21
Подкрепляемое обучение

В этой главе мы представим *подкрепляемое обучение* (reinforcement learning, RL), в котором используется подход к *машинному обучению* (МО), отличающийся от подхода, принятого в контролируемых и неконтролируемых алгоритмах, рассматривавшихся до сих пор. Подкрепляемое обучение привлекло огромное внимание в качестве основной движущей силы нескольких самых захватывающих инновационных прорывов в области ИИ. Мы увидим, что интерактивная и онлайновая природа подкрепляемого обучения делает его особенно хорошо подходящим для торговой и инвестиционной сфер.

Подкрепляемое обучение — это вычислительный подход, в котором агент целенаправленно обучает себя сам, взаимодействуя со стохастической окружающей средой, о которой агент имеет неполную информацию. Подкрепляемое обучение призвано автоматизировать то, как агент принимает решения для достижения долгосрочной цели, усваивая значение состояний и действий на основе вознаградительного сигнала. Конечная цель состоит в том, чтобы выработать линию поведения, так называемую политику, в которой закодированы правила поведения и которая увязывает состояния с действиями.

Подкрепляемое обучение считается наиболее похожим на то, как происходит усвоение знаний человеком, которое тоже нередко протекает в результате действий в реальном мире и наблюдений за последствиями. Оно отличается от контролируемого обучения тем, что вместо обобщения репрезентативных, помеченных образцов целевой концепции подкрепляемое обучение оптимизирует поведение агента по одному опытному случаю за раз, основываясь на скалярном вознаградительном сигнале. Более того, подкрепляемое обучение не останавливается на предсказаниях, а принимает в отношении целенаправленного принятия решений сквозную перспективу, включая действия и их последствия.

В этой главе вы узнаете, как формулировать задачу подкрепляемого обучения и как применять различные технические решения. Мы рассмотрим модельные и безмодельные методы, представим среду OpenAI Gym ("тренажерный зал ИИ") и объединим глубокое обучение с подкрепляемым обучением с целью тренировки агента, который передвигается по сложной окружающего его среде. Наконец, мы покажем

вам, как адаптировать подкрепляемое обучение к алгоритмической торговле путем моделирования агента, который взаимодействует с финансовым рынком, пытаясь оптимизировать целевую функцию.

Более конкретно, в этой главе мы рассмотрим следующие темы:

- как определять *задачу марковского процесса принятия решений* (Markov Decision Problem, MDP);
- как использовать цикл по ценностным значениям и цикл по политике для решения задачи марковского процесса принятия решений;
- как применять *Q*-обучение в окружающей среде с дискретными состояниями и действиями;
- как строить и тренировать агента глубокого *Q*-обучения в непрерывной окружающей среде;
- как использовать платформу OpenAI Gym для тренировки торгового агента подкрепляемого обучения.

Ключевые элементы подкрепляемого обучения

Задачи подкрепляемого обучения характерны несколькими элементами, которые отличают его от установочных условий МО, которые мы рассматривали до сих пор. В следующих двух разделах описываются ключевые функциональные средства, необходимые для определения и решения задачи подкрепляемого обучения путем усвоения политики, которая автоматизирует решения. Указанные средства используют обозначения и обычно лежат в русле положений книги "Reinforcement Learning: An Introduction" ("Подкрепляемое обучение: введение", 2018, **http://incompleteideas.net/book/RLbook2018.pdf**) Ричарда Саттона (Richard Sutton) и Эндрю Барто (Andrew Barto) и лекции Дэвида Сильвера (David Silver) в университете UCL (**http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html**), обе из которых рекомендуются для дальнейшего изучения за пределами краткого обзора, который позволяет объем этой главы.

Задачи подкрепляемого обучения призваны оптимизировать решения агента на основе целевой функции в отношении окружающего его среды. Окружающая среда предоставляет агенту информацию о своем состоянии, назначает вознаграждение за действия и переводит агента в новые состояния, в зависимости от распределений вероятностей, о которых агент может знать или не знать. Указанная среда может быть полностью или частично наблюдаемой, а также содержать других агентов. Конструирование среды, как правило, требует значительных предварительных конструкторских усилий с целью обеспечения целенаправленного самообучения агента во время тренировки.

Задачи подкрепляемого обучения отличаются сложностью их пространств состояний и действий, которые могут быть дискретными или непрерывными. Последнее требует средств МО для аппроксимирования функциональной связи между состоя-

ниями, действиями и их ценностью. Они также требуют от нас обобщения, исходя из подмножества состояний и действий, испытываемых агентом во время тренировки.

Компоненты интерактивной системы подкрепляемого обучения

Решение сложных задач обычно требует упрощенной модели, которая выделяет ключевые аспекты. На рис. 21.1 показаны отличительные особенности задачи подкрепляемого обучения.

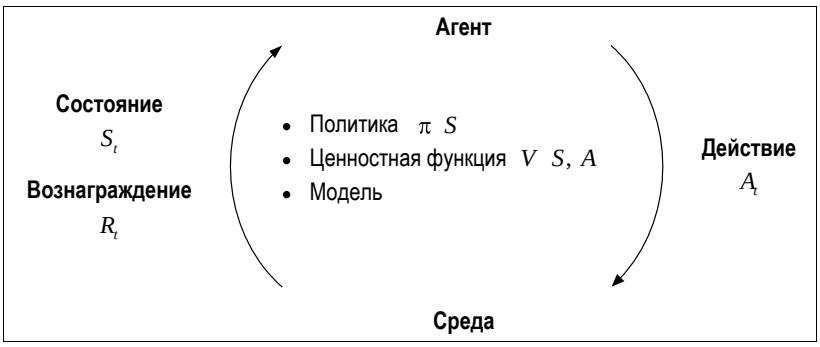

Рис. 21.1. Отличительные особенности задачи подкрепляемого обучения

К ним обычно относятся следующие:

- наблюдения агента о состоянии среды;
- множество доступных агенту действий;
- политика, которая регулирует решения агента.

В дополнение к этому окружающая среда эмитирует вознаградительный сигнал, который служит отражением нового состояния, возникающего в результате действия агента. По сути, агент обычно усваивает ценностную функцию, которая формирует его суждение о действиях. Агент имеет целевую функцию для обработки вознаградительного сигнала и трансляции ценностных (субьективных) суждений в оптимальную политику.

Политика — от состояний к действиям

В любой момент времени политика задает поведение агента. Она увязывает любое состояние, которое агент может встретить на своем пути, с одним или несколькими действиями. В простой среде с ограниченным числом состояний и действий политика может быть простой подстановочной таблицей, заполняемой во время тренировки.

В случае непрерывных состояний и действий политика принимает форму функции, которую МО может помочь аппроксимировать. Политика может также предусмат-

ривать значительные вычисления, как в случае программы AlphaZero, в которой с целью принятия решения о наилучшем действии для определенного состояния игры используется древесный поиск. Политика также может быть стохастической и назначать действиям вероятности при наличии заданного состояния.

Вознаграждения — самообучение на основе действий

Вознаградительный сигнал — это одно-единственное значение, которое посылается агенту на каждом временном шаге. Цель агента — максимизировать суммарное вознаграждение, получаемое с течением времени. Вознаграждение также может быть стохастической функцией состояния и действий. Как правило, они дисконтируются с целью обеспечить схождение и отразить затухание значения с течением времени.

Вознаграждения представляют собой единственный способ, которым агент узнает о ценности своих решений в конкретном состоянии и надлежащим образом модифицирует политику. Вследствие решающего влияния вознаградительного сигнала на процесс самообучения агента он часто является самой сложной частью конструирования системы подкрепляемого обучения.

Вознаграждения должны четко сообщать, чего агент должен добиться (в отличие от того, как он должен это сделать), причем для правильного кодирования этой информации могут потребовать знания предметной области. Например, при разработке торгового агента может потребоваться определить вознаграждение за решения о покупке, удержании и продаже актива. Они могут быть ограничены прибылью и убытком, но могут также включать волатильностные и рисковые соображения.

Ценностная функция — хорошие решения в долгосрочной перспективе

Вознаграждение обеспечивает немедленную обратную связь по факту совершения действий. Однако решение задачи подкрепляемого обучения требует решений, которые создают ценность в долгосрочной перспективе. Именно здесь возникает ценностная функция: она резюмирует полезность состояний или действий в определенном состоянии с точки зрения их долгосрочного вознаграждения[1].

Другими словами, ценность состояния — это суммарное вознаграждение, которое агент ожидает получить в будущем, начиная с этого состояния. Немедленное вознаграждение может быть хорошим индикатором будущих вознаграждений, но агент также должен учитывать случаи, когда после низких вознаграждений следуют гораздо более высокие результаты, которые могут произойти (или наоборот).

[1] Ценностная функция (value function) — функция состояния (или пары "состояния — действия"), которая оценивает, насколько хорошо агенту находиться в данном состоянии (или насколько хорошо выполнять данное действие в данном состоянии).
См. **http://www.incompleteideas.net/book/ebook/node34.html**. — *Прим. перев.*

Следовательно, ценностные оценки призваны предсказывать будущие вознаграждения. Вознаграждения являются ключевым входом, и цель ценностной оценки заключается в достижении большего вознаграждения. Однако методы подкрепляемого обучения сосредоточены на усвоении точных значений, которые активируют правильные решения, при этом эффективно задействуя (часто ограниченный) опыт.

Существуют также подходы подкрепляемого обучения, которые не опираются на ценностные функции, например, рандомизированные оптимизационные методы, такие как генетические алгоритмы или симулированное закаливание, которые направлены на поиск оптимального поведения путем эффективного разведывания пространства политики. Однако текущий интерес к подкрепляемому обучению в основном обусловлен методами, которые прямо или косвенно вычисляют ценность состояний и действий.

Методы градиентов политики — это новейшая разработка, опирающаяся на параметризованную дифференцируемую функцию политики, которая может быть непосредственно оптимизирована по отношению к цели с помощью градиентного спуска. Справочные материалы со ссылками на исследовательские работы и алгоритмы см. в репозитории GitHub по адресу: **https://github.com/PacktPublishing/Hands-On-MachineLearning-for-Algorithmic-Trading**.

Модельные агенты против безмодельных

Модельные подходы подкрепляемого обучения усваивают модель окружающей среды, позволяющую агенту планировать заранее, предсказывая последствия своих действий. Такая модель может быть использована, например, для предсказания следующего состояния и вознаграждения на основе текущего состояния и действия. Это предсказание является основой для планирования, т. е. принятия решения о наилучшем курсе действий путем рассмотрения возможных вариантов будущего до того, как они материализуются.

Более простые безмодельные методы, напротив, учатся на основе проб и ошибок. Современные методы подкрепляемого обучения охватывают целый спектр от низкоуровневых методов проб и ошибок до высокоуровневого совещательного планирования, при этом правильный подход зависит от сложности и способности усваивать окружающую среду.

Как решать задачи подкрепляемого обучения

Методы подкрепляемого обучения призваны учиться на опыте тому, как предпринимать действия, которые достигают долгосрочной цели. В этой связи агент и среда взаимодействуют над последовательностью дискретных временных шагов через интерфейс действий, вознаграждений и наблюдений о состоянии, описанных в предыдущем разделе.

Ключевые сложности в решении задач подкрепляемого обучения

Решение задач подкрепляемого обучения требует от нас обратиться к двум уникальным вызовам: проблеме учета заслуг[2] и компромиссу между разведыванием и эксплуатацией.

Учет заслуг

В подкрепляемом обучении вознаградительные сигналы могут возникать значительно позже действий, которые способствовали результату, тем самым усложняя ассоциацию действий с их последствиями. Проблема учета заслуг заключается в точном оценивании плюсов и минусов действий в заданном состоянии вследствие этих задержек. Алгоритмы подкрепляемого обучения должны находить способ распределять заслуги за положительные и отрицательные результаты среди многочисленных решений, которые могли быть вовлечены в его порождение.

Разведывание против эксплуатации

Динамическая и интерактивная природа подкрепляемого обучения приводит к тому, что агент вычисляет ценность состояний и действий до того, как он испытал все релевантные траектории на своем опыте. Следовательно, он способен принимать решения, но они основаны на неполностью усвоенном знании. Решения, которые эксплуатируют только прошлый (успешный) опыт и не разведывают неизвестную территорию, могут ограничивать контакт агента с окружающей средой и мешать ему усваивать оптимальную политику. Алгоритм подкрепляемого обучения должен уравновешивать этот компромисс — слишком малый объем разведывания, скорее всего, будет производить смещенные ценностные предположения и неоптимальные политики, в то время как слишком малый объем эксплуатации изначально препятствует самообучению.

Фундаментальные подходы к решению задач подкрепляемого обучения

Существует множество подходов к решению задач подкрепляемого обучения, в которых необходимо найти оптимальные правила поведения агента.

- Методы *динамического программирования* (dynamic programming, DP) делают зачастую нереалистичное допущение о полном знании окружающей среды, но являются концептуальной основой для большинства других подходов.

[2] Учет заслуг (credit assignment) — это процесс выявления среди множества действий, выбранных в эпизоде, тех, которые отвечают за конечный результат. И более того, это попытка определить лучшие и худшие решения, выбранные во время эпизода, для того чтобы лучшие решения подкреплялись, а худшие штрафовались. См. **https://www.igi-global.com/dictionary/credit-assignment/40159**. — *Прим. перев.*

- Методы *Монте-Карло* (Monte Carlo, MC) усваивают иноформацию об окружающей среде, а также минусах и плюсах от разных решений путем выборки из всей последовательности состояний-действий-вознаграждений.
- Усвоение *временной разности* (temporal difference, TD) значительно повышает эффективность выборки за счет усваивания информации из более коротких последовательностей. С этой целью оно опирается на бутстрапирование, под которым понимается уточнение своих оценок на основе своих же собственных предыдущих оценок.

Когда задача подкрепляемого обучения, описанная в предыдущем разделе, включает четко определенные переходные вероятности и ограниченное число состояний и действий, она может быть сформулирована как конечная задача марковского процесса принятия решений (Markov decision process, MDP), для которой динамическое программирование способно вычислять точное решение. Бóльшая часть нынешней теории подкрепляемого обучения фокусируется на конечных задачах марковского процесса принятия решений, но практические применения используются для более общих случаев. Неизвестные переходные вероятности требуют эффективного извлечения выборок с целью выяснения вероятностного распределения.

Подходы к непрерывным пространствам состояний и/или действий часто задействуют МО для аппроксимирования ценностной функции либо функции политики. Отсюда следует, что они интегрируют контролируемое обучение и, в частности, методы глубокого обучения, которые мы обсуждали в нескольких последних главах. Однако в контексте подкрепляемого обучения эти методы сталкиваются с индивидуальными трудностями:

- вознаградительный сигнал напрямую не отражает целевую концепцию, такую как помеченный образец;
- распределение наблюдений зависит от действий агента и политики, которая сама по себе является предметом процесса усвоения.

В следующих далее разделах представлены и демонстрируются различные методы решения. Мы начнем с методов динамического программирования, именуемых циклом по ценностным значениям и циклом по политике, которые ограничены конечной задачей марковского процесса принятия решений с известными переходными вероятностями. Как мы увидим в следующем разделе, они являются фундаментом для *Q*-обучения, которое основано на усвоении временной разности и не требует информации о переходных вероятностях. Оно нацелено на аналогичные динамическому программированию результаты, но с меньшими вычислениями и без принятия допущения об идеальной модели окружающей среды. Наконец, мы расширим область до непрерывных состояний и введем глубокое *Q*-обучение.

Динамическое программирование — цикл по ценностным значениям и цикл по политике

Конечные задачи марковского процесса принятия решений (MDP) представляют собой простой, но фундаментальный математический каркас. Мы вводим траектории вознаграждений, которые агент стремится оптимизировать, и определяем ценностную функцию и функцию политики, которые используются для формулирования оптимизационной задачи и уравнений Беллмана, составляющих основу методов решения.

Конечные задачи марковского процесса принятия решений

Конечные задачи марковского процесса принятия решений очерчивают рамки взаимодействия между агентом и окружающей средой как задачу принятия последовательных решений в течение ряда временных шагов $t = 1, ..., T$, которые составляют эпизод. При этом исходят из того, что временные шаги являются дискретными, однако каркас может быть расширен до непрерывного времени.

Абстракция, предоставляемая задачами марковского процесса принятия решений, делает ее применение легко адаптируемым ко многим контекстам. Временные шаги могут быть произвольными интервалами, а действия и состояния принимать любую форму, которая может быть выражена численно.

Из марковского свойства следует, что текущее состояние описывает процесс полностью, т. е. процесс не имеет памяти. Информация из прошлых состояний не добавляет ценности при попытке предсказать будущее процесса. Благодаря этим свойствам указанный каркас использовался для моделирования цен активов, которые подпадают под действие гипотезы об эффективном рынке, обсуждавшейся нами в *главе 5*.

Последовательности состояний, действий и вознаграждений

Задачи марковского процесса принятия решений выполняются следующим образом: на каждом шаге *t* агент наблюдает состояние окружающей среды $S_t \in S$ и выбирает действие $A_t \in A$, где S и A — это соответственно множества состояний и действий. На следующем временном шаге $t+1$ агент получает вознаграждение $R_{t+1} \in R$ и переходит в состояние S_{t+1}. Со временем указанная задача порождает траекторию, S_0, A_0, R_0, S_1, A_1, R_1, ..., и это продолжается до тех пор, пока агент не достигнет терминального состояния и эпизод не закончится.

Конечные задачи марковского процесса принятия решений с ограниченным числом действий *A*, состояний *S* и вознаграждений *R* включают четко определенные дискретные распределения вероятностей над этими элементами. Вследствие марков-

ского свойства эти распределения зависят только от предыдущего состояния и действия.

Вероятностный характер траекторий влечет за собой то, что агент максимизирует математическое ожидание суммы будущих вознаграждений. Кроме того, вознаграждения обычно дисконтируются с использованием коэффициента $0 \le \gamma \le 1$ с целью отражения их временной ценности. В случае задач, которые не являются эпизодическими, а продолжаются бесконечно, коэффициент дисконтирования строго меньше 1 необходим для того, чтобы избежать бесконечных вознаграждений и обеспечить схождение. Следовательно, агент максимизирует дисконтированное математическое ожидание суммы будущих возвратов R_t, обозначаемой как G_t:

$$G_t = E\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots \right] = \sum_{s=0}^{T} \gamma^s E\ R_{t+s} \quad .$$

Эта связь также может быть определена рекурсивно, поскольку сумма, начинающаяся со второго шага, совпадает с G_{t+1}, дисконтированном один раз:

$$G_t = R_{t+1} + \gamma G_{t+1}.$$

В следующем разделе мы увидим, что этот тип рекурсивной связи часто используется для формулирования алгоритмов подкрепляемого обучения.

Ценностные функции — как оценивать долгосрочное вознаграждение

Как мы упоминали ранее, политика π увязывает все состояния с вероятностными распределениями над действиями, вследствие чего вероятность выбора действия A_t в состоянии S_t может быть выражена как $\pi\ a|s = P\ A_t = a | S_t = s$. Ценностная функция оценивает долгосрочный возврат для каждого состояния или пары "состояние — действие". Принципиально важно найти политику, которая является оптимальным отображением состояний в действия.

Функция "состояние — ценность" $v_\pi(S)$ для политики π дает долгосрочную ценность *v* состояния *S*, как математическое ожидание возврата *G*, для агента, который начинает в *s* и затем всегда придерживается политики π. Она определяется следующим образом, где E_π обозначает математическое ожидание, когда агент придерживается политики π:

$$v_\pi(s) = E_\pi\ G_t | S_t = s\ = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s \right].$$

Мы можем вычислить *ценностную функцию "состояние — действие"* $q_\pi(s, a)$ схожим образом, как математическое ожидание возврата, начиная в состоянии *s*, принимая действие *a*, а затем всегда придерживаясь политики π:

$$q_\pi(s, a) = E_\pi\ G_t | S_t = s, A_t = a\ = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a \right].$$

Уравнение Беллмана

Уравнения Беллмана определяют рекурсивную связь между ценностными функциями для всех состояний s в S и любого из их последующих состояний s' в соответствии с политикой π. Они делают это, раскладывая ценностную функцию на немедленное вознаграждение и дисконтированную ценность следующего состояния:

$$v_\pi(s) \doteq E\left[G_t \mid S_t = s\right] =$$
$$= E\left[\underbrace{R_{t+1}}_{\text{вознаграждение}} + \underbrace{\gamma v\left(S_{t+1}\right)}_{\text{дисконтированная ценность}}\right] =$$
$$= \sum_a \pi(a \mid s) \sum_{s'}\sum_r p(s', r \mid s, a)\left[r + \gamma v_\pi(s')\right] \quad \forall s.$$

Указанное уравнение гласит, что для заданной политики ценность состояния должна равняться ожидаемой ценности его последующих состояний в рамках указанной политики плюс ожидаемое вознаграждение, заработанное в результате прибытия в упомянутое последующее состояние.

Из него вытекает, что если мы знаем значения последующих состояний для имеющихся в настоящее время действий, то можем заглянуть вперед на один шаг и вычислить ожидаемое значение текущего состояния. Поскольку это справедливо для всех состояний S, указанное выражение определяет множество $n = |S|$ уравнений. Аналогичная зависимость соблюдается для $q(s, a)$.

На рис. 21.2 эта рекурсивная связь подытоживается: в текущем состоянии агент выбирает действие a на основе политики π. Окружающая среда откликается назначением вознаграждения, которое зависит от результирующего нового состояния s'.

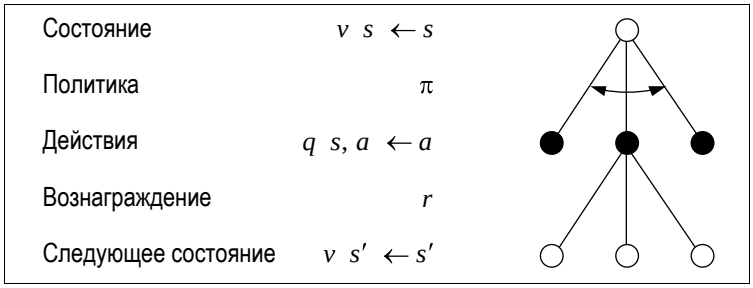

Рис. 21.2. Сводка рекурсивной связи: в текущем состоянии агент выбирает действие *a* на основе политики π

От ценностной функции к оптимальной политике

Решением задачи подкрепляемого обучения является политика, которая оптимизирует кумулятивное вознаграждение. Политики и ценностные функции тесно взаимосвязаны: оптимальная политика дает оценку ценностного значения для каждого

состояния $v_\pi(s)$ или пары "состояние — действие" $q_\pi(s, a)$, которое по крайней мере является таким же высоким, как и для любой другой политики, поскольку ценность является кумулятивным вознаграждением в рамках данной политики. Следовательно, функции оптимальной ценности $v^*(s) = \max_\pi v_\pi(s)$ и $q^*(s, a) = \max_\pi q_\pi(s, a)$ неявно определяют оптимальные политики и решают задачу марковского процесса принятия решений.

Функции оптимальной ценности v^* и q^* также удовлетворяют уравнениям Беллмана из предыдущего раздела. Эти уравнения оптимальности Беллмана могут опустить явную ссылку на политику, как это вытекает из v^* и q^*. Применительно к $v^*(s)$ рекурсивная связь приравнивает текущее значение к сумме немедленного вознаграждения за выбор наилучшего действия в текущем состоянии и ожидаемой дисконтированной ценности последующих состояний:

$$v^*(s) = \max_a q^*(s, a) = \max_a R_t + \gamma \sum_{s'} p\ s' | s, a\ v^*\ s' \ .$$

Применительно к функции оптимальной ценности "состояния — действия" $q^*(s, a)$ уравнение оптимальности Беллмана разлагает текущую ценность состояния-действия на сумму вознаграждения за вмененное текущее действие и дисконтированное ожидаемое значение наилучшего действия во всех последующих состояниях:

$$q^*(s) = R_t + \gamma \sum_{s'} p\ s' | s, a\ v^*\ s' = R_t + \gamma \sum_{s'} p\ s' | s, a\ \max_a q^*\ s, a \ .$$

Из условия оптимальности следует, что наилучшая политика состоит в том, чтобы всегда выбирать действие, которое максимизирует математическое ожидание жадным образом, т. е. рассматривать результат только одного временного шага.

Условия оптимальности, определенные двумя предыдущими выражениями, нелинейны из-за оператора max и не имеют решения в замкнутой форме. Взамен подходы на основе задачи марковского процесса принятия решений опираются на итеративность, такую как цикл по политике и цикл по ценностным значениям, или *Q*-обучение, которое мы рассмотрим далее.

Цикл по политике

Динамическое программирование — это универсальный метод решения задач, который способен раскладывать на более мелкие, перекрывающиеся подзадачи с рекурсивной структурой, позволяющие повторно использовать промежуточные результаты. Задачи марковского процесса принятия решений укладываются в указанный "законопроект" из-за рекурсивных уравнений оптимальности Беллмана и кумулятивной природы ценностной функции. В частности, принцип оптимальности применим, поскольку оптимальная политика заключается в подборе оптимального действия, а затем следования оптимальной политике.

Динамическое программирование требует знания переходных вероятностей задачи марковского процесса принятия решений. Это часто не так, но многие методы в более общих случаях придерживаются подхода, аналогичного принятому в динамическом программировании, и усваивают недостающую информацию из данных.

Динамическое программирование полезно для задачи предсказания, которая вычисляет ценностную функцию, и задачи управления, которая фокусируется на оптимальных решениях и выводит политику (при этом по ходу вычисляя ценностную функцию).

Алгоритм цикла по политике (policy iteration) призван отыскивать оптимальную политику путем повтора следующих двух шагов до тех пор, пока политика не сойдется, т. е. перестанет изменяться больше заданного порога:

- *вычисление политики* обновляет ценностную функцию на основе текущей политики;
- *улучшение политики* обновляет политику так, чтобы действия максимизировали ожидаемую одношаговую ценность.

При вычислении ценностной функции вычисление политики опирается на уравнения Беллмана. Более конкретно, для обновления значения текущего состояния оно выбирает действие, определяемое текущей политикой, и суммирует результирующее вознаграждение и дисконтированную ценность следующего состояния.

Улучшение политики, в свою очередь, изменяет политику так, что по каждому состоянию политика выполняет действие, которое производит наибольшую ценность в следующем состоянии. Это улучшение именуется *жадным*, потому что оно учитывает возврат только одного временно́го шага. Цикл по политике всегда сходится к оптимальной политике и часто делает это за относительно малое число итераций.

Цикл по ценностным значениям

Цикл по политике требует оценивания политики для всех состояний после каждой итерации, и оценивание может быть дорогостоящим, к примеру, для политики на основе дерева поиска.

Цикл по ценностным значениям (value iteration) упрощает этот процесс, сворачивая этап вычисления и улучшения политики. На каждом временном шаге он перебирает все состояния и выбирает наилучшее жадное действие на основе текущей оценки ценностного значения следующего состояния. Затем он использует одношаговый взгляд вперед, вытекающий из уравнения оптимальности Беллмана, обновляя ценностную функцию для текущего состояния.

Соответствующее правило обновления ценностной функции $v_{k+1}(s)$ является почти идентичным обновлению оценки политики — оно просто добавляет максимизацию над имеющимися действиями:

$$v_{k+1}(s) \leftarrow \max_a \sum_{s'} \sum_r p\ \ s', r \mid s, a\ \left[r + \gamma v_k\ \ s' \right].$$

Алгоритм останавливается, когда ценностная функция сошлась, и выдает жадную политику, выводимую из его вычисления ценностной функции. Он также гарантированно сходится к оптимальной политике.

Обобщенный цикл по политике

На практике существует несколько способов усечения цикла по политике, например, путем оценивания политики k раз перед ее улучшением. Это просто означает, что оператор max будет применяться только на каждой k-й итерации.

Большинство алгоритмов подкрепляемого обучения вычисляют ценностную функцию и функцию политики и для схождения к решению опираются на взаимодействие между вычислением и улучшением политики, как показано на рис. 21.3. Обобщенный подход улучшает политику по отношению к ценностной функции, при этом корректируя ценностную функцию в соответствии с политикой.

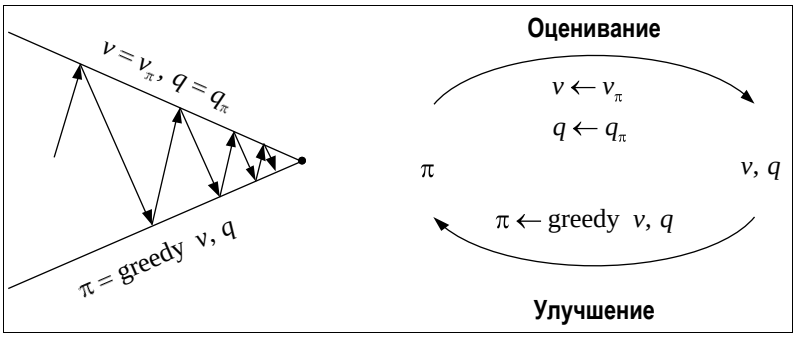

Рис. 21.3. Взаимодействие между вычислением и улучшением политики с целью схождения к решению

Схождение требует, чтобы ценностная функция соответствовала политике, которая, в свою очередь, должна стабилизироваться, действуя жадно в отношении ценностной функции. Таким образом, оба процесса стабилизируются только тогда, когда обнаруживается политика, жадная в отношении собственной ценностной функции. Это означает, что уравнение оптимальности Беллмана соблюдается, и, следовательно, функция политики и ценностная функция являются оптимальными.

Динамическое программирование на языке Python

В этом разделе мы применим цикл по ценностным значениям и цикл по политике к игрушечной окружающей среде, состоящей из решетки 3×4, изображенной на рис. 21.4 со следующими признаками.

- *Состояния*: 11 состояний, представленных в виде двумерных координат. Одно поле недоступно, и два верхних состояния в крайнем правом столбце являются терминальными, т. е. они заканчивают эпизод.

- *Действия*: ходы на каждом шаге, т. е. вверх, вниз, влево и вправо. Среда рандомизирована, вследствие чего действия могут иметь непреднамеренные результаты. Для каждого действия существуют 80%-я вероятность перехода в ожидаемое состояние и 10%-я вероятность перехода в смежном направлении (например, вправо или влево вместо вверх либо вверх или вниз вместо вправо).
- *Вознаграждения*: как показано на рис. 21.4 слева, каждое состояние в результате дает $-0,02$, за исключением вознаграждений $+1/-1$ в терминальных состояниях.

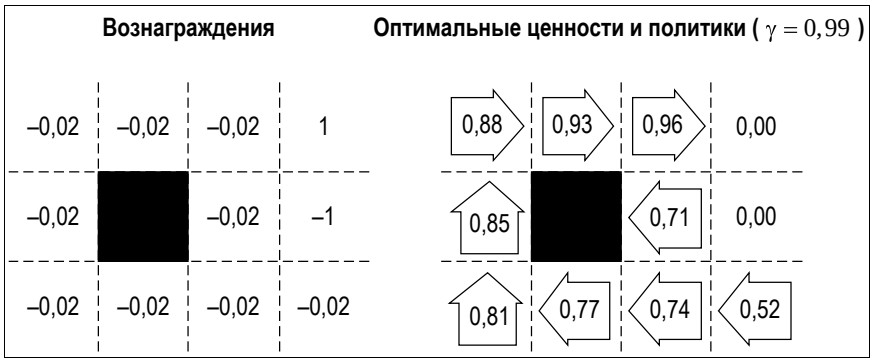

Рис. 21.4. Игрушечная окружающая среда, состоящая из решетки 3×4. Справа показаны ценностная функция и соответствующая политика в виде стрелок, которые диктуют направление движения агента

На рис. 21.4 справа демонстрируются ценностная функция и соответствующая политика в виде стрелок, которые диктуют направление движения агента.

Настройка решетчатого мира GridWorld

Начнем с определения параметров окружающей среды:

```
grid_size = (3, 4)
blocked_cell = (1, 1)
baseline_reward = -0.02
absorbing_cells = {(0, 3): 1, (1, 3): -1}

actions = ['L', 'U', 'R', 'D']

num_actions = len(actions)

probs = [.1, .8, .1, 0]
```

Нам часто нужно будет конвертировать из одномерного представления в двумерное, и наоборот, поэтому для указанной цели мы определим две вспомогательные функции; состояния являются одномерными, а ячейки — соответствующими двумерными координатами:

```
to_1d = lambda x: np.ravel_multi_index(x, grid_size)
to_2d = lambda x: np.unravel_index(x, grid_size)
```

Кроме того, для того чтобы сделать исходный код более сжатым, мы предварительно вычисляем некоторые точки данных:

```
num_states = np.product(grid_size)
cells = list(np.ndindex(grid_size))
states = list(range(len(cells)))

cell_state = dict(zip(cells, states))
state_cell= dict(zip(states, cells))

absorbing_states = {to_1d(s):r for s, r in absorbing_cells.items()}
blocked_state = to_1d(blocked_cell)
```

Мы сохраняем вознаграждения для каждого состояния:

```
state_rewards = np.full(num_states, baseline_reward)
state_rewards[blocked_state] = 0
for state, reward in absorbing_states.items():
    state_rewards[state] = reward

state_rewards

array([-0.02, -0.02, -0.02,  1.,
       -0.02,  0.,   -0.02, -1.,
       -0.02, -0.02, -0.02, -0.02])
```

Для того чтобы учесть вероятностную окружающую среду, нам также нужно вычислить вероятностное распределение над фактическим ходом для заданного действия:

```
action_outcomes = {}
for i, action in enumerate(actions):
    probs_ = dict(zip([actions[j % 4] for j in range(i, num_actions + i)], probs))
    action_outcomes[actions[(i + 1) % 4]] = probs_

action_outcomes

{'U': {'L': 0.1, 'U': 0.8, 'R': 0.1, 'D': 0},
 'R': {'U': 0.1, 'R': 0.8, 'D': 0.1, 'L': 0},
 'D': {'R': 0.1, 'D': 0.8, 'L': 0.1, 'U': 0},
 'L': {'D': 0.1, 'L': 0.8, 'U': 0.1, 'R': 0}}
```

Теперь мы готовы вычислить матрицу переходов, которая является ключевым входом в задачу марковского процесса принятия решений.

Вычисление матрицы переходов

Матрица переходов определяет вероятность оказаться в определенном состоянии S для каждого предыдущего состояния и действия A: $P\ s' | s, a$. Мы продемонстрируем библиотеку pymdptoolbox и применим один из доступных нам форматов для указания переходов и вознаграждений. Для обеих переходных вероятностей мы создадим массив NumPy размера $A \times S \times S$.

Сначала мы вычисляем целевую ячейку для каждой стартовой ячейки и ход:

```
def get_new_cell(state, move):
    cell = to_2d(state)
    if actions[move] == 'U':
        return cell[0] - 1, cell[1]
    elif actions[move] == 'D':
        return cell[0] + 1, cell[1]
    elif actions[move] == 'R':
        return cell[0], cell[1] + 1
    elif actions[move] == 'L':
        return cell[0], cell[1] - 1
```

Следующая функция использует аргументы `state`, `action` и `outcome` для заполнения переходных вероятностей и вознаграждений:

```
def update_transitions_and_rewards(state, action, outcome):
    if state in absorbing_states.keys() or state == blocked_state:
        transitions[action, state, state] = 1
    else:
        new_cell = get_new_cell(state, outcome)

        p = action_outcomes[actions[action]][actions[outcome]]

        if new_cell notin cells or new_cell == blocked_cell:
            transitions[action, state, state] += p
            rewards[action, state, state] = baseline_reward
        else:
            new_state= to_1d(new_cell)
            transitions[action, state, new_state] = p
            rewards[action, state, new_state] = state_rewards[new_state]
```

Мы генерируем значения перехода и вознаграждения, создавая структуры-заполнители и перебирая декартово произведение $A \times S \times S$ следующим образом:

```
rewards = np.zeros(shape=(num_actions, num_states, num_states))
transitions = np.zeros((num_actions, num_states, num_states))
actions_ = list(range(num_actions))

for action, outcome, state in product(actions_, actions_, states):
    update_transitions_and_rewards(state, action, outcome)
```

```
rewards.shape, transitions.shape
```

```
((4,12,12), (4,12,12))
```

Цикл по ценностным значениям

Прежде всего, мы создаем алгоритм цикла по ценностным значениям, который немного проще, поскольку он реализует вычисление и улучшение политики за один шаг. Мы улавливаем состояния, для которых нам нужно обновить ценностную функцию, исключая терминальные состояния, которые имеют значение 0 из-за отсутствия вознаграждений ($+1/-1$ назначаются стартовому состоянию), и пропускаем заблокированную ячейку:

```
skip_states = list(absorbing_states.keys())+[blocked_state]
states_to_update = [s for s in states if s not in skip_states]
```

Затем мы инициализируем ценностную функцию и устанавливаем коэффициент дисконтирования `gamma` и порог схождения `epsilon`:

```
V = np.random.rand(num_states)
V[skip_states] = 0

gamma = .99
epsilon = 1e-5
```

Алгоритм обновляет ценностную функцию с помощью уравнения оптимальности Беллмана и завершается, когда норма L_1 в V изменяется меньше, чем `epsilon` в абсолютном выражении:

```
converged = False
while not converged:
    V_ = np.copy(V)
    for state in states_to_update:
        q_sa = np.sum(transitions[:, state] * (rewards[:, state] + gamma* V), axis=1)
        V[state] = np.max(q_sa)
    if np.sum(np.fabs(V - V_)) < epsilon:
        converged = True
```

Алгоритм сходится за 16 итераций и 0,0117 секунды. Он производит следующую ниже оптимальную оценку ценностного значения, которая вместе с вмененной оптимальной политикой изображена на рис. 20.4 справа:

```
pd.DataFrame(V.reshape(grid_size))
```

```
          0         1         2         3
0  0.884143  0.925054  0.961986  0.000000
1  0.848181  0.000000  0.714643  0.000000
2  0.808344  0.773327  0.736099  0.516082
```

Цикл по политике

Цикл по политике предусматривает отдельные шаги по вычислению и улучшению. Мы определяем часть, связанную с улучшением, выбирая действие, которое максимизирует сумму ожидаемого вознаграждения и ценности следующего состояния. Обратите внимание, что мы временно заполняем вознаграждения для терминальных состояний во избежание игнорирования действий, которые приведут нас туда:

```
def policy_improvement(value, transitions):
    for state, reward in absorbing_states.items():
        value[state] = reward
    return np.argmax(np.sum(transitions * value, 2),0)
```

Мы инициализируем ценностную функцию, как упоминалось ранее, и задействуем политику случайного старта:

```
pi = np.random.choice(list(range(num_actions)), size=num_states)
```

Алгоритм чередует оценивание политики для жадно выбираемого действия и улучшения политики до тех пор, пока политика не стабилизируется:

```
converged = False
while not converged:
    pi_ = np.copy(pi)
    for state in states_to_update:
        action = policy[state]
        V[state] = np.dot(transitions[action, state],
                          (rewards[action, state] + gamma* V))
        pi = policy_improvement(V.copy(), transitions)
    if np.array_equal(pi_, pi):
        converged = True
```

Цикл по политике сходится только после трех итераций. Политика стабилизируется до того, как алгоритм найдет оптимальную ценностную функцию, при этом оптимальная политика немного отличается, в частности, предлагая ход вверх вместо более безопасного хода влево для поля рядом с отрицательным терминальным состоянием. Этого можно избежать путем ужесточения критериев сходимости, например, потребовав стабильную политику из нескольких раундов или добавления порогового значения для ценностной функции.

Решение задач марковского процесса принятия решений с помощью библиотеки pymdptoolbox

Мы также можем решать задачи марковского процесса принятия решений с помощью Python-библиотеки pymdptoolbox, которая содержит еще несколько алгоритмов, в том числе *Q*-обучение.

Для того чтобы выполнить функцию `ValueIteration`, следует инстанцировать соответствующий объект с требуемыми параметрами конфигурации и матрицами вознаграждений и переходов перед вызовом метода `run`:

```
vi = mdp.ValueIteration(transitions=transitions,
                        reward=rewards,
                        discount=gamma,
                        epsilon=epsilon)
vi.run()
```

Оценка ценностной функции соответствует результату в предыдущем разделе:

```
np.allclose(V.reshape(grid_size), np.asarray(vi.V).reshape(grid_size))
```

Функция `PolicyIteration` работает схожим образом:

```
pi = mdp.PolicyIteration(transitions=transitions,
                         reward=rewards,
                         discount=gamma,
                         max_iter=1000)
pi.run()
```

Она также дает ту же политику, но ценностная функция варьируется в зависимости от прогона и не должна обязательно достигать оптимального значения перед схождением политики.

Выводы

На рис. 21.4 справа диаграммы GridWorld показана оптимальная оценка ценностного значения, произведенная циклом по ценностным значениям и соответствующей жадной политикой. Негативные вознаграждения в сочетании с неопределенностью в окружающей среде порождают оптимальную политику, которая предусматривает уход от негативного терминального состояния.

Результаты чувствительны как к вознаграждениям, так и к коэффициенту дисконтирования. Стоимость отрицательного состояния влияет на политику в окружающих полях, и вам следует модифицировать пример в соответствующем блокноте для того, чтобы выявить пороговые уровни, которые изменяют выбор оптимального действия.

Q-обучение

Q-обучение было в числе первых инновационных прорывов в подкрепляемом обучении, когда в 1989 г. этот метод был разработан Крисом Уоткинсом (Chris Watkins) для его кандидатской диссертации (**http://www.cs.rhul.ac.uk/~chrisw/thesis.html**). Указанный подход вводит инкрементное динамическое программирование для управления задачей марковского процесса принятия решений без знания о матрицах переходов и вознаграждений или их моделирования, которые мы использовали для циклов по ценностным значениям и по политике в предыдущем разделе. Доказательство его сходимости последовало три года спустя в работе Уоткинса (Watkins) и Даяна (Dayan) (**http://www.gatsby.ucl.ac.uk/~dayan/papers/wd92.html**).

Q-обучение непосредственно оптимизирует функцию действия-ценности q, аппроксимируя q^*. Обучение происходит вне политики, т. е. алгоритму не нужно выбирать действия на основе политики, которая подразумевается только ценностной функцией. Однако сходимость требует, чтобы все пары "состояние — действие" продолжали обновляться на протяжении всего тренировочного процесса. Простой способ обеспечить это состоит в использовании ε -жадной политики.

Компромисс между разведыванием и эксплуатацией — ε-жадная политика

ε -Жадность — это простая политика, которая обеспечивает разведывание новых действий в заданном состоянии, а также эксплуатирует усвоенный опыт, рандомизированный выбором действий. ε -Жадная политика выбирает действие случайным образом с вероятностью ε и лучшее действие в соответствии с ценностной функцией в противном случае.

Алгоритм *Q*-обучения

Алгоритм *Q*-обучения неуклонно улучшает ценностную функцию состояния-действия после случайной инициализации для заданного числа эпизодов. На каждом временном шаге он выбирает действие на основе ε -жадной политики и использует темп усвоения α для обновления ценностной функции следующим образом:

$$Q\ S_t, A_t\ \leftarrow Q\ S_t, A_t\ + \alpha\left[R_{t_1} + \gamma \max_a Q\ S_{t+1}, a\ - Q\ S_t, A_t\ \right].$$

Обратите внимание, что указанный алгоритм не вычисляет математические ожидания, поскольку он знает переходные вероятности. Он усваивает функцию Q из вознаграждений, производимых ε -жадной политикой, и ее текущим значением ценностной функции для следующего состояния.

Использование значения ценностной функции для улучшения этого значения называется *бутстрапированием*, т. е. дословно, вытягиванием себя за шнурки ботинок. Алгоритм *Q*-обучения является частью алгоритмов усвоения временной разности (TD). Усвоение временной разности не дожидается получения окончательного вознаграждения за эпизод. Вместо этого алгоритм обновляет свои оценки, используя значения промежуточных состояний, которые ближе к окончательному вознаграждению. В этом случае промежуточное состояние находится на один шаг впереди.

Тренировка агента *Q*-обучения с помощью языка Python

В этом разделе мы продемонстрируем, как строить агента *Q*-обучения, используя решетку 3×4 состояний из предыдущего раздела. Мы натренируем агента в тече-

ние 2500 эпизодов, применим темп усвоения $\alpha = 0{,}1$ и $\varepsilon = 0{,}05$ для ε -жадной политики (подробности см. в блокноте gridworld_q_learning.ipynb):

```
max_episodes = 2500
alpha = .1
epsilon = .05
```

Затем мы случайно проинициализируем ценностную функцию состояния-действия как массив NumPy размера *число состояний* × *число действий*:

```
Q = np.random.rand(num_states, num_actions)
skip_states = list(absorbing_states.keys())+[blocked_state]
Q[skip_states] = 0
```

Указанный алгоритм генерирует 2500 эпизодов, которые начинаются в случайном месте и протекают в соответствии с ε -жадной политикой до завершения, обновляя ценностную функцию в соответствии с правилом *Q*-обучения:

```
for episode in range(max_episodes):
    state = np.random.choice([s for s in states if s not in skip_states])
    while not state in absorbing_states.keys():
        if np.random.rand() < epsilon:
            action = np.random.choice(num_actions)
        else:
            action = np.argmax(Q[state])
        next_state = np.random.choice(states, p=transitions[action, state])
        reward = rewards[action, state, next_state]
        Q[state, action] += alpha * (reward + gamma * np.max(Q[next_state])-Q[state,
                                                                        action])
        state = next_state
```

Эпизоды занимают 0,6 секунды и сходятся к ценностной функции, довольно близкой к результату примера с циклом по ценностным значениям из предыдущего раздела. Реализация на основе библиотеки pymdptoolbox работает аналогично предыдущим примерам (подробности см. в блокноте).

Глубокое подкрепляемое обучение

В предыдущем разделе мы увидели, как *Q*-обучение позволяет усваивать оптимальную ценностную функцию состояния-действия q^* в окружающей среде с дискретными действиями и состояниями с использованием итерационных обновлений на основе уравнения Беллмана.

В этом разделе мы выполним адаптацию указанного алгоритма к непрерывным состояниям и действиям, где мы не можем использовать табличное решение, которое просто заполняет массив значениями состояния-действия. Вместо этого мы увидим, как аппроксимировать q^*, используя нейронную сеть, построив глубокую *Q*-сеть с различными уточнениями, призванными ускорить схождение. Затем мы рассмот-

рим, каким образом платформа OpenAI Gym может использоваться для применения указанного алгоритма к окружающей среде лунного посадочного модуля.

Аппроксимация ценностной функции с помощью нейронной сети

Пространства непрерывных состояний и/или действий приводят к бесконечному числу переходов, которые делают невозможным табулирование значений состояний-действий, как в предыдущем разделе. Вместо этого мы аппроксимируем функцию *Q*, усваивая непрерывное параметризованное отображение из тренировочных образцов.

Мотивированные успехом нейронных сетей в других областях, которые мы обсуждали в *главах 17–20*, глубокие нейронные сети также стали популярными для аппроксимирования ценностных функций. Однако МО в контексте подкрепляемого обучения, где данные генерируются в результате взаимодействия модели с окружающей средой, используя (возможно, рандомизированную) политику, сталкивается с определенными трудностями.

- В случае непрерывных состояний агент не способен посетить большинство состояний и, таким образом, должен обобщать.
- Контролируемое обучение призвано обобщать на основе образца из одинаково распределенных и взаимно независимых образцов, которые являются репрезентативными и правильно помеченными. В контексте подкрепляемого обучения есть только один образец за один шаг времени, вследствие чего обучение должно происходить в режиме онлайн.
- Образцы могут быть сильно коррелированы, когда последовательные состояния похожи, и распределение поведения над состояниями и действиями не является стационарным, а изменяется в результате самообучения агента.

В следующих далее разделах мы рассмотрим несколько технических решений, призванных реагировать на эти дополнительные трудности.

Алгоритм глубокого *Q*-обучения и его расширения

Алгоритм глубокого *Q*-обучения вычисляет ценность имеющихся действий для заданного состояния с помощью глубокой нейронной сети. Он был представлен в работе компании DeepMind Technologies "Playing Atari with Deep Reinforcement Learning" ("Игры с Atari с помощью глубокого подкрепляемого обучения", 2013: **https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf**), где агенты учились играть в компьютерные игры исключительно на основе пиксельных входных данных.

Алгоритм глубокого *Q*-обучения аппроксимирует ценностную функцию действия *q*, заучивая набор весов θ многослойной *глубокой Q-сети* (Deep Q Network, DQN), которая увязывает состояния с действиями таким образом, что $q(s, a, \theta) \approx q^*(s, a)$.

Указанный алгоритм применяет градиентный спуск к функции потери, определяемой как квадрат разности между оценкой цели, вычисленной DQN-сетью

$$y_i = E\left[r + \gamma \max_{a'} Q\ s', a'; \theta^- \mid s, a\ \right],$$

и ее оценкой ценностного значения действия текущей пары "состояние — действие" $Q\ s, a; \theta$ для усвоения параметров сети:

$$L_i\ \theta_i\ = E\left[\left(\overbrace{\underbrace{y_i}_{Q\text{-цель}} - \underbrace{Q\ s, a; \theta}_{\text{текущее предсказание}}}^{TD\text{-ошибка}} \right)^2 \right].$$

Как цель, так и текущая оценка зависят от набора весов, что подчеркивает отличие от контролируемого обучения, где цели фиксируются до начала тренировки.

Вместо того чтобы вычислять полный градиент, алгоритм *Q*-обучения использует стохастический градиентный спуск и обновляет веса θ после каждого временного шага *i*. Для разведывания пространства состояний-действий агент использует ε-жадную политику, которая выбирает случайное действие с вероятностью ε, и придерживается жадной политики, которая выбирает действия с самым высоким предсказанным значением *q* в противном случае.

Воспроизведение опыта

Воспроизведение опыта сохраняет историю состояний, действий, вознаграждений и переходов в следующее состояние, которые агент испытывал на своем опыте. Оно произвольно отбирает мини-пакеты из этого опыта для обновления сетевых весов на каждом временном шаге до того, как агент выберет ε-жадное действие.

Воспроизведение опыта повышает эффективность образцов, снижает автокорреляцию образцов, собираемых во время онлайнового самообучения, и ограничивает обратную связь из-за текущих весов, которые производят тренировочные образцы, могущие привести к локальным минимумам или расхождению.

Медленно изменяющаяся целевая сеть

Для того чтобы еще больше ослабить цикл обратной связи со стороны текущих сетевых параметров на обновлениях нейросетевых весов, указанный алгоритм был расширен компанией Deep Mind за счет управления на человеческом уровне посредством глубокого подкрепляемого обучения (2015: **https://web.stanford.edu/class/psych209/Readings/MnihEtAlHassibis15NatureControlDeepRL.pdf**) и стал использовать медленно изменяющуюся целевую сеть.

Целевая сеть имеет ту же архитектуру, что и *Q*-сеть, но ее веса θ' периодически обновляются только после λ шагов, когда они копируются из *Q*-сети и оставляют-

ся постоянными в противном случае. Целевая сеть генерирует целевые предсказания на основе временной разности, т. е. она занимает место *Q*-сети, оценивая:

$$y_i = E\left[r + \gamma \max_{a'} Q\ \ s', a'; \theta^- \mid s, a\ \ \right].$$

Двойное глубокое *Q*-обучение

Экспериментально было показано, что *Q*-обучение переоценивает ценности действий, поскольку оно намеренно отбирает максимальные оценочные значения действий. Это смещение может негативно влиять на процесс усвоения и результирующую политику, если она не применяется равномерно и не изменяет предпочтения действий, как показано Хадо Ван Хасселтом (Hado van Hasselt) в работе "Deep Reinforcement Learning with Double Q-learning" ("Глубокое подкрепляемое обучение с помощью двойного Q-обучения", 2015: **https://arxiv.org/abs/1509.06461**).

Для того чтобы отделить оценку ценности действий от выбора действий, алгоритм двойного глубокого *Q*-обучения (Double Deep QLearning, DDQN) использует веса θ одной сети для выбора наилучшего действия в следующем состоянии и веса θ' другой сети для обеспечения соответствующей оценки ценности действия, т. е.:

$$y_i = E\left[r + \gamma Q\ \ s', \arg\max_a Q\ \ S_{t+1}, a, \theta_t\ \ ; \theta'_t\ \ \right].$$

Один из вариантов состоит в случайном выборе одной из двух идентичных сетей для тренировки на каждой итерации, вследствие чего их веса будут различаться. Более эффективная альтернатива — опереться на целевую сеть и вместо этого предоставить θ'.

Платформа OpenAI Gym — лунный посадочный модуль

OpenAI Gym — это платформа подкрепляемого обучения, которая предоставляет стандартизированные окружающие среды для оперативного и эталонного тестирования алгоритмов подкрепляемого обучения с использованием языка Python. Указанная платформа допускает расширение и регистрацию клиентских сред.

Окружающая среда лунного посадочного модуля Lunar Lander (LL) v2 требует, чтобы агент контролировал свое движение в двух измерениях, основываясь на дискретном пространстве действий и наблюдениях о низкоразмерных состояниях, которые охватывают положение, ориентацию и скорость. На каждом временном шаге окружающая среда обеспечивает наблюдение о новом состоянии и положительном или отрицательном вознаграждении. Каждый эпизод содержит вплоть до 1000 временных шагов. На рис. 21.5 показаны выборочные кадры успешной посадки после 250 эпизодов агентом, которого мы будем тренировать.

Более конкретно, агент наблюдает восемь аспектов состояния, включая шесть непрерывных и два дискретных элемента. Основываясь на наблюдаемых элементах,

агент знает свое местоположение, направление, скорость движения, а также произошло ли (частичное) прилунение. Однако он не знает, куда он должен двигаться, используя свои доступные действия, или наблюдать внутреннее состояние окружающей среды в смысле понимания правил, которые управляют его движением.

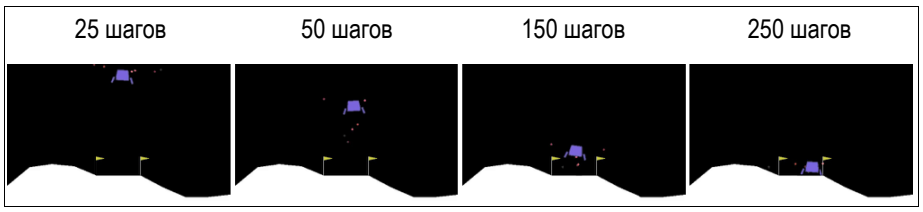

Рис. 21.5. Выборочные кадры успешной посадки агентом после 250 эпизодов

На каждом временном шаге агент управляет своим движением, используя одно из четырех дискретных действий. Он ничего не может сделать (и продолжит свой текущий путь), запускает свой главный двигатель (чтобы уменьшить движение вниз) или управляет ориентацией модуля влево или вправо с помощью соответствующих двигателей ориентации. Топливных ограничений нет.

Цель состоит в том, чтобы посадить посадочный модуль между двумя флагами на посадочную площадку в координатах (0, 0), но посадка вне площадки возможна. Агент накапливает вознаграждения в интервале 100–140 за перемещение в сторону площадки в зависимости от точного места посадки. Тем не менее уход от цели инвертирует знак вознаграждения, которое агент получил бы, двигаясь к площадке. Контакт с лунной поверхностью каждой опорой добавляет десять очков, а использование главного двигателя стоит $-0{,}3$ очков.

Эпизод завершается, если агент прилуняется или падает, соответственно добавляя или вычитая 100 очков, или после 1000 временных шагов. Решение задачи посадки лунного модуля требует достижения совокупного вознаграждения в среднем не менее 200 за более 100 эпизодов подряд.

Сеть DDQN с использованием библиотеки TensorFlow

Блокнот lunar_lander_deep_q_learning.ipynb реализует агента на основе сети DDQN, в котором используется библиотека TensorFlow и окружающая среда лунного модуля платформы OpenAI Gym. В этом разделе освещаются ключевые элементы реализации; более подробную информацию см. в блокноте.

Архитектура сети DQN

Сеть DQN была впервые применена к предметной области компьютерных игр Atari с наблюдениями о высокоразмерных изображениях с упором на сверточные слои. Более низкоразмерное представление состояний в задаче лунного модуля делает полносвязные слои более подходящим выбором *(см. главу 17)*.

Более конкретно, сеть связывает восемь входов с четырьмя выходами, которые соответствуют значениям Q для каждого действия, вследствие чего для вычисления значений действия требуется только один прямой проход. Сеть DQN тренируется на функции предыдущей потери с помощью оптимизатора Adam. Агент сети DQN использует три плотносвязных слоя по 256 единиц каждый и регуляризацию активности L_2. Использование GPU вместе с образом Docker библиотеки TensorFlow *(см. главы 17 и 19)* может значительно ускорить производительность тренировки нейронной сети.

Настройка среды платформы OpenAI

Мы начнем с создания экземпляра и извлечения ключевых параметров из окружающей среды лунного посадочного модуля (LL):

```
env = gym.make('LunarLander-v2')
state_dim = env.observation_space.shape[0]      # число размерностей в состоянии
n_actions = env.action_space.n                  # число действий
max_episode_steps = env.spec.max_episode_steps  # макс. число шагов на эпизод
env.seed(42)
```

Мы также будем использовать встроенные обертки, которые позволяют периодически сохранять видеоролики, показывающие результативность агента:

```
from gym import wrappers

env = wrappers.Monitor(env,
directory=monitor_path.as_posix(),
video_callable=lambda count: count % video_freq == 0, force=True)
```

Во время работы на сервере или контейнере Docker без дисплея можно использовать библиотеку pyvirtualdisplay.

Гиперпараметры

Результативность агента довольно чувствительна к нескольким гиперпараметрам. Начнем с коэффициента дисконтирования и темпа усвоения:

```
gamma=.99,          # коэффициент дисконтирования
learning_rate=5e-5  # темп усвоения
```

Мы будем обновлять целевую сеть каждые 100 временных шагов, хранить до 1 млн прошлых эпизодов в памяти воспроизведения и отбирать мини-пакеты по 1024 элементов из памяти для тренировки агента:

```
tau=100             # частота обновления целевой сети
replay_capacity=int(1e6)
minibatch_size=1024
```

ε -Жадная политика начинается с чистого разведывания в $\varepsilon = 1$, линейного затухания до 0,05 за 20 тыс. временных шагов и экспоненциального затухания в последствии:

```
epsilon_start=1.0
epsilon_end=0.05
epsilon_linear_steps=2e4
epsilon_exp_decay=0.99
```

Вычислительный граф сети DDQN

Ключевые элементы вычислительного графа сети DDQN включают переменные-заполнители для последовательностей состояний, действий и вознаграждений:

```
# вход в Q-сеть
state = tf.placeholder(dtype=tf.float32, shape=[None, state_dim])

# вход в целевую сеть
next_state = tf.placeholder(dtype=tf.float32, shape=[None, state_dim])

# индексы действий (индексы выходов Q-сети)
action = tf.placeholder(dtype=tf.int32, shape=[None])

# вознаграждения для вычисления цели
reward = tf.placeholder(dtype=tf.float32, shape=[None])

# индикаторы для вычисления цели
not_done = tf.placeholder(dtype=tf.float32, shape=[None])
```

Функция `create_network` генерирует три плотных слоя, которые могут быть натренированы и/или повторно использованы в соответствии с требованиями *Q*-сети и ее более медленно двигающейся целевой сети:

```
def create_network(s, layers, trainable, reuse, n_actions=4):
    """Сгенерировать Q-сеть и целевую сеть с той же структурой"""
    for layer, units in enumerate(layers):
        x = tf.layers.dense(inputs=s if layer == 0 else x,
                            units=units,
                            activation=tf.nn.relu,
                            trainable=trainable,
                            reuse=reuse,
                            name='dense_{}'.format(layer))
    return tf.squeeze(tf.layers.dense(inputs=x,
                                      units=n_actions,
                                      trainable=trainable,
                                      reuse=reuse,
                                      name='output'))
```

Мы создадим две сети DQN для предсказаний значений q для текущего и следующего состояний, где мы будем держать веса для второй сети, которая фиксирована во время предсказания следующего состояния:

```
with tf.variable_scope('Q_Network'):
    # Q-сеть применительно к текущему наблюдению
    q_action_values = create_network(state,
                                     layers=layers,
                                     trainable=True,
                                     reuse=False)

    # Q-сеть применительно к следующему наблюдению
    next_q_action_values = tf.stop_gradient(create_network(next_state,
                                                           layers=layers,
                                                           trainable=False,
                                                           reuse=True))
```

Вдобавок к этому, мы создадим целевую сеть, которую будем обновлять через каждые `tau` периодов:

```
with tf.variable_scope('Target_Network', reuse=False):
    target_action_values = tf.stop_gradient(create_network(next_state,
                                                           layers=layers,
                                                           trainable=False,
                                                           reuse=False))
```

Цель y_i и предсказанное значение q вычисляются следующим образом:

```
# Вычисление целевого Q
targets = reward + not_done * gamma * tf.gather_nd(target_action_values,
                                          tf.stack((tf.range(minibatch_size),
                                                    tf.cast(tf.argmax(
                                                         next_q_action_values,
                                                         axis=1),
                                                         tf.int32)),
                                                         axis=1))

# Оценочные значения Q для (s,a) из воспроизведения опыта
predicted_q_value = tf.gather_nd(q_action_values,
                                 tf.stack((tf.range(minibatch_size), action), axis=1))
```

Наконец, функция потери `loss`, используемая для стохастического градиентного спуска, представлена среднеквадратической погрешностью/ошибкой (MSE) между целью и предсказанием:

```
losses = tf.squared_difference(targets, predicted_q_value)
loss = tf.reduce_mean(losses)
loss += tf.add_n([tf.nn.l2_loss(var) for var in q_network_variables if 'bias'
                                                        not in var.name]) * l2_reg * 0.5
```

Блокнот содержит полный тренировочный цикл, включая воспроизведение опыта, стохастический градиентный спуск и обновления медленной целевой сети, а также

обширный мониторинг. Веса нейронной сети для натренированных агентов сохраняются на диск и могут быть перезагружены для тестирования результативности.

Результативность

Предыдущие гиперпараметрические настройки позволяют агенту решать задачу окружающей среды в пределах 290 эпизодов. На рис. 21.6 слева показаны эпизодные вознаграждения и их скользящая средняя за 100 периодов. На рис. 21.6 справа показаны затухание разведывания и число шагов в эпизодах. Имеется отрезок примерно из 100 эпизодов, которые часто занимают 1000 временных шагов, пока агент снижает разведывание и учится летать, прежде чем начать довольно сносно приземляться.

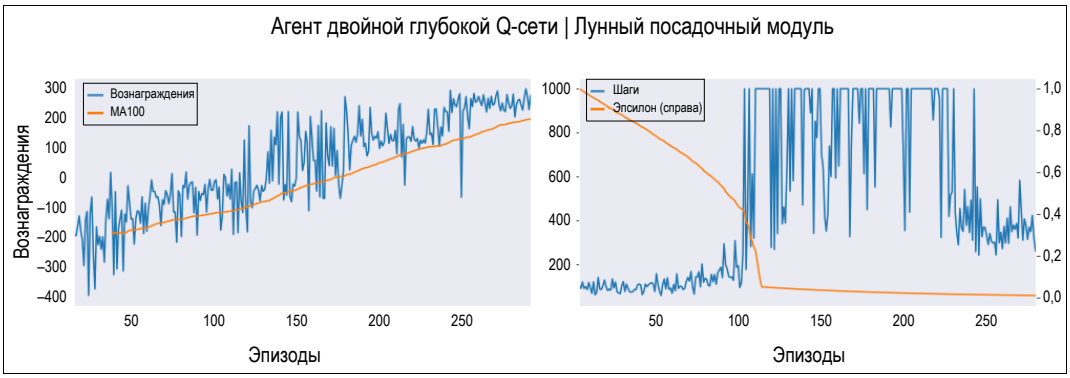

Рис. 21.6. Результативность агента двойной глубокой *Q*-сети

Подкрепляемое обучение для торговли на финансовых рынках

Для того чтобы натренировать торгового агента, нам нужно создать рыночную среду, которая предоставляет ценовую и другую информацию, предлагает действия, связанные с торговлей финансовыми активами, и отслеживает портфель, вознаграждая агента соответствующим образом.

Конструирование торговой среды в платформе OpenAI

Платформа "тренажерный зал ИИ" OpenAI Gym позволяет конструировать, регистрировать и потреблять среды, которые придерживаются ее архитектуры, как описано в ее документации (**https://github.com/openai/gym/tree/master/gym/envs#how-to-create-new-environments-for-gym**).

В сценарном файле trading_env.py реализован пример, иллюстрирующий создание класса, реализующего необходимые методы `step()` и `reset()`.

Торговая среда состоит из трех классов, которые взаимодействуют с целью обеспечения действий агента. Класс `DataSource` загружает временной ряд, генерирует несколько объектов и предоставляет агенту последние наблюдения на каждом временном шаге. Класс `TradingSimulator` отслеживает позиции, сделки и стоимость, а еще результативность. Он также реализует и записывает результаты эталонной стратегии покупки и удержания. Класс `TradingEnvironment` сам оркестрирует этот процесс.

Перед использованием клиентской среды, так же как мы использовали среду лунного посадочного модуля Lunar Lander, нам нужно ее зарегистрировать:

```
from gym.envs.registration import register

register(
    id='trading-v0',
    entry_point='trading_env:TradingEnvironment',
    max_episode_steps=1000)
```

Элементарная торговая игра

Для того чтобы натренировать агента, нам нужно организовать простую игру с ограниченным набором вариантов, относительно низкоразмерным состоянием и другими параметрами, которые можно легко модифицировать и расширить.

Более конкретно, среда выбирает временной ценовой ряд для одного тикера, используя случайную дату начала для имитации торгового периода, который по умолчанию содержит 252 дня или 1 год. Состояние содержит (шкалированную) цену и объем, а также несколько технических индикаторов, таких как процентильные ранги цены и объема, индекс относительной силы (RSI), а также 5- и 21-дневные финансовые возвраты. Агент может выбрать одно из трех действий:

- *покупка* — инвестировать капитал в акцию, заняв длинную позицию;
- *флэт* — остаться в наличности;
- *продажа* — шортить, заняв короткую позицию, равную сумме капитала.

Окружающая среда учитывает стоимость торговли, которая по умолчанию установлена в 10 базисных пунктов (bps)[3]. Она также вычитает временную стоимость в 1 bps за период. Она отслеживает стоимость портфеля агента, вычисленную по показателю *стоимости чистых активов* (net asset value, NAV), и сравнивает ее

[3] Базисный пункт (basis point, bps или bips во мн. ч.) — это единица измерения, используемая в финансах для описания процентного изменения стоимости или ставки финансового инструмента. Один базисный пункт эквивалентен сотой доле процента, 0,01%, или 0,0001 в десятичной форме. Схожим образом, дробный базисный пункт, такой как 1,5 базисных пункта, эквивалентен 0,015% или 0,00015 в десятичной форме. — См. **https://www.investopedia.com/ask/answers/what-basis-point-bps/**. — *Прим. перев.*

с рыночным портфелем (который торгует без трений с целью поднять планку для агента).

Результативность глубокого *Q*-обучения на фондовом рынке

Для реализации торговой игры мы используем того же самого агента сети DDQN и нейросетевую архитектуру, которые успешно научились ориентироваться в окружающей среде лунного посадочного модуля. Мы даем разведке продолжаться в течение 500 тыс. временных шагов (около 2000 одногодичных торговых периодов) с линейным затуханием ε до 0,1 и экспоненциальным затуханием с коэффициентом 0,9999 впоследствии.

Мы можем создать экземпляр окружающей среды, используя желаемые торговые издержки и тикер:

```
trading_environment = gym.make('trading-v0')
trading_environment.env.trading_cost_bps = 1e-3
trading_environment.env.time_cost_bps = 1e-4
trading_environment.env.ticker = 'AAPL'
trading_environment.seed(42)
```

На рис. 21.7 показаны скользящее среднее финансовых возвратов агента и рынка за 100 периодов слева и доля последних 100 периодов, когда агент превосходил рынок справа. В примере используются данные по акциям компании Apple (AAPL), для которых имеется около 9000 ежедневных наблюдений цены и объема. Тренировка остановилась после 14 тыс. торговых периодов, когда агент обыграл рынок 10 раз подряд.

Пример показывает, как результативность агента значительно улучшается, исследуя с более высокой скоростью в течение первых ~3000 периодов (т. е. лет) и при-

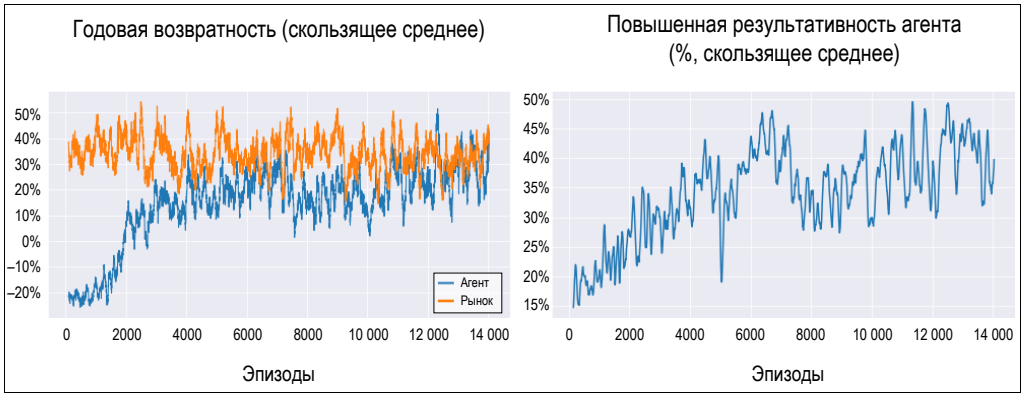

Рис. 21.7. Скользящее среднее финансовых возвратов агента и рынка за 100 периодов (слева) и доля последних 100 периодов, когда агент превосходил рынок (справа)

ближается к уровню, когда он превосходит рынок около 40% времени, несмотря на трансакционные издержки. В нескольких случаях он побивает рынок примерно в половине из 100 периодов.

Этот относительно простой агент использует ограниченную информацию, выходящую за рамки последних рыночных данных и вознаградительного сигнала, по сравнению с автоматически обучающимися моделями, которые мы рассмотрели в других главах этой книги. Тем не менее, он учится получать прибыль и приближаться к рынку (после тренировки на данных за несколько тысяч лет, что занимает около 30 минут).

Имейте в виду, что использование одной акции также намного увеличивает риск переподгонки к данным. Вы можете протестировать натренированного агента на новых данных, используя сохраненную модель (см. блокнот, посвященный лунному посадочному модулю).

В заключение этой главы мы продемонстрировали механику создания торговой среды на основе подкрепляемого обучения и поэкспериментировали с элементарным агентом, использующим небольшое число технических индикаторов. Вам следует попытаться расширить и среду, и агента так, чтобы выбирать из нескольких активов, определять их позиции и управлять рисками.

Резюме

В этой главе мы представили другой класс задач МО, которые сосредоточены на автоматизации принятия решений агентами, взаимодействующими с окружающей средой. Мы рассмотрели ключевые функциональные средства, необходимые для постановки задач подкрепляемого обучения, и различные методы их решения.

Мы стали свидетелями того, как формулировать и анализировать задачу подкрепляемого обучения в виде конечной задачи марковского процесса принятия решений и как вычислять решение с использованием цикла по ценностным значениям и цикла по политике. Затем мы перешли к более реалистичным ситуациям, когда переходные вероятности и вознаграждения агенту неизвестны, и увидели, как *Q*-обучение строится на ключевой рекурсивной связи, задаваемой уравнением оптимальности Беллмана в случае задачи марковского процесса принятия решений. Мы увидели, как решать задачи подкрепляемого обучения с помощью языка Python для простых задач марковского процесса принятия решений и более сложных окружающих сред с помощью *Q*-обучения.

Наконец, мы расширили область применения до непрерывных состояний и действий и применили алгоритм глубокого *Q*-обучения к более сложной окружающей среде спускаемого лунного модуля.