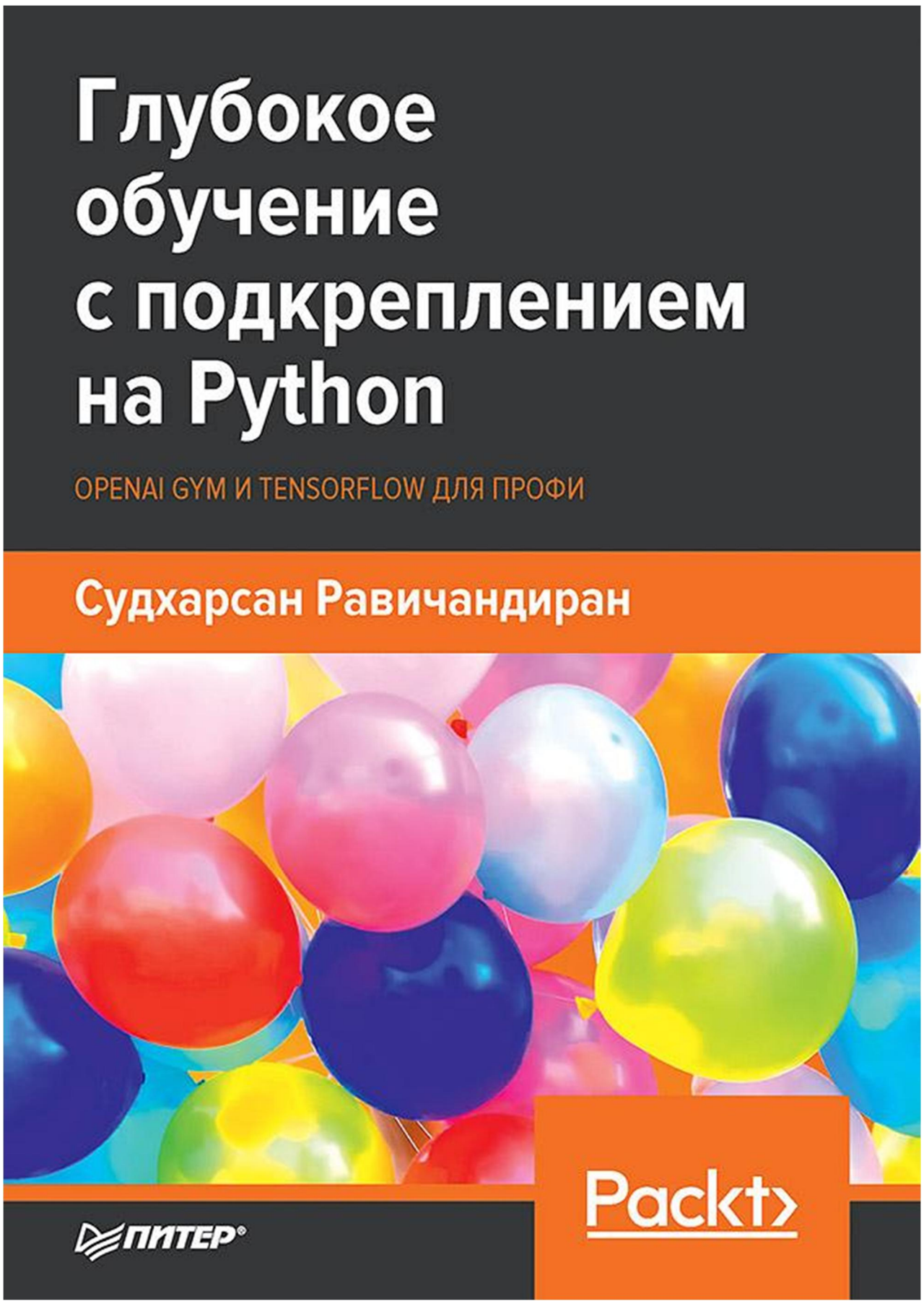


Судхарсан Равичандиран
Глубокое обучение с подкреплением на Python. OpenAI Gym и TensorFlow для профи
ПИТЕР
2019
Переводчик *Е. Матвеев*
Технический редактор *А. Руденко*
Литературный редактор *А. Руденко*
Художник *Л. Егорова*
Корректоры *Н. Викторова, М. Молчанова (Котова)*
Верстка *Л. Егорова*

Судхарсан Равичандиран
Глубокое обучение с подкреплением на Python. OpenAI Gym и TensorFlow для профи. — СПб.: Питер, 2019.

ISBN 978-5-4461-1251-7

Посвящается моим обожаемым родителям, брату Картикеяну и моему лучшему другу Нихилу Адитья

Об авторе

Судхарсан Равичандиран — специалист по обработке и анализу данных, горячий поклонник искусственного интеллекта и видеоблогер. Он получил степень бакалавра в области computer science в Университете Анны и занимается исследованиями практической реализации глубокого обучения и обучения с подкреплением, включая обработку естественных языков и компьютерное зрение. Ранее работал внештатным веб-дизайнером и разработчиком, участвовал в создании ряда сайтов, отмеченных наградами. В настоящее время принимает участие в проектах с открытым кодом и часто отвечает на вопросы на Stack Overflow (`www.stackoverflow.com`).

Хочу поблагодарить своих замечательных родителей и брата Картикеяна, которые вдохновляли и вселяли в меня уверенность в успешности этого проекта. Огромное спасибо моему лучшему (в буквальном смысле!) другу Нихилу Адитья (Nikhil Aditya), редактору Амрите и верной термокружке, которая сохраняла для меня горячий бодрящий кофе! Без их поддержки мне не удалось бы завершить эту книгу.

О научных редакторах

Суджит Пал (Sujit Pal) — руководитель технических исследований в Elsevier Labs, группы разработки новейших технологий компании Reed-Elsevier Group. Занимается исследованиями в области семантического поиска, обработки естественных языков, машинного и глубокого обучения. В Elsevier работал над несколькими инициативными проектами, включая оценку и совершенствование качества поиска, классификацию изображений и выявление дубликатов, аннотацию и разработку антологий медицинских и научных текстов. Он написал книгу о глубоком обучении совместно с Антонио Галли (Antonio Gulli) и пишет о технологиях в своем блоге *Salmon Run* (`http://sujitpal.blogspot.com/`).

Сурьядипан Рамамурти(Suriyadeepan Ramamoorthy) — исследователь искусственного интеллекта и инженер из AI researcher and engineer в Пондичерри (Индия). Основная тематика его работ — понимание естественных языков и формирование рассуждений. Он активно пишет в блоге, посвященном глубокому обучению.

В SAAMA Technologies он применяет расширенные методы глубокого обучения для анализа биомедицинских текстов. Являясь ярым сторонником свободно распространяемого ПО, активно участвует в

проектах по его разработке в сообществе FSFTN. Также интересуется коллективными сетями, визуализацией данных и творческим программированием.

Предисловие

Обучение с подкреплением — бурно развивающаяся дисциплина машинного обучения (МО), которая приближает нас к созданию истинного искусственного интеллекта. Это доступное руководство объясняет все с самого начала на подробных примерах, написанных на Python.

Для кого написана эта книга

Эта книга предназначена для разработчиков МО и энтузиастов глубокого обучения, интересующихся искусственным интеллектом и желающих освоить метод обучения с подкреплением. Прочитайте эту книгу и станьте экспертом в области обучения с подкреплением, реализуя практические примеры в работе или вне ее. Знания в области линейной алгебры, математического анализа и языка программирования Python помогут вам понять логику изложения материала.

Что в книге

Глава 1, *«Введение в обучение с подкреплением»*, поможет понять, что такое обучение с подкреплением и как оно работает. Вы узнаете о его элементах — агентах, средах, политиках и моделях, а также о различных типах сред, платформ и библиотек, используемых в нем. В завершающей части главы рассмотрены некоторые примеры применения обучения с подкреплением.

В главе 2, *«Знакомство с OpenAI и TensorFlow»*, описана настройка машины для различных задач обучения с подкреплением. Вы узнаете, как подготовить машину и установить на ней Anaconda, Docker, OpenAI Gym, Universe и TensorFlow. Описаны моделирование агентов в OpenAI Gym и построение бота для видеоигры. Изложены основы TensorFlow и порядок использования TensorBoard для визуализации.

Глава 3, *«Марковский процесс принятия решений и динамическое программирование»*, объясняет, что собой представляют марковские цепи и процессы. Вы увидите, как задачи обучения с подкреплением могут моделироваться в форме марковских процессов принятия решений. Также рассмотрены некоторые фундаментальные концепции: функции ценности, *Q*-функции и уравнение Беллмана. Вы узнаете, что такое динамическое программирование и как решается задача о замерзшем озере с использованием итераций по ценности и политикам.

В главе 4, *«Методы Монте-Карло в играх»*, объяснены методы Монте-Карло и разновидности методов прогнозирования Монте-Карло, в том числе при первом посещении и при каждом посещении. Вы узнаете,

как использовать эти методы для игры в блек-джек. Также рассмотрены методы управления Монте-Карло с привязкой к политике и без нее.

Глава 5, *«Обучение на основе временных различий»*, посвящена обучению на основе временных различий (TD), TD-прогнозированию и управляющим методам TD с политикой и без, таким как *Q*-обучение и SARSA. Вы узнаете, как задача о такси решается средствами *Q*-обучения и SARSA.

В главе 6, *«Задача о многоруком бандите»*, рассмотрена одна из классических задач обучения с подкреплением — задача о многоруком бандите (MAB) или *k*-руком бандите. Здесь показано, как решить эту задачу с применением различных стратегий исследования, включая эпсилон-жадную стратегию, softmax-исследование, UCB и выборку Томпсона. В завершающей части этой главы продемонстрировано применение задачи MAB для показа правильного рекламного баннера.

В главе 7, *«Основы глубокого обучения»*, изложены фундаментальные концепции глубокого обучения. Дано понятие нейросети и представлены ее типы, включая RNN, LSTM и CNN. Для демонстрации материала построены сети, решающие такие задачи, как генерирование текстов песен и классификация модных товаров.

Глава 8, *«Игры Atari с использованием Deep Q Network»*, посвящена одному из самых популярных алгоритмов обучения с подкреплением — глубокой *Q*-сети (DQN). Вы узнаете о различных компонентах DQN и увидите, как построить агента, играющего в игры Atari с использованием DQN. В завершение рассмотрены некоторые улучшения архитектуры DQN, такие как двойные и дуэльные DQN.

Глава 9, *«Игра Doom в глубокой рекуррентной Q-сети»*, объясняет принципы работы глубоких рекуррентных *Q*-сетей (DRQN) и их отличия от DQN. Вы узнаете, как построить агента для игры Doom на базе DRQN. В завершение главы рассмотрены глубокие рекуррентные *Q*-сети с вниманием, добавляющие механизм внимания в архитектуру DRQN.

Глава 10, *«Асинхронная преимущественная сеть “актор-критик”»*, объясняет принципы работы асинхронных преимущественных сетей «актор-критик» (A3C). Рассмотрена архитектура A3C, благодаря которой вы узнаете, как на базе A3C построить агента для подъема на гору.

В главе 11, *«Градиенты политик и оптимизация»*, показано, как градиенты политик помогают выбрать правильную политику без обязательного использования *Q*-функции. Также рассмотрены глубокий детерминированный метод градиента политики и современные методы оптимизации, такие как оптимизация политики доверительной области и оптимизация ближайшей политики.

Глава 12, *«”Автогонки” с использованием DQN»*, содержит подробное описание построения агента для победы в игре «Автогонки» на базе дуэльной DQN.

Глава 13, *«Последние достижения и следующие шаги»*, содержит информацию о различных достижениях в области обучения с

подкреплением: агентах, дополненных воображением; обучении на человеческих предпочтениях; глубоком *Q*-обучении на примере демонстраций и ретроспективном воспроизведении опыта. В завершение представлены различные типы методов обучения с подкреплением, такие как иерархическое и инвертированное обучения с подкреплением.

Необходимое программное обеспечение

Для этой книги вам понадобятся следующие программные продукты:

- Anaconda.
- Python.
- Любой веб-браузер.
- Docker.

Загрузка файлов с примерами кода

Файлы с примерами кода для этой книги можно загрузить с сайта `www.packtpub.com`. Если вы приобрели эту книгу не на сайте, обратитесь на страницу `www.packtpub.com/support` и зарегистрируйтесь — файлы будут отправлены вам по электронной почте.

Также можно загрузить файлы с кодом на сайте:

1. Введите свои регистрационные данные или зарегистрируйтесь на `www.packtpub.com`.
2. Выберите вкладку `SUPPORT`.
3. Щелкните на ссылке `CodeDownloads & Errata`.
4. Введите название книги в поле `Search` и выполните инструкции, появившиеся на экране.

После того как файл будет загружен, распакуйте его с помощью новейшей версии одной из следующих программ:

- WinRAR/7-Zip для Windows;
- Zipeg/iZip/UnRarX для Mac;
- 7-Zip/PeaZip для Linux.

Архив примеров этой книги также размещен на GitHub по адресу `https://github.com/PacktPublishing/Hands-On-Reinforcement-Learning-with-Python`. Если код будет обновлен, изменения появятся в существующем репозитории GitHub.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу `comp@piter.com` (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства `www.piter.com` вы найдете подробную информацию о наших книгах.

1. Введение в обучение с подкреплением

Обучение с подкреплением (**RL**, reinforcement learning) — область машинного обучения, в которой обучение осуществляется посредством взаимодействия с окружающей средой. Это целенаправленное обучение, в котором обучаемый не получает информации о том, какие действия следует выполнять, вместо этого он узнает о последствиях своих действий. Сейчас область обучения с подкреплением стремительно развивается, в ней появляется множество разнообразных алгоритмов, она становится одной из самых активных областей исследования в сфере **искусственного интеллекта** (**AI**,artificial intelligence).

В этой главе рассматриваются следующие темы:

- Фундаментальные концепции RL.
- Алгоритм RL.
- Интерфейс агента со средой.
- Типы сред RL.
- Платформы RL.
- Практическое применение RL.

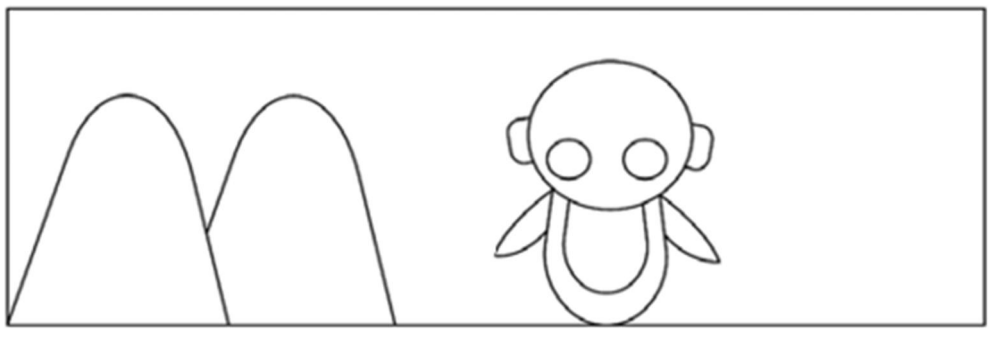
Что такое RL?

Представьте, что вы учите собаку ловить мячик на лету. Вы не сможете на словах объяснить собаке, что она должна поймать мячик; вместо этого вы просто кидаете мячик и каждый раз, когда собака ловит его, даете ей кусочек сахара. Если собака не поймала мячик, вы не даете ей сахар. Вскоре собака понимает, при каких действиях она получает сахар, и начинает повторять эти действия.

Аналогичным образом в среде RL вы не учите агента, что и как он должен делать, вместо этого вы даете агенту награду за каждое выполненное действие. Награда может быть положительной или отрицательной. Тогда агент начинает выполнять действия, при которых он получает положительную награду. Таким образом, обучение превращается в процесс проб и ошибок. В нашей аналогии собака представляет агента;, сахар, полученный за пойманный мячик, — положительную награду, а отсутствие сахара — отрицательную.

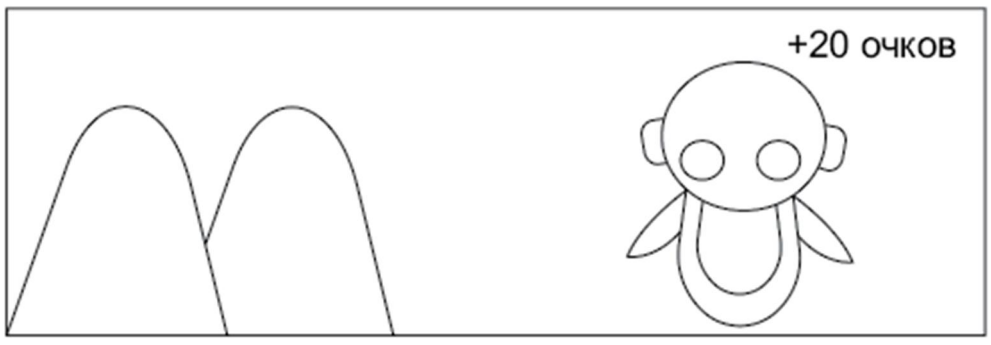
Также возможны отсроченные награды. Например, награда может выдаваться не на каждом шаге, а только после выполнения задачи. В некоторых случаях награда выдается на каждом шаге, чтобы узнать, не совершает ли обучаемый ошибки.

Представьте, что вы хотите научить робота ходить так, чтобы он не застревал, натолкнувшись на горку; при этом вы не запрещаете роботу идти в направлении горки:



Вместо этого, если робот сталкивается с горкой и застревает, вы отнимаете у него 10 очков. Робот понимает, что столкновение с горкой приводит к отрицательному результату, и больше не пойдет в этом направлении:

Когда робот идет в правильном направлении, вы даете ему 20 очков. Робот понимает, какой путь правильный, и пытается максимизировать награду, перемещаясь в правильном направлении:



Агент RL может исследовать различные действия, которые обеспечивают положительную награду, или же **эксплуатировать**(использовать) предыдущее действие, которое привело к положительной награде. Если агент RL исследует различные действия, существует высокая вероятность того, что агент получит отрицательную награду, так как не все действия будут лучшими. Если агент RL эксплуатирует только известное лучшее действие, существует высокая вероятность упустить лучшее действие, которое может принести более высокую награду. Всегда существует компромисс между исследованием и **эксплуатацией**. Невозможно заниматься исследованием и эксплуатацией одновременно. Дилемма исследования/эксплуатации будет более подробно рассмотрена в следующих главах.

Алгоритм RL

Типичный алгоритм RL состоит из следующих этапов:

1. Агент взаимодействует со средой, выполняя действие.
2. Агент выполняет действие и переходит из одного состояния в другое.
3. Агент получает награду на основании выполненного действия.
4. В зависимости от награды агент понимает, было действие хорошим или плохим.
5. Если действие было хорошим, то есть если агент получил положительную награду, то агент предпочитает выполнить это действие еще раз; в противном случае агент пытается выполнить другое действие,

приводящее к положительной награде. Таким образом, по сути, происходит процесс обучения методом проб и ошибок.

Чем RL отличается от других парадигм машинного обучения

При контролируемом обучении машина (агент) учится на тренировочных данных, имеющих помеченные наборы входных и выходных данных. Модель экстраполирует и обобщает полученную информацию, чтобы ее можно было применить к данным, которыми она не располагает. Также существует внешний контролер, располагающий полной базой знаний о среде и наблюдающий за тем, как агент выполняет свою задачу.

Вспомните только что рассмотренную аналогию с собакой. Чтобы научить собаку ловить мячик при контролируемом обучении, вы должны явно приказать ей повернуть налево, переместиться направо, пройти вперед пять шагов, поймать мячик и т.д. В RL вместо этого вы просто кидаете мячик и каждый раз, когда собака его ловит, — даете ей сахар (награду). Собака учится ловить мячик, потому что при этом она получает сахар.

При неконтролируемом обучении модели передаются тренировочные данные, которые содержат только входной набор; модель учится определять скрытые закономерности во вводе. Существует распространенное заблуждение, что RL является разновидностью неконтролируемого обучения, но это не так. При неконтролируемом обучении модель изучает скрытую структуру, тогда как в RL модель учится на максимизации наград. Допустим, вы хотите порекомендовать пользователю новые фильмы. Неконтролируемое обучение анализирует похожие фильмы, просмотренные пользователем, и выдает предложения, а RL постоянно получает обратную связь от пользователя, выясняет его предпочтения, строит на их основе базу знаний и предлагает новый фильм.

Также существует другая разновидность обучения — обучение с частичным контролем: по сути, это сочетание контролируемого и неконтролируемого обучения. Оно основано на оценке функции как по помеченным, так и по непомеченным данным, тогда как RL является взаимодействием между агентом и его средой. Таким образом, RL полностью отличается от всех остальных парадигм машинного обучения.

Элементы RL

Ниже перечислены основные элементы RL.

Агент

Агент (agent) — программа, способная принимать осмысленные решения; фактически она играет роль обучаемого в RL. Агенты выполняют действия, контактируя со средой, и получают награды в

зависимости от своих действий — например, перемещая персонажа в видеоигре Super Mario.

Функция политики

Политика (policy) определяет поведение агента в среде. Способ выбора агентом действия, которое он будет выполнять, зависит от политики. Допустим, вы хотите добраться из дома на работу. Существует множество возможных вариантов маршрута; одни пути короткие, другие длинные. Эти пути называются «политиками», потому что они представляют собой способ выполнения действия для достижения цели. Политика часто обозначается символом π и может быть реализована таблицей соответствия или сложным процессом поиска.

Функция ценности

Функция ценности (value function) определяет, насколько хорошо для агента *A* пребывание в конкретном состоянии. Она зависит от политики и часто обозначается *v* (*s*). Ее значение равно ожидаемому суммарному результату, получаемому агентом, начиная с исходного состояния. Функций ценности может быть несколько; «оптимальной функцией ценности» называется та, которая обеспечивает наибольшую ценность по всем состояниям по сравнению с другими функциями ценности. Аналогичным образом «оптимальной политикой» называется политика, имеющая оптимальную функцию ценности.

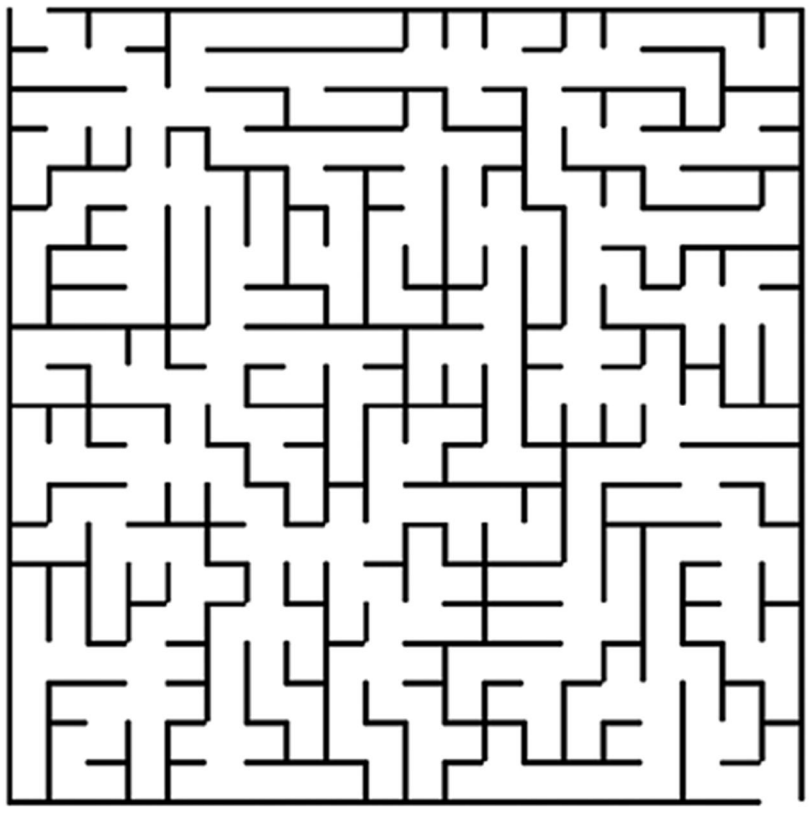
Модель

Модель (model) является представлением среды с точки зрения агента. Обучение делится на два типа: с моделью и без модели. В обучении с моделью агент эксплуатирует ранее усвоенную информацию для выполнения задачи, а при обучении без модели агент просто полагается на метод проб и ошибок для выполнения правильного действия. Предположим, вы хотите быстрее добираться из дома на работу. В обучении с моделью вы просто используете уже имеющуюся информацию (карту), а в обучении без модели вы не пользуетесь имеющейся информацией, пробуете разные пути и выбираете самый быстрый.

Интерфейс агента со средой

Агенты — программы, выполняющие действие A_t в момент времени t для перехода из состояния S_t в другое состояние S_{t+1}. На основании своих действий агенты получают числовой результат R от среды. В конечном итоге задачей RL является поиск оптимальных действий, повышающих числовой результат:

Рассмотрим концепцию RL на примере поиска пути в лабиринте:



Цель игры — добраться до выхода и не заблудиться в лабиринте. Общая схема выглядит так:

• агент — тот, кто перемещается по лабиринту (то есть наша программа/алгоритм RL);

• среда — лабиринт;

• состояние — позиция в лабиринте, на которой в данный момент находится агент;

• агент выполняет действие, перемещаясь из одного состояния в другое;

• агент получает положительный результат, если при выполнении действия он не наталкивается на препятствие, и отрицательный результат — если при своем действии он наталкивается на препятствие и не может добраться до конечной точки;

• цель — пройти лабиринт и добраться до выхода.

Типы сред RL

Все, с чем взаимодействует агент, называется средой. Можно сказать, что среда — это внешний мир. Она включает все, что является внешним по отношению к агенту. Существует несколько разновидностей сред, которые описаны в следующих разделах.

Детерминированная среда

Среда называется *детерминированной*, если последствия действий полностью известны по текущему состоянию. Например, в шахматной партии мы знаем точный результат хода каждой фигурой.

Стохастическая среда

Среда называется *стохастической*, если результат не может быть определен по текущему состоянию из-за повышенной неопределенности. Например, мы никогда не знаем, какое число выпадет на кубике.

Среда с полной информацией

В *среде с полной информацией* агент может определить состояние системы в любой момент. Например, в шахматной партии состояние системы (то есть положение всех фигур на доске) известно в любой момент времени, так что игрок может принять оптимальное решение.

Среда с неполной информацией

В *среде с неполной информацией* агент не может определить состояние системы в любой момент времени. Например, при игре в покер мы понятия не имеем, какие карты получил соперник.

Дискретная среда

Если существует конечный набор доступных действий для перехода из одного состояния в другое, среда называется *дискретной*. Например, в шахматной партии набор ходов конечен.

Непрерывная среда

Если существует бесконечный набор доступных действий для перехода из одного состояния в другое, среда называется *непрерывной*. Например, для перемещения из дома на работу существует множество потенциальных маршрутов.

Эпизодические и неэпизодические среды

Эпизодические среды также называются **непоследовательными**. В таких средах текущее действие агента не влияет на будущее действие, тогда как в *неэпизодических* (или **последовательных**) средах, напротив, будущее действие зависит от текущего. А именно, в эпизодической среде агент выполняет независимые задачи, тогда как в неэпизодической среде все действия агента связаны между собой.

Одноагентные и многоагентные среды

Как подсказывает само название, в *одноагентной* среде используется только один агент, а в *многоагентной*среде агентов несколько. Многоагентные среды широко применяются для решения сложных задач.

Разные агенты действуют в совершенно разных средах, взаимодействуя друг с другом. Многоагентные среды в основном являются стохастическими, так как они обладают повышенным уровнем неопределенности.

Платформы RL

Платформы RL используются для моделирования, построения и экспериментирования с алгоритмами RL в среде. Существует много разных платформ RL, описанных в следующих разделах.

OpenAI Gym и Universe

OpenAI Gym — инструментарий для построения, оценки и сравнения алгоритмов RL. Он совместим с алгоритмами, написанными в таких фреймворках, как TensorFlow, Theano, Keras и т.д. OpenAI Gym прост и доступен для понимания, он не делает никаких предположений относительно структуры агента и предоставляет интерфейс ко всем задачам RL.

OpenAI Universe представляет собой расширение OpenAI Gym. Этот инструментарий предоставляет возможность проводить тренировку и оценку агентов в широком спектре сложных сред, от простых до сред реального времени. Он обладает неограниченным доступом ко многим игровым средам. При использовании Universe любая программа может быть преобразована в среду Gym без доступа к внутренней реализации программы, ее исходному коду или API, так как работа Universe основана на автоматическом запуске программы за виртуальной сетью, вычисляющей состояние удаленного рабочего стола.

DeepMind Lab

DeepMind Lab — еще одна замечательная платформа для AI-исследований с использованием агентов. DeepMind Lab предоставляет богатую моделируемую среду, которая используется в качестве лабораторной установки для запуска разных алгоритмов RL. Платформа отличается широкими возможностями настройки и расширения. Визуальное оформление в научно-фантастическом стиле выглядит очень эффектно и реалистично.

RL-Glue

RL-Glue предоставляет интерфейс для соединения агентов, сред и программ, даже если они написаны на разных языках программирования. Вы получаете возможность передавать своих агентов и среды другим разработчикам для дальнейшего развития вашей работы. Такая совместимость существенно расширяет возможности повторного использования разработок.

Проект Malmo

Проект Malmo — еще одна платформа для экспериментов с AI от компании Microsoft, построенная на базе Minecraft, интегрируемая со сложной средой и обладающая хорошей гибкостью для ее настройки. Также проект Malmo поддерживает разгон (overclocking), позволяющий программистам воспроизводить сценарии быстрее, чем в стандартном Minecraft. В отличие от OpenAI Universe, в настоящее время Malmo поддерживает только игровые среды Minecraft.

ViZDoom

ViZDoom, как подсказывает название, представляет собой AI-платформу на базе Doom. ViZDoom предоставляет поддержку разных агентов и конкурентную среду для тестирования агента. Однако ViZDoom поддерживает только игровую среду Doom: внеэкранную прорисовку, режимы однопользовательской и многопользовательской игры.

Практическое применение RL

Благодаря всем современным исследованиям и достижениям, RL быстро проникает в повседневное применение в разных областях, от компьютерных игр до автоматического управления автомобилем. Некоторые примеры применения RL перечислены ниже.

Образование

Многие платформы дистанционного образования используют RL для того, чтобы предоставить персонализированный контент каждому без исключения студенту. Одни студенты лучше воспринимают видеоинформацию, другие предпочитают практические задания, третьи лучше учатся по конспектам. RL используется для формирования образовательного контента, персонализированного для каждого студента в соответствии с его особенностями восприятия, причем настройка может динамически изменяться в зависимости от поведения пользователя.

Медицина и здравоохранение

RL находит множество применений в медицине и здравоохранении, среди которых можно выделить индивидуально подобранную терапию, диагностику на основании медицинских снимков, формирование терапевтических стратегий при принятии клинических решений, сегментацию медицинских снимков и т.д.

Производство

В производстве роботы с искусственным интеллектом используются для размещения объектов в нужном положении. Если робот устанавливает объект в правильном (или неправильном) положении, он запоминает

объект и обучается для выполнения своей работы с большей точностью. Применение интеллектуальных роботов сокращает затраты на рабочую силу и повышает производительность труда.

Управление ресурсами

RL широко используется при управлении ресурсами — одном из важнейших аспектов коммерческой деятельности. К числу реализуемых в этой области процессов относятся: управление цепочкой поставок, прогнозирование спроса и некоторые складские операции (например, размещение продуктов на складах для эффективного использования складского пространства). Аналитики Google из DeepMind разработали алгоритмы RL для эффективного сокращения энергопотребления в своем собственном компьютерном центре.

Финансы

RL широко используется для управления портфелями ценных бумаг — процесса постоянного перераспределения средств между различными финансовыми инструментами, а также прогнозирования и торговли на рынке коммерческих финансовых сделок. Банк JP Morgan успешно использовал RL для улучшения результативности сделок по крупным заказам.

Обработка естественного языка и машинное распознавание образов

Глубокое обучение с подкреплением (DRL, deep reinforcement learning), объединяющее мощь глубокого обучения и RL, добилось значительных успехов в областях обработки естественного языка (NLP, natural language processing) и машинного распознавания образов (CV, computer vision). DRL используется для обобщения текста, извлечения информации, машинного перевода и распознавания образов, оно обеспечивает большую точность по сравнению с современными системами.

Итоги

В этой главе были изложены основы и некоторые ключевые концепции обучения с подкреплением (RL). Мы рассмотрели различные элементы RL и типы его сред. Также были описаны существующие платформы RL и возможности применения RL в разных областях.

В **главе 2**, *«Знакомство с OpenAI и TensorFlow»*, вы узнаете, как установить OpenAI и TensorFlow. Далее вы научитесь моделировать среды и готовить агентов для обучения в среде.

Вопросы

1. Что такое обучение с подкреплением (RL)?
2. Чем RL отличается от других парадигм машинного обучения?
3. Что такое агенты и как они обучаются?

4. Чем функция политики отличается от функции ценности?
5. Чем отличается обучение с моделью от обучения без модели?
6. Какие существуют типы сред в RL?
7. Чем OpenAI Universe отличается от других платформ RL?
8. Перечислите некоторые примеры применения RL.

Дополнительные источники

Обзор RL: `https://www.cs.ubc.ca/~murphyk/Bayes/pomdp.html`.

2. Знакомство с OpenAI и TensorFlow

OpenAI — некоммерческая исследовательская компания в области **искусственного интеллекта**(**AI**), основанная Илоном Маском (Elon Musk) и Сэмом Альтманом (Sam Altman) для проведения работ по построению обобщенного искусственного интеллекта. Деятельность OpenAI спонсируется отраслевыми лидерами и самыми известными компаниями. Продукт OpenAI существует в двух разновидностях, Gym и Universe, при помощи которых можно имитировать реалистичные среды, строить алгоритмы **обучения с подкреплением** (**RL**) и тестировать агентов в этих средах.

TensorFlow — библиотека машинного обучения с открытым кодом от компании Google, широко используемая для обработки числовых данных. В следующих главах OpenAI и TensorFlow будут использованы для построения и оценки мощных алгоритмов RL.

В этой главе рассмотрены следующие темы:

- Подготовка системы: установка Anaconda, Docker, OpenAI Gym / Universe и TensorFlow.
- Моделирование среды с использованием OpenAI Gym и Universe.
- Обучение робота ходьбе.
- Построение бота для видеоигры.
- Основные понятия TensorFlow.
- Использование TensorBoard.

Подготовка системы

Установка OpenAI — нетривиальное дело; необходимо правильно выполнить целый ряд действий для настройки системы и запуска. Давайте посмотрим, как настроить машину и установить на ней OpenAI Gym и Universe.

Установка Anaconda

Во всех примерах книги используется версия Python, которая называется Anaconda. Anaconda — дистрибутив Python с открытым кодом, который широко используется для научных вычислений и обработки больших объемов данных. Anaconda предоставляет отличную систему управления

пакетами с поддержкой Windows, MacOS и Linux. Anaconda устанавливается во многих популярных пакетах, предназначенных для научных вычислений: NumPy, SciPy и т.д.

Чтобы загрузить Anaconda, откройте страницу `https://www.anaconda.com/download/`. На странице представлены варианты загрузки Anaconda для разных платформ.

Если вы используете Windows или Mac, то можете загрузить графическую программу установки для своей архитектуры и выполнить установку из этой программы.

Пользователи Linux выполняют следующие действия:

1. Откройте окно терминала и введите следующую команду для загрузки Anaconda:

```
wget
https://repo.continuum.io/archive/Anaconda3-
5.0.1-Linux-x86_64.sh
```

2. После завершения загрузки установите Anaconda следующей командой:

```
bash Anaconda3-5.0.1-Linux-x86_64.sh
```

После успешной установки Anaconda необходимо создать новую среду Anaconda, которая, по сути, является виртуальной средой. Для чего нужна виртуальная среда? Допустим, вы работаете над проектом *A*, который использует NumPy версии 1.14, и над проектом *B*, использующим NumPy версии 1.13. Таким образом, для работы над проектом *B* нужно либо понизить версию NumPy, либо переустановить Anaconda. В каждом проекте используются разные библиотеки с разными версиями, которые не подходят для других проектов. Вместо того чтобы понижать или повышать версии или переустанавливать Anaconda для каждого нового проекта, мы используем виртуальную среду. Для текущего проекта создается изолированное окружение, так что каждый проект может иметь собственные зависимости и не влиять на другие проекты. Следующая команда создает такую среду и присваивает ей имя `universe`:

```
conda create --name universe python=3.6 anaconda
```

Для активации среды используется следующая команда:

```
source activate universe
```

Установка Docker

После Anaconda необходимо установить Docker. Docker упрощает развертывание приложений для реальной эксплуатации. Допустим, вы построили на своем компьютере `localhost`-приложение, которое использует TensorFlow и другие библиотеки, и теперь хотите развернуть свои приложения на сервере. Все эти зависимости также должны быть установлены на сервере. Docker позволяет упаковать приложение вместе с зависимостями (полученный пакет называется *контейнером*), благодаря чему приложение можно просто запускать на сервере в форме

контейнера без использования каких-либо внешних зависимостей. OpenAI не поддерживает Windows, поэтому для установки OpenAI в Windows необходимо использовать Docker. Кроме того, поддержка Docker необходима большей части OpenAI Universe для моделирования среды. Теперь посмотрим, как происходит установка Docker.

Чтобы загрузить Docker, откройте страницу `https://docs.docker.com/`; на ней вы найдете ссылку `Get Docker`. Если выбрать ее, откроется набор вариантов для разных операционных систем. Если вы используете Windows или Mac, то можете загрузить графическую программу установки Docker и выполнить установку из этой программы.

Пользователи Linux выполняют следующие действия:

1. Откройте окно терминала и введите следующую команду:

```
sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    software-properties-common
```

2. Затем введите команду:

```
curl -fsSL
https://download.docker.com/linux/ubuntu/gpg |
sudo apt-key add-
```

3. После этого введите команду:

```
sudo add-apt-repository \
   "deb [arch=amd64]
https://download.docker.com/linux/ubuntu \
   $(lsb_release -cs) \
   stable"
```

4. И наконец, введите команды:

```
sudo apt-get update
sudo apt-get install docker-ce
```

Для использования Docker необходимо быть участником группы пользователей Docker. Чтобы присоединиться к группе пользователей Docker, введите следующие команды:

```
sudo adduser $(whoami) docker
newgrp docker
groups
```

Чтобы проверить успешность установки Docker, запустите встроенную программу `hello-world`:

```
sudo service docker start
sudo docker run hello-world
```

Чтобы вам не приходилось каждый раз использовать `sudo` для работы с Docker, введите следующие команды:

```
sudo groupadd docker
```

```
sudo usermod -aG docker $USER
sudo reboot
```

Установка OpenAI Gym и Universe

Перейдем к установке OpenAI Gym и Universe. Перед этим необходимо установить несколько зависимостей. Для начала активируйте только что созданную среду `conda` следующей командой:

```
source activate universe
```

Затем установите следующие зависимости:

```
sudo apt-get update
sudo apt-get install golang libcupti-dev libjpeg-turbo8-dev make tmux htop
  chromium-browser git cmake zlib1g-dev libjpeg-dev xvfb libav-tools xorg-dev
  python-opengl libboost-all-dev libsdl2-dev swig

conda install pip six libgcc swig
conda install opencv
```

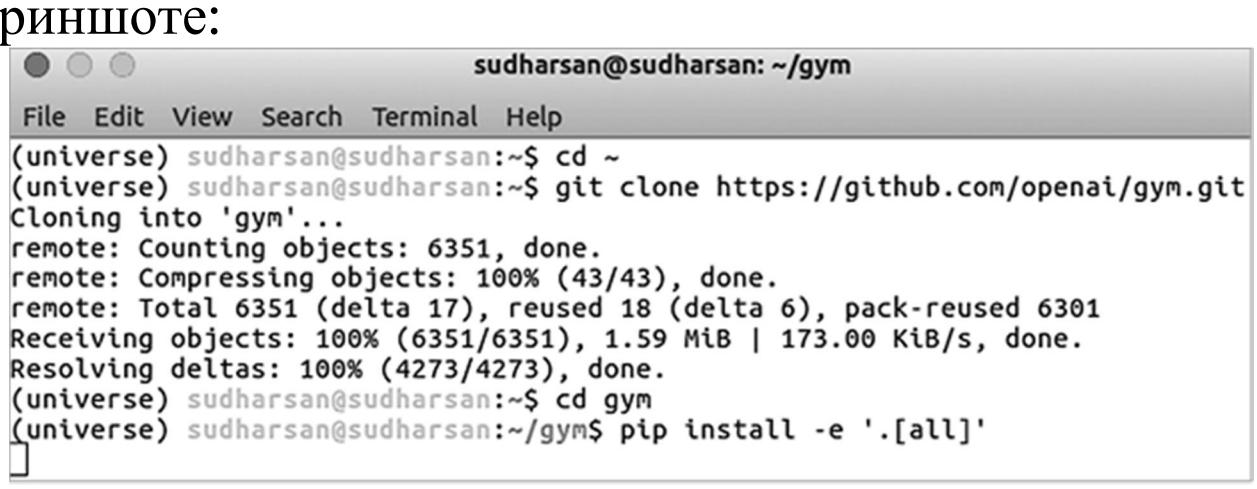
В этой книге будет использоваться `gym` версии 0.7.0, так что вы можете напрямую установить `gym` следующей командой:

```
pip install gym==0.7.0
```

Также можно клонировать репозиторий `gym` и установить новейшую версию следующими командами:

```
cd ~
git clone https://github.com/openai/gym.git
cd gym
pip install -e '.[all]'
```

Приведенные команды загружают репозиторий `gym` и устанавливают `gym` в форме пакета, как показано на следующем скриншоте:



Типичные ошибки

Существует достаточно высокая вероятность того, что при установке `gym` вы столкнетесь с типичными ошибками; некоторые из них перечислены ниже. Если вы получите такую ошибку, просто выполните следующие команды и попробуйте провести установку заново:

- Failed building wheel forpachi-py или Failed building wheelfor pachi-py atari-py:

```
sudo apt-get update
sudo apt-get install xvfb libav-tools xorg-dev libsdl2-dev swig
cmake
```

- Failed building wheel formujoco-py:

```
git clone https://github.com/openai/mujoco-py.git
cd mujoco-py
sudo apt-get update
sudo apt-get install libgl1-mesa-dev libgl1-mesa-glx libosmesa6-dev
python3-pip python3-numpy python3-scipy
pip3 install -r requirements.txt
sudo python3 setup.py install
```

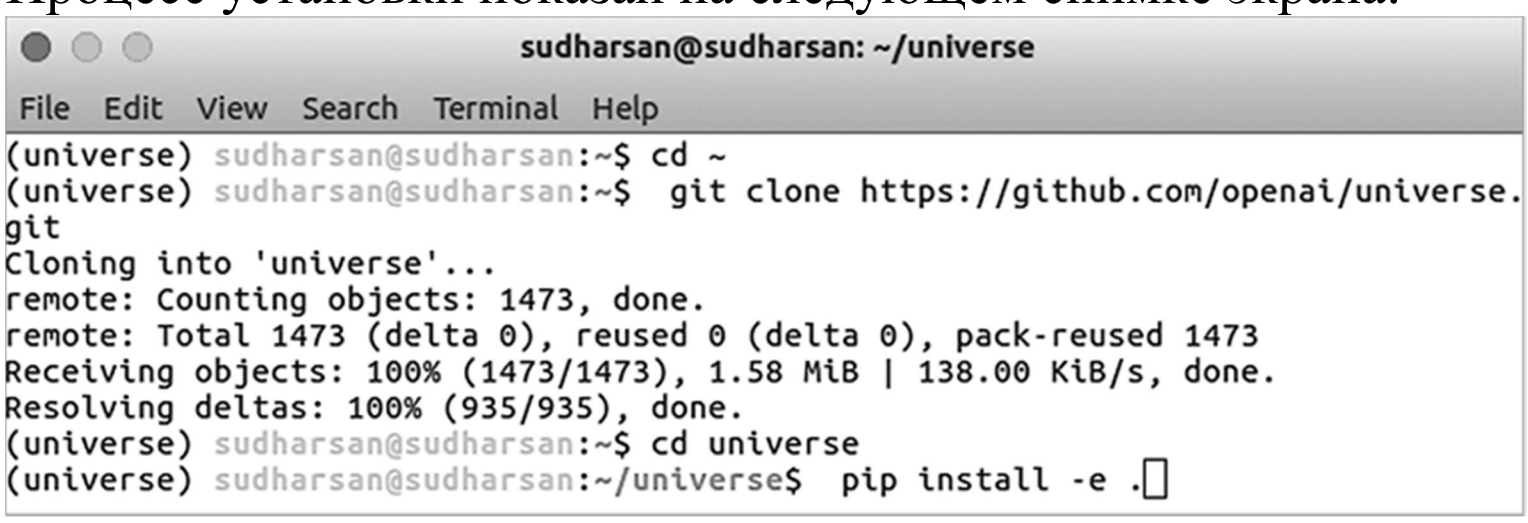
- Error: command ‘gcc’ failedwith exit status 1:

```
sudo apt-get update
sudo apt-get install python-dev
sudo apt-get install libevent-dev
```

Аналогичным образом можно установить OpenAI Universe загрузкой репозитория `universe` и установкой `universe` в форме пакета:

```
cd ~
git clone https://github.com/openai/universe.git
cd universe
pip install -e .
```

Процесс установки показан на следующем снимке экрана:



Как было сказано выше, Open AI Universe необходима установка Docker, поскольку большинство сред Universe выполняется внутри контейнеров Docker.

Итак, построим образ Docker и присвоим ему имя `universe`:

```
docker build -t universe .
```

После того как образ Docker будет построен, выполните следующую команду для запуска контейнера из образа Docker:

```
docker run --privileged --rm -it -p 12345:12345 -p 5900:5900 -e
```

```
DOCKER_NET_HOST=172.17.0.1 universe /bin/bash
```

OpenAI Gym

В OpenAI Gym можно моделировать разнообразные среды и заниматься разработкой, оценкой и сравнением алгоритмов RL. В этом разделе я объясню, как использовать Gym.

Базовое моделирование

Посмотрим, как смоделировать среду с шестом на тележке.

1. Сначала импортируйте библиотеку:

```
import gym
```

2. Затем создайте экземпляр среды функцией `make`:

```
env = gym.make('CartPole-v0')
```

3. Инициализируйте среду методом `reset`:

```
env.reset()
```

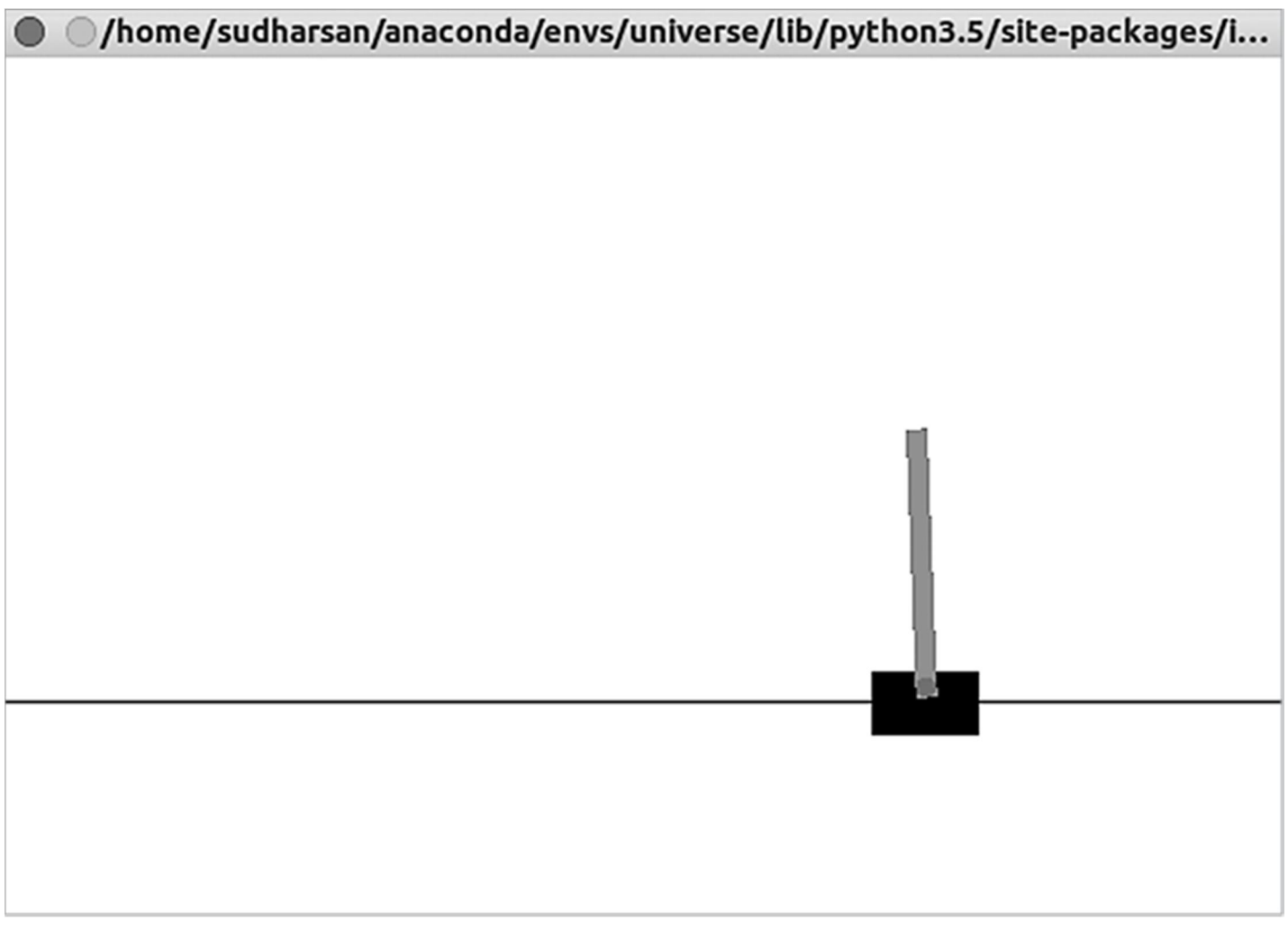
4. Выполните в цикле несколько квантов времени с прорисовкой среды при каждом кванте.

```
for _ in range(1000):
    env.render()
    env.step(env.action_space.sample())
```

Полный код выглядит так:

```
import gym
env = gym.make('CartPole-v0')
env.reset()
for _ in range(1000):
    env.render()
    env.step(env.action_space.sample())
```

Запустив эту программу, вы увидите вывод — среду с шестом на тележке:



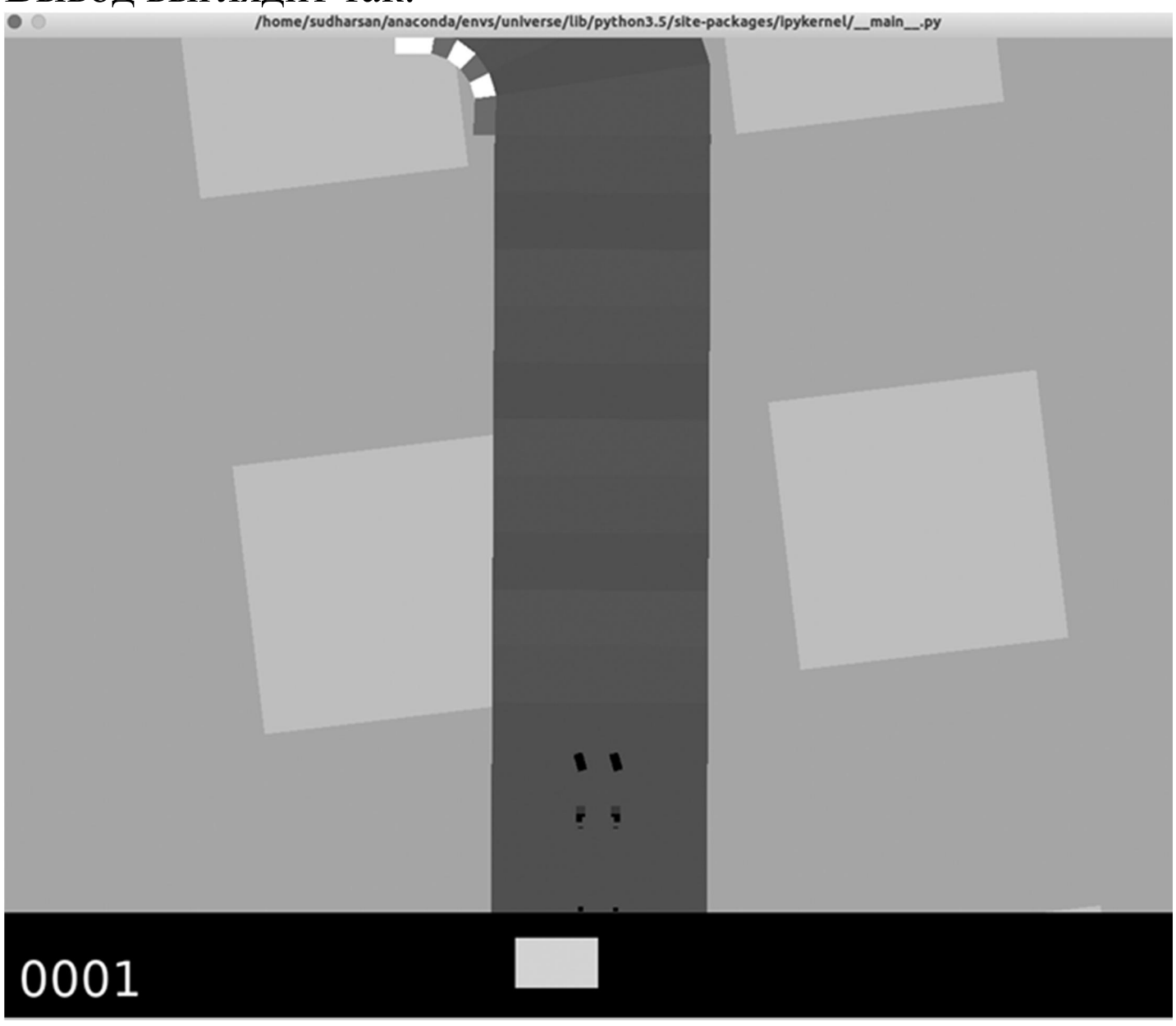
OpenAI Gym предоставляет множество моделируемых сред для обучения, оценки и построения агентов. Чтобы получить информацию о доступных средах, либо обратитесь на сайт, либо введите команду для вывода списка доступных сред:

```
from gym import envs
print(envs.registry.all())
```

Gym предоставляет много разных интересных сред. Давайте смоделируем среду «Автогонок»:

```
import gym
env = gym.make('CarRacing-v0')
env.reset()
for _ in range(1000):
    env.render()
    env.step(env.action_space.sample())
```

Вывод выглядит так:



Робот учится ходить

А теперь попробуем научить робота ходить при помощи Gym.

Стратегия заключается в том, что при движении робота вперед он получает *X* очков награды, а если робот двигаться не может, то теряет *Y*очков. Таким образом, робот учится ходить с расчетом на максимизацию награды.

Сначала мы импортируем библиотеку, а затем создаем экземпляр эксперимента функцией `make`. Open AI Gym предоставляет среду с именем **`BipedalWalker-v2`** для обучения агентов-роботов на простой местности:

```
import gym
env = gym.make('BipedalWalker-v2')
```

Затем для каждого эпизода (взаимодействия агента со средой между исходным и конечным состоянием) среда инициализируется методом `reset`:

```
for episode in range(100):
  observation = env.reset()
```

После этого в цикле выполняется прорисовка среды:

```
for i in range(10000):
env.render()
```

Мы выполняем случайную выборку действий из пространства действий среды. С каждой средой связывается пространство действий, содержащее все возможные допустимые действия:

```
action = env.action_space.sample()
```

Для каждого шага действия сохраняются значения `observation`, `reward`, `done` и `info`:

```
observation, reward, done, info =
env.step(action)
```

- `observation` — объект, представляющий наблюдение за состоянием среды (например, состояние робота на местности).
- `reward` — награды, полученные в результате предыдущего действия (например, награда, полученная роботом за успешное движение вперед).
- `done` — логическое значение; если оно равно `true`, это значит, что эпизод завершен (то есть робот научился ходить или эксперимент пришел к полной неудаче). После завершения эпизода среда инициализируется для следующего эпизода вызовом `env .reset()`.
- `info` — информация, полезная для отладки.

Если значение `done` равно `true`, мы выводим количество временных квантов, задействованных в эпизоде, и прерываем текущий эпизод:

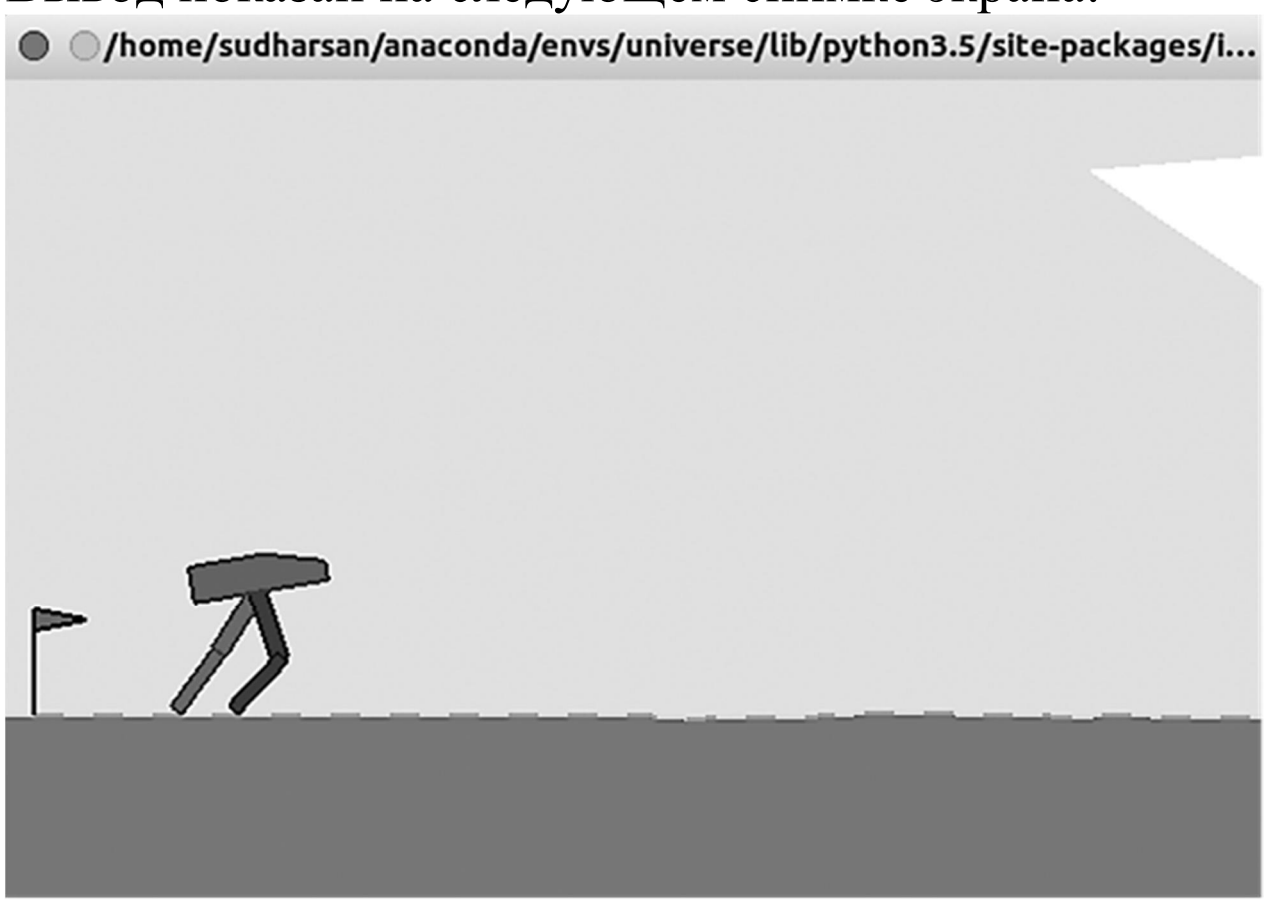
```
if done:
  print("{} timesteps taken for the
Episode".format(i+1))
  break
```

Полный код:

```
import gym
env = gym.make('BipedalWalker-v2')
for i_episode in range(100):
observation = env.reset()
for t in range(10000):
    env.render()
    print(observation)
    action = env.action_space.sample()
```

```
        observation, reward, done, info =
env.step(action)
        if done:
            print("{} timesteps taken for the
episode".format(t+1))
            break
```

Вывод показан на следующем снимке экрана:



OpenAI Universe

OpenAI Universe предоставляет широкий спектр реалистичных игровых сред. Это расширение OpenAI Gym. Universe дает возможность тренировать и оценивать агентов в разнообразных игровых средах, от простых до сложных сред реального времени, а также предоставляет неограниченный доступ ко многим игровым средам.

Построение бота для видеоигры

А теперь посмотрим, как построить бота для видеоигры «Автогонки». Машина должна двигаться вперед, не сталкиваясь с препятствиями или другими машинами.

Сначала мы импортируем необходимые библиотеки:

```
import gym
import universe # register universe environment
import random
```

Затем среда «Автогонок» моделируется функцией `make`:

```
env = gym.make('flashgames.NeonRace-v0')
env.configure(remotes=1) #автоматически создает
локальный контейнер docker
```

Создадим переменные для перемещения машины:

```
# Движение налево
left = [('KeyEvent', 'ArrowUp', True),
('KeyEvent', 'ArrowLeft', True),
```

```
            ('KeyEvent', 'ArrowRight', False)]

    # Движение направо
    right = [('KeyEvent', 'ArrowUp', True),
('KeyEvent', 'ArrowLeft', False),
             ('KeyEvent', 'ArrowRight', True)]

    # Движение вперед
    forward = [('KeyEvent', 'ArrowUp', True),
('KeyEvent', 'ArrowRight', False),
               ('KeyEvent', 'ArrowLeft', False),
('KeyEvent', 'n', True)]
```

Инициализируем ряд других переменных:

```
    # Переменная turn определяет, нужно ли повернуть
    turn = 0
    # Все награды сохраняются в списке rewards
    rewards = []

    # Переменная buffer_size используется как
пороговое значение
    buffer_size = 100

    # Изначально выбирается действие forward, при
котором машина
    # просто двигается вперед без поворота
    action = forward
```

Теперь дадим возможность нашему игровому агенту играть в бесконечном цикле, который непрерывно выполняет действие на основании взаимодействия со средой:

```
    while True:
        turn -= 1
    # Допустим, изначально машина двигается вперед
без поворота.
    # Проверим значение turn. Если оно меньше 0,
    # то поворачивать не нужно, и машина просто
двигается вперед.
        if turn <= 0:
            action = forward
            turn = 0
```

Затем вызовем `env.step()` для выполнения действия (на данный момент это движение вперед) для одного кванта:

```
    action_n = [action for ob in observation_n]
    observation_n, reward_n, done_n, info =
env.step(action_n)
```

Для каждого кванта времени результаты сохраняются в переменных `observation_n`, `reward_n`, `done_n` и `info_n`:

- `observation_n` — состояние машины;
- `reward_n` — награда, полученная за предыдущее действие, если машина успешно двигалась вперед без столкновения с препятствиями;
- `done_n` — логическая переменная; значение `true` говорит о том, что игра закончена;
- `info_n` — используется для целей отладки.

Очевидно, агент (машина) не может постоянно ехать вперед всю игру; он должен поворачивать, обходить препятствия, а также время от времени сталкиваться с другими машинами. Но агент должен определить, нужно ли повернуть, и если нужно, то в каком направлении.

Сначала вычисляется среднее значение наград, полученных до настоящего момента; если оно равно 0, значит, машина где-то застряла в процессе движения вперед, и ей нужно повернуть. В каком направлении поворачивать? А вы помните **функции политики**, представленные в главе 1?

Согласно концепции игры, здесь имеются две политики: поворот налево и поворот направо. Мы выберем случайную политику, вычислим награду и займемся ее улучшением.

Для этого мы сгенерируем случайное число; если оно меньше 0,5, машина повернет направо, а если нет — налево. Позднее награды будут обновлены, и на основании наград мы определим, какое направление лучше:

```
if len(rewards) >= buffer_size:
        mean = sum(rewards)/len(rewards)

        if mean == 0:
            turn = 20
            if random.random() < 0.5:
                action = right
            else:
                action = left
        rewards = []
```

Затем для каждого эпизода (допустим, игра закончена) среда инициализируется заново (то есть запускается с начала) методом `env.render()`:

```
  env.render()
```

Полный код выглядит так:

```
import gym
import universe # register universe environment
import random

env = gym.make('flashgames.NeonRace-v0')
```

```
  env.configure(remotes=1) # automatically creates
a local docker container
  observation_n = env.reset()

  # Объявление действий
  # Движение налево
  left = [('KeyEvent', 'ArrowUp', True),
('KeyEvent', 'ArrowLeft', True),
          ('KeyEvent', 'ArrowRight', False)]

  # Движение направо
  right = [('KeyEvent', 'ArrowUp', True),
('KeyEvent', 'ArrowLeft', False),
           ('KeyEvent', 'ArrowRight', True)]

  # Движение вперед
  forward = [('KeyEvent', 'ArrowUp', True),
('KeyEvent', 'ArrowRight',
  False),
          ('KeyEvent', 'ArrowLeft', False),
('KeyEvent', 'n', True)]

  # Переменная turn определяет, нужно ли повернуть
  turn = 0
  # Все награды сохраняются в списке rewards
  rewards = []
  # Переменная buffer_size используется как
пороговое значение
  buffer_size = 100
  # Изначально выбирается действие forward
  action = forward

  while True:
      turn -= 1
      if turn <= 0:
          action = forward
          turn = 0
      action_n = [action for ob in observation_n]
      observation_n, reward_n, done_n, info =
env.step(action_n)
      rewards += [reward_n[0]]
      if len(rewards) >= buffer_size:
          mean = sum(rewards)/len(rewards)
```

```
            if mean == 0:
                turn = 20
                if random.random() < 0.5:
                    action = right
                else:
                    action = left
            rewards = []

        env.render()
```

Запустив программу, вы увидите, как машина учится двигаться без столкновений с препятствиями или другими машинами.

TensorFlow

TensorFlow — библиотека с открытым кодом от компании Google, широко используемая для обработки числовых данных. В частности, она находит применение в построении моделей глубокого обучения — подмножества машинного обучения. Она использует графы потоков данных, которые могут совместно использоваться и выполняться на многих различных платформах. Тензор представляет собой многомерный массив, поэтому название TensorFlow буквально обозначает поток многомерных массивов (тензоров) на графе вычислений.

После того как в системе будет установлен пакет Anaconda, установить TensorFlow будет совсем несложно. Независимо от того, какую платформу вы используете, TensorFlow легко устанавливается следующими командами:

```
source activate universe
```

```
conda install -c conda-forge tensorflow
```

совет

Не забудьте активировать среду `universe` перед установкой TensorFlow.

Чтобы проверить, успешно ли прошла установка TensorFlow, просто выполните программу `Hello World`:

```
import tensorflow as tf
hello = tf.constant("Hello World")
sess = tf.Session()
print(sess.run(hello))
```

Переменные, константы и заместители

Переменные, константы и заместители — фундаментальные элементы TensorFlow. Впрочем, неопытные разработчики их часто путают, поэтому мы последовательно рассмотрим эти элементы и разберемся, чем же они отличаются.

Переменные

Переменные (variables) используются для хранения значений. В частности, они поставляют входные данные для других операций графа вычислений. Для создания переменных в TensorFlow используется функция `tf.Variable()`. В следующем примере мы создадим переменную со значениями из случайного нормального распределения и присвоим ей имя `weights`:

```
weights = tf.Variable(tf.random_normal([3, 2], stddev=0.1), name="weights")
```

После определения переменной необходимо явно создать операцию инициализации с использованием метода `tf.global_variables_initializer()`, который выделяет ресурсы для переменной.

Константы

Константы (constants), в отличие от переменных, не могут изменять хранящиеся в них значения. Константы неизменны; значение, присвоенное константе, остается на протяжении всей работы программы. Для создания констант используется функция `tf.constant()`:

```
x = tf.constant(13)
```

Заместители

Заместители (placeholders) можно рассматривать как переменные, для которых вы определяете только тип и размеры, но не присваиваете значение. Заместители определяются без значений; последние будут получены во время выполнения программы. У заместителей имеется необязательный аргумент с именем `shape`, который задает размеры

данных. Если `shape` имеет значение `None`, то во время выполнения заместителю могут быть присвоены данные любого размера. Заместители определяются функцией `tf.placeholder()`:

```
x = tf.placeholder("float", shape=None)
```

Проще говоря, `tf.Variable` используется для хранения данных, а `tf.placeholder` — для поставки внешних данных.

Граф вычислений

В TensorFlow все представляется *графом вычислений*, который состоит из ребер и узлов. Узлы соответствуют математическим операциям (сложение, умножение и т.д.), а ребра — тензорам. Графы вычислений чрезвычайно эффективно работают для оптимизации ресурсов и упрощают распределенные вычисления.

Допустим, имеется узел *B*, ввод которого зависит от вывода узла *A*; такие зависимости называются *прямыми зависимостями*.

Пример:

```
A = tf.multiply(8,5)
B = tf.multiply(A,1)
```

Если узел *B* не зависит от узла *A* для получения своего ввода, такая зависимость называется *косвенной*.

Пример:

```
A = tf.multiply(8,5)
B = tf.multiply(4,3)
```

Наличие информации о зависимостях позволяет распределить независимые вычисления среди доступных ресурсов и сократить время их выполнения.

Каждый раз, когда вы импортируете TensorFlow, автоматически создается граф по умолчанию, а все создаваемые вами узлы будут с ним связаны.

Сеансы

Графы вычислений только определяются в программе; для выполнения графов используются *сеансы* TensorFlow:

```
sess = tf.Session()
```

Сеанс для графа вычислений создается методом `tf.Session()`, который выделяет память для хранения текущего значения переменной. После создания сеанса граф выполняется методом `sess.run()`.

Чтобы выполнить что-либо в TensorFlow, необходимо запустить сеанс TensorFlow для экземпляра:

```
import tensorflow as tf
a = tf.multiply(2,3)
print(a)
```

Вместо обычного значения `6` программа выведет объект TensorFlow. Как упоминалось ранее, при импортировании TensorFlow создается граф

вычислений по умолчанию, а все узлы `a`, которые мы создадим, присоединяются к графу. Для выполнения графа необходимо инициализировать сеанс TensorFlow следующим образом:

```
# Импортирование tensorflow
import tensorflow as tf

# Инициализация переменных
a = tf.multiply(2,3)

# Создать сеанс tensorflow для выполнения
with tf.Session() as sess:
  # запустить сеанс
  print(sess.run(a))
```

Приведенный код выведет `6`.

TensorBoard

TensorBoard — инструмент визуализации TensorFlow, который может использоваться для наглядного представления графа вычислений. Также с его помощью можно строить графики различных количественных метрик и результатов нескольких промежуточных вычислений. С TensorBoard легко представляются сложные модели, что может быть полезно для отладки и распространения результатов.

Построим простой граф вычислений и отобразим его наглядное представление в TensorBoard.

Начнем с импортирования библиотеки:

```
import tensorflow as tf
```

Затем инициализируем переменные:

```
a = tf.constant(5)
b = tf.constant(4)
c = tf.multiply(a,b)
d = tf.constant(2)
e = tf.constant(3)
f = tf.multiply(d,e)
g = tf.add(c,f)
```

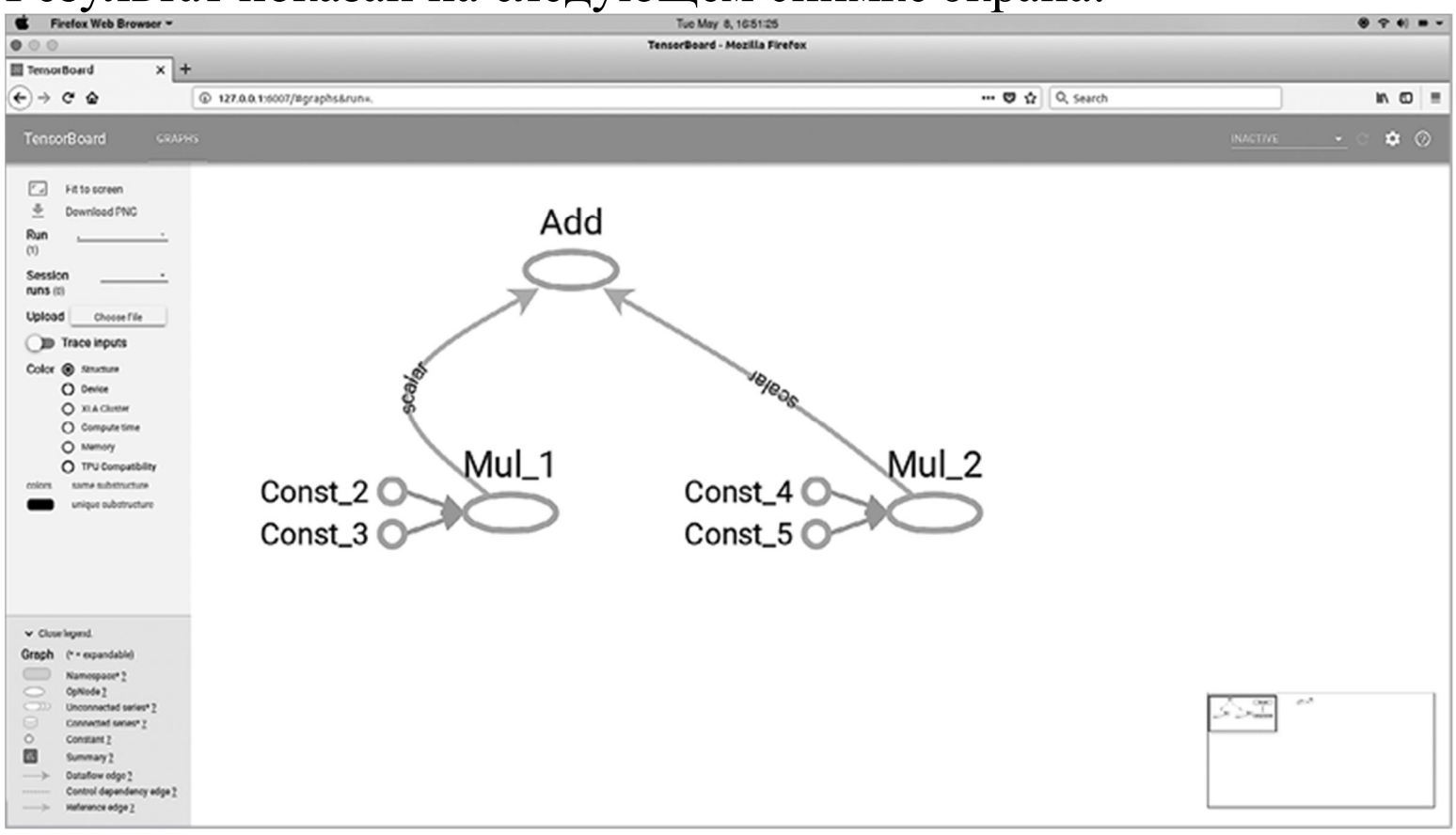
Теперь создадим сеанс TensorFlow. Результаты графа будут записаны в файл под названием **`output`** с использованием метода `tf.summary.FileWriter()`:

```
with tf.Session() as sess:
  writer = tf.summary.FileWriter("output",
sess.graph)
  print(sess.run(g))
  writer.close()
```

Чтобы запустить TensorBoard, откройте терминал, найдите рабочий каталог и введите команду:

```
tensorboard --logdir=output --port=6003
```

Результат показан на следующем снимке экрана:



Добавление групп

Группы используются для сокращения сложности и лучшего понимания модели. Взаимосвязанные узлы помещаются в одну группу. Так, в предыдущем примере граф можно разбить на две разные группы с именами `Computation` и `Result`. Взглянув на пример, можно заметить, что узлы `a`–`e` выполняют вычисления, а узел `g` вычисляет результат. Следовательно, эти узлы можно разделить на разные группы для простоты. Группы создаются функцией `tf.name_scope()`.

Воспользуемся функцией `tf.name_scope()` для создания группы `Computation`:

```
with tf.name_scope("Computation"):
    a = tf.constant(5)
    b = tf.constant(4)
    c = tf.multiply(a,b)
    d = tf.constant(2)
    e = tf.constant(3)
    f = tf.multiply(d,e)
```

Затем используем функцию `tf.name_scope()` для группы `Result`:

```
with tf.name_scope("Result"):
    g = tf.add(c,f)
```

Взгляните на группу `Computation`; ее можно дополнительно разбить на части для еще большей наглядности. Мы создадим группу `Part1`, объединяющую узлы `a`-`c`, и группу `Part2` с узлами `d`-`e`, так как эти группы не зависят друг от друга:

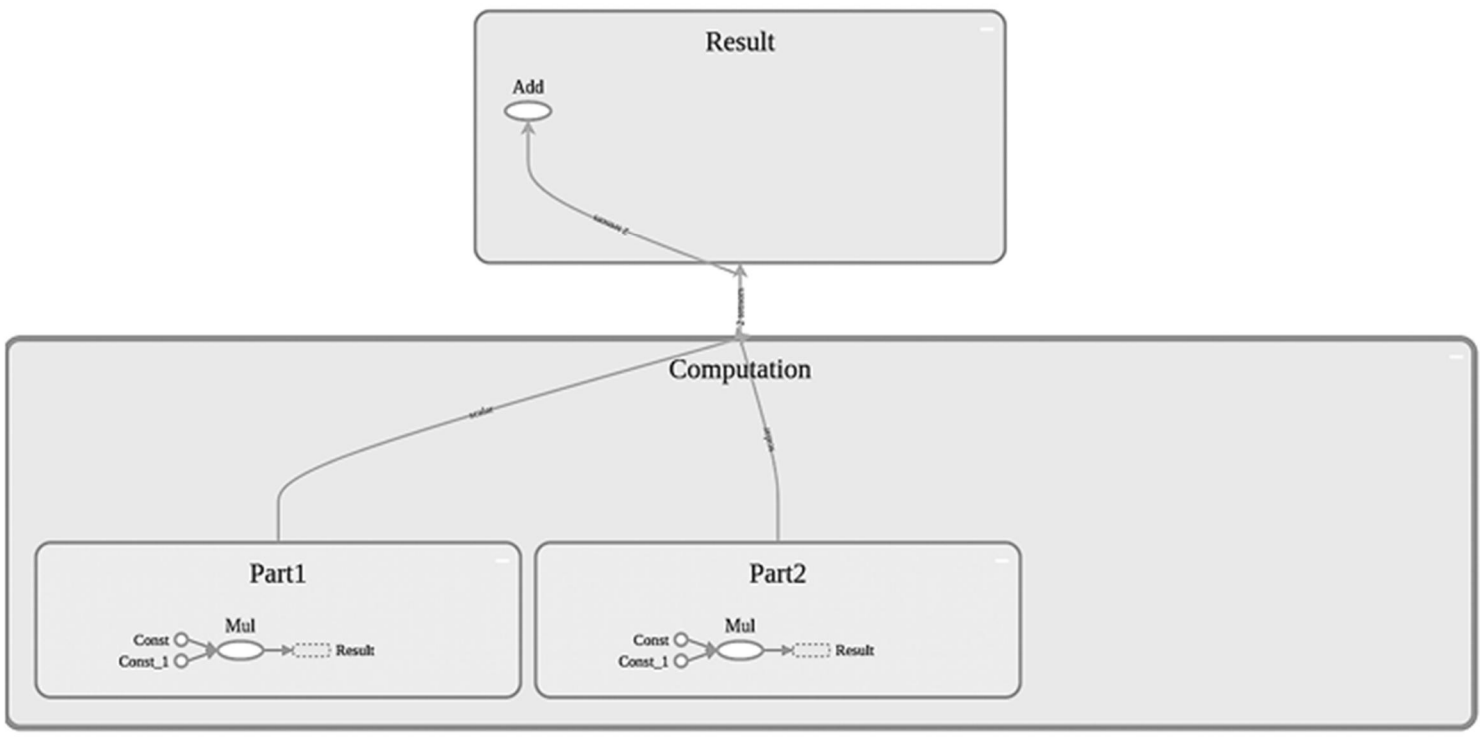
```
with tf.name_scope("Computation"):
    with tf.name_scope("Part1"):
        a = tf.constant(5)
```

```
        b = tf.constant(4)
        c = tf.multiply(a,b)
    with tf.name_scope("Part2"):
        d = tf.constant(2)
        e = tf.constant(3)
        f = tf.multiply(d,e)
```

Чтобы лучше понять структуру группировки, выведите ее наглядное представление в TensorBoard. Полный код выглядит так:

```
import tensorflow as tf
with tf.name_scope("Computation"):
    with tf.name_scope("Part1"):
        a = tf.constant(5)
        b = tf.constant(4)
        c = tf.multiply(a,b)
    with tf.name_scope("Part2"):
        d = tf.constant(2)
        e = tf.constant(3)
        f = tf.multiply(d,e)

with tf.name_scope("Result"):
    g = tf.add(c,f)

with tf.Session() as sess:
  writer = tf.summary.FileWriter("output",
sess.graph)
  print(sess.run(g))
  writer.close()
```

Из следующего рисунка вы сможете легко понять, как группировка упрощает понимание логических связей и сокращает сложность. Группировка особенно часто применяется в работе со сложными проектами, в которых она помогает лучше понять функциональность графа и взаимные зависимости между узлами:



Итоги

В этой главе вы узнали, как подготовить систему и установить Anaconda, Docker, OpenAI Gym, Universe и TensorFlow. Вы также узнали, как создавать модели в OpenAI и как настраивать агентов для обучения в среде OpenAI. Вам были представлены основные принципы использования TensorFlow и возможности наглядного представления графов в TensorBoard.

Глава 3 посвящена марковскому процессу принятия решений и динамическому программированию, а также решению задачи о замерзшем озере с использованием функций ценности и политики.

Вопросы

1. Для чего и как создаются новые среды в Anaconda?
2. Для чего используется Docker?
3. Как моделировать среду в OpenAI Gym?
4. Как проверить список всех доступных сред в OpenAI Gym?
5. OpenAI Gym и Universe — одно и то же? Если нет, то чем они отличаются?
6. Чем переменные TensorFlow отличаются от заместителей?
7. Что такое «граф вычислений»?
8. Для чего нужны сеансы TensorFlow?
9. Для чего используется TensorBoard и как запустить эту программу?

Дополнительные источники

Блог OpenAI: `https://blog.openai.com.`

Среды OpenAI: `https://gym.openai.com/envs/.`

Официальный веб-сайт TensorFlow: `https://www.tensorflow.org/.`

3. Марковский процесс принятия решений и динамическое программирование

Марковский процесс принятия решений(MDP, Markov decision process) предоставляет математическую основу для решения проблем **обучения с подкреплением** (**RL**). Почти все задачи RL могут моделироваться в форме MDP, широко применяемой для его оптимизации. В этой главе вы узнаете, что такое MDP и как его использовать. Также будет представлено динамическое программирование — эффективный метод решения сложных задач.

В этой главе рассмотрены следующие темы:

- Марковские цепи и марковские процессы.
- Марковский процесс принятия решений.
- Награды и возврат.
- Уравнение Беллмана.

• Решение уравнения Беллмана средствами динамического программирования.

• Решение задачи о замерзшем озере с использованием функций ценности и политики.

Марковские цепи и марковские процессы

Прежде чем браться за MDP, сначала необходимо познакомиться с марковскими цепями и марковскими процессами, лежащими в его основе.

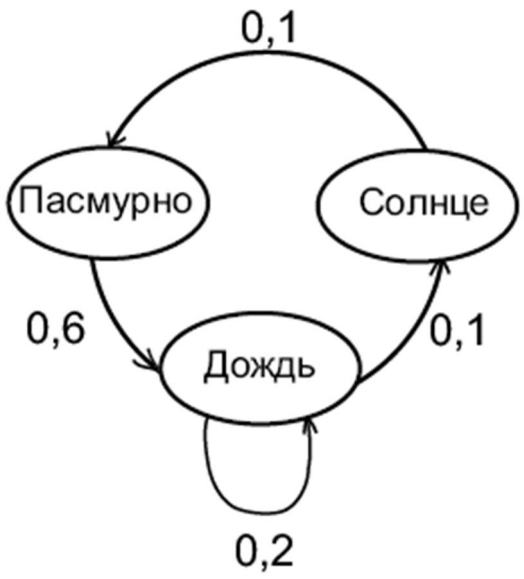
Марковское свойство гласит, что будущее зависит только от настоящего, но не от прошлого. Марковская цепь представляет собой вероятностную модель, которая для прогнозирования следующего состояния зависит исключительно от текущего состояния, но не от предыдущих состояний, — иначе говоря, будущее условно независимо от прошлого. Марковская цепь строго соответствует марковскому свойству.

Например, если вы знаете, что текущее состояние погоды — «пасмурно», то можно предсказать, что следующим состоянием может быть «дождь». Мы пришли к выводу о том, что следующим состоянием может быть «дождь» только на основании текущего состояния («пасмурно»), но не прошлых состояний (которые могут быть любыми — «солнце», «ветер» и т.д.). Однако марковское свойство выполняется не для всех процессов. Например, бросок кубика (следующее состояние) не зависит от результата предыдущего броска (текущее состояние).

Переход из одного состояния в другое так и называется — *переходом*, а его вероятность называется *вероятностью перехода*. Вероятности перехода можно свести в таблицу, показанную ниже; эта таблица называется *марковской таблицей*. Таблица содержит вероятности перехода в следующее состояние из текущего состояния:

Текущее состояние	Следующее состояние	Вероятность перехода
Пасмурно	Дождь	0,6
Дождь	Дождь	0,2
Солнце	Пасмурно	0,1
Дождь	Солнце	0,1

Марковскую цепь также можно представить в виде рисунка с обозначенными вероятностями переходов:



Здесь представлены вероятности перехода из одного состояния в другое.

Все еще не понимаете, что такое марковские цепи? Хорошо, давайте посмотрим на примере.

Я: «Чем вы занимаетесь?»

Вы: «Читаю о марковских цепях».

Я: «Чем собираетесь заняться после чтения?»

Вы: «Лягу спать».

Я: «Уверены?»

Вы: «Вероятно. Если не захочется спать, посмотрю телевизор».

Я: «Класс! Это и есть марковская цепь».

Вы: «Да ну?»

Эту беседу можно представить в виде марковской цепи и представить ее графически:

В основе марковских цепей лежит базовый принцип: будущее зависит только от настоящего, но не от прошлого. Стохастический процесс называется марковским процессом, если он обладает марковским свойством.

Марковский процесс принятия решений

Марковский процесс принятия решений(*MDP*) является расширением марковских цепей. Он формирует математическую основу для моделирования ситуаций с принятием решений. Почти все задачи обучения с подкреплением могут быть смоделированы на базе MDP.

MDP характеризуется пятью важными элементами:

• Набор состояний (*S*), в котором может находиться агент.

• Набор действий (*A*), которые могут выполняться агентом для перехода из одного состояния в другое.

• Вероятность перехода ($P^a_{ss'}$) — вероятность перехода из одного состояния *s* в другое состояние *s′* посредством выполнения действия *a*.

• Вероятность награды ($R^a_{ss'}$) — вероятная награда, получаемая агентом при переходе из одного состояния *s* в другое состояние *s'* посредством выполнения действия *a*.

• Поправочный коэффициент (γ), управляющий важностью немедленных и будущих наград. Он будет подробно рассмотрен ниже.

Награды и возврат

Как вы узнали, в RL агент взаимодействует со средой, выполняя действия и переходя из одного состояния в другое. В зависимости от выполняемых действий агент получает награду. Награда представляет собой обычное числовое значение: допустим, +1 для «хорошего» действия или –1 — для «плохого». Как решить, было действие хорошим или плохим? При поиске пути в лабиринте хорошим считается перемещение, которое не приводит к столкновению со стеной, а плохим — перемещение со столкновением.

Агент пытается максимизировать сумму наград (накопленную награду), полученную от среды, а не каждую немедленную награду. Сумма наград, полученных агентом от среды, называется *возвратом*. Таким образом, сумма наград (возврат), полученная агентом, может быть выражена посредством следующей формулы:

$R_t = r_{t+1} + r_{t+2} + \ldots + r_T$.

Здесь r_{t+1} — награда, получаемая агентом в кванте времени t_0 при выполнении действия a_0 для перехода из одного состояния в другое. r_{t+2} — награда, получаемая агентом в кванте времени t_1 при выполнении действия a_1 для перехода из одного состояния в другое. Наконец, r_T — награда, получаемая агентом в последнем кванте времени T при выполнении действия для перехода из одного состояния в другое.

Эпизодические и непрерывные задачи

Эпизодическими задачами называются задачи, имеющие завершающее состояние (конец). В RL эпизоды рассматриваются как взаимодействия агента со средой от исходного до завершающего состояния.

Например, в видеоигре «Автогонки» вы начинаете игру (исходное состояние) и играете, пока игра не будет окончена (завершающее состояние). Это называется *эпизодом*. После того как игра окончена, вы перезапускаете игру, начиная следующий эпизод с исходного состояния независимо от состояния в предыдущей игре. Таким образом, каждый эпизод не зависит от других эпизодов.

У *непрерывных* задач завершающего состояния не бывает. Непрерывные задачи никогда не оканчиваются. Например, у робота — личного помощника нет завершающего состояния.

Поправочный коэффициент

Вы увидели, что целью агента является максимизация возврата. Для эпизодической задачи можно определить возврат по формуле $R_t = r_{t+1} + r_{t+2} + \ldots + r_T$, где T — завершающее состояние эпизода, и попытаться максимизировать возврат R_t.

Так как у непрерывной задачи завершающего состояния нет, можно определить возврат для непрерывных задач по формуле $R_t = r_{t+1} + r_{t+2} + \ldots + \ldots$ до бесконечности. Но как максимизировать возврат, если суммирование не прекращается?

Для этого было введено понятие поправочного коэффициента. Переопределим возврат с поправочным коэффициентом (γ) в следующем виде:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \ldots = \textbf{(3.1)}$$

$$= \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}. \quad \textbf{(3.2)}$$

Поправочный коэффициент определяет относительную важность будущих и немедленных наград. Его значение лежит в диапазоне от 0 до 1. Поправочный коэффициент 0 означает, что немедленные награды более важны, а поправочный коэффициент 1 означает, что будущие награды важнее немедленных.

С поправочным коэффициентом 0 никакого обучения вообще не будет, поскольку учитываться будут только немедленные награды; с другой стороны, поправочный коэффициент 1 будет неограниченно стремиться к будущим наградам, что может завести в бесконечность. Таким образом, оптимальное значение поправочного коэффициента лежит в диапазоне от 0,2 до 0,8.

Важность немедленных и будущих наград распределяется в зависимости от сценария их использования. В некоторых ситуациях будущие награды важнее немедленных, и наоборот. В шахматной партии целью является мат королю противника. Если назначить слишком большую важность немедленным наградам, которые даются за такие действия, как захват игроком вражеской фигуры и т.д., агент будет учиться реализовывать эту подцель, вместо того чтобы учиться достигать настоящей цели. Таким образом, в данном случае будущие награды важнее, тогда как в других случаях предпочтение отдается немедленным наградам перед будущими. (Что бы вы выбрали — получить конфету сейчас или через год?)

Функция политики

В **главе 1** упоминалась функция политики, связывающая состояния с действиями. Она обозначается символом π.

Функцию политики (policy function) можно представить в виде $\pi(s) : S \rightarrow A$, то есть как отображение множества состояний на множество действий. По сути функция политики указывает, какое действие должно выполняться в каждом состоянии. Нашей конечной целью является нахождение оптимальной политики, задающей для каждого состояния правильное действие, максимизирующее награду.

Функция ценности состояния

Функция ценности состояния (state value function) также называется просто «функцией ценности». Она указывает, насколько хорошо для

агента пребывание в конкретном состоянии при политике π, то есть оценивает ценность состояния при следовании политике. Чаще всего функцию ценности обозначают $V(s)$.

Она определяется так:

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[R_t \mid s_t = s \right].$$

Она показывает ожидаемый возврат, начинающийся с состояния s, в соответствии с политикой π. Значение R_t подставляется в функцию ценности из формулы **(3.2)** следующим образом:

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

Обратите внимание: функция ценности состояния и ее изменения зависят от выбранной политики.

Функции состояния можно представить в виде таблицы. Допустим, имеются два состояния, и оба следуют политике π. На основании ценности этих двух состояний можно сказать, насколько хорошо для нашего агента находиться в этом состоянии при следовании политике. Чем больше ценность, тем лучше состояние:

Состояние	Ценность
Состояние 1	0,3
Состояние 2	0,9

На основании приведенной таблицы можно сказать, что находиться в состоянии 2 хорошо, поскольку оно обладает высокой ценностью. Далее в этой главе будет показано, как интуитивно оценивать такие значения.

Функция ценности состояния/действия (Q-функция)

Функция ценности состояния/действия также называется *Q-функцией*. Она показывает, насколько хорошо для агента выполнять конкретное действие в состоянии с политикой π. *Q*-функция обозначается $Q(s)$ и определяет ценность выполнения действия в состоянии при следовании политике π.

Q-функцию можно определить следующим образом:

$$Q^{\pi}(s,\ a) = \mathbb{E}_{\pi}\left[R_t \mid s_t = s,\ a_t = a \right].$$

Функция задает ожидаемый возврат, начиная с состояния s с действием a в соответствии с политикой π. Значение R_tподставляется в *Q*-функцию из **(3.2)**следующим образом:

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

Q-функция, в отличие от функции ценности, определяет полезность действия в состоянии, а не полезность самого состояния.

Q-функции, как и функции ценности состояния, могут быть представлены в виде таблицы. Такая таблица также называется *Q*-

таблицей. Допустим, имеются два состояния и два действия; *Q*-таблица выглядит примерно так:

Состояние	Действие	Ценность
Состояние 1	Действие 1	0,03
Состояние 1	Действие 2	0,02
Состояние 2	Действие 1	0,5
Состояние 2	Действие 2	0,9

Итак, *Q*-таблица показывает ценность всех возможных пар «состояние/действие». Из таблицы следует, что выполнение действия 1 в состоянии 1 и действия 2 в состоянии 2 является предпочтительным вариантом, поскольку оно обладает более высокой ценностью.

Каждый раз, когда мы говорим о функции ценности $V(s)$ или Q-функции $Q(s, a)$, на самом деле имеется в виду таблица ценности и Q-таблица, представленные выше.

Уравнение Беллмана и оптимальность

Уравнение Беллмана, названное по имени американского математика Ричарда Беллмана (Richard Bellman), используется для решения задач MDP. В RL оно встречается на каждом шагу. Когда мы говорим о «решении задач MDP», на самом деле имеется в виду задача нахождения оптимальных функций политики и ценности. Существует много разных функций ценности для разных политик. *Оптимальной функцией ценности* $V^*(s)$ называется функция ценности, которая обеспечивает максимальную ценность по сравнению со всеми другими функциями ценности:

$$V^*(s) = \max_{\pi} V^{\pi}(s).$$

Аналогичным образом *оптимальной политикой* называется политика, обеспечивающая оптимальную функцию ценности.

Поскольку оптимальная функция ценности $V^*(s)$ имеет более высокую ценность по сравнению со всеми остальными функциями ценности (то есть обеспечивает максимальный возврат), ее значение вычисляется как максимум по всем Q-функциям. Итак, оптимальная функция ценности легко вычисляется определением максимума по Q-функциям:

$$V^*(s) = \max_{a} Q^*(s, a). \quad \textbf{(3.3)}$$

Уравнение Беллмана для функции ценности может быть представлено в следующем виде (о том, как было выведено это уравнение, рассказано в следующем разделе):

$$V^{\pi}(s) = \sum_{a} \pi(s, a) \sum_{s'} P^{a}_{ss'} \left[R^{a}_{ss'} + \gamma V^{\pi}(s') \right].$$

Оно обозначает рекурсивную связь между ценностью состояния, ценностью его последующего состояния и среднего значения по всем возможным вариантам.

Кроме того, уравнение Беллмана для *Q*-функции может быть представлено в следующем виде:

$$Q^{\pi}(s,\ a) = \sum_{s'} P^{a}_{ss'} \left[R^{a}_{ss'} + \gamma \sum_{a'} Q^{\pi}(s',\ a') \right]. \quad \textbf{(3.4)}$$

Подставляя уравнение **(3.4)** в **(3.3)**, получаем:

$$V^{*}(s) = \max_{a} \sum_{s'} P^{a}_{ss'} \left[R^{a}_{ss'} + \gamma \sum_{a'} Q^{\pi}(s',\ a') \right].$$

Предыдущее уравнение называется *уравнением оптимальности Беллмана*. Далее вы увидите, как находить оптимальные политики, решая это уравнение.

Вывод уравнения Беллмана для функции ценности и Q-функции

Теперь посмотрим, как вывести уравнения Беллмана для функций ценности и *Q*-функций.

Если математика вас не интересует, этот раздел можно пропустить, тем не менее эта математика будет невероятно захватывающей.

Сначала мы определяем $P^{a}_{ss'}$ как вероятность перехода из состояния s в состояние s' при выполнении действия a:

$$P^{a}_{ss'} = pr\left(s_{t+1} = s' \,|\, s_t = s,\ a_t = a\right).$$

$R^{a}_{ss'}$ определяется как вероятная награда при переходе из состояния s в состояние s' при выполнении действия a:

$$R^{a}_{ss'} = \mathbb{E}\left(R_{t+1} \,|\, s_t = s,\ s_{t+1} = s',\ a_t = a\right) =$$

$$= \gamma \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^{k} r_{t+k+2} \,|\, s_{t+1} = s' \right] \text{ из } \textbf{(3.2)}. \quad \textbf{(3.5)}$$

Мы знаем, что функция ценности может быть представлена в следующем виде:

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[R_t \,|\, s_t = s \right],$$

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[r_{t+1} + \gamma r_{t+2} + \gamma^{2} r_{t+3} + \ldots \,|\, s_t = s \right] \text{ из } \textbf{(3.1)}.$$

Функцию ценности можно переписать с вынесением первой награды из суммы:

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^{k} r_{t+k+2} \,|\, s_t = s \right]. \quad \textbf{(3.6)}$$

Часть функции ценности задает ожидаемый возврат при нахождении в состоянии s и выполнении действия a политикой π.

Таким образом, ожидаемый возврат можно переписать в явном виде, суммируя все возможные действия и награды:

$$\mathbb{E}_{\pi}\left[r_{t+1} \,|\, s_t = s \right] = \sum_{a} \pi(s,\ a) \sum_{s'} P^{a}_{ss'}\ R^{a}_{ss'}.$$

В правой части $R^a_{ss'}$ заменяется из уравнения **(3.5)**:

$$\sum_a \pi(s,\, a) \sum_{s'} P^a_{ss'} \gamma \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \,|\, s_{t+1} = s' \right].$$

В левой части значение r_{t+1}подставляется из уравнения **(3.2)**:

$$\mathbb{E}_\pi \left[\gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \,|\, s_t = s \right].$$

Таким образом, итоговое уравнение ожидаемого возврата приходит к следующему виду:

$$\mathbb{E}_\pi \left[\gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \,|\, s_t = s \right] = \sum_a \pi(s,\, a) \sum_{s'} P^a_{ss'} \gamma \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \,\middle|\, s_{t+1} = s' \right]. \quad \textbf{(3.7)}$$

Теперь ожидаемый возврат **(3.7)** в функции ценности **(3.6)** можно заменить в следующем виде:

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

$\mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \,|\, s_{t+1} = s' \right]$ можно заменить на $V\pi\,(s')$ из уравнения **(3.6)**, и итоговая функция ценности приходит к следующему виду:

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

Практически так же выводится уравнение Беллмана для *Q*-функции; итоговое уравнение выглядит так:

$$Q^\pi(s,\, a) = \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma \sum_{a'} Q^\pi(s',\, a') \right].$$

При наличии уравнений Беллмана для функции ценности и *Q*-функции становится понятно, как искать оптимальные политики.

Решение уравнения Беллмана

Оптимальные политики могут быть найдены решением уравнения оптимальности Беллмана. Для этого решения используется метод, называемый динамическим программированием.

Динамическое программирование

Динамическое программирование (*DP*, dynamic programming) — метод решения сложных задач, в котором вместо одной сложной задачи решаются несколько более простых подзадач, на которые она разбита, после чего для каждой подзадачи сохраняются решения. Если в ходе разбиения одна подзадача встретится несколько раз, то, вместо того чтобы вычислять его повторно, мы используем уже вычисленное решение. Таким образом, DP способствует сокращению времени вычислений. Динамическое программирование находит применение в разных областях: информатике, математике, биоинформатике и т.д.

Для решения уравнения Беллмана используются два мощных алгоритма:

- итерация по ценности;

• итерация по политикам.

Итерация по ценности

При *итерации по ценности* мы начинаем со случайной функции ценности. Очевидно, она не будет оптимальной, поэтому мы ищем новую улучшенную функцию ценности по итеративному алгоритму до тех пор, пока не будет найдена оптимальная функция ценности.

После ее нахождения можно легко вывести оптимальную политику:

Процесс итерации по ценности состоит из следующих шагов.

1. Инициализируется случайная функция ценности, то есть случайное значение для каждого состояния.

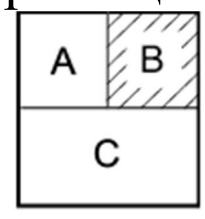
2. Вычисляется *Q*-функция для всех пар состояния/действия *Q* (*s*, *a*).

3. Функция ценности обновляется максимальным значением из *Q* (*s*, *a*).

4. Эти шаги повторяются до тех пор, пока изменение функции ценности не станет очень малым.

Чтобы этот процесс стал более понятным, проведем его вручную шаг за шагом.

Возьмем таблицу, изображенную ниже. Допустим, текущим является состояние **A**, а цель — достичь состояния **C** без посещения состояния **B**. Доступны всего два действия: 0 — перемещение направо/налево, 1 — перемещение вверх/вниз:



Вам удастся найти оптимальную политику? Оптимальной политикой будет та, которая приказывает выполнить действие 1 в состоянии **A**, чтобы мы могли достичь **C** без посещения **B**. Как ее найти? Давайте посмотрим.

Инициализировать случайную функцию ценности (то есть функцию со случайными значениями для всех состояний). Присвоим всем состояниям ценность 0:

Состояние	Ценность
A	0
B	0
C	0

Теперь вычислим значение *Q* для всех пар состояний/действий.

Q-функция сообщает ценность действия в каждом состоянии. Начнем с вычисления *Q* для состояния **A**. Вспомните уравнение *Q*-функции. Для вычисления необходимы вероятности переходов и наград. Рассмотрим переходы и вероятные награды для состояния **A** в следующем виде:

Состояние (s)	Действие (a)	Следующее состояние (s')	Вероятность перехода ($P^a_{ss'}$)	Награда перехода ($R^a_{ss'}$)
A	0	A	0,1	0
A	0	B	0,4	–1,0
A	0	C	0,3	1,0
A	1	A	0,3	0
A	1	B	0,1	–2,0
A	1	C	0,5	1,0

Q-функция для состояния **A** может быть вычислена по следующей формуле:

Q (*s*, *a*) = *Вероятность перехода* × (*вероятная награда* + *гамма* × *ценность следующего перехода*).

Здесь *гамма* — поправочный коэффициент (будем считать его равным 1).

Q для состояния **A** и действия 0:

$Q(A, 0) = (P^0_{лл} \times (R^0_{лл} + \gamma \times \textit{ценность } A)) + (P^0_{лв}(R^0_{лв} + \gamma \times \textit{ценность } B)) +$

$+ (P^0_{лс}(R^0_{лс} + \gamma \times \textit{ценность } C))$;

$Q(A, 0) = (0{,}1 \times (0 + 0)) + (0{,}4 \times (-1{,}0 + 0)) + (0{,}3 \times (1{,}0 + 0))$.

Теперь вычислим значение *Q* для состояния **A** и действия 1:

$Q(A, 1) = (P^1_{лл} \times (R^1_{лл} + \gamma \times \textit{ценность } A)) + (P^1_{лв}(R^1_{лв} + \gamma \times \textit{ценность } B)) +$

$+ (P^1_{лс}(R^1_{лс} + \gamma \times \textit{ценность } C))$.

Обновим информацию в таблице *Q*:

Состояние	Действие	Ценность
A	0	−0,1
A	1	0,3
B	0	
B	1	
C	0	
C	1	

Обновим функцию ценности максимальным значением из *Q* (*s*, *a*).

Если взглянуть на предыдущую *Q*-функцию, *Q* (*A*, 1) имеет более высокую ценность, чем *Q* (*A*, 0), поэтому мы обновим ценность состояния **A**значением *Q* (*A*, 1):

Состояние	Ценность
A	0,3
B	
C	

Аналогичным образом вычисляется *Q* для всех пар состояний/действий, а функция ценности каждого состояния обновляется значением *Q*, которое имеет наивысшую ценность по всем состояниям/действиям. Обновленная функция ценности представлена ниже. Это результат первой итерации:

A 0,3	B –0,2		Состояние	Ценность
C 0,5			A	0,3
			B	–0,2
			C	0,5

Эти шаги повторяются несколько раз. А именно, повторяются шаги 2 и 3 (в каждой итерации при вычислении *Q* вместо функции ценности, которая была инициализирована случайно, используется обновленная функция ценности).

Результат второй итерации:

A 0,7	B –0,1		Состояние	Ценность
C 0,5			A	0,7
			B	–0,1
			C	0,5

Результат третьей итерации:

A 0,71	B –0,1		Состояние	Ценность
C 0,53			A	0,71
			B	–0,1
			C	0,53

Но когда нужно остановить итерации? Процесс останавливается тогда, когда изменение ценности между последовательными итерациями достаточно мало; присмотревшись ко второй и третьей итерации, вы увидите, что функция ценности почти не изменилась. При выполнении этого условия итерации прекращаются, а функция считается оптимальной функцией ценности.

Итак, мы нашли оптимальную функцию ценности. Как вывести оптимальную политику?

Все очень просто. Мы вычислим *Q*-функцию с финальной оптимальной функцией ценности. Допустим, вычисленная *Q*-функция выглядит так:

Состояние	Действие	Ценность
A	0	–0,53
A	1	0,98
B	0	–0,2
B	1	–0,3
C	0	0,2
C	1	0,01

Из этой *Q*-функции в каждом состоянии выбираются действия с максимальной ценностью. В состоянии **A** максимальной ценностью

обладает действие 1 — это оптимальная политика. Итак, если выполнить действие 1 в состоянии **A**, можно достичь **C** без посещения **B**.

Итерация по политикам

В отличие от итераций по ценности, итерация по политикам начинается со случайной политики, после чего находится функция ценности для этой политики; если функция ценности не оптимальна, то находится новая улучшенная политика. Процесс повторяется до тех пор, пока не будет найдена оптимальная политика.

Итерация по политикам состоит из двух шагов:

1. **Оценка политики**: функция ценности оценивается для случайно выбранной политики.
2. **Улучшение политики**: если эта функция ценности окажется неоптимальной, находится новая улучшенная политика:

Процесс итерации по политикам состоит из следующих шагов:

1. Сначала инициализируется случайная политика.
2. Для этой случайной политики находится функция ценности, которая проверяется на оптимальность; это называется оценкой политики.

3. Если функция ценности не оптимальна, находится новая улучшенная политика; это называется улучшением политики.

4. Эти шаги повторяются до тех пор, пока не будет найдена оптимальная политика.

Чтобы этот процесс стал более понятным, проведем его вручную шаг за шагом.

Рассмотрим ту же таблицу, которая была приведена в разделе *«Итерация по ценности»*. Наша цель — найти оптимальную политику:

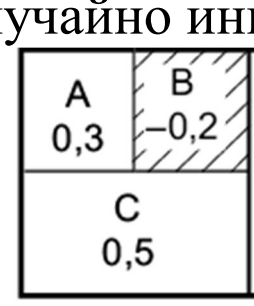
1. Инициализировать случайную функцию политики.

Инициализация случайной функции политики заключается в назначении случайных действий для каждого состояния, например:

A –> 0
B –> 1
C –> 0

2. Найти функцию ценности для случайно инициализированной политики.

Теперь необходимо найти функцию ценности с использованием случайно инициализированной политики.



Состояние	Ценность
A	0,3
B	–0,2
C	0,5

Теперь у нас имеется новая функция ценности, соответствующая случайно инициализированной политике. Вычислим новую политику с использованием новой функции ценности. Как это сделать? Механизм очень похож на тот, который мы использовали в разделе *«Итерация по ценности»*. Значение *Q* вычисляется для новой функции ценности, после чего для каждого состояния выполняются действия, обладающие максимальной ценностью.

Допустим, новая политика дает следующие значения:

A –> 0
B –> 1
C –> 1

Мы проверяем старую политику (то есть случайно инициализированную политику) и новую политику. Если они совпадают, значит, произошло схождение, то есть была найдена оптимальная политика. Если нет, старая политика (случайная) обновляется, и вся процедура повторяется с шага 2.

Голова идет кругом? Рассмотрим псевдокод:

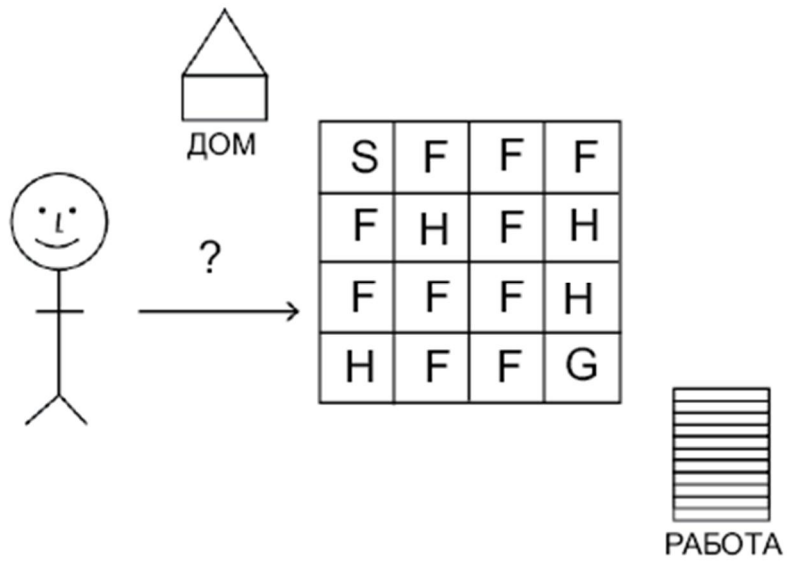
```
policy_iteration():
    инициализировать случайную политику
    for i in no_of_iterations:
        Q_value = value_function(random_policy)
```

```
            new_policy = максимальная пара состояния
/ действия из ценности Q
            if random_policy == new policy:
                break
            random_policy = new_policy
        return policy
```

Решение задачи о замерзшем озере

Даже если вы ничего не поняли из того, что я рассказал, не огорчайтесь — концепции будут применены к задаче о замерзшем озере.

Представьте, что между вашим домом и работой лежит замерзшее озеро; вы должны пройти по нему, чтобы добраться до работы. Но не все так просто! На ледяной поверхности есть полыньи, так что вы должны действовать осторожно и не угодить в них.



На этом рисунке:

• **S** — исходная позиция (дом).

• **F** — участок ледяной поверхности, по которому можно идти.

• **H** — полыньи, которых следует избегать.

• **G** — цель (работа).

А теперь используем агента, чтобы найти правильный путь на работу. Цель агента — найти оптимальный путь из **S** в **G**, не попав в **H**. Как он это сделает? Мы будем давать агенту 1 очко награды, если он правильно прошел по льду, и 0 очков за падение в полынью, чтобы агент мог определить правильное действие. Теперь агент пытается найти оптимальную политику, которая подразумевает выбор правильного пути, максимизирующего награду. Если агент максимизирует полыньи и достигает конечной точки, очевидно, награда максимизируется. Задачу можно смоделировать в форме MDP (см. выше). Формулировка MDP состоиз из следующих компонентов:

• **Состояния**: множество состояний. Здесь имеются 16 состояний (помеченных маленькими квадратиками на сетке).

• **Действия**: набор всех возможных действий (направо, налево, вверх, вниз); это четыре возможных действия, которые могут быть выбраны агентом в среде замерзшего озера.

• **Вероятности перехода**: вероятности перехода из одного состояния (**F**) в другое состояние (**H**) выполнением действия *a*.

• **Вероятности наград**: вероятность получения награды при переходе из одного состояния (**F**) в другое состояние (**H**) выполнением действия *a*.

Теперь целью становится решение задачи MDP. Решение требует нахождения оптимальных политик. Сейчас я представлю три специальные функции:

• **Функция политики**: указывает, какое действие должно выполняться в каждом состоянии.

• **Функция ценности**: указывает желательное состояние.

• Q**-функция**: указывает, насколько желательно действие в конкретном состоянии.

Когда мы говорим о «хороших» функциях, что это означает? То, насколько хорошо функция максимизирует награды.

Затем функция ценности и *Q*-функция представляются с использованием специального уравнения, называемого уравнение оптимальности Беллмана. Решив это уравнение, мы найдем оптимальную политику.

Здесь под решением уравнения понимается нахождение правильной функции ценности и политики. Если мы найдем правильную функцию ценности и политику, это будет оптимальный путь, который обеспечит максимальные награды.

Для решения уравнения оптимальности Беллмана будет использоваться специальный метод, называемый динамическим программированием. Чтобы применить DP, динамика модели должна быть известна заранее, что, по сути, означает, что вероятности перехода и вероятности наград среды модели должны быть известны заранее. Так как мы знаем динамику модели, здесь можно применить динамическое программирование. Для поиска оптимальной политики используются два специальных DP-алгоритма:

• итерация по ценности;

• итерация по политикам.

Итерация по ценности

Проще говоря, при итерации по ценности мы сначала инициализируем функцию ценности некоторым случайным значением. Почти наверняка случайное значение не будет оптимальным, поэтому мы перебираем все состояния и находим новую функцию ценности; итерация прекращается тогда, когда будет найдена оптимальная функция ценности. Из найденной функции легко извлекается оптимальная политика.

Рассмотрим итерацию по ценности на примере задачи о замерзшем озере.

Прежде всего импортируйте необходимые библиотеки:

```
import gym
import numpy as np
```

Затем создайте среду замерзшего озера с использованием OpenAI Gym:

```
env = gym.make('FrozenLake-v0')
```

Начнем с исследования среды.

Количество состояний в среде равно 16 (сетка 4 × 4):

```
print(env.observation_space.n)
```

Количество действий в среде равно 4 (вверх, вниз, налево и направо):

```
print(env.observation_space.n)
```

Определим функцию `value_iteration()`, которая возвращает оптимальную функцию ценности (таблицу ценности). Прежде всего разберемся, как работает функция шаг за шагом, а потом рассмотрим весь код.

Сначала случайная таблица ценности инициализируется 0 для всех состояний; также инициализируется количество итераций:

```
value_table = np.zeros(env.observation_space.n)
no_of_iterations = 100000
```

Затем в начале каждой итерации `value_table` копируется в `updated_value_table`:

```
for i in range(no_of_iterations):
    updated_value_table = np.copy(value_table)
```

Затем вычисляется *Q*-таблица и из нее выбирается максимальная пара состояние/действие, обладающая наибольшей ценностью в таблице.

Чтобы лучше понять код, воспользуемся решенным ранее примером; мы вычислили *Q* для состояния **A** и действия 1 в предыдущем примере:

Q (*A*, 1) = (0,3 × (0+0)) + (0,1 × (–1,0 + 0)) + (0,5 + (1,0 + 0));

Q (*A*, 1) = 0,5.

Вместо того чтобы создавать *Q*-таблицу для каждого состояния, мы создаем список с именем `Q_value`, а затем для каждого действия в состоянии создается список `next_states_rewards`, в котором хранится `Q_value` для состояния следующего перехода. Затем значения `next_state_rewards` суммируются и присоединяются к `Q_value`.

Обратимся к предыдущему примеру с состоянием **A** и действием 1. (0,3 × (0 + 0)) — награда следующего состояния для состояния **A**, а (0,1 × (–1,0 + 0)) — награда следующего состояния для состояния перехода **B**. (0,5 + (1,0 + 0)) — награда следующего состояния для состояния перехода **C**. Все эти результаты суммируются в переменной `next_state_reward` и присоединяются к `Q_value`, которое будет равно 0,5.

При вычислении `next_state_rewards` для всех действий в состоянии и присоединении результата к *Q* мы берем максимальное значение *Q* и обновляем его как ценность состояния:

```
for state in range(env.observation_space.n):
    Q_value = []
    for action in range(env.action_space.n):
        next_states_rewards = []
```

```
            for next_sr in env.P[state][action]:
                trans_prob, next_state, reward_prob,
_ = next_sr
                next_states_rewards.append((trans_pr
ob * (reward_prob + gamma *
  updated_value_table[next_state])))
              Q_value.append(np.sum(next_states_rewa
rds))
            # Взять максимальное значение Q и
обновить его как ценность
            # состояния.
            value_table[state] = max(Q_value)
```

Затем мы проверяем, было ли достигнуто схождение, то есть очень ли мала разность между таблицей ценности и обновленной таблицей ценности. Как определить, что она очень мала? Мы определяем переменную `threshold`, а затем проверяем, превышает ли разность этот порог; если разность меньше, то цикл прерывается, а функция ценности возвращается как оптимальная функция ценности:

```
  threshold = 1e-20
  if (np.sum(np.fabs(updated_value_table -
value_table)) <= threshold):
      print ('Value-iteration converged at
iteration# %d.' %(i+1))
      break
```

А теперь рассмотрим полный код `value_iteration()` для лучшего понимания:

```
  def value_iteration(env, gamma = 1.0):
      value_table =
np.zeros(env.observation_space.n)
      no_of_iterations = 100000
      threshold = 1e-20

      for i in range(no_of_iterations):
          updated_value_table =
np.copy(value_table)

          for state in
range(env.observation_space.n):
              Q_value = []

              for action in
range(env.action_space.n):
                  next_states_rewards = []
```

```
                    for next_sr in
env.P[state][action]:
                        trans_prob, next_state,
reward_prob, _ = next_sr
                        next_states_rewards.append((
trans_prob * (reward_prob +
  gamma * updated_value_table[next_state])))

                    Q_value.append(np.sum(next_state
s_rewards))
                value_table[state] = max(Q_value)
            if (np.sum(np.fabs(updated_value_table -
value_table)) <=
  threshold):
                print ('Value-iteration converged
at iteration# %d.' %(i+1))
                break
    return value_table, Q_value
```

Таким образом, мы можем вычислить `optimal_value_function` с использованием `value_iteration`:

```
  optimal_value_function =
value_iteration(env=env,gamma=1.0)
```

После того как функция `optimal_value_function` будет найдена, как извлечь из нее оптимальную политику? Мы вычисляем значение *Q*, используя действие с оптимальной ценностью, и выбираем действия, обладающие наивысшим значением *Q* для каждого состояния, формируя оптимальную политику. Эта задача решается функцией `extract_policy()`; мы разберем ее шаг за шагом.

Сначала определяется случайная политика; мы присвоим ей значение 0 для всех состояний:

```
  policy = np.zeros(env.observation_space.n)
```

Далее для каждого состояния строится таблица `Q_table` и для каждого действия в этом состоянии вычисляется значение *Q*, которое добавляется в `Q_table`:

```
  for state in range(env.observation_space.n):
        Q_table = np.zeros(env.action_space.n)
        for action in range(env.action_space.n):
            for next_sr in env.P[state][action]:
                trans_prob, next_state,
reward_prob, _ = next_sr
                Q_table[action] += (trans_prob *
(reward_prob + gamma *
  value_table[next_state]))
```

Затем мы выберем политику для состояния как действие с наибольшим значением *Q*:

```
policy[state] = np.argmax(Q_table)
```

Рассмотрим полный код функции:

```
def extract_policy(value_table, gamma = 1.0):

    policy = np.zeros(env.observation_space.n)
    for state in range(env.observation_space.n):
        Q_table = np.zeros(env.action_space.n)
        for action in range(env.action_space.n):
            for next_sr in env.P[state][action]:
                trans_prob, next_state,
reward_prob, _ = next_sr
                Q_table[action] += (trans_prob *
(reward_prob + gamma *
value_table[next_state]))
        policy[state] = np.argmax(Q_table)
    return policy
```

Затем оптимальная политика `optimal_policy` вычисляется следующим образом:

```
optimal_policy =
extract_policy(optimal_value_function, gamma=1.0)
```

В результате мы получим вывод — оптимальную политику, то есть действия, которые должны выполняться в каждом состоянии:

```
array([0., 3., 3., 3., 0., 0., 0., 0., 3., 1.,
0., 0., 0., 2., 1., 0.])
```

А теперь я приведу полный код программы:

```
import gym
import numpy as np
env = gym.make('FrozenLake-v0')

def value_iteration(env, gamma = 1.0):
    value_table =
np.zeros(env.observation_space.n)
    no_of_iterations = 100000
    threshold = 1e-20
    for i in range(no_of_iterations):
        updated_value_table =
np.copy(value_table)
        for state in
range(env.observation_space.n):
            Q_value = []
            for action in
range(env.action_space.n):
```

```
                    next_states_rewards = []
                    for next_sr in
env.P[state][action]:
                        trans_prob, next_state,
reward_prob, _ = next_sr
                        next_states_rewards.append((
trans_prob * (reward_prob +
  gamma * updated_value_table[next_state])))
                    Q_value.append(np.sum(next_state
s_rewards))
                value_table[state] = max(Q_value)
            if (np.sum(np.fabs(updated_value_table -
value_table)) <=
  threshold):
                print ('Value-iteration converged
at iteration# %d.' %(i+1))
                break
        return value_table

  def extract_policy(value_table, gamma = 1.0):
        policy = np.zeros(env.observation_space.n)
        for state in range(env.observation_space.n):
            Q_table = np.zeros(env.action_space.n)
            for action in range(env.action_space.n):
                for next_sr in env.P[state][action]:
                    trans_prob, next_state,
reward_prob, _ = next_sr
                    Q_table[action] += (trans_prob *
(reward_prob + gamma *
  value_table[next_state]))
            policy[state] = np.argmax(Q_table)
        return policy

  optimal_value_function =
value_iteration(env=env,gamma=1.0)
  optimal_policy =
extract_policy(optimal_value_function, gamma=1.0)

  print(optimal_policy)
```

Итерация по политикам

Итерация по политикам начинается с инициализации случайной политики. Затем оцениваются случайные политики, которые были инициализированы, — насколько они хороши или плохи. Но как

оценивать политики? Для этого мы будем вычислять для случайно инициализированных политик функции ценности. Если результат окажется плохим, нужно искать новую политику. Этот процесс повторяется до тех пор, пока не будет найдена хорошая политика. А теперь посмотрим, как решить задачу о замерзшем озере с помощью итерации по политикам.

Прежде чем рассматривать итерацию по политикам, сначала разберемся в том, как вычислить функцию ценности для заданной политики.

Таблица `value_table` инициализируется нулями по количеству состояний:

```
value_table = np.zeros(env.nS)
```

Затем для каждого состояния из политики определяется действие и вычисляется функция ценности для этого действия и состояния:

```
        updated_value_table =
np.copy(value_table)
        for state in range(env.nS):
            action = policy[state]
            value_table[state] = sum([trans_prob
* (reward_prob + gamma *
  updated_value_table[next_state])
                                      for trans_prob,
next_state, reward_prob, _ in
  env.P[state][action]])
```

Итерация прерывается, когда разность между `value_table` и `updated_value_table` падает ниже порога `threshold`:

```
  threshold = 1e-10
  if (np.sum((np.fabs(updated_value_table -
value_table))) <= threshold):
      break
```

Полный код функции:

```
  def compute_value_function(policy, gamma=1.0):
      value_table = np.zeros(env.nS)
      threshold = 1e-10
      while True:
          updated_value_table =
np.copy(value_table)
          for state in range(env.nS):
              action = policy[state]
              value_table[state] = sum([trans_prob
* (reward_prob + gamma *
  updated_value_table[next_state])
```

```
                                for trans_prob,
next_state, reward_prob, _ in
  env.P[state][action]])
            if (np.sum((np.fabs(updated_value_table
- value_table))) <=
  threshold):
                break
    return value_table
```

А теперь разберемся, как выполняется итерация по политикам, шаг за шагом.

Сначала `random_policy` инициализируется массивом `NumPy`, заполненным нулями, размер которого соответствует количеству состояний:

```
  random_policy =
np.zeros(env.observation_space.n)
```

Затем для каждой итерации вычисляется `new_value_function` в соответствии со случайной политикой:

```
  new_value_function =
compute_value_function(random_policy, gamma)
```

Для извлечения политики будет использоваться вычисленное значение `new_value_function`. Здесь применяется та же функция `extract_policy`, что и в итерации по ценности:

```
  new_policy = extract_policy(new_value_function,
gamma)
```

Теперь мы проверяем, было ли достигнуто схождение, то есть была ли найдена оптимальная политика, для чего `random_policy` сравнивается с новой политикой `new_policy`. Если они не отличаются, то итерация прерывается; в противном случае `random_policy` обновляется `new_policy`:

```
  if (np.all(random_policy == new_policy)):
      print ('Policy-Iteration converged at step
%d.' %(i+1))
      break
  random_policy = new_policy
```

Полный код функции `policy_iteration`:

```
  def policy_iteration(env,gamma = 1.0):
      random_policy =
np.zeros(env.observation_space.n)
      no_of_iterations = 200000
      gamma = 1.0
      for i in range(no_of_iterations):
          new_value_function =
compute_value_function(random_policy, gamma)
```

```
            new_policy =
extract_policy(new_value_function, gamma)
            if (np.all(random_policy ==
new_policy)):
                print ('Policy-Iteration converged
at step %d.' %(i+1))
                break
            random_policy = new_policy
        return new_policy
```

Для получения оптимальной политики `optimal_policy` можно воспользоваться `policy_iteration`:

```
optimal_policy = policy_iteration(env, gamma =
1.0)
```

В результате мы получаем вывод — оптимальную политику, то есть действия, которые должны выполняться в каждом состоянии:

```
array([0., 3., 3., 3., 0., 0., 0., 0., 3., 1.,
0., 0., 0., 2., 1., 0.])
```

Полный код программы:

```
import gym
import numpy as np

env = gym.make('FrozenLake-v0')

def compute_value_function(policy, gamma=1.0):
    value_table = np.zeros(env.nS)
    threshold = 1e-10
    while True:
        updated_value_table =
np.copy(value_table)
        for state in range(env.nS):
            action = policy[state]
            value_table[state] = sum([trans_prob
* (reward_prob + gamma *
updated_value_table[next_state])
                                      for trans_prob,
next_state, reward_prob, _ in
env.P[state][action]])
        if (np.sum((np.fabs(updated_value_table
- value_table))) <=
threshold):
            break
    return value_table

def extract_policy(value_table, gamma = 1.0):
```

```
    policy = np.zeros(env.observation_space.n)
    for state in range(env.observation_space.n):
        Q_table = np.zeros(env.action_space.n)
        for action in range(env.action_space.n):
            for next_sr in env.P[state][action]:
                trans_prob, next_state,
reward_prob, _ = next_sr
                Q_table[action] += (trans_prob *
(reward_prob + gamma *
  value_table[next_state]))
        policy[state] = np.argmax(Q_table)
    return policy

def policy_iteration(env,gamma = 1.0):
    random_policy =
np.zeros(env.observation_space.n)
    no_of_iterations = 200000
    gamma = 1.0
    for i in range(no_of_iterations):
        new_value_function =
compute_value_function(random_policy, gamma)
        new_policy =
extract_policy(new_value_function, gamma)
        if (np.all(random_policy ==
new_policy)):
            print ('Policy-Iteration converged
at step %d.' %(i+1))
            break
        random_policy = new_policy
    return new_policy

print (policy_iteration(env))
```

Таким образом, оптимальная политика, которая определяет действие, выполняемое в каждом состоянии, может быть вычислена с использованием итераций по ценности и политикам для решения задачи о замерзшем озере.

Итоги

В этой главе вы узнали, что такое марковские цепи и марковские процессы и как задачи RL представляются с использованием марковского процесса принятия решений (MDP). Также было рассмотрено уравнение Беллмана, которое было решено для вывода оптимальной политики средствами динамического программирования.

В **главе 4** будет описан поиск по дереву методом Монте-Карло, а также место этого метода в построении интеллектуальных игр.

Вопросы

1. Что такое «марковское свойство»?
2. Для чего нужен марковский процесс принятия решений?
3. В какой ситуации немедленная награда предпочтительна?
4. Что такое «поправочный коэффициент»?
5. Для чего используется функция Беллмана?
6. Как выводится уравнение Беллмана для *Q*-функции?
7. Как связаны между собой функция ценности и *Q*-функция?
8. Чем итерация по ценности отличается от итерации по политикам?

Дополнительные источники

Материалы гарвардских лекций по MDP: `http://am121.seas.harvard.edu/site/wp-content/uploads/2011/03/MarkovDecisionProcesses-HillierLieberman.pdf.`

4. Методы Монте-Карло в играх

Монте-Карло — один из самых популярных и широко используемых алгоритмов в различных областях: от физики и механики до информатики. Он применяется в **обучении с подкреплением** (**RL**), когда модель среды неизвестна. В **главе 3** было рассмотрено использование метода **динамического программирования**(**DP**) для нахождения оптимальной политики при известной динамике модели (то есть поиске вероятностей наград и переходов). Но как определить оптимальную политику, если динамика модели неизвестна? В таких случаях используется алгоритм Монте-Карло; он чрезвычайно эффективен для нахождения оптимальных политик без полной информации о среде.

В этой главе рассмотрены следующие темы:

- Методы Монте-Карло.
- Прогнозирование методами Монте-Карло.
- Игра в блек-джек по стратегии Монте-Карло.
- Управляющие методы Монте-Карло.
- MC-ES.
- Метод Монте-Карло с привязкой к политике.
- Метод Монте-Карло без привязки к политике.

Метод Монте-Карло

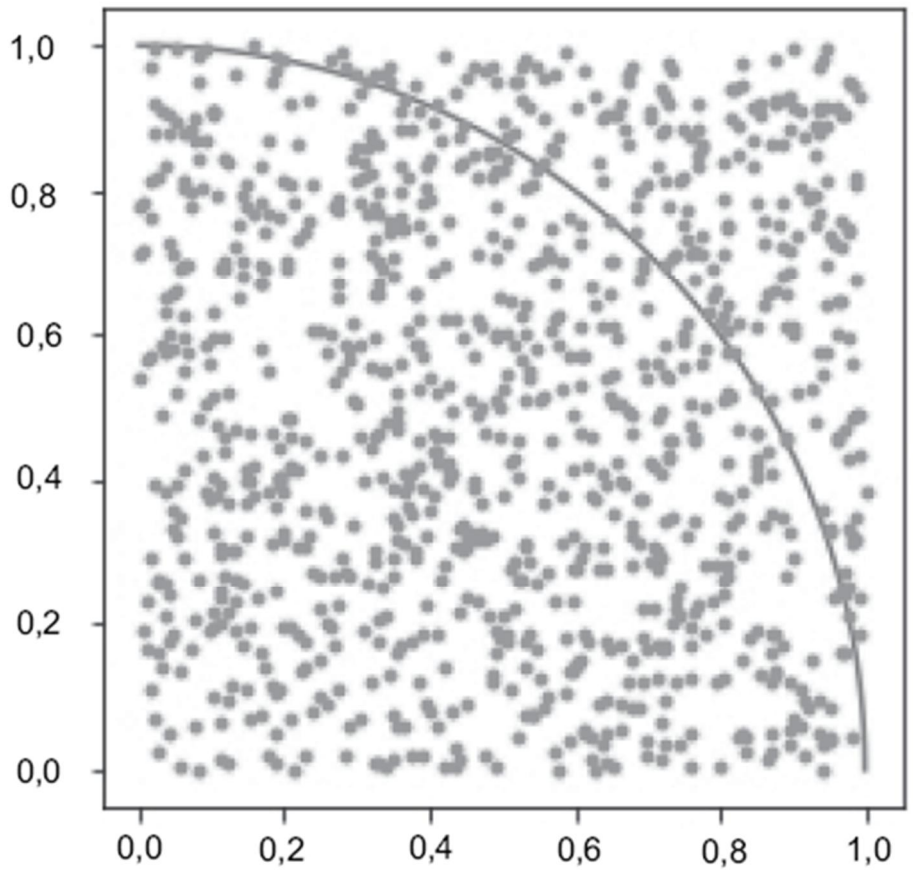
Метод Монте-Карло представляет собой статистический метод нахождения приближенных решений с использованием случайной выборки; иначе говоря, этот метод аппроксимирует вероятный результат за счет отработки множественных испытаний. Простой пример поможет вам лучше понять его суть.

Забавный факт

Метод Монте-Карло обязан названием дяди Станислава Улама, который часто брал взаймы у родственников на азартные игры в казино Монте-Карло.

Оценка значения π методом Монте-Карло

Представим четверть круга, помещенную в квадрат, как показано на следующем рисунке, и сгенерируем случайные точки внутри квадрата. Одни точки попадают внутрь круга, другие — оказываются за его пределами:



Можно написать следующую пропорцию:

$$\frac{\text{Площадь круга}}{\text{Площадь квадрата}} = \frac{\text{Количество точек внутри круга}}{\text{Количество точек внутри квадрата}}.$$

Мы знаем, что площадь круга равна πr^2, а площадь квадрата равна a^2:

$$\frac{\pi r^2}{a^2} = \frac{\text{Количество точек внутри круга}}{\text{Количество точек внутри квадрата}}.$$

Будем считать, что радиус круга равен 1/2, а сторона квадрата равна 1. Это позволяет произвести замену:

$$\frac{\pi\left(\frac{1}{2}\right)^2}{1^2} = \frac{\text{Количество точек внутри круга}}{\text{Количество точек внутри квадрата}}.$$

Приходим к следующему результату:

$$\pi = 4 \times \frac{\text{Количество точек внутри круга}}{\text{Количество точек внутри квадрата}}.$$

Последовательность действий для оценки π очень проста:

1. Сначала генерируются случайные точки внутри квадрата.

2. Количество точек, находящихся внутри круга, вычисляется по формуле $x^2 + y^2 <= размер$.

3. Приближенное значение π вычисляется умножением 4 на частное от деления количества точек внутри круга на количество точек внутри квадрата.

4. Увеличение количества случайных точек (размера выборки) повышает качество аппроксимации.

А теперь посмотрим, как это делается на Python, шаг за шагом. Прежде всего импортируются необходимые библиотеки:

```
import numpy as np
import math
import random
import matplotlib.pyplot as plt
%matplotlib inline
```

Затем инициализируется размер квадрата и количества точек внутри квадрата и круга. Также указывается размер выборки, то есть количество генерируемых случайных точек. Мы определяем область `arc`, которая фактически представляет четверть круга:

```
square_size = 1
points_inside_circle = 0
points_inside_square = 0
sample_size = 1000
arc = np.linspace(0, np.pi/2, 100)
```

Затем определяется функция `generate_points()`, которая генерирует случайные точки внутри квадрата:

```
def generate_points(size):
    x = random.random()*size
    y = random.random()*size
    return (x, y)
```

Следующая функция `is_in_circle()` проверяет, находится ли сгенерированная точка внутри круга:

```
def is_in_circle(point, size):
    return math.sqrt(point[0]**2 + point[1]**2)
<= size
```

Третья функция вычисляет приближенное значение π:

```
def compute_pi(points_inside_circle,
points_inside_square):
    return 4 * (points_inside_circle /
points_inside_square)
```

Далее в цикле генерируются случайные точки внутри квадрата и увеличивается переменная `points_inside_square`, после чего проверяется, лежит ли сгенерированная точка внутри круга. Если да, — переменная `points_inside_circle` увеличивается:

```
plt.axes().set_aspect('equal')
plt.plot(1*np.cos(arc), 1*np.sin(arc))
```

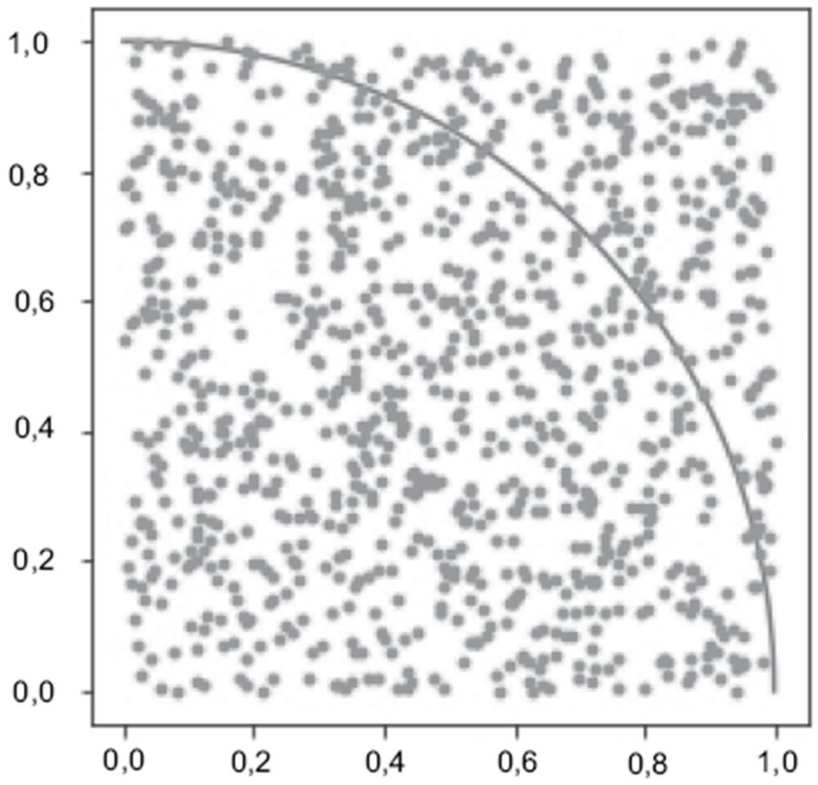
```
    for i in range(sample_size):
        point = generate_points(square_size)
        plt.plot(point[0], point[1], 'c.')
        points_inside_square += 1
        if is_in_circle(point, square_size):
            points_inside_circle += 1
```

Для вычисления оценки π используется функция `compute_pi()`:

```
print("Approximate value of pi is {}"
.format(calculate_pi(points_inside_circle,
points_inside_square)))
```

При запуске программы вы получите следующий результат:

```
Approximate value of pi is 3.144
```



Полный код программы:

```
import numpy as np
import math
import random
import matplotlib.pyplot as plt
%matplotlib inline
square_size = 1
points_inside_circle = 0
points_inside_square = 0
sample_size = 1000
arc = np.linspace(0, np.pi/2, 100)

def generate_points(size):
    x = random.random()*size
    y = random.random()*size
    return (x, y)

def is_in_circle(point, size):
    return math.sqrt(point[0]**2 + point[1]**2)
<= size
```

```
    def compute_pi(points_inside_circle,
points_inside_square):
        return 4 * (points_inside_circle /
points_inside_square)

    plt.axes().set_aspect('equal')
    plt.plot(1*np.cos(arc), 1*np.sin(arc))

    for i in range(sample_size):
        point = generate_points(square_size)
        plt.plot(point[0], point[1], 'c.')
        points_inside_square += 1
        if is_in_circle(point, square_size):
            points_inside_circle += 1

    print("Approximate value of pi is {}"
    .format(calculate_pi(points_inside_circle,
points_inside_square)))
```

Метод Монте-Карло был использован для вычисления приближенного значения `pi` с использованием случайной выборки, при которой внутри квадрата генерируются случайные точки. Чем больше размер выборки, тем более качественной будет аппроксимация. А теперь вы увидите, как методы Монте-Карло применяются в области обучения с подкреплением.

Прогнозирование методом Монте-Карло

В области динамического программирования для решения задач марковского процесса принятия решений (MDP) используются методы итераций по ценности и итераций по политикам. В обоих методах для нахождения оптимальной политики необходима информация о вероятностях переходов и наград. Но как решать задачи MDP, если вы не располагаете информацией о вероятностях наград и переходов? В таких случаях применяется метод Монте-Карло, для которого необходимы только выборочные последовательности состояний/действий/наград. Это метод актуален только в решении эпизодических задач и относится к категории алгоритмов обучения без модели.

Основной принцип метода Монте-Карло очень прост. Помните, как в **главе 3** мы определяли оптимальную функцию ценности и как выводили оптимальную политику?

Функция ценности фактически определяет ожидаемый возврат для состояния S с политикой π. Здесь вместо ожидаемого возврата используется средний возврат.

Примечание

При прогнозировании методом Монте-Карло функция ценности аппроксимируется средним возвратом вместо ожидаемого возврата.

При прогнозировании методом Монте-Карло можно оценить функцию ценности при любой заданной политике. Последовательность действий прогнозирования методом Монте-Карло очень проста:

1. Функции ценности присваивается случайное значение.
2. Инициализируется пустой список `return` для хранения возвратов.
3. Затем для каждого состояния в эпизоде вычисляется возврат.
4. Возврат присоединяется к списку `return`.
5. Вычисляется среднее значение `return`, которое становится функцией ценности.

Следующая блок-схема представляет суть прогнозирования методом Монте-Карло еще нагляднее:

Алгоритмы прогнозирования методом Монте-Карло делятся на две категории:

- метод Монте-Карло с первым посещением;
- метод Монте-Карло с каждым посещением.

Метод Монте-Карло с первым посещением

Как было показано выше, в методе Монте-Карло функция ценности аппроксимируется вычислением среднего возврата. В методе *с первым посещением* возврат усредняется только при первом посещении состояния в эпизоде. Представьте агента для игры «Змеи и лестницы»: существует достаточно высокая вероятность того, что агент вернется в прежнее состояние, если попадет на штрафное поле со змеей. Так, при повторном посещении состояния средний возврат не будет вычисляться.

Метод Монте-Карло с каждым посещением

В методе *с каждым посещением*возврат усредняется при каждом посещении состояния в эпизоде. Вернемся к примеру с игрой «Змеи и лестницы»: если агент возвращается в некоторое состояние после попадания на штрафное поле, это можно учитывать в среднем возврате, несмотря на повторность посещения состояния.

Игра в блек-джек по стратегии Монте-Карло

Чтобы лучше понять метод Монте-Карло, применим его к игре блек-джек — популярной карточной игре, в которую обычно играют в казино. Цель игры — набрать количество очков, которое максимально близко к 21, но не превышает его. За валетов, дам и королей начисляется по 10 очков. Тузы дают 1 очко или 11 очков по выбору игрока. Количество очков за остальные карты равно их номиналу.

Правила игры очень просты:

• В игре участвует один или несколько игроков и крупье (казино, дилер).

• Каждый игрок играет только против крупье, но не против других игроков.

• Каждому игроку раздаются две карты. Обе — в открытую, то есть видны другим игрокам.

• Крупье тоже получает две карты: одну в открытую, а другую — в закрытую. Таким образом, игрокам видна лишь одна из карт крупье.

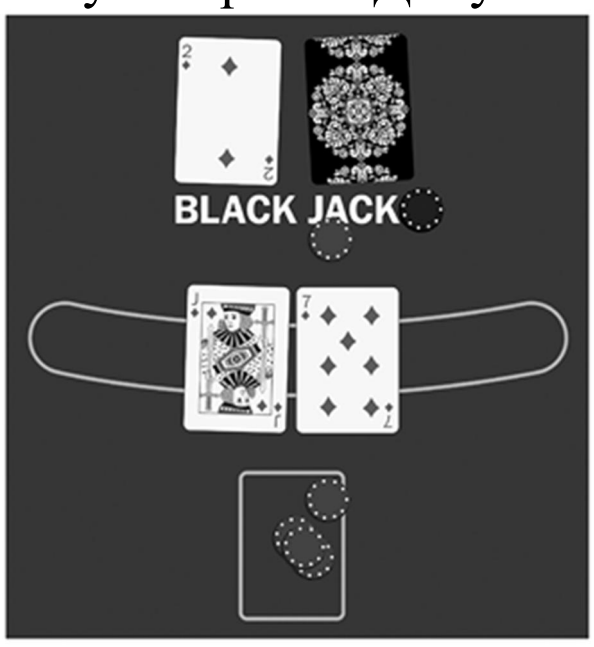
• Если игрок набирает 21 очко на двух картах (допустим, игрок получил валета и туза, что дает 10 + 11 = 21), игрок побеждает; такая ситуация называется блек-джек.

• Если крупье тоже набирает 21 очко на двух картах, раздача заканчивается вничью.

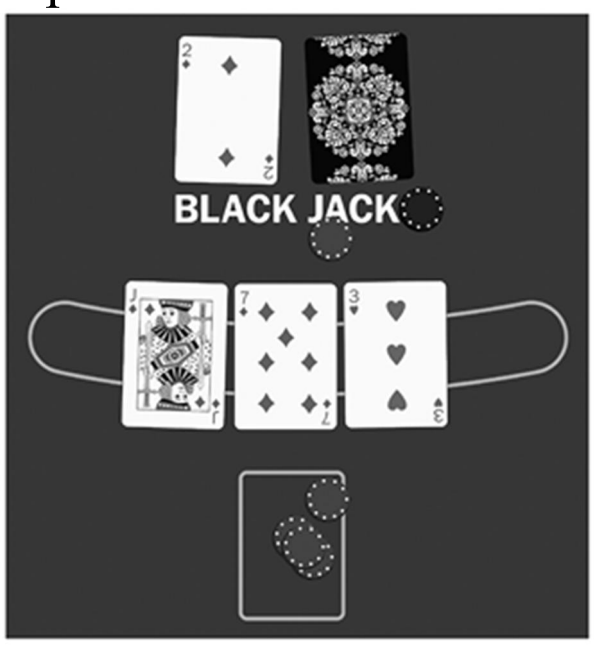
• В каждом раунде игрок решает, хочет ли он получить еще одну карту.

• Если сумма очков на картах игрока превышает 21, происходит «перебор», а крупье выигрывает.

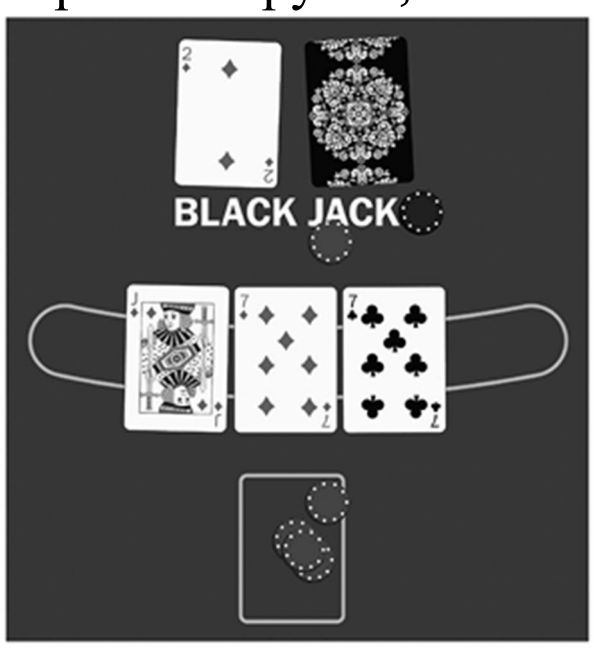
Чтобы вы лучше поняли, как играть в блек-джек, просто разыграем пробную партию. Допустим, вы — игрок, а я — крупье:



Здесь в игре участвует один игрок и крупье. Обе стороны получают по две карты. Карты игрока сдаются в открытую (видны сопернику), тогда как крупье получает одну карту в открытую, а другую в закрытую. В первом раунде вам достается валет и семерка (10 + 7 = 17), а я как крупье показываю вам только одну карту — двойку. Вторая карта лежит рубашкой вверх. Теперь вы должны решить, взять ли еще одну карту или остановиться. Если вы захотите взять карту и это будет тройка, вы наберете 10 + 7 + 3 = 20 очков, что еще ближе к 21, чем 17:



Но если вам придет семерка, количество очков составит 10 + 7 + 7 = 24, что больше 21. Перебор, вы проиграли. Если же вы решите остаться с исходными картами, у вас будет всего 17 очков. После этого проверяется сумма очков на картах у крупье. Если она больше 17 и не превышает 21, выигрывает крупье; в остальных случаях выигрываете вы:



Награды:

• +1, если игрок выигрывает.

• –1, если игрок проигрывает.

• 0, если партия завершилась вничью.

Возможные действия:

• **Взять карту** (hit).

• **Не брать карту** (stand).

Игрок должен принять решение о количестве очков за туза. Если сумма очков на картах игрока равна 10, он берет карту и это оказывается туз, он считается за 11 очков, 10 + 11 = 21. Но если сумма очков игрока равна 15, пришедшего туза нельзя считать за 11 очков, так как это приведет к перебору: 15 + 11 = 26. Если игроку достался туз, который

можно посчитать за 11 очков без перебора, будем называть такого туза **полезным**. Если же присваивание тузу 11 очков приведет к перебору, туз называется **бесполезным**.

Посмотрим, как реализовать игру блек-джек на базе алгоритма Монте-Карло с первым посещением. Начнем с импортирования необходимых библиотек:

```
import gym
from matplotlib import pyplot
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from collections import defaultdict
from functools import partial
%matplotlib inline
plt.style.use('ggplot')
```

Затем создадим среду игры блек-джек с использованием OpenAI Gym:

```
env = gym.make('Blackjack-v0')
```

Затем определим функцию политики, которая получает текущее состояние и проверяет его. Если оно больше либо равно `20`, функция возвращает `0`, а если меньше — возвращает `1`. Другими словами, если количество очков меньше `20`, то функция берет карту (1), а если больше либо равно 20 — не берет (0):

```
def sample_policy(observation):
    score, dealer_score, usable_ace =
observation
    return 0 if score >= 20 else 1
```

Перейдем к генерированию эпизодов, представляющих отдельный раунд игры. Сначала мы разберем код шаг за шагом, а потом я приведу его полностью.

Состояния, действия и награды определяются в виде списков (`states`, `actions` и `rewards` соответственно), после чего среда инициализируется вызовом `env.reset` и сохраняется в переменной `observation`:

```
states, actions, rewards = [], [], []
observation = env.reset()
```

Затем, пока не будет достигнуто завершающее состояние (конец эпизода), происходит следующее:

1. `observation` присоединяется к списку состояний `states`:

```
states.append(observation)
```

2. Новое действие создается функцией `sample_policy` и присоединяется к списку действий:

```
action = sample_policy(observation)
actions.append(action)
```

3. Затем для каждого шага в среде сохраняются значения `state`, `reward`и `done` (признак достижения завершающего состояния), а награды присоединяются к списку наград:

```
observation, reward, done, info = env.step(action)
rewards.append(reward)
```

4. Если достигнуто завершающее состояние, цикл прерывается:

```
if done:
    break
```

5. Полный код функции `generate_episode`:

```
def generate_episode(policy, env):
    states, actions, rewards = [], [], []
    observation = env.reset()
    while True:
        states.append(observation)
        action = policy(observation)
        actions.append(action)
        observation, reward, done, info = env.step(action)
        rewards.append(reward)
        if done:
            break

    return states, actions, rewards
```

С генерированием эпизодов мы разобрались. Как же провести игру? Для этого необходимо знать ценность каждого состояния. Для получения ценности каждого состояния будет использоваться метод с первым посещением.

Сначала программа инициализирует пустую таблицу в виде словаря для хранения ценностей всех состояний:

```
value_table = defaultdict(float)
```

Затем для заданного количества эпизодов выполняются следующие действия:

1. Генерируется эпизод с сохранением состояний и наград; переменная `returns` для суммы наград получит значение 0:

```
states, _, rewards = generate_episode(policy, env)
returns = 0
```

2. Затем для каждого шага сохранятся награды в переменной `R`, состояния — в переменной `S`, а значение `returns` вычислится как накопленная сумма наград:

```
for t in range(len(states) - 1, -1, -1):
    R = rewards[t]
    S = states[t]
```

```
    returns += R
```

3. Метод с первым посещением будет реализован через проверку, посещается ли состояние впервые, если да, то вычисляется среднее значение `returns` и результат присваивается как ценность состояния:

```
  if S not in states[:t]:
    N[S] += 1
    value_table[S] += (returns - V[S]) / N[S]
```

4. Полный код функции поможет вам лучше понять ее логику:

```
  def first_visit_mc_prediction(policy, env,
n_episodes):
    value_table = defaultdict(float)
    N = defaultdict(int)

    for _ in range(n_episodes):
        states, _, rewards =
generate_episode(policy, env)
        returns = 0
        for t in range(len(states) - 1, -1, -1):
            R = rewards[t]
            S = states[t]
            returns += R
            if S not in states[:t]:
                N[S] += 1
                value_table[S] += (returns -
V[S]) / N[S]
    return value_table
```

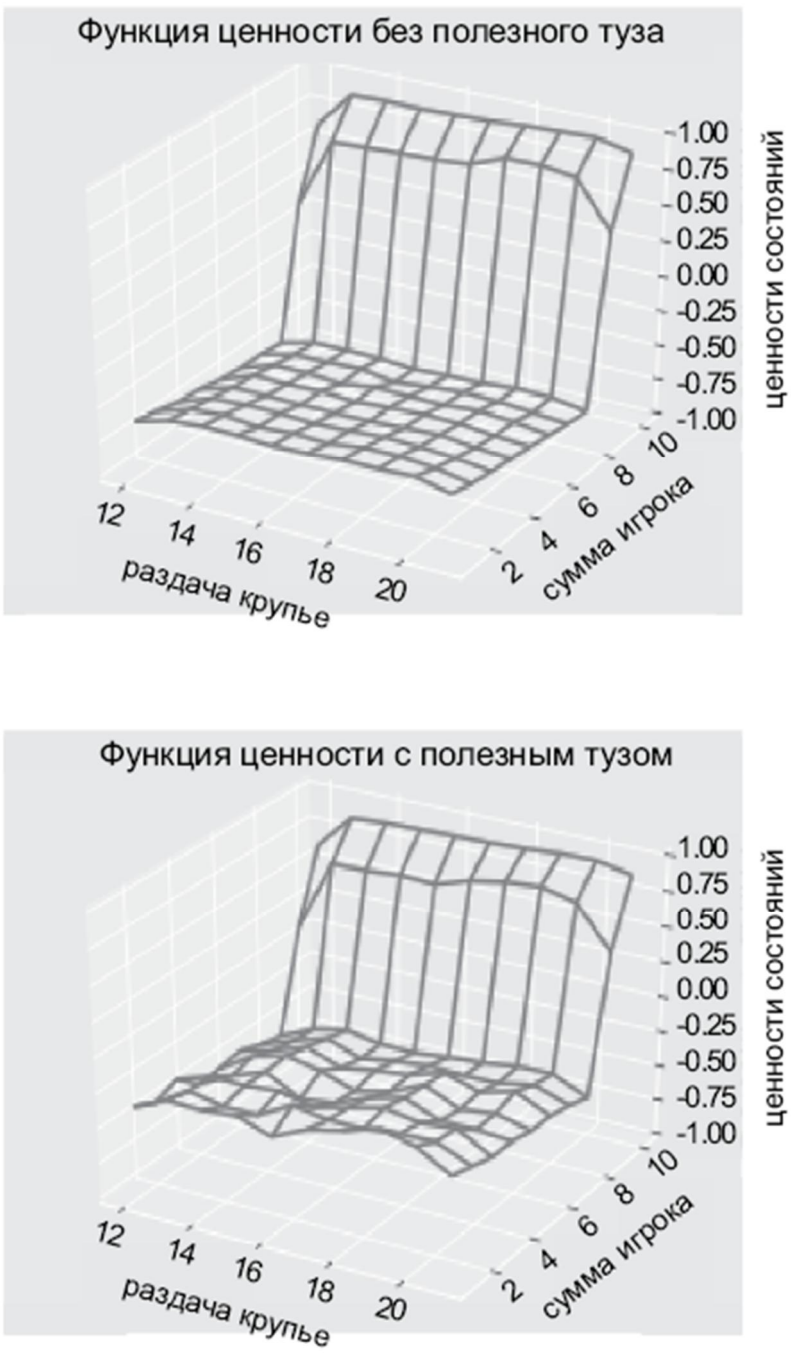
5. Далее последует определение ценности каждого состояния:

```
  value = first_visit_mc_prediction(sample_policy,
env,
  n_episodes=500000)
```

Для примера выведем ценности нескольких состояний:

```
print(value)
defaultdict(float,
            {(4, 1, False): -1.024292170184644,
             (4, 2, False): -1.8670191351012455,
             (4, 3, False): 2.211363314854649,
             (4, 4, False): 16.903201033000823,
             (4, 5, False): -5.786238030898542,
             (4, 6, False): -16.218211752577602,
```

Также можно построить график ценности состояний, чтобы посмотреть, как прошло схождение:



Полный код выглядит так:

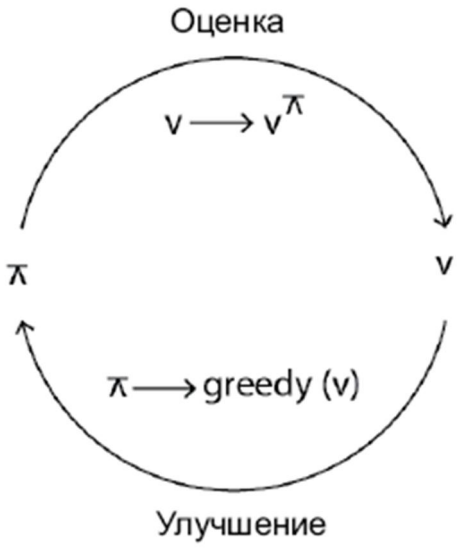
```
import numpy
import gym
from matplotlib import pyplot
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from collections import defaultdict
from functools import partial
%matplotlib inline

plt.style.use('ggplot')

## Среда Blackjack

env = gym.make('Blackjack-v0')

env.action_space, env.observation_space

def sample_policy(observation):
    score, dealer_score, usable_ace =
observation
    return 0 if score >= 20 else 1
```

```
  def generate_episode(policy, env):
      states, actions, rewards = [], [], []
      observation = env.reset()
      while True:
          states.append(observation)
          action = sample_policy(observation)
          actions.append(action)
          observation, reward, done, info =
env.step(action)
          rewards.append(reward)
          if done:
              break

      return states, actions, rewards

  def first_visit_mc_prediction(policy, env,
n_episodes):
      value_table = defaultdict(float)
      N = defaultdict(int)

      for _ in range(n_episodes):
          states, _, rewards =
generate_episode(policy, env)
          returns = 0
          for t in range(len(states) - 1, -1, -1):
              R = rewards[t]
              S = states[t]
              returns += R
              if S not in states[:t]:
                  N[S] += 1
                  value_table[S] += (returns -
value_table[S]) / N[S]
      return value_table

  def plot_blackjack(V, ax1, ax2):
      player_sum = numpy.arange(12, 21 + 1)
      dealer_show = numpy.arange(1, 10 + 1)
      usable_ace = numpy.array([False, True])

      state_values = numpy.zeros((len(player_sum),
                                  len(dealer_show)
,
                                  len(usable_ace))
)
```

```
    for i, player in enumerate(player_sum):
        for j, dealer in enumerate(dealer_show):
            for k, ace in enumerate(usable_ace):
                state_values[i, j, k] =
V[player, dealer, ace]

    X, Y = numpy.meshgrid(player_sum,
dealer_show)

    ax1.plot_wireframe(X, Y, state_values[:, :,
0])
    ax2.plot_wireframe(X, Y, state_values[:, :,
1])
    for ax in ax1, ax2:
        ax.set_zlim(-1, 1)
        ax.set_ylabel('player sum')
        ax.set_xlabel('dealer showing')
        ax.set_zlabel('state-value')
  fig, axes = pyplot.subplots(nrows=2, figsize=(5,
8),
  subplot_kw={'projection': '3d'})
  axes[0].set_title('value function without usable
ace')
  axes[1].set_title('value function with usable
ace')
  plot_blackjack(value, axes[0], axes[1])
```

Управление методом Монте-Карло

В прогнозировании методом Монте-Карло центральное место занимала оценка функции ценности. В управлении на первый план выходит ее оптимизация, то есть улучшение, нежели оценка. Управляющие методы подразумевают новую разновидность итераций — *итерацию по обобщенным политикам*, где оценка политики и ее улучшение взаимодействуют друг с другом. Таким образом, алгоритм выполняется как кольцевой переход между оценкой и улучшением политики, причем политика всегда улучшается относительно функции ценности, и наоборот. Если изменений нет, можно сказать, что произошло *схождение*, то есть была обнаружена комбинация оптимальной функции ценности и оптимальной политики:



А теперь рассмотрим различные управляющие методы Монте-Карло.

MC-ES

Здесь, в отличие от методов динамического программирования, ценность состояния не определяется. Вместо этого алгоритм концентрируется на ценности действия. Если модель среды известна, то достаточно ценности состояний. При неизвестной динамике модели способа точного определения ценности состояния не существует.

Внимание к ценности действия выбрано по причине зависимости ценности состояния от политики. Допустим, в игре блек-джек вы находитесь в состоянии с суммой очков 20. Какова ценность этого состояния? Это зависит исключительно от политики. Если выбрать политику со взятием карты, то это, безусловно, плохое состояние с очень низкой ценностью. Но если выбрать политику с отказом от взятия карты, то это состояние определенно хорошее. Таким образом, важнее определить ценность действия.

Как это сделать? Вспомните *Q*-функцию из **главы 3**. *Q*-функция, обозначаемая как *Q* (*s*, *a*), используется для определения желательности действия в конкретном состоянии. Фактически она определяет пару состояние/действие.

Но тут возникает проблема исследования. Как можно знать ценность состояния/действия, если вы еще не находились в этом состоянии? Не исследовав все возможные состояния и действия, можно случайно упустить хорошие награды.

Допустим, в игре блек-джек вы находитесь в состоянии с суммой очков 20. Если опробовать только действие **взятия карты**, дающее отрицательную награду, вы узнаете, что пребывание в этом состоянии нежелательно. Но если опробовать действие **отказа от взятия карты**, награда будет положительной, а само состояние окажется лучшим среди всех. Следовательно, каждый раз, когда вы оказываетесь в этом состоянии, нужно выбирать действие отказа. Чтобы определить, какое действие является лучшим (то есть оптимальным), необходимо исследовать все возможные действия в каждом состоянии. Как это сделать?

Позвольте мне представить новую концепцию *выбора начала исследования методом Монте-Карло*, или *MC-ES* (Monte Carlo exploring

starts), согласно которой в каждом эпизоде мы начинаем со случайного состояния как исходного и выполняем действие. Таким образом, при большом количестве эпизодов реально покрыть все состояния со всеми возможными действиями.

Алгоритм MC-ES очень прост:

• Инициализировать *Q*-функцию и политику случайными значениями, а `returns` — пустым списком.

• Запустить эпизод со случайно инициализированной политикой.

• Для всех уникальных пар состояние/действие, встречающихся в эпизоде, вычислить возврат, который присоединится к списку `returns`.

• Возврат должен вычисляться только для уникальных пар состояние/действие — нет смысла хранить избыточную информацию.

• Вычислить среднее значение по списку `returns`, которое будет присвоено *Q*-функции.

• Наконец, для состояния выбрать оптимальную политику (использовать действие с максимальным $Q(s, a)$).

• Процесс будет повторяться бесконечно или столько раз, сколько требует количество эпизодов, чтобы покрыть все уникальные пары состояние/действие.

Блок-схема этого процесса:

Метод Монте-Карло с привязкой к политике

В MC-ES исследуются все пары состояние/действие и выбирается пара, обеспечивающая максимальную ценность. Но представьте ситуацию с большим количеством состояний и действий. Если применить алгоритм MC-ES в такой ситуации, исследование всех пар и выбор лучшей займет слишком много времени. Как решить эту проблему? Управляющие методы делятся на две категории: с привязкой к политике и без привязки. В первой используется *жадная* (greedy) политика ε. Что такое «жадный алгоритм»?

Жадный алгоритм выбирает лучший вариант, доступный на данный момент, хотя этот вариант может оказаться не оптимальным, если рассматривать задачу в целом. Представьте, что вам нужно найти наименьшее число в списке чисел. Вместо того чтобы искать прямо в списке, можно разделить его на три подсписка и найти наименьшее число в одном из них (локальный оптимум). Это число может не являться наименьшим для целого списка (глобальный оптимум). Но если действовать «жадно», то наименьшее число в текущем подсписке (на данный момент) можно считать таковым для всего списка.

Жадная стратегия выбирает оптимальное действие среди уже исследованных. Оптимальным считается действие, обладающее наивысшей ценностью.

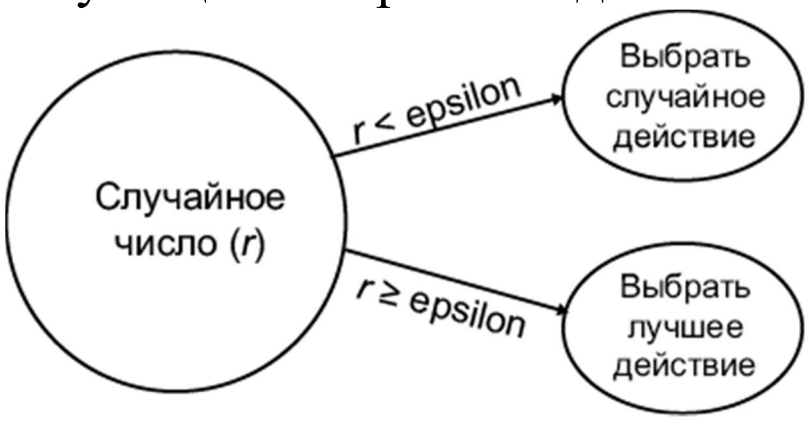
Допустим, вы исследовали некоторые действия в состоянии 1, как показано в *Q*-таблице:

Состояние	Действие	Ценность
Состояние 1	Действие 0	0,5
Состояние 1	Действие 1	0,1
Состояние 1	Действие 2	0,8

Если рассуждать «жадно», по результатам исследования вы выберете действие с максимальной ценностью. В этом примере действие 2 обладает высокой ценностью, поэтому выбирается оно. Однако в состоянии 1 могут быть другие действия с высокой ценностью, которые не были исследованы. Что необходимо искать: лучшее действие или действие, лучшее из всех исследованных действий? Это называется *дилеммой компромисса между исследованием и эксплуатацией*. Допустим, вы послушали Эда Ширана, вам очень понравилось, и вы продолжаете слушать только его (эксплуатация). Вопрос в том, остановиться ли на проверенном Эде Ширане или оценить другую музыку (исследование)?

Чтобы избежать этой дилеммы, мы вводим новую политику, называемую *эпсилон-жадной стратегией*. Все действия опробуются с ненулевой вероятностью (epsilon), а с вероятностью 1 – epsilon выбирается одно с максимальной ценностью. То есть вместо того чтобы

просто эксплуатировать лучшее действие, мы случайным образом исследуем разные действия. Если значение `epsilon`будет равно 0, исследование не будет выполнено — будет использована обычная жадная стратегия. Если же значение epsilon будет равно 1 — будет выполнено только исследование. Значение epsilon будет уменьшаться со временем, поскольку заниматься исследованиями до бесконечности незачем. Таким образом, со временем политика переходит на эксплуатацию «хороших» действий:



Допустим, `epsilon` присвоено значение 0,3. В следующем коде генерируется случайное число из равномерного распределения, и если значение меньше `epsilon` (0,3), выбирается случайное действие (в данном случае просто ищется другое действие). Если случайное значение из равномерного распределения больше 0,3, выбирается действие с наибольшей ценностью. Таким образом, с вероятностью epsilon будут исследованы новые действия и выбрано лучшее действие из исследованных с вероятностью 1 – epsilon:

```
  def epsilon_greedy_policy(state, epsilon):
      if random.uniform(0,1) < epsilon:
            return env.action_space.sample()
      else:
            return
max(list(range(env.action_space.n)), key = lambda
x:
  q[(state,x)])
```

Давайте представим, что другие действия в состоянии 1 были исследованы с эпсилон-жадной стратегией (хотя и не все пары), а *Q*-таблица выглядит так:

Состояние	Действие	Ценность
Состояние 1	Действие 0	0,5
Состояние 1	Действие 1	0,1
Состояние 1	Действие 2	0,8
Состояние 1	Действие 4	0,93

В состоянии 1 действие 4 обладает более высокой ценностью, чем действие 2, найденное до этого. С эпсилон-жадной стратегией ищутся другие действия и выбирается лучшее.

Метод Монте-Карло с привязкой к политике пошагово выглядит так:

1. Инициализировать случайную политику и случайную *Q*-функцию.

2. Инициализировать список с именем `return` для хранения возвратов.

3. Сгенерировать эпизод со случайной политикой π.

4. Сохранить возврат каждой пары состояние/действие в эпизоде в списке `returns`.

5. Затем вычислить среднее значение по списку `returns` и присвоить его *Q*-функции.

6. Теперь вероятность выбора действия `a` в состоянии `s` будет определяться значением `epsilon`.

7. С вероятностью 1 – epsilon будет выбрано действие с максимальным *Q*-значением.

8. С вероятностью epsilon будут исследованы разные действия.

Метод Монте-Карло без привязки к политике

Метод Монте-Карло без привязки к политике — еще один интересный управляющий метод. Он содержит две политики: одна — политика поведения, другая — целевая политика. Агенты следуют одной политике (политике поведения), но время от времени пытаются расширить имеющуюся информацию и улучшить другую (целевую политику). Две политики никак не связаны друг с другом. Политика поведения исследует все возможные состояния и действия, поэтому она также называется *мягкой политикой*, а целевая политика называется *жадной*(выбирается политика с максимальной ценностью).

Наша цель — оценить *Q*-функцию для целевой политики π, где поведение агентов будет определяться мягкой политикой μ. Как это сделать? Можно оценить ценность по общим эпизодам, происходящим в μ. Но как оценить общие эпизоды между двумя политиками? Мы воспользуемся новым методом, называемым *выборкой по значимости*. Этот метод оценивает значения из одного распределения по точкам данных из другого распределения.

Существуют два типа выборки по значимости:

• обычная;

• взвешенная.

В *обычной* выборке по значимости мы, по сути, берем отношение возвратов, получаемых по политике поведения и целевой политике, тогда как во *взвешенной* выборке мы берем взвешенное среднее и *C* — накопленная сумма весов.

Разберем этот алгоритм шаг за шагом:

1. Инициализировать $Q(s, a)$ случайными значениями, $C(s, a)$ — нулями, а вес w — единицей.

2. Выбрать целевую политику по жадному принципу, то есть обладающую максимальным значением из *Q*-таблицы.

3. Выбрать политику поведения не по жадному принципу — любую пару состояние/действие.

4. Затем начать эпизод: выполнить действие *a* в состоянии *s* согласно политике поведения и сохранить награду. Все это будет повторяться до конца эпизода.

5. Теперь для каждого состояния в эпизоде сделать следующее:

1) Вычислить возврат *G*. Мы знаем, что возврат равен сумме наград с учетом поправочного коэффициента: *G* = *поправочный коэффициент* × *G* + *награда*.

2) Обновить *C* (*s*, *a*) по формуле *C* (*s*, *a*) = *C* (*s*, *a*) + *w*.

3) Обновить *Q* (*s*, *a*): $$Q(s,a) = Q(s,a) + \frac{w}{C(s,a)} \times (G - Q(s,a))$$.

4) Обновить значение *w*: $$w = w \times \frac{1}{\text{политика поведения}}$$.

Итоги

В этой главе вы узнали, как работает метод Монте-Карло и как он используется для решения задач MDP, если модель среды неизвестна. Мы рассмотрели два разных метода: прогнозирование методом Монте-Карло, которое используется для оценки функции ценности, и управляющий метод Монте-Карло, который используется для оптимизации функции ценности.

Мы разобрали две разновидности прогнозирования методом Монте-Карло: метод с первым посещением, при котором возврат усредняется только при первом посещении состояния в эпизоде, и метод с каждым посещением, при котором возврат усредняется при каждом посещении состояния в эпизоде.

Что касается управляющих методов Монте-Карло, мы рассмотрели два разных алгоритма. Сначала был рассмотрен управляющий метод MC-ES, используемый для всех пар состояние/действие. Далее в отдельности были разобраны: метод с привязкой к политике, где применяется эпсилон-жадная стратегия, и метод без привязки к политике, использующий две политики одновременно.

В **главе 5** будет рассмотрен другой алгоритм обучения без модели.

Вопросы

1. Что такое метод Монте-Карло?

2. Вычислите приближенное значение «золотого сечения» методом Монте-Карло.

3. Для чего используется прогнозирование методом Монте-Карло?

4. Чем метод Монте-Карло с первым посещением отличается от метода Монте-Карло с каждым посещением?

5. Для чего оценивается ценность пар состояние/действие?

6. Чем управляющий метод Монте-Карло с привязкой к политике отличается от управляющего метода Монте-Карло без привязки к политике?

7. Напишите код Python для игры блек-джек на основе управляющего метода Монте-Карло с привязкой к политике.

Дополнительные источники

Презентация Дэвида Силвера (David Silver), посвященная прогнозированию без модели: `http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/MC-TD.pdf`.

Презентация Дэвида Силвера (David Silver), посвященная управляющим методам Монте-Карло без модели: `http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/control.pdf`.

5. Обучение на основе временных различий

В **главе 4** вы узнали о методе Монте-Карло для решения задач **марковского процесса принятия решений** (**MDP**), где динамика модели среды неизвестна заранее (в отличие от динамического программирования). Мы рассмотрели прогнозирование методом Монте-Карло для оценки функции ценности и управляющие методы для ее дальнейшей оптимизации. Тем не менее метод Монте-Карло не идеален. Например, он применим только к эпизодическим задачам. Если эпизод очень длинный, возможно, вам придется слишком долго ждать вычисления функции ценности. В таких условиях лучше использовать другой алгоритм, называемый *обучением на основе временных различий* (*TD*, temporal difference), который относится к категории алгоритмов обучения без модели — не требуя сведений о модели, он может применяться к неэпизодическим задачам.

В этой главе будут рассмотрены следующие темы:

- Обучение на основе временных различий (TD).
- *Q*-обучение.
- SARSA.
- Планирование поездок на такси средствами *Q*-обучения и SARSA.
- Различия между *Q*-обучением и SARSA.

Обучение на основе временных различий

Алгоритм обучения на основе временных различий был предложен Р.С. Саттоном (Richard S. Sutton) в 1988 году. Он обладает сильными сторонами как метода Монте-Карло (работает в неизвестной среде), так и **динамического программирования**(не требует ожидания конца эпизода для оценки функции ценности). В данном алгоритме оценка аппроксимируется на основании ранее полученной оценки, что также называется *компенсационной обратной связью* (bootstrapping). Как вы

видели, в методах Монте-Карло компенсационная обратная связь не поддерживается, а оценка формируется только к концу эпизода.

Прогнозирование на основе временных различий

При прогнозировании на основе временных различий (TD), как и в методе Монте-Карло, ищется ценность состояний. Однако в методе Монте-Карло функция ценности оценивается простым средним возвратом, тогда как в TD-обучении ценность текущего состояния обновляется по текущему состоянию. Как это происходит? В TD-обучении для обновления ценности состояния используется так называемое правило *обновления на основе временных различий*:

$V(s) = V(s) + \alpha\,(r + \gamma V(s') - V(s))$.

Ценность предыдущего состояния =ценность предыдущего состояния + скорость_обучения × (награда + поправочный коэффициент (ценность текущего состояния) – ценность предыдущего состояния).

Что фактически означает это уравнение?

Интуиция подсказывает, что результат будет равен разности между фактической наградой ($r + \gamma V(s')$) и ожидаемой наградой ($V(s)$), умноженной на скорость обучения (learning rate). *Скорость обучения*, также называемая *размером шага*(step size), нужна для схождения.

Вы обратили внимание? Вычисляется разность между фактическим и прогнозируемым значением в виде ($r + \gamma V(s')$). Фактически это погрешность, и мы будем называть ее *TD-погрешностью*. На нескольких итерациях попытаемся ее минимизировать.

Попробуем понять суть TD-прогнозирования на примере с замерзшим озером из предыдущих глав. Ниже изображена среда замерзшего озера. Сначала мы инициализируем функцию ценности нулями, как у $V(s)$ на следующем рисунке:

	1	2	3	4
1	S	F	F	F
2	F	H	F	H
3	F	F	F	H
4	H	F	F	G

Сост.	Цен.
(1,1)	0
(1,2)	0
(1,3)	0
…	…
(4,4)	0

Допустим, вы находитесь в исходном состоянии (s) **(1,1)**, выбираете действие «направо», переходите в следующее состояние (s') **(1,2)** и получаете награду (r) –0,3. Как обновить ценность состояния на основании этой информации?

Вспомните уравнение TD-обновления:

$V(s) = V(s) + \alpha\,[r + \gamma V(s') - V(s)]$.

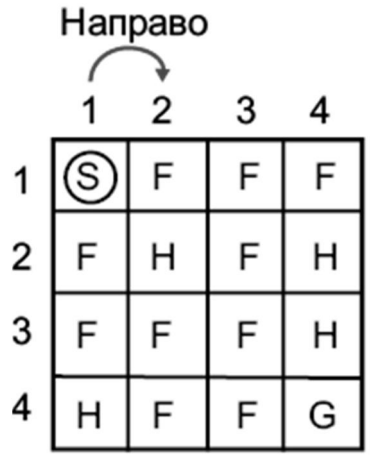
Будем считать, что скорость обучения (α) равна 0,1, а поправочный коэффициент (γ) равен 0,5. Мы знаем, что ценность состояния **(1,1)**, то есть $v(s)$, равна 0, а ценность следующего состояния **(1,2)**, то есть $V(s')$,

тоже равна 0. Полученная награда (*r*) равна –0,3. Подставим данные в правило TD:

$V(s) = 0 + 0{,}1[-0{,}3 + 0{,}5(0) - 0]$;

$v(s) = -0{,}03$.

Таким образом, ценность состояния **(1,1)** в таблице ценности обновляется значением –0,03, как видно из следующего рисунка:



Сост.	Цен.
(1,1)	–0,03
(1,2)	0
(1,3)	0
…	…
(4,4)	0

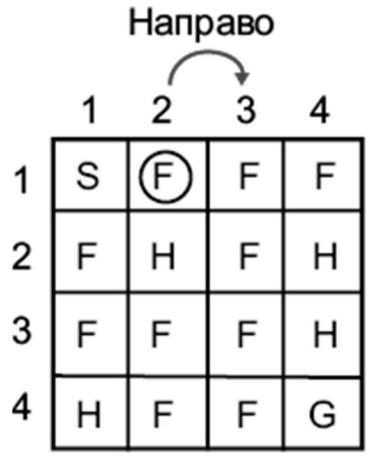
Находясь в состоянии (*s*) **(1,2)**, мы выбираем действие «направо», переходим в следующее состояние (*s′*) **(1,3)** и получаем награду (*r*) –0,3. Как теперь обновить ценность состояния **(1,2)**?

Как и прежде, подставим значения в уравнение TD-обновления:

$V(s) = 0 + 0{,}1[-0{,}3 + 0{,}5(0) - 0]$;

$V(s) = -0{,}03$.

Итак, ценность состояния **(1,2)**равна –0,03, и мы заносим ее в таблицу ценности:



Сост.	Цен.
(1,1)	–0,03
(1,2)	–0,03
(1,3)	0
…	…
(4,4)	0

Теперь агент находится в состоянии (*s*) **(1,3)**; предположим, было выбрано действие «налево». Мы снова возвращаемся к состоянию (*s′*) **(1,2)** и получаем награду (*r*) –0,3. Ценность состояния **(1,3)** равна 0, а ценность следующего состояния **(1,2)**, как видно из таблицы ценности, равна –0,03:

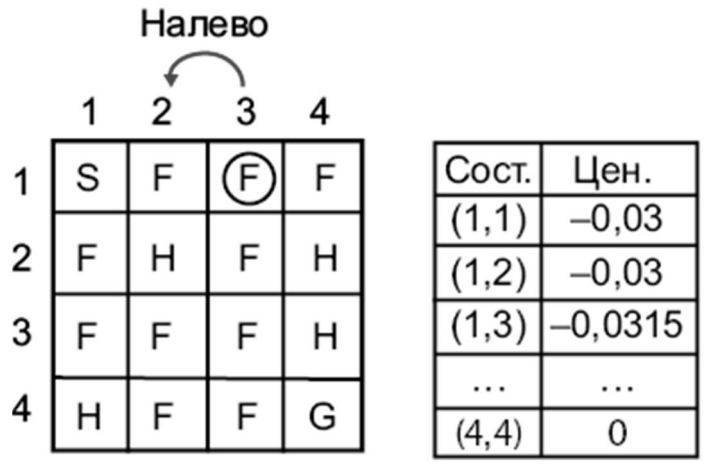
После этого ценность состояния (**1,3)** обновляется следующим образом:

$V(s) = 0 + 0{,}1[-0{,}3 + 0{,}5\ (-0{,}03) - 0)]$;

$V(s) = 0{,}1[-0{,}315]$;

$V(s) = -0{,}0315$.

Ценность состояния (**1,3)**, равная –0,0315, обновляется в таблице ценности так, как показано ниже:



Сост.	Цен.
(1,1)	–0,03
(1,2)	–0,03
(1,3)	–0,0315
…	…
(4,4)	0

Аналогичным образом обновляются ценности всех состояний. Последовательность действий алгоритма TD-прогнозирования выглядит так:

1. Инициализировать $V(s)$ нулями или произвольными значениями.
2. Запустить эпизод, для каждого шага в эпизоде выполнить действие Aв состоянии s, получить награду R и перейти в следующее состояние (s').
3. Обновить ценность предыдущего состояния по правилу TD-обновления.
4. Повторять шаги 3 и 4, пока не будет достигнуто завершающее состояние.

TD-управление

В TD-прогнозировании оценивается функция ценности; в TD-управлении она оптимизируется. Для этого используются две разновидности управляющего алгоритма:

- **алгоритм обучения без привязки к политике**: *Q*-обучение;
- **алгоритм обучения с привязкой к политике**: SARSA.

Q-обучение

Q-обучение — это очень простой TD-алгоритм, широко применяемый на практике. В алгоритмах управления ценность состояния не рассматривается; в *Q*-обучении интерес представляет ценность пар состояние/действие — эффект выполнения действия a в состоянии s.

Значение Q обновляется по следующей формуле:

$$Q(s, a) = Q(s, a) + \alpha(r + \gamma \max Q(s', a') - Q(s, a)).$$

Это уравнение напоминает правило обновления для TD-прогнозирования, но с небольшим отличием. Разберем его шаг за шагом:

1. Инициализация Q-функции произвольными значениями.
2. Выбор действия из состояния с использованием эпсилон-жадной стратегии ($\varepsilon > 0$) и переход в новое состояние.
3. Обновление Q предыдущего состояния по следующему правилу:

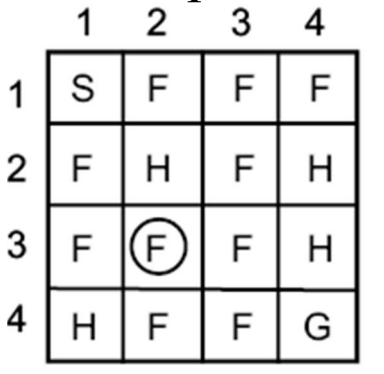
$$Q(s, a) = Q(s, a) + \alpha\,(r + \gamma \max Q(s', a') - Q(s, a)).$$

4. Повторение шагов 3 и 4 до достижения завершающего состояния.

Рассмотрим работу алгоритма на конкретном примере.

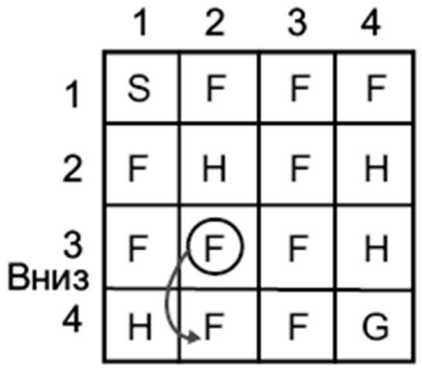
Возьмем уже знакомый пример с замерзшим озером. Допустим, вы находитесь в состоянии **(3,2)** с двумя возможными действиями («налево»

и «направо»). Обратимся к рисунку и рассмотрим применение эпсилон-жадной стратегии:



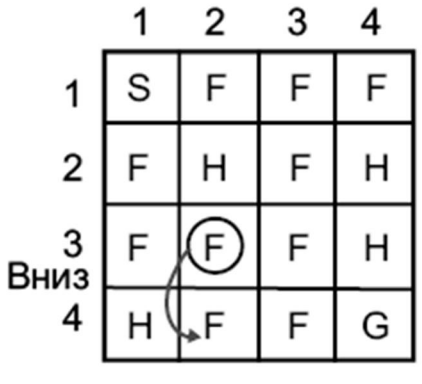
Сост.	Действ.	Цен.
(3,2)	Направо	0,1
(3,2)	Налево	0,5

В *Q*-обучении действие выбирается с использованием эпсилон-жадной стратегии: либо с вероятностью epsilon исследуется новое действие, либо с вероятностью 1 – epsilon выбирается лучшее из известных действий. Допустим, с вероятностью epsilon было исследовано новое действие «вниз»:



Сост.	Действ.	Цен.
(3,2)	Направо	0,1
(3,2)	Налево	0,5
(3,2)	Вниз	0,8

Мы выполнили действие «вниз» в состоянии **(3,2)** и достигли нового состояния **(4,2)** с использованием эпсилон-жадной стратегии. Как обновить ценность предыдущего состояния **(3,2)** с использованием правила обновления? Очень просто. Взгляните на *Q*-таблицу:



Сост.	Действ.	Цен.
(3,2)	Направо	0,1
(3,2)	Налево	0,5
(3,2)	Вниз	0,8
(4,2)	Вверх	0,3
(4,2)	Вниз	0,5
(4,2)	Направо	0,8

Допустим, значение α равно 0,1, а поправочный коэффициент — 1:

$Q(s, a) = Q(s, a) + \alpha (r + \gamma \max Q(s', a) - Q(s, a))$;

Q ((3,2), *вниз*) = Q ((3,2), *вниз*) + 0,1 (0,3 + 1max [Q ((4,2), *действие*)] – Q ((3,2),*вниз*).

Можно сказать, что, согласно *Q*-таблице, ценность состояния **(3,2)** с действием «вниз», то есть Q ((3,2),*вниз*), равна 0,8.

Что такое max [Q ((4,2), *действие*)] для состояния **(4,2)**? Мы исследовали только три действия («вверх», «вниз» и «направо»), поэтому максимум определяется только для этих действий. (Здесь эпсилон-жадная стратегия не применяется, а просто выбирается действие с максимальной ценностью.)

Теперь можно подставить значения по приведенной *Q*-таблице:

Q ((3,2), *вниз*) = 0,8 + 0,1 (0,3 + 1max [0,3, 0,5, 0,8] – 0,8) =

= 0,8 + 0,1(0,3 + 1(0,8) – 0,8) =

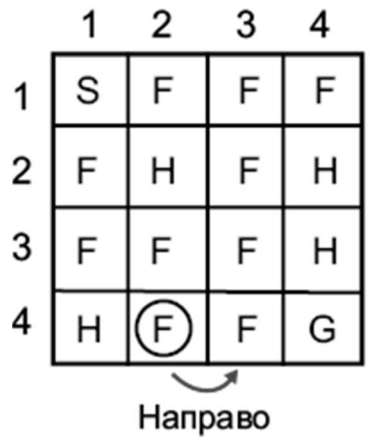
= 0,83.

В результате ценность Q ((3,2),*вниз*) обновляется до 0,83.

примечание

Важно запомнить, что при выборе действия применяется эпсилон-жадная стратегия: либо с вероятностью epsilon исследуются новые действия, либо с вероятностью 1 – epsilon выбирается действие с максимальной ценностью. При обновлении Q эпсилон-жадная стратегия не применяется, а просто выбирается действие с максимальной ценностью.

Итак, мы находимся в состоянии **(4,2)** и должны выполнить действие. Какое действие должно выполняться? Допустим, по эпсилон-жадной стратегии выбирается лучшее действие. В **(4,2)** действие «направо» обладает максимальной ценностью, поэтому выбирается именно оно:



Сост.	Действ.	Цен.
(4,2)	Вверх	0,3
(4,2)	Вниз	0,5
(4,2)	Направо	0,8

Мы находимся в состоянии **(4,3)**, поскольку в состоянии **(4,2)** было выбрано действие «направо». Ценность предыдущего состояния обновляется следующим образом:

Q ((4,2), *направо*) = Q ((4,2), *направо*) + 0,1 (0,3 + 1max [Q ((4,3), *действие*)] – – Q ((4,2), *направо*).

В следующей Q-таблице для состояния **(4,3)** исследованы только два действия (вверх и вниз), поэтому максимальная ценность определяется на основании только этих действий. (При этом эпсилон-жадная стратегия не применяется, а просто выбирается действие с максимальной ценностью.)

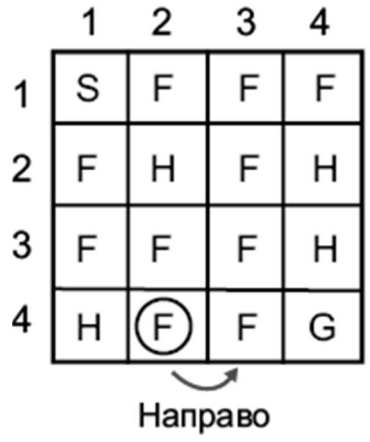
Q ((4,2), *направо*) = Q ((4,2), *направо*) + 0,1(0,3 + 1max [(Q (4,3), *вверх*), (Q (4,3), *вниз*)] – Q ((4,2), направо).

Q ((4,2), *направо*) = 0,8 + 0,1(0,3 + 1max [0,1, 0,3] – 0,8) =

= 0,8 + 0,1 (0,3 + 1(0,3) – 0,8) =

= 0,78.

Взгляните на следующую Q-таблицу:



Сост.	Действ.	Цен.
(4,2)	Вверх	0,3
(4,2)	Вниз	0,5
(4,2)	Направо	0,8
(4,3)	Вверх	0,1
(4,3)	Вниз	0,3

Здесь ценность состояния Q ((4,2), *направо*) обновляется значением 0,78.

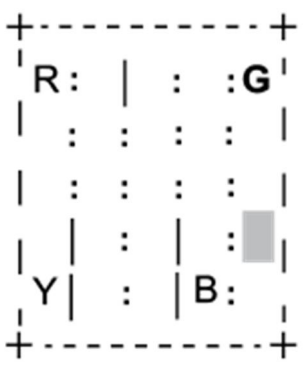
Так оцениваются ценности пар состояние/действие в Q-обучении. Чтобы решить, какое действие должно выполняться, мы применяем

эпсилон-жадную стратегию и при обновлении Q просто выбираем действие с максимальной ценностью. Приведу блок-схему алгоритма:

Решение задачи о такси с использованием Q-обучения

Чтобы продемонстрировать работу алгоритма, предположим, что агент является водителем. На поле отмечены четыре точки, агент должен подобрать пассажира в одной точке и высадить в другой. Агент получает +20 очков награды за успешную высадку и –1 очко за каждый затраченный квант времени. Агент также теряет 10 очков за ошибочные посадки и высадки. Задача агента — научиться подбирать и высаживать пассажиров в правильных местах за короткое время без ошибок.

Ниже изображена среда; буквы (**R**, **G**, **Y**, **B**) изображают разные точки, а маленький прямоугольник — агента, управляющего такси:



А теперь посмотрим, как выглядит реализация:

```
import gym
import random
```

Теперь создайте среду средствами Gym:

```
env = gym.make("Taxi-v1")
```

Как выглядит среда? Вот так:

```
env.render()
```

Начнем с инициализации скорости обучения `alpha`, переменных `epsilon` и `gamma`:

```
alpha = 0.4
gamma = 0.999
epsilon = 0.017
```

Затем инициализируем *Q*-таблицу — словарь для хранения пар состояние/действие:

```
q = {}
for s in range(env.observation_space.n):
    for a in range(env.action_space.n):
        q[(s, a)] = 0.0
```

Теперь нужно определить функцию для обновления *Q*-таблицы по правилу *Q*-обучения; из приведенного ниже кода видно, что функция выберет действие, обладающее максимальной ценностью для пары состояние/действие, и сохранит его в переменной `qa`. Затем значение *Q* предыдущего состояния обновится по правилу обновления:

$Q(s, a) = Q(s, a) + \alpha\,(r + \gamma \max Q(s', a') - Q(s, a))$.

```
def update_q_table(prev_state, action, reward, nextstate, alpha, gamma):
    qa = max([q[(nextstate, a)] for a in range(env.action_space.n)])
    q[(prev_state,action)] += alpha * (reward + gamma * qa -
q[(prev_state,action)])
```

Также необходимо определить функцию для применения эпсилон-жадной стратегии, в параметрах которой отразится состояние и значение `epsilon`. Генерируется случайное число с равномерным распределением; если оно меньше `epsilon`, исследуется другое

действие в состоянии, а если нет — выбирается действие с максимальным значением `q`:

```
def epsilon_greedy_policy(state, epsilon):
    if random.uniform(0,1) < epsilon:
        return env.action_space.sample()
    else:
        return
max(list(range(env.action_space.n)), key = lambda
x:
q[(state,x)])
```

Объединив все эти функции, мы увидим, как работает *Q*-обучение:

```
# Для каждого эпизода
for i in range(8000):

    r = 0
    # Инициализировать среду

    prev_state = env.reset()
    while True:
        # Для каждого состояния выбрать действие
по эпсилон-жадной политике
        action =
epsilon_greedy_policy(prev_state, epsilon)
        # выполнить выбранное действие и перейти
в следующее состояние
        nextstate, reward, done, _ =
env.step(action)
        # Функция update_q_table() обновляет Q-
таблицу
        # по правилу обновления.

        update_q_table(prev_state, action,
reward, nextstate, alpha, gamma)
        # Затем переменная предыдущего состояния
обновляется следующим
        # состоянием
        prev_state = nextstate

        # Награды сохраняются в r
        r += reward
        # Если процесс завершен, то есть
достигнуто завершающее состояние
        # эпизода, прервать цикл и начать
следующий эпизод
```

```
            if done:
                break

    print("total reward: ", r)

env.close()
```

Ниже приведен полный код:

```
import random
import gym

env = gym.make('Taxi-v1')

alpha = 0.4
gamma = 0.999
epsilon = 0.017

q = {}
for s in range(env.observation_space.n):
for a in range(env.action_space.n):
q[(s, a)] = 0

def update_q_table(prev_state, action, reward,
nextstate, alpha, gamma):
qa = max([q[(nextstate, a)] for a in
range(env.action_space.n)])
q[(prev_state,action)] += alpha * (reward +
gamma * qa -
q[(prev_state,action)])

def epsilon_greedy_policy(state, epsilon):
if random.uniform(0,1) < epsilon:
return env.action_space.sample()
else:
return max(list(range(env.action_space.n)), key
= lambda x: q[(state,x)])

for i in range(8000):
    r = 0
    prev_state = env.reset()
    while True:
        env.render()
        # Для каждого состояния выбрать действие
по эпсилон-жадной политике
```

```
            action =
epsilon_greedy_policy(prev_state, epsilon)
            # выполнить выбранное действие, перейти
в следующее состояние
            # и получить награду.
            nextstate, reward, done, _ =
env.step(action)
            # Функция update_q_table() обновляет Q-
таблицу
            # по правилу обновления.
            update_q_table(prev_state, action,
reward, nextstate, alpha, gamma)
            # Затем переменная предыдущего состояния
обновляется
            # следующим состоянием
            prev_state = nextstate

            # Награды сохраняются в r
            r += reward

            # Если процесс завершен, то есть
достигнуто завершающее состояние
            # эпизода, прервать цикл и начать
следующий эпизод
            if done:
                break

        print("total reward: ", r)

    env.close()
```

SARSA

State-Action-Reward-State-Action(*SARSA*) — алгоритм TD-управления с привязкой к политике. Как и в случае с *Q*-обучением, здесь нас интересует ценность пары состояние/действие, нежели пары состояние/ценность. В SARSA *Q*-таблица обновляется по следующему правилу:

$$Q(s, a) = Q(s, a) + \alpha(r + \gamma Q(s', a') - Q(s, a)).$$

Возможно, вы заметили, что в этом уравнении нет составляющей max $Q(s', a')$, как в *Q*-обучении, — здесь используется просто $Q(s', a')$. Чтобы лучше понять суть происходящего, попробуем воспроизвести некоторые шаги алгоритма.

1. Инициализация *Q*-функции произвольными значениями.

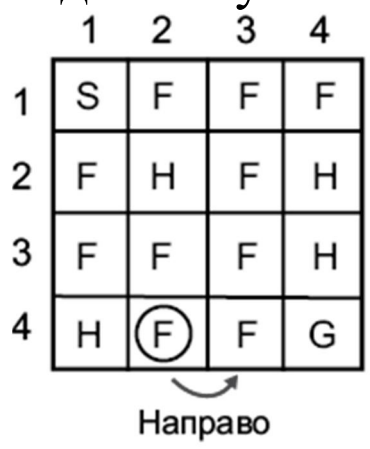
2. Выбор действия из состояния с использованием эпсилон-жадной стратегии (ε > 0) и переход в новое состояние.

3. Обновление *Q* предыдущего состояния по следующему правилу:

$Q(s, a) = Q(s, a) + \alpha (r + \gamma Q(s', a') - Q(s, a))$,

где a' — действие, выбранное по эпсилон-жадной стратегии (ε > 0).

Теперь повторим алгоритм шаг за шагом на примере с замерзшим озером. Допустим, вы находитесь в состоянии **(4,2)** и выбираете действие на основании эпсилон-жадной стратегии. С вероятностью 1 – epsilon находится лучшее действие — «направо»:



Сост.	Действ.	Цен.
(4,2)	Вверх	0,3
(4,2)	Вниз	0,5
(4,2)	Направо	0,8

Вы оказываетесь в состоянии **(4,3)**после выполнения действия «направо» в состоянии **(4,2)**. Как обновить ценность предыдущего состояния **(4,2)**? Будем считать, что значение α равно 0,1, награда — 0,3, а поправочный коэффициент — 1:

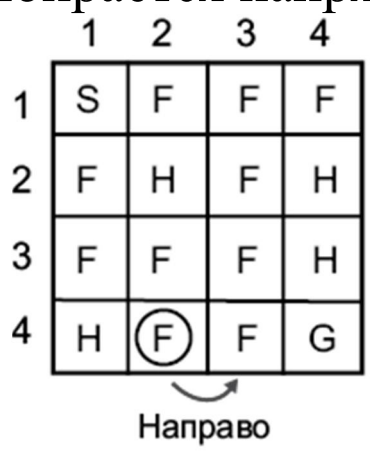
$Q(s, a) = Q(s, a) + \alpha (r + \gamma Q(s', a') - Q(s, a))$;

$Q((4,2), \textit{направо}) = Q((4,2), \textit{направо}) + 0{,}1\,(0{,}3 + 1Q((4,3), \textit{действие})) -$

$- Q((4,2), \textit{направо})$.

Как определить значение для $Q((4,3), \textit{действие})$? В отличие от *Q*-обучения, мы не просто выбираем $\max(Q(4,3), \textit{действие})$ — в SARSA используется эпсилон-жадная стратегия.

Взгляните на следующую *Q*-таблицу. В состоянии **(4,3)**исследованы два действия. В отличие от *Q*-обучения, максимальное действие не выбирается напрямую (например, «вниз»):



Сост.	Действ.	Цен.
(4,2)	Вверх	0,3
(4,2)	Вниз	0,5
(4,2)	Направо	0,8
(4,3)	Вверх	0,1
(4,3)	Вниз	0,3

Здесь снова используется эпсилон-жадная стратегия: либо исследование с вероятностью epsilon, либо выбор с вероятностью 1 – epsilon. Допустим с вероятностью epsilon мы исследуем новое действие «направо»:

Сост.	Действ.	Цен.
(4,2)	Вверх	0,3
(4,2)	Вниз	0,5
(4,2)	Направо	0,8
(4,3)	Вверх	0,1
(4,3)	Вниз	0,3
(4,3)	Направо	0,9

Q ((4,2), *направо*) = Q ((4,2), *направо*) + 0,1(0,3 + 1(Q (4,3), *направо*) – – Q ((4,2), *направо*);

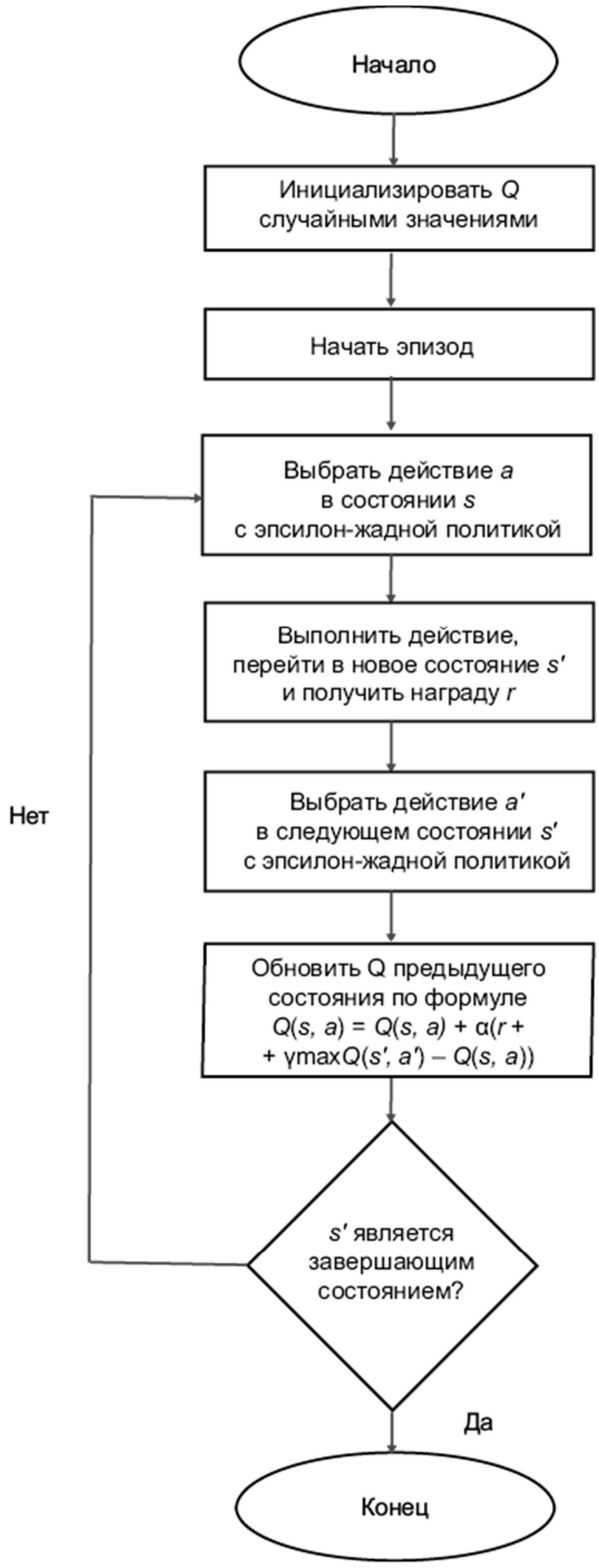
Q ((4,2), *направо*) = 0,8 + 0,1 (0,3 + 1(0,9) – 0,8) =

= 0,8 + 0,1 (0,3 + 1(0,9) – 0,8) =

= 0,84.

Так в SARSA определяется ценность пар состояние/действие. Действие выбирается с применением эпсилон-жадной стратегии, которая вновь используется для выбора действия при обновлении Q.

Следующая блок-схема объясняет алгоритм SARSA:



Решение задачи о такси методом SARSA

Применим SARSA для решения той же задачи о такси:

```
import gym
import random
env = gym.make('Taxi-v1')
```

Как и прежде, инициализируем переменные `alpha`, `gamma` и `epsilon`. *Q*-таблица содержит словарь:

```
alpha = 0.85
gamma = 0.90
epsilon = 0.8

Q = {}
```

```
for s in range(env.observation_space.n):
    for a in range(env.action_space.n):
        Q[(s, a)] = 0.0
```

Как обычно, определим функцию `epsilon_greedy` для исследования:

```
def epsilon_greedy(state, epsilon):
    if random.uniform(0,1) < epsilon:
        return env.action_space.sample()
    else:
        return
max(list(range(env.action_space.n)), key = lambda
x:
Q[(state,x)])
```

Далее следует сам алгоритм SARSA:

```
for i in range(4000):
    # Накапливаемая награда всех эпизодов
сохраняется в r
    r = 0
    # Для каждой итерации инициализируется
состояние,
    state = env.reset()
    # после чего выбирается действие с
применением эпсилон-жадной политики
    action = epsilon_greedy(state,epsilon)
    while True:
        # выполнить выбранное действие и перейти
в следующее состояние
        nextstate, reward, done, _ =
env.step(action)
        # выбрать следующее действие с
применением эпсилон-жадной политики
        nextaction =
epsilon_greedy(nextstate,epsilon)
        # вычислить Q предыдущего состояния по
правилу обновления
        Q[(state,action)] += alpha * (reward +
gamma *
  Q[(nextstate,nextaction)]-Q[(state,action)])
        # наконец, мы обновляем state и action
следующим состоянием
        # и следующим действием
        action = nextaction
        state = nextstate
        r += reward
```

```
            # Если достигнуто завершающее состояние
            # эпизода - прервать цикл
            if done:
                break

    env.close()
```

Запустим программу и посмотрим, как алгоритм SARSA ищет оптимальный путь.

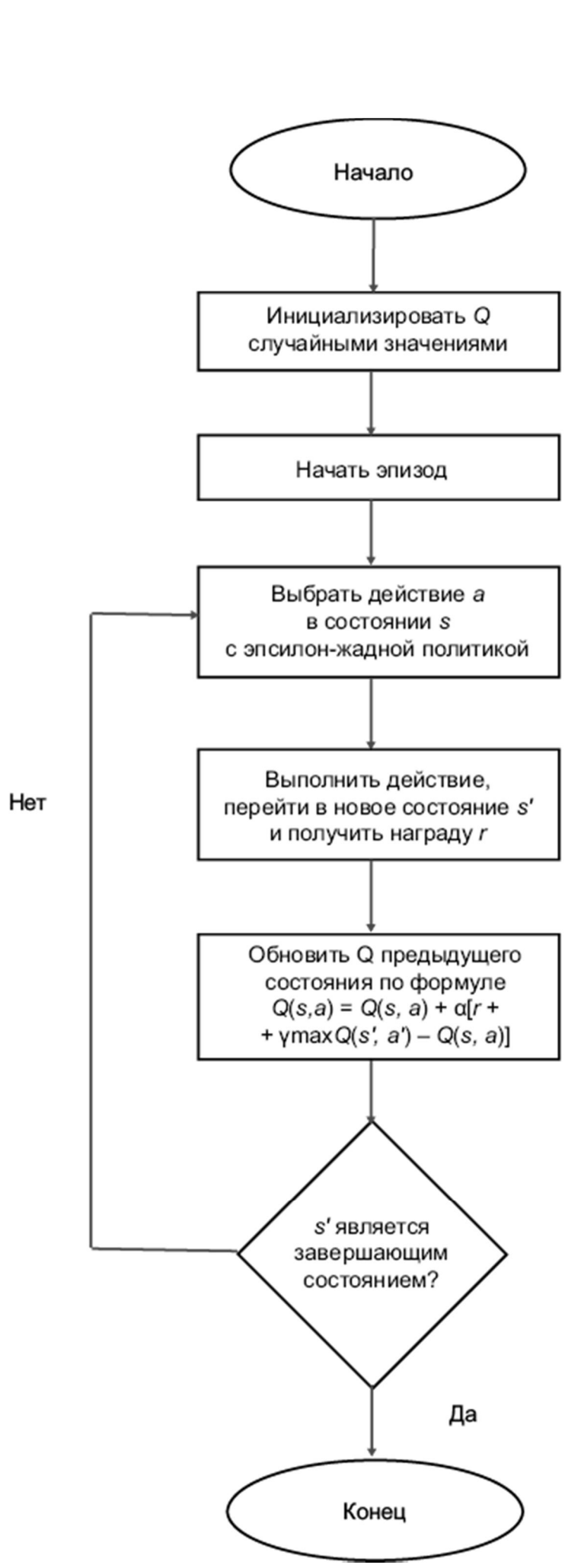
Полный код программы:

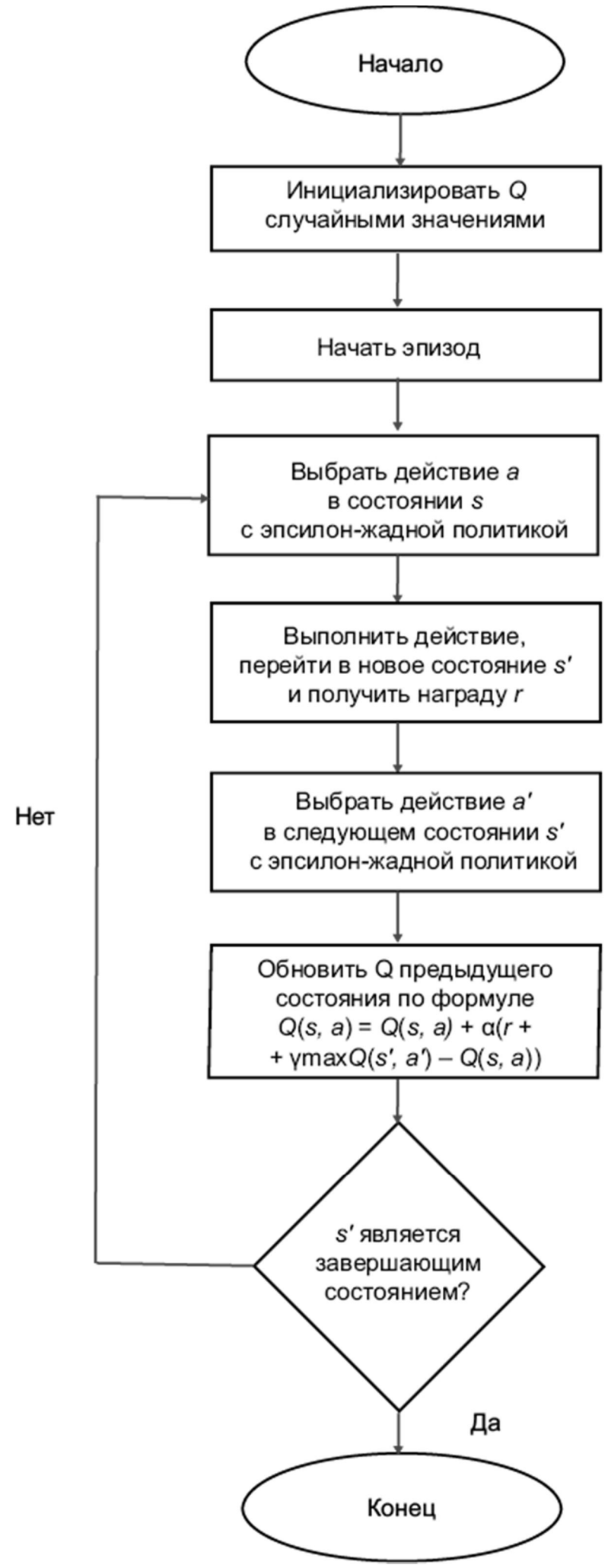
```
    # Импортировать необходимые библиотеки и
инициализировать среду

    import gym
    import random
    env = gym.make('Taxi-v1')

    alpha = 0.85
    gamma = 0.90
    epsilon = 0.8

    # Инициализировать Q-таблицу как словарь для
хранения
    # пар состояние/действие
    Q = {}
    for s in range(env.observation_space.n):
        for a in range(env.action_space.n):
            Q[(s, a)] = 0.0

    # Функция epsilon_greedy для выполнения действия
    # с применением эпсилон-жадной политики
    def epsilon_greedy(state, epsilon):
        if random.uniform(0,1) < epsilon:
            return env.action_space.sample()
        else:
            return
max(list(range(env.action_space.n)), key = lambda
x:
    Q[(state,x)])

    for i in range(4000):
        # Накапливаемая награда всех эпизодов
сохраняется в r
        r = 0
```

```
    # Для каждой итерации инициализируется состояние,
    state = env.reset()
    # после чего выбирается действие с применением эпсилон-жадной политики
    action = epsilon_greedy(state,epsilon)
    while True:
        # выполнить выбранное действие и перейти в следующее состояние
        nextstate, reward, done, _ = env.step(action)
        # выбрать следующее действие с применением эпсилон-жадной политики
        nextaction = epsilon_greedy(nextstate,epsilon)
        # вычислить Q предыдущего состояния по правилу обновления
        Q[(state,action)] += alpha * (reward + gamma *
  Q[(nextstate,nextaction)]-Q[(state,action)])

        # наконец, мы обновляем state и action следующим состоянием
        # и следующим действием
        action = nextaction
        state = nextstate
        r += reward
        # Если достигнуто завершающее состояние
        # эпизода - прервать цикл
        if done:
            break

env.close()
```

Различия между Q-обучением и SARSA

Многие разработчики неизбежно будут путать *Q*-обучение и SARSA. Пожалуй, стоит четко выделить главные различия между ними. Взгляните на следующие блок-схемы:





Заметили различия? В *Q*-обучении действие выполняется с применением эпсилон-жадной стратегии, и при обновлении *Q* мы просто выбирали максимальное действие. В SARSA действие также выбирается с применением эпсилон-жадной стратегии, но при обновлении *Q*действие выбиралось снова с применением эпсилон-жадной стратегии.

Итоги

В этой главе был представлен еще один алгоритм обучения без модели, избавленный от недостатков методов Монте-Карло. Были представлены как методы прогнозирования, так и управляющие методы. При TD-прогнозировании ценность состояния обновлялась с учетом следующего состояния. Из области управляющих методов были представлены два разных алгоритма: *Q*-обучение и SARSA.

Вопросы

1. Чем TD-обучение отличается от метода Монте-Карло?
2. Что называется TD-погрешностью?
3. Как построить интеллектуального агента на базе *Q*-обучения?
4. Чем *Q*-обучение отличается от SARSA?

Дополнительные источники

Исходная статья Ричарда Саттона (Richard Sutton) о TD:`https://pdfs.semanticscholar.org/9c06/865e912788a6a51470724e087853d7269195.pdf.`

6. Задача о многоруком бандите

В предыдущих главах были представлены фундаментальные концепции **обучения с подкреплением** (**RL**) и некоторые алгоритмы RL, а также моделирование RL-задач в форме **марковского процесса принятия решений** (**MDP**). Также были описаны различные алгоритмы (с моделью и без модели), используемые для решения MDP. В этой главе будет рассмотрена одна из классических задач RL — **задача о многоруком бандите**, или **MAB**(multi-armed bandit). Вы узнаете, что собой представляет эта задача, как она решается при помощи разных алгоритмов и как выбрать наиболее подходящий рекламный баннер, который будет получать наибольшее число переходов, средствами MAB. Также будет описан так называемый контекстный бандит, широко применяемый при построении рекомендационных систем.

В этой главе рассматриваются следующие темы:

- Задача о многоруком бандите (MAB).
- Эпсилон-жадный алгоритм.
- Алгоритм softmax-исследования.
- Алгоритм верхней границы доверительного интервала.
- Алгоритм выборки Томпсона.
- Практические применения MAB.
- Выбор подходящего рекламного баннера с использованием MAB.
- Контекстные бандиты.

Задача MAB

Задача о многоруком бандите (*MAB*) — одна из классических задач в RL. Многорукий бандит — игровой автомат; азартная игра, в которой игрок нажимает на рычаг и получает награду (выигрыш) на основании случайно сгенерированного распределения вероятностей. Отдельный автомат называется «одноруким бандитом»; если же таких автоматов несколько, это называется «многоруким бандитом» или *k*-руким бандитом.

Многорукий бандит выглядит так:

Так как каждый автомат дает выигрыш из собственного вероятностного распределения, наша цель — определить, какой автомат принесет максимальную накапливаемую награду за конкретный промежуток времени. Таким образом, на каждом временном кванте t агент выполняет действие a_t, то есть нажимает рычаг игрового автомата и получает награду r_t, при этом цель агента заключается в максимизации накапливаемой награды.

Определим ценность руки $Q(a)$ как среднюю награду, получаемую при нажатии рычага:

$$Q(a) = \frac{\text{Сумма наград, полученных на руке}}{\text{Общее количество активизаций руки}}.$$

Таким образом, *оптимальной* будет называться рука, обеспечивающая максимальную накапливаемую награду:

$Q(a^*) = \text{Max}Q(a)$.

Цель агента — найти оптимальную руку и минимизировать потери, которые могут определяться как затраты на получение информации о том, какая из k рук является оптимальной. Как найти лучшую руку? Исследовать все руки или выбрать ту руку, которая уже дает максимальную накапливаемую награду? Мы снова сталкиваемся с дилеммой компромисса между исследованием и эксплуатацией. В этой главе будет показано, как эта дилемма решается с помощью различных стратегий исследования:

- эпсилон-жадная стратегия;
- softmax-исследование;
- алгоритм верхней границы доверительного интервала;
- алгоритм выборки Томпсона.

Прежде чем двигаться дальше, установите среды `bandit` в OpenAI Gym; для этого введите в терминале следующие команды:

```
git clone https://github.com/JKCooper2/gym-bandits.git
cd gym-bandits
pip install -e .
```

После установки импортируйте `gym` и `gym_bandits`:

```
import gym_bandits
import gym
```

Теперь нужно инициализировать среду; мы будем использовать MAB с десятью руками:

```
env = gym.make("BanditTenArmedGaussian-v0")
```

Размер пространства действий равен 10, так как у бандита 10 рук. Соответственно, команда

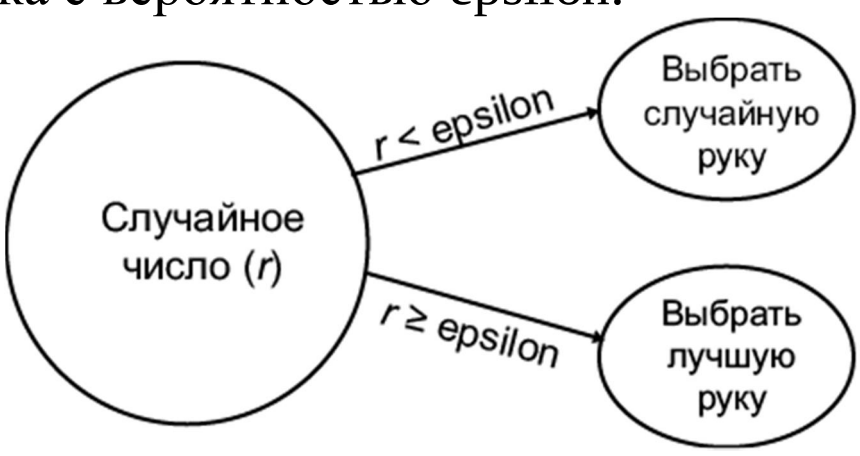
```
env.action_space
```

выводит следующий результат:

```
10
```

Эпсилон-жадная стратегия

Вы уже неоднократно сталкивались с эпсилон-жадной стратегией; здесь выбирается либо лучшая рука с вероятностью 1 – epsilon, либо случайная рука с вероятностью epsilon:



Посмотрим, как выбрать лучшую руку с применением эпсилон-жадной стратегии:

1. Инициализировать все переменные:

```
# Количество раундов (итераций)
num_rounds = 20000

# Количество нажатий на рычаг
count = np.zeros(10)

# Сумма наград для каждой руки
sum_rewards = np.zeros(10)

# Q, то есть средняя награда
Q = np.zeros(10)
```

2. Затем определить функцию `epsilon_greedy`:

```
def epsilon_greedy(epsilon):
    rand = np.random.random()
    if rand < epsilon:
        action = env.action_space.sample()
    else:
        action = np.argmax(Q)
    return action
```

3. Перейти к нажатию рычагов:

```
for i in range(num_rounds):
```

```
    # Выбрать руку с применением эпсилон-жадной политики
    arm = epsilon_greedy(0.5)
    # Получить награду
    observation, reward, done, info = env.step(arm)
    # Обновить счетчик для этой руки
    count[arm] += 1
    # Включить в сумму наград от руки
    sum_rewards[arm]+=reward
    # Вычислить Q - среднюю награду от руки
    Q[arm] = sum_rewards[arm]/count[arm]

print( 'The optimal arm is {}'.format(np.argmax(Q)))
```

Результат выполнения:

```
The optimal arm is 3
```

Алгоритм softmax-исследования

Softmax-исследование, также называемое *исследованием Больцмана*, — еще одна стратегия, применяемая для нахождения оптимальной руки. В эпсилон-жадной стратегии все не лучшие руки считаются равноправными, а в softmax-исследовании рука выбирается на основании вероятности из распределения Больцмана. Эта вероятность определяется следующей формулой:

$$P_t(a) = \frac{\exp(Q_t(a)/\tau)}{\sum_{i=1}^{n}\exp(Q_t(i)/\tau)}.$$

Здесь τ — температурный коэффициент, который определяет, сколько случайных рук можно исследовать. При высоких значениях τ все руки будут исследоваться с равной вероятностью, а при низких значениях τ будут выбираться руки с высокой наградой. Последовательность действий выглядит так:

1. Инициализация переменных:

```
# Количество раундов (итераций)
num_rounds = 20000

# Количество нажатий на рычаг
count = np.zeros(10)

# Сумма наград для каждой руки
sum_rewards = np.zeros(10)

# Q, то есть средняя награда
Q = np.zeros(10)
```

2. Определение softmax-функции:

```
def softmax(tau):
    total = sum([math.exp(val/tau) for val in
Q])
    probs = [math.exp(val/tau)/total for val in
Q]
    threshold = random.random()
    cumulative_prob = 0.0
    for i in range(len(probs)):
        cumulative_prob += probs[i]
        if (cumulative_prob > threshold):
            return i
    return np.argmax(probs)
```

3. Переход к нажатию рычагов:

```
for i in range(num_rounds):
    # Выбрать руку с применением softmax-
политики
    arm = softmax(0.5)
    # Получить награду
    observation, reward, done, info =
env.step(arm)
    # Обновить счетчик для этой руки
    count[arm] += 1
    # Просуммировать награды от руки
    sum_rewards[arm]+=reward
    # Вычислить Q - среднюю награду от руки
    Q[arm] = sum_rewards[arm]/count[arm]
print( 'The optimal arm is
{}'.format(np.argmax(Q)))
```

Результат:

```
The optimal arm is 3
```

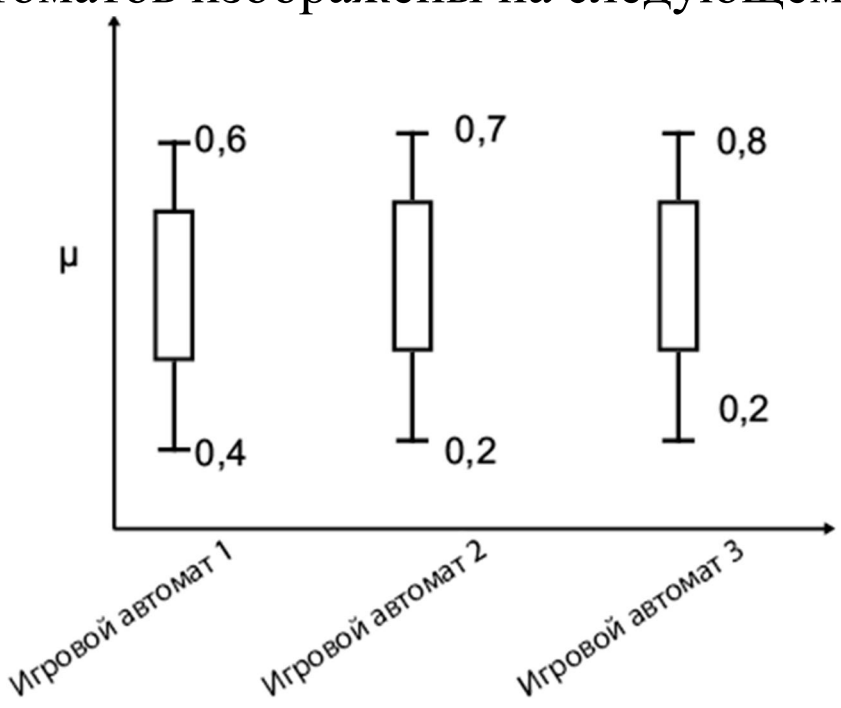
Алгоритм верхней границы доверительного интервала

Эпсилон-жадные и softmax-алгоритмы обеспечивают исследование случайных действий с определенной вероятностью. Исследование разных вариантов может привести к опробованию действий, вообще не несущих хорошей награды. При этом не хотелось бы упустить те хорошие руки, которые принесли низкую награду в начальных итерациях. Для решения таких проблем существует алгоритм *верхней границы доверительного интервала*(*UCB*, Upper Confidence Bound), основанный на принципе оптимизма перед лицом неопределенности.

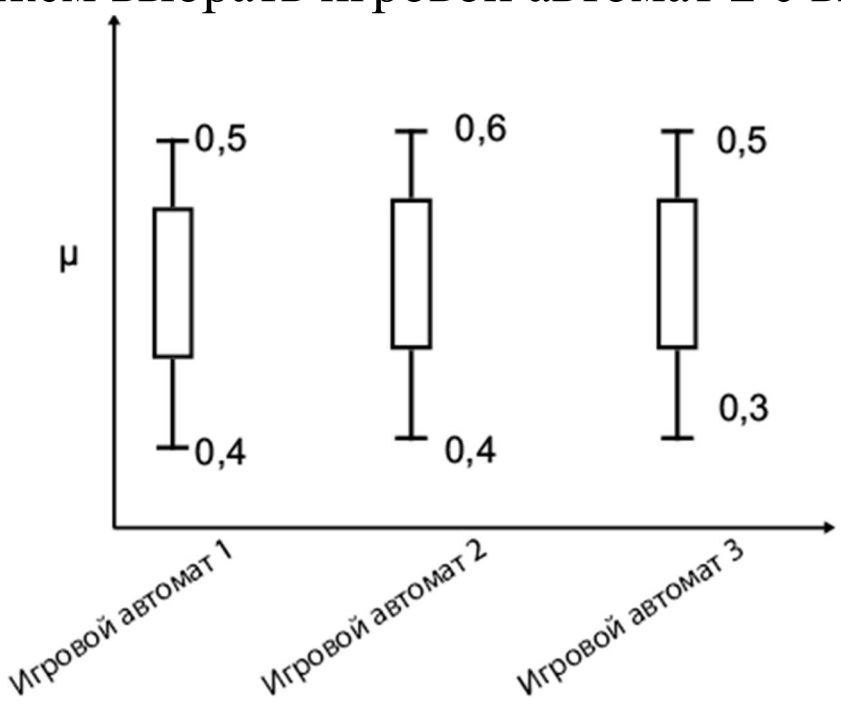
Алгоритм UCB помогает выбрать лучшую руку на основании доверительного интервала. Что это такое? Допустим, есть две руки. Мы опробуем оба варианта; первая рука обеспечивает награду 0,3, а вторая — 0,8. Тем не менее по одной попытке нельзя прийти к выводу, что рука

2 обеспечивает более высокую награду. Нужно попробовать нажать на рычаг несколько раз, вычислить среднее значение награды, полученной для каждой руки, и выбрать вариант с более высоким средним значением. Но как определить правильное среднее значение для каждой из этих рук? На помощь приходит *доверительный интервал* — интервал, в котором лежит средняя награда для руки. Если доверительный интервал руки 1 имеет вид [0,2, 0,9], то это означает, что среднее значение для руки 1 лежит в этом интервале. Значение 0,2 называется *нижней границей доверительного интервала*, а 0,9 называется *верхней границей доверительного интервала*, или *UCB*. Алгоритм UCB выбирает для исследования игровой автомат с более высоким значением UCB.

Допустим, есть три игровых автомата, и на каждом из автоматов вы сыграли 10 раз. Доверительные интервалы для этих трех игровых автоматов изображены на следующем рисунке:



Мы видим, что игровой автомат 3 имеет высокое значение UCB. Тем не менее не стоит делать вывод, что этот игровой автомат обеспечивает хорошую награду, на основании всего 10 попыток. С другой стороны, после нескольких попыток доверительный интервал будет более точным. Таким образом, со временем доверительный интервал сужается к фактическому значению, как показано на следующем рисунке. Итак, мы можем выбрать игровой автомат 2 с высоким значением UCB:



Принцип работы алгоритма UCB очень прост:

1. Выбрать действие (руку) с высокой суммой средней награды и верхней границей доверительного интервала.
2. Нажать на рычаг и получить награду.
3. Обновить награду и границу доверительного интервала для руки.

Но как вычислить UCB?

Для этого можно воспользоваться формулой $\sqrt{\frac{2\log(t)}{N(a)}}$, где $N(a)$ — количество нажатий на рычаг, а t — общее количество раундов.

Итак, в алгоритме UCB рука выбирается по следующей формуле:

$$Arm = \arg\max_a \left[Q(a) + \sqrt{\frac{2\log(t)}{N(a)}} \right].$$

Сначала инициализируются переменные:

```
# Количество раундов (итераций)
num_rounds = 20000

# Количество нажатий на рычаг
count = np.zeros(10)

# Сумма наград для каждой руки
sum_rewards = np.zeros(10)

# Q, то есть средняя награда
Q = np.zeros(10)
```

Затем определяется функция UCB:

```
def UCB(iters):
    ucb = np.zeros(10)
    # Исследовать все руки
    if iters < 10:
        return i
    else:
        for arm in range(10):
            # Вычислить верхнюю границу
            upper_bound =
math.sqrt((2*math.log(sum(count))) / count[arm])
            # Добавить верхнюю границу к
значению Q
            ucb[arm] = Q[arm] + upper_bound
        # Вернуть руку с максимальным значением
        return (np.argmax(ucb))
```

А теперь начинаем нажимать на рычаг:

```
for i in range(num_rounds):
    # Выбрать руку с применением UCB
    arm = UCB(i)
    # Получить награду
```

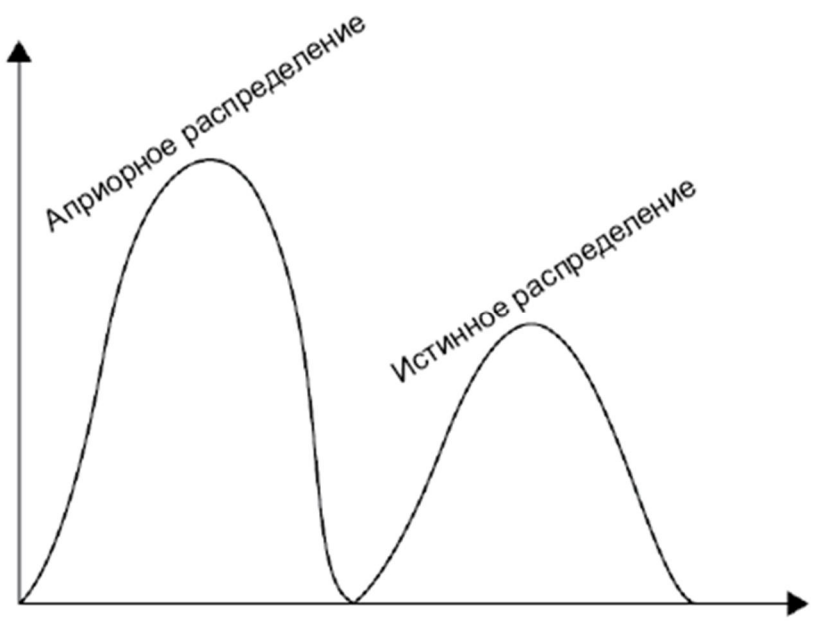
```
        observation, reward, done, info =
env.step(arm)
        # Обновить счетчик для этой руки
        count[arm] += 1
        # Включить в сумму наград от руки
        sum_rewards[arm]+=reward
        # Вычислить Q - среднюю награду от руки
        Q[arm] = sum_rewards[arm]/count[arm]
    print( 'The optimal arm is
{}'.format(np.argmax(Q)))
```

Результат:

```
The optimal arm is 1
```

Алгоритм выборки Томпсона

Выборка Томпсона (*TS*, Thompson sampling) — еще один популярный алгоритм для преодоления дилеммы компромисса между исследованием и эксплуатацией. Это вероятностный алгоритм, основанный на априорном распределении. Стратегия TS очень проста: сначала вычисляется априорная средняя награда для каждой из *k* рук с помощью *n* проб и находятся *k* распределений. Эти исходные распределения не будут совпадать с истинным распределением, поэтому мы будем использовать термин «*априорное распределение*»:

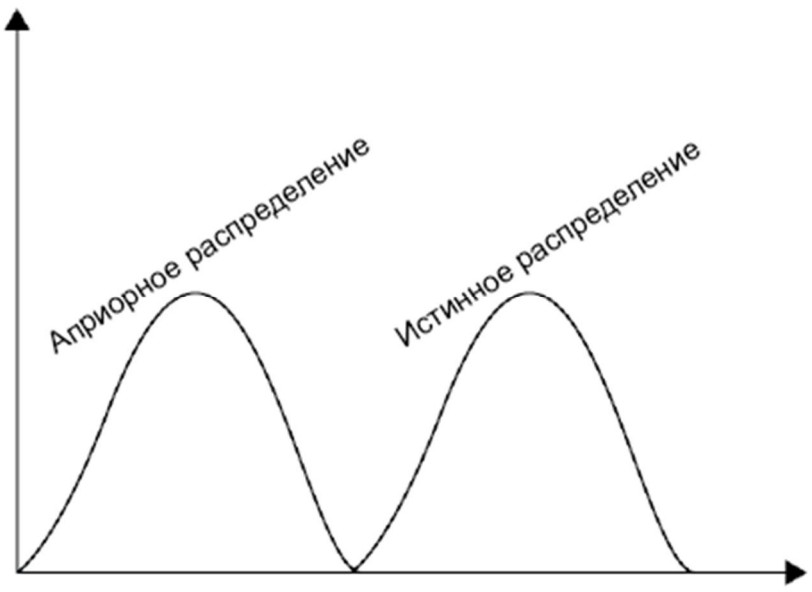


Для вычисления априорного распределения будет использоваться бета-распределение. Значение бета-распределения [*альфа*, *бета*] лежит в интервале [0,1]. *Альфа* представляет количество получений положительной награды, а *бета* — количество получений отрицательной награды.

Теперь посмотрим, как алгоритм TS помогает с выбором лучшей руки. Последовательность действий TS выглядит так:

1. Получить значения каждого *k*распределения. Использовать их в качестве априорного среднего.
2. Выбрать руку с наибольшим априорным средним.
3. Использовать наблюдаемую награду для изменения априорного распределения.

Через несколько раундов априорное распределение будет напоминать истинное распределение:



Реализация TS на Python сделает алгоритм более понятным. Все начинается с инициализации переменных:

```
# Количество раундов (итераций)
num_rounds = 20000

# Количество нажатий на рычаг
count = np.zeros(10)

# Сумма наград для каждой руки
sum_rewards = np.zeros(10)

# Q, то есть средняя награда
Q = np.zeros(10)

# Инициализировать значения alpha и beta
alpha = np.ones(10)
beta = np.ones(10)
```

Затем определяется функция `thompson_sampling`:

```
def thompson_sampling(alpha,beta):
    samples =
[np.random.beta(alpha[i]+1,beta[i]+1) for i in
range(10)]

    return np.argmax(samples)
```

А теперь начинаем играть с многоруким бандитом, используя TS:

```
for i in range(num_rounds):

# Выбрать руку с применением выборки Томпсона
arm = thompson_sampling(alpha,beta)

# Получить награду
observation, reward, done, info = env.step(arm)
```

```
# Обновить счетчик для этой руки
count[arm] += 1

# Включить в сумму наград от руки
sum_rewards[arm]+=reward

# Вычислить Q - среднюю награду от руки
Q[arm] = sum_rewards[arm]/count[arm]

# Если награда положительная, увеличить alpha
if reward >0:
alpha[arm] += 1

# Если награда отрицательная, увеличить beta
  else:
  beta[arm] += 1

  print( 'The optimal arm is
{}'.format(np.argmax(Q)))
```

Результат:

```
The optimal arm is 3
```

Практические применения MAB

До настоящего момента мы рассматривали задачу MAB и возможности ее решения с разными стратегиями исследования. Однако модель многорукого бандита используется не только для игровых автоматов, она находит и другие практические применения.

Бандиты используются как замена для AB-тестирования — одного из самых распространенных классических методов тестирования. Допустим, на вашем сайте используются две версии страницы перехода. Как определить, какая версия больше нравится вашим пользователям? Для этого следует провести AB-тестирование.

В данном методе назначаются два разных временных периода для исследования и для эксплуатации, что влечет за собой значительные потери. Минимизировать потери позволяют различные стратегии, которые актуальны для решения задачи MAB. Вместо того чтобы отдельно выполнять исследование и эксплуатацию, можно воспользоваться адаптивным режимом и провести два процесса одновременно.

Бандиты широко применяются для оптимизации веб-сайтов, максимизации показателя эффективности рекламы, сетевой рекламы, проведения кампаний и т.д. Если вы проводите краткосрочную кампанию, то AB-тестирование потратит почти все ваше время на исследование и эксплуатацию, поэтому в данном случае бандиты будут очень полезны.

Выбор подходящего рекламного баннера с использованием MAB

Допустим, вы управляете веб-сайтом. У вас есть пять разных баннеров одной рекламы, и вы хотите знать, какой баннер привлечет пользователя. Мы смоделируем эту задачу в форме задачи бандита. Допустим, пять баннеров соответствуют пяти рукам бандита; если пользователь щелкает по рекламе, присуждается 1 очко награды, а если нет — 0 очков.

При нормальном AB-тестировании перед выбором лучшего баннера проводится полное исследование всех пяти образцов, которое требует значительной траты времени и ресурсов. Вместо этого мы используем хорошую стратегию исследования и найдем баннер, дающий больше наград (больше переходов).

Как обычно, все начинается с импортирования необходимых библиотек:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

Смоделируем набор данных с размерами 5 × 10 000: столбец определяет тип баннера `Banner_type`, а ячейки содержат 0 или 1 в зависимости от того, щелкнул пользователь на рекламе (`1`) или не щелкнул (`0`):

```
df = pd.DataFrame()
df['Banner_type_0'] =
np.random.randint(0,2,100000)
df['Banner_type_1'] =
np.random.randint(0,2,100000)
df['Banner_type_2'] =
np.random.randint(0,2,100000)
df['Banner_type_3'] =
np.random.randint(0,2,100000)
df['Banner_type_4'] =
np.random.randint(0,2,100000)
```

Просмотрим несколько строк данных:

```
df.head()
```

	Banner type 0	Banner type 1	Banner type 2	Banner type 3	Banner type 4
0	1	1	0	1	1
1	0	1	1	1	0
2	1	1	0	0	1
3	0	0	0	0	1
4	0	1	1	1	1
5	0	1	1	0	1
6	1	0	0	1	1
7	0	1	1	0	1
8	0	0	1	0	1
9	0	0	0	1	0

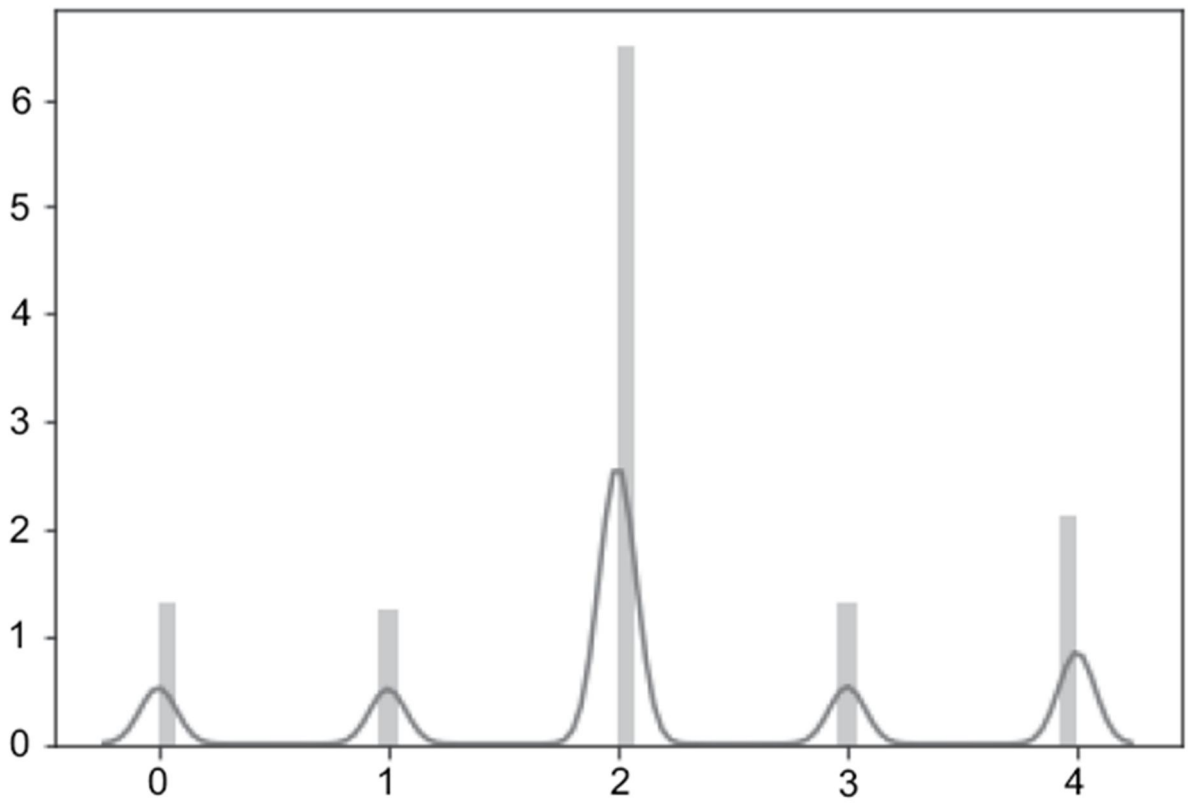
```
num_banner = 5
no_of_iterations = 100000
banner_selected = []
count = np.zeros(num_banner)
Q = np.zeros(num_banner)
sum_rewards = np.zeros(num_banner)
```

Затем определяется эпсилон-жадная стратегия:

```
def epsilon_greedy(epsilon):
    random_value = np.random.random()
    choose_random = random_value < epsilon
    if choose_random:
        action = np.random.choice(num_banner)
    else:
        action = np.argmax(Q)
    return action

for i in range(no_of_iterations):
    banner = epsilon_greedy(0.5)
    reward = df.values[i, banner]
    count[banner] += 1
    sum_rewards[banner]+=reward
    Q[banner] =
sum_rewards[banner]/count[banner]
    banner_selected.append(banner)
```

Мы можем нанести результаты на рисунок и посмотреть, какой баннер приносит наибольшее количество переходов:

```
sns.distplot(banner_selected)
```



Контекстные бандиты

Вы увидели, как бандиты используются для выбора рекламного баннера, наиболее подходящего для большинства пользователей. Тем не менее у разных пользователей предпочтения в области баннеров отличаются. Пользователю *A* нравится баннер типа 1, а пользователь *B* может предпочесть баннер типа 3. А значит, рекламные баннеры должны персонализироваться в зависимости от поведения пользователя. Как это сделать? Для этого будет введена новая разновидность бандитов — *контекстный бандит*.

В обычной задаче MAB агент выполняет действие и получает награду. Для контекстных бандитов, вместо того чтобы просто выполнять действия, мы также учитываем состояние среды. Состояние содержит контекст. В данном случае состояние задается поведением пользователя, поэтому мы выполняем действия (показ рекламы) в зависимости от состояния (поведения пользователя), которое обеспечивает максимальную награду (переходы по рекламной ссылке). По этой причине контекстные бандиты широко применяются для персонализации контента, то есть его соответствия предпочтениям пользователя. Они помогают решить проблемы «холодного старта» в рекомендационных системах. Netflix использует контекстных бандитов для персонализации оформления телевизионных шоу.

Итоги

В этой главе была рассмотрена задача о многоруком бандите (MAB) и возможность ее практического применения в разных областях. Были представлены различные методы разрешения дилеммы компромисса между исследованием и эксплуатацией. Сначала мы рассмотрели эпсилон-жадную стратегию, при которой с вероятностью epsilon = 0 выполняется чистая эксплуатация, а с вероятностью 1 – epsilon — чистое исследование. Мы рассмотрели алгоритм UCB, выбирающий лучшее

действие с максимальным значением верхней границы, и алгоритм TS, выбирающий лучшее действие с использованием бета-распределения.

Следующие главы посвящены глубокому обучению и его применению для решения задач RL.

Вопросы

1. Что собой представляет задача о многоруком бандите (MAB)?
2. Что такое «дилемма компромисса между исследованием и эксплуатацией»?
3. Для чего нужна вероятность epsilon в эпсилон-жадной стратегии?
4. Как решается дилемма компромисса между исследованием и эксплуатацией?
5. Что такое «алгоритм UCB»?
6. Чем выборка Томпсона отличается от алгоритма UCB?

Дополнительные источники

Применение контекстных бандитов для персонализации:`https://www.microsoft.com/en-us/research/blog/contextual-bandit-breakthrough-enables-deeper-personalization/.`

Как Netflix использует контекстных бандитов:`https://medium.com/netflix-techblog/artwork-personalization-c589f074ad76.`

Кооперативная фильтрация с использованием MAB:`https://arxiv.org/pdf/1708.03058.pdf.`

7. Основы глубокого обучения

До настоящего момента вы изучали работу **обучения с подкреплением (RL)**. Сейчас мы займемся **глубоким обучением с подкреплением** (**DRL**, deep reinforcement learning) — сочетанием глубокого обучения и RL. В последнее время DRL все больше интересует специалистов и серьезно влияет на решение многих задач RL. Чтобы понять DRL, необходимо иметь определенную подготовку. Итак, *глубокое обучение* представляет собой подмножество дисциплины машинного обучения, ориентированное на использование нейросетей. Оно существует уже около 10 лет, но обрело популярность только сейчас благодаря достижениям в области вычислительных технологий и доступности огромных объемов данных, в работе с которыми алгоритмы глубокого обучения превосходят по эффективности любые классические алгоритмы машинного обучения. В этой главе будут представлены некоторые алгоритмы глубокого обучения: **рекуррентные нейросети (RNN), долгая краткосрочная память (LSTM) и сверточная нейросеть (CNN))**, а также примеры их практического применения.

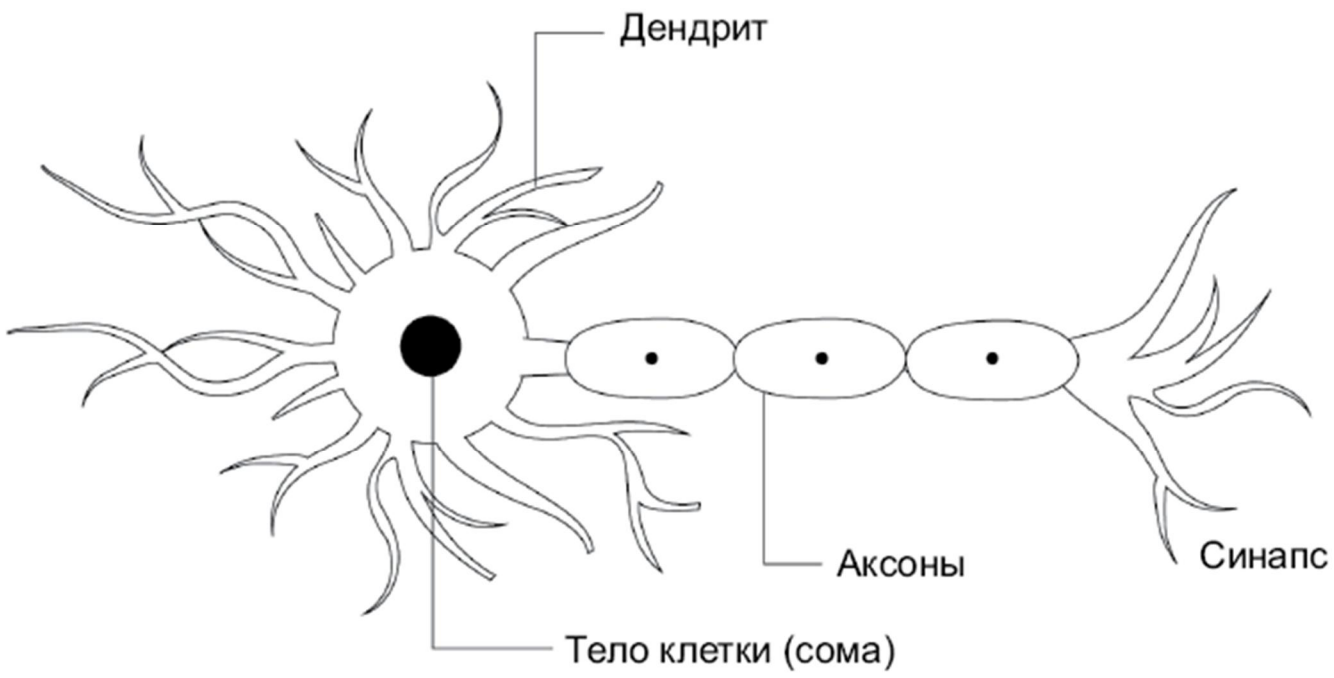
В этой главе рассматриваются следующие темы:

- Искусственные нейроны.

• Искусственные нейросети (ANN).
• Построение нейросети для классификации рукописного написания цифр.
• RNN.
• LSTM.
• Генерирование текстов песен на базе LSTM.
• CNN.
• Классификация модных продуктов с использованием CNN.

Искусственные нейроны

Прежде чем описывать суть ANN, сначала необходимо понять, что такое нейроны и как работают нейроны нашего мозга. *Нейрон* можно определить как основную вычислительную единицу человеческого мозга. Наш мозг состоит приблизительно из сотни миллиардов нейронов, которые связываются друг с другом синапсами и получают входные данные из окружающей среды, от органов чувств или от других нейронов через ветвеобразные структуры, называемые дендритами. Входные данные в разной степени важны для вычисления результата, поэтому каждому из них присваивается весовой коэффициент, после чего они суммируются в соме (теле клетки) и обрабатываются. Из тела клетки они перемещаются по аксонам и передаются другим нейронам. Отдельный биологический нейрон показан на следующем рисунке:



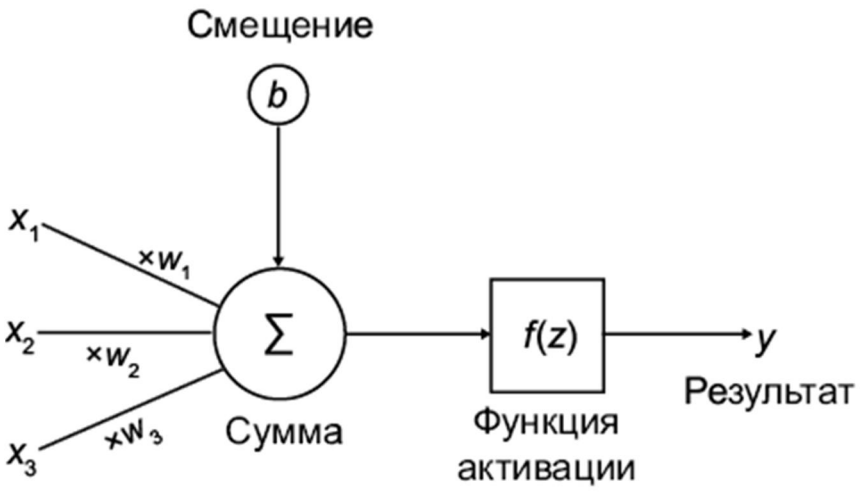
Как же работают искусственные нейроны? Допустим, имеются три входных сигнала x_1, x_2 и x_3, по которым должен прогнозироваться выход y. Входные сигналы умножаются на веса w_1, w_2 и w_3, а затем суммируются: $x_1 \times w_1 + x_2 \times w_2 + x_3 \times w_3$. Но почему входные сигналы умножаются на весовые коэффициенты?

Потому что не все входные сигналы одинаково важны для вычисления выхода y. Предположим, x_2 важнее для вычисления выхода по сравнению с двумя другими сигналами. Тогда w_2 присваивается более высокое значение, чем другим. Таким образом, после умножения на весовой коэффициент значение x_2будет выше значений x_1 и x_3. Далее к полученной сумме добавляется значение b, называемое *смещением*. Таким образом, $z = (x_1 \times w_1 + x_2 \times w_2 + x_3 \times w_3) + b$, или:

$$z = \sum(\text{входы} \times \text{веса}) + \text{смещение}.$$

Вам не кажется, что формула вычисления *z* напоминает уравнение линейной регрессии? Разве это не обычное уравнение прямой: $z = mx + b$, где *m* — веса (коэффициенты), *x* — входные сигналы, а *b* — смещение (точка пересечения с осью)? Да, пожалуй.

Чем тогда нейроны отличаются от линейной регрессии? Они позволяют добавить в результат *z* нелинейность за счет применения функции *f* (), называемой *функцией активации*, или *передаточной функцией*. Таким образом, результат определяется формулой $y = f(z)$. Один искусственный нейрон изображен на следующем рисунке:



В нейронах мы получаем входы *x*, умножаем их на веса *w* и добавляем смещение *b*, после чего применяем к результату функцию активации $f(z)$ и получаем выход *y*.

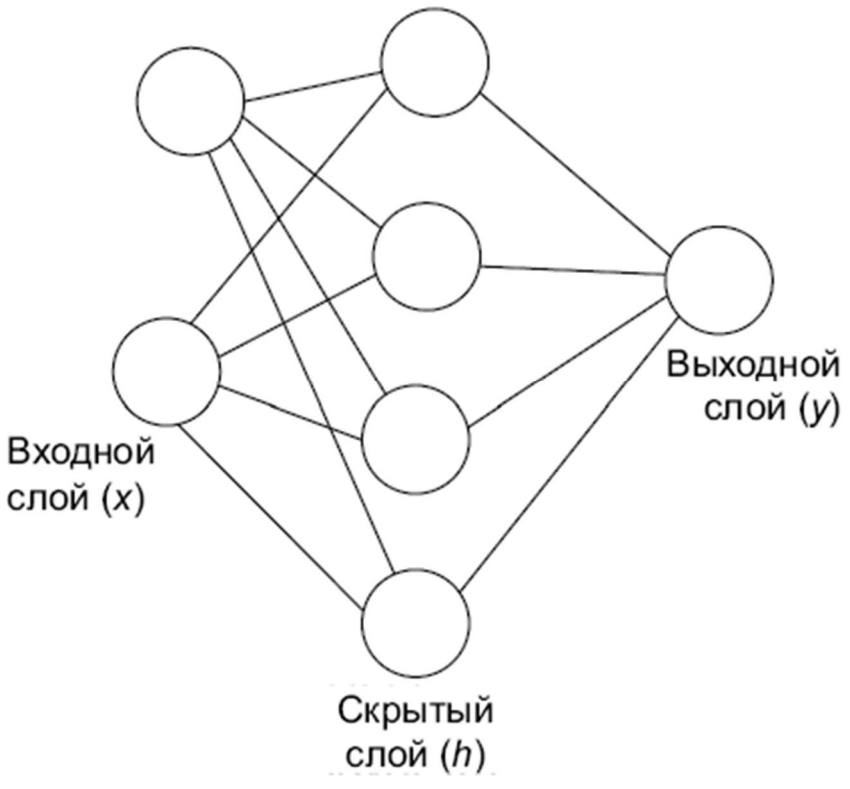
ANN

Нейроны — классная штука, да? Но один нейрон не может выполнять сложные задачи, поэтому наш мозг состоит из миллиардов нейронов, организованных по слоям, образующим сеть. Искусственные нейроны тоже упорядочены по слоям. Все слои соединяются между собой так, чтобы информация передавалась с одного на другой.

Типичная *искусственная нейросеть*, или *ANN* (artificial neural network), состоит из следующих слоев:

- входной слой;
- скрытый слой;
- выходной слой.

Каждый слой состоит из нейронов, при этом нейроны одного слоя способны взаимодействовать только с нейронами других слоев, но не между собой. Типичная структура ANN показана на следующем рисунке:



Входной слой

На входном слое сеть получает входные данные. Количество нейронов здесь равно количеству входных сигналов. Каждый входной сигнал каким-то образом влияет на формирование выхода; он умножается на весовой коэффициент, а при передаче на следующий уровень к сумме прибавляется смещение.

Скрытый слой

Любой слой между входным и выходным называется «скрытым». Здесь обрабатываются данные, полученные с входного слоя, и формируются сложные отношения между входом и выходом. Иначе говоря, на этом слое идентифицируются закономерности в наборе данных. Количество скрытых слоев может быть любым, и мы вольны выбирать количество скрытых слоев в соответствии с задачей. Для очень простой задачи бывает достаточно всего одного скрытого слоя, тогда как для сложных задач, например распознавания образов, используется множество скрытых слоев, на каждом из которых извлекаются признаки изображения, упрощающие распознавание. Сеть, использующая множество скрытых слоев, называется *глубокой нейронной сетью*.

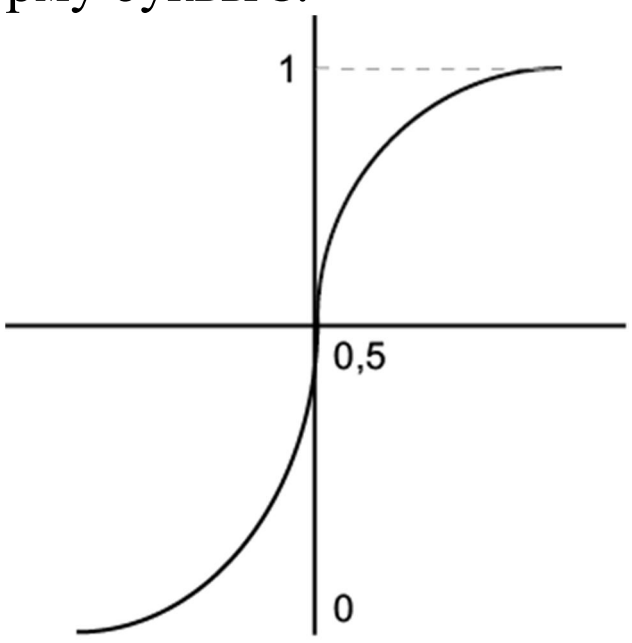
Выходной слой

После обработки входных данных последний скрытый слой отправляет свой результат выходному слою. Как подсказывает само название, выходной слой формирует решение задачи, и количество нейронов здесь зависит от ее типа. Если это задача бинарной классификации, то количество нейронов должно указывать, к какому классу относится вход. Если это задача классификации, скажем, с пятью возможными классами и вы хотите знать вероятность того, что каждый класс представляет выход, то количество нейронов на выходном слое равно пяти — каждый выдает соответствующую вероятность. Если же это задача регрессии, то выходной слой состоит из одного нейрона.

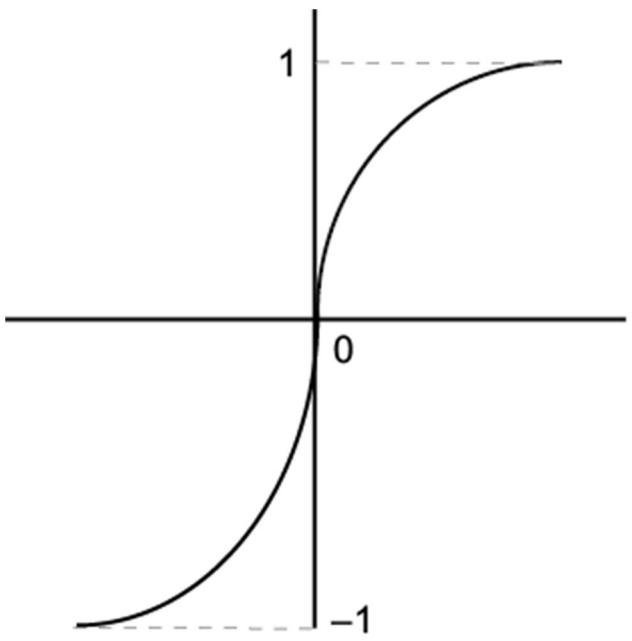
Функции активации

Функции активации используются для введения нелинейности в нейросети и имеют вид $f(z)$, где z = (*входы* × *веса*) + *смещение*. Существуют разные типы функций активации:

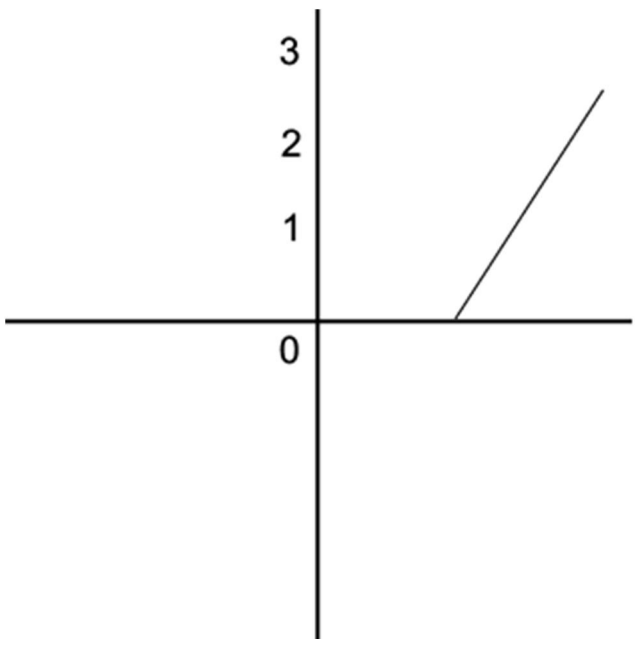
• *Сигмоидальная функция* — одна из самых распространенных. Она масштабирует значение в диапазоне от 0 до 1. Ее формула: $f(z)=\frac{1}{1+e^{-z}}$. Также данная функция может называться логистической. График имеет форму буквы *s*:



• *Функция гиперболического тангенса*, в отличие от сигмоидальной функции, масштабирует значение в диапазоне от –1 до 1. Ее формула: $f(z)=\frac{e^{2z}-1}{e^{2z}+1}$. График также имеет форму буквы *s*, но его центр находится в нулевой точке:



• *Функция ReLU*, или функция *блока линейной ректификации*(rectified linear unit), тоже широко распространена. Ее формула: $f(z) = \max(0, z)$. Значение $f(z)$ равно 0, если $z < 0$, и равно z, если $z \geq 0$:



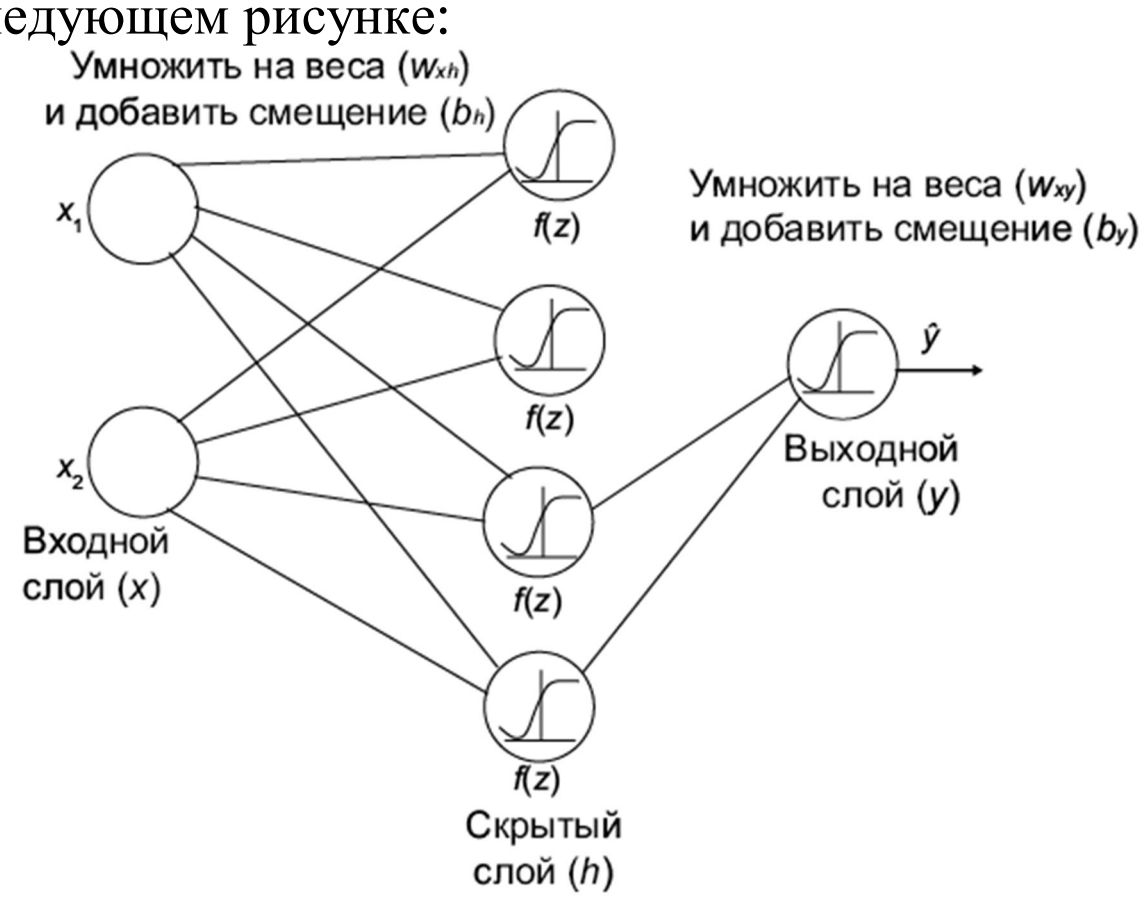
• *Softmax-функция* является обобщенной формой сигмоидальной функции. Она обычно применяется на последнем слое сети при выполнении задач классификации с множеством классов. Она задает вероятности того, что каждый класс является выходным, а следовательно, сумма softmax-значений всегда равна 1. Ее формула:

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

.

Подробнее об ANN

Мы знаем, что в искусственных нейронах входные сигналы умножаются на веса, к сумме прибавляется смещение, а потом функция активации генерирует выход. Теперь вы увидите, как это происходит в нейронных сетях, в которых нейроны организованы по слоям. Количество слоев в сети равно количеству скрытых и выходных слоев. Входной слой в расчет не принимается. Рассмотрим двухуровневую нейросеть с одним входным, одним скрытым и одним выходным слоем, как показано на следующем рисунке:



Допустим, есть два входных значения x_1 и x_2, для которых требуется вычислить выходное значение y. С двумя входами количество нейронов на входном слое будет равно 2. Эти входы умножаются на веса, к сумме прибавляется смещение, а полученное значение передается на скрытый слой, где будет применена функция активации. Таким образом, первым

делом необходимо инициализировать матрицу весов. В реальном мире мы не знаем, какое входное значение по-настоящему важно и должно получить высокий вес для вычисления выхода. По этой причине мы выполняем случайную инициализацию весов и смещения. Обозначим веса и смещение, переходящие с входного слоя на скрытый, w_{xh} и b_hсоответственно. Размеры матрицы должны быть равны *[количество нейронов на текущем слое × количество нейронов на следующем слое]*. Почему? По базовому правилу умножения матриц: для произведения двух матриц *AB* количество столбцов в матрице *A* должно быть равно количеству строк в матрице *B*. Таким образом, размеры матрицы весов w_{xh}равны *[количество нейронов на входном слое × количество нейронов на скрытом слое]*, то есть 2 × 4:

$z_1 = xw_{xh} + b.$

Таким образом, z_1 = (*входы* × *веса*) + *смещение*. Теперь результат передается скрытому слою. Здесь к z_1применяется функция активации. Рассмотрим следующую сигмоидальную функцию:

$a_1 = \sigma(z_1).$

После применения функции активации результат a_1 умножается на новую матрицу весов и прибавляется новое значение смещения. Эту матрицу весов и смещение можно обозначить w_{hy} и b_y соответственно. Размеры этой матрицы весов будут равны *[количество нейронов на скрытом слое × количество нейронов на выходном слое]*. Так как в нашем примере на скрытом слое находятся четыре нейрона, а на выходном — один, размеры матрицы w_{hy} будут равны 4 × 1. Мы умножаем a_1 на матрицу весов w_{hy}, прибавляем смещение b_y и передаем результат на следующий — выходной — слой:

$z_2 = a_1xw_{hy} + b_y.$

Теперь на выходном слое сигмоидальная функция применяется к z_2, что дает результат

$\hat{y} = \sigma(z_2).$

Весь процесс от входного до выходного слоя называется *прямым распространением*:

```
    def forwardProp():
        z1 = np.dot(x,wxh) + bh
        a1 = sigmoid(z1)
        z2 = np.dot(a1,why) + by
        yHat = sigmoid(z2)
```

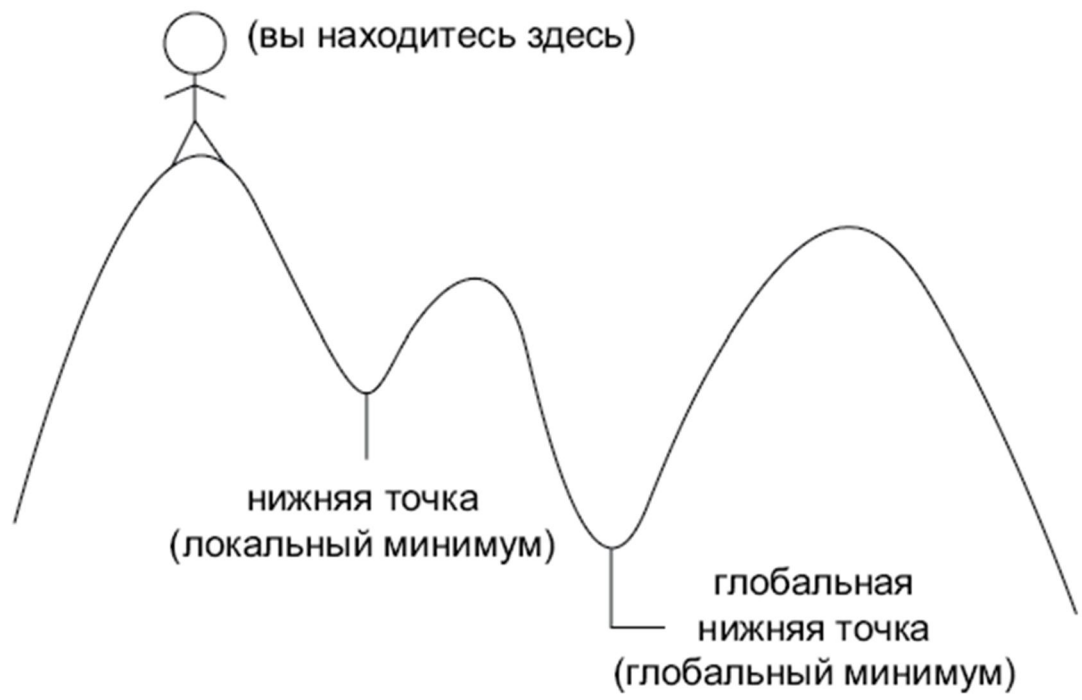
Прямое распространение — отличная штука, не так ли? Но как проверить на правильность выход, сгенерированный нейросетью? Необходимо определить новую функцию, которая называется *функцией стоимости* (*J*), или *функцией потерь*; она сообщит, насколько эффективно работала нейросеть. Есть много разных функций стоимости. Мы будем использовать *среднеквадратическую ошибку*, которая может быть определена как половина квадрата разности между фактическим (*y*) и прогнозируемым ($\hat{y}$) значением:

$$J = \frac{1}{2}(y - \hat{y})^2.$$

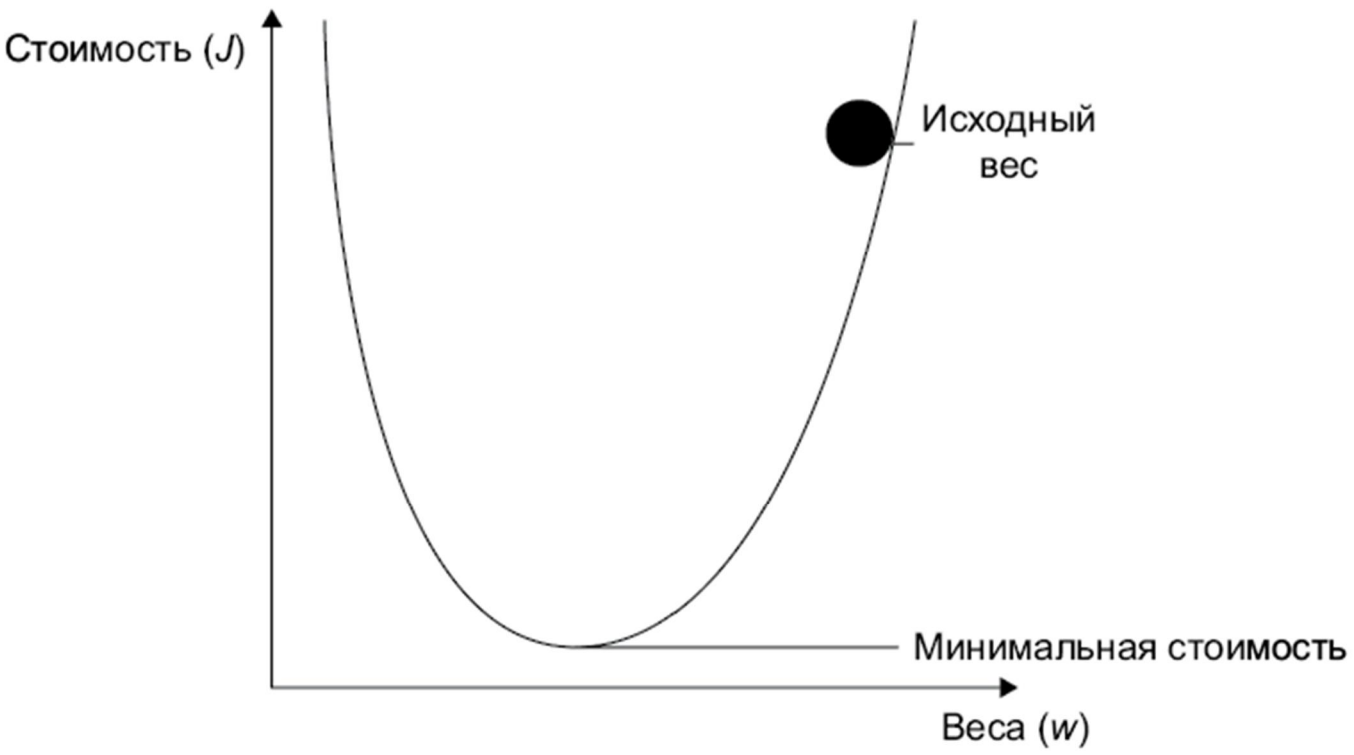
Наша цель — минимизировать функцию стоимости для повышения качества прогнозов нейросети. Как это сделать? Например, изменением некоторых значений в прямом распространении. Очевидно, изменению подлежат не вход и не выход; остаются значения весов и смещения. Матрицы весов и смещения были инициализированы случайным образом, так что результат идеальным не будет. Мы отрегулируем матрицы весов (w_{xh} и w_{hy}) так, чтобы нейросеть давала хороший результат. Как? Вашему вниманию предлагается новый метод, который называется *градиентным спуском*.

Градиентный спуск

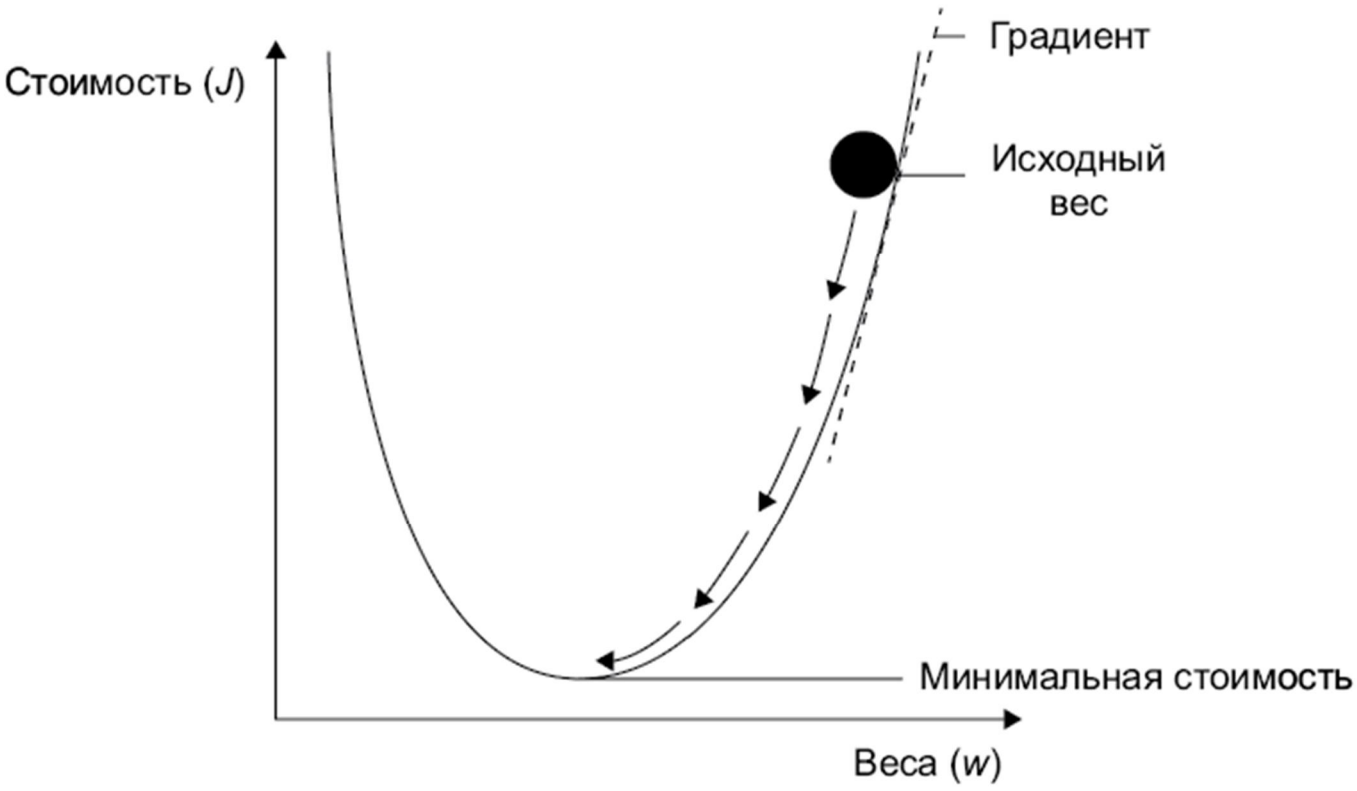
В результате прямого распространения мы оказались на выходном слое. Теперь проведем обратное распространение сети с выходного слоя до входного и обновим веса, вычислив градиент функции стоимости по отношению к весам для минимизации ошибки. Непонятно? Начнем с аналогии. Представьте, что вы стоите на вершине холма, как на следующем рисунке, и хотите достичь его нижней точки. Для этого нужно пройти вниз по холму. Возможно, по пути вам будут попадаться области, которые могут показаться нижними точками холма, но вам нужно добраться до *самой* нижней точки из всех. Другими словами, вы не должны застрять где-то на склоне холма, полагая, что это глобальная нижняя точка:



Аналогичным образом можно представить и функцию стоимости. Ниже изображен график стоимости относительно весов. Наша цель — минимизировать функцию стоимости, поэтому необходимо достичь самой нижней точки (с минимальной стоимостью). Точкой обозначен исходный вес (то есть наше текущее положение на холме). Смещая точку вниз, мы можем достичь положения с минимальной ошибкой, то есть самой нижней точки функции стоимости (глобальной нижней точки на холме):



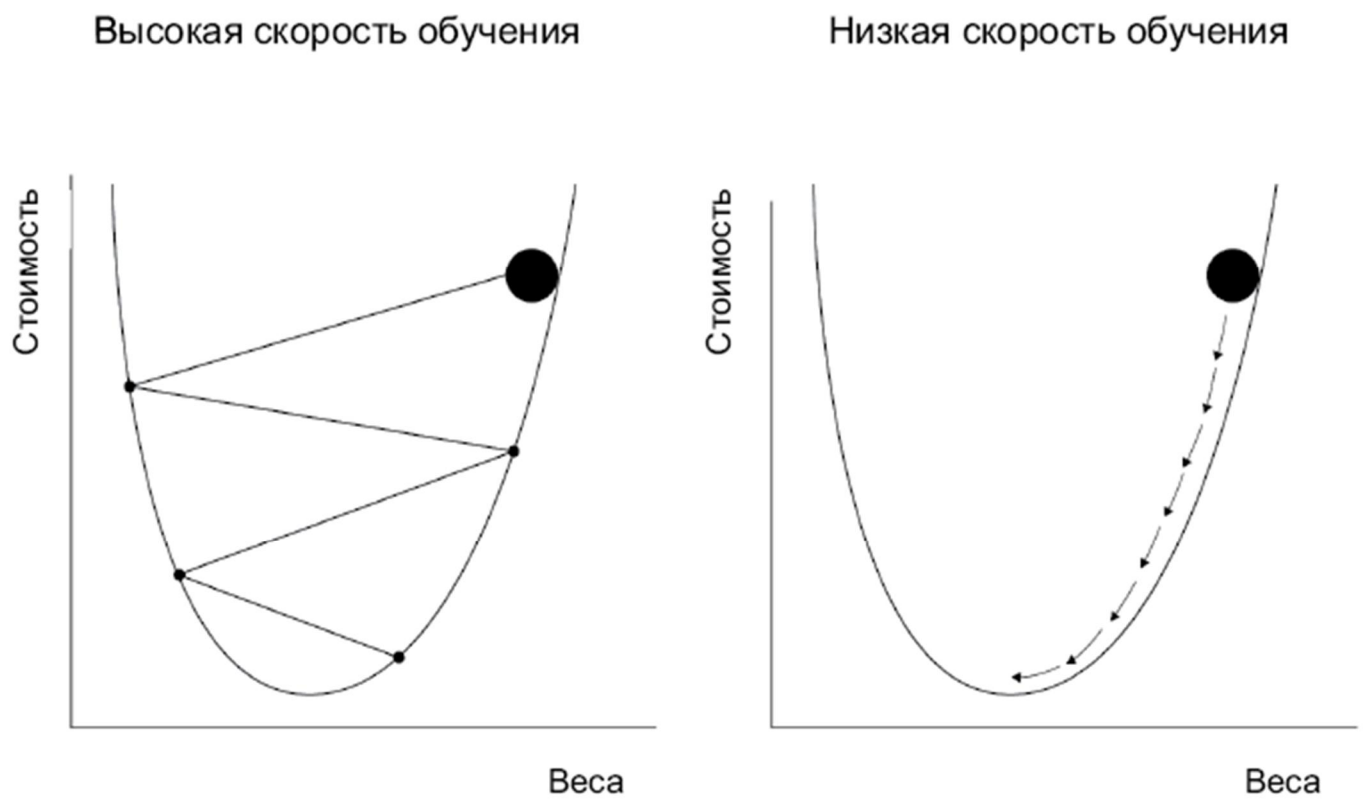
Как смещать точку (исходный вес) вниз? Как спускаться и достичь самой нижней точки? Можно делать это, вычисляя градиент функции стоимости для точки. *Градиенты*представляют собой производные, которые в действительности определяют угол наклона касательной. На следующем рисунке показан такой спуск:



После вычисления градиентов старые веса обновляются по следующему правилу:

$$w = w - \alpha \frac{\partial J}{\partial w}.$$

Что такое α? Этот коэффициент известен как *скорость обучения*. Если скорость обучения мала, мы делаем небольшой шаг вниз, и градиентный спуск происходит медленно. Если же скорость обучения велика, шаг будет большим, а спуск будет происходить быстро, но, возможно, вам не удастся достичь глобального минимума и вы застрянете в локальном. Итак, оптимальная скорость обучения должна выбираться по следующей схеме:



Рассмотрим происходящее с математической точки зрения. Сейчас я приведу интересные выкладки, так что самое время припомнить курс матанализа. Итак, есть два значения: w_{xh} для входных весов и w_{hy} — для выходных. Эти веса необходимо обновить в соответствии с правилом обновления весов. Для этого сначала необходимо вычислить производную функции стоимости по весам. Так как распространение происходит в обратном направлении (то есть мы идем от выходного слоя к входному), сначала займемся w_{hy} — вычислим производную J по w_{hy}.

Как это сделать? Вспомните функцию стоимости $J = ½(y – \hat{y})$. Вычислить производную напрямую невозможно, потому что w_{hy} в J не встречается.

Вспомните формулы прямого распространения, приведенные в следующем виде:

$\hat{y} = \sigma(z_2)$,

$z_2 = a_1 x w_{hy} + b_y$.

Сначала вычисляется частная производная по $\hat{y}$, а затем из $\hat{y}$ вычисляется частная производная по z_2. Из z_2 можно напрямую вычислить производную w_{hy}.

Таким образом, уравнение принимает вид:

$$\frac{\partial J}{\partial w_{hy}} = \frac{\partial J}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial z_2} \times \frac{dz_2}{dw_{hy}} \text{ . } \mathbf{(7.1)}$$

Вычислим каждую из составляющих:

$$\frac{\partial J}{\partial \hat{y}} = (y - \hat{y}),$$

$$\frac{\partial J}{\partial z_2} = \sigma',$$

где σ' — производная сигмоидальной функции активации. Мы знаем, что сигмоидальная функция имеет вид: $\sigma = \frac{1}{1+e^{-z}}$, поэтому формула ее производной:

$$\sigma' = \frac{e^{-z}}{(1+e^{-z})^2}.$$

$$\frac{dz_2}{dw_{hy}} = a_1.$$

Все эти значения будут подставлены в первую формулу **(7.1)**.

Теперь нужно вычислить производную J по новому весу w_{xh}. И снова напрямую взять производную Jпо w_{xh} не получится, потому что w_{xh} в Jне встречается. Следовательно, придется снова воспользоваться формулой сложной производной; вспомните последовательность действий прямого распространения:

$\hat{y} = \sigma(z_2)$;

$z_2 = a_1 x w_{hy} + b_y$;

$a_1 = \sigma(z_1)$;

$z_1 = x w_{hy} + b$.

Теперь формула вычисления градиента для веса w_{xh} выглядит так:

$$\frac{\partial J}{\partial w_{xh}} = \frac{\partial J}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial z_2} \times \frac{\partial z_2}{\partial a_1} \times \frac{\partial a_1}{\partial z_1} \times \frac{dz_1}{dw_{xh}}. \textbf{(7.2)}$$

Вычислим каждую из составляющих:

$$\frac{\partial J}{\partial \hat{y}} = (y - \hat{y});$$

$$\frac{\partial J}{\partial z_2} = \sigma';$$

$$\frac{\partial z_2}{\partial a_1} = w_{hy};$$

$$\frac{\partial a_1}{\partial z_1} = \sigma';$$

$$\frac{dz_1}{dw_{xh}} = x.$$

После того как будут вычислены градиенты для обоих весов, предыдущие веса будут обновлены по правилу обновления весов.

А теперь займемся программированием. Взгляните на уравнения **(7.1)** и **(7.2)** $\frac{\partial J}{\partial \hat{y}}$ и $\frac{\partial J}{\partial z_2}$. Вычислять их снова и снова не нужно.

Определим переменную с именем `delta3`:

```
delta3 = np.multiply(-(y-yHat),sigmoidPrime(z2))
```

Теперь вычислим градиент для w_{hy}:

```
dJ_dWhy = np.dot(a1.T,delta3)
```

И градиент для w_{xh}:

```
delta2 = np.dot(delta3,Why.T)*sigmoidPrime(z1)
dJ_dWxh = np.dot(X.T,delta2)
```

Обновим веса по правилу обновления весов:

```
Wxh += -alpha * dJ_dWhy
Why += -alpha * dJ_dWxh
```

Полный код функции обратного распространения:

```
def backProp():
        delta3 = np.multiply(-(y-
yHat),sigmoidPrime(z2))
```

```
            dJdW2 = np.dot(a1.T, delta3)
            delta2 =
np.dot(delta3,Why.T)*sigmoidPrime(z1)
            dJdW1 = np.dot(X.T, delta2)
            Wxh += -alpha * dJdW1
            Why += -alpha * dJdW2
```

Прежде чем двигаться дальше, необходимо познакомиться с некоторыми часто встречающимися терминами из области нейросетей.

• *Прямой проход* — прямое распространение с входного слоя на выходной.

• *Обратный проход* — обратное распространение с выходного слоя на входной.

• *Эпоха* — сколько раз нейросеть видит все обучающие данные. Можно сказать, что для всех обучающих данных одна эпоха равна одному прямому и одному обратному проходам.

• *Размер пакета* — количество обучающих точек данных, используемых в одном прямом и в одном обратном проходах.

• *Количество итераций* — количество проходов (*один проход =один прямой проход + один обратный проход*).

Допустим, доступны 12 000 обучающих точек данных, а размер пакета составляет 6000. На завершение одной эпохи потребуются две итерации. При первой итерации передаются первые 6000 точек данных и выполняется прямой и обратный проход; при второй итерации передаются следующие 6000 точек данных и выполняется прямой и обратный проход. После двух итераций нейросеть увидит все 12 000 обучающих точек данных, что составит одну эпоху.

Нейросети в TensorFlow

А теперь посмотрим, как создать простую нейросеть, распознающую рукописные цифры, на базе TensorFlow. Мы воспользуемся популярным набором данных MNIST, который содержит подборку снабженных метками изображений рукописных цифр для обучения.

Сначала необходимо импортировать TensorFlow и загрузить набор данных **`tensorflow.examples.tutorial.mnist`**:

```
  import tensorflow as tf
  from tensorflow.examples.tutorials.mnist import
input_data
  mnist = input_data.read_data_sets("/tmp/data/",
one_hot=True)
```

Посмотрим, какая информация содержится в данных:

```
  print("No of images in training set
{}".format(mnist.train.images.shape))
  print("No of labels in training set
{}".format(mnist.train.labels.shape))
```

```
  print("No of images in test set
{}".format(mnist.test.images.shape))
  print("No of labels in test set
{}".format(mnist.test.labels.shape))
```

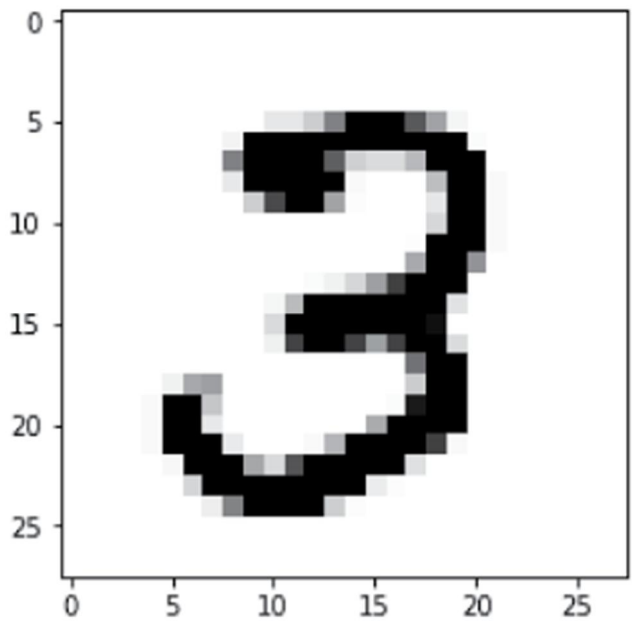
Результат выполнения выглядит так:

```
No of images in training set (55000, 784)
No of labels in training set (55000, 10)
No of images in test set (10000, 784)
No of labels in test set (10000, 10)
```

Обучающий набор содержит 55 000 изображений, каждое изображение имеет размер 784. Также набор содержит 10 меток (значения от 0 до 9). Тестовый набор состоит из 10 000 изображений.

Выведем входное изображение, чтобы понять, как оно выглядит:

```
img1 = mnist.train.images[41].reshape(28,28)
plt.imshow(img1, cmap='Greys')
```



Переходим к построению нейросети. Она будет двухслойной: с одним входным слоем, одним скрытым слоем и одним выходным слоем, который спрогнозирует рукописную цифру.

Сначала определим заместителей для входа и выхода. Так как размер входных данных равен 784, заместителя для входа можно определить так:

```
x = tf.placeholder(tf.float32, [None, 784])
```

Что здесь обозначает `None`? Количество переданных точек данных (размер пакета), которое будет определяться динамически во время выполнения.

Так как выход делится на `10`классов, определение выходного заместителя может выглядеть так:

```
y = tf.placeholder(tf.float32, [None, 10]
```

Затем инициализируем гиперпараметры:

```
learning_rate = 0.1
epochs = 10
batch_size = 100
```

Далее веса и смещение между входным и скрытым слоями определяются в переменных `w_xh` и `b_h` соответственно. Матрица весов

инициализируется значениями, случайно выбранными из нормального распределения со стандартным отклонением 0,03:

```
  w_xh = tf.Variable(tf.random_normal([784, 300], stddev=0.03), name='w_xh')
  b_h = tf.Variable(tf.random_normal([300]), name='b_h')
```

Затем веса и смещение между скрытым и выходным слоями определяются в переменных `w_hy` и `b_y` соответственно:

```
  w_hy = tf.Variable(tf.random_normal([300, 10], stddev=0.03), name='w_hy')
  b_y = tf.Variable(tf.random_normal([10]), name='b_y')
```

Выполним обратное распространение. Вспомните, какие действия для этого необходимы:

```
  z1 = tf.add(tf.matmul(x, w_xh), b_h)
  a1 = tf.nn.relu(z1)
  z2 = tf.add(tf.matmul(a1, w_hy), b_y)
  yhat = tf.nn.softmax(z2)
```

Мы определим функцию стоимости как *потерю перекрестной энтропии*, также известную как *логарифмическая потеря*, в следующем виде:

$$-\sum_i y_i \log(\hat{y}_i),$$

где y_i — фактическое значение, а $\hat{y}_i$ — прогнозируемое значение:

```
  cross_entropy = tf.reduce_mean(-tf.reduce_sum(y * tf.log(yhat),
  reduction_indices=[1]))
```

Наша цель — минимизация функции стоимости. Один из способов сделать это заключается в обратном распространении сети с выполнением градиентного спуска. С TensorFlow вычислять градиенты вручную не придется, можно воспользоваться встроенной функцией-оптимизатором:

```
  optimiser =
  tf.train.GradientDescentOptimizer(learning_rate=learning_rate).minimize(cro
  ss_entropy)
```

Для оценки модели необходимо вычислить точность:

```
  correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(yhat, 1))
  accuracy =
tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

Как вы уже видели, работа TensorFlow основана на построении графа вычислений, и написанный нами код будет выполняться только после запуска сеанса TensorFlow. Давайте займемся этим.

Начнем с инициализации переменных TensorFlow:

```
init_op = tf.global_variables_initializer()
```

Запустим сеанс TensorFlow и начнем обучение модели:

```
with tf.Session() as sess:
    sess.run(init_op)
    total_batch = int(len(mnist.train.labels) /
batch_size)
    for epoch in range(epochs):
        avg_cost = 0
        for i in range(total_batch):
            batch_x, batch_y =
  mnist.train.next_batch(batch_size=batch_size)
            _, c = sess.run([optimiser,
cross_entropy],
                            feed_dict={x: batch_x,
y: batch_y})
            avg_cost += c / total_batch
        print("Epoch:", (epoch + 1), "cost
=""{:.3f}".format(avg_cost))
    print(sess.run(accuracy, feed_dict={x:
mnist.test.images, y:
  mnist.test.labels}))
```

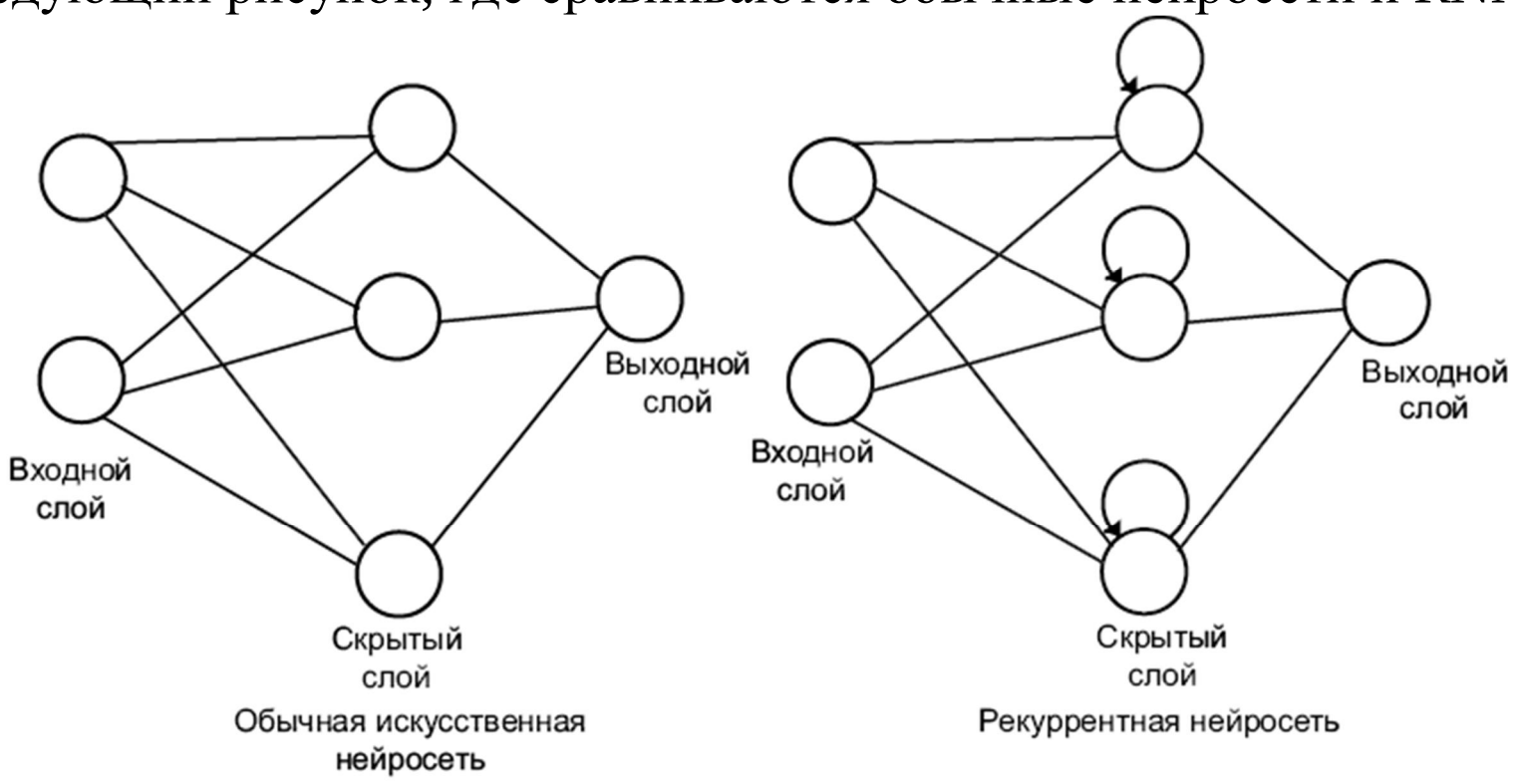
RNN

Птицы летают в ____. Если я попрошу вас вписать недостающее слово, вы, скорее всего, предложите «небе». Почему? Вы прочитали всю последовательность и подобрали слово на основании понимания контекста предложения. Если же дать такое задание обычной нейросети, она не справится. Дело в том, что ее выход основан только на текущем входе, то есть на предыдущем элементе «в». Поскольку входы не зависят друг от друга, нейросеть не сможет работать в ситуациях, когда нужно запомнить последовательность входов для прогнозирования следующего.

Как заставить сеть запомнить всю последовательность для прямого прогнозирования? На помощь приходит *рекуррентная нейросеть*, или *RNN*. Она прогнозирует выход на основании не только текущего входа, но и предыдущего скрытого состояния. Возникает вопрос: почему выбрано предыдущее скрытое состояние, а не предыдущий вход? Дело в том, что предыдущий вход хранит информацию о предыдущем слове, тогда как предыдущее скрытое состояние хранит информацию обо всем предложении, то есть помнит контекст, который необходим для прогнозирования.

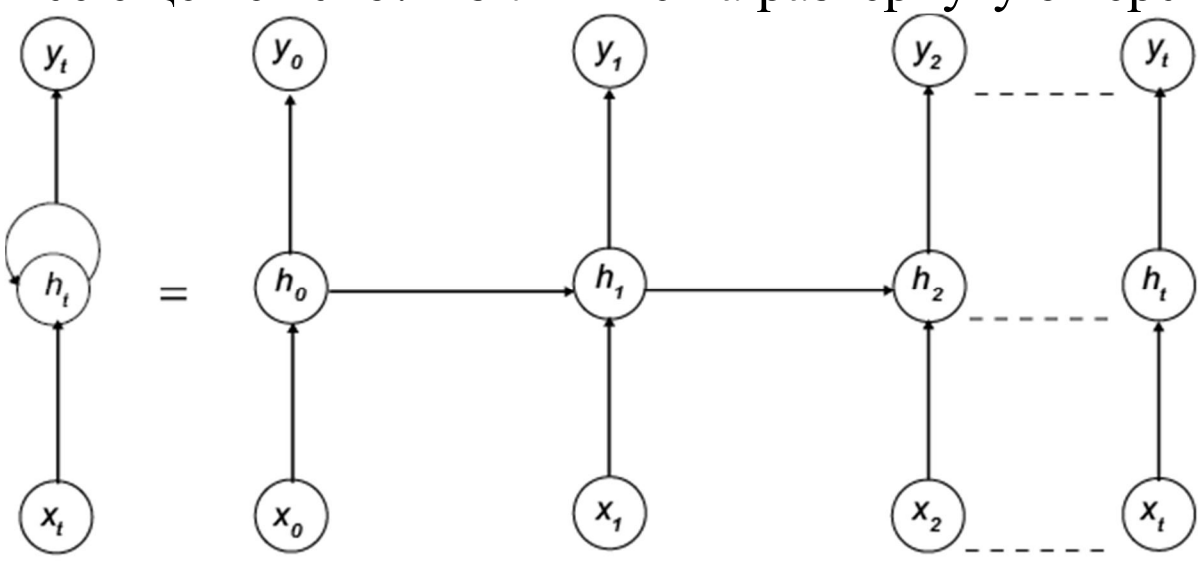
RNN — особая разновидность нейросетей, широко применяемая в работе с последовательными данными, или, точнее, с данными, для которых важно последовательное упорядочение. По сути, RNN обладает памятью, в которой хранится предыдущая информация. Эта

разновидность нейросетей широко используется в решении различных задач обработки естественного языка (NLP): машинном переводе, анализе эмоциональной окраски высказываний и т.д. Также RNN находят применение в анализе данных временных рядов — например, на биржевых рынках. Все еще не ясно, что такое RNN? Взгляните на следующий рисунок, где сравниваются обычные нейросети и RNN:



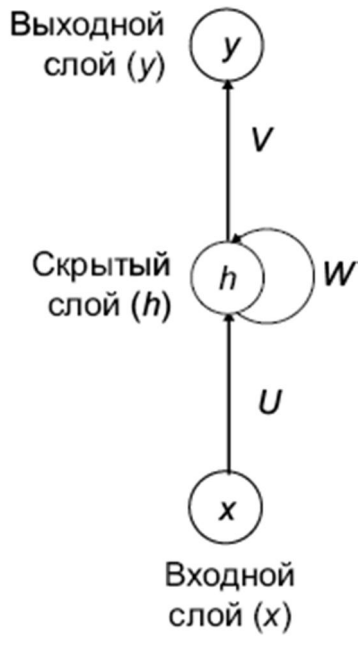
Заметили отличия? В скрытых состояниях RNN содержит цикл, влияющий на процесс вычисления выхода.

Все еще не ясно? Взгляните на развернутую версию RNN:



Как видите, выход y_1прогнозируется на основании текущего входа x_1, текущего скрытого состояния h_1 и предыдущего скрытого состояния h_0. Также посмотрите, как вычисляется выход y_2: для этого используется текущий вход x_2, текущее скрытое состояние h_2 и предыдущее скрытое состояние h_1. Так работают сети RNN: они предсказывают выход на основании как текущего входа, так и предыдущего скрытого состояния. Эти скрытые состояния можно назвать памятью, так как в них хранится информация о том, что видела сеть до настоящего момента.

А теперь немного математики:



На этом рисунке:

• *U* — матрица весов для перехода из входного слоя на скрытое состояние;

• *W* — матрица весов для перехода из одного скрытого состояния в другое;

• *V* — матрица весов для перехода из скрытого состояния на выходной слой.

Итак, при прямом проходе вычисления происходят по следующей формуле:

$h_t = \phi\,(Ux_t + Wx_{t-1})$.

Скрытое состояние в момент времени t = tanh([*матрица весов для перехода из входного слоя в скрытое состояние* × *вход*] + [*матрица весов для перехода из одного скрытого состояния в другое* × *предыдущее скрытое состояние в момент t–1*]):

$\hat{y}_t = \sigma(vh_t)$.

Иначе говоря, *вывод в момент времени t* = *сигмоидальная функция*(*матрица весов для перехода из скрытого состояния в выходной слой* × *скрытое состояние на момент t*).

Также функцию потерь можно определить как потерю перекрестной энтропии:

Потери = $-y_t \log \hat{y}_t$;

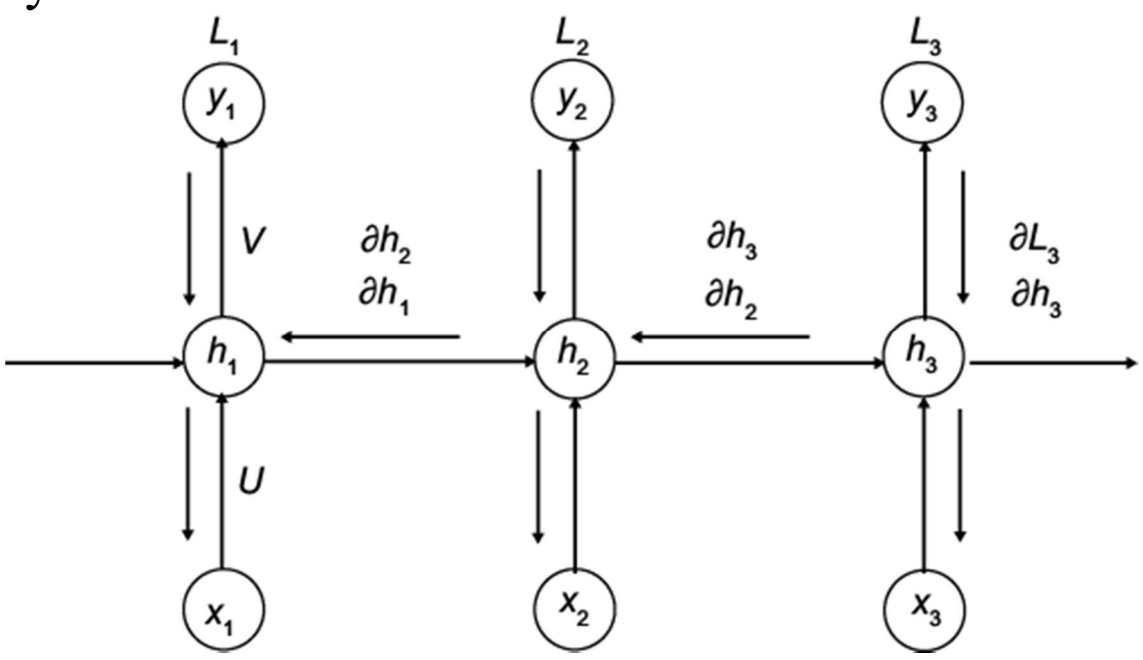
$$\text{Суммарные потери} = \sum_t y_t \log \hat{y}_t.$$

В приведенном примере y_t — фактическое слово в момент времени *t*, а $\hat{y}_t$ — прогнозируемое слово в момент времени *t*. Поскольку в качестве обучающей точки данных мы используем всю последовательность, общие потери будут равны сумме потерь на каждом временном шаге.

Обратное распространение во времени

Как обучать RNN? Можно воспользоваться обратным распространением. Но поскольку в RNN существует зависимость от всех временных шагов, градиенты каждого выхода будут зависеть не только от текущего временного шага, но и от предыдущего. Будем называть этот механизм *обратным распространением во времени* (*BPTT*, backpropagation through time). По сути, это то же обратное распространение, но с необходимыми для RNN особенностями. Чтобы

понять, как оно происходит, рассмотрим развернутую версию RNN на рисунке ниже.



L_1, L_2 или L_3 — потери на каждом временном шаге. Теперь необходимо вычислить градиенты потерь в отношении матриц весов U, V и W. По аналогии с тем, как мы ранее вычисляли общие потери суммированием потерь, матрицы весов обновляются суммой градиентов на каждом временном шаге:

$$\frac{\partial L}{\partial V} = \sum_t \frac{\partial L_t}{\partial V}.$$

Тем не менее у этого метода есть недостаток. Вычисления градиента должны производиться с учетом функции активации. При использовании сигмоидальной функции или функции гиперболического тангенса градиент будет очень малым. При обратном распространении сети по многим временным шагам и перемножении градиентов последние будут становиться все меньше и меньше. Это называется *проблемой исчезающего градиента*. Из-за того что градиенты исчезают со временем, мы не можем извлечь информацию о долгосрочных зависимостях — другими словами, RNN не способна хранить информацию долго.

Проблема исчезающего градиента встречается не только в RNN, но и в других глубоких сетях с большим количеством скрытых слоев при использовании сигмоидальной функции или функции гиперболического тангенса. Также существует *проблема взрывного градиента*, когда значения градиента превышают 1. Перемножение таких градиентов приводит к очень большим числам.

Одно из возможных решений — использование ReLU в качестве функции активации. Однако существует разновидность RNN, которая называется *LSTM* и может эффективно решать проблему исчезающего градиента. В следующем разделе вы увидите, как это работает.

RNN с долгой краткосрочной памятью

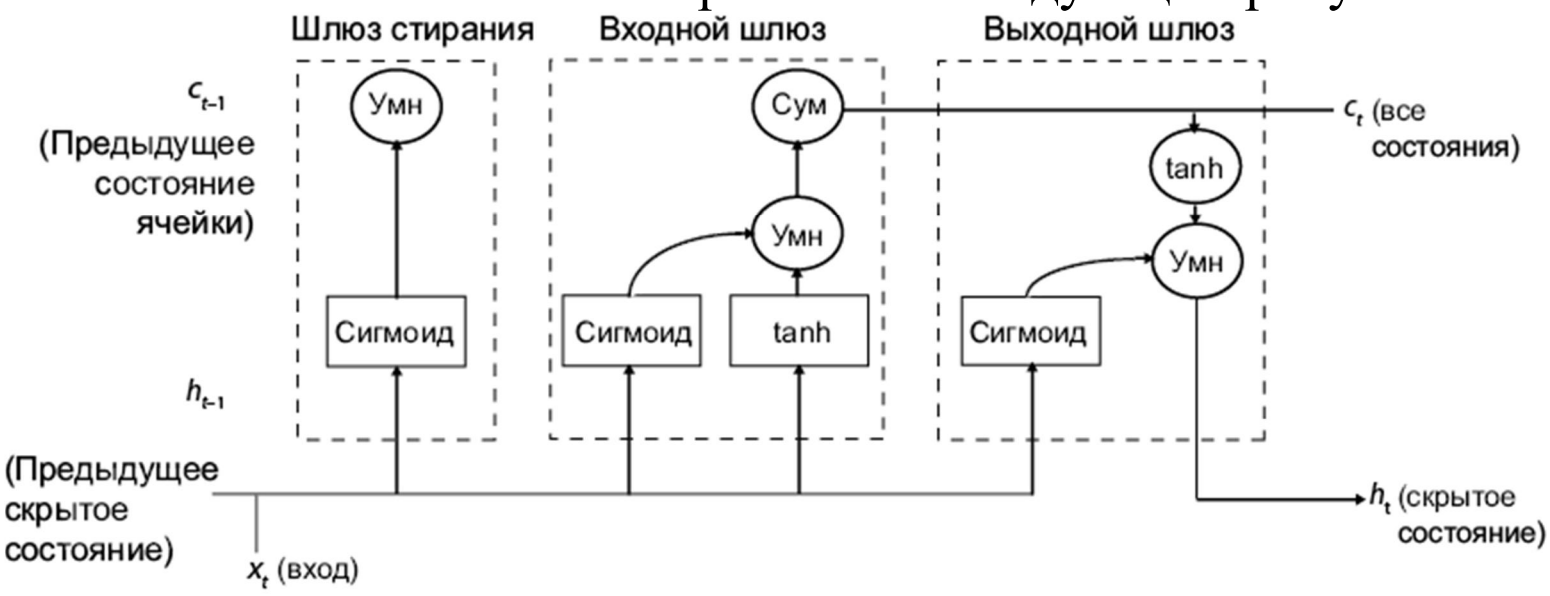
Сети RNN — интересная штука, верно? Но вы уже увидели одну проблему с обучением RNN, которая называлась проблемой исчезающего градиента. Рассмотрим ее чуть подробнее. Небо ___. RNN может легко спрогнозировать недостающее слово «голубое» на

основании виденной информации. Но при этом RNN не может отрабатывать долгосрочные зависимости. Что это значит? Допустим, «Арчи прожил в Китае 20 лет. Он любит хорошую музыку. Он обожает комиксы. Он свободно владеет ___». Скорее всего, вы предложите заполнить пропуск словом «китайским». Как вы это предсказали? Поняли, что, если Арчи прожил в Китае 20 лет, то, скорее всего, бегло говорит на китайском. Однако RNN не сможет сделать такой вывод из-за неспособности удерживать информацию долго. Что с этим делать?

На помощь приходит долгая краткосрочная память, или LSTM.

LSTM (long short-term memory) — разновидность RNN, которая решает проблему исчезающего градиента. Она хранит информацию в памяти настолько долго, насколько потребуется. Фактически ячейки RNN заменяются LSTM.

Типичная ячейка LSTM изображена на следующем рисунке:



Ячейки LSTM называются *памятью* и отвечают за хранение информации. Но как долго эта информация должна оставаться в памяти? Когда можно удалить старую информацию и обновить ячейку новой? Все эти решения должны приниматься тремя специальными шлюзами:

- шлюз стирания;
- входной шлюз;
- выходной шлюз.

Взгляните на изображение ячейки LSTM: верхняя горизонтальная линия C_t называется *состоянием ячейки*. Здесь хранится информация о состоянии ячейки, постоянно обновляемая шлюзами LSTM.

А теперь посмотрим, как функционируют эти шлюзы.

• *Шлюз стирания* отвечает за принятие решения о том, какая информация не должна находиться в состоянии ячейки. Рассмотрим следующее утверждение:

«Гарри хорошо поет. Он живет в Нью-Йорке. Зейн тоже хорошо поет».

Как только речь заходит о Зейне, сеть понимает, что обсуждение переключилось с Гарри на Зейна и информация о Гарри больше не нужна. Шлюз стирания удаляет информацию о Гарри из состояния ячейки.

• *Входной шлюз* отвечает за принятие решений о том, какая информация должна храниться в памяти. Рассмотрим тот же пример:

«Гарри хорошо поет. Он живет в Нью-Йорке. Зейн тоже хорошо поет».

После того как шлюз стирания удаляет информацию из состояния ячейки, входной шлюз решает, какая информация должна храниться в памяти. Поскольку информация о Гарри удаляется из состояния ячейки шлюзом стирания, входной шлюз решает обновить состояние ячейки информацией о Зейне.

• *Выходной шлюз* отвечает за принятие решений о том, какая информация должна выдаваться из состояния ячейки в момент времени *t*. Рассмотрим следующий фрагмент:

«Дебютный альбом Зейна пользовался большим успехом. Браво ____».

Выходной уровень должен спрогнозировать для заполнения пропуска имя «Зейн».

Генерирование текстов песен посредством LSTM RNN

А теперь посмотрим, как использовать LSTM для генерирования текстов песен Зейна Малика. Набор данных с текстами песен Зейна можно загрузить по адресу **https://github.com/sudharsan13296/Hands-On-Reinforcement-Learning-With-Python/blob/master/07.%20Deep%20Learning%20Fundamentals/data/ZaynLyrics.txt.**

Работа начинается с импортирования необходимых библиотек:

```
import tensorflow as tf
import numpy as np
```

Затем читается файл с текстами песен:

```
with open("Zayn_Lyrics.txt","r") as f:
    data=f.read()
    data=data.replace('\n','')
    data = data.lower()
```

Убедимся в том, что данные были успешно загружены:

```
data[:50]
"now i'm on the edge can't find my way it's
inside "
```

Теперь все символы сохраняются в переменной `all_chars`:

```
all_chars=list(set(data))
```

Количество уникальных символов сохраняется в `unique_chars`:

```
unique_chars = len(all_chars)
```

А общее количество символов сохраняется в переменной `total_chars`:

```
total_chars =len(data)
```

Сначала мы присвоим каждому символу индекс. `char_to_ix` будет содержать отображение символа на индекс, а `ix_to_char` — отображение индекса на символ:

```
char_to_ix = { ch:i for i,ch in
enumerate(all_chars) }
```

```
ix_to_char = { i:ch for i,ch in
enumerate(all_chars) }
```

Пример:

```
char_to_ix['e']
9

ix_to_char[9]
e
```

Затем определяется функция `generate_batch`, которая генерирует входные и целевые значения. Целевые значения равны сдвигу входного значения, умноженному на `i`.

Например, если `input` = `[12,13,24]` со значением сдвига `1`, то целевые значения будут равны `[13,24]`:

```
def generate_batch(seq_length,i):
    inputs = [char_to_ix[ch] for ch in
data[i:i+seq_length]]
    targets = [char_to_ix[ch] for ch in
data[i+1:i+seq_length+1]]
    inputs=np.array(inputs).reshape(seq_length,1
)
    targets=np.array(targets).reshape(seq_length
,1)
    return inputs,targets
```

Мы определим длину последовательности, скорость обучения и количество узлов, которое равно числу нейронов:

```
seq_length = 25
learning_rate = 0.1
num_nodes = 300
```

Построим LSTM RNN. TensorFlow предоставляет функцию `BasicLSTMCell()` для построения ячеек LSTM; вы должны задать количество единиц в ячейке LSTM и тип используемой функции активации.

Итак, мы создаем ячейку LSTM и строим сеть RNN с этой ячейкой при помощи функции `tf.nn.dynamic_rnn()`, которая возвращает выход и значение состояния:

```
def build_rnn(x):
        cell=
tf.contrib.rnn.BasicLSTMCell(num_units=num_nodes,
activation=tf.nn.relu)
        outputs, states =
tf.nn.dynamic_rnn(cell, x, dtype=tf.float32)
        return outputs,states
```

Теперь создадим заместителя для входа `X` и цели `Y`:

```
X=tf.placeholder(tf.float32,[None,1])
```

```
Y=tf.placeholder(tf.float32,[None,1])
```

Преобразуем `X` и `Y` в `int`:

```
X=tf.cast(X,tf.int32)
Y=tf.cast(Y,tf.int32)
```

Также создадим `onehot`-представления для `X` и `Y`:

```
X_onehot=tf.one_hot(X,unique_chars)
Y_onehot=tf.one_hot(Y,unique_chars)
```

Получим выходы и состояния от RNN вызовом функции `build_rnn`:

```
outputs,states=build_rnn(X_onehot)
```

Транспонируем выход:

```
outputs=tf.transpose(outputs,perm=[1,0,2])
```

Инициализируем веса и смещение:

```
W=tf.Variable(tf.random_normal((num_nodes,unique
_chars),stddev=0.001))
B=tf.Variable(tf.zeros((1,unique_chars)))
```

Вычислим выход, умножая выход на веса и прибавляя смещение:

```
Ys=tf.matmul(outputs[0],W)+B
```

Теперь выполним softmax-активацию и получим вероятности:

```
prediction = tf.nn.softmax(Ys)
```

Потеря `cross_entropy` будет вычислена в следующем виде:

```
cross_entropy=tf.reduce_mean(tf.nn.softmax_cross
_entropy_with_logits(labels
=Y_onehot,logits=Ys))
```

Наша цель — минимизация потери, поэтому мы выполним обратное распространение для сети и проведем градиентный спуск:

```
optimiser =
tf.train.GradientDescentOptimizer(learning_rate=
learning_rate).minimize(cro
ss_entropy)
```

Затем будет определена вспомогательная функция `predict`, которая даст индексы следующего прогнозируемого символа в соответствии с моделью RNN:

```
def predict(seed,i):
    x=np.zeros((1,1))
    x[0][0]= seed
    indices=[]
    for t in range(i):
        p=sess.run(prediction,{X:x})
        index =
np.random.choice(range(unique_chars), p=p.ravel())
        x[0][0]=index
        indices.append(index)
    return indices
```

Затем будет задан размер пакета `batch_size`, количество пакетов и количество эпох, а также величина сдвига `shift` для генерирования пакета:

```
batch_size=100
total_batch=int(total_chars//batch_size)
epochs=1000
shift=0
```

Наконец, мы создаем сеанс TensorFlow и строим модель:

```
init=tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)
    for epoch in range(epoch):
        print("Epoch {}:".format(epoch))
        if shift + batch_size+1 >= len(data):
            shift =0
        # Получить ввод и цель для каждого
пакета функцией
        #  generate_batch, которая сдвигает ввод
на величину shift,
        # и сформировать цель
        for i in range(total_batch):
            inputs,targets=generate_batch(batch_
size,shift)
            shift += batch_size
            # calculate loss
            if(i%100==0):
                loss=sess.run(cross_entropy,feed
_dict={X:inputs,
  Y:targets})
                # Получаем индекс следующего
спрогнозированного символа
                # при помощи функции predict
                index =predict(inputs[0],200)
                # передаем индекс в словарь
ix_to_char
                # и получаем символ
                txt = ''.join(ix_to_char[ix] for
ix in index)
                print('Iteration %i: '%(i))
                print ('\n %s \n' % (txt, ))
            sess.run(optimiser,feed_dict={X:inpu
ts,Y:targets})
```

Как видно из результатов, в исходной эпохе выход состоит из случайных символов, но по мере обучения результаты улучшаются:

```
Epoch 0:
Iteration 0:

wsadrpud,kpswkypeqawnlfyweudkgt,khdi nmgof' u
vnvlmbis .
snsblp,podwjqehb,e;g-
'fyqjsyeg,byjgyotsrdf;;u,h.a;ik'sfc;dvtauofd.,q.
;npsw'wjy-quw'quspfqw-
.
.
.
Epoch 113:
Iteration 0:
i wanna see you, yes, and she said yes!
```

Сверточные нейросети

Сверточные нейросети (CNN) составляют особую разновидность нейросетей, широко используемую в компьютерном распознавании образов. Возможности применения CNN простираются от получения информации в автомобилях с автономным управлением до автоматического отмечания друзей на фотографиях в Facebook. CNN используют пространственную информацию. Как они распознают изображения? Разберемся шаг за шагом.

CNN обычно состоит из трех слоев:

- сверточный слой;
- слой подвыборки;
- полносвязный слой.

Сверточный слой

Изображение, поданное на вход, в действительности преобразуется в матрицу значений пикселов. Эти значения лежат в диапазоне от 0 до 255, а размеры матрицы равны *[высота изображения × ширина изображения × количество каналов]*. Если входное изображение имеет размеры 64 × 64, то размеры матрицы пикселов составят 64 × 64 × 3 , где 3 — количество каналов. Изображение в оттенках серого имеет 1 канал, а цветные изображения — 3 канала (RGB). Взгляните на следующую фотографию; при подаче на вход она будет преобразована в матрицу значений пикселов. Чтобы лучше понять происходящее, ограничимся изображениями в оттенках серого — при одном канале матрица получится двумерной.

Входное изображение:

А это матрица значений:

13	8	18	63	7
5	3	1	2	33
1	9	0	7	16
3	16	5	8	18
5	7	81	36	9

Итак, изображение представлено матрицей. Что дальше? Как сети идентифицировать изображение по значениям пикселов? Для этого необходимо ввести операцию *свертки*(convolution). Эта операция извлекает важные признаки изображения для понимания, что на нем представлено. Допустим, на фотографии — собака; как вы думаете, какие признаки изображения помогут вам это понять? Наверное, строение тела, морда, ноги, хвост и т.д. Операции свертки помогут сети узнать признаки, присущие собакам. А теперь разберемся, как именно.

Каждое изображение характеризуется матрицей. Допустим, у вас имеется матрица пикселов изображения собаки; назовем ее *входной матрицей*. Также рассмотрим другую матрицу $n \times n$, которая называется *фильтром*:

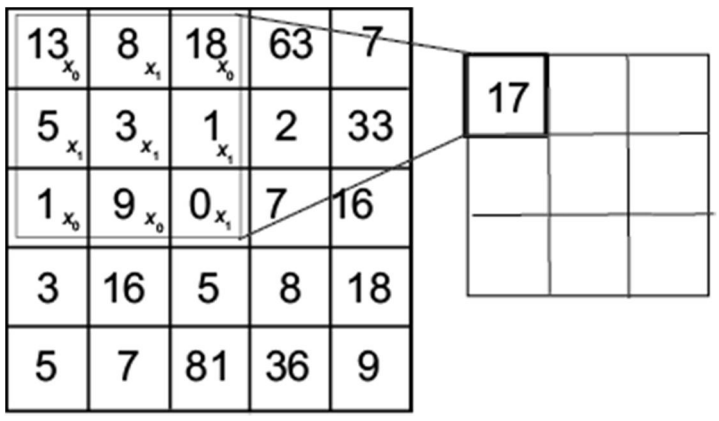
13	8	18	63	7
5	3	1	2	33
1	9	0	7	16
3	16	5	8	18
5	7	81	36	9

Входная матрица

0	1	0
1	1	0
0	0	1

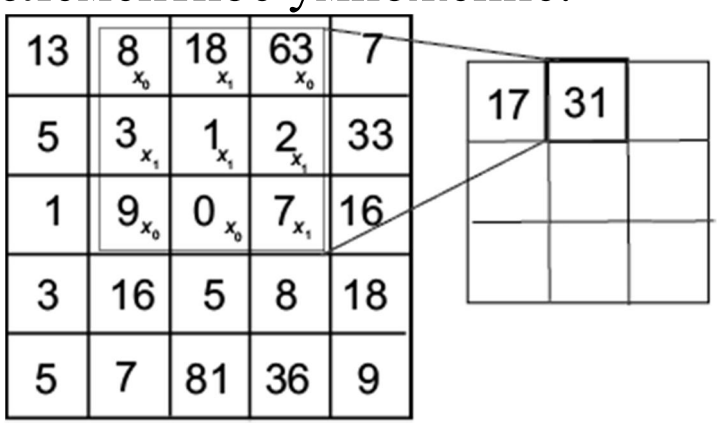
Фильтр

Фильтр перемещается по входной матрице на 1 пиксел и выполняет поэлементное умножение, результатом которого является одно число. Непонятно? Взгляните на следующий рисунок:



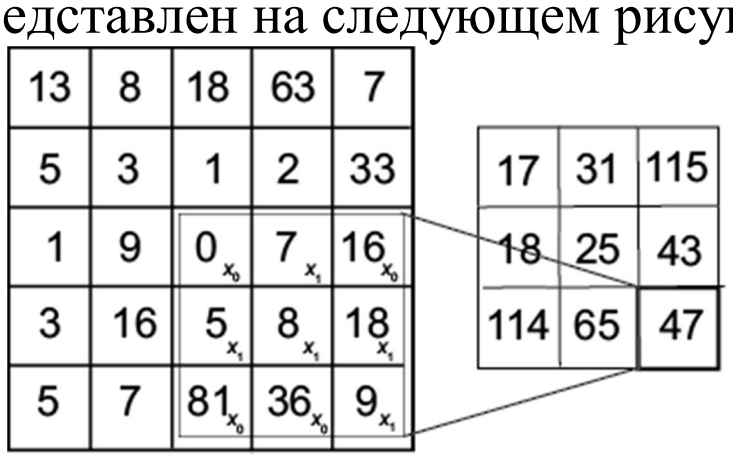
(13 × 0) + (8 × 1) + (18 × 0) + (5 × 1) + (3 × 1) + (1 × 1) + (1 × 0) + (9 × 0) + (0 × 1) = 17.

Сместим фильтр по входной матрице на 1 пиксел и выполним поэлементное умножение:



(8 × 0) + (18 × 1) + (63 × 0) + (3 × 1) + (1 × 1) + (2 × 1) + (9 × 0) + (0 × 0) + (7 × 1) = 31.

Матрица-фильтр скользит по всей входной матрице с выполнением поэлементного умножения; в результате будет получена новая матрица, называемая *картой признаков*, или *картой активации*. Процесс свертки представлен на следующем рисунке:



Исходное изображение и результат свертки:

Реальное изображение Свернутое изображение

Как видите, фильтр обнаружил в изображении контур и построил свернутое изображение. Разные фильтры используются для извлечения разных признаков изображения.

Например, если воспользоваться фильтром повышения резкости $\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$, свернутое изображение будет выглядеть так:

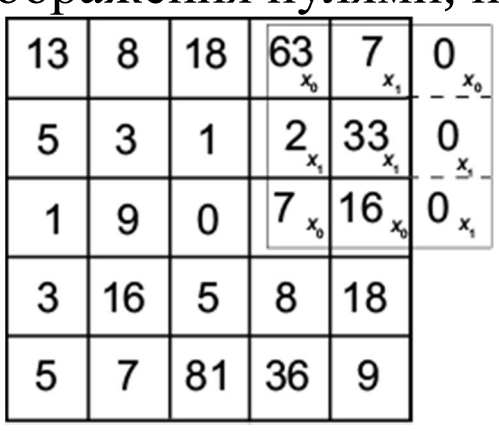
Таким образом, фильтры отвечают за извлечение признаков реального изображения посредством операции свертки. К изображению будут применяться несколько фильтров для извлечения разных признаков, создающих карту. Глубина карты признаков равна количеству используемых фильтров. Если вы используете 5 фильтров и строите 5 карт признаков, то глубина карты будет равна 5:

Карта признаков глубиной 5

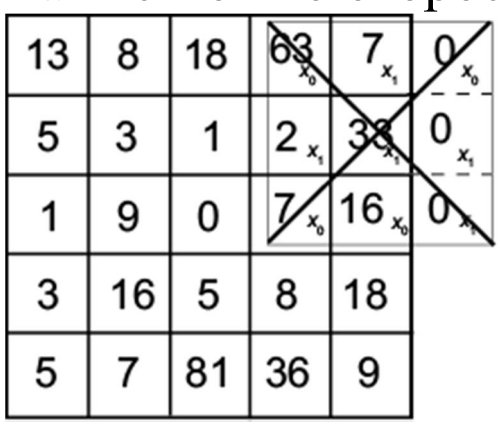
При большом количестве фильтров наша сеть лучше поймет изображение за счет извлечения большого количества признаков. В ходе построения CNN указывать значения для матрицы фильтров не нужно — оптимальные значения будут определены во время процесса обучения. Тем не менее вы должны задать количество и размеры фильтров.

Ранее говорилось о смещении фильтра над входной матрицей на 1 пиксел и выполнении операции свертки. Смещение не обязательно ограничивать одним пикселом; входная матрица может сместиться на любое количество пикселов, которое называется *шагом* (stride).

Но что произойдет, когда скользящее окно (матрица-фильтр) достигнет края изображения? В этом случае входная матрица дополнится нулями, чтобы фильтр можно было применить у края. Дополнение изображения нулями, называемое *расширением свертки*, происходит так:



Также можно отбросить эту область, сделав *сужение свертки*:



После выполнения свертки применяется функция активации ReLU для введения нелинейности.

Слой подвыборки

За сверточным слоем следует *слой подвыборки* (pooling). Он сохраняет минимум необходимых подробностей для сокращения размеров карт признаков и объема вычислений. Например, чтобы определить, присутствует ли на изображении собака, нам не обязательно понимать, где она расположена, — нужны только признаки собаки. Таким образом, уровень подвыборки сокращает пространственные размеры карт, оставляя только важные признаки. Существуют несколько типов операций подвыборки в границах окна. *Максимальная подвыборка* — одна из самых распространенных — основана на получении максимального значения из карты признаков.

Максимальная подвыборка с фильтром 2 × 2 и шагом 2 выполняется следующим образом:

При *усредняющей подвыборке*вычисляется среднее значение элементов на карте признаков, а при *суммирующей подвыборке* — сумма этих элементов. Операция подвыборки влияет на высоту и ширину, но не изменяет глубину карты.

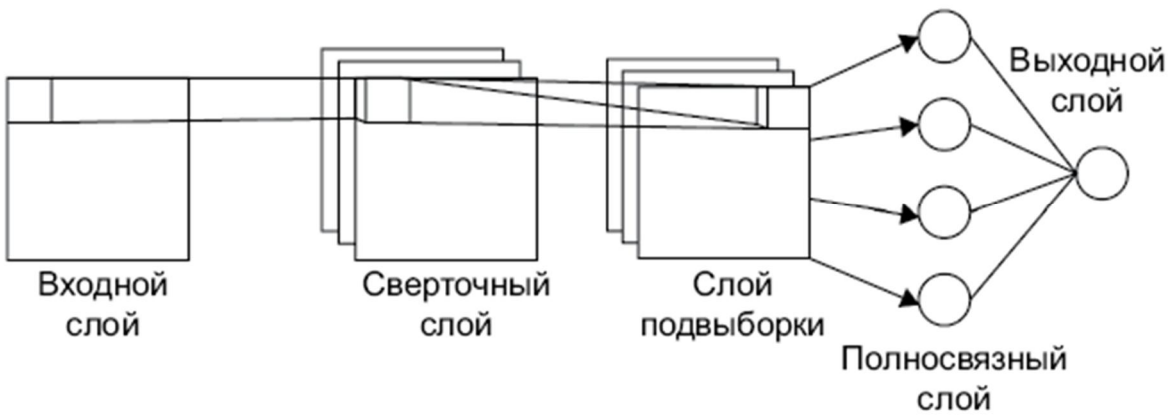
Полносвязный слой

Сеть может содержать несколько сверточных слоев, за которыми следуют слои подвыборки. Все эти слои только извлекают признаки из входного изображения и строят карты активации. Но как классифицировать на изображении собаку, основываясь только на одних картах активации? Потребуется ввести новый слой, называемый *полносвязным*. Он получает на входе карты активации (которые к этому моменту фактически представляют признаки изображения), применяет функцию активации и генерирует выход. Полносвязный слой в действительности представляет собой обычную

нейросеть с входным, скрытым и выходным слоями. Здесь вместо входного слоя используется сверточный слой со слоем подвыборки, которые совместно генерируют карты активации как входные данные.

Архитектура CNN

А теперь посмотрим, как все эти слои формируют архитектуру CNN:



Сначала изображение передается сверточному слою, на котором операция свертки применяется для извлечения признаков. Затем карты признаков передаются слою подвыборки, где происходит сокращение размеров. (Мы можем добавить любое количество сверточных уровней и уровней подвыборки в зависимости от конкретной ситуации.) После этого в конце добавляется нейросеть с одним скрытым уровнем (полносвязный уровень), который обеспечивает классификацию изображения.

Классификация предметов одежды с использованием CNN

А теперь посмотрим, как CNN может использоваться для классификации предметов одежды.

Как обычно, первым шагом становится импортирование необходимых библиотек:

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Перейдем к чтению данных. Набор данных доступен в `tensorflow.examples`, поэтому данные можно прочитать напрямую:

```
from tensorflow.examples.tutorials.mnist import input_data
fashion_mnist = input_data.read_data_sets('data/fashion/', one_hot=True)
```

Проверим основные характеристики загруженных данных:

```
print("No of images in training set {}".format(fashion_mnist.train.images.shape))
print("No of labels in training set {}".format(fashion_mnist.train.labels.shape))
```

```
print("No of images in test set
{}".format(fashion_mnist.test.images.shape))
print("No of labels in test set
{}".format(fashion_mnist.test.labels.shape))
```

```
No of images in training set (55000, 784)
No of labels in training set (55000, 10)
No of images in test set (10000, 784)
No of labels in test set (10000, 10)
```

Итак, обучающий набор состоит из 55 000 точек данных, а тестовый набор — из 10 000 точек. Для данных определены 10 меток, что означает 10 категорий предметов одежды:

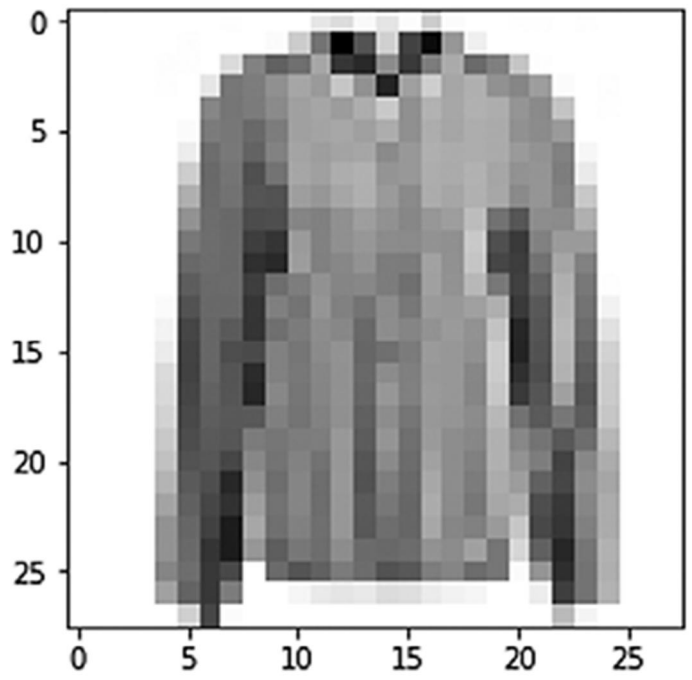
```
labels = {
0: 'Майка',
1: 'Брюки',
2: 'Свитер',
3: 'Платье',
4: 'Пальто',
5: 'Сандалии',
6: 'Рубашка',
7: 'Тапки',
8: 'Сумка',
9: 'Ботильоны'
}
```

Рассмотрим некоторые изображения:

```
img1 =
fashion_mnist.train.images[41].reshape(28,28)
# получить соответствующую целочисленную метку
из данных
# с onehot-кодированием
label1 = np.where(fashion_mnist.train.labels[41]
== 1)[0][0]
# Вывести точку данных
print("y = {} ({})".format(label1,
labels[label1]))
plt.imshow(img1, cmap='Greys')
```

Пример вывода и графического изображения:

```
y = 6 (Рубашка)
```

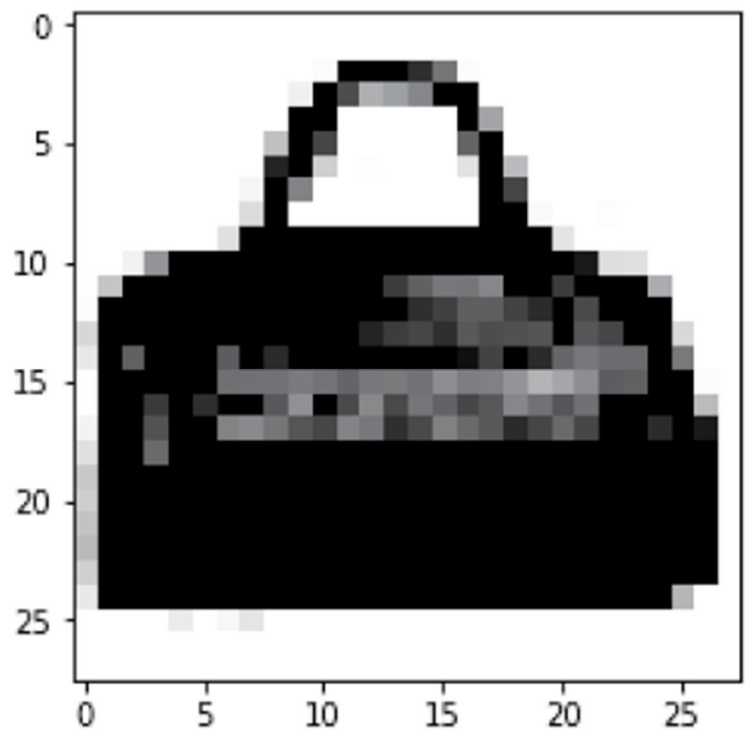


Действительно, рубашка. Другой пример:

```
img1 =
fashion_mnist.train.images[19].reshape(28,28)
  # получить соответствующую целочисленную метку
из данных
  # с onehot-кодированием
  label1 = np.where(fashion_mnist.train.labels[19]
== 1)[0][0]
  # Вывести точку данных
  print("y = {} ({})".format(label1,
labels[label1]))
  plt.imshow(img1, cmap='Greys')
```

Вывод и графическое изображение:

```
y = 8 (Сумка)
```



Это сумка, не поспоришь!

Итак, мы должны построить сверточную нейросеть, которая классифицирует эти изображения по соответствующим категориям. Мы определяем заместителей для входных изображений и выходных меток. Так как размер входного изображения равен `784`, заместитель для входа `X` определяется так:

```
x = tf.placeholder(tf.float32, [None, 784])
```

Выход необходимо подогнать под формат `[p,q,r,s]`, где `q` и `r` — фактические размеры выходного изображения (28 × 28), а `s` — число каналов. Так как мы работаем только с изображениями в оттенках серого, значение `s` равно 1. Из `p` выводится количество обучающих точек данных, то есть размер пакета. Так как размер пакета неизвестен, мы можем присвоить ему значение `-1` — значение будет динамически изменяться в процессе обучения:

```
x_shaped = tf.reshape(x, [-1, 28, 28, 1])
```

Так как в данных определены 10 разных меток, заместители для выхода определяются следующим образом:

```
y = tf.placeholder(tf.float32, [None, 10])
```

Теперь необходимо определить функцию `conv2d`, которая непосредственно выполняет операцию свертки, то есть поэлементное умножение входной матрицы (`x`) на фильтр (`w`) с шагом 1 и заполнением нулями.

Мы задаем шаги `strides` = `[1, 1,1, 1]`. Первое и последнее значения `strides` равны `1`, потому что мы не хотим переключаться между обучающими точками данных и разными каналами. Второе и третье значения `strides` тоже равны 1 — это означает, что фильтр смещается на 1 пиксел как по вертикали, так и по горизонтали:

```
def conv2d(x, w):
    return tf.nn.conv2d(x, w, strides=[1, 1, 1, 1], padding='SAME')
```

Далее определяем функцию `maxpool2d` для выполнения подвыборки. Это будет максимальная подвыборка с шагом 2 и дополнением нулями (`SAME`). `ksize` определяет размеры окна подвыборки:

```
def maxpool2d(x):
  return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1],
padding='SAME')
```

Затем нужно определить веса и смещения. Мы построим сверточную сеть с двумя сверточными слоями, за которыми следуют полносвязный и выходной слои, поэтому мы определим веса для всех этих уровней. Веса выступают как фильтры сверточных слоев.

Итак, матрица весов инициализируется `[filter_shape[0],filter_shape[1],number_of_input_channel,filter_size]`.

Мы используем фильтр 5 × 5 и присваиваем ему размер 32. Так как мы используем изображения в оттенках серого, количество входных каналов равно 1, и матрица весов будет иметь вид `[5,5,1,32]`:

```
w_c1 = tf.Variable(tf.random_normal([5,5,1,32]))
```

Так как второй сверточный слой получает вход от первого сверточного слоя, у которого количество каналов на выходе равно `32`, количество входных каналов следующего слоя тоже будет равно `32`:

```
w_c2 =
tf.Variable(tf.random_normal([5,5,32,64]))
```

Затем мы инициализируем смещения:

```
b_c1 = tf.Variable(tf.random_normal([32]))
b_c2 = tf.Variable(tf.random_normal([64]))
```

Далее мы выполняем операции на первом сверточном слое, то есть операцию свертки с входом `x` и активациями ReLU, за которыми следует максимальная подвыборка:

```
conv1 = tf.nn.relu(conv2d(x, w_c1) + b_c1)
conv1 = maxpool2d(conv1)
```

Теперь результат первого сверточного слоя будет передан второму, на котором операция свертки будет выполнена с результатом первого сверточного слоя с активациями ReLU, за которыми последует максимальная подвыборка:

```
conv2 = tf.nn.relu(conv2d(conv1, w_c2) + b_c2)
conv2 = maxpool2d(conv2)
```

После двух сверточных слоев с операциями свертки и подвыборки входное изображение уменьшается с 28 × 28 × 1 до 7 × 7 × 1. Необходимо деструктурировать этот выход, перед тем как подавать его на полносвязный слой. А затем результат второго сверточного слоя будет подан на полносвязный слой, умножен на веса, к результату будет прибавлено смещение и применены активации ReLU:

```
x_flattened = tf.reshape(conv2, [-1, 7*7*64])
w_fc =
tf.Variable(tf.random_normal([7*7*64,1024]))
b_fc = tf.Variable(tf.random_normal([1024]))
fc = tf.nn.relu(tf.matmul(x_flattened,w_fc)+
b_fc)
```

Теперь необходимо определить веса и смещение для выходного уровня:

```
w_out = tf.Variable(tf.random_normal([1024,
10]))
b_out = tf.Variable(tf.random_normal([10]))
```

Чтобы получить выход, мы умножаем результат полносвязного слоя на матрицу весов и прибавляем смещение. Для получения вероятностей выхода используется softmax-функция активации:

```
output = tf.matmul(fc, w_out)+ b_out
yhat = tf.nn.softmax(output)
```

Функция потерь может быть определена в форме перекрестной энтропии. Для оптимизации функции потерь вместо оптимизатора градиентного спуска используется новая разновидность оптимизатора — так называемый *оптимизатор*

Adam(**https://www.tensorflow.org/api_docs/python/tf/train/AdamOptimizer**):

```
  cross_entropy =
  tf.reduce_mean(tf.nn.softmax_cross_entropy_with_
logits(logits=output,
  labels=y))optimiser =
  tf.train.AdamOptimizer(learning_rate=learning_ra
te).minimize(cross_entropy)
```

Теперь вычисляется точность `accuracy`:

```
  correct_prediction = tf.equal(tf.argmax(y, 1),
tf.argmax(yhat, 1))
  accuracy =
tf.reduce_mean(tf.cast(correct_prediction,
tf.float32))
```

И определяются гиперпараметры:

```
epochs = 10
batch_size = 100
```

Создадим сеанс TensorFlow и построим модель:

```
init_op = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init_op)
    total_batch =
int(len(fashion_mnist.train.labels) / batch_size)
     # Для каждой эпохи
    for epoch in range(epochs):
         avg_cost = 0
         for i in range(total_batch):
             batch_x, batch_y =
  fashion_mnist.train.next_batch(batch_size=batch_
size)
             _, c = sess.run([optimiser,
cross_entropy],
                             feed_dict={x: batch_x,
y: batch_y})
             avg_cost += c / total_batch
         print("Epoch:", (epoch + 1), "cost
=""{:.3f}".format(avg_cost))
    print(sess.run(accuracy, feed_dict={x:
mnist.test.images, y:
  mnist.test.labels}))
```

Итоги

В этой главе вы познакомились с работой нейросети на примере ее построения для классификации рукописных цифр с использованием TensorFlow. Также были показаны различные типы нейросетей, способные хранить информацию в памяти, такие как RNN. Были продемонстрированы сети LSTM, предназначенные для решения проблемы исчезающего градиента за счет использования нескольких шлюзов для хранения информации в памяти настолько долго, насколько потребуется. Также была продемонстрирована нейросеть для распознавания образов — CNN. Вы узнали, как в CNN применяются разные слои и как построить сеть CNN для распознавания предметов одежды с использованием TensorFlow.

В **главе 8** вы увидите, как нейросети помогают повысить эффективность обучения агентов RL.

Вопросы

1. Чем линейная регрессия отличается от нейросети?
2. Для чего нужна функция активации?
3. Зачем вычислять градиент при градиентном спуске?
4. Какими преимуществами обладает RNN?
5. В чем суть проблем исчезающего и взрывного градиента?
6. Какие шлюзы используются в LSTM?
7. Для чего нужен слой подвыборки?

Дополнительные источники

CNN в стэнфордском курсе: `https://www.youtube.com/watch?v=NfnWJUyUJYUlist=PLkt2uSq6rBVctENoVBg1TpCC7OQi31AlC.`

Эта публикация в блоге поможет вам сделать первые шаги в мире RNN: `http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/.`

8. Игры Atari с использованием Deep Q Network

Deep Q Network (*DQN*) — один из популярных алгоритмов **глубокого обучения с подкреплением** (**DRL**). Этот алгоритм произвел фурор после своего выхода. Он был предложен исследователями из компании DeepMind, принадлежащей Google, и достигал результатов человеческого уровня в любой игре Atari, получая на входе только экраны игры.

В этой главе вы узнаете, как работает DQN и как построить для любой игры Atari сеть DQN. Также мы рассмотрим некоторые усовершенствования, внесенные в архитектуру DQN, например двойные сети DQN и дуэльную сетевую архитектуру.

В этой главе разобраны следующие темы:

• Deep Q Network (DQN).

- Архитектура DQN.
- Построение агента для игр Atari.
- Двойные сети DQN.
- Приоритетное воспроизведение опыта.

Что такое DQN?

Прежде чем двигаться дальше, вспомним, что такое *Q*-функция. Эта функция, также называемая функцией ценности состояния/действия, указывает, насколько эффективным является применение действия `a` в состоянии `s`. Значения всех возможных действий для каждого состояния хранятся в таблице, называемой *Q*-таблицей, и мы выбираем в качестве оптимального то действие, которое обладает максимальной ценностью в состоянии.

Для изучения *Q*-функции применялось *Q*-обучение — алгоритм обучения на основе временных различий без привязки к политике (см. **главу 5**).

До сих пор мы видели среды с конечным числом состояний и ограниченными наборами действий, в которых был возможен исчерпывающий поиск оптимального значения *Q* по всем возможным парам состояние/действие. Но представьте среду, в которой число состояний очень велико, а в каждом состоянии доступно множество действий. Перебор всех действий в каждом состоянии занял бы слишком много времени. Другой, более эффективный подход основан на приближении *Q*-функции с некоторым параметром θ в форме $Q(s, a; \theta) \approx Q^*(s, a)$. Мы используем нейросеть с весами, чтобы аппроксимировать значение *Q* для всех возможных действий в каждом состоянии, ее можно назвать *Q*-сетью. Хорошо, но как обучить сеть и как будет выглядеть целевая функция? Вспомните правило обновления для *Q*-обучения:

$$Q(s, a) = Q(s, a) + \alpha(r + \gamma \max Q(s', a') - Q(s, a)).$$

$r + \gamma \max Q(s', a')$ — целевое значение, а $Q(s, a)$ — прогнозируемое значение; мы стараемся свести это значение к минимуму за счет использования правильной политики.

Аналогичным образом в DQN можно определить функцию потерь как квадрат разности между целевым и прогнозируемым значениями; мы также стараемся минимизировать потери за счет обновления весов θ:

$$потери = (y_i - Q(s, a; \theta))^2,$$

где $y_i = r + \gamma \max_{a'} Q(s', a'; \theta)$.

Для обновления весов и минимизации потерь используется градиентный спуск. В двух словах, в DQN нейросети используются как функции-аппроксиматоры для *Q*-функции, а для минимизации ошибок используется градиентный спуск.

Архитектура DQN

Теперь, когда вы в общих чертах представляете, как устроена сеть DQN, мы более подробно рассмотрим ее работу и архитектуру для игр Atari.

Сначала будут представлены отдельные компоненты, а затем алгоритм в целом.

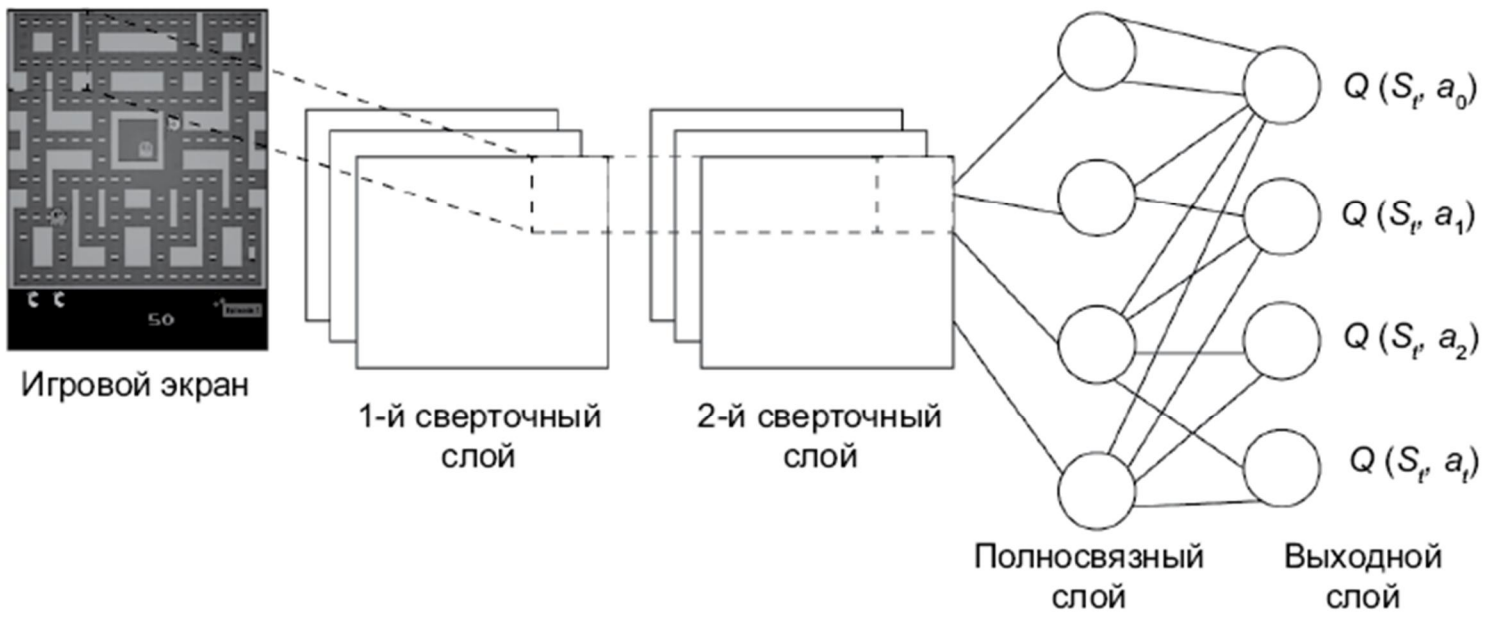
Сверточная сеть

Первый слой DQN занимает сверточная сеть, входом которой служит необработанный кадр игрового экрана. Мы берем кадр и передаем его сверточным слоям, для того чтобы понять состояние игры. Но необработанные кадры состоят из 210 × 160 пикселов с палитрой из 128 цветов; очевидно, что их прямая подача будет занимать слишком много памяти и вычислительных ресурсов. По этой причине мы понижаем разрешение до 84 × 84 пикселов и меняем значения RGB на оттенки серого, а потом передаем обработанный игровой экран на вход сверточным слоям. Для понимания игрового экрана сверточный слой выявляет пространственные отношения между различными объектами на изображении. Мы используем два сверточных слоя, за которыми идет полносвязный слой с функцией активации ReLU. Слой подвыборки в данном случае не используется.

Слой подвыборки полезен при обнаружении объектов или классификации, когда позиция объекта на изображении вам неинтересна и вы хотите знать лишь то, присутствует объект на изображении или нет. Но для понимания игрового экрана позиция объекта важна, поскольку она передает состояние игры. Например, в классической игре *Pong* недостаточно убедиться в присутствии шарика на экране — нужно знать его позицию, чтобы выполнить следующее перемещение. Вот почему в нашей архитектуре уровень подвыборки не используется.

Хорошо, как вычислить Q? Если передать один игровой экран и одно действие на вход DQN, вы получите значение Q. Но для этого потребуется один полный прямой проход, так как в состоянии много действий. Кроме того, в игре с одним прямым проходом для каждого действия будет много состояний, что подразумевает высокие вычислительные затраты. По этой причине мы просто передаем игровой экран на вход и используем значения Q для всех возможных действий в состоянии, для чего количество единиц на выходном уровне приводится в соответствие с количеством действий в игровом состоянии.

Архитектура DQN показана на следующем рисунке. Мы передаем ей игровой экран, а она выдает значения Q для всех действий в данном игровом состоянии:



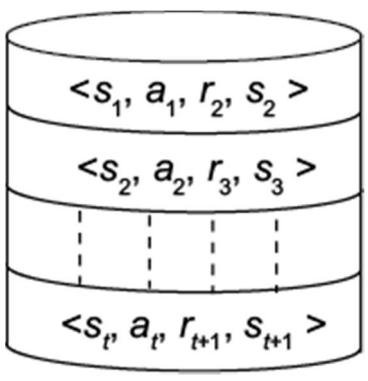
Для предсказания значений *Q* игрового состояния используется не только текущий экран, но и четыре предыдущих экрана. Почему? Вспомните игру Pac-Man: задача персонажа Pac-Man — перемещаться по экрану и поедать точки. По текущему экрану невозможно определить, в каком направлении движется Pac-Man. Но с предыдущими экранами это направление легко определяется. Итак, входные данные состоят из текущего игрового экрана и четырех предыдущих экранов.

Воспроизведение опыта

Мы знаем, что в средах RL агент переходит из состояния *s* в следующее состояние *s′*, выполняя действие *a* и получая награду *r*. Мы сохраняем эту информацию перехода в форме <*s*, *a*, *r*, *s′*> в *буфере воспроизведения опыта*. Эти переходы называются *опытом агента*.

Ключевая идея воспроизведения опыта заключается в том, что сеть DQN обучается с переходами, взятыми из буфера воспроизведения (вместо обучения на основе последних переходов). Данные из опыта агента коррелированы, а случайная выборка точек данных из буфера воспроизведения сокращает корреляцию, что помогает агенту учиться эффективнее на более широком спектре опыта.

Также эта случайная выборка помогает избавиться от явления *переобучения* (overfitting) с коррелированным опытом, характерного для нейросетей. Такую равномерную выборку можно использовать для оценки опыта. Воспроизведение опыта больше похоже на очередь, чем на список. В буфере хранится фиксированное количество последних данных, поэтому при поступлении новой информации старая удаляется:



Целевая сеть

В функции потерь вычисляется квадрат разности между целевым и прогнозируемым значением:

$$потери = (r + \gamma \max_{a'} Q(s', a'; \theta) - Q(s, a; \theta))^2.$$

Одна функция Q используется для вычисления целевого и прогнозируемого значения. Вы видите, что в приведенной формуле одни веса θ используются как для целевого, так и для прогнозируемого значения Q. Поскольку одна сеть вычисляет как прогнозируемое, так и целевое значение, между ними может существовать расхождение.

Для предотвращения этой проблемы можно использовать отдельную сеть, называемую *целевой сетью*, для вычисления целевого значения. И наша функция потерь примет следующий вид:

$$потери = (r + \gamma \max_{a'} Q(s', a'; \theta') - Q(s, a; \theta))^2.$$

Вы заметили, что целевое значение Q использует параметр θ' вместо θ. Наша Q-сеть для прогнозирования Q определяет правильные веса θ с использованием градиентного спуска. Целевая сеть «замораживается» на несколько временных шагов, после чего веса целевой сети обновляются копированием весов из сети Q. Замораживание целевой сети на некоторое время и обновление ее весов весами фактической Q-сети стабилизирует обучение.

Нормализация наград

Как назначать награды? Распределение наград изменяется в зависимости от игры. В некоторых играх можно назначать награды вида +1 за выигрыш, –1 за проигрыш и 0 за отсутствие результата, но в других случаях назначаются награды вида +100 за действие и +50 за другое действие. Для предотвращения этой проблемы все награды нормализуются до значений –1 и +1.

Понимание алгоритма

Теперь посмотрим, как работает DQN в целом.

1. Выполняется предварительная обработка и передача игрового экрана (состояние s) сети DQN, которая возвращает значения Q для всех возможных действий в состоянии.

2. Выбирается действие с использованием эпсилон-жадной стратегии: с вероятностью epsilon выбирается случайное действие a, а с вероятностью 1 – epsilon выбирается действие с максимальным значением Q, например $a = \text{argmax}(Q(s, a; \theta))$.

3. Действие a выполняется в состоянии s с переходом в новое состояние s' и получением награды. Следующее состояние s' является предварительно обработанным изображением следующего игрового экрана.

4. Переход сохраняется в буфере воспроизведения в форме $<s, a, r, s'>$.

5. Из буфера воспроизведения выбираются случайные пакеты переходов и вычисляются потери.

6. Мы знаем, что *потери* $=(r+\gamma\max_{a'}Q(s',a';\theta')-Q(s,a;\theta))^2$, то есть мы ищем квадрат разности между целевым и прогнозируемым значением Q.

7. Выполняется градиентный спуск в отношении фактических параметров сети θ для минимизации потерь.

8. Через каждые k шагов фактические сетевые веса θ копируются в веса целевой сети θ'.

9. Эти шаги повторяются для Mэпизодов.

Построение агента для игр Atari

Рассмотрим построение агента для любой игры Atari. Полный код в формате книги Jupyter с объяснениями доступен по адресу **https://github.com/sudharsan13296/Hands-On-Reinforcement-Learning-With-Python/blob/master/08.%20Atari%20Games%20with%20DQN/8.8%20Building%20an%20Agent%20to%20Play%20Atari%20Games.ipynb**.

Сначала импортируются необходимые библиотеки:

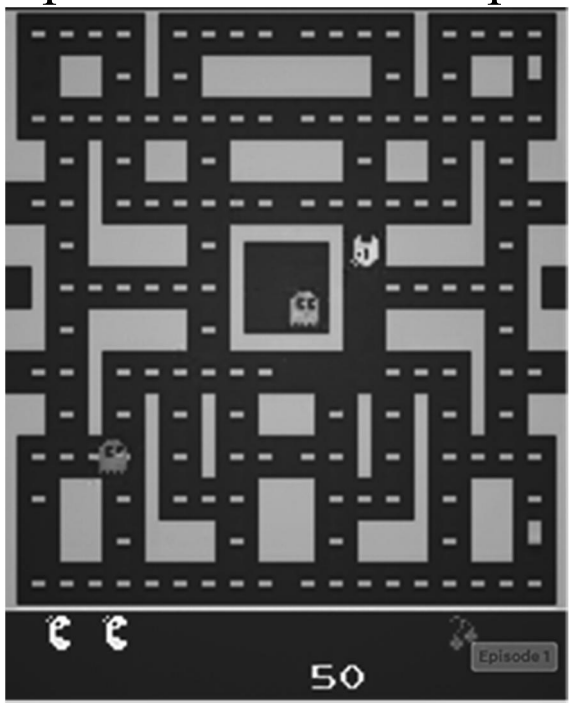
```
import numpy as np
import gym
import tensorflow as tf
from tensorflow.contrib.layers import flatten, conv2d, fully_connected
from collections import deque, Counter
import random
from datetime import datetime
```

Мы можем использовать любые игровые среды Atari, доступные по адресу: **http://gym.openai.com/envs/#atari**.

В данном примере используется игровая среда Pac-Man:

```
env = gym.make("MsPacman-v0")
n_outputs = env.action_space.n
```

Среда Pac-Man изображена на следующем рисунке:



Определим функцию `preprocess_observation` для предварительной обработки входного игрового экрана. Функция уменьшает размер изображения и преобразует палитру в оттенки серого:

```
color = np.array([210, 164, 74]).mean()

def preprocess_observation(obs):

    # Обрезать и масштабировать изображение
    img = obs[1:176:2, ::2]

    # Преобразовать изображение в оттенки серого
    img = img.mean(axis=2)

    # Улучшить контраст изображения
    img[img==color] = 0

    # Нормализовать изображение от -1 до +1
    img = (img - 128) / 128 - 1

    return img.reshape(88,80,1)
```

Определим функцию `q_network` для построения *Q*-сети. Входными данными для *Q*-сети будет игровое состояние `X`.

Мы строим *Q*-сеть с тремя сверточными слоями, дополненеными нулями, за которыми следует полносвязный слой:

```
tf.reset_default_graph()

def q_network(X, name_scope):
    # Инициализировать уровни
    initializer =
tf.contrib.layers.variance_scaling_initializer()

    with tf.variable_scope(name_scope) as scope:

        # Инициализировать сверточные уровни
        layer_1 = conv2d(X, num_outputs=32,
kernel_size=(8,8), stride=4,
  padding='SAME', weights_initializer=initializer)
        tf.summary.histogram('layer_1',layer_1)
        layer_2 = conv2d(layer_1,
num_outputs=64, kernel_size=(4,4),
  stride=2, padding='SAME',
weights_initializer=initializer)
        tf.summary.histogram('layer_2',layer_2)
        layer_3 = conv2d(layer_2,
num_outputs=64, kernel_size=(3,3),
  stride=1, padding='SAME',
weights_initializer=initializer)
```

```
            tf.summary.histogram('layer_3',layer_3)
            # Деструктурировать результат layer_3,
перед тем как
            # передавать его на полносвязный уровень
            flat = flatten(layer_3)

            fc = fully_connected(flat,
num_outputs=128,
  weights_initializer=initializer)
            tf.summary.histogram('fc',fc)
            output = fully_connected(fc,
num_outputs=n_outputs,
  activation_fn=None,
weights_initializer=initializer)
            tf.summary.histogram('output',output)

            # В vars хранятся параметры сети
(например, веса).
            vars = {v.name[len(scope.name):]: v for
v in
  tf.get_collection(key=tf.GraphKeys.TRAINABLE_VAR
IABLES, scope=scope.name)}
            return vars, output
```

Определим функцию `epsilon_greedy` для применения эпсилон-жадной стратегии, где мы выбираем либо лучшее действие с вероятностью 1 – epsilon, либо случайное действие с вероятностью epsilon.

Мы используем эпсилон-жадную стратегию с затуханием, при которой значение epsilon уменьшается со временем, чтобы исследования не продолжались бесконечно. Так, со временем наша стратегия будет эксплуатировать только хорошие действия:

```
  epsilon = 0.5
  eps_min = 0.05
  eps_max = 1.0
  eps_decay_steps = 500000
  def epsilon_greedy(action, step):
      p = np.random.random(1).squeeze()
      epsilon = max(eps_min, eps_max - (eps_max-
eps_min) *
  step/eps_decay_steps)
      if np.random.rand() < epsilon:
          return np.random.randint(n_outputs)
      else:
          return action
```

Инициализируем буфер воспроизведения опыта длины 20 000, в котором хранятся данные опыта.

Весь опыт агента (состояние, действие, награды) хранится в буфере воспроизведения опыта, и из этого мини-пакета выполняется выборка для обучения сети:

```
def sample_memories(batch_size):
    perm_batch = np.random.permutation(len(exp_buffer))[:batch_size]
    mem = np.array(exp_buffer)[perm_batch]
    return mem[:,0], mem[:,1], mem[:,2], mem[:,3], mem[:,4]
```

Определим все гиперпараметры:

```
num_episodes = 800
batch_size = 48
input_shape = (None, 88, 80, 1)
learning_rate = 0.001
X_shape = (None, 88, 80, 1)
discount_factor = 0.97

global_step = 0
copy_steps = 100
steps_train = 4
start_steps = 2000
logdir = 'logs'
```

Определим заместителя для входных данных — например, состояния игры:

```
X = tf.placeholder(tf.float32, shape=X_shape)
```

Введем логическую переменную с именем `in_training_mode` для переключения режима обучения:

```
in_training_mode = tf.placeholder(tf.bool)
```

Построим *Q*-сеть, которая получает ввод `X` и генерирует значения *Q* для всех действий в состоянии:

```
mainQ, mainQ_outputs = q_network(X, 'mainQ')
```

Аналогичным образом построим целевую *Q*-сеть:

```
targetQ, targetQ_outputs = q_network(X, 'targetQ')
```

Определим заместителя для ценности действий:

```
X_action = tf.placeholder(tf.int32, shape=(None,))
Q_action = tf.reduce_sum(targetQ_outputs * tf.one_hot(X_action, n_outputs),
axis=-1, keep_dims=True)
```

Параметры основной *Q*-сети копируются в целевую *Q*-сеть:

```
  copy_op = [tf.assign(main_name,
targetQ[var_name]) for var_name, main_name
  in mainQ.items()]
  copy_target_to_main = tf.group(*copy_op)
```

Определим заместителя для выхода (например, действия):

```
  y = tf.placeholder(tf.float32, shape=(None,1))
```

Вычислим потери — квадрат разности между фактическим и прогнозируемым значением:

```
  loss = tf.reduce_mean(tf.square(y - Q_action))
```

Для минимизации потерь введем оптимизатор `AdamOptimizer`:

```
  optimizer =
tf.train.AdamOptimizer(learning_rate)
  training_op = optimizer.minimize(loss)
```

Определим файлы журналов для визуализации в TensorBoard:

```
  loss_summary = tf.summary.scalar('LOSS', loss)
  merge_summary = tf.summary.merge_all()
  file_writer = tf.summary.FileWriter(logdir,
tf.get_default_graph())
```

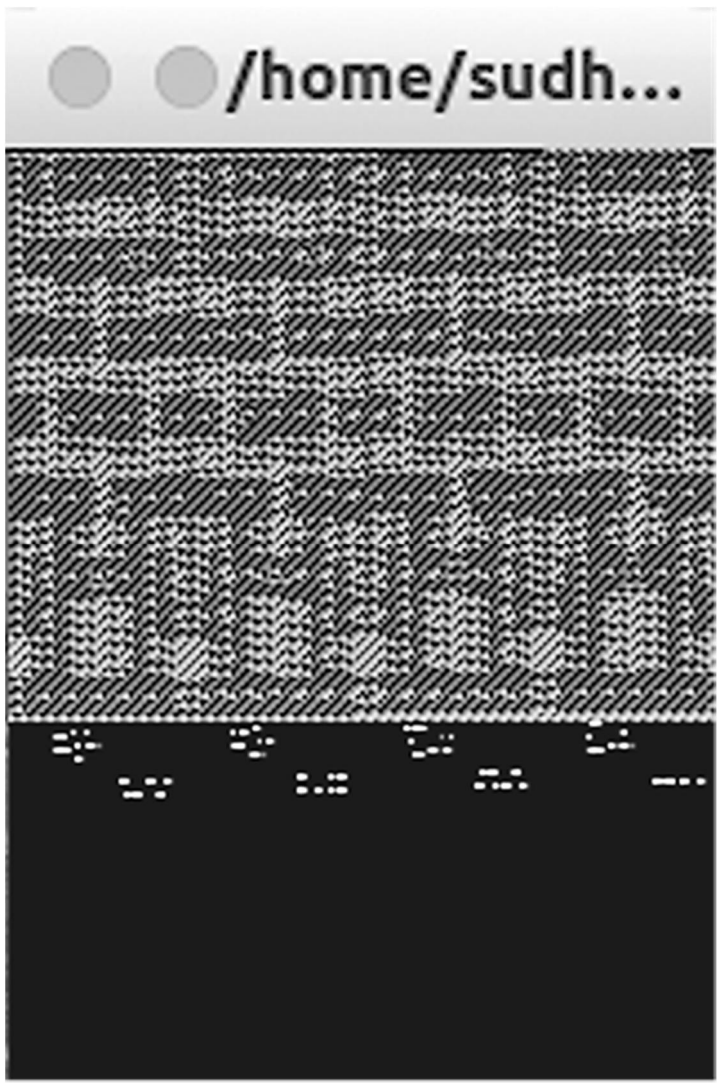
Теперь откроем сеанс TensorFlow и запустим модель:

```
  init = tf.global_variables_initializer()
  with tf.Session() as sess:
      init.run()
      # для каждого эпизода
      for i in range(num_episodes):
          done = False
          obs = env.reset()
          epoch = 0
          episodic_reward = 0
          actions_counter = Counter()
          episodic_loss = []

          # Пока состояние не является завершающим
          while not done:

              #env.render()
               # Получить предварительно
обработанный игровой экран
              obs = preprocess_observation(obs)
              # Передать игровой экран и получить
              # значения Q для каждого действия
              actions =
mainQ_outputs.eval(feed_dict={X:[obs],
  in_training_mode:False})
```

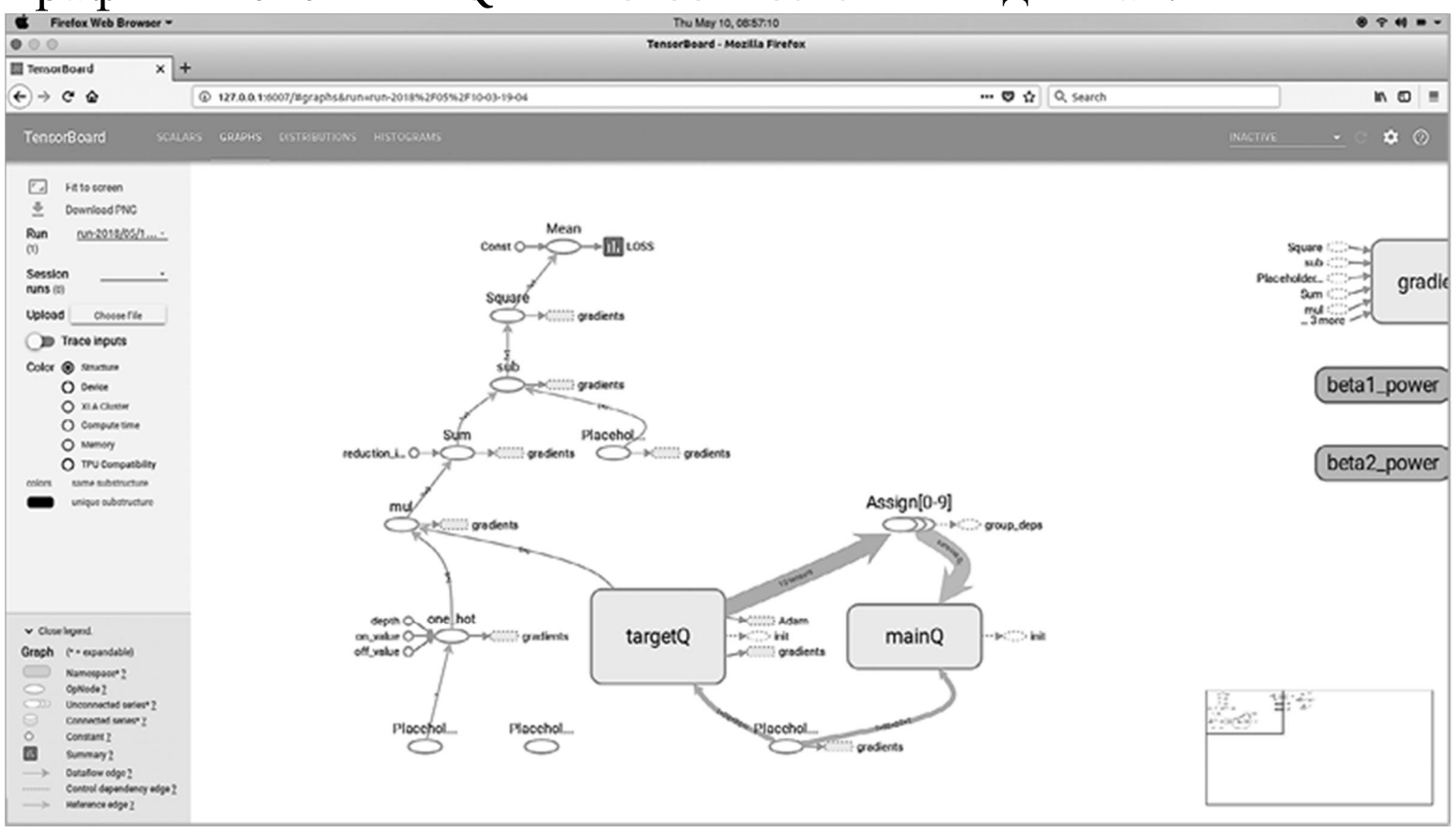
```
                # Получить действие
                action = np.argmax(actions, axis=-1)
                actions_counter[str(action)] += 1

                # Выбрать действие с использованием
                # эпсилон-жадной политики
                action = epsilon_greedy(action,
global_step)
                # Теперь выполнить действие и
перейти в следующее
                # состояние next_obs, получить
награду
                next_obs, reward, done, _ =
env.step(action)

                # Сохранить переход как опыт
                # в буфере воспроизведения
                exp_buffer.append([obs, action,
  preprocess_observation(next_obs), reward, done])
                # Через несколько шагов Q-сеть
обучается
                # на данных из буфера
воспроизведения опыта
                if global_step % steps_train == 0
and global_step >
  start_steps:
                    # Пример опыта
                    o_obs, o_act, o_next_obs, o_rew,
o_done =
  sample_memories(batch_size)

                    # состояния
                    o_obs = [x for x in o_obs]

                    # следующие состояния
                    o_next_obs = [x for x in
o_next_obs]

                    # следующие действия
                    next_act =
mainQ_outputs.eval(feed_dict={X:o_next_obs,
  in_training_mode:False})

                    # награда
```

```
                    y_batch = o_rew +
discount_factor * np.max(next_act,
  axis=-1) * (1-o_done)

                    # объединить все сводки и
записать в файл
                    mrg_summary =
merge_summary.eval(feed_dict={X:o_obs,
  y:np.expand_dims(y_batch, axis=-1),
X_action:o_act,
  in_training_mode:False})
                    file_writer.add_summary(mrg_summ
ary, global_step)

                    # Провести обучение сети и
вычислить потери
                    train_loss, _ = sess.run([loss,
training_op],
  feed_dict={X:o_obs, y:np.expand_dims(y_batch,
axis=-1), X_action:o_act,
  in_training_mode:True})
                    episodic_loss.append(train_loss)
                # Через некоторый промежуток времени
веса главной Q-сети
                # копируются в целевую Q-сеть
                if (global_step+1) % copy_steps == 0
and global_step >
  start_steps:
                    copy_target_to_main.run()
                obs = next_obs
                epoch += 1
                global_step += 1
                episodic_reward += reward
            print('Epoch', epoch, 'Reward',
episodic_reward,)
```

Выход выглядит так:

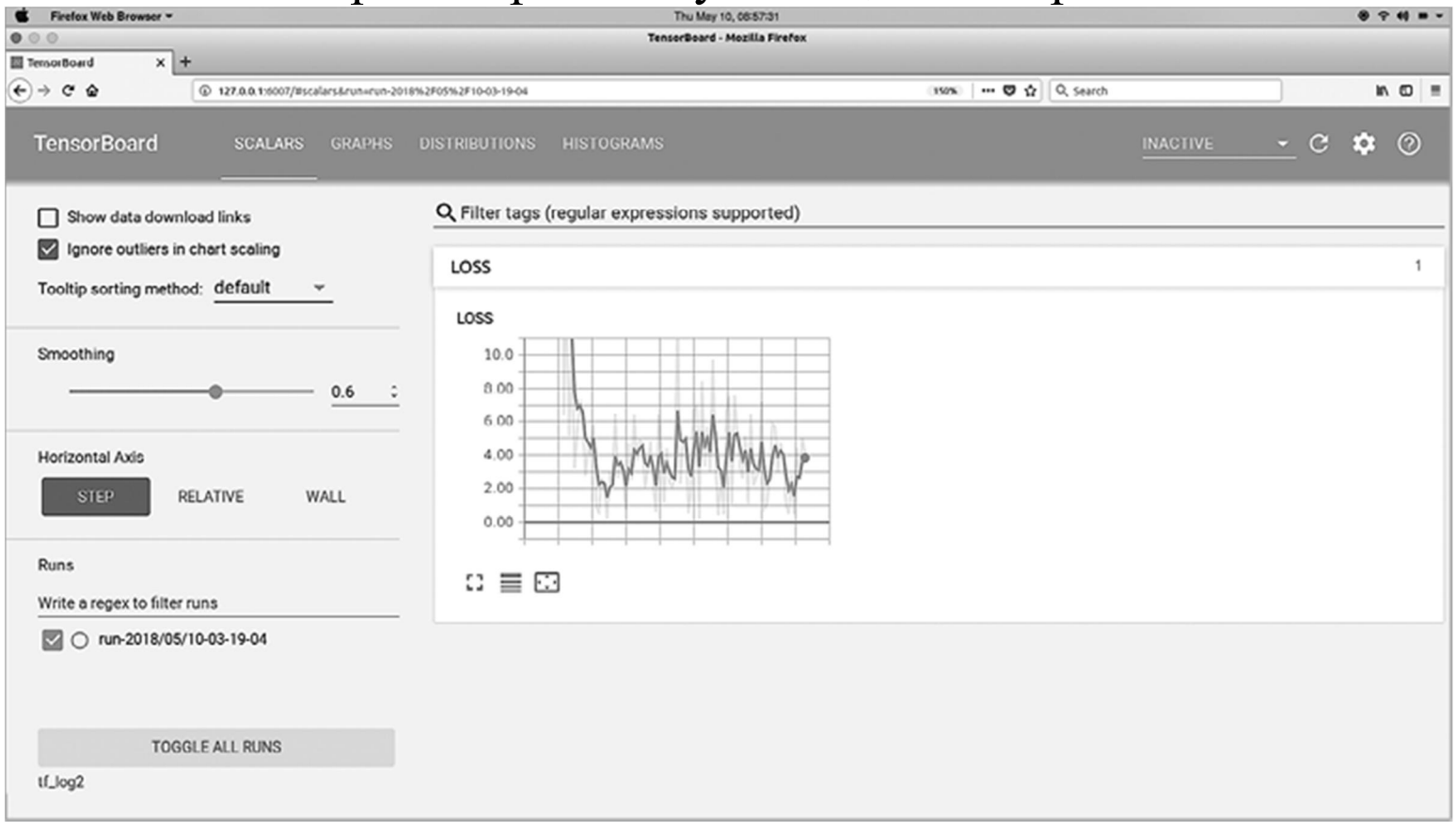


Граф вычислений DQN в TensorBoard выглядит так:



Визуализация распределения весов в основной и целевой сети имеет вид:

Также можно просмотреть визуализацию потерь:



Двойная сеть DQN

Глубокое *Q*-обучение — классная штука, верно? Его можно обобщить для любых игр Atari. Но проблема с сетями DQN заключается в том, что они склонны переоценивать значения *Q*. Это происходит из-за оператора `max`в уравнении *Q*-обучения. Оператор `max` использует одно значение для выбора и оценки действия. Что я имею в виду? Предположим, мы находимся в состоянии *s* с пятью доступными действиями от a_1 до a_5. Допустим, a_3 — лучшее действие. Оценка значений *Q*для всех действий в состоянии *s* будет сопровождаться шумом и отличаться от фактического значения *Q*. Например, из-за шума действие a_2будет обладать более высоким значением, чем оптимальное действие a_3, и будет выбрано как лучшее.

Для решения этой задачи можно определить две разные независимо обучаемые *Q*-функции. Одна будет использоваться для выбора действия, а другая — для оценки. Такую возможность можно реализовать простой настройкой целевой функции DQN. Вспомните ее:

$$y_i^{\text{DQN}} = r + \gamma \max_{a'} Q(s', a'; \theta').$$

Целевую функцию можно изменить следующим образом:

$$y_i^{\text{DoubleDQN}} = r + \gamma Q(s, \arg\max Q(s, a; \theta^-); \theta').$$

В предыдущей формуле используются две *Q*-функции с разными весами. *Q*-функция с весами θ' используется для выбора действия, а другая *Q*-функция с весами θ^- — для оценки действия. Также можно поменять местами роли этих двух *Q*-функций.

Приоритетное воспроизведение опыта

В архитектуре DQN случайная выборка точек данных из опыта используется для устранения корреляций между ними. Однако метод равномерной выборки переходов из памяти воспроизведения не является оптимальным. Вместо этого можно назначить переходам приоритеты и осуществлять выборку с учетом приоритета. Как это сделать?

Приоритетными становятся переходы с высокой погрешностью TD, которая задает разность между оцениваемым и фактическим *Q*-значением. Поскольку такие переходы отклоняются от нашей оценки, они представляют для обучения наибольший интерес. Допустим, вы пытаетесь решить набор задач, но две задачи были решены неправильно. После этого вы концентрируетесь на этих двух задачах, пытаясь понять, что пошло не так и как исправить ошибку:

Используются две основные схемы назначения приоритетов — пропорциональная и ранговая.

При *пропорциональном назначении*приоритет определяется по следующей формуле:

$p_i = (\delta_i + \varepsilon)^{\alpha}$,

где p_i — приоритет перехода *i*, δ_i — TD-погрешность перехода *i*, а ε — просто некоторая положительная константа, которая гарантирует, что каждый переход обладает ненулевым приоритетом. Если значение δ равно 0, добавление ε делает переход приоритетным (вместо нулевого приоритета). Однако такой переход будет иметь более низкий приоритет, чем переходы с ненулевым значением δ. Экспонента обозначает используемую степень приоритизации. При $\alpha = 0$ мы имеем обычный равномерный случай.

Теперь этот приоритет можно преобразовать в вероятность по следующей формуле:

$$p_i = \frac{p_i}{\sum_k p_k}.$$

При *ранговой* схеме приоритеты назначаются по формуле:

$$p_i = \left(\frac{1}{rank(i)} \right)^{\alpha}.$$

rank(*i*) задает местонахождение перехода *i* в буфере воспроизведения, в котором переходы отсортированы по убыванию погрешности TD. После вычисления приоритета мы можем преобразовать его в вероятность по той же формуле:

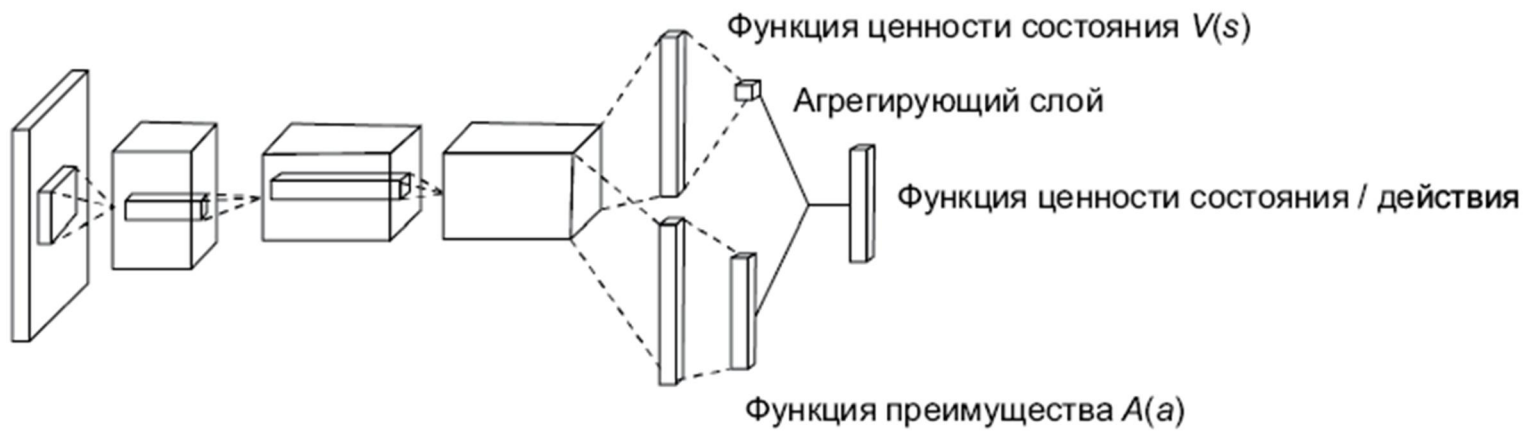
$$p_i = \frac{p_i}{\sum_k p_k}.$$

Архитектура дуэльных сетей

Мы знаем, что *Q*-функция определяет, насколько хорошо для агента выполнение действия *a* в состоянии *s*, а функция ценности определяет, насколько хорошо для агента пребывание в состоянии *s*. Теперь мы введем новую функцию, называемую *функцией преимущества*; она может быть определена как разность между функцией ценности и средней ценности этого состояния. Функция преимущества указывает, насколько хорошо для агента выполнение данного действия по сравнению с другими действиями.

Таким образом, функция ценности задает желательность состояния, а функция преимущества задает желательность действия. Что произойдет, если объединить эти функции? Результат сообщит, насколько желательно для агента выполнение действия *a* в состоянии *s*, то есть фактически мы получим *Q*-функцию. Таким образом, *Q*-функцию можно определить как сумму функции ценности и функции преимущества: $Q(s, a) = V(s) + A(a)$.

А теперь посмотрим, как работает архитектура дуэльных сетей. На следующей схеме показана архитектура дуэльной сети DQN:



Архитектура дуэльной DQN практически не отличается от архитектуры DQN, не считая того, что полносвязный уровень в конце делится на два потока. Один поток вычисляет функцию ценности, а другой — функцию преимущества. В конце эти два потока объединяются агрегирующим слоем, и мы получаем *Q*-функцию.

Зачем разбивать вычисление *Q*-функции на два потока? Во многих состояниях вычисление оценок ценности всех действий не так уж важно, особенно если состояние имеет большое пространство действий; большинство действий не будет влиять на состояние. Кроме того, возможно существование многих действий с избыточными эффектами. В

таких случаях дуэльная сеть DQN оценивает *Q* точнее существующей архитектуры DQN:

• Первый поток (поток функции ценности) полезен при большом количестве действий в состоянии, а также в ситуациях, в которых не обязательно оценивать ценность каждого действия.

• Второй поток (поток функции преимущества) полезен в тех случаях, когда сети приходится решать, какое действие является предпочтительным по сравнению с другими.

Агрегирующий слой объединяет значения этих двух потоков и формирует *Q*-функцию. Таким образом, дуэльная сеть превосходит стандартную архитектуру DQN по эффективности и надежности.

Итоги

Эта глава была посвящена очень популярному алгоритму глубокого обучения с подкреплением — DQN. Вы увидели, как глубокие нейросети используются для аппроксимации *Q*-функции, а также узнали, как построить агента для игр Atari. Здесь были рассмотрены некоторые улучшения DQN — например, двойная архитектура DQN, предотвращающая переоценку значений *Q*, приоритетное воспроизведение опыта и архитектура дуэльных сетей, которая разбивает вычисление *Q*-функции на два потока (ценности и преимущества).

В **главе 9** рассматривается интересная разновидность DQN — сети DRQN, использующие RNN для аппроксимации *Q*-функции.

Вопросы

1. Что такое DQN?
2. Для чего нужно воспроизведение опыта?
3. Зачем создавать отдельную целевую сеть?
4. Почему в алгоритме DQN возникает переоценка?
5. Как двойные сети DQN избегают переоценки *Q*?
6. Как назначаются приоритеты в приоритетном воспроизведении опыта?
7. Для чего нужна дуэльная архитектура?

Дополнительные источники

Статья о DQN: `https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf.`

Статья о двойной архитектуре DQN: `https://arxiv.org/pdf/1509.06461.pdf.`

Архитектура дуэльных сетей: `https://arxiv.org/pdf/1511.06581.pdf.`

9. Игра Doom в глубокой рекуррентной Q-сети

В главе 8 было показано, как построить на базе **DQN** (**Deep Q Network**) агента для игр Atari. Мы воспользовались нейросетями для аппроксимации *Q*-функции, использовали сверточную нейросеть (CNN) для анализа входного экрана игры и задействовали четыре последних экрана игры для лучшего понимания ее текущего состояния. В этой главе вы узнаете, как повысить эффективность DQN за счет использования **рекуррентных нейросетей** (**RNN**). Также будут рассмотрены некоторые задачи **марковского процесса принятия решений** (**MDP**) с неполной информацией и возможности их решения с использованием **DRQN**(**deep recurrent Q network**). После этого вы узнаете, как построить агента для игры в Doom с использованием DRQN. Наконец, мы рассмотрим разновидность DRQN, которая называется **DARQN** (**deep attention recurrent Q network**).

В этой главе разобраны следующие темы:

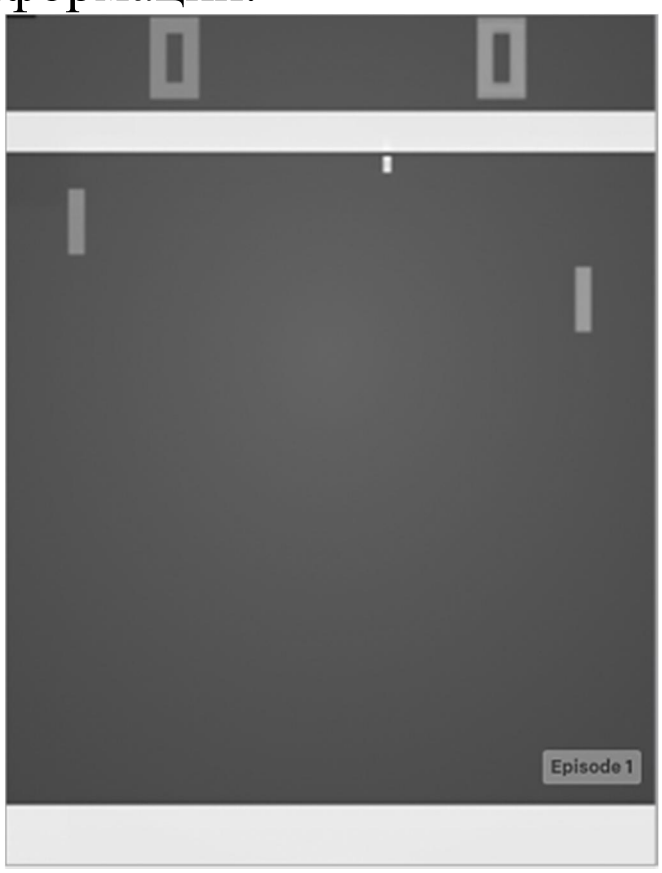
- DRQN.
- Марковский процесс принятия решений (MDP) с неполной информацией.
- Архитектура DRQN.
- Построение агента для игры в Doom с использованием DRQN.
- DARQN.

DRQN

Для чего нужны сети DRQN, когда DQN справляется с играми Atari на уровне человека? Чтобы ответить на этот вопрос, начнем с проблемы **марковского процесса принятия решений с неполной информацией**(**POMDP**). Среда называется *MDP-средой с неполной информацией*, если имеющаяся информация об этой среде ограниченна. В предыдущих главах мы имели дело с MDP-средами с полной информацией, в которых известны все возможные действия и состояния — хотя агент может не знать вероятности переходов и наград, он располагает полной информацией о среде, например, о среде замерзшего озера, в которой мы отлично знали все состояния и действия среды и легко моделировали ее в MDP-среду с полной информацией. Однако большинство реальных сред относится к категории сред с неполной информацией, где мы не видим все возможные состояния. Представьте агента, который учится ходить в реальном мире; очевидно, такой агент не обладает полной информацией ни о среде, ни о том, что находится за пределами его области видимости. В POMDP состояния частично известны, но хранение информации о прошлых состояниях в памяти может помочь агенту лучше понять природу среды и улучшить политику. Таким образом, в POMDP для выбора оптимального действия необходимо хранить информацию о предыдущих состояниях.

Чтобы вы вспомнили, о чем говорилось в предыдущих главах, представьте игру *Pong*. Глядя на текущий игровой экран, вы можете определить позицию шарика, но для выбора оптимального действия

необходимо также знать направление и скорость движения шарика. Простой взгляд на текущий игровой экран не дает нам такой информации.



Для решения этой проблемы, вместо того чтобы рассматривать только текущий игровой экран, мы будем передавать четыре игровых экрана, как делали это в DQN. Четыре игровых экрана мы передадим на вход сверточного слоя наряду с текущим игровым экраном и получим значения *Q* для всех возможных действий в состоянии. Но позволит ли использование всего четырех экранов понять разные среды? Нельзя исключать, что в некоторых средах для лучшего понимания текущего состояния игры потребуется до ста игровых экранов. С другой стороны, накопление последних *n* игровых экранов замедлит процесс обучения и увеличит размер буфера воспроизведения опыта.

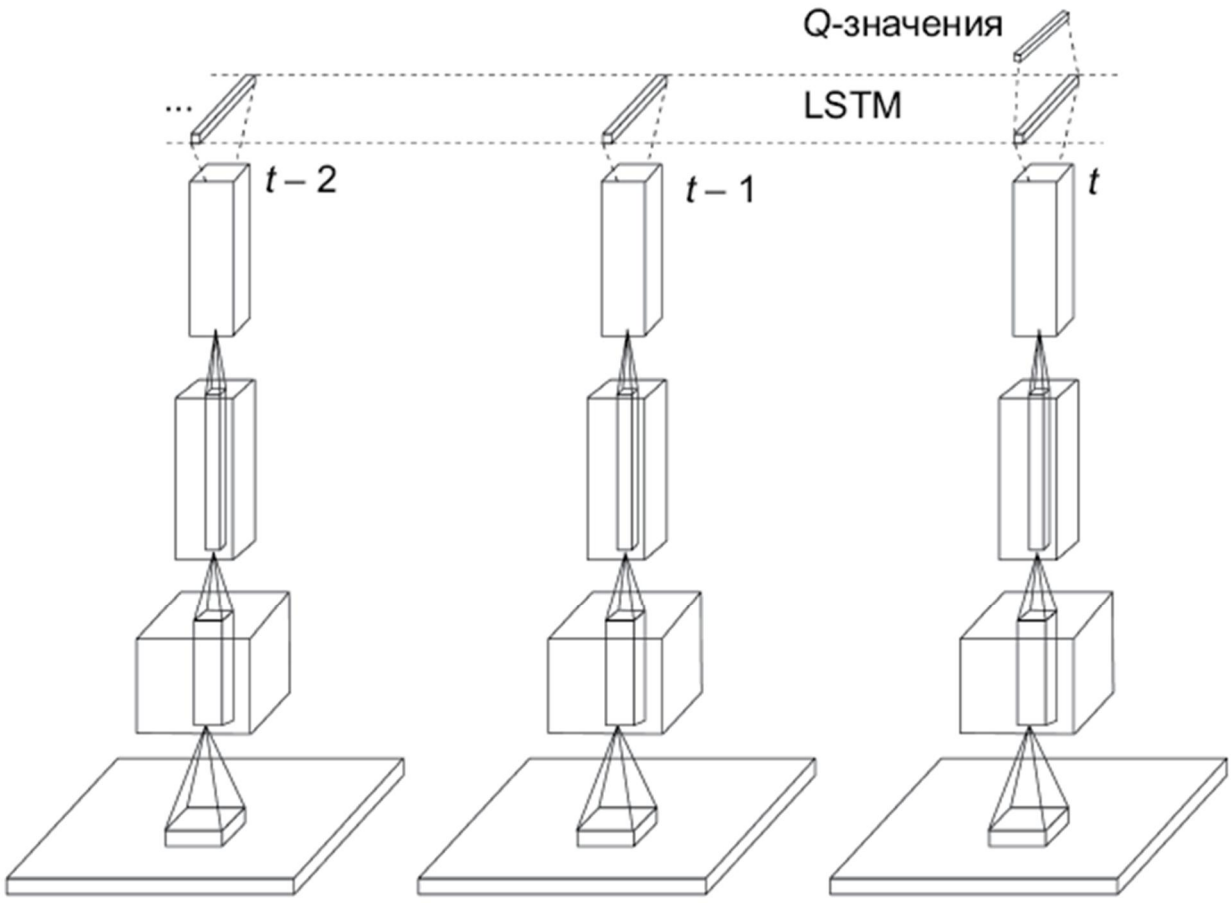
Мы можем воспользоваться RNN, чтобы понять информацию о предыдущих состояниях и хранить ее в течение любого требуемого времени. В **главе 7** мы узнали, как **рекуррентные нейросети с долгой краткосрочной памятью** (**LSTMRNN**) используются для генерирования текста и как в них формируется контекст слов за счет сохранения, стирания и обновления информации по мере необходимости. Мы изменим архитектуру DQN и дополним ее слоем LSTM для понимания предыдущей информации, в архитектуре DQN первый полносвязный слой после сверточного уровня заменим на LSTM RNN. Дополнительно решится проблема неполной информации, поскольку агент получит возможность запоминать предыдущие состояния и улучшать политику.

Архитектура DRQN

Архитектура DRQN представлена на следующей диаграмме. Она напоминает архитектуру DQN, но первый полносвязный слой после сверточного заменен в ней на LSTM RNN.

Итак, мы передаем игровой экран на вход сверточного слоя, где происходит свертка изображения и строится карта признаков, которая

передается слою LSTM. Этот слой обладает памятью, он хранит информацию о важных предыдущих состояниях игры и обновляет свою память со временем по мере необходимости. Он выводит значения *Q*, прошедшие через полносвязный слой. Таким образом, в отличие от DQN, мы не оцениваем $Q(s_t, a_t)$ напрямую. Вместо этого оценивается *estimate* $Q(h_t, a_t)$, где h_t — вход, возвращенный сетью на предыдущем временном шаге. Иначе говоря, $h_t = \text{LSTM}(h_{t-1}, a_t)$. Так как мы используем RNN, сеть обучается посредством обратного распространения во времени.



А как насчет буфера воспроизведения опыта? В DQN для предотвращения корреляции данных используется воспроизведение опыта с информацией о переходах и случайная выборка точек данных для обучения сети. В случае DRQN весь эпизод хранится в буфере опыта, и мы случайно выбираем *n* шагов из случайного пакета эпизодов. Такой подход позволяет учитывать как рандомизацию, так и прошлый опыт.

Обучение агента для игры в Doom

Doom — чрезвычайно популярный шутер с видом от первого лица. Цель игры — уничтожение монстров. *Doom*также относится к категории MDP с неполной информацией, так как область зрения агента (игрока) ограничивается 90 градусами. Агент понятия не имеет, что происходит в оставшейся части среды. Теперь разберемся, как использовать DRQN для тренировки агента, играющего в *Doom*.

Вместо OpenAI Gym мы воспользуемся пакетом ViZDoom для моделирования среды *Doom*, в которой будет осуществляться обучение агента. За дополнительной информацией о пакете ViZDoom обращайтесь на официальный веб-сайт по адресу: **`http://vizdoom.cs.put.edu.pl/`**. Для установки ViZDoom необходима следующая команда:

```
pip install vizdoom
```

ViZDoom предоставляет большую подборку сценариев *Doom*. Вы найдете эти сценарии в папке пакета `vizdoom/scenarios`.

Базовая игра Doom

Прежде чем браться за дело, познакомимся со средой `vizdoom` на простом примере.

1. Как обычно, начнем с импортирования библиотек:

```
from vizdoom import *
import random
import time
```

2. Создим экземпляр `DoomGame`:

```
game = DoomGame()
```

3. Мы знаем, что ViZDoom предоставляет множество сценариев *Doom*. Давайте загрузим базовый сценарий `basic`:

```
game.load_config("basic.cfg")
```

4. Метод `init()` инициализирует игру по сценарию:

```
game.init()
```

5. Теперь определим набор действий `actions` с onehot-кодировкой:

```
shoot = [0, 0, 1]
left = [1, 0, 0]
right = [0, 1, 0]
actions = [shoot, left, right]
```

6. Можно начать игру:

```
no_of_episodes = 10

for i in range(no_of_episodes):
    # Для каждого эпизода - начать игру
    game.new_episode()
    # Выполнять в цикле до завершения эпизода
    while not game.is_episode_finished():
        # Получить состояние игры
        state = game.get_state()
        img = state.screen_buffer
        # Получить игровые переменные
        misc = state.game_variables
        # Выполнить случайное действие и получить награду
        reward = game.make_action(random.choice(actions))
        print(reward)
    # Небольшая пауза перед началом следующего эпизода
    time.sleep(2)
```

После запуска программы мы увидим вывод:

Doom с DRQN

Теперь посмотрим, как использовать алгоритм DRQN для обучения агента в игре *Doom*. Положительные награды будут присваиваться за уничтожение монстров, а отрицательные — за потерю жизни и боеприпасов. Полный код можно загрузить в формате Jupyter с пояснениями по адресу: **https://github.com/sudharsan13296/Hands-On-Reinforcement-Learning-With-Python/blob/master/09.%20Playing%20Doom%20Game%20using%20DRQN/9.5%20Doom%20Game%20Using%20DRQN.ipynb**.

За код, приведенный в этом разделе, благодарю Luthanicus (**https://github.com/Luthanicus/losaltoshackathon-drqn**).

Начнем с импортирования всех необходимых библиотек:

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from vizdoom import *
import timeit
import math
import os
import sys
```

Определим функцию `get_input_shape` для вычисления окончательной формы входного изображения после его обработки на сверточном слое:

```
def get_input_shape(Image,Filter,Stride):
```

```
    layer1 = math.ceil(((Image - Filter + 1) /
Stride))
    o1 = math.ceil((layer1 / Stride))
    layer2 = math.ceil(((o1 - Filter + 1) /
Stride))
    o2 = math.ceil((layer2 / Stride))
    layer3 = math.ceil(((o2 - Filter + 1) /
Stride))
    o3 = math.ceil((layer3 / Stride))
    return int(o3)
```

Теперь определим класс `DRQN`, реализующий алгоритм DRQN. Комментарии перед каждой строкой кода помогут вам понять его смысл:

```
class DRQN():
    def __init__(self, input_shape, num_actions,
initial_learning_rate):
        # Инициализировать все гиперпараметры

        self.tfcast_type = tf.float32
        # Форма входных данных (длина, ширина,
каналы)
        self.input_shape = input_shape
        # Количество действий в среде
        self.num_actions = num_actions
        # Скорость обучения для нейросети
        self.learning_rate =
initial_learning_rate
        # Теперь определяем гиперпараметры
сверточной нейросети

        # Размер фильтра
        self.filter_size = 5
        # Количество фильтров
        self.num_filters = [16, 32, 64]
        # Размер шага
        self.stride = 2
        # Размер пула
        self.poolsize = 2
        # Форма сверточного уровня
        self.convolution_shape =
get_input_shape(input_shape[0],
  self.filter_size, self.stride) *
get_input_shape(input_shape[1],
  self.filter_size, self.stride) *
self.num_filters[2]
```

```
            # Теперь мы определяем гиперпараметры
для рекуррентной
            # нейросети и завершающего уровня
            # Количество нейронов
            self.cell_size = 100
            # Количество скрытых уровней
            self.hidden_layer = 50
            # Вероятность прореживания
            self.dropout_probability = [0.3, 0.2]

            # Гиперпараметры для оптимизации
            self.loss_decay_rate = 0.96
            self.loss_decay_steps = 180

            # Инициализировать все переменные для
CNN

            # Инициализировать заместителя для ввода
с формой
            # (длина, ширина, канал)
            self.input = tf.placeholder(shape =
(self.input_shape[0],
  self.input_shape[1], self.input_shape[2]), dtype
= self.tfcast_type)
            # Также инициализировать форму целевого
вектора, размер которого
            # равен количеству действий
            self.target_vector =
tf.placeholder(shape = (self.num_actions, 1),
  dtype = self.tfcast_type)

            # Инициализировать карты признаков для
соответствующих 3 фильтров
            self.features1 =
tf.Variable(initial_value =
  np.random.rand(self.filter_size,
self.filter_size, input_shape[2],
  self.num_filters[0]),
                                                dtype =
self.tfcast_type)
            self.features2 =
tf.Variable(initial_value =
  np.random.rand(self.filter_size,
self.filter_size, self.num_filters[0],
```

```
  self.num_filters[1]),
                                                dtype =
self.tfcast_type)
            self.features3 =
tf.Variable(initial_value =
  np.random.rand(self.filter_size,
self.filter_size, self.num_filters[1],
  self.num_filters[2]),
                                                dtype =
self.tfcast_type)

            # Инициализировать переменные для RNN
            # О том, как работают RNN, рассказано в
главе 7
            self.h = tf.Variable(initial_value =
np.zeros((1, self.cell_size)),
  dtype = self.tfcast_type)
            # матрица весов от скрытого к скрытому
уровню
            self.rW = tf.Variable(initial_value =
np.random.uniform(
                                                  low
= -np.sqrt(6. /
  (self.convolution_shape + self.cell_size)),
                                                  high
= np.sqrt(6. /
  (self.convolution_shape + self.cell_size)),
                                                  size
= (self.convolution_shape,
  self.cell_size)),
                                        dtype =
self.tfcast_type)
            # матрица весов от входного к скрытому
уровню
            self.rU = tf.Variable(initial_value =
np.random.uniform(
                                                  low
= -np.sqrt(6. / (2 *
  self.cell_size)),
                                                  high
= np.sqrt(6. / (2 *
  self.cell_size)),
                                                  size
= (self.cell_size,
```

```
  self.cell_size)),
                                        dtype =
self.tfcast_type)
            # матрица весов от скрытого к выходному
уровню
            self.rV = tf.Variable(initial_value =
np.random.uniform(
                                                low
= -np.sqrt(6. / (2 *
  self.cell_size)),
                                                high
= np.sqrt(6. / (2 *
  self.cell_size)),
                                                size
= (self.cell_size,
  self.cell_size)),
                                        dtype =
self.tfcast_type)
            # смещение
            self.rb = tf.Variable(initial_value =
np.zeros(self.cell_size),
  dtype = self.tfcast_type)
            self.rc = tf.Variable(initial_value =
np.zeros(self.cell_size),
  dtype = self.tfcast_type)
            # Инициализировать веса и смещение сети
напрямую
            self.fW = tf.Variable(initial_value =
np.random.uniform(
                                                low
= -np.sqrt(6. /
  (self.cell_size + self.num_actions)),
                                                high
= np.sqrt(6. /
  (self.cell_size + self.num_actions)),
                                                size
= (self.cell_size,
  self.num_actions)),
                                        dtype =
self.tfcast_type)
            # смещение
            self.fb = tf.Variable(initial_value =
np.zeros(self.num_actions),
  dtype = self.tfcast_type)
```

```
        # скорость обучения
        self.step_count =
tf.Variable(initial_value = 0, dtype =
  self.tfcast_type)
        self.learning_rate =
tf.train.exponential_decay(self.learning_rate,
   self.step_count,
   self.loss_decay_steps,
   self.loss_decay_steps,
   staircase = False)
        # Переходим к построению сети

        # Первый сверточный уровень
        self.conv1 = tf.nn.conv2d(input =
tf.reshape(self.input, shape =
  (1, self.input_shape[0], self.input_shape[1],
self.input_shape[2])), filter
  = self.features1, strides = [1, self.stride,
self.stride, 1], padding =
  "VALID")
        self.relu1 = tf.nn.relu(self.conv1)
        self.pool1 = tf.nn.max_pool(self.relu1,
ksize = [1, self.poolsize,
  self.poolsize, 1], strides = [1, self.stride,
self.stride, 1], padding =
  "SAME")

        # Второй сверточный уровень
        self.conv2 = tf.nn.conv2d(input =
self.pool1, filter =
  self.features2, strides = [1, self.stride,
self.stride, 1], padding =
  "VALID")
        self.relu2 = tf.nn.relu(self.conv2)
        self.pool2 = tf.nn.max_pool(self.relu2,
ksize = [1, self.poolsize,
  self.poolsize, 1], strides = [1, self.stride,
self.stride, 1], padding =
  "SAME")
```

```
            # Третий сверточный уровень
            self.conv3 = tf.nn.conv2d(input =
self.pool2, filter =
  self.features3, strides = [1, self.stride,
self.stride, 1], padding =
  "VALID")
            self.relu3 = tf.nn.relu(self.conv3)
            self.pool3 = tf.nn.max_pool(self.relu3,
ksize = [1, self.poolsize,
  self.poolsize, 1], strides = [1, self.stride,
self.stride, 1], padding =
  "SAME")

            # Добавить прореживание и изменить форму
ввода
            self.drop1 = tf.nn.dropout(self.pool3,
self.dropout_probability[0])
            self.reshaped_input =
tf.reshape(self.drop1, shape = [1, -1])

            # Теперь строим рекуррентную нейросеть,
которая получает
            # ввод с последнего уровня сверточной
сети
            self.h =
tf.tanh(tf.matmul(self.reshaped_input, self.rW) +
  tf.matmul(self.h, self.rU) + self.rb)
            self.o = tf.nn.softmax(tf.matmul(self.h,
self.rV) + self.rc)

            # Добавить прореживание в RNN
            self.drop2 = tf.nn.dropout(self.o,
self.dropout_probability[1])
            # Результат RNN передается уровню прямой
подачи
            self.output =
tf.reshape(tf.matmul(self.drop2, self.fW) +
self.fb,
  shape = [-1, 1])
            self.prediction = tf.argmax(self.output)

            # Вычислить потери
```

```
            self.loss =
tf.reduce_mean(tf.square(self.target_vector -
  self.output))
            # Оптимизатор Adam используется для
минимизации погрешности
            self.optimizer =
tf.train.AdamOptimizer(self.learning_rate)
            # Вычислить градиенты потерь и обновить
градиенты
            self.gradients =
self.optimizer.compute_gradients(self.loss)
            self.update =
self.optimizer.apply_gradients(self.gradients)

            self.parameters = (self.features1,
self.features2, self.features3,
                               self.rW, self.rU,
self.rV, self.rb, self.rc,
                               self.fW, self.fb)
```

Определим класс `ExperienceReplay` для реализации функций буфера воспроизведения опыта. Весь опыт агента (то есть состояние, действие и награды) сохраняется в буфере воспроизведения опыта, и мы формируем мини-выборку опыта для обучения сети:

```
class ExperienceReplay():
    def __init__(self, buffer_size):
        # Буфер для хранения перехода
        self.buffer = []
        # Размер буфера
        self.buffer_size = buffer_size
    # Удалить старый переход, если размер буфера
достиг предела.
    # Буфер работает по принципу очереди: когда
приходит новый элемент,
    # старый элемент уходит!
    def appendToBuffer(self, memory_tuplet):
        if len(self.buffer) > self.buffer_size:
            for i in range(len(self.buffer) -
self.buffer_size):
                self.buffer.remove(self.buffer[0
])
        self.buffer.append(memory_tuplet)
    # Определить функцию sample для выборки
случайного числа n
    # из переходов
```

```
    def sample(self, n):
        memories = []
        for i in range(n):
            memory_index = np.random.randint(0,
len(self.buffer))
            memories.append(self.buffer[memory_i
ndex])
        return memories
```

Определим функцию `train` для обучения сети:

```
def train(num_episodes, episode_length,
learning_rate, scenario =
"deathmatch.cfg", map_path = 'map02', render =
False):
    # Параметр для вычисления значения Q
    discount_factor = .99
    # Частота обновления опыта в буфере
    update_frequency = 5
    store_frequency = 50
    # Для вывода
    print_frequency = 1000

    # Инициализировать переменные для хранения
    # суммарных наград и суммарных потерь
    total_reward = 0
    total_loss = 0
    old_q_value = 0

    # Инициализировать списки для хранения
    # эпизодических наград и потерь
    rewards = []
    losses = []

    # А теперь можно браться за дело!
    # Сначала инициализировать среду doomgame
    game = DoomGame()
    # Местонахождение файла сценария
    game.set_doom_scenario_path(scenario)
    # Задать путь к файлу карты
    game.set_doom_map(map_path)

    # Задать разрешение и формат экрана
    game.set_screen_resolution(ScreenResolution.
RES_256X160)
    game.set_screen_format(ScreenFormat.RGB24)
```

```
    # Чтобы добавить частицы и эффекты, просто
задайте соответствующей
    # переменной значение true или false
    game.set_render_hud(False)
    game.set_render_minimal_hud(False)
    game.set_render_crosshair(False)
    game.set_render_weapon(True)
    game.set_render_decals(False)
    game.set_render_particles(False)
    game.set_render_effects_sprites(False)
    game.set_render_messages(False)
    game.set_render_corpses(False)
    game.set_render_screen_flashes(True)

    # Кнопки действий, которые должны быть
доступны для агента
    game.add_available_button(Button.MOVE_LEFT)
    game.add_available_button(Button.MOVE_RIGHT)
    game.add_available_button(Button.TURN_LEFT)
    game.add_available_button(Button.TURN_RIGHT)
    game.add_available_button(Button.MOVE_FORWAR
D)
    game.add_available_button(Button.MOVE_BACKWA
RD)
    game.add_available_button(Button.ATTACK)
    # Добавим еще одну кнопку, которая
называется DELTA. Предыдущие
    # кнопки работают по принципу клавиш на
клавиатуре
    # и принимают только логические значения.

    # Кнопка DELTA эмулирует мышь с
положительными
    # и отрицательными значениями. Она будет
полезна в среде
    # для проведения исследования.
    game.add_available_button(Button.TURN_LEFT_R
IGHT_DELTA, 90)
    game.add_available_button(Button.LOOK_UP_DOW
N_DELTA, 90)

    # Инициализировать массив для действий
```

```
    actions = np.zeros((game.get_available_buttons_size(),
      game.get_available_buttons_size()))
    count = 0
    for i in actions:
        i[count] = 1
        count += 1
    actions = actions.astype(int).tolist()

    # Добавить игровые переменные, счетчики боеприпасов,
    # здоровья и убийств
    game.add_available_game_variable(GameVariable.AMMO0)
    game.add_available_game_variable(GameVariable.HEALTH)
    game.add_available_game_variable(GameVariable.KILLCOUNT)

    # Установить episode_timeout, чтобы эпизод автоматически
    # прерывался после некоторого временного шага.
    # Также устанавливается переменная episode_start_time
    # для пропуска исходных событий
    game.set_episode_timeout(6 * episode_length)
    game.set_episode_start_time(10)
    game.set_window_visible(render)
    # Можно включить звук, присвоив set_sound_enable значение true
    game.set_sound_enabled(False)

    # Если присвоить living_reward значение 0, агент будет
    # награждаться за каждое перемещение, даже если оно
    # не приносит пользы
    game.set_living_reward(0)

    # В doom предусмотрены разные режимы: игрок, зритель, асинхронный
    # игрок и асинхронный зритель.
```

```
    # В режиме зрителя люди играют, а агент обучается на примере их игры.
    # В режиме игрока агент играет сам, поэтому мы выбираем
    # именно этот режим.
    game.set_mode(Mode.PLAYER)

    # Инициализировать игровую среду.
    game.init()

    # Теперь создать экземпляр класса DRQN и создать
    # сети DRQN (сеть действий и целевую сеть)
    actionDRQN = DRQN((160, 256, 3), game.get_available_buttons_size() - 2, learning_rate)
        targetDRQN = DRQN((160, 256, 3), game.get_available_buttons_size() - 2, learning_rate)
        # Также создать экземпляр класса ExperienceReplay с размером буфера 1000
        experiences = ExperienceReplay(1000)

        # Для хранения моделей
        saver = tf.train.Saver({v.name: v for v in actionDRQN.parameters}, max_to_keep = 1)

        # Теперь запустить процесс обучения.
        # Инициализировать переменные для выборки и хранения переходов
        # из буфера опыта
        sample = 5
        store = 50
        # Запустить сеанс tensorflow
        with tf.Session() as sess:
            # Инициализировать все переменные tensorflow
            sess.run(tf.global_variables_initializer())
            for episode in range(num_episodes):
                # Запустить новый эпизод
                game.new_episode()
```

```
                # Играть, пока не будет достигнуто
ограничение
                # по длине эпизода
                for frame in range(episode_length):
                    # Получить состояние игры
                    state = game.get_state()
                    s = state.screen_buffer
                    # Выбрать действие
                    a =
actionDRQN.prediction.eval(feed_dict =
  {actionDRQN.input: s})[0]
                    action = actions[a]
                    # Выполнить действие и сохранить
награду
                    reward =
game.make_action(action)
                    # Обновить суммарную награду
                    total_reward += reward

                    # Если эпизод закончен, прервать
цикл
                    if game.is_episode_finished():
                        break
                    # Сохранить переход в буфере
опыта
                    if (frame % store) == 0:
                        experiences.appendToBuffer((
s, action, reward))

                    # Произвести выборку из буфера
опыта
                    if (frame % sample) == 0:
                        memory =
experiences.sample(1)
                        mem_frame = memory[0][0]
                        mem_reward = memory[0][2]
                        # Провести обучение сети
                        Q1 =
actionDRQN.output.eval(feed_dict =
  {actionDRQN.input: mem_frame})
                        Q2 =
targetDRQN.output.eval(feed_dict =
  {targetDRQN.input: mem_frame})
```

```
                        # Задать скорость обучения
                        learning_rate =
actionDRQN.learning_rate.eval()
                        # Вычислить значение Q
                        Qtarget = old_q_value +
learning_rate * (mem_reward +
  discount_factor * Q2 - old_q_value)
                        # Обновить значение Q
                        old_q_value = Qtarget

                        # Вычислить потери
                        loss =
actionDRQN.loss.eval(feed_dict =
  {actionDRQN.target_vector: Qtarget,
actionDRQN.input: mem_frame})
                        # Обновить общие потери
                        total_loss += loss

                        # Обновить обе сети
                        actionDRQN.update.run(feed_d
ict =
  {actionDRQN.target_vector: Qtarget,
actionDRQN.input: mem_frame})
                        targetDRQN.update.run(feed_d
ict =
  {targetDRQN.target_vector: Qtarget,
targetDRQN.input: mem_frame})

                rewards.append((episode,
total_reward))
                losses.append((episode, total_loss))

                print("Episode %d - Reward = %.3f,
Loss = %.3f." % (episode,
  total_reward, total_loss))

                total_reward = 0
                total_loss = 0
```

Проведем обучение на 10 000 эпизодах, каждый из которых имеет длину 300:

```
  train(num_episodes = 10000, episode_length =
300, learning_rate = 0.01,
  render = True)
```

При запуске программы воспроизводится вывод, показанный на следующей иллюстрации. Вы можете проследить за тем, как агент обучается от эпизода к эпизоду:

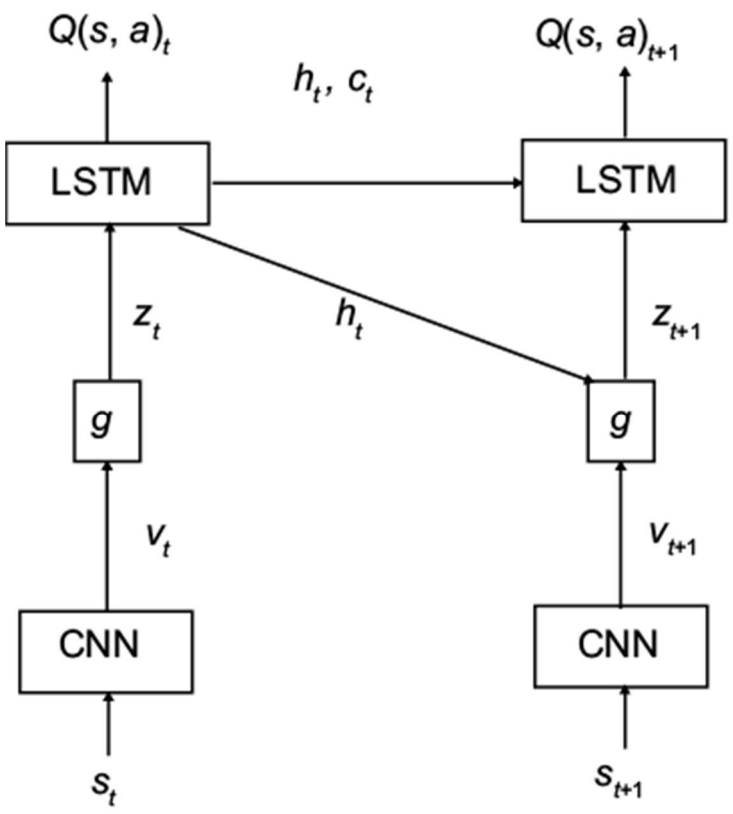
DARQN

Мы улучшили архитектуру DQN, добавив рекуррентный слой для сохранения временной зависимости, и назвали результат DRQN. Как вы думаете, можно ли продолжить улучшение архитектуры DRQN? Да, можно добавить *слой внимания*(attention) над сверточным слоем. Для чего? В данном случае слово «внимание» интерпретируется вполне буквально. Механизмы внимания широко применяются в разметке изображений, обнаружении объектов и т.д. Представьте задачу разметить изображения с помощью нейросети; чтобы понять, что находится на изображении, и сгенерировать подпись, сеть должна уделить внимание конкретному объекту.

Аналогичным образом мы можем уделить внимание небольшим областям изображения. В конечном итоге это приводит к сокращению количества параметров в сети, времени обучения и тестирования. В отличие от DRQN, слои LSTM в DARQN не ограничиваются хранением информации предыдущего состояния для выбора следующего оптимального действия; они также сохраняют информацию для принятия решения о том, на какой области изображения следует сосредоточиться на следующем этапе.

Архитектура DARQN

Архитектура DARQN изображена на следующем рисунке:



Она состоит из трех типов слоев: сверточного, внимания и рекуррентных LSTM. Игровой экран передается в виде графического изображения сверточной сети, которая обрабатывает изображение и генерирует карты признаков. Затем карты признаков передаются слою внимания, где они преобразуются в вектор, а результаты — в их линейную комбинацию (контекстные векторы). Контекстные векторы наряду с предыдущими скрытыми состояниями затем передаются уровню LSTM, где выдаются два результата: в одном содержится значение *Q* для принятия решений о том, какое действие должно выполняться в состоянии, в другом — решение, на какой области изображения следует сосредоточиться на следующем временном шаге, чтобы сгенерировать более качественные контекстные векторы.

Внимание делится на две категории.

• *Мягкий механизм внимания* (soft attention): мы знаем, что карты признаков, произведенные сверточным слоем, подаются на вход слою внимания, который затем генерирует контекстный вектор. С мягким механизмом внимания эти контекстные векторы представляют собой взвешенное среднее всех выходных результатов (карт признаков), произведенных сверточным слоем. Веса выбираются в соответствии с относительной важностью признаков.

• *Жесткий механизм внимания*: с жестким механизмом внимания мы ограничиваемся конкретным участком изображения в момент времени t согласно некоторой политике выбора участка π. Политика представляется нейронной сетью, в которой веса являются параметрами политики, а выход определяет вероятность выбора участка. Впрочем, жесткие механизмы внимания по эффективности лишь незначительно превосходят мягкие.

Итоги

В этой главе вы узнали, как DRQN используется для запоминания информации о предыдущих состояниях и как решается проблема MDP с неполной информацией. Вы увидели, как обучить агента для игры в Doom с использованием алгоритма DRQN. Также была рассмотрена

архитектура DARQN как усовершенствованная разновидность DRQN, добавляющая слой внимания поверх сверточного слоя. После этого были представлены две разновидности механизма внимания, а именно мягкий и жесткий.

В **главе 10** будет рассмотрена другая интересная разновидность алгоритма глубокого обучения с подкреплением — асинхронная преимущественная сеть «актор-критик».

Вопросы

1. Чем DQN отличается от DRQN?
2. Какими недостатками обладает DQN?
3. Как организуется воспроизведение опыта в DQN?
4. Чем DRQN отличается от DARQN?
5. Для чего нужны сети DARQN?
6. Какие существуют разновидности механизма внимания?
7. Для чего включается режим `living_reward` в *Doom*?

Дополнительные источники

Статья о DRQN: `https://arxiv.org/pdf/1507.06527.pdf`.

Применение DRQN в FPS-играх: `https://arxiv.org/pdf/1609.05521.pdf`.

Статья о DARQN: `https://arxiv.org/pdf/1512.01693.pdf`.

10. Асинхронная преимущественная сеть «актор-критик»

В предыдущих главах вы видели, какими возможностями обладают **DQN**, в том числе в играх Atari. Однако мы столкнулись с проблемой: такие сети требуют значительных вычислительных мощностей и времени обучения. По этой причине компания DeepMind, принадлежащая Google, разработала новый алгоритм, называемый **асинхронным преимущественным алгоритмом «актор-критик»** (**A3C**, Asynchronous Advantage Actor Critic), который превосходит другие алгоритмы глубокого обучения с подкреплением, так как он требует меньших вычислительных мощностей и времени обучения. Главная идея в основе A3C — использование нескольких агентов, которые обучаются параллельно и объединяют накопленный опыт. В этой главе вы узнаете, как работает A3C. После этого я расскажу, как построить агента для подъема на гору на машине с использованием A3C.

В этой главе разобраны следующие темы:

- Алгоритм асинхронной преимущественной сети «актор-критик».
- Три «А».
- Архитектура A3C.
- Как работает A3C.

- Подъем на гору с использованием A3C.
- Визуализация средствами TensorBoard.

Асинхронный преимущественный алгоритм «актор-критик»

A3C произвел фурор и занял место DQN. Кроме уже упоминавшихся его преимуществ, он также обеспечивает неплохую точность по сравнению с другими алгоритмами, а также хорошо работает как в непрерывных, так и в дискретных пространствах действий. В нем используются несколько агентов, которые учатся параллельно с разными политиками исследования в отдельных копиях фактической среды. Затем опыт, полученный агентами, обобщается глобальным агентом. Глобальный агент также называется *главной сетью*, или *глобальной сетью*, а агенты называются *работниками* (workers). Теперь мы подробно рассмотрим, как работает A3C и чем он отличается от DQN.

Три «А»

Прежде чем переходить к подробностям, уточним, что вообще означает название A3C. Почему «три А»?

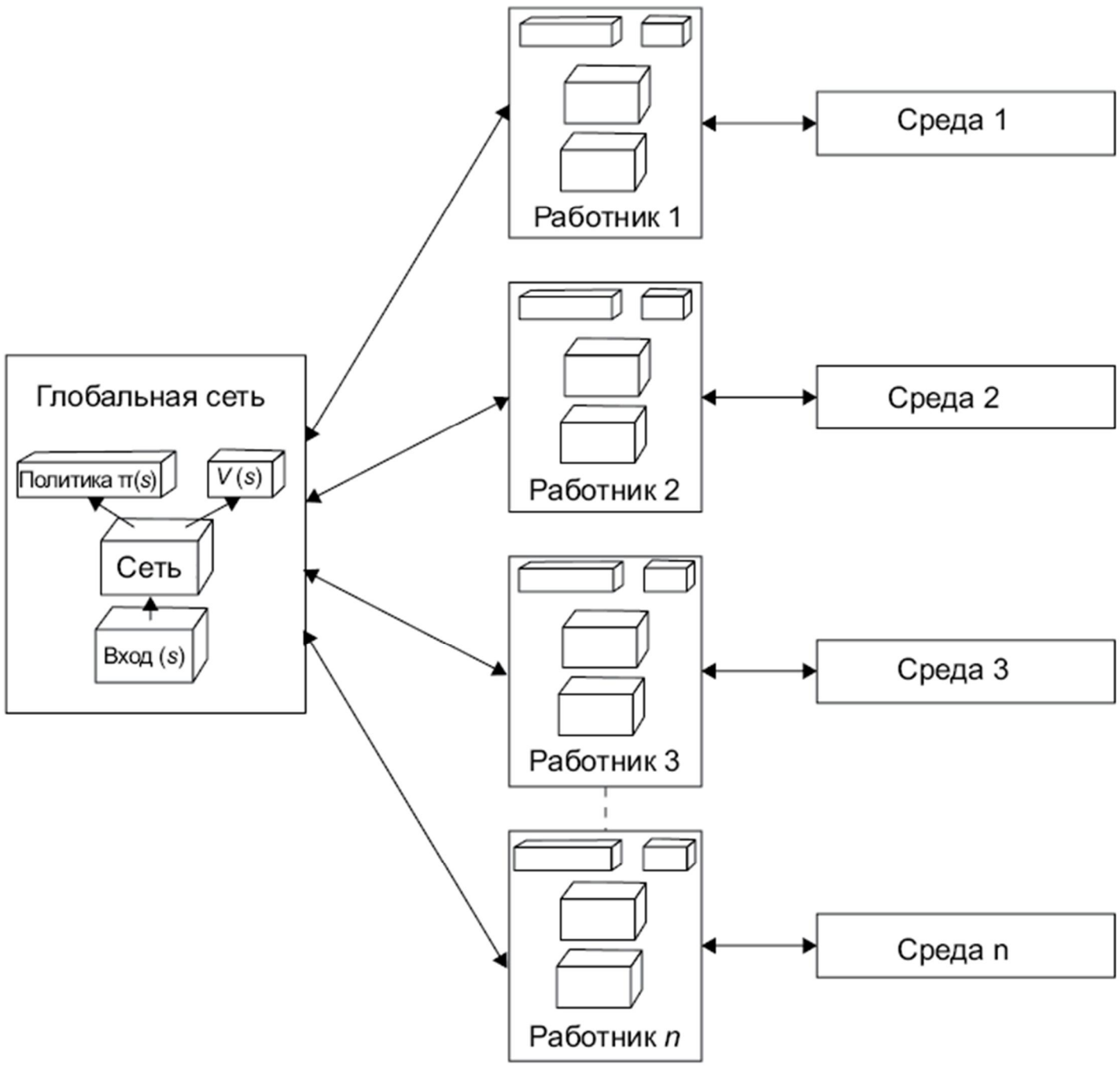
Первое «А» — от «asynchronous» (асинхронность) — указывает, как работает алгоритм. Вместо одного агента, ищущего оптимальную политику (как в DQN), используются несколько агентов, взаимодействующих со средой одновременно, и каждому из них предоставляется для работы собственная копия среды. Эти агенты (работники) сообщают свои результаты отдельному агенту — глобальной сети, которая накапливает результаты обучения.

Второе «А» — от «advantage» (преимущество). Вы уже видели функцию преимущества при обсуждении дуэльной архитектуры DQN. Функция преимущества может быть определена как разность между Q-функцией и функцией ценности. Мы знаем, что Q-функция задает желательность действия в состоянии, а функция ценности — желательность самого состояния. А теперь задумайтесь, что по сути означает разность между ними? Она указывает, насколько хорошо для агента выполнить действие a в состоянии sпо сравнению с другими действиями.

Третье «А» — от «actor critic» (актор-критик). В этой архитектуре существуют два типа сетей: актор и критик. Задача актора — изучить политики, а критика — оценить, насколько хорошую политику изучил актор.

Архитектура A3C

Рассмотрим архитектуру A3C. Взгляните на следующий рисунок:



Как упоминалось ранее, каждый из агентов-работников взаимодействует с собственной копией среды. Работник обучается политике, вычисляет градиент ее потерь и обновляет градиенты в глобальной сети. Глобальная сеть одновременно обновляется всеми агентами. Одно из преимуществ A3C заключается в том, что, в отличие от DQN, здесь не используется память воспроизведения опыта, так как корреляция между данными опыта разных агентов практически полностью исключается. Механизм воспроизведения опыта требует большого объема памяти. A3C это не нужно, затраты памяти и времени вычислений сокращаются.

Как работает A3C

Сначала агенты-работники сбрасывают глобальную сеть и начинают взаимодействовать со средой. Каждый работник применяет свою политику исследования в поиске оптимальной политики. После этого они вычисляют ценность и потери политики, а также градиент потерь, после чего обновляют градиенты глобальной сети. Цикл продолжается: агенты-работники сбрасывают глобальную сеть и повторяют тот же процесс. Прежде чем разобрать функцию ценности и функцию потерь политики, посмотрим, как вычисляется функция преимущества. Известно, что преимущество равно разности между Q-функцией и функцией ценности:

$$A(s, a) = Q(s, a) - V(s).$$

Так как значение Q не вычисляется в A3C напрямую, мы используем скорректированный возврат R как оценку значения Q. Он может быть записан в следующем виде:

$$R = r_n + \gamma r_{n-1} + \gamma^2 r_{n-2}$$

Q-функция заменяется скорректированным возвратом Rследующим образом:

$A(s, a) = R - V(s)$.

Теперь можно записать потерю ценности как квадрат разности между скорректированным возвратом и ценностью состояния:

$$\text{Потеря ценности } (L_v) = \sum (R - V(s))^2.$$

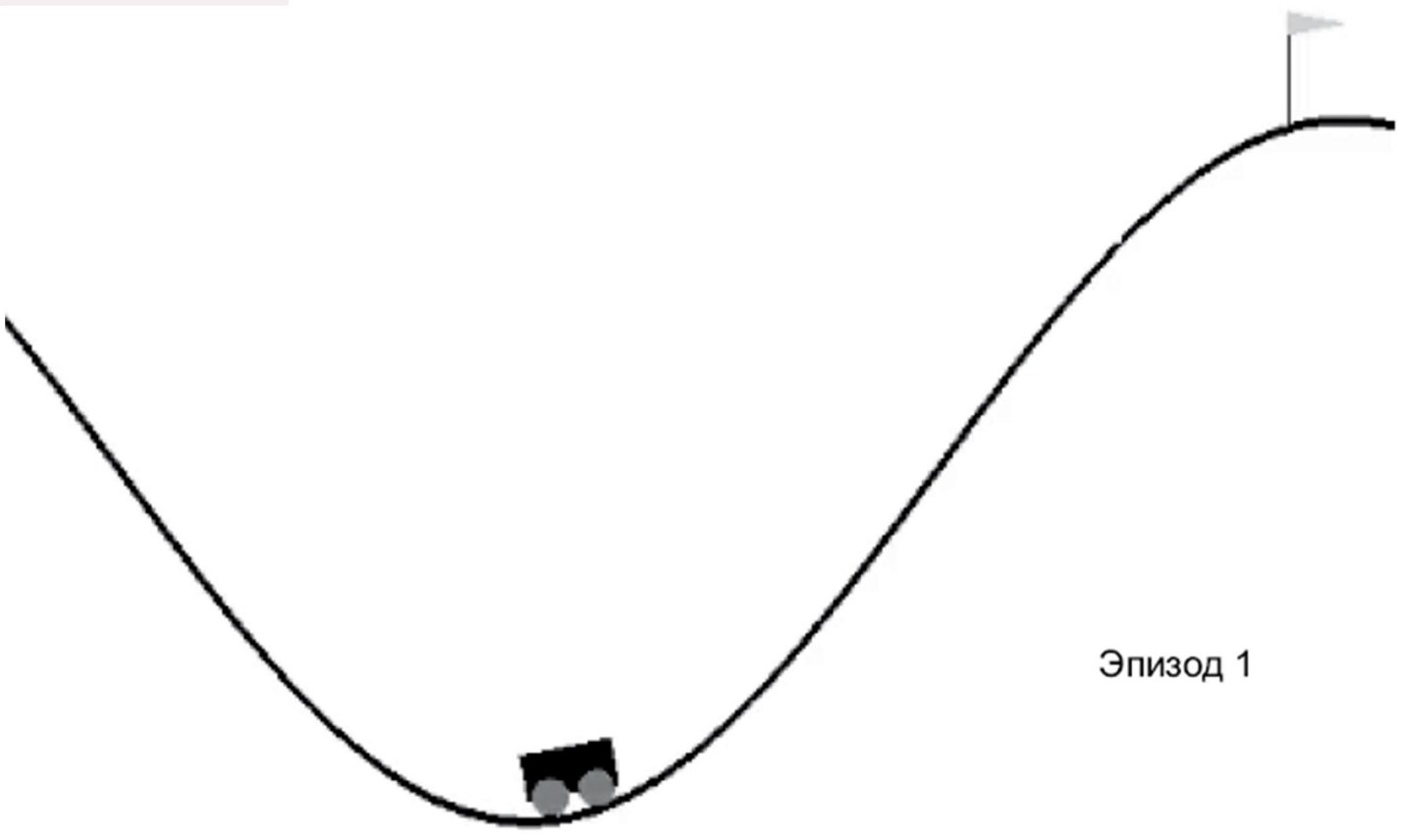
А потеря политики может быть определена следующим образом:

Потеря политики $(L_p) = \log(\pi(s)) \times A(s) \times \beta H(\pi)$.

Новая составляющая $H(\pi)$ — энтропия — обеспечивает достаточное исследование политики. Она оценивает разброс вероятностей действий. При высокой энтропии все действия имеют близкие вероятности, поэтому агент будет не уверен в том, какое действие следует выполнить. При снижении энтропии одно действие имеет более высокую вероятность, чем другие, и агент может выбрать действие с высокой вероятностью. Таким образом, включение энтропии в функцию потерь заставляет агента продолжать исследования и не застревать в локальных оптимумах.

Подъем на гору с использованием A3C

Чтобы лучше понять логику A3C, рассмотрим пример с подъемом на гору. Наш агент — автомобиль — находится между двумя горами; цель — подняться на гору справа. Машина не может заехать на гору за один раз; ей придется двигаться назад и вперед, чтобы набрать импульс. Если агент тратит меньше энергии на подъем, он получает высокую награду. Код этого раздела был написан Стефаном Бошенридтером (Stefan Boschenriedter) (`https://github.com/stefanbo92/A3C-Continuous`). Среда выглядит так:



Эпизод 1

Полный код в формате книги Jupyter с объяснениями доступен по адресу: **https://github.com/sudharsan13296/Hands-On-Reinforcement-Learning-With-Python/blob/master/10.%20Aysnchronous%20Advantage%20Actor%20Critic%20Network/10.5%20Drive%20up%20the%20Mountain%20Using%20A3C.ipynb**.

Начнем с импортирования необходимых библиотек:

```
import gym
import multiprocessing
import threading
import numpy as np
import os
import shutil
import matplotlib.pyplot as plt
import tensorflow as tf
```

Теперь инициализируем все параметры:

```
# Количество работников
no_of_workers = multiprocessing.cpu_count()

# Максимальное количество шагов на эпизод
no_of_ep_steps = 200

# Общее количество эпизодов
no_of_episodes = 2000

global_net_scope = 'Global_Net'

# Настройка частоты обновления глобальной сети
update_global = 10

# Коэффициент корректировки
gamma = 0.90

# Коэффициент энтропии
entropy_beta = 0.01

# Скорость обучения для актора
lr_a = 0.0001

# Скорость обучения для критика
lr_c = 0.001
# Переменная для прорисовки среды
render=False
```

```
# Каталог для хранения журналов
log_dir = 'logs'
```

Инициализируем среду `MountainCar`:

```
env = gym.make('MountainCarContinuous-v0')
env.reset()
```

Задаем количество состояний и действий, а также `action_bound`:

```
no_of_states = env.observation_space.shape[0]
no_of_actions = env.action_space.shape[0]
action_bound = [env.action_space.low,
env.action_space.high]
```

Сеть «актор-критик» определяется в классе `ActorCritic`. Как обычно, мы сначала разберем код каждой функции класса, а в конце рассмотрим весь код в целом. Код дополнен комментариями для лучшего понимания, а в конце будет приведен «чистый» код без комментариев.

```
class ActorCritic(object):
    def __init__(self, scope, sess,
globalAC=None):
        # Инициализировать сеанс и оптимизатор
RMSProp
        # для сетей актора и критика
        self.sess=sess
        self.actor_optimizer =
tf.train.RMSPropOptimizer(lr_a,
  name='RMSPropA')
        self.critic_optimizer =
tf.train.RMSPropOptimizer(lr_c,
  name='RMSPropC')

        # Для глобальной сети:
        if scope == global_net_scope:
            with tf.variable_scope(scope):
                # Инициализировать состояния,
построить сети
                # актора и критика
                self.s =
tf.placeholder(tf.float32, [None, no_of_states],
  'S')
                # Получить параметры сетей
актора и критика
                self.a_params, self.c_params =
self._build_net(scope)[-2:]
        # Для локальной сети:
        else:
            with tf.variable_scope(scope):
```

```
                    # Инициализировать состояние,
действие и целевое значение
                    # как v_target
                    self.s =
tf.placeholder(tf.float32, [None, no_of_states],
  'S')
                    self.a_his =
tf.placeholder(tf.float32, [None,
  no_of_actions], 'A')
                    self.v_target =
tf.placeholder(tf.float32, [None, 1],
  'Vtarget')
                    # Так как мы находимся в
непрерывном пространстве
                    # действий, вычисляем среднее
значение и дисперсию
                    # для выбора действия
                    mean, var, self.v,
self.a_params, self.c_params =
  self._build_net(scope)

                    # Затем вычисляется погрешность
td как разность
                    # v_target - v
                    td = tf.subtract(self.v_target,
self.v, name='TD_error')

                    # Минимизировать погрешность TD
                    with
tf.name_scope('critic_loss'):
                        self.critic_loss =
tf.reduce_mean(tf.square(td))
                    # Обновить среднее и дисперсию,
умножив среднее на границы
                    # и прибавить var 1e-4

                    with
tf.name_scope('wrap_action'):
                        mean, var = mean *
action_bound[1], var + 1e-4
                    # Сгенерировать распределение с
использованием
                    # обновленного среднего и
дисперсии
```

```
                    normal_dist =
tf.contrib.distributions.Normal(mean, var)
                    # Теперь вычисляем потери
актора.
                    # Вспомните, как выглядит
функция потерь.
                    with
tf.name_scope('actor_loss'):
                        # Вычислить первую
составляющую потерь - log(pi(s))
                        log_prob =
normal_dist.log_prob(self.a_his)
                        exp_v = log_prob * td
                        # Вычислить энтропию по
распределению действий
                        # для обеспечения более
полного исследования
                        entropy =
normal_dist.entropy()
                        # Итоговые потери можно
определить в виде
                        self.exp_v = exp_v +
entropy_beta * entropy
                        # Затем пытаемся
минимизировать потери
                        self.actor_loss =
tf.reduce_mean(-self.exp_v)
                    # Выбрать действие из
распределения
                    # и произвести отсечение по
границам
                    with
tf.name_scope('choose_action'):
                        self.A =
  tf.clip_by_value(tf.squeeze(normal_dist.sample(1
), axis=0),
  action_bound[0], action_bound[1])
                    # Вычислить градиенты для сетей
актора и критика
                    with
tf.name_scope('local_grad'):

                        self.a_grads =
tf.gradients(self.actor_loss,
```

```
  self.a_params)
                        self.c_grads =
tf.gradients(self.critic_loss,
  self.c_params)

                # Обновить веса глобальной сети
                with tf.name_scope('sync'):
                    # Извлечь веса глобальной сети в
локальные сети
                    with tf.name_scope('pull'):
                        self.pull_a_params_op =
[l_p.assign(g_p) for l_p, g_p
  in zip(self.a_params, globalAC.a_params)]
                        self.pull_c_params_op =
[l_p.assign(g_p) for l_p, g_p
  in zip(self.c_params, globalAC.c_params)]
                    # Занести локальные градиенты в
глобальную сеть
                    with tf.name_scope('push'):
                        self.update_a_op =
  self.actor_optimizer.apply_gradients(zip(self.a_
grads, globalAC.a_params))
                        self.update_c_op =
  self.critic_optimizer.apply_gradients(zip(self.c
_grads, globalAC.c_params))


    # Определить функцию _build_net для
построения
    # сетей актора и критика
    def _build_net(self, scope):
    # Инициализировать веса
        w_init =
tf.random_normal_initializer(0., .1)
        with tf.variable_scope('actor'):
            l_a = tf.layers.dense(self.s, 200,
tf.nn.relu6,
  kernel_initializer=w_init, name='la')
            mean = tf.layers.dense(l_a,
no_of_actions,
  tf.nn.tanh,kernel_initializer=w_init,
name='mean')
            var = tf.layers.dense(l_a,
no_of_actions, tf.nn.softplus,
```

```
  kernel_initializer=w_init, name='var')
            with tf.variable_scope('critic'):
                l_c = tf.layers.dense(self.s, 100,
tf.nn.relu6,
  kernel_initializer=w_init, name='lc')
                v = tf.layers.dense(l_c, 1,
kernel_initializer=w_init,
  name='v')
            a_params =
tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES
,
  scope=scope + '/actor')
            c_params =
tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES
,
  scope=scope + '/critic')
            return mean, var, v, a_params, c_params
        # Обновить локальные градиенты в глобальной
сети
        def update_global(self, feed_dict):
            self.sess.run([self.update_a_op,
self.update_c_op], feed_dict)
        # Получить глобальные параметры в локальных
сетях
        def pull_global(self):
            self.sess.run([self.pull_a_params_op,
self.pull_c_params_op])
        # Выбрать действие
        def choose_action(self, s):
            s = s[np.newaxis, :]
            return self.sess.run(self.A, {self.s:
s})[0]
```

Теперь инициализируем класс `Worker`:

```
class Worker(object):
    def __init__(self, name, globalAC, sess):
        # Инициализировать среду для каждого
работника
        self.env =
gym.make('MountainCarContinuous-v0').unwrapped
        self.name = name
        # Создать агента ActorCritic для каждого
работника
        self.AC = ActorCritic(name, sess,
globalAC)
```

```
            self.sess=sess
        def work(self):
            global global_rewards, global_episodes
            total_step = 1

            # Для хранения состояния, действия, награды
            buffer_s, buffer_a, buffer_r = [], [], []
            # Выполнять в цикле, если координатор
активен, а глобальный
            # эпизод не достиг максимума
            while not coord.should_stop() and
global_episodes < no_of_episodes:
                # Инициализировать среду выполнением
сброса
                s = self.env.reset()
                # Сохранить эпизодическую награду
                ep_r = 0
                for ep_t in range(no_of_ep_steps):
                    # Выполнить прорисовку только для
работника 1
                    if self.name == 'W_0' and render:
                        self.env.render()
                    # Выбрать действие на основании
политики
                    a = self.AC.choose_action(s)

                    # Выполнить действие (a),
получить награду (r)
                    # и перейти к следующему
состоянию (s_)
                    s_, r, done, info =
self.env.step(a)
                    # Присвоить done значение true
при достижении
                    # максимального шага для эпизода
                    done = True if ep_t ==
no_of_ep_steps - 1 else False
                    ep_r += r
                    # Сохранить состояние, действие
и награду в буфере
                    buffer_s.append(s)
                    buffer_a.append(a)
                    # Нормализовать награду
                    buffer_r.append((r+8)/8)
```

```
                    # Глобальная сеть обновляется
после конкретного
                    # временного шага
                    if total_step % update_global ==
0 or done:
                        if done:
                            v_s_ = 0
                        else:
                            v_s_ =
self.sess.run(self.AC.v, {self.AC.s:
  s_[np.newaxis, :]})[0, 0]
                        # Буфер для цели v
                        buffer_v_target = []
                        for r in buffer_r[::-1]:
                            v_s_ = r + gamma * v_s_
                            buffer_v_target.append(v
_s_)
                        buffer_v_target.reverse()
                        buffer_s, buffer_a,
buffer_v_target =
  np.vstack(buffer_s), np.vstack(buffer_a),
np.vstack(buffer_v_target)
                        feed_dict = {
                            self.AC.s:
buffer_s,
                            self.AC.a_his:
buffer_a,
                            self.AC.v_targe
t: buffer_v_target,
                            }
                        # Обновить глобальную сеть
                        self.AC.update_global(feed_d
ict)
                        buffer_s, buffer_a, buffer_r
= [], [], []
                        # Получить глобальные
параметры
                        # для локального ActorCritic
                        self.AC.pull_global()
                    s = s_
                    total_step += 1
                    if done:
                        if len(global_rewards) < 5:
```

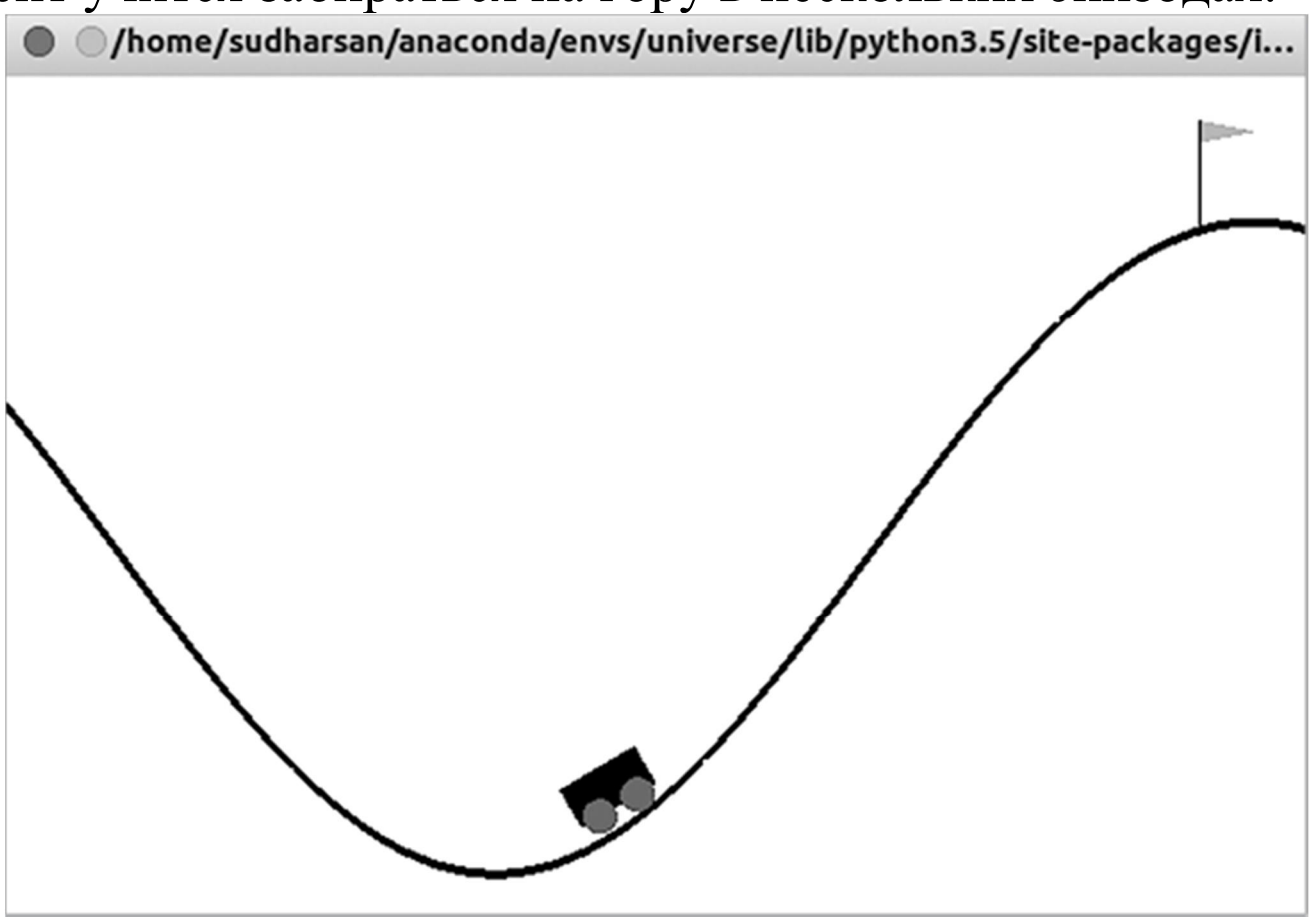
```
                    global_rewards.append(ep
_r)
                else:
                    global_rewards.append(ep
_r)
                global_rewards[-1]
=(np.mean(global_rewards[-5:]))
                global_episodes += 1
                break
```

Откроем сеанс TensorFlow и запустим модель:

```
# Создать список для хранения глобальных наград
и эпизодов
global_rewards = []
global_episodes = 0

# Начать сеанс tensorflow
sess = tf.Session()

with tf.device("/cpu:0"):
# Создать экземпляр класса ActorCritic
    global_ac =
ActorCritic(global_net_scope,sess)
    workers = []
    # Перебрать работников в цикле
    for i in range(no_of_workers):
        i_name = 'W_%i' % i
        workers.append(Worker(i_name,
global_ac,sess))

coord = tf.train.Coordinator()
sess.run(tf.global_variables_initializer())

# Сохранить данные в журнале для визуального
представления
# графа в tensorboard

if os.path.exists(log_dir):
    shutil.rmtree(log_dir)

tf.summary.FileWriter(log_dir, sess.graph)

worker_threads = []

# Запустить работников
```

```
for worker in workers:

    job = lambda: worker.work()
    t = threading.Thread(target=job)
    t.start()
    worker_threads.append(t)
coord.join(worker_threads)
```

Вывод показан на рисунке ниже. Запустив программу, вы увидите, как агент учится забираться на гору в нескольких эпизодах:

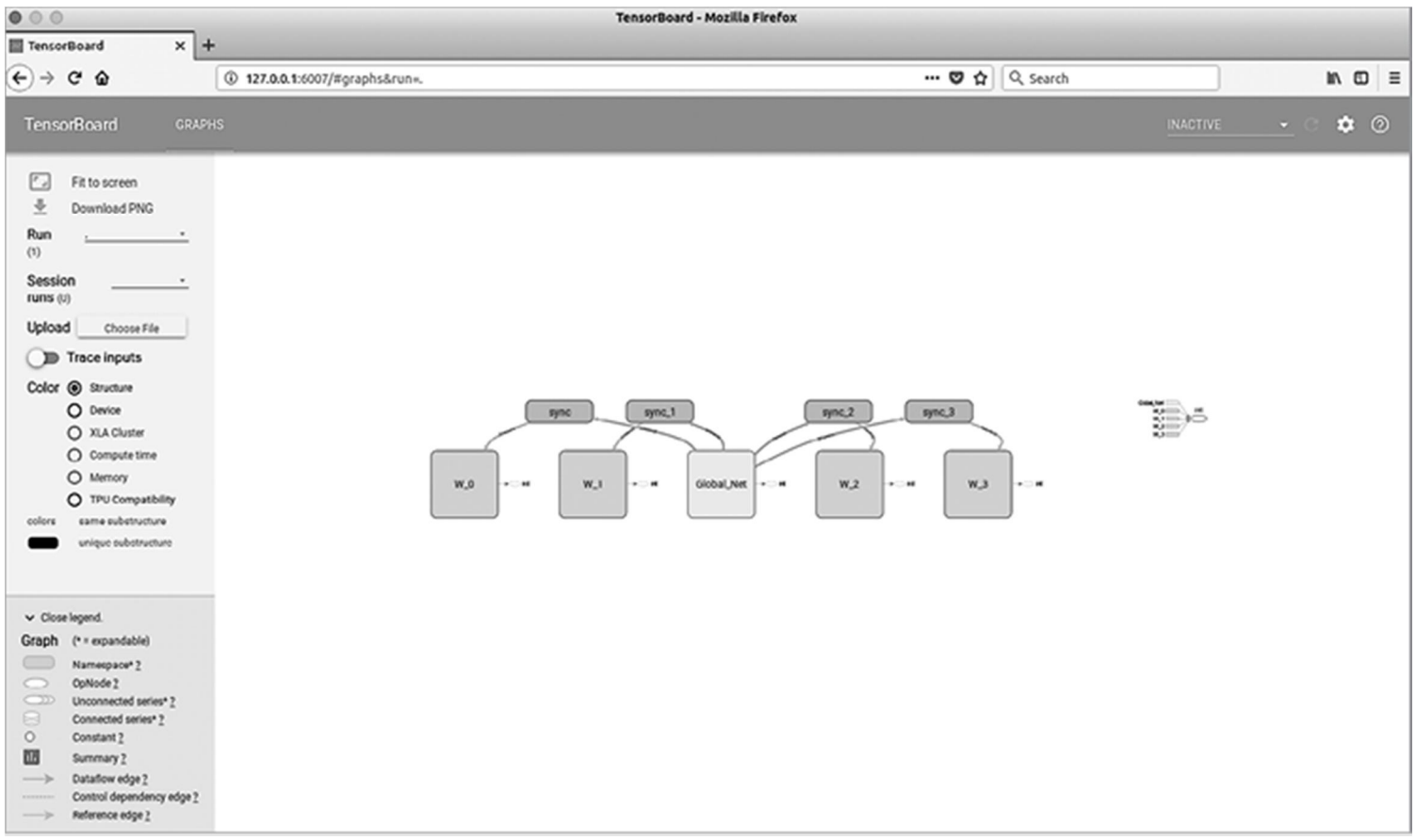


Визуализация в TensorBoard

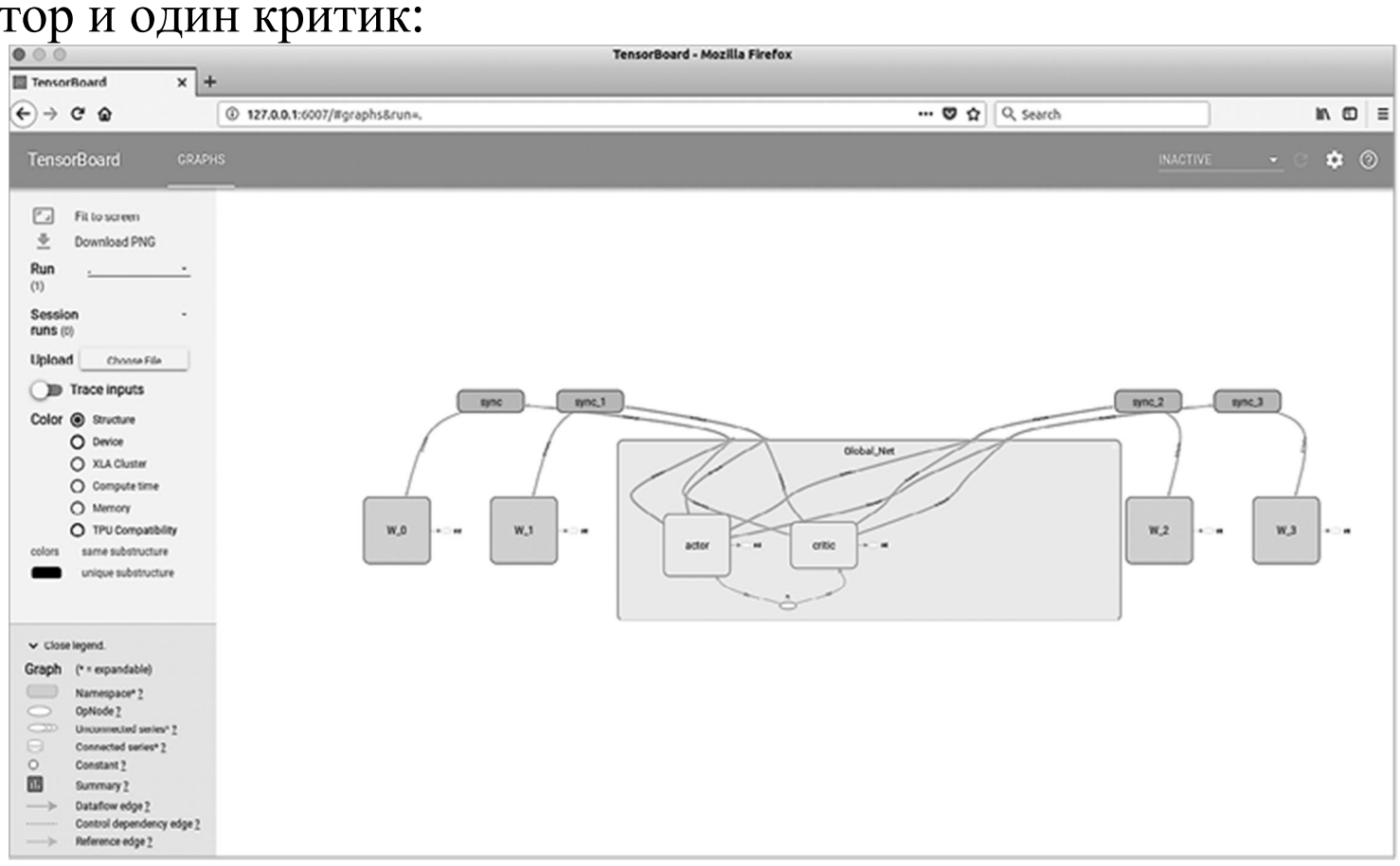
Построим наглядное представление нашей сети в TensorBoard. Чтобы запустить TensorBoard, откройте терминал и введите следующую команду:

```
tensorboard --logdir=logs --port=6007 --host=127.0.0.1
```

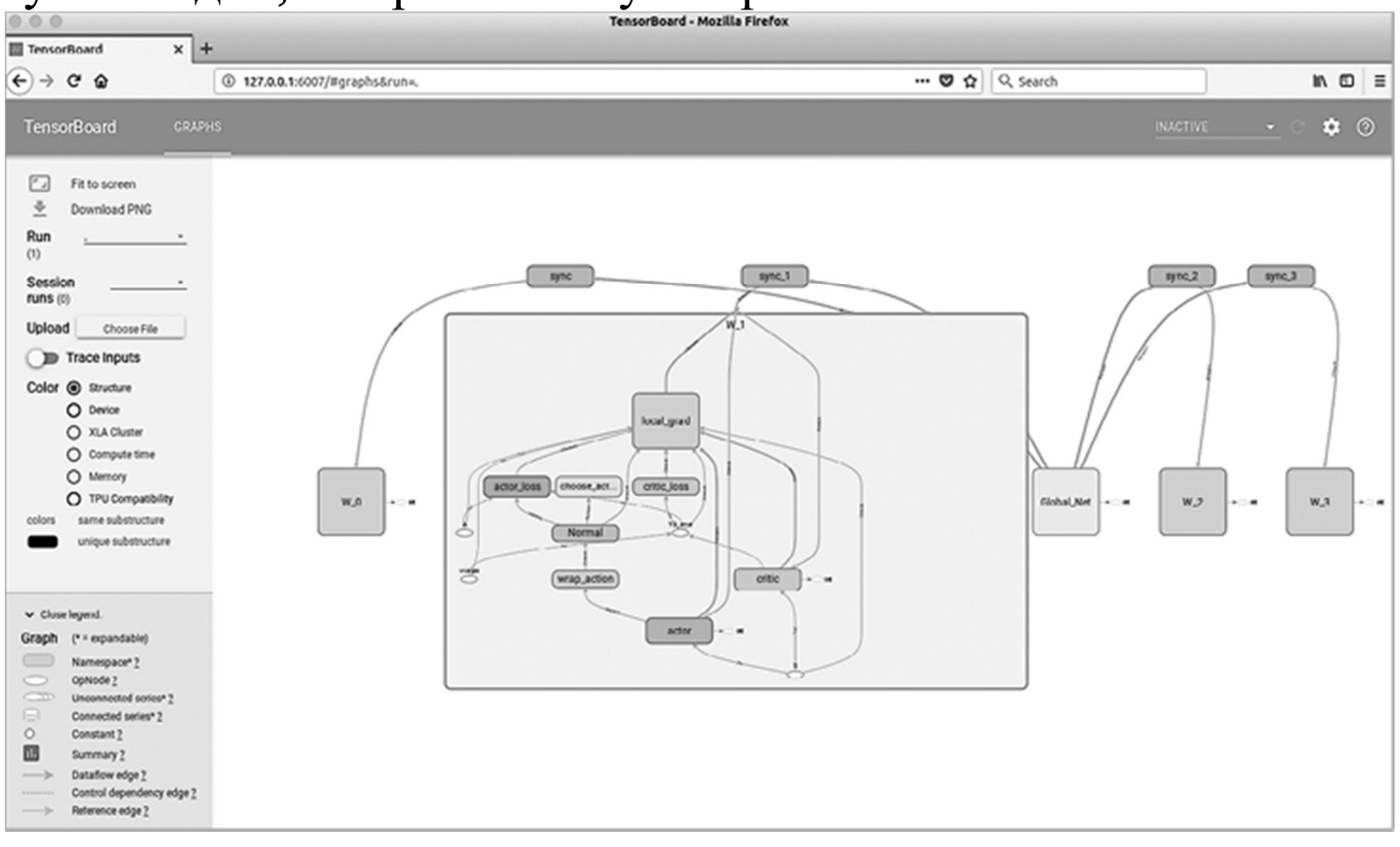
Так выглядит наша сеть A3C. Она состоит из одной глобальной сети и четырех работников:



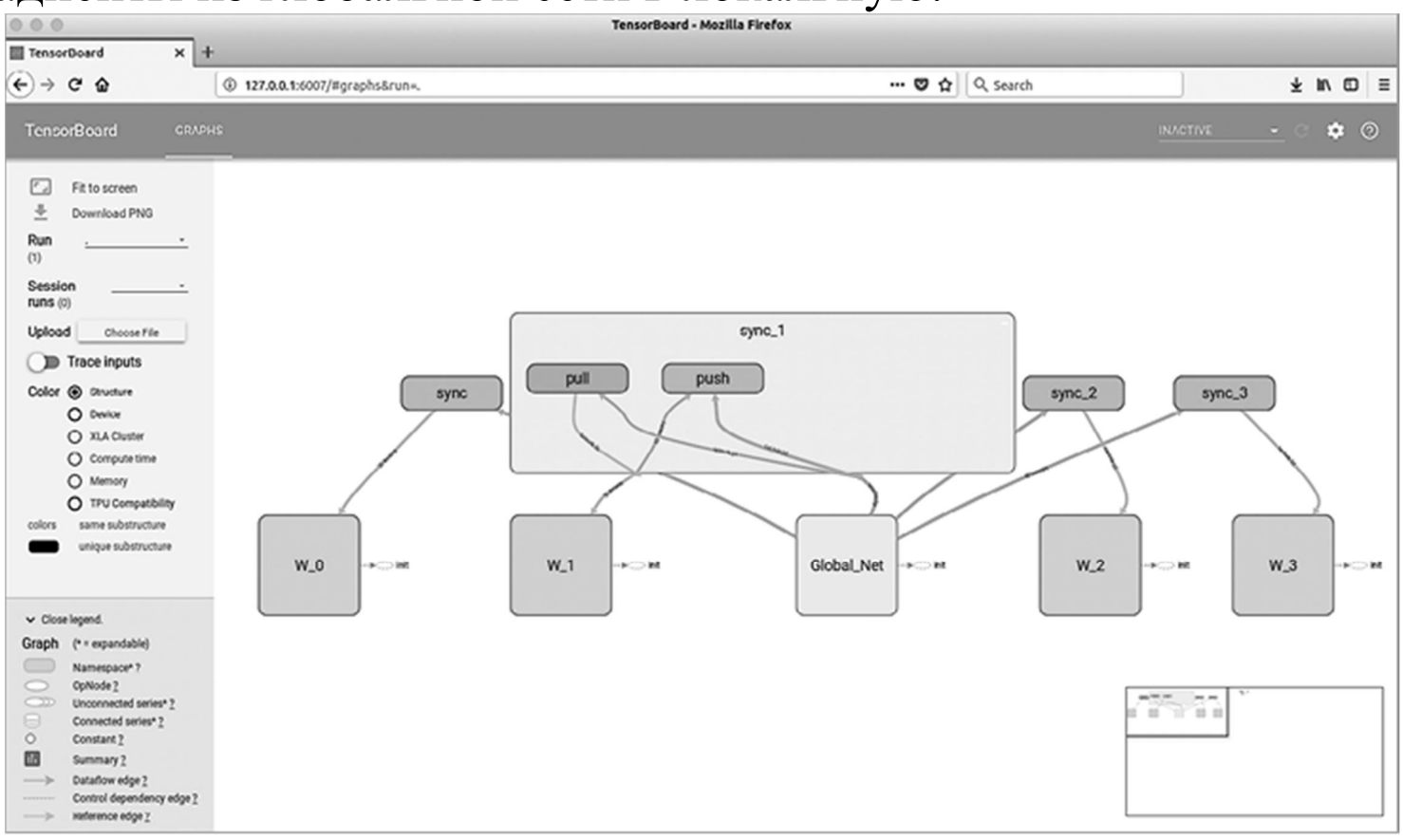
Раскроем глобальную сеть; вы видите, что в ней задействован один актор и один критик:



А что происходит с работниками? Раскроем сеть работника. Из рисунка видно, как работают узлы работников:



Что за синхроузел (sync node)? Что он делает? Этот узел передает локальные градиенты из локальной сети в глобальную и выводит градиенты из глобальной сети в локальную:



Итоги

В этой главе мы рассмотрели, как работают сети A3C. «Асинхронность» означает, что несколько агентов работают независимо друг от друга, взаимодействуя с разными копиями среды. «Преимущество» указывает на использование функции преимущества — разности между *Q*-функцией и функцией ценности. Наконец, «актор-критик» — это алгоритм с двумя типами сетей, где сеть актора отвечает за генерирование политики, а сеть критика оценивает политику, сгенерированную сетью актора. Были рассмотрены примеры использования A3C, включая пример подъема машины на гору с использованием этого алгоритма.

В **главе 11** будут представлены методы градиента политики, которые оптимизируют политику напрямую, не требуя наличия *Q*-функции.

Вопросы

1. Что такое A3C?
2. Что означают три «A» в названии?
3. Назовите преимущество A3C перед DQN.
4. Чем отличаются глобальные сети от работников?
5. Для чего в функцию потерь включается энтропия?
6. Объясните принцип работы A3C.

Дополнительные источники

Статья об A3C: `https://arxiv.org/pdf/1602.01783.pdf.`

A3C с расширенным зрением: `http://cs231n.stanford.edu/reports/2017/pdfs/617.pdf.`

11. Градиенты политик и оптимизация

В трех последних главах мы исследовали различные алгоритмы глубокого обучения с подкреплением, такие как **DQN** (**deep Q network**), **DRQN** (**deep recurrent Q network**) и **A3C** (**asynchronous advantage actor critic**). Целью в них является нахождение оптимальной политики для максимизации наград. И мы использовали *Q*-функцию, определяющую, какое действие лучше всего выполнить в состоянии. Как вы думаете, возможно ли найти оптимальную политику без использования *Q*-функции? Да. Методы градиентов политик позволяют это сделать.

В этой главе будут рассмотрены градиенты политик и их методы, такие как глубокие детерминированные градиенты политик и современные методы оптимизации политик: оптимизация политики доверительной области и оптимизация ближайшей политики.

В этой главе рассматриваются следующие темы:

- Градиенты политик.
- Посадка на Луну с использованием градиентов политик.
- Глубокие детерминированные градиенты политик.
- Раскачивание маятника с использованием **глубокого детерминированного градиента политики** (**DDPG**).
- Оптимизация политики доверительной области.
- Оптимизация ближайшей политики.

Градиент политики

Градиент политики — один из замечательных алгоритмов из области **обучения с подкреплением** (**RL**), обеспечивающих прямую оптимизацию политики, параметризованной по некоторому параметру. До сих пор для поиска оптимальной политики использовалась *Q*-функция. Теперь посмотрим, как найти оптимальную политику без *Q*-функции. Сначала определим функцию политики в форме $\pi(a \mid s)$, то есть как вероятность использования действия a в состоянии s. Политика параметризуется по параметру θ в форме $\pi(a \mid s; \theta)$, что позволяет выбрать лучшее действие в состоянии.

Метод градиента политики обладает рядом преимуществ, к тому же он способен обрабатывать непрерывное пространство с бесконечным количеством действий и состояний. Допустим, вы строите машину с автоматизированным управлением. Она должна ехать, не сталкиваясь с другими машинами, чтобы получать положительные награды, и для этого определенным образом обновляются параметры модели. В этом заключается базовая идея градиента политики: параметр модели обновляется с расчетом на максимизацию награды. Рассмотрим этот процесс более подробно.

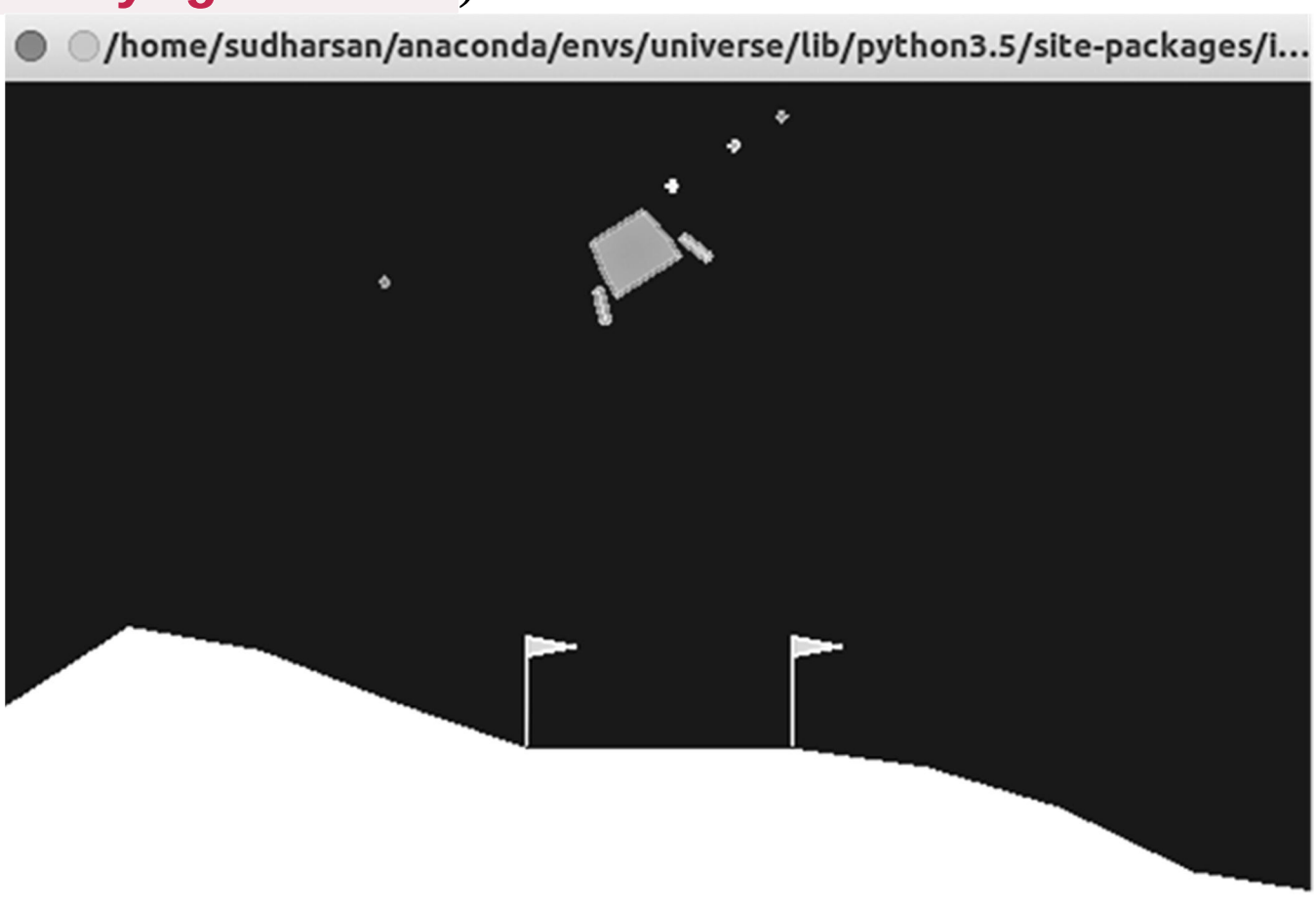
Нейросеть используется для поиска оптимальной политики; эта сеть называется *сетью политики*. Входом для сети политики будет состояние,

а выходом — вероятность каждого действия в этом состоянии. При наличии таких вероятностей мы можем выбрать действие из распределения и выполнить его в состоянии. Однако выбранное действие может и не быть лучшим. Это нормально — мы выполняем действие и сохраняем награду. Аналогичным образом выполняются другие действия в каждом состоянии. Награды становятся обучающими данными. Затем выполняется градиентный спуск и обновление градиентов таким образом, чтобы действия, обеспечивающие высокую награду в состоянии, имели высокую вероятность, а действия с низкой наградой — низкую. Как выглядит функция потерь? Мы используем softmax-потери перекрестной энтропии, а затем умножаем потери на величину награды.

Посадка на Луну с градиентами политик

Допустим, агент управляет посадочным модулем, а его цель — правильная посадка на площадку. Если агент сажает модуль в стороне от площадки, он теряет награду, а эпизод завершается при крушении или переходе агента в неподвижное состояние. В среде доступны четыре действия: не делать ничего, запустить левый двигатель ориентации, запустить главный двигатель и запустить правый двигатель ориентации.

Теперь посмотрим, как обучить агентов для выполнения правильной посадки с градиентами политик. Код, использованный в этом разделе, был написан Гэбриелом (Gabriel) (`https://github.com/gabrielgarza/openai-gym-policy-gradient`):



Сначала импортируются необходимые библиотеки:

```
import tensorflow as tf
import numpy as np
from tensorflow.python.framework import ops
import gym
import numpy as np
import time
```

Затем определяется класс `PolicyGradient`, реализующий алгоритм градиента политики. Для начала рассмотрим каждую функцию по отдельности. Полный код в формате книги Jupyter с объяснениями доступен по адресу **`https://github.com/sudharsan13296/Hands-On-Reinforcement-Learning-With-Python/blob/master/11.%20Policy%20Gradients%20and%20Optimization/11.2%20Lunar%20Lander%20Using%20Policy%20Gradients.ipynb`**:

```
  class PolicyGradient:
      # В методе __init__ инициализируются все переменные
      def __init__(self, n_x,n_y,learning_rate=0.01, reward_decay=0.95):
          # Количество состояний в среде
          self.n_x = n_x
          # Количество действий в среде
          self.n_y = n_y
          # Скорость обучения сети
          self.lr = learning_rate
          # Поправочный коэффициент
          self.gamma = reward_decay
          # Инициализировать списки для хранения наблюдений,
          # действий и наград
          self.episode_observations, self.episode_actions,
  self.episode_rewards = [], [], []
          # Определение функции build_network для построения
          # нейросети
          self.build_network()
          # Сохранить стоимость, то есть потери
          self.cost_history = []
          # Инициализировать сеанс tensorflow
          self.sess = tf.Session()
          self.sess.run(tf.global_variables_initializer())
```

Определяется функция `store_transition` для хранения переходов, то есть состояния, действия и награды. Эта информация используется для обучения сети:

```
    def store_transition(self, s, a, r):
        self.episode_observations.append(s)
        self.episode_rewards.append(r)
```

```
        # Сохранить действия в виде списка
массивов
        action = np.zeros(self.n_y)
        action[a] = 1
        self.episode_actions.append(action)
```

Определяется функция `choose_action` для выбора действия по заданному состоянию:

```
    def choose_action(self, observation):

        # Изменить размеры observation до
(num_features, 1)
        observation = observation[:, np.newaxis]
        # Запустить прямое распространение
        # для получения softmax-вероятностей
        prob_weights =
self.sess.run(self.outputs_softmax, feed_dict =
  {self.X: observation})

        # Выбрать действие с использованием
смещенной выборки,
        # которая возвращает индекс действия
        action =
np.random.choice(range(len(prob_weights.ravel())),
  p=prob_weights.ravel())
        return action
```

Определяется функция `build_network` для построения нейросети:

```
    def build_network(self):
        # Заместители для ввода X и для вывода Y
        self.X = tf.placeholder(tf.float32,
shape=(self.n_x, None),
  name="X")
        self.Y = tf.placeholder(tf.float32,
shape=(self.n_y, None),
  name="Y")
        # Заместитель для награды
        self.discounted_episode_rewards_norm =
tf.placeholder(tf.float32,
  [None, ], name="actions_value")

        # Строится 3-уровневая нейросеть с 2
скрытыми уровнями
        # и 1 выходным уровнем
        # Количество нейронов в скрытом уровне
```

```
            units_layer_1 = 10
            units_layer_2 = 10
            # Количество нейронов в выходном уровне
            units_output_layer = self.n_y
            # Инициализировать веса и смещение с
использованием
            # метода
tf.contrib.layers.xavier_initializer библиотеки
tensorflow
            W1 = tf.get_variable("W1",
[units_layer_1, self.n_x], initializer =
  tf.contrib.layers.xavier_initializer(seed=1))
            b1 = tf.get_variable("b1",
[units_layer_1, 1], initializer =
  tf.contrib.layers.xavier_initializer(seed=1))
            W2 = tf.get_variable("W2",
[units_layer_2, units_layer_1],
  initializer =
tf.contrib.layers.xavier_initializer(seed=1))
            b2 = tf.get_variable("b2",
[units_layer_2, 1], initializer =
  tf.contrib.layers.xavier_initializer(seed=1))
            W3 = tf.get_variable("W3", [self.n_y,
units_layer_2], initializer =
  tf.contrib.layers.xavier_initializer(seed=1))
            b3 = tf.get_variable("b3", [self.n_y,
1], initializer =
  tf.contrib.layers.xavier_initializer(seed=1))

          # Затем выполняется прямое распространение

          Z1 = tf.add(tf.matmul(W1,self.X), b1)
          A1 = tf.nn.relu(Z1)
          Z2 = tf.add(tf.matmul(W2, A1), b2)
          A2 = tf.nn.relu(Z2)
          Z3 = tf.add(tf.matmul(W3, A2), b3)
          A3 = tf.nn.softmax(Z3)

            # Применить softmax-функцию активации на
выходном уровне
            logits = tf.transpose(Z3)
            labels = tf.transpose(self.Y)
            self.outputs_softmax =
tf.nn.softmax(logits, name='A3')
```

```
        # Функция потерь определяется как потери перекрестной энтропии
        neg_log_prob = tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=labels)
        # Награда исходя из потерь
        loss = tf.reduce_mean(neg_log_prob * self.discounted_episode_rewards_norm)

        # Использовать оптимизатор adam для минимизации потерь
        self.train_op = tf.train.AdamOptimizer(self.lr).minimize(loss)
```

Определяется функция `discount_and_norm_rewards`, которая вычисляет скорректированную нормализованную награду:

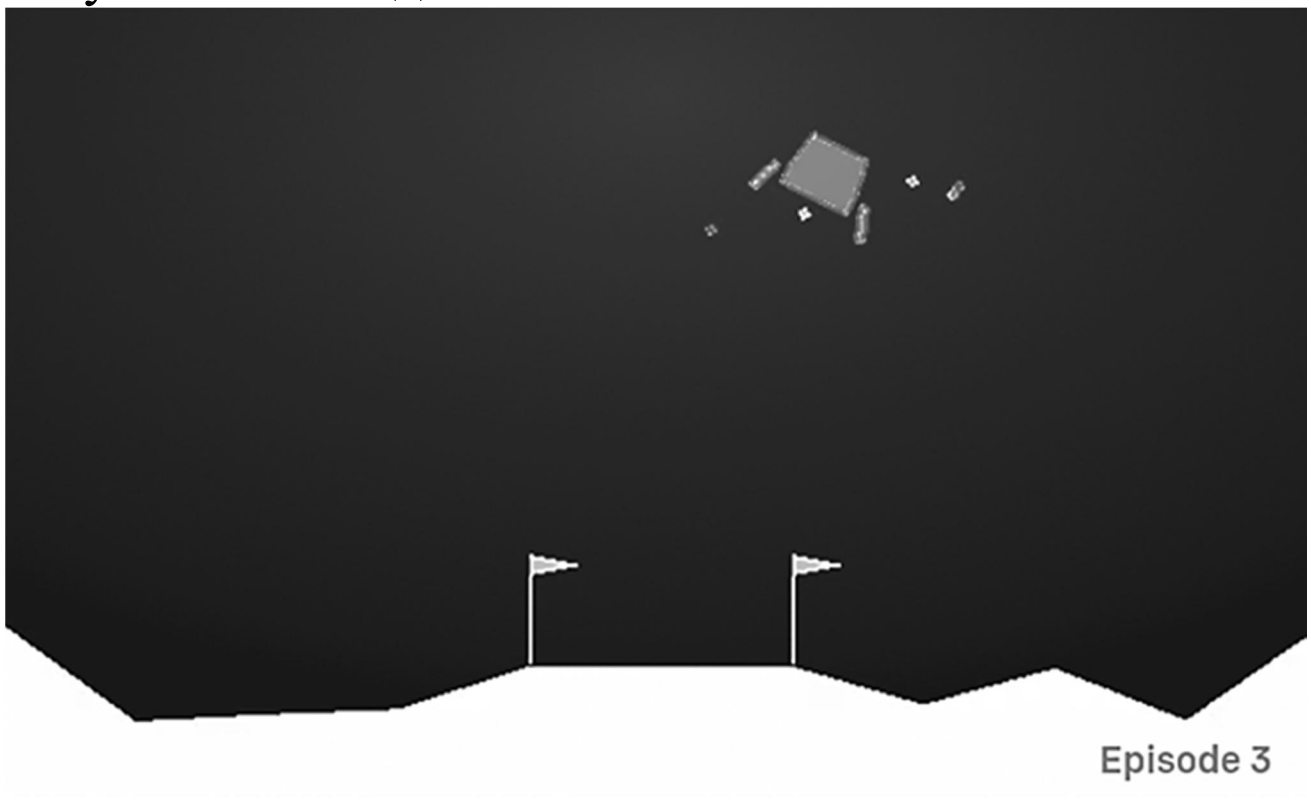
```
    def discount_and_norm_rewards(self):
        discounted_episode_rewards = np.zeros_like(self.episode_rewards)
        cumulative = 0
        for t in reversed(range(len(self.episode_rewards))):
            cumulative = cumulative * self.gamma + self.episode_rewards[t]
            discounted_episode_rewards[t] = cumulative

        discounted_episode_rewards -= np.mean(discounted_episode_rewards)
        discounted_episode_rewards /= np.std(discounted_episode_rewards)
        return discounted_episode_rewards
```

А теперь можно заняться непосредственным обучением:

```
    def learn(self):
        # Скорректировать и нормализовать награду эпизода
        discounted_episode_rewards_norm = self.discount_and_norm_rewards()

        # Провести обучение сети
        self.sess.run(self.train_op, feed_dict={
            self.X: np.vstack(self.episode_observations).T,
```

```
                self.Y:
np.vstack(np.array(self.episode_actions)).T,
                self.discounted_episode_rewards_nor
m:
  discounted_episode_rewards_norm,
          })
          # Сбросить данные эпизода
          self.episode_observations,
self.episode_actions,
  self.episode_rewards = [], [], []

          return discounted_episode_rewards_norm
```

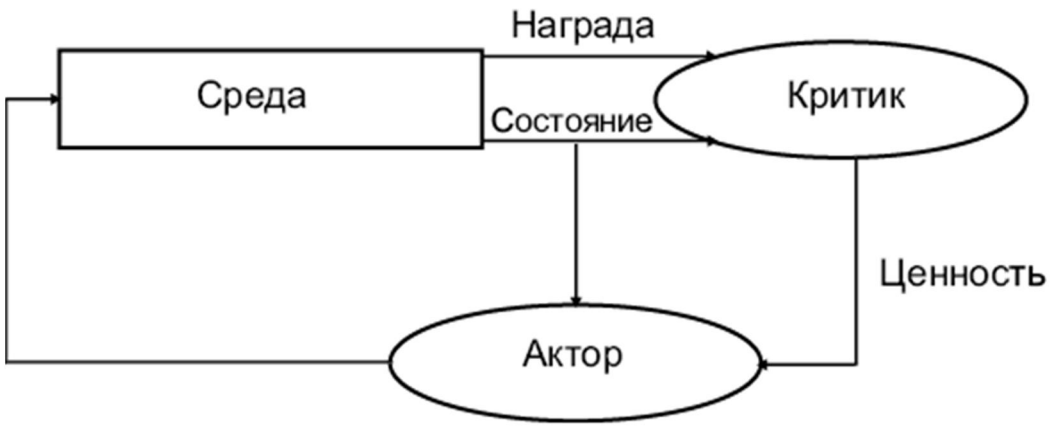
Результат выглядит так:



Глубокий детерминированный градиент политики

В **главе 8** мы рассмотрели сети DQN и применили их для игр Atari. Это были дискретные среды с конечным набором действий. Но представьте непрерывное пространство — например, при обучении робота ходьбе, — в таких средах *Q*-обучение не подойдет, потому что поиск жадной стратегии потребует большого объема оптимизаций на каждом шаге. Даже если как-то привести непрерывную среду к дискретному виду, мы можем потерять важные признаки и получить огромный набор пространств действий, который усложнит схождение.

По этой причине мы используем архитектуру алгоритма «актор-критик» с двумя сетями — актора и критика. Она объединяет градиент политики и функции ценности состояния/действия. Сеть **актора**должна определить лучшие действия в состоянии посредством настройки параметра θ, а сеть **критика** — оценить действие, сгенерированное актором, вычислив погрешность TD. Таким образом, мы применяем градиент политики к сети **актора** для выбора действий, а сеть **критика**оценивает действие, произведенное сетью актора, с использованием погрешности TD. Архитектура «актор-критик» изображена на рисунке:



По аналогии с DQN здесь используется буфер, в котором сети актора и критика обучены на мини-выборке опыта. Также здесь есть отдельная целевая сеть для вычисления потерь.

Например, в игре *Pong* мы имеем разные признаки с разными масштабами: позиция, скорость и т.д. Чтобы добиться единого масштаба, применяется метод, называемый *пакетной нормализацией*. Все признаки нормализуются с единичным средним и дисперсией. Как исследовать новые действия? В непрерывной среде количество действий равно n. Мы добавляем шум N к действиям, производимым сетью актора. Для генерирования этого шума используется процесс, называемый *случайным процессом Орнштейна—Уленбека*.

Теперь подробно рассмотрим алгоритм DDPG.

Допустим, имеются две сети: актора и критика. Сеть актора $\mu(s; \theta^{\mu})$ получает состояние на входе и выдает действие, у которого θ^{μ} — веса сети актора. Сеть критика с представлением $Q(s, a; \theta^{Q})$ получает на входе состояние и действие и возвращает значение Q, где θ^{Q} — веса сети критика.

Аналогичным образом определяется целевая сеть для актора и критика в форме $\mu(s; \theta^{\mu})$ и $Q(s, a; \theta^{Q})$ соответственно, где $\theta^{\mu'}$ и $\theta^{Q'}$ — веса целевой сети.

Веса сети актора обновляются градиентами политики, а веса сети критика — градиентами, вычисленными по погрешности TD.

Сначала выбирается действие, в которое добавляется шум исследования N: $\mu(s; \theta^{\mu}) + N$. Мы выполняем это действие в состоянии s, получаем награду r и переходим в новое состояние s'. Информация перехода сохраняется в буфере воспроизведения опыта.

После нескольких итераций мы выполняем выборку переходов из буфера воспроизведения и обучаем сеть, после чего вычисляем целевое значение Q: $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$. Погрешность TD вычисляется в виде:

$$L = \frac{1}{M}\sum_i (y_i - Q(s_i, a_i|\theta^{Q})^2),$$

где M — размер выборки из буфера воспроизведения опыта, используемой для обучения. Веса сети критика обновляются градиентами, вычисленными по этим потерям L.

Аналогичным образом веса сети политики обновляются с использованием градиента политики. После этого веса сетей актора и критика обновляются в целевой сети. Веса целевых сетей обновляются

медленно, что способствует повышению стабильности; этот механизм называется *мягкой заменой*(soft replacement):

$\theta' < -\tau\theta + (1 - \tau)\,\theta'$.

Раскачивание маятника

Имеется маятник со случайной исходной позицией. Цель агента — раскачать маятник, чтобы он оказался в вертикальном состоянии. Посмотрим, как сделать это с применением DDPG. Автором кода, использованного в этом разделе, является wshuail (**https://github.com/wangshuailong/reinforcement_learning_with_Tensorflow/tree/master/DDPG**).

Сначала импортируются необходимые библиотеки:

```
import tensorflow as tf
import numpy as np
import gym
```

Затем определяются гиперпараметры:

```
# Количество шагов в каждом эпизоде
epsiode_steps = 500

# Скорость обучения для актора
lr_a = 0.001

# Скорость обучения для критика
lr_c = 0.002

# Поправочный коэффициент
gamma = 0.9

# Мягкая замена
alpha = 0.01

# Размер буфера воспроизведения
memory = 10000

# Размер пакета для обучения
batch_size = 32
render = False
```

Алгоритм DDPG реализуется в классе `DDPG`. Разобьем класс на функции и рассмотрим их по отдельности. Начнем с инициализаций:

```
class DDPG(object):
    def __init__(self, no_of_actions, no_of_states, a_bound,):
        # Инициализировать память по количеству действий,
```

```
            # количеству состояний и определенным
размерам памяти
            self.memory = np.zeros((memory,
no_of_states * 2 + no_of_actions +
  1), dtype=np.float32)
            # Инициализировать указатель на буфер
опыта
            self.pointer = 0
            # Инициализировать сеанс tensorflow
            self.sess = tf.Session()
            # Инициализировать дисперсию ОУ-процесса
для исследования политик
            self.noise_variance = 3.0
            self.no_of_actions, self.no_of_states,
self.a_bound =
  no_of_actions, no_of_states, a_bound,
            # Заместитель для текущего состояния,
следующего состояния
            # и наград
            self.state = tf.placeholder(tf.float32,
[None, no_of_states], 's')
            self.next_state =
tf.placeholder(tf.float32, [None, no_of_states],
  's_')
            self.reward = tf.placeholder(tf.float32,
[None, 1], 'r')
            # Построить сеть актора с разной
первичной (eval) и целевой сетью
            with tf.variable_scope('Actor'):
                self.a =
self.build_actor_network(self.state, scope='eval',
  trainable=True)
                a_ =
self.build_actor_network(self.next_state,
scope='target',
  trainable=False)
            # Построить сеть критика с отдельной
первичной (eval)
            # и целевой сетью
            with tf.variable_scope('Critic'):
                q =
self.build_crtic_network(self.state, self.a,
scope='eval',
  trainable=True)
```

```
                q_ =
self.build_crtic_network(self.next_state, a_,
  scope='target', trainable=False)

            # Инициализировать параметры сети
            self.ae_params =
tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES,
  scope='Actor/eval')
            self.at_params =
tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES,
  scope='Actor/target')
            self.ce_params =
tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES,
  scope='Critic/eval')
            self.ct_params =
tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES,
  scope='Critic/target')

            # Обновить целевое значение
            self.soft_replace = [[tf.assign(at, (1-
alpha)*at+alpha*ae),
  tf.assign(ct, (1-alpha)*ct+alpha*ce)]
                for at, ae, ct, ce in
zip(self.at_params, self.ae_params,
  self.ct_params, self.ce_params)]
            # Вычислить целевое значение Q;
            # мы знаем, что Q(s, a) = reward + gamma
* Q'(s', a')
            q_target = self.reward + gamma * q_
            # Вычислить погрешность TD,
            # то есть фактическое прогнозируемое
значение
            td_error =
tf.losses.mean_squared_error(labels=(self.reward +
gamma
  * q_), predictions=q)
            # Обучить сеть критика с применением
оптимизатора Adam
            self.ctrain =
tf.train.AdamOptimizer(lr_c).minimize(td_error,
  name="adam-ink", var_list = self.ce_params)
            # Вычислить потери в сети актора
            a_loss = - tf.reduce_mean(q)
```

```
        # Обучить сеть актора с применением оптимизатора Adam
        # для минимизации потерь
        self.atrain = tf.train.AdamOptimizer(lr_a).minimize(a_loss,
  var_list=self.ae_params)
        # Инициализировать средство записи для визуализации сети
        # в tensorboard
        tf.summary.FileWriter("logs", self.sess.graph)
        # Инициализировать все переменные
        self.sess.run(tf.global_variables_initializer())
```

Как выбирается действие в DDPG? Для этого в пространство действий добавляется шум. Для генерирования шума используется случайный процесс Орнштейна — Уленбека:

```
  def choose_action(self, s):
        a = self.sess.run(self.a, {self.state: s[np.newaxis, :]})[0]
        a = np.clip(np.random.normal(a, self.noise_variance), -2, 2)
        return a
```

Затем определяется функция `learn`, где происходит фактическое обучение. Здесь из буфера опыта выбирается пакет состояний, действий, наград и следующего состояния. Обучение сетей актора и критика происходит так:

```
    def learn(self):
        # Мягкая замена
        self.sess.run(self.soft_replace)

        indices = np.random.choice(memory, size=batch_size)
        batch_transition = self.memory[indices, :]
        batch_states = batch_transition[:, :self.no_of_states]
        batch_actions = batch_transition[:, self.no_of_states:
  self.no_of_states + self.no_of_actions]
        batch_rewards = batch_transition[:, -self.no_of_states - 1: -
  self.no_of_states]
```

```
            batch_next_state = batch_transition[:, -
self.no_of_states:]

            self.sess.run(self.atrain, {self.state:
batch_states})
            self.sess.run(self.ctrain, {self.state:
batch_states, self.a:
  batch_actions, self.reward: batch_rewards,
self.next_state:
  batch_next_state})
```

Определим функцию `store_transition`, которая сохраняет всю информацию в буфере и выполняет обучение:

```
    def store_transition(self, s, a, r, s_):
            trans = np.hstack((s, a,[r],s_))
            index = self.pointer % memory
            self.memory[index, :] = trans
            self.pointer += 1

            if self.pointer > memory:
                self.noise_variance *= 0.99995
                self.learn()
```

Определим функцию `build_actor_network` для построения сети актора:

```
    def build_actor_network(self, s, scope,
trainable):
            # DPG актора
            with tf.variable_scope(scope):
                l1 = tf.layers.dense(s, 30,
activation = tf.nn.tanh, name =
  'l1', trainable = trainable)
                a = tf.layers.dense(l1,
self.no_of_actions, activation =
  tf.nn.tanh, name = 'a', trainable = trainable)
                return tf.multiply(a, self.a_bound,
name = "scaled_a")
```

Определим функцию `build_crtic_network`:

```
    def build_crtic_network(self, s, a, scope,
trainable):
            # Q-обучение критика
            with tf.variable_scope(scope):
                n_l1 = 30
                w1_s = tf.get_variable('w1_s',
[self.no_of_states, n_l1],
  trainable = trainable)
```

```
            w1_a = tf.get_variable('w1_a',
[self.no_of_actions, n_l1],
  trainable = trainable)
            b1 = tf.get_variable('b1', [1,
n_l1], trainable = trainable)
            net = tf.nn.tanh( tf.matmul(s, w1_s)
+ tf.matmul(a, w1_a) + b1
  )

            q = tf.layers.dense(net, 1,
trainable = trainable)
            return q
```

Инициализируем среду Gym функцией `make`:

```
env = gym.make("Pendulum-v0")
env = env.unwrapped
env.seed(1)
```

Получим количество состояний:

```
no_of_states = env.observation_space.shape[0]
```

…количество действий:

```
no_of_actions = env.action_space.shape[0]
```

…и верхнюю границу действия:

```
a_bound = env.action_space.high
```

Создадим объект для класса `DDPG`:

```
ddpg = DDPG(no_of_actions, no_of_states,
a_bound)
```

Инициализируем список для хранения суммарных наград:

```
total_reward = []
```

Зададим количество эпизодов:

```
no_of_episodes = 300
```

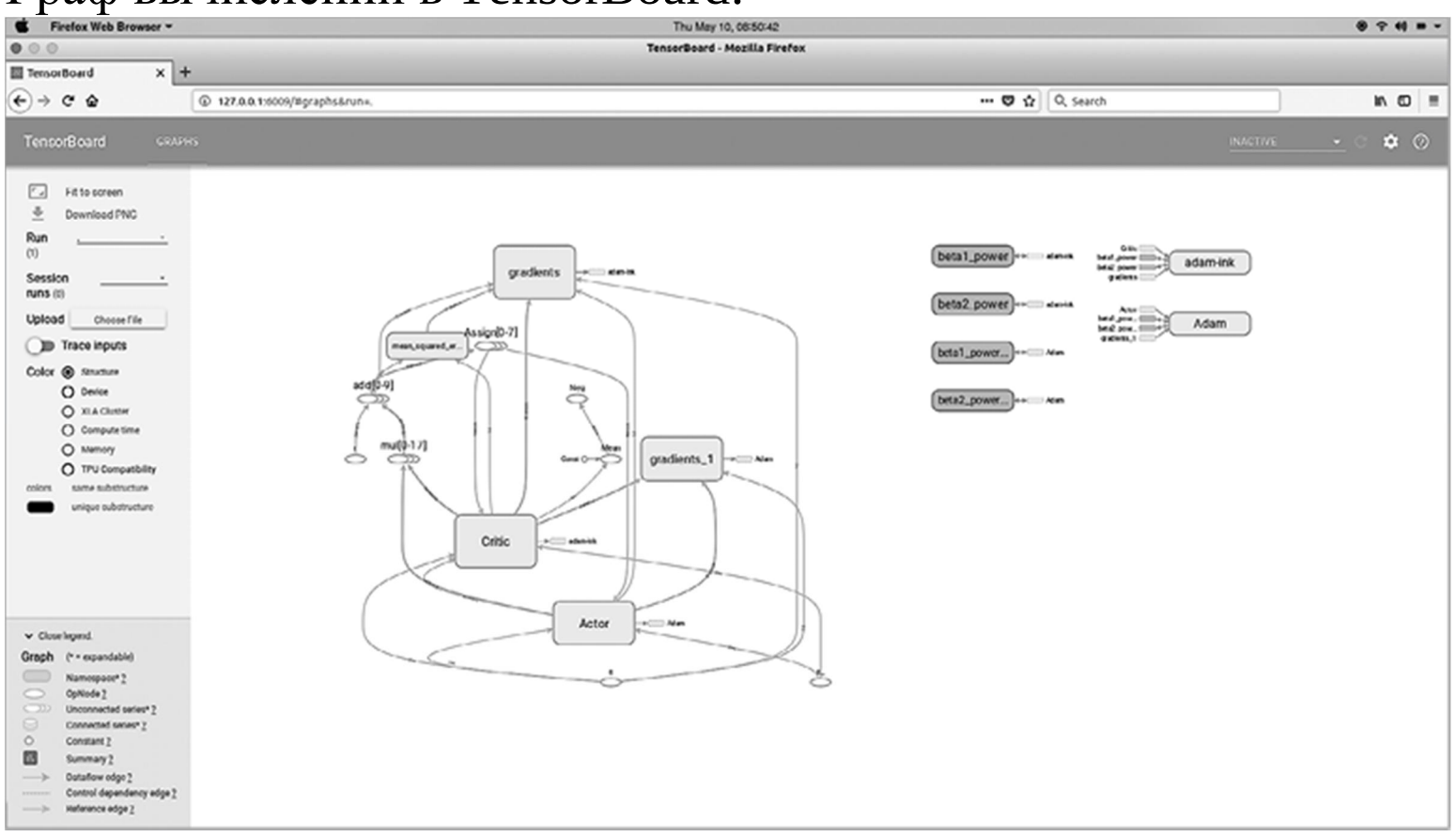
Перейдем к обучению:

```
# Для каждого эпизода
for i in range(no_of_episodes):
    # Инициализировать среду
    s = env.reset()
    # Награда эпизода
    ep_reward = 0
    for j in range(epsiode_steps):
        env.render()

        # Выбрать действие, добавляя шум в
процессе ОУ
        a = ddpg.choose_action(s)
        # Выполнить действие и перейти в
следующее состояние s
```

```
            s_, r, done, info = env.step(a)
            # Сохранить переход в буфере опыта.
            # Выбрать мини-пакет опыта и обучить
сеть.
            ddpg.store_transition(s, a, r, s_)
            # Обновить текущее состояние как
следующее
            s = s_
            # Увеличить награду эпизода
            ep_reward += r
            if j == epsiode_steps-1:
                # Сохранить суммарную награду.
                total_reward.append(ep_reward)
                # Вывести награды, полученные в
каждом эпизоде
                print('Episode:', i, ' Reward: %i' %
int(ep_reward))
                break
```

Результат выглядит так:

Граф вычислений в TensorBoard:



Оптимизация политики доверительной области

Прежде чем разбираться в сути *оптимизации политики доверительной области* (*TRPO*, trust region policy optimization), необходимо понять, что такое *ограниченная* (constrained) оптимизация политики. Мы знаем, что в RL агенты обучаются методом проб и ошибок для максимизации награды. Чтобы найти лучшую политику, агенты исследуют все возможные действия, в том числе плохие. Самые большие проблемы возникают при обучении в реальном мире и при неправильно спроектированных функциях награды. Например, представьте агента, который обучается ходить, не сталкиваясь с препятствиями. Чтобы определить лучшую политику, он исследует различные действия и сталкивается с препятствиями, узнавая о награде. Однако столкновения небезопасны для агента, особенно при обучении в среде реального мира. По этой причине была введена концепция обучения на базе ограничений. Мы устанавливаем порог, и если вероятность столкновения с препятствием ниже этого порога, считается, что агент находится в безопасности; в противном случае его положение небезопасно. Ограничение вводится для того, чтобы агент находился в безопасной области.

В TRPO применяется итеративное улучшение политики: мы устанавливаем ограничение, согласно которому расстояние *Кульбака—Лейбнера* (*KL*) между старой и новой политикой должно быть меньше некоторой константы δ. Это ограничение называется *ограничением доверительной области*.

Что же такое расстояние *KL*? Оно сообщает, в какой степени два вероятностных распределения отличаются друг от друга. Поскольку наши политики являются вероятностными распределениями относительно действий, расстояние *KL*сообщает, насколько новая политика удалена от старой. Почему следует поддерживать расстояние между старой и новой политикой ниже некоторой константы? Потому что новая политика не должна слишком далеко отходить от старой. Ведь если новая политика уйдет слишком далеко, это повлияет на эффективность обучения агента, а также приведет к совершенно иному поведению при обучении. TRPO направлена на улучшение политики (то есть максимизацию награды), но также нужно сохранить уверенность в том, что ограничение доверительной области не нарушается. Для оптимизации параметра сети θ с соблюдением ограничения используется метод сопряженного градиентного спуска (`http://www.idi.ntnu.no/~elster/tdt24/tdt24-f09/cg.pdf`). Алгоритм гарантирует монотонное улучшение политики; он продемонстрировал отличные результаты в различных непрерывных средах.

Теперь посмотрим, как работает TRPO с математической точки зрения; вы можете пропустить этот раздел, если математика вас не интересует.

Обозначим общую ожидаемую скорректированную награду $\eta(\pi)$:

$$\eta(\pi) = \mathbf{E}_{s_0,\, a_0 \ldots \pi'}\left[\sum_{t=0}^{\infty} \gamma^t r(s_t)\right].$$

Теперь рассмотрим новую политику π'; она может быть определена как ожидаемый возврат политики в контексте преимуществ перед старой политикой π:

$$\eta(\pi') = \eta(\pi) + \mathbf{E}_{s_0,\, a_0 \ldots \pi'}\left[\sum_{t=0}^{\infty} \gamma^t A_{\pi}(s_t,\, a_t)\right].$$

Почему мы используем преимущества относительно старой политики? Потому что мы измеряем, насколько хорош результат новой политики π' по сравнению со средней эффективностью старой политики π.
Предыдущее уравнение можно переписать с суммой состояний вместо временных шагов:

$$\begin{aligned}
\eta(\pi') &= \eta(\pi) + \mathbf{E}_{s_0,\, a_0 \ldots \pi'}\left[\sum_{t=0}^{\infty} \gamma^t A_{\pi}(s_t,\, a_t)\right] = \\
&= \eta(\pi) + \sum_{t=0}^{\infty} \sum_{s} P(s_t = s \mid \pi') \sum_{a} \pi'(a \mid s) \gamma^t A_{\pi}(s,\, a) = \\
&= \eta(\pi) + \sum_{s} \sum_{t=0}^{\infty} P(s_t = s \mid \pi') \sum_{a} \pi'(a \mid s) \gamma^t A_{\pi_0}(s,\, a) = \\
&= \eta(\pi) + \sum_{s} \rho_{\pi'}(s) \sum_{a} \pi'(a \mid s) A_{\pi}(s,\, a).
\end{aligned}$$

ρ — скорректированные частоты посещений:

$$\rho_{\pi}(s) = P(s_0 = s) + \gamma P(s_1 = s) + \gamma^2 P(s_2 = s) + \ldots$$

В предыдущем уравнении $\eta(\pi')$ существует сложная зависимость $\rho_{\pi'}(s)$ от π', что усложняет оптимизацию уравнения. По этой причине мы введем локальную аппроксимацию $L_{\pi}(\pi')$ для $\eta(\pi')$:

$$L_{\pi}(\pi') = \eta(\pi) + \sum_{s} \rho_{\pi}(s) \sum_{a} \pi'(a \mid s) A_{\pi}(s,\, a).$$

L_{π} использует частоту посещений ρ_{π} вместо $\rho_{\pi'}$, то есть мы игнорируем изменение частоты посещений состояния из-за изменений в политике. Проще говоря, мы принимаем, что частота посещений состояния у новой и старой политики не отличается. При вычислении градиента L_{π}, который также улучшает η в отношении некоторого параметра θ, мы не можем быть уверены в том, насколько большой шаг следует сделать.

Ш. Какад (Sh. Kakade) и Дж. Лэнгфорд (J. Langford) предложили новый метод обновления политики, называемый *консервативной итерацией по политике*:

$\pi_{нов}(a \mid s) = (1 - \alpha)\pi_{стар}(a \mid s) + \alpha\pi'(a \mid s)$, **(11.1)**

$\pi_{нов}$ — новая политика, $\pi_{стар}$ — старая политика.

$\pi' = \arg\max_{\pi'} L_{\pi_{стар}}(\pi')$, то есть π' — политика, максимизирующая $L_{\pi_{стар}}$.

Какад и Лэнгфорд вывели следующее уравнение из **(11.1)**:

$$\eta(\pi') \geq L_{\pi}(\pi') - CD_{KL}^{\max}(\pi,\, \pi'), \quad \textbf{(11.2)}$$

где C — штрафной коэффициент, равный $\frac{4 \in \gamma}{(1-\alpha)^2}$, а $D_{KL}^{\max}(\pi,\, \pi')$ обозначает расстояние KL между старой и новой политикой.

Если присмотреться к уравнению **(11.2)**, можно заметить, что ожидаемая долгосрочная награда η монотонно возрастает при условии максимизации правой части.

Определим выражение в правой части с именем $M_i(\pi)$ следующим образом:

$$M_i(\pi) \geq L_{\pi_i}(\pi) - CD_{KL}^{\max}(\pi_i, \pi). \quad \textbf{(11.3)}$$

Подставляя **(11.3)** в **(11.2)**, получаем:

$$\eta(\pi_i + 1) \geq M_i(\pi_i + 1). \quad \textbf{(11.4)}$$

Так как мы знаем, что расстояние *KL* между двумя идентичными политиками будет равно 0, можно записать уравнение:

$$\eta(\pi) = M_i(\pi_i). \quad \textbf{(11.5)}$$

Объединяя уравнения **(11.4)** и **(11.5)**, получаем:

$$\eta(\pi_{i+1}) - \eta(\pi) \geq M_i(\pi_{i+1}) - M_i(\pi_i).$$

Как нетрудно понять, в этом уравнении максимизация M_iгарантирует максимизацию ожидаемой награды. Поэтому наша цель — максимизация M_i. Так как мы используем параметризованные политики, заменим π на θ в предыдущем уравнении; $\theta_{стар}$ — политика, которую вы хотите улучшить:

$$\text{максимизация}_\theta \quad \left[L_{\theta_{стар}}(\theta) - CD_{KL}^{\max}(\theta_{стар}, \theta \right].$$

Однако из-за присутствия штрафного коэффициента *C* в этом уравнении размер шага будет очень малым, что, в свою очередь, приведет к замедлению обновлений. По этой причине мы устанавливаем ограничение расстояния *KL* для старой и новой политики, которое является ограничением доверительной области и помогает найти оптимальный размер шага:

$$\text{максимизация}_\theta \quad L_{\theta_0}(\theta),$$

$$\text{при условии} \quad D_{KL}^{\max}(\theta_{стар}, \theta) \leq \delta.$$

Теперь задача ограничения расстояния *KL* касается каждой точки в пространстве состояний. В пространствах с высокой размерностью ее решение непрактично. Поэтому мы используем эвристическое приближение, которое рассматривает среднее расстояние *KL*в следующем виде:

$$\bar{D}_{KL}^{\rho}(\theta_{стар}, \theta) := \mathbf{E}_{s\sim\rho}\left[D_{KL}(\pi_{\theta_1}(\bullet|s) \| \pi_{\theta_2}(\bullet|s)) \right].$$

Итак, теперь мы можем переписать целевую функцию с ограничением среднего расстояния *KL*:

$$\text{максимизация}_\theta \quad L_{\theta_{стар}}(\theta),$$

$$\text{при условии} \quad \bar{D}_{KL}^{\rho\theta_{стар}}(\theta_{стар}, \theta) \leq \delta.$$

Расширяя значение *L*, получаем:

$$\text{максимизация}_\theta \quad \sum_s \rho\theta_{стар}(S) \sum_a \pi_\theta(a|s) A_{\theta_{стар}}(s, a),$$

$$\text{при условии} \quad \bar{D}_{KL}^{\rho\theta_{стар}}(\theta_{стар}, \theta) \leq \delta.$$

В предыдущем уравнении суммирование по состояниям $\sum_{s} \rho\theta_{\text{стар}}$ заменяется ожиданием $E_{s\sim\rho\theta_{\text{стар}}}$, а суммирование по действиям заменяется оценочной функцией выборки по значимости:

$$\sum_{a} \pi_{\theta}(a|s_n) A_{\theta_{\text{стар}}}(s_n, a) = E_{a\sim q}\left[\frac{\pi_{\theta}(a|s_n)}{\pi_{\theta\text{стар}} q(a|s_n)} A_{\theta_{\text{стар}}}(s_n, a)\right].$$

Затем целевые значения преимущества $A_{\theta_{\text{стар}}}$ заменяются значениями $Q_{\theta_{\text{стар}}}$. Окончательная целевая функция принимает вид:

$$\text{максимизация}_{\theta} \quad E_s \pi_{\theta_{\text{стар}}}, a\pi_{\theta_{\text{стар}}}\left[\frac{\pi_{\theta}(a|s)}{\pi_{\theta_{\text{стар}}}(a|s)} A_{\theta_{\text{стар}}}(s, a)\right],$$

$$\text{при условии} \quad E_{s,\pi\theta_{\text{стар}}}\left[D_{KL}(\pi_{\theta_{\text{стар}}}(\bullet|s) \| \pi_{\theta_{\text{стар}}}(\bullet|s))\right] \le \delta.$$

Оптимизация упомянутой ранее целевой функции, содержащей ограничение, называется *ограниченной оптимизацией*. Наше ограничение заключается в поддержании среднего расстояния *KL*между старой и новой политикой не больше δ. Мы используем метод сопряженного градиентного спуска для оптимизации предыдущей функции.

Оптимизация ближайшей политики

А теперь мы рассмотрим другой алгоритм, называемый *оптимизацией ближайшей политики* (*PPO*, proximal policy optimization), который был предложен исследователями из OpenAI. Он используется как усовершенствованная версия **TRPO** и зарекомедовал себя в решении многих сложных задач **RL**. Вспомните суррогатную целевую функцию TRPO. Она представляла собой задачу ограниченной оптимизации, в которой устанавливалось ограничение: среднее расстояние *KL*между старой и новой политикой должно быть не больше δ. Однако TRPO требует значительных вычислительных мощностей для нахождения сопряженных градиентов при выполнении ограниченной оптимизации.

PPO изменяет целевую функцию TRPO, вводя ограничение штрафной составляющей для предотвращения попыток вычисления сопряженных градиентов. Посмотрим, как работает PPO. Определим $r_t(\theta)$ как отношение вероятностей между новой и старой политикой. Таким образом, целевая функция может быть записана в виде:

$$L^{\text{CPI}}(\theta) = \hat{E}_t\left[\frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta_{\text{стар}}}(a_t|s_t)} \hat{A}_t\right] = \hat{E}_t\left[r_t(\theta)\hat{A}_t\right].$$

L^{CPI} обозначает консервативную итерацию по политикам. Но максимизация *L* приведет к неограниченному обновлению политики. Мы переопределим целевую функцию, добавив штрафную составляющую, которая наказывает чрезмерное обновление. Целевая функция примет вид:

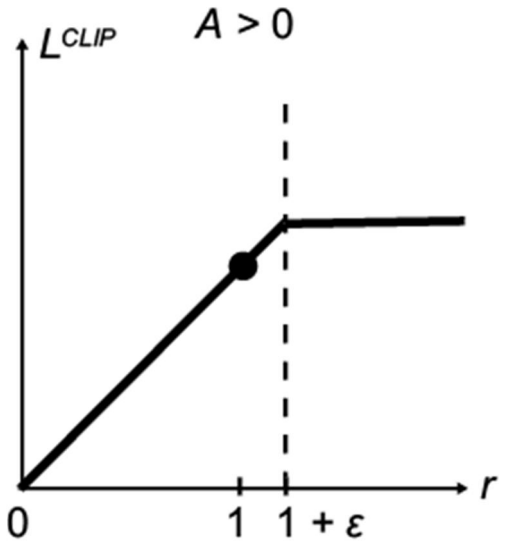
$$L^{CLIP}(\theta) = \hat{E}_t\left[\min r_t(\theta)\hat{A}_t,\ clip(r_t(\theta),\ 1-\varepsilon,\ 1+\varepsilon)\hat{A}_t\right].$$

В уравнение была добавлена новая составляющая $clip(r_t(\theta),\ 1-\varepsilon,\ 1+\varepsilon)\hat{A}_t$. Зачем? Значение $r_t(\theta)$ усекается в интервале $[1 - \varepsilon, 1 + \varepsilon]$; если значение $r_t(\theta)$ приводит к возрастанию целевой функции, пределы интервала сдерживают этот процесс.

Отношение вероятностей усекается по $1 - \varepsilon$ или ε в двух случаях:

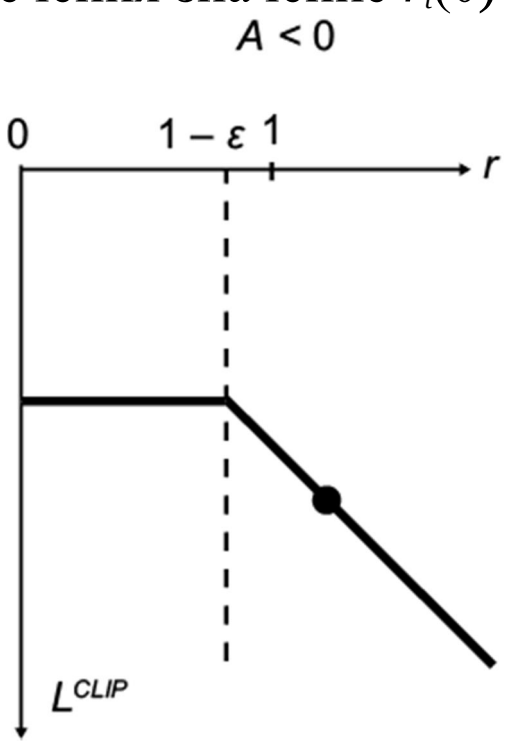
• **Случай 1**: $\hat{A}_t > 0$.

Если преимущество положительно, то есть соответствующему действию должно отдаваться предпочтение перед средним значением всех остальных действий. Значение $r_t(\theta)$ повышается для этого действия, что повышает вероятность его выбора. Из-за усечения значение $r_t(\theta)$ не превысит $1 + \varepsilon$:



• **Случай 2**: $\hat{A}_t < 0$.

Если преимущество отрицательно, это означает, что действие не является значимым и не должно приниматься. В этом случае мы сокращаем значение $r_t(\theta)$ для этого действия, чтобы уменьшить вероятность его выбора. Аналогичным образом из-за выполнения усечения значение $r_t(\theta)$ не упадет ниже $1 - \varepsilon$:



При использовании архитектуры нейросети необходимо определить функцию потерь, которая включает погрешность функции ценности для нашей целевой функции. Мы также добавляем потерю энтропии, чтобы гарантировать достаточную ширину исследований, как было сделано в A3C. Таким образом, итоговая целевая функция принимает вид:

$$L_t^{CLIP+VP+S}(\theta) = \hat{E}_t\left[L_t^{CLIP}(\theta) - c_1 L_t^{VP}(\theta) + c_2 S\left[\pi_\theta\right](s_t)\right],$$

где c_1 и c_2 — коэффициенты, L_t^{VP} — потеря квадратичной ошибки между фактической и целевой функцией ценности, то есть $(V_\theta(s_t - V_t^{\text{target}})^2$, а S — прирост энтропии.

Итоги

Из этой главы вы узнали о методах градиентов политик, которые напрямую оптимизируют политику без необходимости в *Q*-функции. Для иллюстрации была решена задача из игры «Посадка на Луну». Также был рассмотрен алгоритм DDPG, который обладает преимуществами градиентов политик и *Q*-функций.

Затем мы разобрали алгоритмы оптимизации политики (такие, как TRPO), обеспечивающие монотонное улучшение политики через ограничение, согласно которому расстояние *KL* между старой и новой политикой не должно превышать δ.

Также была рассмотрена оптимизация ближайшей политики, в которой ограничение заменено штрафом, наложенным на процесс чрезмерного обновления политики. В **главе 12** будет показано, как построить агента для игры «Автогонки».

Вопросы

1. Что такое «градиенты политик»?
2. Почему градиенты политик эффективно работают?
3. Для чего используется сеть «актор-критик» в DDPG?
4. Что такое «задача ограниченной оптимизации»?
5. Что такое «доверительная область»?
6. Как PPO преодолевает недостатки TRPO?

Дополнительные источники

Статья о DDPG: `https://arxiv.org/pdf/1509.02971.pdf.`

Статья о TRPO: `https://arxiv.org/pdf/1502.05477.pdf.`

Статья о PPO: `https://arxiv.org/pdf/1707.06347.pdf.`

12. «Автогонки» с использованием DQN

Из нескольких последних глав вы узнали, как работает глубокое *Q*-обучение посредством аппроксимации *Q*-функции нейросетью. В них были рассмотрены различные усовершенствования **DQN(deep Q network**), такие как двойное *Q*-обучение, дуэльные сетевые архитектуры и глубокие рекуррентные *Q*-сети. Вы увидели, как DQN использует буфер воспроизведения для хранения опыта агента и обучает сеть на примере мини-выборки из буфера. Также были реализованы DQN для игр Atari и **DRQN** (**deep recurrent Q network**) для игры Doom. В этой главе будет рассмотрена подробная реализация дуэльной DQN, которая

очень похожа на обычную DQN, не считая того, что последний полносвязный слой в ней разбивается на два потока, а именно поток ценности и поток преимущества, которые затем объединяются для обобщения *Q*-функции. Вы увидите, как обучить агента для игры «Автогонки» на базе дуэльной DQN.

В этой главе рассматриваются следующие темы:

- Функции-обертки среды.
- Дуэльная сеть.
- Буфер воспроизведения.
- Обучение сети.
- «Автогонки».

Функции-обертки среды

Код, использованный в этой главе, был взят из GitHub-репозитория Джакомо Шпиглера (Giacomo Spigler) (**`https://github.com/spiglerg/DQN_DDQN_Dueling_and_DDPG_Tensorflow`**). В этой главе код приводится с обильными комментариями. За полностью структурированным кодом обращайтесь к GitHub-репозиторию.

Сначала импортируются необходимые библиотеки:

```
import numpy as np
import tensorflow as tf
import gym
from gym.spaces import Box
from scipy.misc import imresize
import random
import cv2
import time
import logging
import os
import sys
```

Определяется класс `EnvWrapper` и некоторые функции-обертки среды:

```
class EnvWrapper:
```

Определяется метод `__init__` и инициализируются переменные:

```
    def __init__(self, env_name, debug=False):
```

Инициализируется среда Gym:

```
        self.env = gym.make(env_name)
```

Определяется `action_space`:

```
        self.action_space =
self.env.action_space
```

Определяется `observation_space`:

```
        self.observation_space = Box(low=0,
high=255, shape=(84, 84, 4))
```

Инициализируется значение `frame_num` для хранения количества кадров:

```
        self.frame_num = 0
```

Инициализируется монитор для записи игрового экрана:

```
        self.monitor = self.env.monitor
```

Инициализируется буфер кадров `frames`:

```
        self.frames = np.zeros((84, 84, 4),
dtype=np.uint8)
```

Инициализируется логический признак `debug`; если ему присвоено значение `true`, отображаются несколько последних кадров:

```
        self.debug = debug

        if self.debug:
            cv2.startWindowThread()
            cv2.namedWindow("Game")
```

Определяется функция `step`, которая получает текущее состояние на входе и возвращает предварительно обработанный кадр следующего состояния:

```
    def step(self, a):
        ob, reward, done, xx = self.env.step(a)
     return self.process_frame(ob), reward,
done, xx
```

Определяется функция `reset` для сброса среды; после сброса возвращается предварительно обработанный игровой экран:

```
    def reset(self):
        self.frame_num = 0
        return
self.process_frame(self.env.reset())
```

Также определяется другая функция для прорисовки среды:

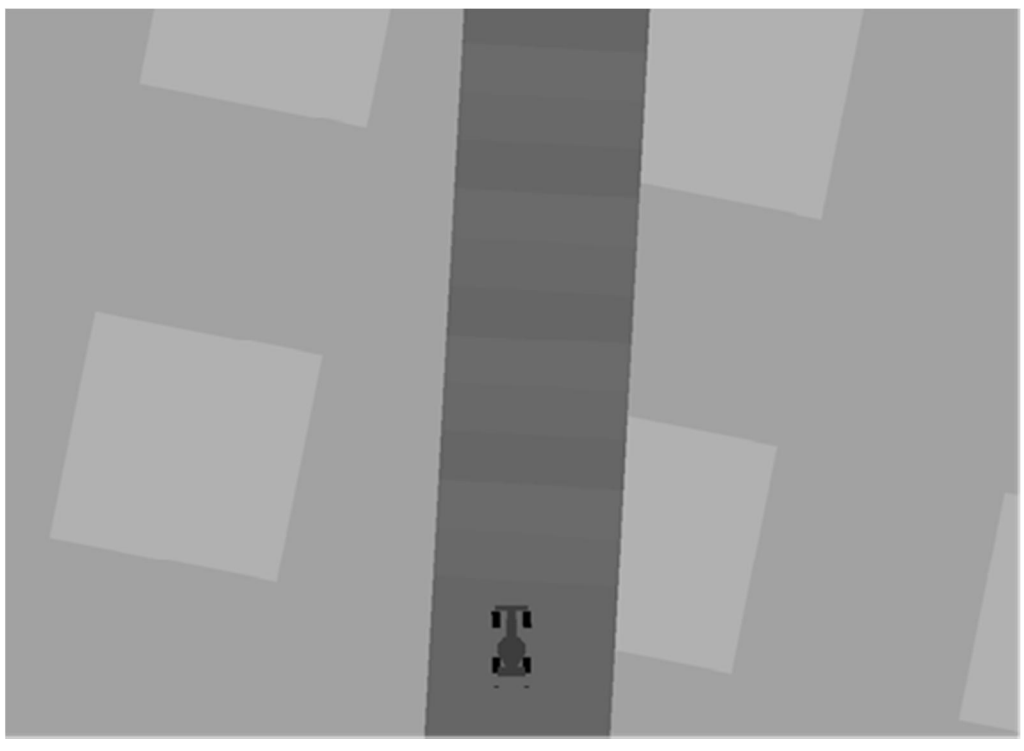
```
    def render(self):
        return self.env.render()
```

Определяется функция `process_frame` для предварительной обработки кадра:

```
    def process_frame(self, frame):

        # Преобразовать изображение в оттенки
серого
        state_gray = cv2.cvtColor(frame,
cv2.COLOR_BGR2GRAY)

        # Изменить размеры
        state_resized =
cv2.resize(state_gray,(84,110))
        #resize
```

```
            gray_final = state_resized[16:100,:]

            if self.frame_num == 0:
                self.frames[:, :, 0] = gray_final
                self.frames[:, :, 1] = gray_final
                self.frames[:, :, 2] = gray_final
                self.frames[:, :, 3] = gray_final

            else:
                self.frames[:, :, 3] =
self.frames[:, :, 2]
                self.frames[:, :, 2] =
self.frames[:, :, 1]
                self.frames[:, :, 1] =
self.frames[:, :, 0]
                self.frames[:, :, 0] = gray_final

            # Увеличить счетчик frame_num

            self.frame_num += 1

            if self.debug:
                cv2.imshow('Game', gray_final)

            return self.frames.copy()
```

После предварительной обработки игровой экран выглядит примерно так:



Дуэльная сеть

Переходим к построению дуэльной DQN; мы построим три сверточных слоя, за которыми следуют два полносвязных слоя, последний из которых будет разбит на два подслоя для потоков ценности и преимущества. Мы используем агрегативный слой, который объединяет

поток ценности с потоком преимущества для вычисления значения `q`. Размеры этих уровней:

Слой 1: 32 фильтра 8 × 8 с шагом 4 + RELU.
Слой 2: 64 фильтра 4 × 4 с шагом 2 + RELU.
Слой 3: 64 фильтра 3 × 3 с шагом 1 + RELU.
Слой 4a: 512 единиц, полносвязный слой + RELU.
Слой 4b: 512 единиц, полносвязный слой + RELU.
Слой 5a: 1 единица FC + RELU (ценность состояния).
Слой 5b: Действия FC + RELU (преимущество).
Слой 6: Агрегативный $V(s) + A(s, a)$.

`class QNetworkDueling(QNetwork):`

Определяем метод `__init__` для инициализации всех слоев:

```
    def __init__(self, input_size, output_size, name):
        self.name = name
        self.input_size = input_size
        self.output_size = output_size
        with tf.variable_scope(self.name):

            # Три сверточных слоя
            self.W_conv1 = self.weight_variable([8, 8, 4, 32])
            self.B_conv1 = self.bias_variable([32])
            self.stride1 = 4

            self.W_conv2 = self.weight_variable([4, 4, 32, 64])
            self.B_conv2 = self.bias_variable([64])
            self.stride2 = 2

            self.W_conv3 = self.weight_variable([3, 3, 64, 64])
            self.B_conv3 = self.bias_variable([64])
            self.stride3 = 1
            # Два полносвязных слоя
            self.W_fc4a = self.weight_variable([7*7*64, 512])
            self.B_fc4a = self.bias_variable([512])
            self.W_fc4b = self.weight_variable([7*7*64, 512])
```

```
                self.B_fc4b =
self.bias_variable([512])

                # Поток ценности
                self.W_fc5a =
self.weight_variable([512, 1])
                self.B_fc5a =
self.bias_variable([1])

                # Поток преимущества
                self.W_fc5b =
self.weight_variable([512, self.output_size])
                self.B_fc5b =
self.bias_variable([self.output_size])
```

Определяем метод `__call__` и выполняем операцию свертки:

```
    def __call__(self, input_tensor):
        if type(input_tensor) == list:
            input_tensor = tf.concat(1,
input_tensor)

        with tf.variable_scope(self.name):
            # Выполнить свертку на трех уровнях

            self.h_conv1 = tf.nn.relu(
tf.nn.conv2d(input_tensor,
  self.W_conv1, strides=[1, self.stride1,
self.stride1, 1], padding='VALID')
  + self.B_conv1 )

            self.h_conv2 = tf.nn.relu(
tf.nn.conv2d(self.h_conv1,
  self.W_conv2, strides=[1, self.stride2,
self.stride2, 1], padding='VALID')
  + self.B_conv2 )

                self.h_conv3 = tf.nn.relu(
tf.nn.conv2d(self.h_conv2,
  self.W_conv3, strides=[1, self.stride3,
self.stride3, 1], padding='VALID')
  + self.B_conv3 )


            # Деструктуризировать сверточный
вывод
```

```
                self.h_conv3_flat =
tf.reshape(self.h_conv3, [-1, 7*7*64])
                # Полносвязный уровень
                self.h_fc4a =
tf.nn.relu(tf.matmul(self.h_conv3_flat,
  self.W_fc4a) + self.B_fc4a)

                self.h_fc4b =
tf.nn.relu(tf.matmul(self.h_conv3_flat,
  self.W_fc4b) + self.B_fc4b)

                # Вычислить поток ценности и поток
преимущества
                self.h_fc5a_value =
tf.identity(tf.matmul(self.h_fc4a,
  self.W_fc5a) + self.B_fc5a)
                self.h_fc5b_advantage =
tf.identity(tf.matmul(self.h_fc4b,
  self.W_fc5b) + self.B_fc5b)

                # Объединить потоки ценности и
преимущества
                self.h_fc6 = self.h_fc5a_value + (
self.h_fc5b_advantage -
  tf.reduce_mean(self.h_fc5b_advantage,
reduction_indices=[1,],
  keep_dims=True) )

            return self.h_fc6
```

Память воспроизведения

Необходимо построить буфер воспроизведения для хранения всего опыта агента, из которого будет выбран мини-пакет для обучения сети:

```
class ReplayMemoryFast:
```

Сначала определяется метод `__init__` и инициализируется размер буфера:

```
    def __init__(self, memory_size,
minibatch_size):

        # Максимальное количество данных для
хранения
        self.memory_size = memory_size

        # Размер мини-пакета
```

```
        self.minibatch_size = minibatch_size
        self.experience =
[None]*self.memory_size
        self.current_index = 0
        self.size = 0
```

Затем определяется функция `store` для сохранения опыта:

```
  def store(self, observation, action, reward,
newobservation, is_terminal):
```

Опыт сохраняется в виде кортежа (текущее состояние, действие, награда, следующее состояние, признак завершающего состояния):

```
        self.experience[self.current_index] =
(observation, action, reward,
  newobservation, is_terminal)
        self.current_index += 1
        self.size = min(self.size+1,
self.memory_size)
```

Если индекс превышает память, его нужно уменьшить на размер памяти:

```
        if self.current_index >=
self.memory_size:
            self.current_index -=
self.memory_size
```

Затем определяется функция `sample` для выборки мини-пакета опыта:

```
  def sample(self):
        if self.size < self.minibatch_size:
            return []

        # Случайная выборка нескольких индексов
        samples_index =
  np.floor(np.random.random((self.minibatch_size,)
)*self.size)

        # Получить опыт по выбранным индексам
        samples = [self.experience[int(i)] for i
in samples_index]

        return samples
```

Обучение сети

А теперь посмотрим, как происходит обучение сети.

Сначала мы определяем класс `DQN`и инициализируем все переменные в методе `__init__`:

```
  class DQN(object):
```

```
    def __init__(self, state_size,
                       action_size,
                       session,
                       summary_writer = None,
                       exploration_period =
1000,
                       minibatch_size = 32,
                       discount_factor = 0.99,
                       experience_replay_buffer
= 10000,
                       target_qnet_update_freque
ncy = 10000,
                       initial_exploration_epsil
on = 1.0,
                       final_exploration_epsilon
= 0.05,
                       reward_clipping = -1,
                        ):
```

Инициализируем все переменные:

```
        self.state_size = state_size
        self.action_size = action_size

        self.session = session
        self.exploration_period =
float(exploration_period)
        self.minibatch_size = minibatch_size
        self.discount_factor =
tf.constant(discount_factor)
        self.experience_replay_buffer =
experience_replay_buffer
        self.summary_writer = summary_writer
        self.reward_clipping = reward_clipping

        self.target_qnet_update_frequency =
target_qnet_update_frequency
        self.initial_exploration_epsilon =
initial_exploration_epsilon
        self.final_exploration_epsilon =
final_exploration_epsilon
        self.num_training_steps = 0
```

Инициализируем основную дуэльную DQN созданием экземпляра класса `QNetworkDueling`:

```
        self.qnet =
QNetworkDueling(self.state_size, self.action_size,
```

```
"qnet")
```

Аналогичным образом инициализируется целевая дуэльная DQN:

```
        self.target_qnet =
QNetworkDueling(self.state_size,
  self.action_size, "target_qnet")
```

Инициализируем оптимизатор `RMSPropOptimizer`:

```
        self.qnet_optimizer =
  tf.train.RMSPropOptimizer(learning_rate=0.00025,
decay=0.99, epsilon=0.01)
```

Для инициализации буфера `experience_replay_buffer` создаем экземпляр класса `ReplayMemoryFast`:

```
        self.experience_replay =
  ReplayMemoryFast(self.experience_replay_buffer,
self.minibatch_size)
        # Создать граф вычислений
        self.create_graph()
```

Затем определяем функцию `copy_to_target_network` для копирования весов из основной сети в целевую:

```
    def copy_to_target_network(source_network,
target_network):
        target_network_update = []
        for v_source, v_target in
zip(source_network.variables(),
  target_network.variables()):

            # Обновить целевую сеть
            update_op =
v_target.assign(v_source)
            target_network_update.append(update_
op)

        return tf.group(*target_network_update)
```

Далее мы определяем функцию `create_graph` и строим граф вычислений:

```
    def create_graph(self):
```

Вычисляем `q_values` и выбираем действие с максимальным значением `q`:

```
        with tf.name_scope("pick_action"):

            # заместитель для состояния
            self.state =
tf.placeholder(tf.float32, (None,)+self.state_size
  , name="state")
```

```
                # заместитель для значений q
                self.q_values =
tf.identity(self.qnet(self.state) ,
  name="q_values")

                # заместитель для прогнозируемых
действий
                self.predicted_actions =
tf.argmax(self.q_values, dimension=1 ,
  name="predicted_actions")

                # Вывести гистограмму для
отслеживания
                # максимальных значений q
                tf.histogram_summary("Q values",
  tf.reduce_mean(tf.reduce_max(self.q_values, 1)))
# Сохранить максимальные

 # значения q для отслеживания

 # хода обучения
```

Далее вычисляем целевую будущую награду:

```
        with
tf.name_scope("estimating_future_rewards"):
                self.next_state =
tf.placeholder(tf.float32,
  (None,)+self.state_size , name="next_state")

                self.next_state_mask =
tf.placeholder(tf.float32, (None,) ,
  name="next_state_mask")
                self.rewards =
tf.placeholder(tf.float32, (None,) ,
  name="rewards")

                self.next_q_values_targetqnet =
  tf.stop_gradient(self.target_qnet(self.next_stat
e),
  name="next_q_values_targetqnet")
                self.next_q_values_qnet =
  tf.stop_gradient(self.qnet(self.next_state),
name="next_q_values_qnet")
```

```
                self.next_selected_actions =
tf.argmax(self.next_q_values_qnet,
  dimension=1)

                self.next_selected_actions_onehot =
  tf.one_hot(indices=self.next_selected_actions,
depth=self.action_size)

                self.next_max_q_values =
tf.stop_gradient( tf.reduce_sum(
  tf.mul( self.next_q_values_targetqnet,
self.next_selected_actions_onehot )
  , reduction_indices=[1,] ) *
self.next_state_mask )

                self.target_q_values = self.rewards
+
  self.discount_factor*self.next_max_q_values
```

Выполняем оптимизацию с использованием оптимизатора `RMSPropOptimizer`:

```
            with tf.name_scope("optimization_step"):
                self.action_mask =
tf.placeholder(tf.float32, (None,
  self.action_size) , name="action_mask")

                self.y = tf.reduce_sum(
self.q_values * self.action_mask ,
  reduction_indices=[1,])

                # УСЕЧЕНИЕ ОШИБКИ
                self.error = tf.abs(self.y -
self.target_q_values)

                quadratic_part =
tf.clip_by_value(self.error, 0.0, 1.0)
                linear_part = self.error -
quadratic_part

                self.loss = tf.reduce_mean(
0.5*tf.square(quadratic_part) +
  linear_part )

                # Оптимизировать градиенты
```

```
                qnet_gradients =
  self.qnet_optimizer.compute_gradients(self.loss,
self.qnet.variables())

                for i, (grad, var) in
enumerate(qnet_gradients):
                    if grad is not None:
                        qnet_gradients[i] =
(tf.clip_by_norm(grad, 10), var)

                self.qnet_optimize =
  self.qnet_optimizer.apply_gradients(qnet_gradien
ts)
```

Веса основной сети копируем в целевую сеть:

```
            with
tf.name_scope("target_network_update"):
                self.hard_copy_to_target =
  DQN.copy_to_target_network(self.qnet,
self.target_qnet)
```

Определяем функцию `store` для сохранения всего опыта в `experience_replay_buffer`:

```
        def store(self, state, action, reward,
next_state, is_terminal):
            # Усечение наград
            if self.reward_clipping > 0.0:
                reward = np.clip(reward, -
self.reward_clipping,
  self.reward_clipping)

            self.experience_replay.store(state,
action, reward, next_state,
  is_terminal)
```

Определяем функцию `action` для выбора действий с использованием эпсилон-жадной стратегии с затуханием:

```
        def action(self, state, training = False):
            if self.num_training_steps >
self.exploration_period:
                epsilon =
self.final_exploration_epsilon
            else:
                epsilon =
self.initial_exploration_epsilon -
  float(self.num_training_steps) *
(self.initial_exploration_epsilon -
```

```
  self.final_exploration_epsilon) /
self.exploration_period

            if not training:
                epsilon = 0.05

            if random.random() <= epsilon:
                action = random.randint(0,
self.action_size-1)
            else:
                action =
self.session.run(self.predicted_actions,
  {self.state:[state] } )[0]

            return action
```

А теперь определим функцию `train` для обучения сети:

```
def train(self):
```

Веса основной сети копируем в целевую сеть:

```
        if self.num_training_steps == 0:
            print "Training starts..."
            self.qnet.copy_to(self.target_qnet)
```

Выполняем выборку опыта из памяти воспроизведения:

```
        minibatch =
self.experience_replay.sample()
```

Из мини-пакета извлекаем состояния, действия, награды и следующие состояния:

```
        batch_states = np.asarray( [d[0] for d
in minibatch] )
        actions = [d[1] for d in minibatch]
        batch_actions = np.zeros(
(self.minibatch_size, self.action_size) )
        for i in xrange(self.minibatch_size):
            batch_actions[i, actions[i]] = 1

        batch_rewards = np.asarray( [d[2] for d
in minibatch] )
        batch_newstates = np.asarray( [d[3] for
d in minibatch] )

        batch_newstates_mask = np.asarray( [not
d[4] for d in minibatch] )
```

Проводим операцию обучения:

```
            scores, _, =
self.session.run([self.q_values,
self.qnet_optimize],
                                                {
self.state: batch_states,
                                                  self.nex
t_state: batch_newstates,
                                                  self.nex
t_state_mask:
  batch_newstates_mask,
                                                  self.rew
ards: batch_rewards,
                                                  self.act
ion_mask: batch_actions} )
```

Обновляем веса целевой сети:

```
            if self.num_training_steps %
self.target_qnet_update_frequency ==
  0:
                self.session.run(
self.hard_copy_to_target )

                print 'mean maxQ in minibatch:
',np.mean(np.max(scores,1))

                str_ =
self.session.run(self.summarize, { self.state:
  batch_states,
                                                  self.nex
t_state: batch_newstates,
                                                  self.nex
t_state_mask:
  batch_newstates_mask,
                                                  self.rew
ards: batch_rewards,
                                                  self.act
ion_mask: batch_actions})
                self.summary_writer.add_summary(str_
, self.num_training_steps)
            self.num_training_steps += 1
```

«Автогонки»

К настоящему моменту мы изучили процесс построения дуэльной DQN. А теперь посмотрим, как ее использовать для игры «Автогонки».

Сначала импортируются все необходимые библиотеки:

```
import gym
import time
import logging
import os
import sys
import tensorflow as tf
```
Затем инициализируются все необходимые переменные:
```
ENV_NAME = 'Seaquest-v0'
TOTAL_FRAMES = 20000000
MAX_TRAINING_STEPS = 20*60*60/3
TESTING_GAMES = 30
MAX_TESTING_STEPS = 5*60*60/3
TRAIN_AFTER_FRAMES = 50000
epoch_size = 50000
MAX_NOOP_START = 30
LOG_DIR = 'logs'
outdir = 'results'
logger = tf.train.SummaryWriter(LOG_DIR)
# Инициализировать сеанс tensorflow
session = tf.InteractiveSession()
```
Построение агента:
```
agent =
DQN(state_size=env.observation_space.shape,
action_size=env.action_space.n,
session=session,
summary_writer = logger,
exploration_period = 1000000,
minibatch_size = 32,
discount_factor = 0.99,
experience_replay_buffer = 1000000,
target_qnet_update_frequency = 20000,
initial_exploration_epsilon = 1.0,
final_exploration_epsilon = 0.1,
reward_clipping = 1.0,
)
session.run(tf.initialize_all_variables())
logger.add_graph(session.graph)
saver = tf.train.Saver(tf.all_variables())
```
Сохранение данных:
```
env.monitor.start(outdir+'/'+ENV_NAME,force =
True,
video_callable=multiples_video_schedule)
num_frames = 0
num_games = 0
```

```
  current_game_frames = 0
  init_no_ops =
np.random.randint(MAX_NOOP_START+1)
  last_time = time.time()
  last_frame_count = 0.0
  state = env.reset()
```

Запускаем обучение:

```
  while num_frames <= TOTAL_FRAMES+1:
      if test_mode:
          env.render()
      num_frames += 1
      current_game_frames += 1
```

Выбираем действие с учетом текущего состояния:

```
      action = agent.action(state, training =
True)
```

Выполняем действия со средой, получаем награду и переходим к следующему состоянию `next_state`:

```
      next_state,reward,done,_ = env.step(action)
```

Информацию перехода сохраняем в `experience_replay_buffer`:

```
      if current_game_frames >= init_no_ops:
          agent.store(state,action,reward,next_sta
te,done)
      state = next_state
```

Обучение агента:

```
      if num_frames>=TRAIN_AFTER_FRAMES:
          agent.train()

      if done or current_game_frames >
MAX_TRAINING_STEPS:
          state = env.reset()
          current_game_frames = 0
          num_games += 1
          init_no_ops =
np.random.randint(MAX_NOOP_START+1)
```

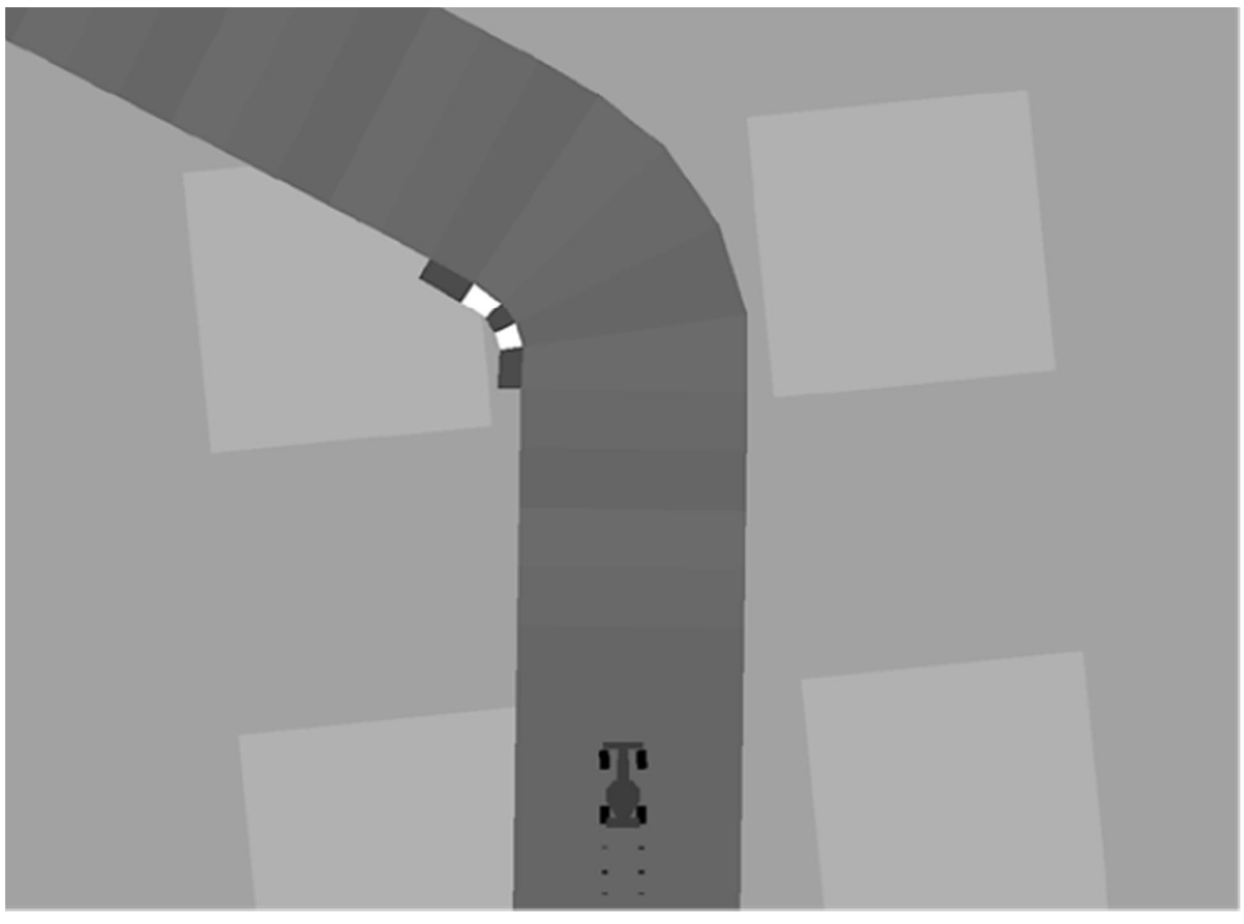
Параметры сети сохраняются после каждой эпохи:

```
      if num_frames % epoch_size == 0 and
num_frames > TRAIN_AFTER_FRAMES:
          saver.save(session,
  outdir+"/"+ENV_NAME+"/model_"+str(num_frames/100
0)+"k.ckpt")
          print "epoch: frames=",num_frames,"
games=",num_games
```

Эффективность проверяется через каждые две эпохи:

```
    if num_frames % (2*epoch_size) == 0 and
num_frames > TRAIN_AFTER_FRAMES:
        total_reward = 0
        avg_steps = 0
        for i in xrange(TESTING_GAMES):
            state = env.reset()
            init_no_ops =
np.random.randint(MAX_NOOP_START+1)
            frm = 0

            while frm < MAX_TESTING_STEPS:
                frm += 1
                env.render()
                action = agent.action(state,
training = False)
                if current_game_frames <
init_no_ops:
                    action = 0
                state,reward,done,_ =
env.step(action)
                total_reward += reward

                if done:
                    break

            avg_steps += frm
        avg_reward =
float(total_reward)/TESTING_GAMES
        str_ = session.run(
tf.scalar_summary('test reward
('+str(epoch_size/1000)+'k)', avg_reward) )
        logger.add_summary(str_, num_frames)
        state = env.reset()

env.monitor.close()
```

Скриншот показывает, как проходит обучение агента для победы в игре «Автогонки»:



Итоги

В этой главе подробно рассматривается реализация дуэльной DQN. Мы начали ее изучение с функций-оберток среды для предварительной обработки игровых экранов, а затем определили класс `QNetworkDueling`. В данном случае была реализована дуэльная *Q*-сеть, которая разбивает итоговый полносвязный слой DQN на поток ценности и поток преимущества, а затем объединяет эти два потока для вычисления значения `q`. Было показано, как создать буфер воспроизведения и хранения опыта и выполнить выборку мини-пакета опыта для обучения сети. Наконец, мы инициализировали среду «Автогонок» средствами OpenAI Gym и провели обучение агента.
В **главе 13** будут представлены некоторые последние достижения в области RL.

Вопросы

1. Чем DQN отличается от дуэльной DQN?
2. Напишите код на Python для буфера воспроизведения.
3. Что такое «целевая сеть»?
4. Напишите код на Python для реализации приоритетного буфера воспроизведения опыта.
5. Напишите функцию на Python для эпсилон-жадной стратегии с затуханием.
6. Чем дуэльная DQN отличается от двойной DQN?
7. Напишите функцию на Python для обновления целевой сети весами основной сети.

Дополнительные источники

Flappy Bird с использованием DQN: `https://github.com/yenchenlin/DeepLearningFlappyBird`.

Super Mario с использованием DQN: `https://github.com/JSDanielPark/tensorflow_dqn_supermario`.

13. Последние достижения и следующие шаги

Поздравляем! Вы добрались до последней главы и прошли долгий путь! Мы начали с основ RL (MDP, методы Монте-Карло и обучение TD), а затем перешли к сложным алгоритмам глубокого обучения с подкреплением, таким как DQN, DRQN и A3C. Также были представлены интересные современные методы градиентов политик (DDPG, PPO и TRPO) и построен последний проект книги — агент для игры «Автогонки». Однако в области RL еще осталось много «белых пятен», и каждый день появляется что-то новое. В этой главе будут представлены некоторые достижения в области RL, включая иерархические и инвертированные методы RL.

В этой главе рассматриваются следующие темы:

- Агенты, дополненные воображением (I2A).
- Обучение на человеческих предпочтениях.
- Глубокое *Q*-обучение на примере демонстраций.
- Ретроспективное воспроизведение опыта.
- Иерархическое обучение с подкреплением.
- Инвертированное обучение с подкреплением.

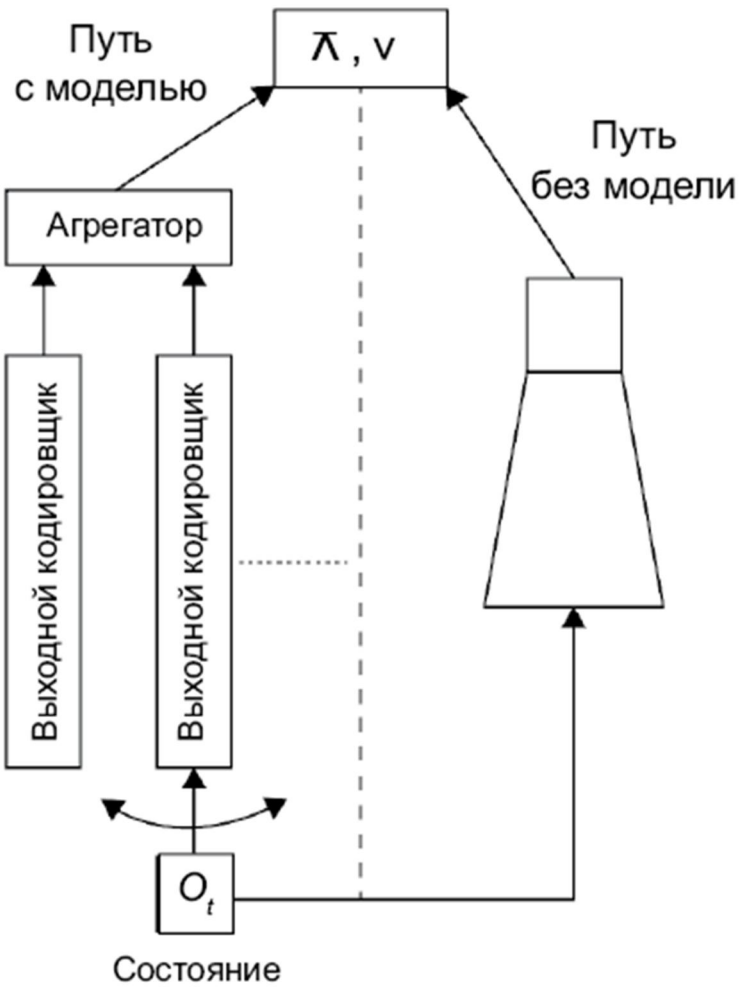
Агенты, дополненные воображением

Вы играете в шахматы? Если бы я предложил сыграть, как бы вы подошли к игре? Прежде чем двигать фигуры на доске, вы попытались бы представить последствия перемещения каждой фигуры и делали бы только те ходы, которые, по вашему мнению, приведут к победе. Итак, прежде чем предпринимать какие-либо действия, вы представляете себе их последствия. Если они благоприятны — вы выполняете эти действия, а если нет — воздерживаетесь от их выполнения.

Аналогичным образом действуют *агенты, дополненные воображением*(*I2A*; imagination augmented agents): прежде чем выполнять действие в среде, они пытаются представить последствия этого действия. Если агент думает, что действие принесет хороший результат, он выполняет его. Дополнение агентов воображением — следующий серьезный шаг на пути к построению обобщенного искусственного интеллекта.

А теперь посмотрим, как в общих чертах работают агенты, дополненные воображением; I2A используют преимущества обучения как с моделью, так и без модели.

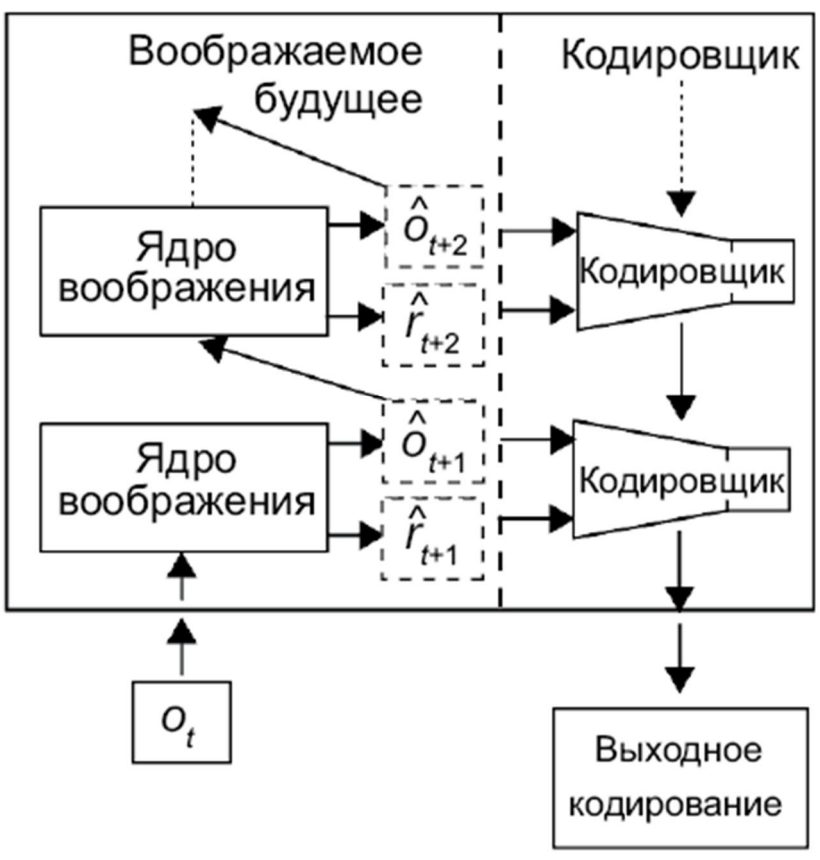
Архитектура I2A выглядит так:



Архитектура I2A

Действие, выполняемое агентом, является результатом как пути с моделью, так и пути без модели. На пути с моделью имеются так называемые *выходные кодировщики*; именно в них агент выполняет задачи воображения. Присмотримся к выходным кодировщикам повнимательнее. Выходной кодировщик устроен так:

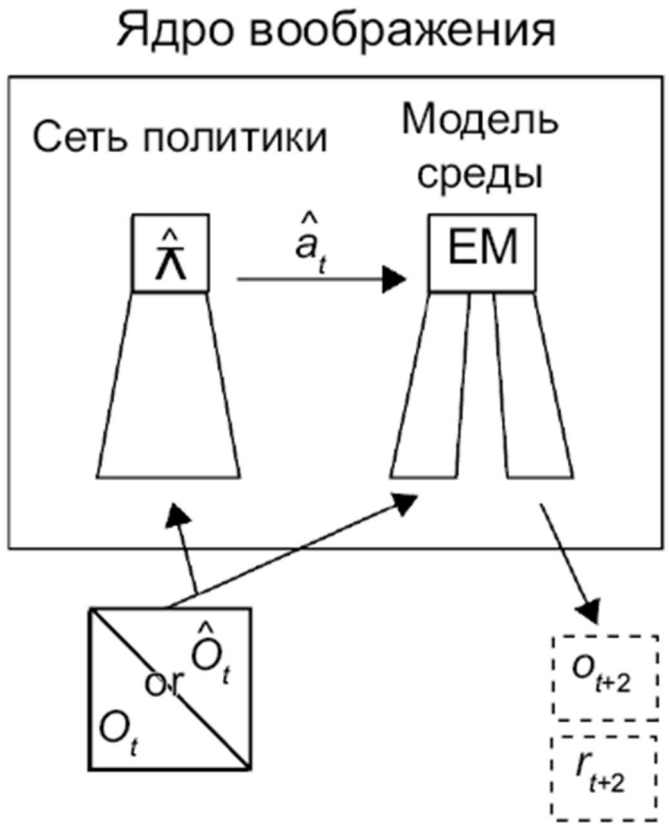
Выходной кодировщик



Выходные кодировщики содержат слои двух типов: воображаемое будущее и кодировщик. В области воображаемого будущего выполняются задачи воображения. Взгляните на рисунок; воображаемое будущее состоит из ядра воображения. При подаче состояния O_t ядру воображения мы получаем новое состояние $\hat{O}_{t+1}$ и награду $\hat{r}_{t+1}$, а при подаче нового состояния $\hat{O}_{t+1}$следующему ядру воображения будет получено новое состояние $\hat{O}_{t+2}$ и награда $\hat{r}_{t+2}$. Если повторить эту

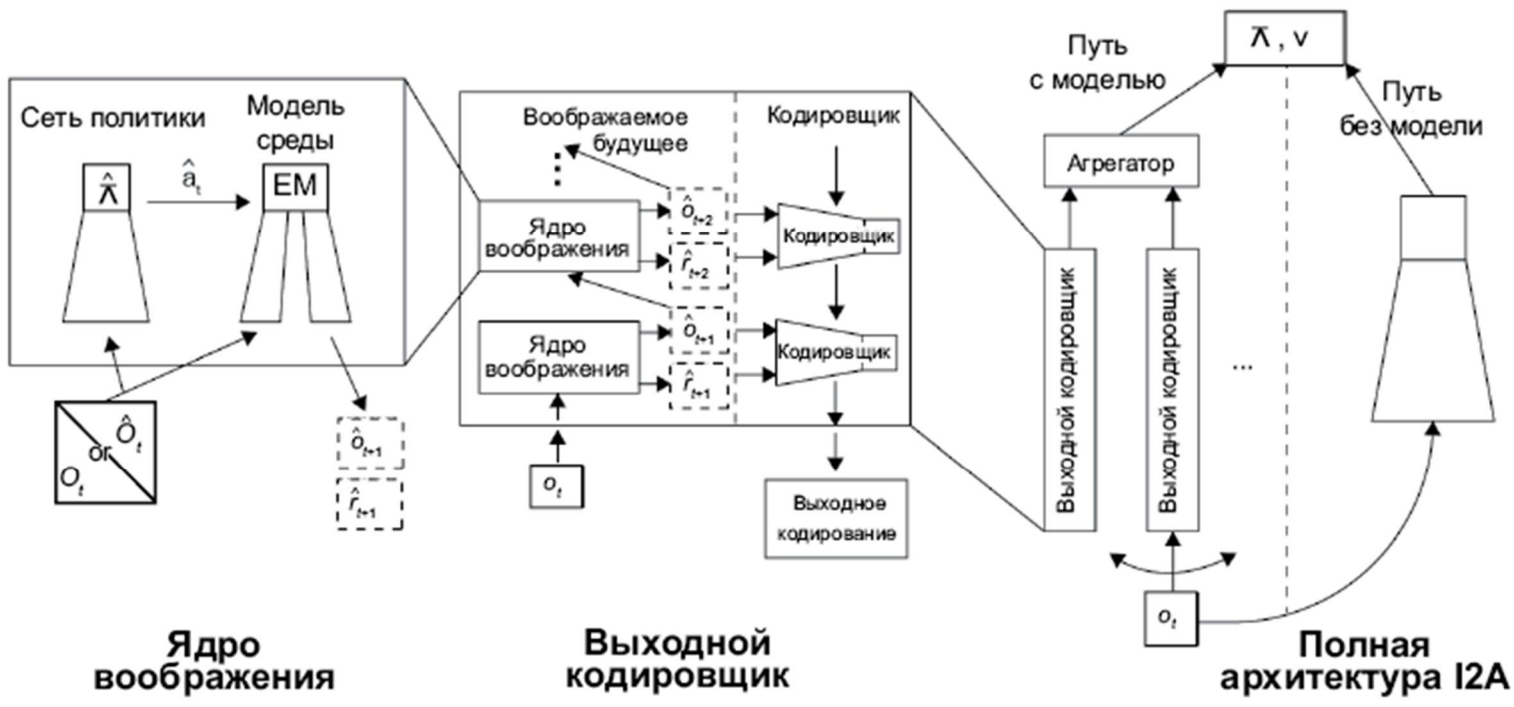
процедуру для *n*шагов, будет получен выход (rollout), который, по сути, представляет пару из состояний и наград, а затем кодировщики, такие как LSTM, будут использованы для кодирования этого выхода. Выходные кодировки фактически описывают будущие воображаемые пути. Несколько выходных кодировщиков обрабатывают разные будущие воображаемые пути, а агрегатор обобщает выходные кодировки.

Но постойте, как работает **ядро воображения**? Из каких компонентов оно состоит? Отдельное ядро воображения показано на следующем рисунке:



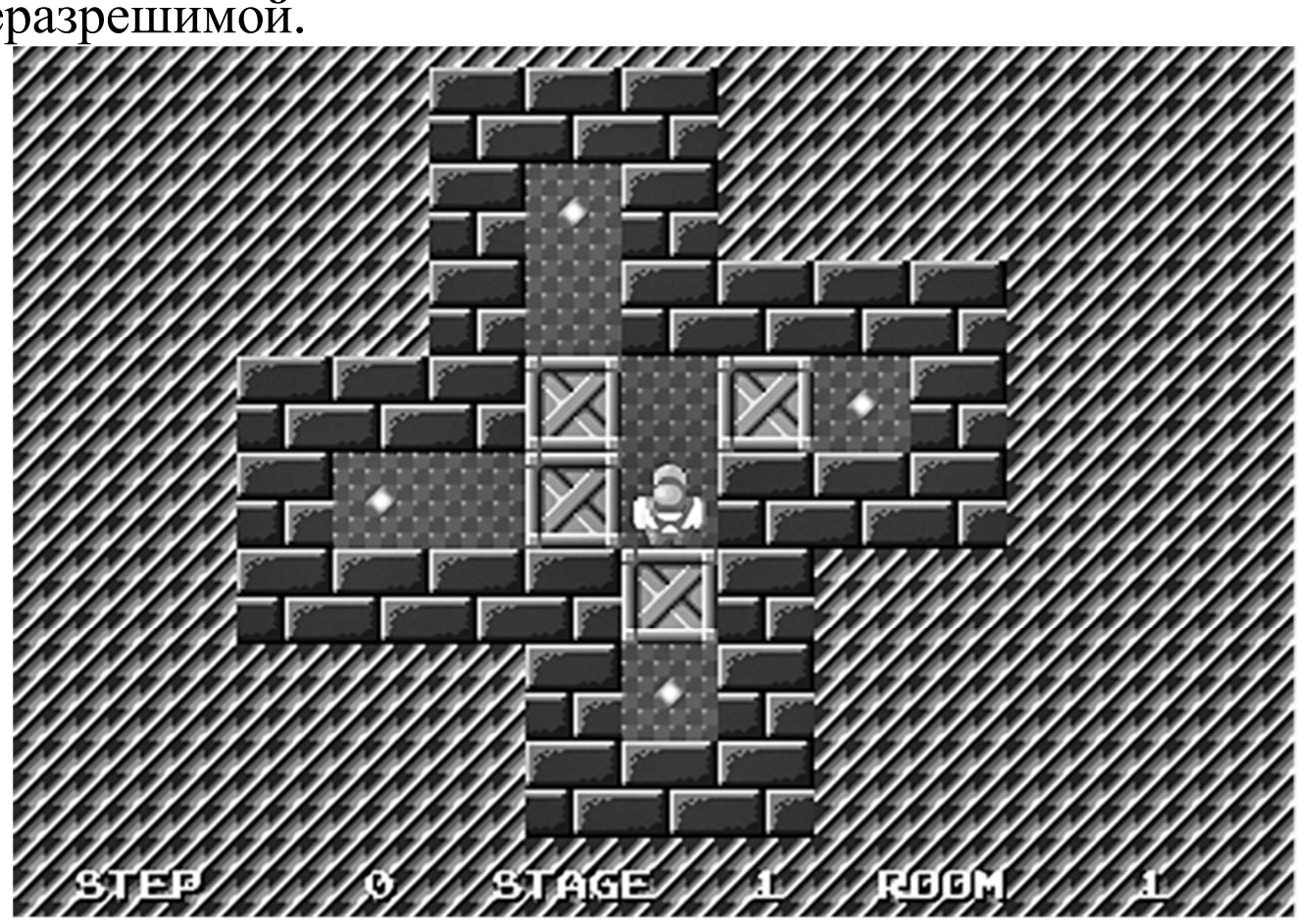
Ядро воображения состоит из *сети политики* и *модели среды*. Все самое важное происходит в модели среды. Она обучается на всех действиях, которые были выполнены агентом до настоящего момента. Получив информацию о состоянии $\hat{O}_t$, она представляет все возможные будущие исходы с учетом опыта и выбирает действие $\hat{a}_t$, обеспечивающее высокую награду.

Архитектура I2A со всеми развернутыми компонентами показана на рисунке:



Вы когда-нибудь играли в Sokoban? Это классическая головоломка, в которой игрок должен расставлять ящики по указанным местам. Правила

игры очень просты: ящики можно толкать, но нельзя тянуть. Если толкнуть ящик в ошибочном направлении, головоломка может стать неразрешимой.

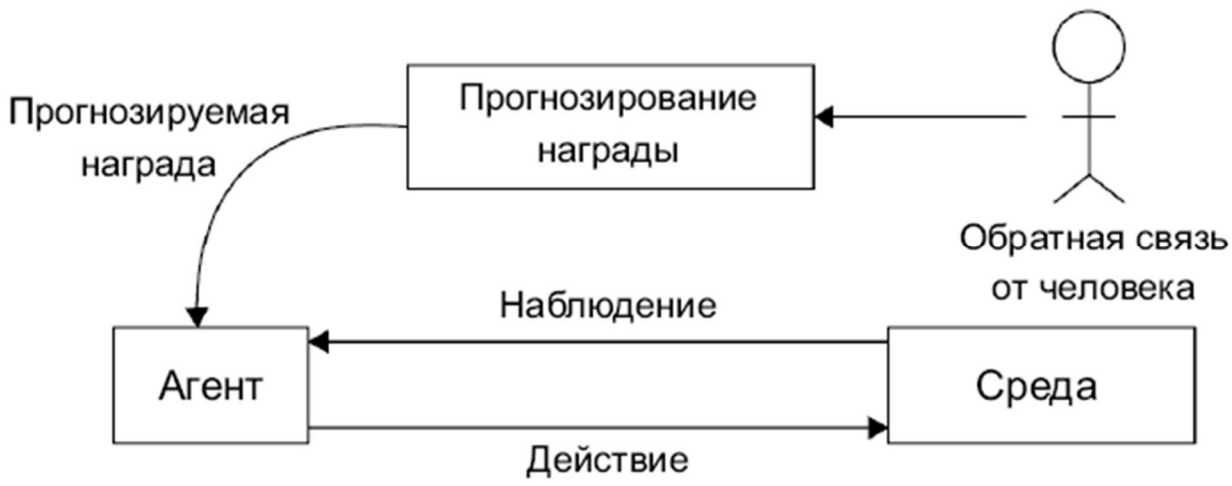


Если предложить вам сыграть в Sokoban, скорее всего, вы постараетесь представить и спланировать дальнейшие действия — ведь любой ошибочный ход может привести к концу игры. Архитектура I2A обеспечит хорошие результаты в таких средах, где агент должен планировать свои действия заранее. Авторы статьи (см. «Дополнительные источники») протестировали эффективность I2A на игре Sokoban и достигли неплохих результатов.

Обучение на человеческих предпочтениях

Обучение на человеческих предпочтениях — одно из главных достижений в области RL. Алгоритм был предложен исследователями из OpenAI и DeepMind. В его основу положена идея обучения агента в зависимости от обратной связи, получаемой от человека. Изначально агенты действуют случайным образом, после чего человеку предоставляются два видеоролика с агентом, выполняющим действие. Человек анализирует видеоклипы и сообщает агенту, какой ролик лучше, то есть в каком ролике агент лучше справляется со своей задачей. Получив обратную связь, агент пытается выполнять действия, предпочтительные для человека, и устанавливает награду соответствующим образом. Разработка функций наград — одна из самых серьезных проблем в RL, так что прямое взаимодействие агента с человеком помогает справиться с этой проблемой и свести к минимуму необходимость в написании сложных целевых функций.

Процесс обучения изображен на рисунке:



Основная последовательность обучения:

1. **Агент** взаимодействует со **средой** по случайной политике.

2. Поведение агента при взаимодействии со **средой**сохраняется в паре 2–3-секундных видеороликов, которые предоставляются человеку.

3. Человек просматривает ролики и определяет, в каком из них агент действует более эффективно. Результат передается модулю прогнозирования награды.

4. Агент получает сигналы о спрогнозированной награде и приводит свои функции цели и награды в соответствие с обратной связью, полученной от человека.

Траектория представляет собой последовательность наблюдений и действий. Обозначим сегмент траектории σ; таким образом, σ = ((o_0, a_0), (o_1, a_1), (o_2, a_2), …(o_{k-1}, a_{k-1})), где *o* — наблюдение, а *a* — действие. Агенты получают наблюдение от среды и выполняют некоторое действие. Допустим, эта последовательность взаимодействий сохраняется в двух сегментах траектории σ_1 и σ_2. Теперь эти две траектории показываются человеку. Если человек отдает предпочтение σ_2 перед сегментом σ_1 (а цель агента — произвести траектории, предпочтительные для человека), функция награды будет настроена соответствующим образом. Эти сегменты траектории сохраняются в базе данных в виде (σ_1,σ_2,μ); если человек отдает предпочтение σ_2 перед σ_1, то μ настраивается так, чтобы сегмент σ_2 стал предпочтительным. Если ни одна из траекторий не является предпочтительной, то оба варианта удаляются из базы данных. Если оба варианта предпочтительны, то μ настраивается для равноценного выбора.

Чтобы понять, как работает алгоритм, вы можете обратиться к нашему видеоролику по адресу `https://youtu.be/oC7Cw3fu3gU`.

Глубокое Q-обучение на примере демонстраций

Вы много узнали о DQN. Мы начали с базовой разновидности DQN, а затем рассмотрели различные усовершенствования: двойную DQN, дуэльную сетевую архитектуру и приоритетное воспроизведение опыта. Также вы научились строить DQN для игр Atari. Взаимодействия агента со средой сохранялись в буфере опыта, а агент использовал этот опыт для обучения. Однако проблема заключалась в том, что повышение эффективности требовало слишком долгого обучения. Для обучения в моделируемых средах это нормально, но при обучении в реальном мире

такой подход создает серьезные проблемы. Для их решения исследователи из компании DeepMind, принадлежащей Google, предложили улучшение DQN, называемое *глубоким Q-обучением на примере демонстраций*(*DQfd*, Deep Q learning from demonstrations).

Если у вас уже имеются демонстрационные данные, вы можете напрямую добавить их в буфер воспроизведения опыта. Например, представьте агента для игр Atari. Располагая демонстрационными данными о том, какое состояние лучше и какое действие предоставляет хорошую награду в состоянии, агент может напрямую использовать их для обучения. Даже небольшая демонстрация повысит эффективность обучения агента и уменьшит время обучения. Так как демонстрируемые данные будут напрямую добавлены в приоритетный буфер воспроизведения опыта, объемы демонстрационных данных и данных собственного опыта агента будут определяться и классифицироваться по приоритету.

Функции потери в DQfd представляют собой сумму различных потерь. Для предотвращения переобучения (overfitting) по демонстрационным данным мы вычисляем регуляризированные потери L2 по сетевым весам. Мы вычисляем потери TD как обычно, а также контролируемые потери, чтобы увидеть, как проходит обучение агента на демонстрационных данных. Авторы этой статьи экспериментировали с DQfd и различными средами; эффективность DQfd была лучше, чем у приоритетной дуэльной двойной DQN.

Видеоролик по ссылке показывает, как DQfd учится играть в Private Eye: `https://youtu.be/4IFZvqBHsFY`.

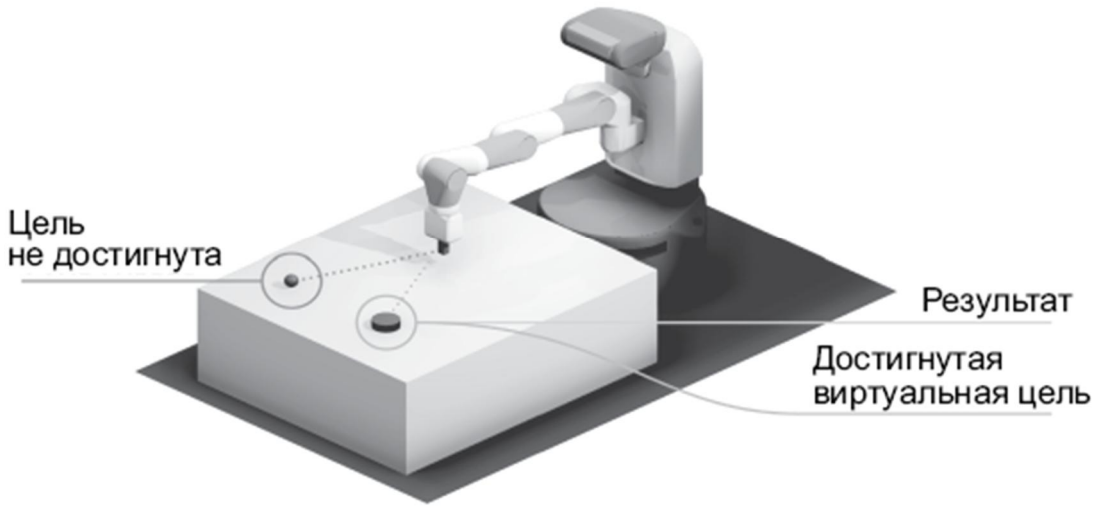
Ретроспективное воспроизведение опыта

Вы уже видели, как воспроизведение опыта используется в DQN для предотвращения корреляции данных. Также вы знаете, что приоритетное воспроизведение лучше простого воспроизведения опыта, потому что оно назначает приоритеты в зависимости от погрешности TD. А теперь мы рассмотрим новый метод, называемый *ретроспективным воспроизведением опыта* (*HER*, hindsight experience replay), который был предложен исследователями из OpenAI для решения проблемы разреженных наград. Помните, как вы учились ездить на велосипеде? При первой попытке вы не могли нормально балансировать. Возможно, вы несколько раз упали в процессе обучения. Но все эти неудачи не означали, что вы ничему не научились. Вы узнали, как *не следует* делать. Даже при том, что вы не научились ездить на велосипеде (цель), вы выполнили другую цель, а именно, вы узнали, как не следует действовать. А ведь мы, люди, учимся именно так — на своих ошибках. Эта идея заложена в основу ретроспективного воспроизведения опыта.

Рассмотрим пример. Обратимся к среде FetchSlide, представленной на рисунке; в этой среде требуется переместить манипулятор и толкнуть шайбу по столу так, чтобы она попала в цель — маленький красный

кружок (источник: `https://blog.openai.com/ingredients-for-robotics-research/`):

В нескольких первых испытаниях агент определенно не мог добиться цели. По этой причине он получал награду –1; это означало, что он делает все не так и цель не достигнута:



Но это не означает, что агент ничему не научился. Агент достиг другой цели — то есть узнал, как перейти ближе к фактической цели. Итак, вместо того чтобы считать этот результат неудачей, мы считаем, что он достиг другой цели. Если повторить этот процесс в нескольких итерациях, агент научится достигать фактической цели. Метод HER может применяться к алгоритмам без привязки к политике. Согласно сравнению, у DDPG с HER схождение достигается быстрее, чем у DDPG без HER. Эффективность HER продемонстрирована в следующем видеоролике: `https://youtu.be/Dz_HuzgMxzo`.

Иерархическое обучение с подкреплением

Недостаток обучения с подкреплением (RL) заключается в том, что оно плохо масштабируется для большого количества пространств состояний и действий, что в конечном счете порождает «проклятие размерности». Для его преодоления был предложен метод *иерархического обучения с подкреплением* (*HRL*, hierarchical reinforcement learning), разделяющий большие задачи на меньшие подзадачи. Допустим, цель агента — добраться из школы до дома. Задача разбивается на набор подзадач: выйти из школьных ворот, сесть в автобус и т.д.

В HRL используются разные методы, в числе которых декомпозиция пространств состояний, абстрагирование состояний и временное абстрагирование. *Декомпозиция пространств состояний*означает их

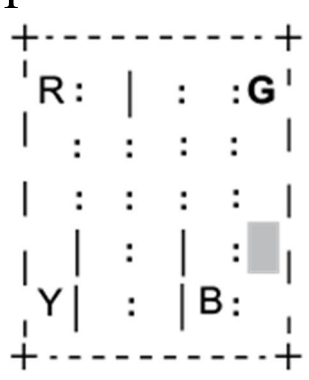
деление на несколько подпространств, в которых мы будем решать задачи. Разбиение пространства состояния также ускоряет исследование, так как агенту не нужно изучать все пространство состояний.
В *абстрагировании состояний* агент игнорирует переменные, не релевантные для достижения текущих подзадач в текущем пространстве состояний. При *временном абстрагировании*последовательность действий и наборы действий группируются, в результате чего один шаг делится на несколько шагов.

Теперь мы можем рассмотреть один из самых распространенных алгоритмов HRL, называемый декомпозицией функции ценности MAXQ.

Декомпозиция функции ценности MAXQ

Декомпозиция функции ценности MAXQ— один из часто используемых алгоритмов в HRL. Разберемся, как он работает. Функция ценности раскладывается на набор функций ценности для каждой из подзадач. Рассмотрим пример, уже приводившийся ранее. Помните задачу о такси, которую мы решали с использованием *Q*-обучения и SARSA?

Имеются четыре точки, агент должен забирать пассажиров в одной точке и высаживать их в другой. Агент получает +20 очков награды за успешную высадку пассажира и –1 очко за каждый временной шаг. Агент также теряет 10 очков за неправильные посадки и высадки. Цель агента — научиться забирать и высаживать пассажиров в правильных местах за короткое время без посадки неправильных пассажиров. Среда показана на следующем рисунке; буквами (**R**, **G**, **Y**, **B**) обозначены разные точки, а маленький прямоугольник представляет такси, которым управляет агент.



Цель разбивается на четыре подзадачи.

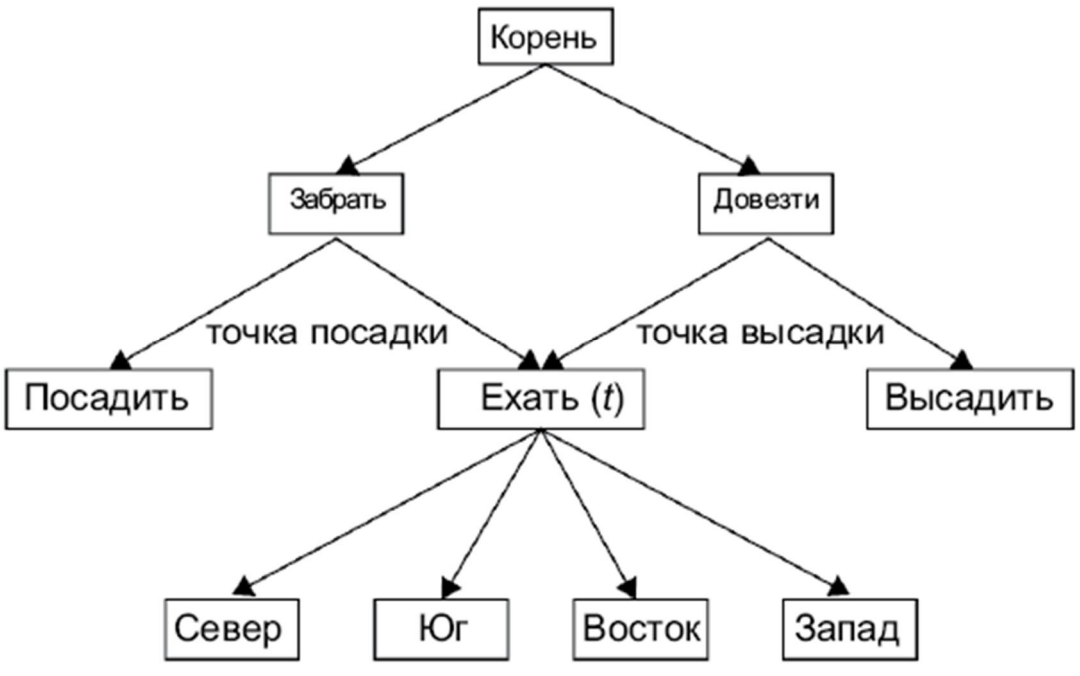
• *Ехать*: цель — перемещение такси от текущей позиции к одной из целевых точек. Подзадача *Ехать*(*t*) должна использовать четыре примитивных действия: перемещение на север, юг, восток или запад.

• *Забрать*: цель — перемещение такси от текущей позиции к позиции пассажира и посадка пассажира.

• *Довезти*: цель — перемещение такси от текущей позиции к конечной точке пассажира и высадка пассажира.

• *Корень*: общая задача.

Все эти подзадачи можно представить на направленном ациклическом графе, который называется *графом задачи*.



На приведенном рисунке видно, что все подзадачи имеют иерархическую структуру. Каждый узел представляет подзадачу, или *примитивное действие*, а каждое ребро представляет способ вызова одной подзадачей ее дочерней подзадачи.

Подзадача *Exать(t)* содержит четыре примитивных действия: восток, запад, север и юг. Подзадача *Забрать* содержит примитивное действие *Посадить* и подзадачу *Exать*; аналогичным образом дело обстоит с подзадачей *Довезти*, которая содержит примитивное действие *Высадить* и подзадачу *Exать*.

В декомпозиции MAXQ MDP *M*делится на набор задач (M_0, M_1, M_2, …, M_n), где M_0 — корневая задача, а M_1, M_2,…, M_n — подзадачи.

Подзадача M_i определяет полу-MDP с состояниями S_i, действиями A_i, функцией вероятности перехода $P_i^{\pi}(s', N \mid s, a)$ и функцией ожидаемой награды $\overline{R}(s, a) = V^{\pi}(a, s)$, где $V^{\pi}(a, s)$ — проекция функции ценности для дочерней задачи M_a в состоянии s.

Если действие a является примитивным действием, мы можем определить $V^{\pi}(a, s)$ как ожидаемую немедленную награду от выполнения действия a в состоянии s:

$$V^{\pi}(a, s) = \sum_{s'} P(s' \mid s, a).\, R(s' \mid s, a).$$

Теперь мы можем переписать приведенную функцию ценности в форме уравнения Беллмана:

$$V^{\pi}(i, s) = V^{\pi}(\pi_i(s), s) + \sum_{s', N} P_i^{\pi}(s', N \mid s, \pi_i(s)) \gamma^N V^{\pi}(i, s'). \quad \textbf{(13.1)}$$

Обозначим функцию ценности состояния/действия Q следующим образом:

$$Q^{\pi}(i, s, a) = V^{\pi}(a, s) + \sum_{s', N} P_i^{\pi}(s', N \mid s, a) \gamma^N Q^{\pi}(i, s', \pi(s')). \quad \textbf{(13.2)}$$

Теперь определим еще одну функцию, которая называется *функцией завершения*, — ожидаемую скорректированную накопленную награду за завершение подзадачи M_i:

$$C^{\pi}(i, s, a) = \sum_{s', N} P_i^{\pi}(s', N \mid s, a) \gamma^N Q^{\pi}(i, s', \pi(s')). \quad \textbf{(13.3)}$$

Для уравнений **(13.2)** и **(13.3)**функцию Q можно записать в следующем виде:

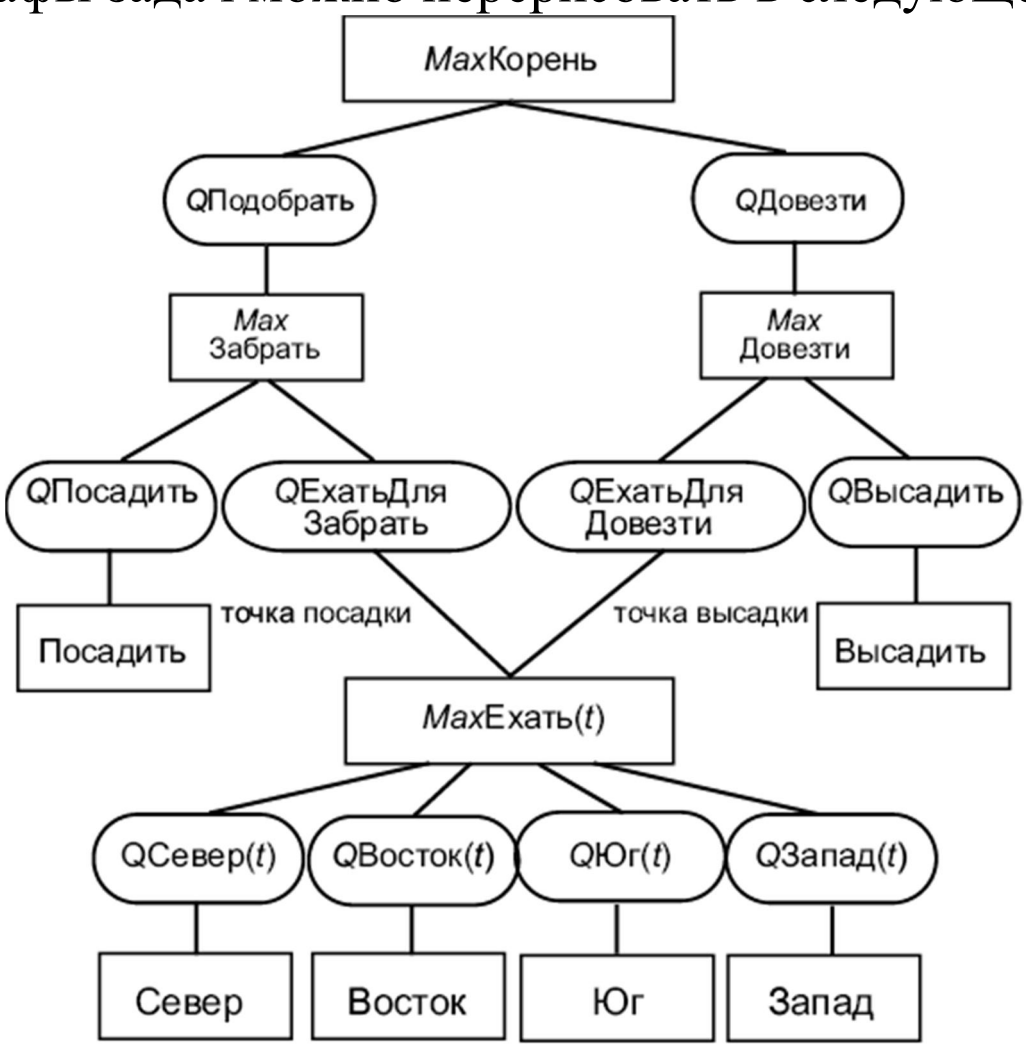
$$Q^{\pi}(i, s, a) = V^{\pi}(a, s) + C^{\pi}(i, s, a).$$

Наконец, функцию ценности можно переопределить в виде

$$V^{\pi}(i, s) = \begin{cases} Q^{\pi}(i, s', \pi_i(s')) & \text{для составных } i \\ \sum_{s'} P(s' | s, i) R(s' | s, i) & \text{для примитивных } i. \end{cases}$$

Предыдущие уравнения осуществляют декомпозицию функции ценности корневой задачи на функции ценности отдельных подзадач.

Для эффективного проектирования и отладки декомпозиций MAXQ графы задач можно перерисовать в следующем виде:



Перестроенный граф содержит узлы двух специальных типов: максимальные узлы и Q-узлы. Максимальные узлы определяют подзадачи из декомпозиции, а Q-узлы — действия, доступные для каждой подзадачи.

Инвертированное обучение с подкреплением

Что же происходит в RL? Мы пытаемся найти оптимальную политику для заданной функции награды. Инвертированное обучение с подкреплением идет в обратном направлении: задается оптимальная политика, и для нее необходимо найти функцию награды. Но в чем польза инвертированного обучения с подкреплением? Разработка функции награды — нетривиальная задача, а плохая функция награды приведет к плохому поведению агента. Мы не всегда знаем подходящую функцию награды заранее, но иногда знаем правильную политику (то есть правильное действие в каждом состоянии).

Таким образом, оптимальная политика передается агенту человеком-экспертом, а агенты пытаются определить функцию награды. Для примера возьмем агента, который обучается ходить в реальной среде;

трудно спроектировать функцию награды для всех действий, которые он может выполнить. Вместо этого можно передать агентам демонстрации (применения оптимальной политики) от эксперта, а агенты попытаются вычислить по ним функцию награды.

В области RL происходят различные усовершенствования и достижения. Теперь, когда вы завершили чтение книги, вы можете переходить к самостоятельному изучению RL и экспериментам с разными проектами. Обучайтесь и подкрепляйте!

Итоги

В этой главе были рассмотрены некоторые недавние достижения в области RL. Вы увидели, как архитектура I2A использует ядро воображения для планирования на будущее и как агенты обучаются на человеческих предпочтениях. Также вам был представлен алгоритм DQfd, который повышает эффективность и сокращает время обучения DQN за счет обучения на демонстрациях. Затем было рассмотрено ретроспективное воспроизведение опыта, при котором агенты учатся на своих неудачах.

Вы узнали об иерархическом обучении с подкреплением, осуществляющем декомпозицию цели на иерархию подцелей, и об инвертированном обучении, при котором агенты пытаются вычислить функцию награды по политике. RL постоянно развивается, и с каждым днем появляются новые интересные достижения. Теперь, когда вы освоили различные алгоритмы обучения с подкреплением, вы сможете строить агентов для выполнения различных задач, а также вносить свой творческий вклад в исследования в области RL.

Вопросы

1. Что собой представляет «воображение» у агента?
2. Что такое «ядро воображения»?
3. Как агенты учатся на человеческих предпочтениях?
4. Чем DQfd отличается от DQN?
5. Что такое «ретроспективное воспроизведение опыта»?
6. Для чего применяется иерархическое обучение с подкреплением?
7. В чем особенность инвертированного обучения с подкреплением?

Дополнительные источники

Статья об I2A: `https://arxiv.org/pdf/1707.06203.pdf`.

Статья о DRL на человеческих предпочтениях: `https://arxiv.org/pdf/1706.03741.pdf`.

Статья о HER: `https://arxiv.org/pdf/1707.01495.pdf`.

Безопасность ИИ: `https://arxiv.org/pdf/1805.00899.pdf`.

Ответы

Глава 1

1. *Обучение с подкреплением* (*RL*, reinforcement learning) — область машинного обучения, в которой обучение базируется на взаимодействии со средой.

2. RL работает методом проб и ошибок, в отличие от других парадигм машинного обучения.

3. *Агенты* — программы, которые принимают разумные решения; являются обучаемой стороной в RL.

4. Функция политики указывает, какое действие следует выполнять в каждом состоянии, а функция ценности определяет, насколько хорошо для агента пребывание в конкретном состоянии.

5. В обучении с моделью агент использует предыдущий опыт, тогда как при обучении без модели предыдущий опыт отсутствует.

6. Детерминированные, стохастические, с полной информацией, с неполной информацией, дискретные, непрерывные, эпизодические и неэпизодические среды.

7. OpenAI Universe предоставляет полнофункциональные среды для обучения агентов RL.

8. См. раздел *«Практическое применение RL»*.

Глава 2

1. `conda create --name universepython=3.6 anaconda`

2. Docker позволяет упаковывать приложения вместе с их зависимостями в контейнерах, чтобы приложения можно было просто запускать на сервере без использования внешних зависимостей из контейнеров Docker.

3. `gym.make(env_name)`

4. `from gym import envs`

5. `print(envs.registry.all())`

6. *OpenAI Universe* — расширение OpenAI Gym, также предоставляющее различные полнофункциональные среды.

7. *Заместитель* используется для получения внешних данных, а *переменные* — для хранения значений.

8. Все в TensorFlow представляется в виде графов вычислений, состоящих из узлов и ребер. *Узлы* представляют математические операции (допустим, сложение, умножение и т.д.), а *ребра* — тензоры.

9. Определение вычислительного графа не приводит к его выполнению; чтобы выполнить граф, используйте сеансы TensorFlow.

Глава 3

1. Марковское свойство гласит, что будущее зависит только от настоящего, но не от прошлого.

2. Марковский процесс принятия решений (MDP) является расширением марковских цепей. Он формирует математическую основу

моделирования ситуаций с принятием решений. Почти все задачи обучения с подкреплением могут быть смоделированы на базе MDP.

3. См. раздел «Поправочный коэффициент».

4. Поправочный коэффициент определяет относительную важность будущих и немедленных наград.

5. Уравнение Беллмана используется для решения задач MDP .

6. См. раздел «Вывод уравнения Беллмана для функции ценности и *Q*-функции».

7. Функция ценности определяет полезность самого состояния, а *Q*-функция определяет полезность действия в этом состоянии.

8. См. разделы «Итерация по ценности» и «Итерация по политикам».

Глава 4

1. Алгоритм Монте-Карло используется в RL в тех случаях, когда модель среды неизвестна.

2. См. раздел «Оценка значения π методом Монте-Карло».

3. При прогнозировании методом Монте-Карло функция ценности аппроксимируется средним возвратом вместо ожидаемого возврата.

4. В методах Монте-Карло возврат усредняется при каждом посещении состояния в эпизоде. В методе Монте-Карло с первым посещением возврат усредняется только при первом посещении состояния в эпизоде.

5. См. раздел «Управление методом Монте-Карло».

6. См. разделы «Метод Монте-Карло с привязкой к политике» и «Метод Монте-Карло без привязки к политике».

7. См. раздел «Игра в блек-джек по стратегии Монте-Карло».

Глава 5

1. Метод Монте-Карло применим только к эпизодическим задачам, тогда как обучение TD может применяться как к эпизодическим, так и к неэпизодическим задачам.

2. Разность между фактическим и прогнозируемым значением называется TD-погрешностью.

3. См. разделы «Прогнозирование на основе временных различий» и «TD-управление».

4. См. раздел «Решение задачи о такси с использованием *Q*-обучения».

5. В *Q*-обучении действие выбирается с использованием эпсилон-жадной стратегии, а при обновлении *Q* просто выбирается максимальное действие. В SARSA действие выбирается с применением эпсилон-жадной стратегии, а при обновлении *Q* эпсилон-жадная стратегия используется для выбора действия.

Глава 6

1. *Многорукий бандит* — игровой автомат; азартная игра, в которой игрок нажимает на рычаг и получает награду (выигрыш) на основании

случайно сгенерированного распределения вероятностей. Отдельный автомат называется «одноруким бандитом»; если же таких автоматов несколько, это называется «многоруким бандитом», или *k*-руким бандитом.

2. Дилемма компромисса между исследованием и эксплуатацией возникает тогда, когда агент не уверен в том, следует ли исследовать новые действия или эксплуатировать лучшее действие с использованием предыдущего опыта.

3. Значение epsilon используется для принятия решения о том, чем следует заняться агенту: исследованием или эксплуатацией. С вероятностью 1 – epsilon выбирается лучшее действие, а с вероятностью epsilon исследуется новое действие.

4. Для разрешения дилеммы компромисса между исследованием и эксплуатацией используются различные алгоритмы: эпсилон-жадная политика, softmax-исследование, алгоритм верхней границы доверительного интервала, алгоритм выборки Томпсона.

5. Алгоритм UCB помогает выбрать лучшую руку на основании доверительного интервала.

6. При выборке Томпсона используется оценка на основе априорного распределения, а в UCB используется доверительный интервал.

Глава 7

1. Нейроны добавляют в результат znелинейность за счет применения функции *f*(), называемой *функцией активации*, или *передаточной функцией*. См. раздел «Искусственные нейроны».

2. Функции активации используются для введения нелинейности в нейросети.

3. Градиент функции стоимости вычисляется по отношению к весам для минимизации ошибки.

4. RNN прогнозирует выход на основании не только текущего входа, но и предыдущего скрытого состояния.

5. Если при обратном распространении сети значение градиентов становится все меньше и меньше, это называется *проблемой исчезающего градиента*; если значение градиента увеличивается, это называется *проблемой взрывного градиента*.

6. Шлюзы — специальные структуры, используемые в LSTM для принятия решений о том, какую информацию следует сохранить, стереть или обновить.

7. Уровень подвыборки используется для сокращения размеров карт признаков и объема вычислений, сохраняя минимум необходимых подробностей.

Глава 8

1. *DQN* (Deep Q network) — нейросеть, используемая для аппроксимации *Q*-функции.

2. Воспроизведение опыта используется для устранения корреляций между обучающими данными.

3. Если прогнозируемое и целевое значения вычисляются одной сетью, между ними может существовать расхождение, поэтому мы используем отдельную *целевую сеть*.

4. Из-за оператора `max` DQN переоценивает значение *Q*.

5. Благодаря двум разным независимо обучаемым *Q*-функциям, двойные DQN избегают переоценки значений *Q*.

6. В приоритетном воспроизведении опыту назначается приоритет, основанный на TD-погрешности.

7. Дуэльная сеть DQN оценивает значение *Q* разбиением вычисления *Q*-функции на вычисления функции ценности и функции преимущества.

Глава 9

1. DRQN использует рекуррентные нейросети (RNN), тогда как DQN применяет базовую нейросеть.

2. DQN не используется для MDP с неполной информацией.

3. См. раздел «Doom с DRQN».

4. В отличие от DRQN, DARQN использует механизм внимания.

5. DARQN используется для сосредоточения на более важной области игрового экрана.

6. Мягкий и жесткий механизмы внимания.

7. Если присвоить `living_reward`значение `0`, агент будет награждаться за каждое перемещение, даже если оно не приносит пользы.

Глава 10

1. *A3C* — асинхронная преимущественная сеть «актор-критик» — использует нескольких агентов для параллельного обучения.

2. A3C требует меньших затрат вычислительных ресурсов и времени обучения по сравнению с DQN.

3. Работники используют собственные копии среды, после чего глобальная сеть обобщает их опыт.

4. Энтропия обеспечивает достаточный объем исследований.

5. См. раздел «Как работает A3C».

Глава 11

1. Градиент политики — один из замечательных алгоритмов в области RL, который напрямую оптимизирует политику, параметризованную по некоторому параметру.

2. Градиенты политики эффективно работают в ситуациях, в которых не нужно вычислять *Q*-функцию для нахождения оптимальной политики.

3. Сеть актора должна определить лучшие действия в состоянии посредством настройки параметра, а сеть критика должна оценить действие, сгенерированное актором.

4. См. раздел «Оптимизация политики доверительной области».

5. Мы применяем итеративное улучшение политики и устанавливаем ограничение, согласно которому расстояние Кульбака—Лейбнера (*KL*) между старой и новой политикой должно быть меньше некоторой константы. Это ограничение называется ограничением доверительной области.

6. PPO изменяет целевую функцию TRPO, заменяя ограничение штрафной составляющей для предотвращения попыток вычисления сопряженных градиентов.

Глава 12

1. DQN вычисляет значение *Q*напрямую, тогда как дуэльная DQN разбивает вычисление значения *Q* на функцию ценности и функцию преимущества.

2. См. раздел «Память воспроизведения».

3. Если одна сеть вычисляет как прогнозируемое, так и целевое значение, между ними может существовать расхождение, поэтому мы используем отдельную *целевую сеть*.

4. См. раздел «Память воспроизведения».

5. См. раздел «Дуэльная сеть».

6. Дуэльная DQN разбивает вычисление значения *Q* на функцию ценности и функцию преимущества, а двойная DQN использует две *Q*-функции для предотвращения переоценки.

7. См. раздел «Дуэльная сеть».

Глава 13

1. Воображение обеспечивает планирование на будущее перед выполнением каких-либо действий.

2. Ядро воображения состоит из *сети политики* и *модели среды*.

3. Агенты многократно получают обратную связь от человека и изменяют свою цель в зависимости от человеческих предпочтений.

4. DQfd использует демонстрационные данные для обучения, тогда как DQN изначально не располагает никакими данными.

5. См. раздел «Ретроспективное воспроизведение опыта».

6. Для преодоления «проклятия размерности» был предложен метод иерархического обучения с подкреплением (HRL), использующий иерархию, в которой большие задачи разделены на меньшие подзадачи.

7. В RL вы пытаетесь найти оптимальную политику для заданной функции награды, а в инвертированном обучении с подкреплением задается оптимальная политика и для нее необходимо найти функцию награды.