно предсказать 57,0 % ходов профессионалов — это очень, очень много, если учесть, что в большинстве позиций вполне разумных ходов может быть несколько. Таким образом, после этого у нас есть сеть, которая может неплохо предсказывать ходы. Кроме того, вдобавок мы обучаем быструю, но не такую умную стратегию p_π: точность у нее «всего» 24,2 %, но одну позицию она способна оценить за две наносекунды (!) вместо трех миллисекунд.

Вторая сеть — RL policy network, от слов reinforcement learning. Это уже сеть, реализующая стратегию обучения с подкреплением, и работает она так: все та же базовая структура 13-слойной сверточной сети инициализируется весами SL policy network, а затем играет сама с собой (точнее, с одной из предыдущих итераций обучения). Веса при этом дообучаются так, чтобы максимизировать вероятность выигрыша. На этом этапе, после (долгого, очень долгого) обучения с подкреплением получается RL policy network, которая самостоятельно, безо всякого расчета вариантов, просто глобальным приближением к оптимальной стратегии выигрывает 80 % игр против SL policy network и около 85 % игр против *Pachi*, одной из лучших MCTS-программ!

Затем нужно использовать полученную стратегию, чтобы обучить собственно функцию оценки позиции $V_\theta(s)$. Мы будем предсказывать результат игры по позиции нейронной сетью (value network), структура которой опять похожа на policy

network, но теперь имеет только один выход (предсказание результата). Если пытаться обучать ее на наборе профессиональных игр, мы довольно быстро ударимся в оверфиттинг, данных в профессиональных партиях явно недостаточно. Поэтому $V_\theta(s)$ обучается на наборе игр RL policy network против самой себя; таких мы можем сгенерировать сколько угодно. В результате получается сеть для глобальной оценки позиции, которая работает очень быстро по сравнению с MCTS и дает качество на уровне MCTS с RL policy (но в 15 тысяч раз быстрее) и гораздо лучше, чем обычный MCTS.

Ну а теперь, когда готова функция оценки позиции, можно и деревья посчитать. AlphaGo строит MCTS-подобное дерево, в котором априорные вероятности инициализируются через SL policy network как $p_\sigma(a \mid s)$. В каждом листе L AlphaGo подсчитывает значение value network $V_\theta(s_L)$ и результат случайных доигрываний этой позиции z_L, порожденных при помощи стратегии p_π, а потом объединяет полученные результаты. Любопытно, что в экспериментах AlphaGo SL policy network для построения дерева сработала лучше (видимо, потому, что она дает более разнообразные варианты), но функцию оценки позиции лучше строить на основе более сильно играющей RL policy network.

Вот так и получилась «бамбуковая хлопушка» для Ли Седоля и одно из самых ярких событий в мире искусственного интеллекта в последние годы. А летом 2016 года на европейском го-конгрессе в Петербурге чемпион Европы Фань Хуэй сделал доклад о том, как AlphaGo может помочь развитию самой игры. Во-первых, успех AlphaGo стал огромным имиджевым успехом для го как игры; во время недели матча Ли Седоля с AlphaGo в мире продавалось в десять раз больше гобанов (досок для игры), чем обычно. Что же до сути игры, Фань Хуэй говорил о том, что AlphaGo стала первой программой, которая действительно следовала важнейшей заповеди го, сформулированной легендарным китайским игроком Сюсаку Хонинбо: «Жадность никогда не побеждает». Многие великие игроки отмечали, что в го очень важно играть спокойно, оставив эмоции; таким спокойным стилем известен никогда не улыбающийся великий чемпион недавнего прошлого, кореец Ли Чанг-Хо; и в этом тоже, конечно, компьютерные программы превосходят людей.

AlphaGo, по словам Фань Хуэя, может дать новое видение мировому го, предложить новые ходы, которые даже лучшие игроки не всегда могут увидеть: в описанных выше пяти партиях с Ли Седолем было несколько таких примеров. Наконец, Фань Хуэй впервые показал визуализацию того, как AlphaGo на самом деле выбирает ходы, и визуализацию оценки позиции, которую делает AlphaGo в процессе игры. Он показал несколько конкретных примеров того, как AlphaGo меняет классическую оценку стандартных позиций, отклоняется от проверенных десятилетиями дзесеки (стандартных продолжений) и получает позиции не хуже, а скорее лучше стандартных. И эти ходы уже начинают появляться в партиях «живых» профессионалов. По мнению Фань Хуэя, этими ходами AlphaGo «освободила» го, показала, что многие продолжения, которые раньше считались плохими и никогда не игрались просто «из общих соображений», теперь станут вполне допустимыми.

читала про шахматы, что игра до ~23 хода играется по стандартным дебютам, и эндшпили тоже стандартные, всем грустно, но есть прикольные решения типа Фишер-рэндом

9.5. Градиент по стратегиям и другие применения

> Виктор неторопливо поднялся с места, оглядел внимательно Электроника, словно выискивая в нем слабое место.
>
> — Превзойдет ли робот человека в обучении? — спросил он.
>
> — Это может случиться, — ответил Элек, — если человек сам перестанет учиться. Машине, между прочим, обучаться труднее, чем человеку, — добавил он.
>
> *Е. Велтистов. Новые приключения Электроника*

До сих пор практически все применения глубокого обучения с подкреплением, которые мы рассматривали, касались исключительно игр: одна из двух главных описанных выше работ о DQN училась играть в го, а другая — и вовсе в компьютерные игры из восьмидесятых. Надо сказать, что на победе в го применения обучения с подкреплением к играм вовсе не закончились; в частности, сейчас стоит ожидать ярких результатов в разных компьютерных играх. Любопытно, что хотя в таких играх, как *StarCraft* и *DotA 2*, не последнюю роль играют рефлексы, микроконтроль и умение точно рассчитать свои действия во времени и пространстве (когда закончится это заклинание, на какое расстояние стреляет морпех и т. п.), и в этом компьютер изначально имеет огромное преимущество над человеком, на самом деле пока в этих киберспортивных дисциплинах боты не могут составить никакой конкуренции людям. Так, на крупнейшем турнире по *DotA 2*, прошедшем в 2017 году *The International 7*, в качестве прорывного результата компания *DeepMind* представила бота, который смог победить лучших игроков в матче один на один — то есть в урезанной версии *DotA 2*, где собственно ничего кроме микроконтроля и нету. Но все-таки нам представляется, что успехи в киберспорте у компьютерных программ тоже не за горами: за *DotA 2* и *StarCraft* уже всерьез взялся *DeepMind*, так что дело обязательно пойдет.

Но есть ли более серьезные, «настоящие» применения для этих моделей? Конечно, есть! Первое и главное из них — *роботика*. Если попытаться обучить искусственную руку переносить вазу с одного стола на другой, неизбежно придется столкнуться с теми же проблемами, с которыми сталкивается обучающийся ходить ребенок из начала этой главы: мы не можем дать датасет с правильными поворотам шарниров искусственной руки, а можем только сказать, разбилась ли ваза. Кроме того, даже если тренировочные вазы не будут разбиваться, все равно очевидно, что миллионы попыток мы в реальной жизни сделать не сможем, то есть обучаться надо быстро. То же относится и, например, к вождению автомобилей: невозможно точно сказать, куда надо поворачивать руль в каждый конкретный момент, можно только сказать, что врезаться ни во что не надо. Поэтому неудивительно, что задачи роботики всегда были в центре обучения с подкреплением, а одна из первых и самых классических задач обучения с подкреплением — это задача балансировки шеста на платформе [33, 359].

Эти задачи отличаются от тех, которые мы до сих пор рассматривали, тем, что в них непрерывны не только состояния (в играх *Atari* состояний тоже было слишком много, чтобы их можно было подсчитать), но и *действия*: шарнир или руль можно повернуть на любой угол. Поэтому в этих задачах обычно используются не TD-обучение и Q-обучение, которые оказались так полезны для разнообразных игр, а другие методы.

В частности, центральный метод обучения с подкреплением в роботике — это *градиентный спуск по стратегиям* (policy gradient) [419]. Идея очень проста: рассмотрим стратегию π, которая делает действие a в зависимости от текущего состояния s и собственных параметров $\boldsymbol{\theta}$, $a \sim \pi(a \mid s; \boldsymbol{\theta})$. Мы хотим максимизировать некоторую целевую функцию J, например $\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]$; поскольку вознаграждения зависят от стратегии, а та задана параметрически, получается, что целевая функция — это функция от $\boldsymbol{\theta}$.

Давайте не будем строить никаких моделей окружающей среды, а просто будем напрямую оптимизировать целевую функцию по $\boldsymbol{\theta}$ градиентным спуском: подсчитаем $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ и сдвинемся в нужную сторону. На первый взгляд кажется, что сейчас мы не можем подсчитать градиент $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ аналитически, как мы делали раньше: слишком уж сложная это функция, ведь теперь в $J(\boldsymbol{\theta})$ спрятан долгий путь применения стратегии $\pi(\theta)$ по ходу целого эпизода, а то и вовсе уходящий на бесконечность. Один возможный выход — вычислять градиент приближенно, численно — давайте немножко поварьируем каждый из параметров θ_k и подсчитаем $\frac{\partial J}{\partial \theta_k} \approx \frac{J(\boldsymbol{\theta}+\epsilon \boldsymbol{u}_k)-J(\boldsymbol{\theta})}{\epsilon}$, где $\boldsymbol{u}_k$ — вектор из нулей с одной единицей в k-й компоненте. Это медленно (нужно вычислять $J(\boldsymbol{\theta} + \epsilon \boldsymbol{u}_k)$ отдельно для каждого параметра) и весьма неточно, но в некоторых приложениях работает неплохо.

Но есть, оказывается, способ лучше! Давайте все-таки посмотрим более пристально на функцию $J(\boldsymbol{\theta})$. Для простоты будем считать, что у нас марковский процесс принятия решений из одного шага, мы начинаем в состоянии $s \in S$ с вероятностью $p_0(s)$, и после одного действия $a \in A$ получаем награду R_s^a. Тогда:

$$J(\boldsymbol{\theta}) = \sum_{s \in S} p_0(s) \sum_{a \in A} \pi(a \mid s; \boldsymbol{\theta}) R_s^a, \text{ и}$$

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \sum_{s \in S} p_0(s) \sum_{a \in A} R_s^a \nabla_{\boldsymbol{\theta}} \pi(a \mid s; \boldsymbol{\theta}).$$

Применим нехитрый трюк — заметим следующее равенство:

$$\nabla_{\boldsymbol{\theta}} \pi(a \mid s; \boldsymbol{\theta}) = \pi(a \mid s; \boldsymbol{\theta}) \frac{\nabla_{\boldsymbol{\theta}} \pi(a \mid s; \boldsymbol{\theta})}{\pi(a \mid s; \boldsymbol{\theta})} = \pi(a \mid s; \boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} \log \pi(a \mid s; \boldsymbol{\theta}).$$

Это значит, что:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \sum_{s \in S} p_0(s) \sum_{a \in A} R_s^a \pi(a \mid s; \boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} \log \pi(a \mid s; \boldsymbol{\theta}) = \mathbb{E}_{\pi(\boldsymbol{\theta})} \left[R_s^a \nabla_{\boldsymbol{\theta}} \log \pi(a \mid s; \boldsymbol{\theta}) \right].$$

Оказывается (этот факт так и называется — policy gradient theorem), что этот результат можно обобщить и на более продолжительные процессы. В целом:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi(\boldsymbol{\theta})} \left[\nabla_{\boldsymbol{\theta}} \log \pi(a \mid s; \boldsymbol{\theta}) Q^{\pi(\boldsymbol{\theta})}(s,a) \right].$$

Фактически это значит, что мы-таки можем подсчитать градиент функции $J(\boldsymbol{\theta})$. Так, например, классический алгоритм REINFORCE [571] просто берет результат $Q^{\pi(\boldsymbol{\theta})}(s,a)$ из текущего эпизода обучения (считая, что последний запуск агента представляет собой разумный сэмпл из этого распределения), а затем обновляет веса $\boldsymbol{\theta}$ при помощи градиента как показано выше. Это работает даже в случае бесконечного горизонта [35].

А более сложные методы строят модель окружающей среды (эта модель называется *критиком*), который служит для того, чтобы более точно оценивать $Q^{\pi(\boldsymbol{\theta})}(s,a)$ [427]. Глубокие нейронные сети здесь служат для того же самого, что и в играх: ими можно моделировать как стратегию $\pi(\boldsymbol{\theta})$, так и функцию $Q^{\pi(\boldsymbol{\theta})}(s,a)$. Чтобы сделать их обучение эффективным, правда, придется применить некоторые трюки, на которых мы сейчас не будем подробно останавливаться; в частности, нужно использовать разные варианты так называемого guided policy search [321], который лучше подходит для стратегий с очень большим числом параметров, как всегда и бывает в нейронных сетях.

Сейчас одно из важных направлений глубокого обучения с подкреплением в роботике — это непосредственный поиск стратегий, в котором нейронные сети обучаются целиком (end-to-end) и распознавать объекты в окружающем мире (компьютерным зрением, то есть по сути сверточными сетями), и представлять сами стратегии [143]. Для этого тоже можно адаптировать guided policy search [91, 413]. Современные подходы к глубокому обучению с подкреплением на основе градиентного спуска по стратегиям и других методов успешно применяется для того, чтобы искать выходы из лабиринтов [310], манипулировать объектами при помощи роботов с манипуляторами [110], двигаться в нужную сторону, руководствуясь визуальными данными (например, обучить дрон с видеокамерой следить за данным объектом) [316, 521], хватать объекты на основе данных с видеокамеры [120], и многое другое (см., например, подробный обзор [324]).

Есть и другие интересные применения. Например, в работе [201] DQN очень интересным образом применяется для того, чтобы улучшать порождение текста. В разделе 8.1 мы уже рассказывали об архитектурах типа encoder-decoder, в которых текст сначала кодируется в некоторое внутреннее представление, а затем декодируется обратно, возможно в виде ответа в диалоге или перевода на другой язык. А в [201] декодирование происходит итеративно:

- сначала декодер порождает первую версию выхода;
- в качестве списка возможных действий для DQN служит перечень слов, которые могут быть использованы, чтобы модифицировать текущее состояние выхода;

- и далее DQN пытается модифицировать выход так, чтобы в результате предложение стало лучше по отношению к целевой метрике качества.

Таким образом, DQN может самостоятельно выбирать, в каком порядке вносить изменения. На практике получается, что DQN сначала концентрируется на том, чтобы декодировать более простые и понятные части предложения, а потом использует их для того, чтобы уточнить более сложные места. В [201] эта идея применялась исключительно для того, чтобы построить автокодировщик текстов, то есть задача была в том, чтобы по поданному на вход предложению восстановить его же из сжатого представления кодировщика, а метрикой качества была BLEU-похожесть между входным и выходным предложением. Но аналогичный подход, скорее всего, можно применить и ко всем остальным задачам обработки естественного языка, где используются архитектуры encoder-decoder.

В целом, одним из важных трендов в современном обучении с подкреплением является то, что оно оказывается полезным не только для таких непосредственных приложений, как вождение автомобилей, манипуляция объектами или игра в го, но и для многих других задач, в том числе, казалось бы, обычных задач обучения с учителем. Нам кажется, что на этом пути нас ждет еще много интересных открытий, часть из которых, возможно, придется и на вашу долю, дорогие читатели. А нам пора двигаться дальше. В последней главе книги мы коснемся более теоретических материй... которые, впрочем, быстро окажутся очень даже прикладными.