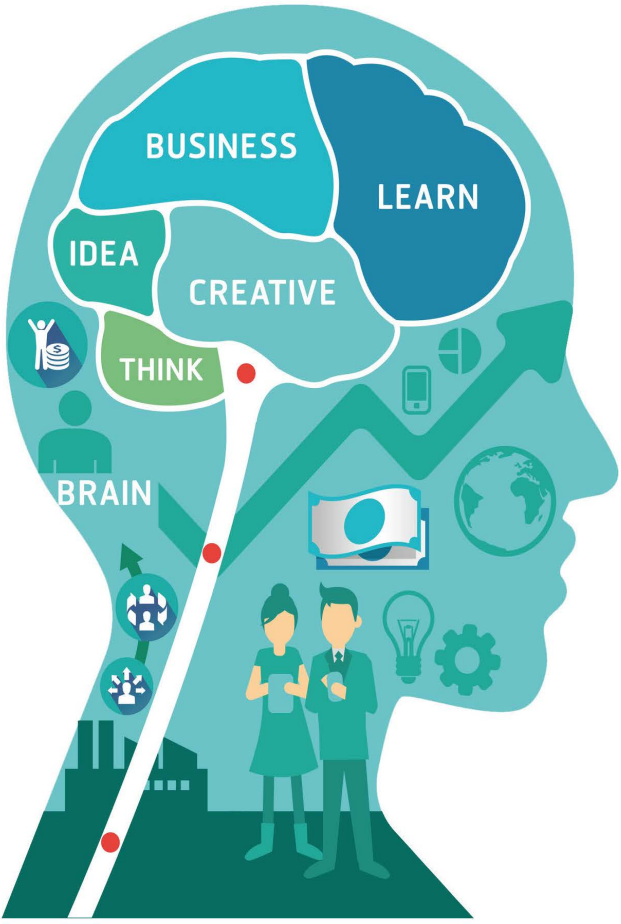
Обучение с подкреплением на PyTorch Сборник рецептов



Юси Лю

Обучение с подкреплением на PyTorch

Сборник рецептов

Юси (Хэйден) Лю

Обучение с подкреплением на PyTorch: сборник рецептов

PyTorch 1.x Reinforcement Learning Cookbook

Over 60 recipes to design, develop, and deploy self-learning AI models using Python

Yuxi (Hayden) Liu

Packt

BIRMINGHAM – MUMBAI

Обучение с подкреплением на PyTorch: сборник рецептов

Свыше 60 рецептов проектирования, разработки и развертывания самообучающихся моделей на Python

Юси (Хэйден) Лю

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
Москва, 2020

УДК 004.85
ББК 32.971.3
Л93

Лю Ю. (Х.)
Л93 Обучение с подкреплением на PyTorch: сборник рецептов / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2020. – 282 с.: ил.

ISBN 978-5-97060-853-1

Библиотека PyTorch выходит на передовые позиции в качестве средства обучения с подкреплением (ОП) благодаря эффективности и простоте ее использования. Эта книга организована как справочник по работе с PyTorch, охватывающий широкий круг тем – от самых азов (настройка рабочей среды) до практических задач (рассмотрение ОП на конкретных примерах).

Вы научитесь использовать алгоритм «многоруких бандитов» и аппроксимацию функций; узнаете, как победить в играх Atari с помощью глубоких Q-сетей и как эффективно реализовать метод градиента стратегии; увидите, как применить метод ОП к игре в блэкджек, к окружающим средам в сеточном мире, к оптимизации рекламы в интернете и к игре Flappy Bird.

Издание предназначено для специалистов по искусственному интеллекту, которым требуется помощь в решении задач ОП. Для изучения материала необходимо знакомство с концепциями машинного обучения; опыт работы с библиотекой PyTorch необязателен, но желателен.

УДК 004.85
ББК 32.971.3

ISBN 978-1-83855-196-4 (англ.)
ISBN 978-5-97060-853-1 (рус.)

Содержание

Об авторе

Юси (Хэйден) Лю – опытный специалист по обработке данных, специализирующийся на разработке моделей и систем машинного и глубокого обучения. Он работал в различных предметных областях, применяя свои познания в обучении с подкреплением. С удовольствием преподает и является автором ряда книг по машинному обучению. Его первая книга «Python Machine Learning By Example» была бестселлером Amazon в Индии в 2017 и 2018 годах. Его перу принадлежат также книги «R Deep Learning Projects» и «Hands-On Deep Learning Architectures with Python», опубликованные издательством Packt. Во время работы над магистерской диссертацией в Торонтском университете написал пять работ, опубликованных в изданиях IEEE и сборниках докладов на конференциях.

О рецензентах

Грег Уолтерс занимается компьютерами и программированием с 1972 года. Отлично владеет языками Visual Basic, Visual Basic .NET, Python и SQL (диалектами MySQL, SQLite, Microsoft SQL Server, Oracle), C++, Delphi, Modula-2, Pascal, C, ассемблером 80x86, COBOL и Fortran. Обучает программированию, через его руки прошло множество людей, которых он учил таким продуктам, как MySQL, Open Database Connectivity, Quattro Pro, Corel Draw!, Paradox, Microsoft Word, Excel, DOS, Windows 3.11, Windows for Workgroups, Windows 95, Windows NT, Windows 2000, Windows XP и Linux. Сейчас на пенсии и в свободное время музицирует и обожает готовить, но всегда готов поработать фрилансером над разными проектами.

Роберт Мони работает над докторской диссертацией в Будапештском университете технологии и экономики (BME), а также является экспертом по глубокому обучению в Континентальном центре компетенций по глубокому обучению в Будапеште. Руководит проектом, направленным на поддержку студенческих исследований в области глубокого обучения и разработки беспилотных автомобилей. Тема его исследований – глубокое обучение с подкреплением в сложных окружающих средах, а конечная цель – применение этой технологии к беспилотным транспортным средствам.

Предисловие

Всплеск интереса к обучению с подкреплением (ОП) объясняется тем, что это революционный подход к автоматизации посредством обучения тому, какие действия следует предпринимать в окружающей среде, чтобы максимизировать полное вознаграждение.

Эта книга представляет собой введение в важные концепции обучения с подкреплением и реализации его алгоритмов с применением библиотеки PyTorch. В каждой главе рассматривается какой-то один метод ОП и его применения в промышленности. Рецепты, содержащие практические примеры, помогут вам обогатить свои знания и навыки в области ОП, в том числе динамическое программирование, методы Монте-Карло, методы на основе временных различий, Q-обучение, решение задачи о многоруком бандите, аппроксимация функций, глубокие Q-сети, методы градиента стратегии. Интересные и легкие для усвоения примеры – игры Atari, блэкджек, сеточный мир, реклама в интернете, машина на горе, игра Flappy Bird – не позволят вам заскучать.

Прочитав книгу, вы будете уверенно владеть распространенными алгоритмами обучения с подкреплением и научитесь применять их к решению различных практических задач.

Предполагаемая аудитория

Специалисты по машинному обучению, по обработке данных и искусственному интеллекту, которым нужна помощь в решении задач ОП. Предполагается предварительное знакомство с концепциями машинного обучения, опыт работы с библиотекой PyTorch необязателен, но желателен.

Структура книги

Глава 1 «Приступаем к обучению с подкреплением и PyTorch» – отправная точка, с которой начинается путешествие в мир обучения с подкреплением и PyTorch. Мы настроим рабочую среду и OpenAI Gym и познакомимся с окружающими средами для экспериментов с ОП, включая CartPole и игры Atari. Здесь же будет рассмотрена реализация таких базовых алгоритмов, как случайный поиск, восхождение на вершину и градиент стратегии. В конце главы будет дан краткий обзор PyTorch.

Глава 2 «Марковский процесс принятия решений и динамическое программирование» начинается с создания марковской цепи и марковского процесса принятия решений (МППР) – понятия, которое лежит в основе большинства алгоритмов обучения с подкреплением. Затем мы рассмотрим два подхода

к решению МППР – итерация по ценности и итерация по стратегиям. Мы ближе познакомимся с МППР и уравнением Беллмана, попрактиковавшись в оценивании стратегии. Также будет продемонстрировано решение интересной игры с подбрасыванием монеты. И в конце мы покажем, как с помощью динамического программирования масштабировать обучение.

Глава 3 «Применение методов Монте-Карло для численного оценивания» посвящена методам Монте-Карло. Для начала мы оценим, чему равно число π. Затем рассмотрим алгоритм с единой стратегией – управление методом Монте-Карло первого посещения – и несколько алгоритмов с разделенной стратегией на основе методов Монте-Карло. Также будут рассмотрены ε-жадная стратегия и взвешенная выборка по значимости.

Глава 4 «TD-обучение и Q-обучение» начинается с подготовки двух окружающих сред: блуждание на краю обрыва и ветреный сеточный мир, которые понадобятся для исследования обучения на основе временных различий (TD-обучения) и Q-обучения. Мы научимся выполнять предсказания с помощью TD-обучения и обсудим Q-обучение как пример алгоритма с разделенной стратегией и SARSA как пример алгоритма с единой стратегией. Мы также сформулируем задачу о такси и покажем, как ее решать с помощью алгоритмов Q-обучения и SARSA. И наконец, будет рассмотрен алгоритм двойного Q-обучения.

В главе 5 «Решение задачи о многоруком бандите» рассматривается алгоритм многорукого бандита – пожалуй, один из самых популярных в обучении с подкреплением. Мы покажем четыре подхода к решению этой задачи: ε-жадная стратегия, исследование с помощью функции softmax, алгоритм верхней доверительной границы и алгоритм на основе выборки Томпсона. Мы также поговорим о рекламе в интернете и продемонстрируем ее решение с помощью алгоритма многорукого бандита. Напоследок разработаем более сложный алгоритм контекстуального бандита и применим его к решению задачи об оптимизации показа рекламных объявлений.

Глава 6 «Масштабирование с помощью аппроксимации функций» посвящена аппроксимации. Мы начнем с подготовки окружающей среды Mountain Car. Объясним, чем аппроксимация функций лучше табличного поиска, и научимся включать аппроксимацию в уже известные алгоритмы Q-обучения и SARSA. Также будет рассмотрена техника пакетного обучения с использованием буфера воспроизведения опыта. И наконец, мы покажем, как, воспользовавшись полученными знаниями, решить задачу о балансировании стержня на тележке.

В главе 7 «Глубокие Q-сети в действии» рассматривается алгоритм глубокой Q-сети (DQN), который считается одним из наиболее передовых методов обучения с подкреплением. Мы разработаем модель DQN и объясним два принципа, лежащих в основе ее работы: буфер воспроизведения и целевая сеть. Для решения игр Atari мы покажем, как интегрировать в DQN сверточную нейронную сеть. Будут рассмотрены два варианта DQN: Double DQN и Dueling DQN. Мы также опишем точную настройку алгоритма Q-обучения, взяв в качестве примера Double DQN.

Глава 8 «Реализация методов градиента стратегии и оптимизация стратегии» посвящена методам градиента стратегии и начинается с реализации алгоритма REINFORCE. Затем мы разработаем алгоритм REINFORCE с базой для

решения задачи о блуждании на краю обрыва. Мы также реализуем алгоритм исполнитель–критик и применим его к решению той же задачи. Чтобы масштабировать детерминированный алгоритм градиента стратегии, воспользуемся приемами, заимствованными из DQN, и разработаем алгоритм глубокого детерминированного градиента стратегии. Ради интереса мы применим метод перекрестной энтропии, чтобы обучить агента балансированию стержня. И наконец, поговорим о том, как масштабировать алгоритм градиента стратегии с помощью асинхронного метода исполнитель–критик и нейронных сетей.

В главе 9 «Кульминационный проект – применение DQN к игре Flappy Bird» мы рассмотрим, как методами обучения с подкреплением можно воспользоваться в игре Flappy Bird. Мы применим все полученные знания, чтобы создать интеллектуального бота. Затем настроим параметры модели и развернем ее. И посмотрим, как долго птица сможет продержаться в воздухе.

Графические выделения

В этой книге для выделения семантически различной информации применяются различные стили. Ниже приведены примеры стилей с пояснениями.

`Код в тексте`: фрагменты кода, имена таблиц базы данных, папок и файлов, URL-адреса, данные, введенные пользователем, адреса в Twitter, например: «Слово `пустая` не означает, что значения всех элементов равны `Null`».

Отдельно стоящие фрагменты кода набраны так:

```
>>> def random_policy():
...     action = torch.multinomial(torch.ones(n_action), 1).item()
...     return action
```

Текст, который вводится на консоли или выводится на консоль, напечатан следующим образом:

```
conda install pytorch torchvision -c pytorch
```

Новые термины, важные слова и слова на экране набраны **полужирным шрифтом**. Так же выделяются элементы интерфейса, например пункты меню и поля в диалоговых окнах. Например: «Этот подход называется **случайным поиском**, потому что вес в каждом испытании выбирается случайно в надежде, что при большом числе испытаний будет найден наилучший вес».

Предупреждения и важные замечания оформлены так.

Советы и рекомендации выглядят так.

Разделы

В этой книге повторяются одни и те же заголовки разделов: *Подготовка*, *Как это делается*, *Как это работает*, *Это еще не все* и *См. также*.

Опишем их назначение.

Подготовка

В этом разделе объясняется, чего ожидать от рецепта, как подготовить программную среду и выполнить все прочие предварительные условия.

Как это делается

Выполнение рецепта по шагам.

Как это работает

Подробное объяснение того, что происходило на каждом шаге, описанном в предыдущем разделе.

Это еще не все

Дополнительная информация, относящаяся к рецепту.

См. также

Ссылки на другую полезную информацию.

Отзывы и пожелания

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

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

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте http://dmkpress.com/authors/publish_book/ или напишите в издательство: dmkpress@gmail.com.

Список опечаток

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

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

Скачивание исходного кода

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

Нарушение авторских прав

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

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

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

Глава 1

Приступаем к обучению с подкреплением и PyTorch

Мы начнем путешествие в мир обучения с подкреплением и PyTorch с простых, но важных алгоритмов: случайный поиск, восхождение на вершину и градиент стратегии. Для начала подготовим среду разработки и OpenAI Gym, чтобы для экспериментов с окружающими средами ОП можно было использовать игры Atari и CartPole. Мы также продемонстрируем пошаговую разработку алгоритмов для решения задачи о балансировании стержня. Кроме того, рассмотрим основы PyTorch и приготовимся к последующим примерам и учебным проектам.

В этой главе приводятся следующие рецепты:

- подготовка среды разработки;
- установка OpenAI Gym;
- окружающие среды Atari;
- окружающая среда CartPole;
- основы PyTorch;
- реализация и оценивание стратегии случайного поиска;
- алгоритм восхождения на вершину;
- алгоритм градиента стратегии.

Подготовка среды разработки

Прежде всего подготовим среду разработки, в т. ч. подходящие версии Python, Anaconda, а также библиотеку PyTorch, с которой будем работать на протяжении всей книги.

Python – это язык, на котором будут реализованы все алгоритмы обучения с подкреплением, описанные в этой книге. Мы будем использовать версию 3, а точнее версию 3.8 или более позднюю. Если вы по-прежнему работаете с Python 2, самое время перейти на Python 3, поскольку Python 2 после 2020 года поддерживаться не будет. Переход не сулит никаких проблем, так что не впадайте в панику.

Anaconda – это дистрибутив Python с открытым исходным кодом (www.anaconda.com/distribution/), специально предназначенный для применения в науке о данных и машинном обучении. Для установки Python-пакетов мы будем использовать входящий в Anaconda менеджер пакетов `conda`, а также программу `pip`.

PyTorch (https://pytorch.org/) – современная библиотека машинного обучения, разработанная подразделением Facebook по исследованиям в области искусственного интеллекта (FAIR) на основе каркаса Torch (http://torch.ch/). В PyTorch вместо массивов NumPy (`ndarray`) используются тензоры, обладающие большей гибкостью и совместимостью с графическими процессорами. Привлеченное широкими возможностями графов вычислений, а также простым и дружественным интерфейсом, сообщество PyTorch ежедневно растет, а библиотеку берут на вооружение все новые и новые технологические гиганты.

Теперь посмотрим, как установить и настроить все эти компоненты.

Как это делается

Начнем с установки Anaconda. Можете пропустить этот раздел, если в вашей системе уже установлен дистрибутив Anaconda для Python 3.6 или 3.7. В противном случае следуйте опубликованным на странице https://docs.anaconda.com/anaconda/install/ инструкциям для своей операционной системы:

- Installing on Windows
- Installing on macOS
- Installing on Linux

Чтобы проверить правильность установки Anaconda и Python, введите в окне терминала в Linux/Mac или в окне командной строки в Windows (начиная с этого места будем употреблять общее название – терминал) команду

```
python
```

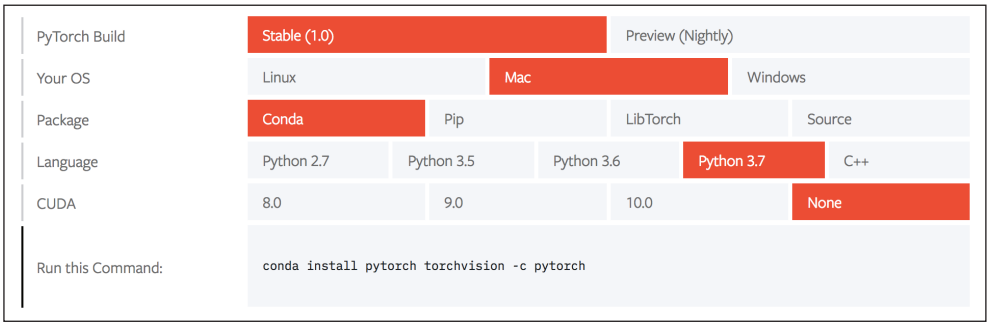
Должно появиться приглашение Python с упоминанием Anaconda:

```
Python 3.7.2 (default, Dec 29 2018, 00:00:04)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda custom (64-bit) on darwin
Type "help", "copyright", "credits" or "license" for more information.
```

Если такая картинка не появилась, проверьте список каталогов (путей), в которых ищется Python.

Следующий шаг – установка PyTorch. Перейдите по адресу https://pytorch.org/get-started/locally/ и выберите описание среды разработки из таблицы[1]:

[1] В настоящее время таблица выглядит иначе, но это типичная проблема: публикация книг отстает от развития программного обеспечения. Впрочем, изменения не принципиальны. – *Прим. перев.*



Здесь мы выбрали **Mac**, **Conda**, **Python 3.7** и локальное выполнение (без CUDA), поэтому в терминале должны ввести такую командную строку:

```
conda install pytorch torchvision -c pytorch
```

Чтобы убедиться в правильности установки PyTorch, выполните показанный ниже код на Python:

```
>>> import torch
>>> x = torch.empty(3, 4)
>>> print(x)
tensor([[ 0.0000e+00, 2.0000e+00, -1.2750e+16, -2.0005e+00],
        [ 9.8742e-37, 1.4013e-45, 9.9222e-37, 1.4013e-45],
        [ 9.9220e-37, 1.4013e-45, 9.9225e-37, 2.7551e-40]])
```

Если будет выведена матрица 3×4, значит, PyTorch установлена правильно. Итак, среда разработки успешно подготовлена.

Как это работает

Мы только что создали тензор PyTorch размера 3×4. Это пустая матрица. Слово `пустая` не означает, что значения всех элементов равны `Null`. На самом деле это неинициализированные числа с плавающей точкой, которые называются местозаместителями. Пользователь должен будет задать их впоследствии. Это очень похоже на пустой массив NumPy.

Это еще не все

Так ли необходимо устанавливать Anaconda и использовать программу `conda` для управления пакетами? Ведь можно же устанавливать пакеты с помощью менеджера `pip`. Но в некоторых отношениях `conda` лучше, чем `pip`, а именно:

- **она корректно обрабатывает зависимости между библиотеками.** Если пакет устанавливается с помощью `conda`, то автоматически будут установлены все его зависимости. А `pip` выдаст предупреждение, и установка будет отменена;
- **корректно разрешаются конфликты между пакетами.** Если для установки пакета необходим другой пакет конкретной версии (например, 2.3 или более поздней), то `conda` автоматически обновит уже установленный пакет;

- **легко создать виртуальную среду.** Виртуальная среда – это автономное дерево пакетов. Для разных приложений или проектов могут понадобиться разные виртуальные среды. Все виртуальные среды изолированы друг от друга. Рекомендуется использовать их, чтобы действия в одном приложении никак не отражались на всех остальных;
- **она совместима с pip**. Мы можем продолжать использовать `pip` вместе с `conda`, выполнив следующую команду:

```
conda install pip
```

См. также

Дополнительные сведения о `conda` можно почерпнуть из следующих ресурсов:

- **руководство пользователя по conda**: https://conda.io/projects/conda/en/latest/user-guide/index.html;
- **создание виртуальных сред и управление ими:** https://conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html.

Чтобы ближе познакомиться с PyTorch, перейдите в раздел «Getting Started» официального пособия по адресу https://pytorch.org/tutorials/#gettingstarted. Рекомендуем прочитать по крайней мере следующие части:

- **What is PyTorch:** https://pytorch.org/tutorials/beginner/blitz/tensor_tutorial.html#sphx-glr-beginner-blitz-tensor-tutorial-py;
- **Learning PyTorch with examples:** https://pytorch.org/tutorials/beginner/pytorch_with_examples.html.

Установка OpenAI Gym

Подготовив среду разработки, мы можем перейти к установке OpenAI Gym. Этот продукт содержит разнообразные окружающие среды для разработки алгоритмов обучения, без него заниматься обучением с подкреплением невозможно.

OpenAI (https://openai.com/) – некоммерческая исследовательская компания, занимающаяся созданием безопасных систем **общего искусственного интеллекта** (artificial general intelligence – **AGI**), которые были бы полезны людям. **OpenAI Gym** – мощный комплект инструментов с открытым исходным кодом, предназначенный для разработки и сравнения алгоритмов ОП. Он предлагает интерфейс к различным имитационным моделям и задачам ОП, от обучения шагающего робота до посадки на луну, от автомобильных гонок до игр Atari. Полный список окружающих сред см. по адресу https://gym.openai.com/envs/. **Агентов** для взаимодействия со средами OpenAI Gym можно программировать с применением любой библиотеки численных расчетов, например PyTorch, TensorFlow или Keras.

Как это делается

Установить Gym можно двумя способами. Первый – с помощью `pip`:

```
pip install gym
```

Если вы пользуетесь `conda`, то не забудьте предварительно установить `pip` в `conda`, выполнив команду:

```
conda install pip
```

Дело в том, что по состоянию на начало 2019 года Gym официально не был включен в состав пакетов, поддерживаемых `conda`.

Второй вариант – собрать Gym из исходного кода.

1. Сначала клонируйте пакет из его Git-репозитория:

   ```
   git clone https://github.com/openai/gym
   ```

2. Затем перейдите в папку загрузки и оттуда установите Gym:

   ```
   cd gym
   pip install -e .
   ```

   Теперь можно экспериментировать с `gym`.

3. Проверьте правильность установки Gym, выполнив такой код:

   ```
   >>> from gym import envs
   >>> print(envs.registry.all())
   dict_values([EnvSpec(Copy-v0), EnvSpec(RepeatCopy-v0),
   EnvSpec(ReversedAddition-v0), EnvSpec(ReversedAddition3-v0),
   EnvSpec(DuplicatedInput-v0), EnvSpec(Reverse-v0), EnvSpec(CartPolev0),
   EnvSpec(CartPole-v1), EnvSpec(MountainCar-v0),
   EnvSpec(MountainCarContinuous-v0), EnvSpec(Pendulum-v0),
   EnvSpec(Acrobot-v1), EnvSpec(LunarLander-v2),
   EnvSpec(LunarLanderContinuous-v2), EnvSpec(BipedalWalker-v2),
   EnvSpec(BipedalWalkerHardcore-v2), EnvSpec(CarRacing-v0),
   EnvSpec(Blackjack-v0)
   ...
   ...
   ```

Если все правильно, то будет выведен длинный список окружающих сред. С некоторыми из них мы поэкспериментируем в следующем рецепте.

Как это работает

По сравнению с простой установкой Gym с помощью `pip`, второй способ обеспечивает большую гибкость в случае, если вы захотите добавить новые среды или модифицировать Gym самостоятельно.

Это еще не все

Возникает вопрос, зачем тестировать алгоритмы обучения с подкреплением в окружающих средах Gym, если настоящие среды могут быть совершенно дру-

гими. Напомним, что в ОП делается не так уж много предположений об окружающей среде, знания о ней собираются в процессе взаимодействия. Кроме того, для сравнения качества различных алгоритмов их нужно применять в одних и тех же стандартизованных средах. Gym является прекрасным средством для тестирования, поскольку содержит много гибких и простых в использовании сред. Его можно сравнить с наборами данных, которые часто применяются для разработки и тестирования алгоритмов в обучении с учителем и без учителя, например MNIST, Imagenet, MovieLens и Thomson Reuters News.

См. также

Ознакомьтесь с официальной документацией по Gym на сайте https://gym.openai.com/docs/.

Окружающие среды Atari

Знакомство с Gym мы начнем с игр Atari.

Окружающие среды Atari (https://gym.openai.com/envs/#atari) основаны на видеоиграх для приставки **Atari 2600**, например Alien, AirRaid, Pong и Space Race. Если вы когда-нибудь играли в эти игры, то этот рецепт развлечет вас. Правда, за вас играть с Space Invaders или еще в какую-то игру будет агент.

Как это делается

Для имитации игры Atari нужно проделать следующие шаги.

1. Перед первым запуском любой окружающей среды Atari необходимо установить зависимости, выполнив в терминале команду

   ```
   pip install gym[atari]
   ```

   Если же для установки Gym вы использовали второй из описанных в предыдущем рецепте способов, то выполните команду

   ```
   pip install -e '.[atari]'
   ```

2. Установив зависимости Atari, импортируем в программу библиотеку `gym`:

   ```
   >>> import gym
   ```

3. Создаем экземпляр окружающей среды `SpaceInvaders`:

   ```
   >>> env = gym.make('SpaceInvaders-v0')
   ```

4. Приводим среду в начальное состояние:

   ```
   >>> env.reset()
   array([[[ 0, 0, 0],
   [ 0, 0, 0],
   [ 0, 0, 0],
   ...,
   ...,
   ```

```
[80, 89, 22],
[80, 89, 22],
[80, 89, 22]]], dtype=uint8)
```

Как видим, при этом возвращается начальное состояние среды.

5. Рисуем среду на экране:

```
>>> env.render()
True
```

Появляется небольшое окно:

Как видим, первоначально у нас есть три жизни (три красных космических корабля).

6. Случайным образом выбираем допустимый ход и выполняем действие:

```
>>> action = env.action_space.sample()
>>> new_state, reward, is_done, info = env.step(action)
```

Метод `step()` возвращает результат действия, а именно:

- `new_state`: новое наблюдение;
- `reward`: вознаграждение за выбранное действие в данном состоянии;
- `is_done`: флаг завершения игры. В среде `SpaceInvaders` он равен `True`, если либо не осталось жизней, либо все пришельцы уничтожены; в противном случае остается равным `False`;
- `info`: дополнительная информация об окружающей среде. В данном случае это количество оставшихся жизней. Бывает полезна при отладке.

Распечатаем значения переменных `is_done` и `info`:

```
>>> print(is_done)
False
>>> print(info)
{'ale.lives': 3}
```

Теперь можно нарисовать среду:

```
>>> env.render()
True
```

Окно игры принимает вид:

Существенных различий с предыдущим не видно, потому что корабль сделал только один ход.

7. Теперь войдем в цикл `while` и позволим агенту сделать столько ходов, сколько он сможет:

```
>>> is_done = False
>>> while not is_done:
...     action = env.action_space.sample()
...     new_state, reward, is_done, info = env.step(action)
...     print(info)
...     env.render()
{'ale.lives': 3}
True
{'ale.lives': 3}
True
.....
.....
{'ale.lives': 2}
True
{'ale.lives': 2}
True
.....
.....
{'ale.lives': 1}
True
{'ale.lives': 1}
True
```

А тем временем мы можем наблюдать за тем, как разворачивается игра, как корабль и пришельцы продолжают двигаться и стрелять. Это забавно. Когда игра закончится, окно будет выглядеть следующим образом:

Как видим, удалось набрать 150 очков. На вашей машине счет может быть больше или меньше, поскольку агент выбирает все действия случайно.

Можно также убедиться, что жизней не осталось:

```
>>> print(info)
{'ale.lives': 0}
```

Как это работает

В Gym экземпляр окружающей среды создается методом `make()`, которому передается имя среды.

Агент выбирает действия случайным образом, обращаясь к методу `sample()`.

Обычно у нас имеется более интеллектуальный агент, обученный тем или иным алгоритмом ОП. Сейчас мы просто продемонстрировали, как можно смоделировать окружающую среду и как агент выбирает действия, не обращая внимания на их результат.

Несколько раз выполнив метод `sample()`, получим:

```
>>> env.action_space.sample()
0
>>> env.action_space.sample()
3
>>> env.action_space.sample()
0
>>> env.action_space.sample()
4
>>> env.action_space.sample()
2
>>> env.action_space.sample()
1
```

```
>>> env.action_space.sample()
4
>>> env.action_space.sample()
5
>>> env.action_space.sample()
1
>>> env.action_space.sample()
0
```

Всего имеется шесть возможных действий. Это можно подтвердить, выполнив такую команду:

```
>>> env.action_space
Discrete(6)
```

Эти действия таковы (в порядке от 0 до 5): No Operation (Ничего не делать), Fire (Огонь), Up (Вверх), Right (Вправо), Left (Влево), Down (Вниз).

Метод `step()` дает возможность агенту выполнить действие с указанным номером. Метод `render()` обновляет окно игры на основе последнего наблюдения за окружающей средой.

Наблюдение `new_state` представлено матрицей 210×160×3:

```
>>> print(new_state.shape)
(210, 160, 3)
```

Это означает, что каждый кадр на экране – RGB-изображение размера 210×160.

Это еще не все

Может возникнуть вопрос, зачем вообще нужно устанавливать зависимости Atari. Дело в том, что есть еще несколько окружающих сред, не являющихся частью установки `gym`, в т. ч. Box2d, Classic control, MuJoCo и Robotics.

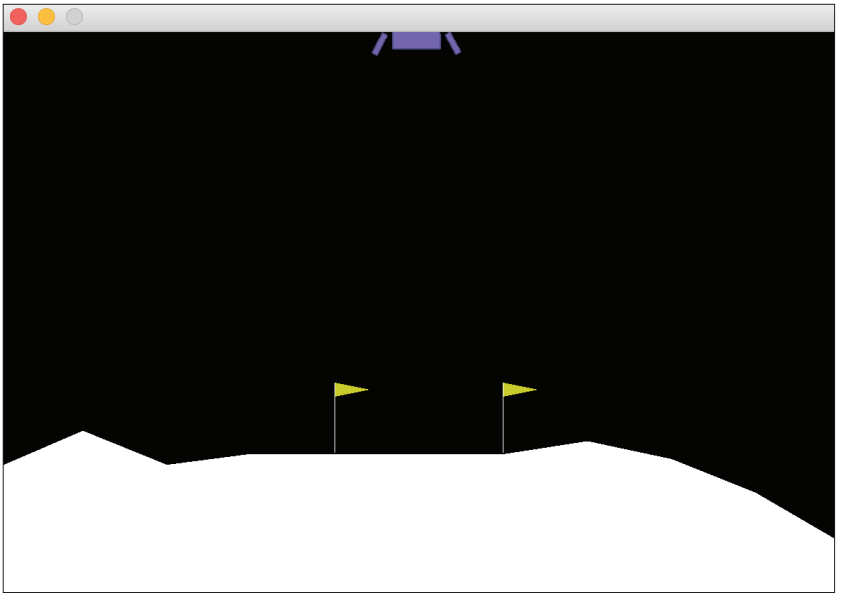
Взять, к примеру, среды Box2d. Чтобы можно было экспериментировать с ними, нужно сначала установить зависимости `Box2d`. Тут тоже есть два способа:

```
pip install gym[box2d]
pip install -e '.[box2d]'
```

После этого можно будет создать среду `LunarLander`:

```
>>> env = gym.make('LunarLander-v2')
>>> env.reset()
array([-5.0468446e-04, 1.4135642e+00, -5.1140346e-02, 1.1751971e-01,
        5.9164839e-04, 1.1584054e-02, 0.0000000e+00, 0.0000000e+00],
    dtype=float32)
>>> env.render()
```

Должно появиться окно игры:



См. также

Если вас интересует какая-то окружающая среда, но вы не знаете, как она называется, можете найти ее в таблице сред на странице по адресу https://github.com/openai/gym/wiki/Table-of-environments. Помимо имени среды, там приведены размер матрицы наблюдений и количество возможных действий.

Окружающая среда CartPole

В этом рецепте мы поработаем еще с одной окружающей средой, чтобы получше познакомиться с Gym. Среда CartPole – классический пример, используемый в исследованиях по обучению с подкреплением.

Задача состоит в том, чтобы удержать в вертикальном положении стержень, шарнирно закрепленный на тележке. На каждом временном шаге агент перемещает тележку влево или вправо на расстояние 1, стремясь к тому, чтобы стержень не упал. Считается, что стержень упал, если он отклонился от вертикали более чем на 12 градусов или если тележка сдвинулась больше чем на 2.4 единицы от начального положения. Эпизод заканчивается при выполнении одного из следующих условий:

- стержень упал;
- количество временных шагов достигло 200.

Как это делается

Для экспериментов со средой `CartPole` выполним следующие шаги.

1. Сначала найдем имя среды в таблице по адресу https://github.com/openai/gym/wiki/Table-ofenvironments. Выясняется, что она называется '`CartPole-v0`', что пространство наблюдений в ней представлено 4-мерным массивом, а возможных действий всего два (логично).
2. Импортируем библиотеку Gym и создадим экземпляр среды `CartPole`:

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

3. Приведем среду в начальное состояние:

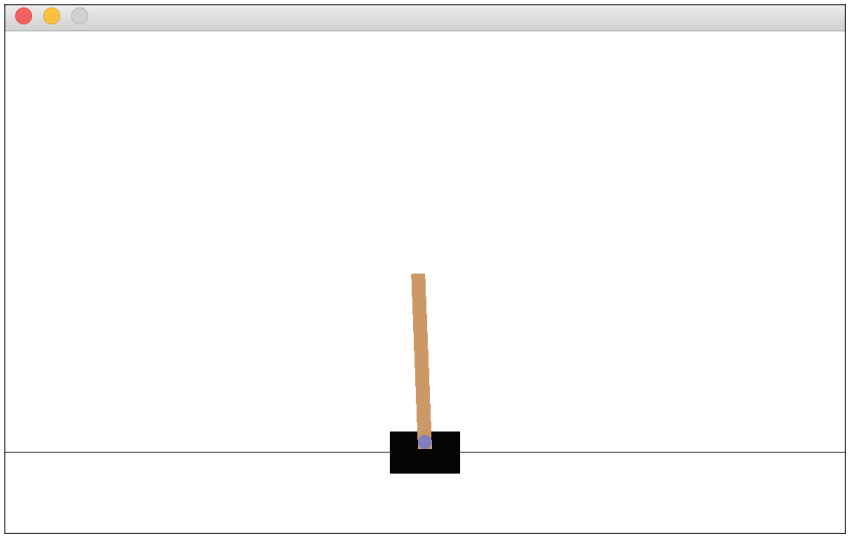
```
>>> env.reset()
array([-0.00153354, 0.01961605, -0.03912845, -0.01850426])
```

   Как и раньше, возвращается начальное состояние среды, представленное массивом из четырех чисел с плавающей точкой.

4. Рисуем среду на экране:

```
>>> env.render()
True
```

   Должно появиться небольшое окно:

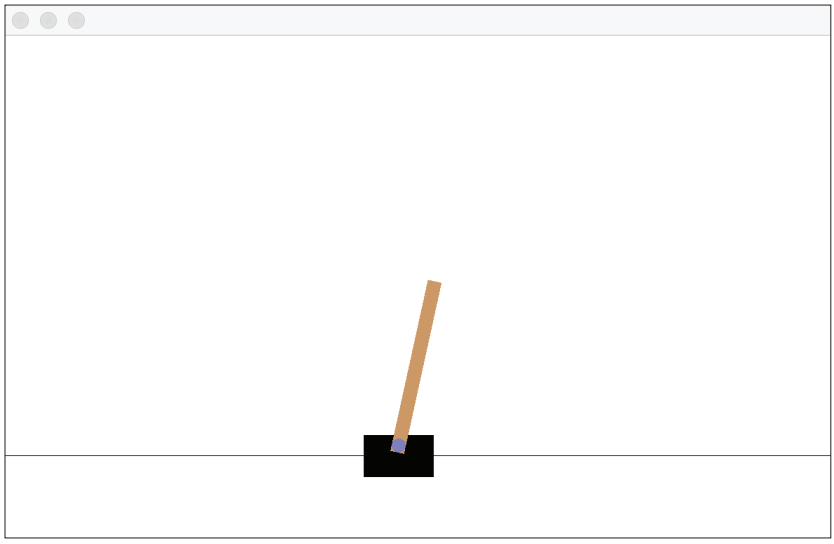


5. Теперь войдем в цикл `while` и позволим агенту сделать столько ходов, сколько он сможет:

```
>>> is_done = False
>>> while not is_done:
...     action = env.action_space.sample()
```

```
...     new_state, reward, is_done, info = env.step(action)
...     print(new_state)
...     env.render()
...
[-0.00114122 -0.17492355 -0.03949854 0.26158095]
True
[-0.00463969 -0.36946006 -0.03426692 0.54154857]
True
……
……
[-0.11973207 -0.41075106 0.19355244 1.11780626]
True
[-0.12794709 -0.21862176 0.21590856 0.89154351]
True
```

Тем временем тележка и стержень двигаются. В конце игры оба остановятся, и окно будет выглядеть примерно так:



Эпизод длится всего несколько шагов, потому что действия – вправо или влево – выбираются случайным образом. Можно ли запомнить весь процесс, чтобы впоследствии воспроизвести его? Можно, для этого нужно добавить две строчки, как показано на шаге 7. Но если вы работаете в системе Mac или Linux, то предварительно нужно выполнить шаг 6 (иначе можно сразу переходить к шагу 7).

6. Для записи видео необходимо установить пакет `ffmpeg`. В Mac это делается командой

   ```
   brew install ffmpeg
   ```

   А в Linux – командой

   ```
   sudo apt-get install ffmpeg
   ```

7. После создания экземпляра `CartPole` добавьте такие две строчки:

```
>>> video_dir = './cartpole_video/'
>>> env = gym.wrappers.Monitor(env, video_dir)
```

В результате все отображаемое на экране будет сохранено в указанном каталоге.

Теперь повторно выполним шаги с 3 по 5. По завершении эпизода в каталоге `video_dir` окажется файл с расширением `.mp4`. Видео очень короткое – всего около 1 секунды.

Как это работает

В этом рецепте мы на каждом шаге распечатываем массив состояния. Но что означает каждый элемент этого массива? Подробные сведения о среде CartPole имеются на вики-странице Gym в GitHub: https://github.com/openai/gym/wiki/CartPole-v0. И вот что означают эти четыре числа:

- положение тележки: число от –2.4 до 2.4. Если положение выходит за пределы этого диапазона, то эпизод завершается;
- скорость тележки;
- угол наклона стержня: если значение меньше –0.209 (–12 градусов) или больше 0.209 (12 градусов), то эпизод завершается;
- скорость верхнего конца стержня.

Действие может принимать значение 0 (сдвинуть тележку влево) или 1 (вправо).

В этой окружающей среде **вознаграждение** равно + 1 на каждом временном шаге вплоть до завершения эпизода. Это можно легко проверить, печатая вознаграждение на каждом шаге. Полное же вознаграждение равно количеству временных шагов.

Это еще не все

Пока что мы прогнали всего один эпизод. Чтобы оценить качество агента, можно прогнать много эпизодов и усреднить полное вознаграждение. Это даст нам представление о качестве агента, выбирающего действия случайным образом:

Пусть число эпизодов равно 10 000:

```
>>> n_episode = 10000
```

В каждом эпизоде вычисляется полное вознаграждение, равное сумме вознаграждений на каждом шаге:

```
>>> total_rewards = []
>>> for episode in range(n_episode):
...     state = env.reset()
...     total_reward = 0
...     is_done = False
...     while not is_done:
...         action = env.action_space.sample()
```

```
...         state, reward, is_done, _ = env.step(action)
...         total_reward += reward
...     total_rewards.append(total_reward)
```

И в самом конце вычисляется среднее полное вознаграждение:

```
>>> print('Среднее полное вознаграждение в {} эпизодах: {}'.format(
          n_episode, sum(total_rewards) / n_episode))
Среднее полное вознаграждение в 10 000 эпизодов: 22.2473
```

В среднем полное вознаграждение при случайном выборе действий составляет 22.25.

Понятно, что выбор действий наугад – не самая разумная стратегия, и в следующих разделах мы улучшим ее. Но пока сделаем перерыв и немного поговорим о самой библиотеке PyTorch.

Основы PyTorch

Как уже было сказано, PyTorch – библиотека численных расчетов, которой мы будем пользоваться в этой книге для реализации алгоритмов обучения с подкреплением.

PyTorch – модная библиотека для научных расчетов и машинного обучения (в т. ч. глубокого), разработанная компанией Facebook. Основная структура данных в ней – тензор, напоминающий массив `ndarray` из библиотеки NumPy. С точки зрения научных вычислений, PyTorch и NumPy примерно равноценны. Однако PyTorch быстрее выполняет обход массивов и операции с ними. Связано это прежде всего с тем, что в PyTorch быстрее производится доступ к элементу. Поэтому все больше народу полагает, что PyTorch в конечном итоге вытеснит NumPy.

Как это делается

Сделаем краткий обзор программирования с использованием PyTorch.

1. В предыдущем рецепте мы создали неинициализированную матрицу. А что, если нужно инициализировать ее случайными значениями? На помощь приходят следующие команды:

   ```
   >>> import torch
   >>> x = torch.rand(3, 4)
   >>> print(x)
   tensor([[0.8052, 0.3370, 0.7676, 0.2442],
           [0.7073, 0.4468, 0.1277, 0.6842],
           [0.6688, 0.2107, 0.0527, 0.4391]])
   ```

   Генерируются случайные числа с плавающей точкой с равномерным распределением в интервале (0, 1).

2. Мы можем задать тип данных возвращаемого тензора. Например, чтобы вернуть тензор двойной точности (`float64`), нужно написать:

```
>>> x = torch.rand(3, 4, dtype=torch.double)
>>> print(x)
tensor([[0.6848, 0.3155, 0.8413, 0.5387],
        [0.9517, 0.1657, 0.6056, 0.5794],
        [0.0351, 0.3801, 0.7837, 0.4883]], dtype=torch.float64)
```

По умолчанию подразумевается тип данных `float`.

3. Далее создадим матрицы, состоящие из одних нулей и из одних единиц:

```
>>> x = torch.zeros(3, 4)
>>> print(x)
tensor([[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]])
>>> x = torch.ones(3, 4)
>>> print(x)
tensor([[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]])
```

4. Вот как можно узнать размер тензора:

```
>>> print(x.size())
torch.Size([3, 4])
```

`torch.Size` является кортежем.

5. Для изменения формы тензора служит метод `view()`:

```
>>> x_reshaped = x.view(2, 6)
>>> print(x_reshaped)
tensor([[1., 1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1., 1.]])
```

6. Тензор можно создать из данных другого типа, например одиночного значения, списка или вложенного списка:

```
>>> x1 = torch.tensor(3)
>>> print(x1)
tensor(3)
>>> x2 = torch.tensor([14.2, 3, 4])
>>> print(x2)
tensor([14.2000, 3.0000, 4.0000])
>>> x3 = torch.tensor([[3, 4, 6], [2, 1.0, 5]])
>>> print(x3)
tensor([[3., 4., 6.],
        [2., 1., 5.]])
```

7. Чтобы получить доступ к элементам тензора, содержащего более одного элемента, можно воспользоваться индексированием, как в NumPy:

```
>>> print(x2[1])
tensor(3.)
>>> print(x3[1, 0])
tensor(2.)
```

```
>>> print(x3[:, 1])
tensor([4., 1.])
>>> print(x3[:, 1:])
tensor([[4., 6.],
        [1., 5.]])
```

Для тензора с одним элементом это можно сделать с помощью метода `item()`:

```
>>> print(x1.item())
3
```

8. Тензор можно преобразовать в массив NumPy и наоборот. Для преобразования тензора в массив NumPy служит метод `numpy()`:

```
>>> x3.numpy()
array([[3., 4., 6.],
       [2., 1., 5.]], dtype=float32)
```

А для преобразования массива NumPy в тензор – метод `from_numpy()`:

```
>>> import numpy as np
>>> x_np = np.ones(3)
>>> x_torch = torch.from_numpy(x_np)
>>> print(x_torch)
tensor([1., 1., 1.], dtype=torch.float64)
```

Отметим, что если входной массив NumPy имеет тип `float`, то выходной тензор будет иметь тип `double`. Иногда необходимо явное приведение типов.

В следующем примере тензор типа `double` преобразуется в тип `float`:

```
>>> print(x_torch.float())
tensor([1., 1., 1.])
```

9. Операции в PyTorch и NumPy похожи. Например, сложение производится следующим образом:

```
>>> x4 = torch.tensor([[1, 0, 0], [0, 1.0, 0]])
>>> print(x3 + x4)
tensor([[4., 4., 6.],
       [2., 2., 5.]])
```

Можно также использовать метод `add()`:

```
>>> print(torch.add(x3, x4))
tensor([[4., 4., 6.],
       [2., 2., 5.]])
```

10. PyTorch поддерживает операции на месте, которые изменяют объект тензора. Например, выполним такую команду:

```
>>> x3.add_(x4)
tensor([[4., 4., 6.],
[2., 2., 5.]])
```

Легко видеть, что x3 стал равен сумме прежнего x3 и x4:

```
>>> print(x3)
tensor([[4., 4., 6.],
        [2., 2., 5.]])
```

Это еще не все

Любой метод, имя которого оканчивается знаком _, выполняется на месте, т. е. в тензор записывается новое значение.

См. также

Полный перечень операций с тензорами в PyTorch опубликован в официальной документации по адресу https://pytorch.org/docs/stable/torch.html. Именно здесь лучше всего искать информацию, если возникла проблема с использованием PyTorch.

Реализация и оценивание стратегии случайного поиска

Итак, мы немного попрактиковались в работе с PyTorch и, начиная с этого рецепта, будем рассматривать более разумные стратегии решения задачи CartPole, чем действия наугад. В этом рецепте мы обсудим стратегию случайного поиска.

Простой, но эффективный подход заключается в том, чтобы отобразить наблюдение на вектор из двух чисел, представляющих два действия. Выбирается действие, ценность которого больше. Линейное отображение описывается матрицей весов размера 4×2, поскольку наблюдения в данном случае четырехмерные. В каждом эпизоде веса генерируются случайным образом и используются для вычисления действия на каждом шаге эпизода. Затем вычисляется полное вознаграждение. Этот процесс повторяется для большого числа эпизодов, и в конце обученной стратегией становится матрица весов, которая принесла наибольшее полное вознаграждение. Такой подход называется **случайным поиском**, поскольку веса случайно выбираются в каждом испытании в надежде, что после большого числа испытаний будут найдены наилучшие веса.

Как это делается

Давайте реализуем алгоритм случайного поиска с помощью PyTorch.

1. Импортируем пакеты Gym и PyTorch и создадим экземпляр окружающей среды:

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

2. Получим размерности пространств наблюдений и действий:

```
>>> n_state = env.observation_space.shape[0]
>>> n_state
 4
>>> n_action = env.action_space.n
>>> n_action
 2
```

Они понадобятся для определения тензора – матрицы весов размера 4×2.

3. Определим функцию, которая имитирует эпизод с данной входной матрицей весов и возвращает полное вознаграждение:

```
>>> def run_episode(env, weight):
...     state = env.reset()
...     total_reward = 0
...     is_done = False
...     while not is_done:
...         state = torch.from_numpy(state).float()
...         action = torch.argmax(torch.matmul(state, weight))
...         state, reward, is_done, _ = env.step(action.item())
...         total_reward += reward
...     return total_reward
```

Здесь массив состояний `state` преобразуется в тензор типа `float`, поскольку нам нужно вычислить линейное отображение – произведение состояния на вес, `torch.matmul(state, weight)`. Действие с большей ценностью выбирается с помощью операции `torch.argmax()`. И не забудьте получить значение результирующего тензора действия, вызвав метод `.item()`, потому что это тензор, содержащий один элемент.

4. Зададим количество эпизодов:

```
>>> n_episode = 1000
```

5. Необходимо запоминать лучшее полное вознаграждение по всем эпизодам и соответствующую ему матрицу весов. Поэтому зададим начальные значения:

```
>>> best_total_reward = 0
>>> best_weight = None
```

Также будем запоминать полное вознаграждение в каждом эпизоде:

```
>>> total_rewards = []
```

6. Теперь можно прогнать `n_episode` эпизодов. Для каждого эпизода выполняются следующие действия:
   - случайным образом выбрать веса;
   - дать агенту возможность предпринять действия в соответствии с линейным отображением;
   - эпизод завершается, и возвращается полное вознаграждение;
   - при необходимости обновить наилучшее полное вознаграждение и наилучшую матрицу весов;
   - запомнить полученное в эпизоде полное вознаграждение.

Ниже приведен соответствующий код:

```
>>> for episode in range(n_episode):
...     weight = torch.rand(n_state, n_action)
...     total_reward = run_episode(env, weight)
...     print('Эпизод {}: {}'.format(episode+1, total_reward))
...     if total_reward > best_total_reward:
...         best_weight = weight
...         best_total_reward = total_reward
...     total_rewards.append(total_reward)
...
Эпизод 1: 10.0
Эпизод 2: 73.0
Эпизод 3: 86.0
Эпизод 4: 10.0
Эпизод 5: 11.0
.....
.....
Эпизод 996: 200.0
Эпизод 997: 11.0
Эпизод 998: 200.0
Эпизод 999: 200.0
Эпизод 1000: 9.0
```

Мы нашли наилучшую стратегию, выполнив 1000 эпизодов случайного поиска. Она параметризована матрицей весов `best_weight`.

7. Прежде чем проверить наилучшую стратегию на тестовых эпизодах, вычислим среднее полное вознаграждение:

```
>>> print('Среднее полное вознаграждение в {} эпизодах: {}'.format(
          n_episode, sum(total_rewards) / n_episode))
Среднее полное вознаграждение в 1000  эпизодов: 47.197
```

Оно в два раза больше, чем для случайной стратегии (22.25).

8. Теперь посмотрим, какие результаты обученная стратегия покажет на 100 новых эпизодах:

```
>>> n_episode_eval = 100
>>> total_rewards_eval = []
>>> for episode in range(n_episode_eval):
...     total_reward = run_episode(env, best_weight)
...     print('Эпизод {}: {}'.format(episode+1, total_reward))
...     total_rewards_eval.append(total_reward)
...
Эпизод 1: 200.0
Эпизод 2: 200.0
```

```
Эпизод 3: 200.0
Эпизод 4: 200.0
Эпизод 5: 200.0
……
……
Эпизод 96: 200.0
Эпизод 97: 188.0
Эпизод 98: 200.0
Эпизод 99: 200.0
Эпизод 100: 200.0
>>> print('Среднее полное вознаграждение в {} эпизодах: {}'.format(
          n_episode, sum(total_rewards_eval) / n_episode_eval))
Среднее полное вознаграждение в 1000  эпизодов: 196.72
```

Как ни странно, среднее вознаграждение при следовании обученной стратегии на тестовых эпизодах оказалось близко к максимуму, равному 200. Но разброс довольно велик – от 160 до 200.

Как это работает

Алгоритм случайного поиска так хорошо работает, потому что окружающая среда CartPole очень простая. Ее состояние определяется всего четырьмя переменными. Напомним, что в игре Atari Space Invaders состояний больше 100 000 (210 * 160 * 3). А размерность пространства действий CartPole в три раза меньше, чем в Space Invaders. Вообще, простые алгоритмы хорошо работают в простых задачах. В нашем случае мы всего лишь искали наилучшее линейное отображение из пространства состояний в пространство действий, случайно выбирая его из множества возможных.

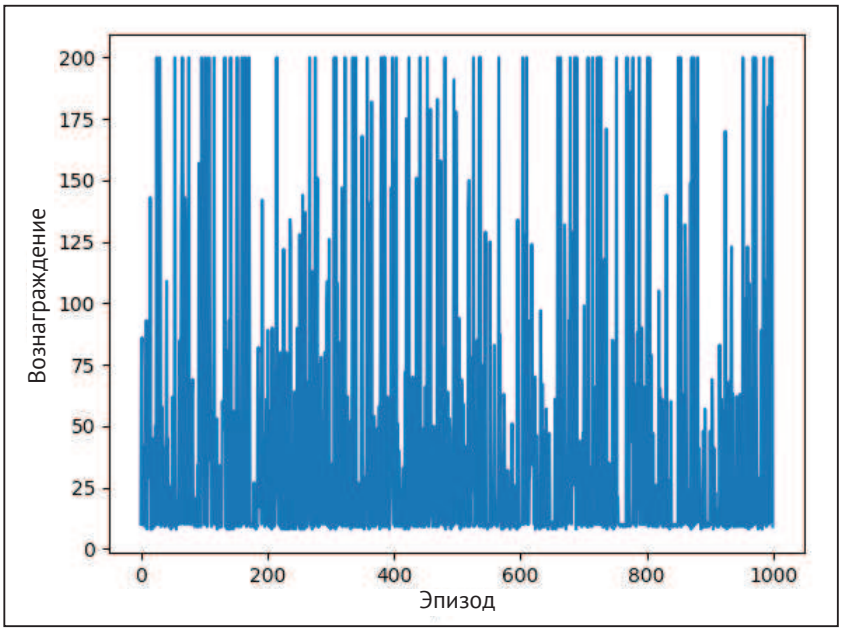
Мы также заметили еще одну интересную вещь: стратегия, обученная методом случайного поиска, оказалась лучше случайного выбора действий. Это потому, что при выборе случайного линейного отображения учитываются наблюдения. Имея больше информации об окружающей среде, мы можем принимать более осмысленные решения, чем при полностью случайном выборе.

Это еще не все

Мы можем построить график полного вознаграждения на этапе обучения:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(total_rewards)
>>> plt.xlabel('Эпизод')
>>> plt.ylabel('Вознаграждение')
>>> plt.show()
```

Результат показан ниже.



Если на вашем компьютере отсутствует библиотека matplotlib, установите ее командой

```
conda install matplotlib
```

Как видно, вознаграждение меняется хаотично, и никакой тенденции к улучшению с ростом числа эпизодов не наблюдается. Что и следовало ожидать.

На графике зависимости вознаграждения от номера эпизода видно, что в некоторых эпизодах вознаграждение достигает 200. После первого такого события обучение можно заканчивать, потому что лучшего результата уже не достичь. Ниже показан код этапа обучения после такого изменения:

```
>>> n_episode = 1000
>>> best_total_reward = 0
>>> best_weight = None
>>> total_rewards = []
>>> for episode in range(n_episode):
...     weight = torch.rand(n_state, n_action)
...     total_reward = run_episode(env, weight)
...     print('Эпизод {}: {}'.format(episode+1, total_reward))
...     if total_reward > best_total_reward:
...         best_weight = weight
...         best_total_reward = total_reward
...     total_rewards.append(total_reward)
...     if best_total_reward == 200:
...         break
Эпизод 1: 9.0
```

```
Эпизод 2: 8.0
Эпизод 3: 10.0
Эпизод 4: 10.0
Эпизод 5: 10.0
Эпизод 6: 9.0
Эпизод 7: 17.0
Эпизод 8: 10.0
Эпизод 9: 43.0
Эпизод 10: 10.0
Эпизод 11: 10.0
Эпизод 12: 106.0
Эпизод 13: 8.0
Эпизод 14: 32.0
Эпизод 15: 98.0
Эпизод 16: 10.0
Эпизод 17: 200.0
```

Стратегия, при которой достигается максимальное вознаграждение, найдена в эпизоде 17. Но это мог бы быть любой другой эпизод, т. к. веса генерируются случайным образом. Чтобы вычислить математическое ожидание необходимого количества эпизодов, можно повторить этот процесс обучения 1000 раз и вычислить среднее количество эпизодов:

```
>>> n_training = 1000
>>> n_episode_training = []
>>> for _ in range(n_training):
...     for episode in range(n_episode):
...     weight = torch.rand(n_state, n_action)
...         total_reward = run_episode(env, weight)
...         if total_reward == 200:
...             n_episode_training.append(episode+1)
...             break
>>> print('Математическое ожидание необходимого числа эпизодов: ',
          sum(n_episode_training) / n_training)
Математическое ожидание необходимого числа эпизодов: 13.442
```

В среднем для нахождения наилучшей стратегии нужно 13 эпизодов.

Алгоритм восхождения на вершину

При рассмотрении стратегии случайного поиска все эпизоды были независимы. На самом деле их можно было бы выполнять параллельно и в итоге выбрать веса, при которых получились наилучшие результаты. Мы лишний раз подтвердили это, построив график зависимости вознаграждения от номера эпизода, на котором нет никакого восходящего тренда. В этом рецепте мы разработаем алгоритм восхождения на вершину, позволяющий передавать дальше знания, накопленные в предыдущих эпизодах.

В начале алгоритма восхождения на вершину веса тоже выбираются случайным образом. Но в каждом эпизоде к весу прибавляется шум. Если полное вознаграждение увеличилось, то мы заменяем веса новыми, в противном случае

оставляем старые. При этом веса от эпизода к эпизоду улучшаются, а не изменяются хаотически в каждом эпизоде.

Как это делается

Реализуем алгоритм восхождения на вершину с помощью PyTorch.

1. Как и прежде, импортируем необходимые пакеты, создадим экземпляр окружающей среды и получим размерности пространства наблюдений и действий.

```
>>> import gym
>>> import torch
>>> env = gym.make('CartPole-v0')
>>> n_state = env.observation_space.shape[0]
>>> n_action = env.action_space.n
```

2. Повторно воспользуемся функцией `run_episode`, написанной в предыдущем рецепте, и не станем повторять ее код. Напомним, что она получает входные веса, имитирует эпизод и возвращает полное вознаграждение.

3. Число эпизодов пусть будет равно 1000:

```
>>> n_episode = 1000
```

4. Мы будем запоминать наилучшее полное вознаграждение и соответствующие веса. Зададим начальные значения:

```
>>> best_total_reward = 0
>>> best_weight = torch.rand(n_state, n_action)
```

   Также будем запоминать полное вознаграждение в каждом эпизоде:

```
>>> total_rewards = []
```

5. Прибавляем к весам шум в каждом эпизоде. Шум масштабируется, чтобы он не затмил собой сами веса. В качестве масштабного коэффициента выберем 0.01:

```
>>> noise_scale = 0.01
```

6. Теперь можно выполнить `n_episode` эпизодов. Случайно выбрав начальный вес, мы затем производим следующие действия:
   - прибавить случайный шум к весу;
   - дать агенту возможность предпринять действия в соответствии с линейным отображением;
   - эпизод завершается, и возвращается полное вознаграждение;
   - если текущее вознаграждение больше максимального на данный момент, то обновить текущее вознаграждение и соответствующий ему вес;
   - иначе оставить наилучшее вознаграждение и вес прежними;
   - запомнить полученное в эпизоде полное вознаграждение.

Ниже приведен соответствующий код:

```
>>> for episode in range(n_episode):
...     weight = best_weight +
                    noise_scale * torch.rand(n_state, n_action)
...     total_reward = run_episode(env, weight)
...     if total_reward >= best_total_reward:
...         best_total_reward = total_reward
...         best_weight = weight
...     total_rewards.append(total_reward)
...     print('Эпизод {}: {}'.format(episode + 1, total_reward))
...
Эпизод 1: 56.0
Эпизод 2: 52.0
Эпизод 3: 85.0
Эпизод 4: 106.0
Эпизод 5: 41.0
……
……
Эпизод 996: 39.0
Эпизод 997: 51.0
Эпизод 998: 49.0
Эпизод 999: 54.0
Эпизод 1000: 41.0
```

Вычисляем среднее полное вознаграждение, полученное с помощью алгоритма восхождения на вершину:

```
>>> print('Среднее полное вознаграждение в {} эпизодах: {}'.format(
          n_episode, sum(total_rewards) / n_episode))
Среднее полное вознаграждение в 1000 эпизодов: 50.024
```

7. Чтобы оценить результаты обучения, повторим весь процесс (код, описанный в шагах 4–6) несколько раз. Можно видеть, что среднее полное вознаграждение сильно флуктуирует:

```
Среднее полное вознаграждение в 1000 эпизодов: 9.261
Среднее полное вознаграждение в 1000 эпизодов: 88.565
Среднее полное вознаграждение в 1000 эпизодов: 51.796
Среднее полное вознаграждение в 1000 эпизодов: 9.41
Среднее полное вознаграждение в 1000 эпизодов: 109.758
Среднее полное вознаграждение в 1000 эпизодов: 55.787
Среднее полное вознаграждение в 1000 эпизодов: 189.251
Среднее полное вознаграждение в 1000 эпизодов: 177.624
Среднее полное вознаграждение в 1000 эпизодов: 9.146
Среднее полное вознаграждение в 1000 эпизодов: 102.311
```

В чем причина такой изменчивости? Как выясняется, если начальные веса были выбраны неудачно, то прибавление небольшого шума почти не приводит к улучшению качества, т. е. сходимость медленная. С другой стороны, даже если начальные веса выбраны хорошо, прибавление слишком большого шума может увести далеко от оптимальных весов,

поставив качество под угрозу. Как сделать обучение алгоритма восхождения на вершину более устойчивым и надежным? Можно адаптировать величину шума к качеству, как мы адаптируем скорость обучения при градиентном спуске. Рассмотрим шаг 8 более детально.

8. Чтобы сделать шум адаптивным, нужно выполнить следующие действия:
   - задать начальный коэффициент шума;
   - если качество в эпизоде улучшилось, уменьшить коэффициент шума. В нашем случае коэффициент уменьшается вдвое, но никогда не становится меньше 0.0001;
   - если качество в эпизоде ухудшилось, увеличить коэффициент шума. В нашем случае коэффициент увеличивается вдвое, но никогда не становится больше 2.

   Ниже приведен соответствующий код:

```
>>> noise_scale = 0.01
>>> best_total_reward = 0
>>> total_rewards = []
>>> for episode in range(n_episode):
...     weight = best_weight +
                    noise_scale * torch.rand(n_state, n_action)
...     total_reward = run_episode(env, weight)
...     if total_reward >= best_total_reward:
...         best_total_reward = total_reward
...         best_weight = weight
...         noise_scale = max(noise_scale / 2, 1e-4)
...     else:
...         noise_scale = min(noise_scale * 2, 2)
...     print('Эпизод {}: {}'.format(episode + 1, total_reward))
...     total_rewards.append(total_reward)
...
Эпизод 1: 9.0
Эпизод 2: 9.0
Эпизод 3: 9.0
Эпизод 4: 10.0
Эпизод 5: 10.0
.....
.....
Эпизод 996: 200.0
Эпизод 997: 200.0
Эпизод 998: 200.0
Эпизод 999: 200.0
Эпизод 1000: 200.0
```

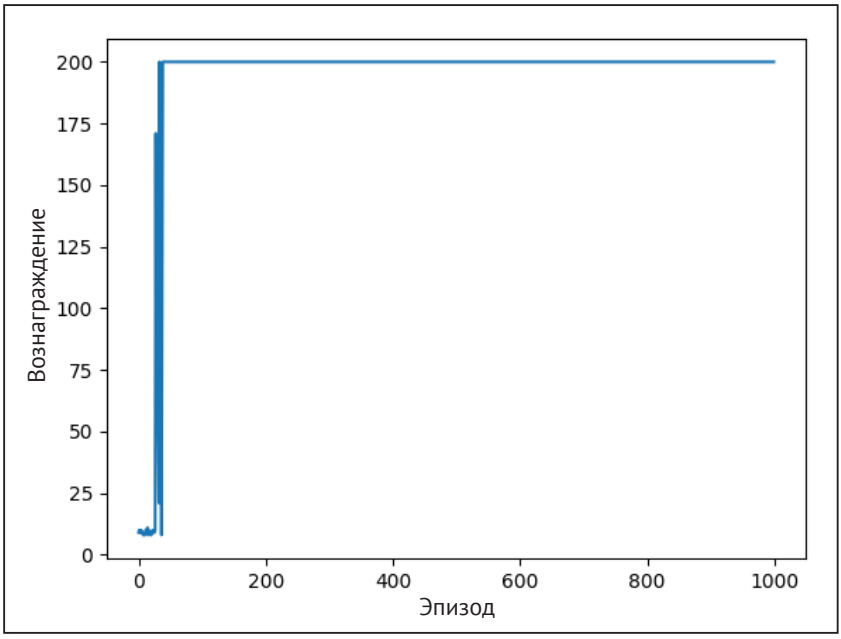
Вознаграждение от эпизода к эпизоду увеличивается. Уже в первых 100 эпизодах оно достигает 200 и остается на этом уровне. Среднее полное вознаграждение тоже выглядит обнадеживающе:

```
>>> print('Среднее полное вознаграждение в {} эпизодах: {}'.format(
          n_episode, sum(total_rewards) / n_episode))
Среднее полное вознаграждение в 1000 эпизодов: 186.11
```

Построим график зависимости полного вознаграждения от номера эпизода.

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(total_rewards)
>>> plt.xlabel('Эпизод')
>>> plt.ylabel('Вознаграждение')
>>> plt.show()
```

Наблюдается отчетливый восходящий тренд с выходом на плато, соответствующее максимальному значению.



Можете выполнить новый процесс обучения несколько раз. По сравнению с обучением с постоянным коэффициентом шума результаты очень устойчивы.

9. Теперь посмотрим, как обученная стратегия поведет себя в 100 новых эпизодах.

```
>>> n_episode_eval = 100
>>> total_rewards_eval = []
>>> for episode in range(n_episode_eval):
...     total_reward = run_episode(env, best_weight)
...     print('Эпизод {}: {}'.format(episode+1, total_reward))
...     total_rewards_eval.append(total_reward)
...
Эпизод 1: 200.0
Эпизод 2: 200.0
Эпизод 3: 200.0
```

```
Эпизод 4: 200.0
Эпизод 5: 200.0
.....
.....
Эпизод 96: 200.0
Эпизод 97: 200.0
Эпизод 98: 200.0
Эпизод 99: 200.0
Эпизод 100: 200.0
```

Вычислим среднее полное вознаграждение:

```
>>> print('Среднее полное вознаграждение в {} эпизодах: {}'.format(n_episode,
sum(total_rewards) / n_episode))
Среднее полное вознаграждение в 1000 эпизодов: 199.94
```

Как видим, среднее полное вознаграждение в тестовых эпизодах близко к максимальному значению 200, полученному при следовании обученной стратегии. Можете повторить эксперимент несколько раз – результаты мало разнятся.

Как это работает

Алгоритм восхождения на вершину позволил добиться гораздо большего качества, чем случайный поиск, просто благодаря прибавлению к весу адаптивного шума в каждом эпизоде. Можно считать, что это частный случай градиентного спуска без целевой переменной. Дополнительный шум играет роль градиента, хотя и выбираемого случайно. Коэффициент шума – это скорость обучения, адаптирующаяся к вознаграждению в предыдущем эпизоде. Целью при восхождении на вершину становится достижение максимального вознаграждения. Теперь агент проходит каждый эпизод не изолированно от других, а использует полученные ранее знания, чтобы выбирать действия более надежно. Вознаграждение, как и следует из названия алгоритма, с каждым эпизодом увеличивается, поскольку веса постепенно приближаются к оптимальным.

Это еще не все

Мы видели, что вознаграждение может достичь максимума уже в первых 100 эпизодах. А нельзя ли остановить обучение по достижении значения 200, как в стратегии случайного поиска? Нет, это не слишком удачная идея. Напомним, что при восхождении на вершину агент непрерывно совершенствуется. Даже найдя вес, при котором вознаграждение максимально, он продолжает искать оптимум в окрестности этого веса. В данном случае под оптимумом понимается стратегия, решающая задачу о балансировании стержня. Согласно вики-странице https://github.com/openai/gym/wiki/CartPole-v0, «решающая» означает, что в 100 последовательных эпизодах среднее вознаграждение не менее 195.

Уточним критерий остановки в соответствии с этим определением.

```
>>> noise_scale = 0.01
>>> best_total_reward = 0
```

```
>>> total_rewards = []
>>> for episode in range(n_episode):
...     weight = best_weight + noise_scale * torch.rand(n_state, n_action)
...     total_reward = run_episode(env, weight)
...     if total_reward >= best_total_reward:
...         best_total_reward = total_reward
...         best_weight = weight
...         noise_scale = max(noise_scale / 2, 1e-4)
...     else:
...         noise_scale = min(noise_scale * 2, 2)
...     print('Эпизод {}: {}'.format(episode + 1, total_reward))
...     total_rewards.append(total_reward)
...     if episode >= 99 and sum(total_rewards[-100:]) >= 19500:
...         break
...
Эпизод 1: 9.0
Эпизод 2: 9.0
Эпизод 3: 10.0
Эпизод 4: 10.0
Эпизод 5: 9.0
.....
.....
Эпизод 133: 200.0
Эпизод 134: 200.0
Эпизод 135: 200.0
Эпизод 136: 200.0
Эпизод 137: 200.0
```

После эпизода 137 задача считается решенной.

См. также

Подробнее об алгоритме восхождения на вершину можно узнать из следующих источников:

- https://en.wikipedia.org/wiki/Hill_climbing;
- https://www.geeksforgeeks.org/introduction-hill-climbing-artificialintelligence/.

Алгоритм градиента стратегии

Последний рецепт в этой главе, посвященной окружающей среде CartPole, относится к алгоритму градиента стратегии. Он, пожалуй, несколько сложнее, чем необходимо для решения этой простой задачи, для которой случайного поиска и алгоритма восхождения на вершину вполне достаточно. Но это выдающийся алгоритм, которым мы еще воспользуемся в более сложных средах.

В алгоритме градиента стратегии веса модели изменяются в направлении градиента в конце каждого эпизода. Как вычисляются градиенты, мы объясним в следующем разделе. Кроме того, на каждом шаге алгоритм производит **выборку** из стратегии на основе вероятностей, вычисленных с использовани-

ем состояний и весов. Теперь выбираемое действие определено не однозначно, как при случайном поиске и восхождении на вершину (когда выбирается действие с большей числовой оценкой). Таким образом, стратегия перестает быть детерминированной, а становится **стохастической**.

Как это делается

Реализуем алгоритм градиента стратегии с помощью PyTorch.

1. Как и прежде, импортируем необходимые пакеты, создадим экземпляр окружающей среды и получим размерности пространства наблюдений и действий.

```
>>> import gym
>>> import torch
>>> env = gym.make('CartPole-v0')
>>> n_state = env.observation_space.shape[0]
>>> n_action = env.action_space.n
```

2. Определим функцию `run_episode`, которая получает на входе веса, имитирует эпизод и возвращает полное вознаграждение и градиенты. Точнее, на каждом шаге она выполняет следующие действия:
   - вычисляет вероятности `probs` обоих действий, зная текущее состояние и входные веса;
   - выбирает действие `action` в соответствии с вычисленными вероятностями;
   - вычисляет производные `d_softmax` функции `softmax`, которой передаются вероятности;
   - делит вычисленные производные `d_softmax` на вероятности и получает производные `d_log` логарифма стратегии;
   - применяет правило дифференцирования сложной функции, чтобы вычислить градиент `grad` по весам;
   - запоминает результирующий градиент `grad`;
   - выполняет действие, увеличивает полное вознаграждение и обновляет состояние.

   Ниже приведен соответствующий код:

```
>>> def run_episode(env, weight):
...     state = env.reset()
...     grads = []
...     total_reward = 0
...     is_done = False
...     while not is_done:
...         state = torch.from_numpy(state).float()
...         z = torch.matmul(state, weight)
...         probs = torch.nn.Softmax()(z)
...         action = int(torch.bernoulli(probs[1]).item())
...         d_softmax = torch.diag(probs) -
                            probs.view(-1, 1) * probs
...         d_log = d_softmax[action] / probs[action]
```

```
...         grad = state.view(-1, 1) * d_log
...         grads.append(grad)
...         state, reward, is_done, _ = env.step(action)
...         total_reward += reward
...         if is_done:
...             break
...     return total_reward, grads
```

После завершения эпизода функция возвращает полное вознаграждение и градиенты, вычисленные на каждом шаге. Эти значения понадобятся для обновления весов.

3. Пусть число эпизодов будет равно 1000:

```
>>> n_episode = 1000
```

Это означает, что функция `run_episode` будет выполнена `n_episode` раз.

4. Инициализируем матрицу весов `weight`:

```
>>> weight = torch.rand(n_state, n_action)
```

Будем запоминать полное вознаграждение в каждом эпизоде:

```
>>> total_rewards = []
```

5. В конце каждого эпизода необходимо обновить веса с учетом вычисленных градиентов. На каждом шаге эпизода вес изменяется на величину *скорость обучения* * *градиент*, вычисленный на этом шаге, * *полное вознаграждение* на оставшихся шагах. Скорость обучения примем равной 0.001:

```
>>> learning_rate = 0.001
```

Теперь прогоним `n_episode` эпизодов:

```
>>> for episode in range(n_episode):
...     total_reward, gradients = run_episode(env, weight)
...     print('Эпизод {}: {}'.format(episode + 1, total_reward))
...     for i, gradient in enumerate(gradients):
...         weight += learning_rate * gradient * (total_reward - i)
...     total_rewards.append(total_reward)
.....
.....
Эпизод 101: 200.0
Эпизод 102: 200.0
Эпизод 103: 200.0
Эпизод 104: 190.0
Эпизод 105: 133.0
.....
.....
Эпизод 996: 200.0
Эпизод 997: 200.0
Эпизод 998: 200.0
Эпизод 999: 200.0
Эпизод 1000: 200.0
```

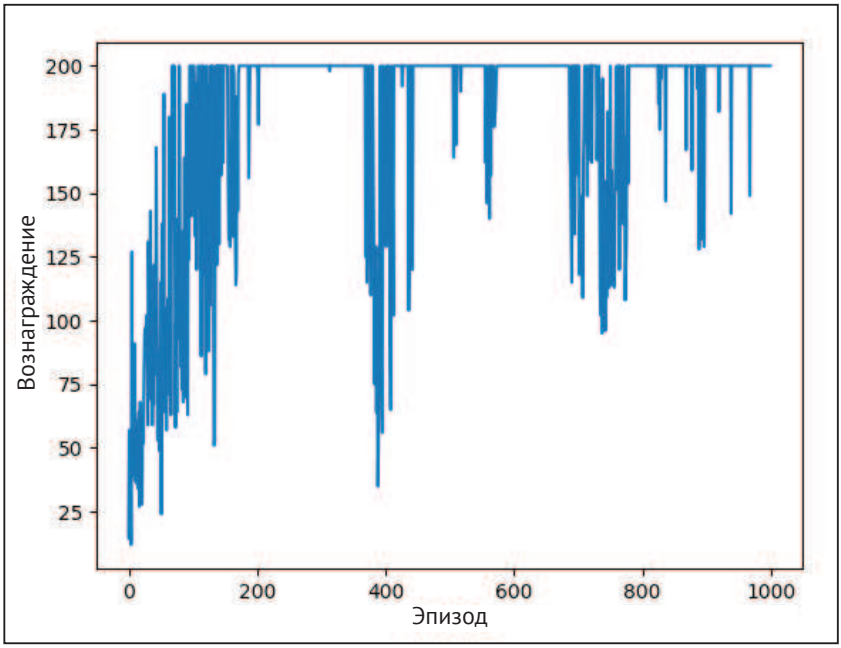
6. Вычислим среднее полное вознаграждение, полученное в алгоритме градиента стратегии:

```
>>> print('Среднее полное вознаграждение в {} эпизодах: {}'.format(
        n_episode, sum(total_rewards) / n_episode))
Среднее полное вознаграждение в 1000 эпизодов: 179.728
```

7. Построим график зависимости полного вознаграждения от номера эпизода:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(total_rewards)
>>> plt.xlabel('Эпизод')
>>> plt.ylabel('Вознаграждение')
>>> plt.show()
```

На графике отчетливо виден восходящий тренд с выходом на плато, соответствующее максимальному значению.



8. Теперь посмотрим, как обученная стратегия поведет себя в 100 новых эпизодах.

```
>>> n_episode_eval = 100
>>> total_rewards_eval = []
>>> for episode in range(n_episode_eval):
...     total_reward, _ = run_episode(env, weight)
...     print('Эпизод {}: {}'.format(episode+1, total_reward))
...     total_rewards_eval.append(total_reward)
...
```

```
Эпизод 1: 200.0
Эпизод 2: 200.0
Эпизод 3: 200.0
Эпизод 4: 200.0
Эпизод 5: 200.0
……
……
Эпизод 96: 200.0
Эпизод 97: 200.0
Эпизод 98: 200.0
Эпизод 99: 200.0
Эпизод 100: 200.0
```

Вычислим среднее полное вознаграждение:

```
>>> print('Среднее полное вознаграждение в {} эпизодах: {}'.
format(n_episode, sum(total_rewards) / n_episode))
Среднее полное вознаграждение в 1000 эпизодов: 199.78
```

Как видим, среднее полное вознаграждение в тестовых эпизодах близко к максимальному значению 200, полученному при следовании обученной стратегии. Можете повторить эксперимент несколько раз – результаты мало разнятся.

Как это работает

В алгоритме градиента стратегии для обучения агента выполняются небольшие шаги, и в конце эпизода веса обновляются в соответствии с вознаграждениями, полученными на этих шагах. Методика, при которой стратегия обновляется, после того как агент прошел весь эпизод до конца, называется градиентом стратегии **Монте-Карло**.

Действие выбирается на основе распределения вероятностей, вычисленного по текущему состоянию и весам модели. Например, если вероятности действий «влево» и «вправо» равны соответственно 0.6 и 0.4, то действие «влево» выбирается в 60 % случаев; это не означает, что обязательно будет выбрано действие «влево», как в алгоритмах случайного поиска и восхождения на вершину.

Мы знаем, что за каждый шаг до завершения эпизода начисляется вознаграждение 1. Поэтому будущее вознаграждение, нужное для вычисления градиента стратегии на каждом шаге, равно числу оставшихся шагов. После каждого эпизода мы используем историю градиента, умноженную на будущее вознаграждение, чтобы обновить веса с применением метода стохастического градиентного подъема. Поэтому чем длиннее эпизод, тем сильнее обновляются веса. В итоге повышается шанс на получение большего полного вознаграждения.

В начале этого раздела мы говорили, что алгоритм градиента стратегии – перебор для такой простой среды, как CartPole, но зато теперь мы готовы к решению более трудных задач.

Это еще не все

Посмотрев на график зависимости вознаграждения от количества эпизодов, можно прийти к выводу, что обучение можно остановить раньше, как только задача будет решена, т. е. среднее полное вознаграждение в 100 последовательных эпизодах окажется не меньше 195. Для этого нужно добавить в код обучения такие строчки:

```
>>> if episode >= 99 and sum(total_rewards[-100:]) >= 19500:
...     break
```

Еще раз выполните обучение. В результате обучение должно прекратиться после нескольких сотен эпизодов:

```
Эпизод 1: 10.0
Эпизод 2: 27.0
Эпизод 3: 28.0
Эпизод 4: 15.0
Эпизод 5: 12.0
.....
.....
Эпизод 549: 200.0
Эпизод 550: 200.0
Эпизод 551: 200.0
Эпизод 552: 200.0
Эпизод 553: 200.0
```

См. также

Дополнительные сведения о методах градиента стратегии см. на странице http://www.scholarpedia.org/article/Policy_gradient_methods.

Глава 2

Марковские процессы принятия решений и динамическое программирование

В этой главе мы, вооружившись PyTorch, продолжим путешествие в мир обучения с подкреплением и рассмотрим **марковские процессы принятия решений** (**МППР**) и динамическое программирование. Мы начнем с создания марковской цепи и МППР, лежащего в основе большинства алгоритмов ОП. На примере оценивания стратегии мы познакомимся с уравнением Беллмана. А затем применим два подхода к решению МППР: итерацию по ценности и итерацию по стратегиям. В качестве примера воспользуемся окружающей средой FrozenLake. В конце главы будет продемонстрировано применение динамического программирования к решению азартной игры с подбрасыванием монеты.

В этой главе приводятся следующие рецепты:

- создание марковской цепи;
- оценивание стратегии;
- имитация окружающей среды FrozenLake;
- решение МППР с помощью алгоритма итерации по ценности;
- решение МППР с помощью алгоритма итерации по стратегиям;
- игра с подбрасыванием монеты.

Технические требования

Для выполнения рецептов в этой главе понадобятся:

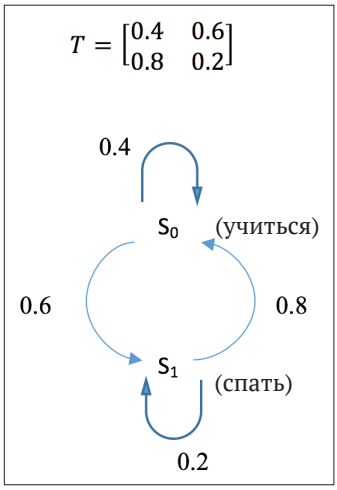
- Python версии 3.6, 3.7 или более поздней;
- Anaconda;

- PyTorch версии 1.0 или более поздней;
- OpenAI Gym.

Создание марковской цепи

Для начала создадим марковскую цепь, на базе которой разработаем МППР.

Марковская цепь описывает последовательность событий, обладающую **марковским свойством**. Она состоит из множества допустимых состояний $S = \{s0, s1, \ldots, sm\}$ и матрицы переходов $T(s, s')$, элементами которой являются вероятности перехода из состояния s в состояние s'. Марковское свойство означает, что будущее состояние процесса зависит только от его текущего состояния и не зависит от прошлых состояний. Иными словами, состояние процесса в момент $t + 1$ зависит только от состояния в момент t. В примере ниже процесс имеет два состояния – учиться и спать, в марковской цепи они обозначены *s0* и *s1* соответственно. Предположим, что матрица переходов выглядит следующим образом:



В следующем разделе мы вычислим матрицу переходов после k шагов. Вероятности, заданные в начальном распределении состояний, например [0.7, 0.3], означают, что в 70 % случаев процесс начинается в состоянии «учиться», а в 30 % – в состоянии «спать».

Как это делается

Чтобы создать и проанализировать марковскую цепь для процесса «учиться–спать», выполним следующие действия.

1. Импортируем библиотеку и определим матрицу переходов:

```
>>> import torch
>>> T = torch.tensor([[0.4, 0.6],
...                   [0.8, 0.2]])
```

2. Вычислим вероятности переходов после k шагов, взяв k = 2, 5, 10, 15 и 20:

```
>>> T_2 = torch.matrix_power(T, 2)
>>> T_5 = torch.matrix_power(T, 5)
>>> T_10 = torch.matrix_power(T, 10)
>>> T_15 = torch.matrix_power(T, 15)
>>> T_20 = torch.matrix_power(T, 20)
```

3. Определим начальное распределение двух состояний:

```
>>> v = torch.tensor([[0.7, 0.3]])
```

4. Вычислим распределение вероятностей состояний после k = 1, 2, 5, 10, 15, 20 шагов:

```
>>> v_1 = torch.mm(v, T)
>>> v_2 = torch.mm(v, T_2)
>>> v_5 = torch.mm(v, T_5)
>>> v_10 = torch.mm(v, T_10)
>>> v_15 = torch.mm(v, T_15)
>>> v_20 = torch.mm(v, T_20)
```

Как это работает

На шаге 2 мы вычисляем вероятности переходов после k шагов, для чего используем матрицу переходов в k-й степени.

```
>>> print("Вероятность перехода после 2 шагов:\n{}".format(T_2))
Вероятность перехода после 2 шагов:
tensor([[0.6400, 0.3600],
        [0.4800, 0.5200]])
>>> print("Вероятность перехода после 5 шагов:\n{}".format(T_5))
Вероятность перехода после 5 шагов:
tensor([[0.5670, 0.4330],
        [0.5773, 0.4227]])
>>> print("Вероятность перехода после 10 шагов:\n{}".format(T_10))
Вероятность перехода после 10 шагов:
tensor([[0.5715, 0.4285],
        [0.5714, 0.4286]])
>>> print("Вероятность перехода после 15 шагов:\n{}".format(T_15))
Вероятность перехода после 15 шагов:
tensor([[0.5714, 0.4286],
         [0.5714, 0.4286]])
>>> print("Вероятность перехода после 20 шагов:\n{}".format(T_20))
Вероятность перехода после 20 шагов:
tensor([[0.5714, 0.4286],
        [0.5714, 0.4286]])
```

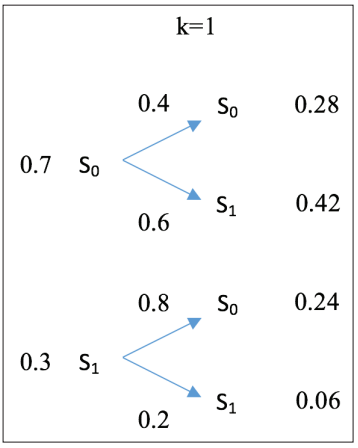
Видно, что после 10–15 шагов вероятности переходов сходятся. Это означает, что вне зависимости от того, в каком состоянии находится процесс, вероятности перехода в состояния s0 и s1 равны соответственно 57.14 % и 42.86 %.

На шаге 4 мы вычислили распределение вероятностей состояний после $k = 1, 2, 5, 10, 15, 20$ шагов, оно равно произведению начального распределения и соответствующей степени матрицы переходов. Приведем результаты:

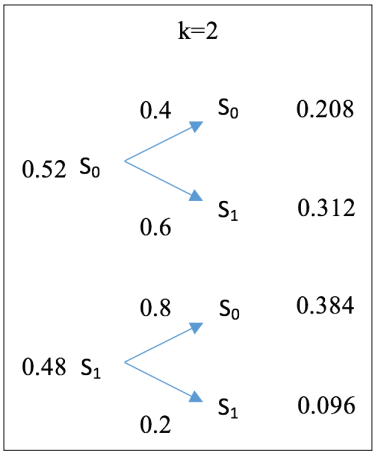
```
>>> print("Распределение состояний после 1 шага:\n{}".format(v_1))
Распределение состояний после 1 шага:
tensor([[0.5200, 0.4800]])
>>> print("Распределение состояний после 2 шагов:\n{}".format(v_2))
Распределение состояний после 2 шагов:
tensor([[0.5920, 0.4080]])
>>> print("Распределение состояний после 5 шагов:\n{}".format(v_5))
Распределение состояний после 5 шагов:
tensor([[0.5701, 0.4299]])
>>> print("Распределение состояний после 10 шагов:\n{}".format(v_10))
Распределение состояний после 10 шагов:
tensor([[0.5714, 0.4286]])
>>> print("Распределение состояний после 15 шагов:\n{}".format(v_15))
Распределение состояний после 15 шагов:
tensor([[0.5714, 0.4286]])
>>> print("Распределение состояний после 20 шагов:\n{}".format(v_20))
Распределение состояний после 20 шагов:
tensor([[0.5714, 0.4286]])
```

Видно, что после 10 шагов распределение сходится. Вероятность в конечном итоге оказаться в состоянии s0 равна 57.14 %, а в состоянии s1 – 42.86 %.

Если начальное распределение вероятностей равно [0.7, 0.3], то после одной итерации оно становится равно [0.52, 0.48]. Детали вычисления показаны на рисунке ниже.



После еще одной итерации распределение принимает вид [0.592, 0.408], как показано на следующем рисунке:



И со временем распределение вероятностей состояний стабилизируется.

Это еще не все

На самом деле независимо от начального состояния процесса распределение вероятностей состояний всегда сходится к [0.5714, 0.4286]. Можете проверить это для других начальных распределений, например [0.2, 0.8] или [1, 0]. После 10 шагов мы придем к распределению [0.5714, 0.4286].

Марковская цепь необязательно сходится, особенно когда содержит переходные состояния. Но если она сходится, то конечное распределение не зависит от начального.

См. также

Дополнительные сведения о марковских цепях можно получить из следующих двух великолепных статей, содержащих красивые визуализации:

- https://brilliant.org/wiki/markov-chains/;
- http://setosa.io/ev/markov-chains/.

Создание МППР

МППР основан на марковской цепи и включает агента и процесс принятия решений. Сейчас мы разработаем МППР и вычислим функцию ценности при оптимальной стратегии.

Помимо множества допустимых состояний $S = \{s0, s1, \ldots, sm\}$, в определение МППР входит множество действий $A = \{a0, a1, \ldots, an\}$, модель переходов $T(s, a, s')$, функция вознаграждения $R(s)$ и коэффициент обесценивания γ. Матрица переходов $T(s, a, s')$ содержит вероятности выбора действия a в состоянии s, которое переводит процесс в состояние s'. Коэффициент обесценивания γ определяет компромисс между вознаграждениями в ближайшем и отдаленном будущем.

Чтобы немного усложнить наш МППР, добавим в процесс «учиться–спать» еще одно состояние – «играть в игры», *s2*. И пусть имеется два действия: *a0* (работать) и *a1* (отдыхать). Матрица переходов $T(s, a, s')$ размера *3 * 2 * 3* имеет вид:

$$T = \begin{cases} \begin{bmatrix} 0.8 & 0.1 & 0.1 \\ 0.1 & 0.6 & 0.3 \end{bmatrix} \\ \begin{bmatrix} 0.7 & 0.2 & 0.1 \\ 0.1 & 0.8 & 0.1 \end{bmatrix} \\ \begin{bmatrix} 0.6 & 0.2 & 0.2 \\ 0.1 & 0.4 & 0.5 \end{bmatrix} \end{cases}$$

Это означает, что если, например, вы выберете действие *a1* в состоянии *s0*, то с вероятностью 60 % перейдете в состояние *s1* (спать – быть может, вы утомились), с вероятностью 30 % – в состояние *s2* (хотите расслабиться и поиграть) и с вероятностью 10 % продолжите учиться (не иначе трудоголик). Определим функцию вознаграждения [+1, 0, –1], которая поощряет тяжкий труд. Очевидно, что в этом случае **оптимальная стратегия** состоит в том, чтобы на каждом шаге выбирать действие *a0* (продолжай учиться, ведь без труда не выловишь и рыбку из пруда). Для начала положим коэффициент обесценивания равным 0.5. В следующем разделе мы вычислим **функцию ценности состояний** (ее также называют просто **функцией ценности**, или, для краткости, **ценностью** или **ожидаемой полезностью**) при следовании оптимальной стратегии.

Как это делается

Для создания МППР выполним следующие шаги.

1. Импортируем библиотеку PyTorch и определим матрицу переходов:

```
>>> import torch
>>> T = torch.tensor([[[0.8, 0.1, 0.1],
...                    [0.1, 0.6, 0.3]],
...                   [[0.7, 0.2, 0.1],
...                    [0.1, 0.8, 0.1]],
...                   [[0.6, 0.2, 0.2],
...                    [0.1, 0.4, 0.5]]]
...                  )
```

2. Определим функцию вознаграждения и коэффициент обесценивания:

```
>>> R = torch.tensor([1., 0, -1.])
>>> gamma = 0.5
```

3. В данном случае оптимальная стратегия – всегда выбирать действие *a0*:

```
>>> action = 0
```

4. Вычисляем ценность V оптимальной стратегии, вычисляя обратную матрицу:

```
>>> def cal_value_matrix_inversion(gamma, trans_matrix, rewards):
...     inv = torch.inverse(torch.eye(rewards.shape[0])
                                    - gamma * trans_matrix)
...     V = torch.mm(inv, rewards.reshape(-1, 1))
...     return V
```

5. Подаем на вход этой функции все переменные, включая вероятности переходов, ассоциированные с действием *a0*:

```
>>> trans_matrix = T[:, action]
>>> V = cal_value_matrix_inversion(gamma, trans_matrix, R)
>>> print("Функция ценности при оптимальной стратегии:\n{}".format(V))
Функция ценности при оптимальной стратегии:
tensor([[ 1.6787],
        [ 0.6260],
        [-0.4820]])
```

Как это работает

В этом до предела упрощенном процессе «учиться–спать–играть» оптимальная стратегия, т. е. стратегия, при которой достигается максимальное полное вознаграждение, состоит в том, чтобы на каждом шаге выбирать действие *a0*. Однако в большинстве случаев все не так просто. Действия, выбираемые на разных шагах, необязательно совпадают. Обычно они зависят от состояния. Поэтому нам приходится решать МППР, т. е. находить оптимальную стратегию в реальных ситуациях.

Функция ценности стратегии измеряет, насколько агенту выгодно находиться в каждом состоянии при следовании данной стратегии. Чем выше ценность, тем лучше состояние.

На шаге 4 мы вычислили ценность V оптимальной стратегии, воспользовавшись **обращением матрицы**. Согласно **уравнению Беллмана**, соотношение между ценностью на шаге $t + 1$ и на шаге t имеет вид:

$$V_{t+1} = R + \gamma * T * V_t.$$

Когда ценность сойдется, т. е. $V_{t+1} = V_t$, мы сможем получить значение V следующим образом:

$$V = R + \gamma * T * V;$$
$$V = (I - \gamma * T)^{-1} * R.$$

Здесь I – единичная матрица, содержащая единицы на главной диагонали и нули во всех остальных позициях.

У решения МППР с помощью обращения матрицы есть важное достоинство – мы всегда получаем точный ответ. Но есть и недостаток – отсутствие масштабируемости. Если число состояний m велико, то обращение матрицы размером $m \times m$ обходится дорого с точки зрения объема вычислений.

Это еще не все

Давайте поэкспериментируем с различными значениями коэффициента обесценивания, начав с 0, – это означает, что нас интересует только непосредственное вознаграждение.

```
>>> gamma = 0
>>> V = cal_value_matrix_inversion(gamma, trans_matrix, R)
>>> print("Функция ценности при оптимальной стратегии:\n{}".format(V))
Функция ценности при оптимальной стратегии:
tensor([[ 1.],
        [ 0.],
        [-1.]])
```

Это совпадает с функцией вознаграждения, потому что мы рассматриваем только вознаграждение, полученное на следующем ходе.

Когда коэффициент обесценивания увеличивается, приближаясь к 1, начинают учитываться будущие вознаграждения. Возьмем, к примеру, $\gamma = 0.99$:

```
>>> gamma = 0.99
>>> V = cal_value_matrix_inversion(gamma, trans_matrix, R)
>>> print("Функция ценности при оптимальной стратегии:\n{}".format(V))
Функция ценности при оптимальной стратегии:
tensor([[65.8293],
        [64.7194],
        [63.4876]])
```

См. также

Шпаргалка по адресу https://cs-cheatsheet.readthedocs.io/en/latest/subjects/ai/mdp.html может служить кратким справочником по МППР.

Оценивание стратегии

Выше мы разработали МППР и вычислили функцию ценности оптимальной стратегии с помощью обращения матрицы. Мы также отметили ограничение этого подхода при больших m (порядка 1000, 10 000 или 100 000). В этом рецепте будет рассмотрен более простой подход – **оценивание стратегии**.

Оценивание стратегии – итеративный алгоритм. Мы начинаем с произвольных ценностей, а затем итеративно улучшаем их, опираясь на **уравнение математического ожидания Беллмана**, добиваясь сходимости. На каждой итерации ценность состояния s при следовании стратегии π обновляется по формуле:

$$V(s) := \sum_a \pi(s,a)\left[R(s,a) + \gamma \sum_{s'} T(s,a,s')V(s')\right].$$

Здесь $\pi(s, a)$ обозначает вероятность выбора действия a в состоянии s при следовании стратегии π, $T(s, a, s')$ – вероятность перехода из состояния s в со-

стояние s' в результате выбора действия a, а $R(s, a)$ – вознаграждение, полученное в состоянии s при выборе действия a.

Остановить итеративный процесс обновления можно двумя способами. Первый – задать фиксированное число итераций, скажем 1000 или 10 000, но подобрать правильное значение бывает трудно. Второй – задать пороговое значение (обычно 0.0001, 0.00001 или что-то в этом роде) и прекращать процесс, когда ценности всех состояний изменяются на величину, меньшую порога.

В следующем разделе мы выполним оценивание стратегии в процессе «учиться–спать–играть» для оптимальной и случайной стратегий.

Как это делается

Разработаем алгоритм оценивания стратегии и применим его к процессу «учиться–спать–играть».

1. Импортируем библиотеку PyTorch и определим матрицу переходов:

```
>>> import torch
>>> T = torch.tensor([[[0.8, 0.1, 0.1],
...                    [0.1, 0.6, 0.3]],
...                   [[0.7, 0.2, 0.1],
...                    [0.1, 0.8, 0.1]],
...                   [[0.6, 0.2, 0.2],
...                    [0.1, 0.4, 0.5]]]
... )
```

2. Определим функцию вознаграждения и коэффициент обесценивания (положим его равным 0.5):

```
>>> R = torch.tensor([1., 0, -1.])
>>> gamma = 0.5
```

3. Определим порог остановки процесса оценивания:

```
>>> threshold = 0.0001
```

4. Определим оптимальную стратегию, при которой всегда выбирается действие *a0*:

```
>>> policy_optimal = torch.tensor([[1.0, 0.0],
...                                [1.0, 0.0],
...                                [1.0, 0.0]])
```

5. Создадим функцию оценивания стратегии, которая принимает стратегию, матрицу переходов, вознаграждения, коэффициент обесценивания и порог и вычисляет ценность.

```
>>> def policy_evaluation(
                policy, trans_matrix, rewards, gamma, threshold):
...     """
...     Оценивает стратегию
...     @param policy: матрица, содержащая вероятности выбора действий
                       в каждом состоянии
```

```
...     @param trans_matrix: матрица переходов
...     @param rewards: вознаграждения в каждом состоянии
...     @param gamma: коэффициент обесценивания
...     @param threshold: оценивание прекращается, как только изменение
                          ценностей всех состояний оказывается меньше порога
...     @return: ценности всех состояний при следовании данной стратегии
...     """
...     n_state = policy.shape[0]
...     V = torch.zeros(n_state)
...     while True:
...         V_temp = torch.zeros(n_state)
...         for state, actions in enumerate(policy):
...             for action, action_prob in enumerate(actions):
...                 V_temp[state] += action_prob * (R[state] +
                        gamma * torch.dot(trans_matrix[state, action], V))
...         max_delta = torch.max(torch.abs(V - V_temp))
...         V = V_temp.clone()
...         if max_delta <= threshold:
...             break
...     return V
```

6. Подставим сюда оптимальную стратегию и все остальные параметры:

```
>>> V = policy_evaluation(policy_optimal, T, R, gamma, threshold)
>>> print(
"Функция ценности при оптимальной стратегии:\n{}".format(V))
Функция ценности при оптимальной стратегии:
tensor([ 1.6786, 0.6260, -0.4821])
```

   Получилось почти то же самое, что при обращении матрицы.

7. Теперь возьмем другую, случайную стратегию и зададим для нее одинаковые вероятности выбора действий:

```
>>> policy_random = torch.tensor([[0.5, 0.5],
...                               [0.5, 0.5],
...                               [0.5, 0.5]])
```

8. Подставим случайную стратегию и все остальные параметры:

```
>>> V = policy_evaluation(policy_random, T, R, gamma, threshold)
>>> print("Функция ценности при случайной стратегии:\n{}".format(V))
Функция ценности при случайной стратегии:
tensor([ 1.2348, 0.2691, -0.9013])
```

Как это работает

Мы только что видели, как можно эффективно вычислить функцию ценности с помощью алгоритма оценивания стратегии. Этот простой итеративный подход называется **приближенным динамическим программированием**. Мы начинаем со случайно выбранных ценностей состояний и итеративно обновляем их, применяя уравнение математического ожидания Беллмана, пока не достигнем сходимости.

На шаге 5 функция оценивания стратегии выполняет следующие действия:

- инициализирует ценности всех состояний нулями;
- обновляет ценности в соответствии с уравнением математического ожидания Беллмана;
- вычисляет максимальное изменение ценностей по всем состояниям;
- если максимальное изменение больше порога, то процесс обновления продолжается. В противном случае оценивание завершается, и возвращаются ценности, вычисленные на последней итерации.

Поскольку алгоритм оценивания стратегии приближенный, результат может отличаться от полученного с помощью обращения матрицы. Но нам и не нужна точная функция ценности. Ко всему прочему, описанный алгоритм справляется с **проклятием размерности**, т. е. с масштабированием вычислений на миллиарды состояний. Поэтому обычно алгоритм оценивания стратегии является предпочтительным.

И еще запомните, что оценивание стратегии используется для **предсказания** результатов стратегии, а не для решения задач **управления**.

Это еще не все

Чтобы составить более полное представление об алгоритме, построим график изменения ценностей в зависимости от номера итерации. Для этого нужно включить в функцию `policy_evaluation` запоминание ценностей на каждой итерации.

```
>>> def policy_evaluation_history(
                policy, trans_matrix, rewards, gamma, threshold):
...     n_state = policy.shape[0]
...     V = torch.zeros(n_state)
...     V_his = [V]
...     i = 0
...     while True:
...         V_temp = torch.zeros(n_state)
...         i += 1
...         for state, actions in enumerate(policy):
...             for action, action_prob in enumerate(actions):
...                 V_temp[state] += action_prob * (R[state] + gamma *
                            torch.dot(trans_matrix[state, action], V))
...         max_delta = torch.max(torch.abs(V - V_temp))
...         V = V_temp.clone()
...         V_his.append(V)
...         if max_delta <= threshold:
...             break
...     return V, V_his
```

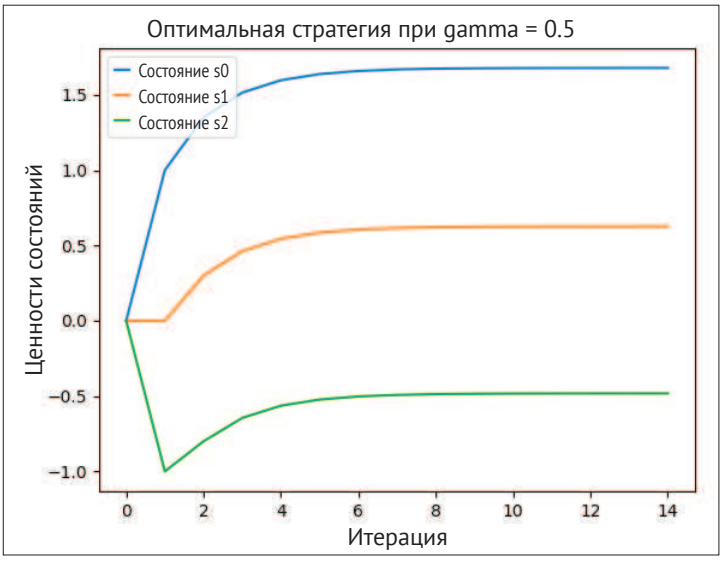
Теперь передадим функции `policy_evaluation_history` оптимальную стратегию, коэффициент обесценивания 0.5 и прочие параметры:

```
>>> V, V_history = policy_evaluation_history(
                        policy_optimal, T, R, gamma, threshold)
```

И нанесем историю изменения ценностей на график:

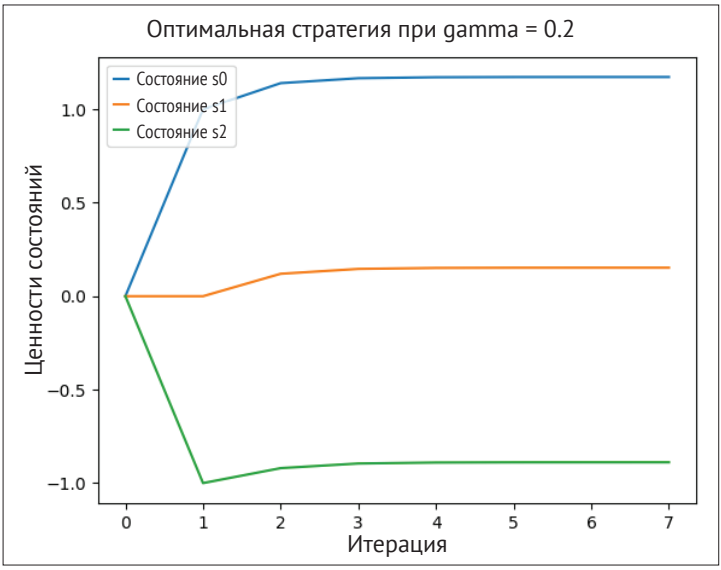
```
>>> import matplotlib.pyplot as plt
>>> s0, = plt.plot([v[0] for v in V_history])
>>> s1, = plt.plot([v[1] for v in V_history])
>>> s2, = plt.plot([v[2] for v in V_history])
>>> plt.title('Оптимальная стратегия при gamma = {}'.format(str(gamma)))
>>> plt.xlabel('Итерация')
>>> plt.ylabel('Ценности состояний')
>>> plt.legend([s0, s1, s2],
...            ["State s0",
...             "State s1",
...             "State s2"], loc="upper left")
>>> plt.show()
```

Ниже показан результат:



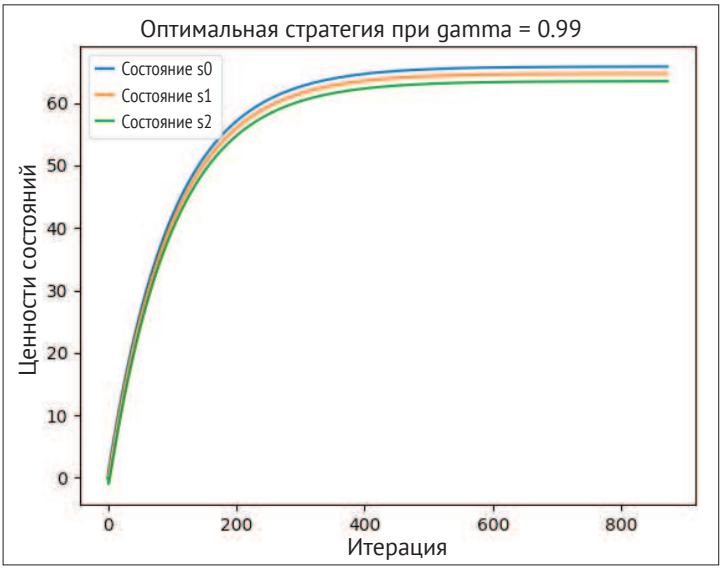
Как видим, стабилизация происходит на итерации с номером от 10 до 14.

Далее выполним тот же код, но с другими коэффициентами обесценивания: 0.2 и 0.99. Для коэффициента 0.2 получается такой график:



Сравнивая с предыдущим, мы видим, что чем меньше коэффициент, тем быстрее сходятся ценности состояний.

Для коэффициента 0.99 получается такой график:



Сравнивая его с предыдущими, мы приходим к выводу, что чем коэффициент больше, тем больше времени необходимо, чтобы ценности сошлись. Коэффициент обесценивания – это компромисс между непосредственным и отдаленным вознаграждениями.

Имитация окружающей среды FrozenLake

Те оптимальные стратегии решения МППР, которые мы рассматривали до сих пор, интуитивно довольно очевидны. Но в большинстве случаев это не так, в чем мы убедимся на примере окружающей среды FrozenLake, который подготовит нас к следующим рецептам.

FrozenLake – типичная окружающая среда Gym с **дискретным** пространством состояний. Задача заключается в том, чтобы переместить агента из начального положения в конечное в сеточном мире, избегая расставленных на пути ловушек. Сетка имеет размер 4×4 (https://gym.openai.com/envs/FrozenLake-v0/) или 8×8 (https://gym.openai.com/envs/FrozenLake8x8-v0/). В сетке могут встречаться ячейки следующих типов:

- **S**: начальное положение;
- **G**: конечное положение, в котором эпизод завершается;
- **F**: замерзшее озеро, по которому можно ходить;
- **H**: полынья, в которой эпизод завершается.

Определены четыре действия: влево (0), вниз (1), вправо (2) и вверх (3). Агенту начисляется вознаграждение +1, если он успешно доберется до цели, и 0 в противном случае. Пространство наблюдений представлено массивом из 16 целых чисел, а возможных действий четыре (естественно).

У этой среды есть особенность, осложняющая обучение: поскольку лед скользкий, агент не всегда движется туда, куда собирался. Например, он может сдвинуться влево или вправо, хотя намеревался идти вниз.

Подготовка

Для экспериментов со средой FrozenLake сначала отыщем ее в таблице окружающих сред на странице https://github.com/openai/gym/wiki/Table-of-environments. Эта среда называется `FrozenLake-v0`.

Как это делается

Для имитации среды FrozenLake размера 4×4 выполним следующие действия.

1. Импортируем библиотеку `gym` и создадим экземпляр среды FrozenLake:

```
>>> import gym
>>> import torch
>>> env = gym.make("FrozenLake-v0")
>>> n_state = env.observation_space.n
>>> print(n_state)
16
>>> n_action = env.action_space.n
>>> print(n_action)
4
```

2. Переведем окружающую среду в исходное состояние:

```
>>> env.reset()
0
```

   Агент начинает работу в состоянии 0.
3. Нарисуем окружающую среду:

```
>>> env.render()
```

4. Сделаем шаг вниз, это возможно:

```
>>> new_state, reward, is_done, info = env.step(1)
>>> env.render()
```

5. Распечатаем все возвращенные данные и убедимся, что агент оказывается в состоянии 4 с вероятностью 33.33 %:

```
>>> print(new_state)
4
>>> print(reward)
0.0
>>> print(is_done)
False
>>> print(info)
{'prob': 0.3333333333333333}
```

   Полученное вознаграждение равно 0, потому что мы еще не достигли цели, а флаг `is_done` равен `False`, потому что эпизод еще не закончен. Мы видим, что агент переходит в состояние 1 или остается в состоянии 0 из-за скользкой поверхности.

6. Чтобы продемонстрировать, как трудно ходить по замерзшему озеру, реализуем случайную стратегию и вычислим среднее полное вознаграждение в 1000 эпизодов. Сначала напишем функцию, которая имитирует один эпизод в среде FrozenLake с заданной стратегией и возвращает полное вознаграждение (мы знаем, что оно равно 0 или 1).

```
>>> def run_episode(env, policy):
...     state = env.reset()
...     total_reward = 0
...     is_done = False
...     while not is_done:
...         action = policy[state].item()
...         state, reward, is_done, info = env.step(action)
...         total_reward += reward
...         if is_done:
...             break
...     return total_reward
```

7. Теперь выполним 1000 эпизодов. В каждом эпизоде будем генерировать и использовать случайную стратегию.

```
>>> n_episode = 1000
>>> total_rewards = []
>>> for episode in range(n_episode):
```

```
...     random_policy = torch.randint(
                        high=n_action, size=(n_state,))
...     total_reward = run_episode(env, random_policy)
...     total_rewards.append(total_reward)
...
>>> print('Среднее полное вознаграждение при случайной стратегии: {}'.format(
          sum(total_rewards) / n_episode))
Среднее полное вознаграждение при случайной стратегии: 0.014
```

Это означает, что в среднем вероятность достичь цели при случайном выборе действий составляет всего 1.4 %.

8. Теперь поэкспериментируем со стратегией случайного поиска. На этапе обучения мы случайно генерируем несколько стратегий и запоминаем ту, которая первой достигает цели:

```
>>> while True:
...     random_policy = torch.randint(high=n_action, size=(n_state,))
...     total_reward = run_episode(env, random_policy)
...     if total_reward == 1:
...         best_policy = random_policy
...         break
```

9. Распечатаем наилучшую стратегию:

```
>>> print(best_policy)
tensor([0, 3, 2, 2, 0, 2, 1, 1, 3, 1, 3, 0, 0, 1, 1, 1])
```

10. Теперь выполним 1000 эпизодов с этой стратегией:

```
>>> total_rewards = []
>>> for episode in range(n_episode):
...     total_reward = run_episode(env, best_policy)
...     total_rewards.append(total_reward)
...
>>> print('Среднее полное вознаграждение при случайной стратегии: {}'.
       format(sum(total_rewards) / n_episode))
Среднее полное вознаграждение при случайной стратегии: 0.208
```

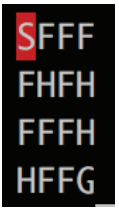
В среднем алгоритм случайного поиска достигает цели в 20.8 % случаев.

Заметим, что результат может сильно варьироваться, потому что не исключено, что выбранная нами стратегия не оптимальна, а цели мы достигли только благодаря скользкому льду.

Как это работает

В этом рецепте мы случайно сгенерировали стратегию, содержащую 16 действий для 16 состояний. Не забывайте, что в среде FrozenLake направление движения лишь отчасти определяется выбранным действием. Это повышает неопределенность управления.

После выполнения кода на шаге 4 будет напечатана следующая матрица 4×4, представляющая замерзшее озеро и ячейку (состояние 0), в которой находится агент:



После выполнения шага 5 сетка будет отражать перемещение агента вниз в состояние 4:

Эпизод завершается, когда выполнено одно из двух условий:

- агент попал в одну из ячеек H (состояния 5, 7, 11, 12). При этом полное вознаграждение будет равно 0;
- агент попал в ячейку G (состояние 15). При этом полное вознаграждение будет равно +1.

Это еще не все

Чтобы детально изучить окружающую среду FrozenLake, включая матрицу переходов и вознаграждения для каждой пары состояние–действие, можно воспользоваться атрибутом P. Например, для состояния 6 это выглядит следующим образом:

```
>>> print(env.env.P[6])
{0: [(0.3333333333333333, 2, 0.0, False), (0.3333333333333333, 5, 0.0,
True), (0.3333333333333333, 10, 0.0, False)], 1: [(0.3333333333333333, 5,
0.0, True), (0.3333333333333333, 10, 0.0, False), (0.3333333333333333, 7,
0.0, True)], 2: [(0.3333333333333333, 10, 0.0, False), (0.3333333333333333,
7, 0.0, True), (0.3333333333333333, 2, 0.0, False)], 3:
[(0.3333333333333333, 7, 0.0, True), (0.3333333333333333, 2, 0.0, False),
(0.3333333333333333, 5, 0.0, True)]}
```

Возвращается словарь, содержащий ключи 0, 1, 2 и 3, представляющие четыре действия. Значением, ассоциированным с ключом, является список перемещений при выборе соответствующего действия в формате: (вероятность перехода, новое состояние, полученное вознаграждение, флаг завершения). Например, если агент находится в состоянии 6 и выбирает действие 1 (вниз), то с вероятностью 33.33 % он окажется в состоянии 5, получив при этом вознаграждение 0 – эпизод на этом завершится. С вероятностью 33.33 % он ока-

жется в состоянии 10, получив вознаграждение 0, и с вероятностью 33.33 % – в состоянии 7, получив вознаграждение 0 и завершив эпизод.

Для состояния 11 результат будет такой:

```
>>> print(env.env.P[11])
{0: [(1.0, 11, 0, True)], 1: [(1.0, 11, 0, True)], 2: [(1.0, 11, 0, True)],
3: [(1.0, 11, 0, True)]}
```

Поскольку проваливание в полынью завершает эпизод, из этого состояния нет никакого выхода.

Остальные состояния проверьте самостоятельно.

Решение МППР с помощью алгоритма итерации по ценности

МППР считается решенным, если найдена оптимальная стратегия. В этом рецепте мы найдем оптимальную стратегию в окружающей среде FrozenLake, применив алгоритм **итерации по ценности**.

Его идея примерно такая же, как в алгоритме оценивания стратегии. Это еще один итеративный алгоритм. В начале работы ценности состояний произвольны, а затем обновляются с применением **уравнения оптимальности Беллмана**, пока не сойдутся. На каждой итерации вместо вычисления математического ожидания (среднего) ценности по всем действиям выбирается действие, при котором ценность оказывается максимальной:

$$V^*(s) := \max_a \left[R(s,a) + \gamma \sum_{s'} T(s,a,s') V^*(s') \right].$$

Здесь $V^*(s)$ обозначает оптимальную ценность состояния, т. е. ценность при следовании оптимальной стратегии, $T(s, a, s')$ – вероятность перехода из состояния s в состояние s' при выборе действия a, а $R(s, a)$ – вознаграждение, полученное в состоянии s при выборе действия a.

Вычислив оптимальные ценности, мы легко можем получить оптимальную стратегию:

$$\pi^*(s) := \operatorname{argmax}_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')].$$

Как это делается

Для имитации среды FrozenLake с помощью алгоритма итерации по ценности выполним следующие действия.

1. Импортируем необходимые библиотеки и создадим экземпляр среды FrozenLake:

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

2. Зададим коэффициент обесценивания 0.99 и порог сходимости 0.0001:

```
>>> gamma = 0.99
>>> threshold = 0.0001
```

3. Определим функцию, которая вычисляет оптимальные ценности, применяя алгоритм итерации по ценности.

```
>>> def value_iteration(env, gamma, threshold):
...     """
...     Имитирует заданную окружающую среду, применяя алгоритм итерации по
...     ценности
...     @param env: имя окружающей среды OpenAI Gym
...     @param gamma: коэффициент  обесценивания
...     @param threshold: обучение заканчивается, когда ценности всех
...                       состояний будут меньше этого значения
...     @return: ценности состояний для оптимальной стратегии
...     """
...     n_state = env.observation_space.n
...     n_action = env.action_space.n
...     V = torch.zeros(n_state)
...     while True:
...         V_temp = torch.empty(n_state)
...         for state in range(n_state):
...             v_actions = torch.zeros(n_action)
...             for action in range(n_action):
...                 for trans_prob, new_state, reward, _ in
                                    env.env.P[state][action]:
...                     v_actions[action] += trans_prob * (reward
                                        + gamma * V[new_state])
...             V_temp[state] = torch.max(v_actions)
...         max_delta = torch.max(torch.abs(V - V_temp))
...         V = V_temp.clone()
...         if max_delta <= threshold:
...             break
...     return V
```

4. Вызовем эту функцию, передав ей имя окружающей среды, коэффициент обесценивания и порог сходимости, а затем напечатаем оптимальные ценности.

```
>>> V_optimal = value_iteration(env, gamma, threshold)
>>> print('Оптимальные ценности:\n{}'.format(V_optimal))
Оптимальные ценности:
tensor([0.5404, 0.4966, 0.4681, 0.4541, 0.5569, 0.0000, 0.3572, 0.0000, 0.5905,
        0.6421, 0.6144, 0.0000, 0.0000, 0.7410, 0.8625, 0.0000])
```

5. Зная оптимальные ценности, напишем функцию, которая строит по ним оптимальную стратегию:

```
>>> def extract_optimal_policy(env, V_optimal, gamma):
...     """
...     Строит оптимальную стратегию, соответствующую оптимальным ценностям
...     @param env: имя окружающей среды OpenAI Gym
```

```
...     @param V_optimal: оптимальные ценности
...     @param gamma: коэффициент  обесценивания
...     @return: оптимальная стратегия
...     """
...     n_state = env.observation_space.n
...     n_action = env.action_space.n
...     optimal_policy = torch.zeros(n_state)
...     for state in range(n_state):
...         v_actions = torch.zeros(n_action)
...         for action in range(n_action):
...             for trans_prob, new_state, reward, _ in
                                    env.env.P[state][action]:
...                 v_actions[action] += trans_prob * (reward
                                + gamma * V_optimal[new_state])
...         optimal_policy[state] = torch.argmax(v_actions)
...     return optimal_policy
```

6. Вызовем эту функцию, передав ей имя окружающей среды, коэффициент обесценивания и оптимальные ценности, а затем напечатаем оптимальную стратегию.

```
>>> optimal_policy = extract_optimal_policy(env, V_optimal, gamma)
>>> print('Оптимальная стратегия:\n{}'.format(optimal_policy))
Оптимальная стратегия:
tensor([0., 3., 3., 3., 0., 3., 2., 3., 3., 1., 0., 3., 3., 2., 1., 3.])
```

7. Мы хотим измерить, насколько хороша оптимальная стратегия. Поэтому выполним с ней 1000 эпизодов и вычислим среднее вознаграждение. Для этого воспользуемся функцией `run_episode` из предыдущего рецепта:

```
>>> n_episode = 1000
>>> total_rewards = []
>>> for episode in range(n_episode):
...     total_reward = run_episode(env, optimal_policy)
...     total_rewards.append(total_reward)
>>> print('Среднее полное вознаграждение при оптимальной стратегии: {}'.
          format(sum(total_rewards) / n_episode))
Среднее полное вознаграждение при оптимальной стратегии: 0.75
```

Следуя оптимальной стратегии, агент добирается до цели в 75 % случаев. Это лучшее, на что можно рассчитывать, когда лед скользкий.

Как это работает

Алгоритм итерации по ценности находит оптимальную функцию ценности, итеративно применяя уравнение оптимальности Беллмана.

Ниже показана еще одна форма уравнения оптимальности Беллмана, которую можно использовать, когда вознаграждения, начисляемые средой, зависят от нового состояния:

$$V^*(s) := \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')].$$

Здесь $R(s, a, s')$ – вознаграждение, полученное при переходе из состояния s в состояние s' в результате действия a. Поскольку это более подходящая форма, мы воспользовались ей в функции `value_iteration`. На шаге 3 выполняются следующие действия:

- инициализировать все ценности нулями;
- обновить ценности в соответствии с уравнением оптимальности Беллмана;
- вычислить максимальное изменение ценностей по всем состояниям;
- если максимальное изменение больше порога, то процесс обновления продолжается. В противном случае оценивание завершается и возвращаются ценности, вычисленные на последней итерации, которые считаются оптимальными.

Это еще не все

Мы добились частоты успехов 75 % при коэффициенте обесценивания 0.99. А как коэффициент обесценивания влияет на качество алгоритма? Поэкспериментируем с другими значениями, а именно: 0, 0.2, 0.4, 0.6, 0.8, 0.99, 1.

```
>>> gammas = [0, 0.2, 0.4, 0.6, 0.8, .99, 1.]
```

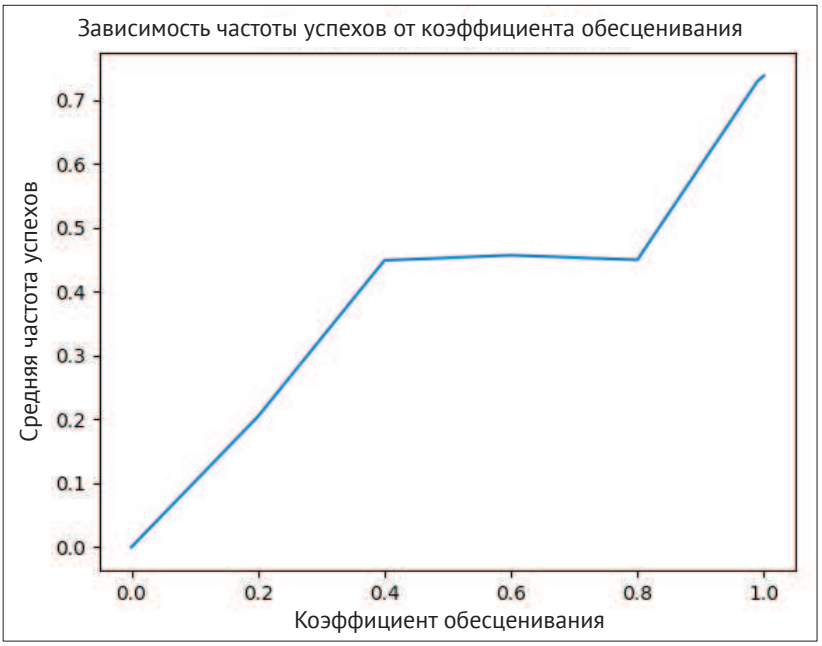
Для каждого коэффициента обесценивания вычислим среднюю частоту успехов в 10 000 эпизодов:

```
>>> avg_reward_gamma = []
>>> for gamma in gammas:
...     V_optimal = value_iteration(env, gamma, threshold)
...     optimal_policy = extract_optimal_policy(env, V_optimal, gamma)
...     total_rewards = []
...     for episode in range(n_episode):
...         total_reward = run_episode(env, optimal_policy)
...         total_rewards.append(total_reward)
...     avg_reward_gamma.append(sum(total_rewards) / n_episode)
```

Построим график зависимости средней частоты успехов от коэффициента обесценивания:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(gammas, avg_reward_gamma)
>>> plt.title('Зависимость частоты успехов от коэффициента обесценивания')
>>> plt.xlabel('Коэффициент обесценивания')
>>> plt.ylabel('Средняя частота успехов')
>>> plt.show()
```

Вот как он выглядит:



Как видим, качество растет по мере увеличения коэффициента обесценивания. Это подтверждает тот факт, что при малом коэффициенте предпочтение отдается непосредственному вознаграждению, а при большом учитываются и будущие вознаграждения.

Решение МППР с помощью алгоритма итерации по стратегиям

Другой подход к решению МППР дает алгоритм **итерации по стратегиям**, который мы и обсудим в этом рецепте.

Алгоритм итерации по стратегиям состоит из двух частей: оценивание стратегии и улучшение стратегии. Вначале стратегия произвольна. На каждой итерации сначала с помощью уравнения математического ожидания Беллмана вычисляются ценности состояний при следовании последней стратегии, а затем с помощью уравнения оптимальности Беллмана на их основе строится улучшенная стратегия. Чередование оценивания и улучшения продолжается до тех пор, пока стратегия не перестанет изменяться.

Сначала реализуем алгоритм итерации по стратегиям для окружающей среды FrozenLake, а затем объясним, как он работает.

Как это делается

Для имитации среды FrozenLake с помощью алгоритма итерации по стратегиям выполним следующие действия.

1. Импортируем необходимые библиотеки и создадим экземпляр среды FrozenLake:

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

2. Зададим коэффициент обесценивания 0.99 и порог сходимости 0.0001:

```
>>> gamma = 0.99
>>> threshold = 0.0001
```

3. Определим функцию `policy_evaluation`, которая вычисляет ценности при следовании заданной стратегии.

```
>>> def policy_evaluation(env, policy, gamma, threshold):
...     """
...     Выполняет оценивание стратегии
...     @param env: имя окружающей среды OpenAI Gym
...     @param policy: матрица стратегии, содержащая вероятности действий
...                    в каждом состоянии
...     @param gamma: коэффициент обесценивания
...     @param threshold: обучение заканчивается, когда ценности всех
...                       состояний будут меньше этого значения
...     @return: ценности при следовании заданной стратегии
...     """
...     n_state = policy.shape[0]
...     V = torch.zeros(n_state)
...     while True:
...         V_temp = torch.zeros(n_state)
...         for state in range(n_state):
...             action = policy[state].item()
...             for trans_prob, new_state, reward, _ in
                                  env.env.P[state][action]:
...                 V_temp[state] += trans_prob * (reward
                                  + gamma * V[new_state])
...         max_delta = torch.max(torch.abs(V - V_temp))
...     V = V_temp.clone()
...     if max_delta <= threshold:
...         break
...     return V
```

Эта функция очень похожа на разработанную в предыдущем рецепте, только дополнительно передается параметр `policy`.

4. Теперь разработаем вторую часть алгоритма итерации по стратегиям – улучшение стратегии.

```
>>> def policy_improvement(env, V, gamma):
...     """
```

```
...     Улучшает стратегию на основе ценностей
...     @param env: имя окружающей среды OpenAI Gym
...     @param V: ценности
...     @param gamma: коэффициент обесценивания
...     @return: стратегия
...     """
...     n_state = env.observation_space.n
...     n_action = env.action_space.n
...     policy = torch.zeros(n_state)
...     for state in range(n_state):
...         v_actions = torch.zeros(n_action)
...         for action in range(n_action):
...             for trans_prob, new_state, reward, _ in
                            env.env.P[state][action]:
...                 v_actions[action] += trans_prob * (reward
                                    + gamma * V[new_state])
...         policy[state] = torch.argmax(v_actions)
...     return policy
```

Здесь мы, пользуясь уравнением оптимальности Беллмана, по заданным ценностям строим улучшенную стратегию.

5. Имея оба компонента, можно реализовать алгоритм итерации по стратегиям:

```
>>> def policy_iteration(env, gamma, threshold):
...     """
...     Имитирует заданную среду с помощью алгоритма итерации по стратегиям
...     @param env: имя окружающей среды OpenAI Gym
...     @param gamma: коэффициент обесценивания
...     @param threshold: обучение заканчивается, когда ценности всех
...                       состояний будут меньше этого значения
...     @return: оптимальные ценности и оптимальная стратегия для данной
...              окружающей среды
...     """
...     n_state = env.observation_space.n
...     n_action = env.action_space.n
...     policy = torch.randint(high=n_action, size=(n_state,)).float()
...     while True:
...         V = policy_evaluation(env, policy, gamma, threshold)
...         policy_improved = policy_improvement(env, V, gamma)
...         if torch.equal(policy_improved, policy):
...             return V, policy_improved
...         policy = policy_improved
```

6. Вызовем эту функцию, передав ей имя окружающей среды, коэффициент обесценивания и порог сходимости:

```
>>> V_optimal, optimal_policy = policy_iteration(env, gamma, threshold)
```

7. Мы нашли оптимальные ценности и оптимальную стратегию. Распечатаем их:

```
>>> print('Optimal values:\n{}'.format(V_optimal))
Optimal values:
```

```
tensor([0.5404, 0.4966, 0.4681, 0.4541, 0.5569, 0.0000, 0.3572, 0.0000, 0.5905,
        0.6421, 0.6144, 0.0000, 0.0000, 0.7410, 0.8625, 0.0000])
>>> print('Optimal policy:\n{}'.format(optimal_policy))
Optimal policy:
tensor([0., 3., 3., 3., 0., 3., 2., 3., 3., 1., 0., 3., 3., 2., 1., 3.])
```

Все то же самое, что в алгоритме итерации по ценности.

Как это работает

Алгоритм итерации по стратегиям объединяет оценивание и улучшение стратегии в одной итерации. На этапе оценивания стратегии ценности при следовании заданной стратегии (не оптимальной) вычисляются с помощью уравнения математического ожидания Беллмана до сходимости:

$$V(s) \coloneqq \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V(s')].$$

Здесь $a = \pi(s)$ – действие, предпринимаемое в состоянии s при следовании стратегии π.

На этапе улучшения стратегия обновляется на основе вычисленной ранее функции ценности $V(s)$ с применением уравнения оптимальности Беллмана

$$\pi(s) \coloneqq \text{argmax}_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V(s')].$$

Эти два шага повторяются, пока стратегия не сойдется. В этот момент последняя стратегия и соответствующая ей функция ценности являются оптимальными. На шаге 5 функция `policy_iteration` выполняет следующие действия:

- инициализирует случайную стратегию;
- вычисляет ценности, пользуясь алгоритмом оценивания стратегии;
- получает улучшенную стратегию на основе вычисленных ценностей;
- если новая стратегия отличается от старой, то обновляет стратегию и переходит к следующей итерации. В противном случае цикл завершается и возвращаются найденные ценности и стратегия.

Это еще не все

Мы только что решили задачу для окружающей среды FrozenLake с помощью алгоритма итерации по стратегиям. Возникает вопрос, что лучше: итерация по ценности или итерация по стратегиям. Есть три случая, когда имеет смысл предпочесть одно другому:

- если действий много, пользуйтесь алгоритмом итерации по стратегиям, поскольку он, скорее всего, сойдется быстрее;
- если действий мало, пользуйтесь алгоритмом итерации по ценности;
- если уже имеется хорошая стратегия (полученная интуитивно или благодаря знакомству с предметной областью), пользуйтесь алгоритмом итерации по стратегиям.

В остальных случаях алгоритмы итерации по ценности и по стратегиям примерно эквивалентны.

В следующем рецепте мы применим оба алгоритма к решению задачи о подбрасывании монеты и посмотрим, какой сойдется быстрее.

См. также

Можете применить полученные знания к замерзшему озеру большего размера, среде `FrozenLake8x8-v0` (https://gym.openai.com/envs/FrozenLake8x8-v0/).

Игра с подбрасыванием монеты

Азартная игра с подбрасыванием монеты известна каждому. В каждом раунде игрок ставит на выпадение орла. Если действительно выпал орел, игрок получает ту сумму, которую поставил, в противном случае теряет свою ставку. Игра продолжается до разорения игрока или до выигрыша определенной суммы (скажем, больше 100 долларов). Предположим, что монета несимметричная, так что орел выпадает в 40 % случаев. Сколько должен поставить игрок, чтобы максимизировать шансы на выигрыш, с учетом своего текущего капитала в каждом раунде? Интересная задачка, не правда ли?

Если орел выпадает чаще, чем в 50 % случаев, то и обсуждать нечего. Игрок просто может каждый раз ставить один доллар и в большинстве случаев останется в выигрыше. Если монета симметричная, то при ставке в один доллар игрок будет выигрывать примерно в 50 % случаев. А вот когда вероятность выпадения орла меньше 50 %, безопасной ставки не существует. И случайная стратегия тоже не подойдет. Необходимо прибегнуть к изученным методам обучения с подкреплением, чтобы ставить по-умному.

Начнем с постановки задачи о подбрасывании монеты как МППР. Это эпизодический конечный МППР без обесценивания, обладающий следующими свойствами:

- состоянием является капитал игрока в долларах. Всего имеется 101 состояние: 0, 1, 2, …, 98, 99, 100+;
- вознаграждение равно 1, если достигнуто состояние 100+; иначе вознаграждение равно 0;
- действие – это сумма, которую игрок ставит в раунде. В состоянии s допустимы действия $1, 2, \ldots, \min(s, 100 - s)$. Например, если у игрока 60 долларов, то он может поставить любую сумму от 1 до 40. Ставить больше 40 не имеет смысла, поскольку это может лишь увеличить потери, не повышая шансов на выигрыш;
- какое состояние наступает после действия, зависит от вероятности выпадения орла. Допустим, она равна 40 %. Тогда следующим состоянием после действия a в состоянии s будет $s + a$ в 40 % случаев и $s - a$ в 60 % случаев;
- процесс завершается в состояниях 0 и 100+.

Как это делается

Сначала решим задачу о подбрасывании монеты с помощью алгоритма итерации по ценности, для чего выполним следующие действия.

1. Импортируем PyTorch:

```
>>> import torch
```

2. Зададим коэффициент обесценивания и порог сходимости:

```
>>> gamma = 1
>>> threshold = 1e-10
```

Коэффициент обесценивания равен 1, потому что в этом МППР обесценивания нет, а порог выбран маленьким, поскольку мы ожидаем, что ценности действий будут невелики, т. к. вознаграждения за все действия, кроме перехода в последнее состояние, равны 0.

3. Определим параметры окружающей среды.
Всего имеется 101 состояние:

```
>>> capital_max = 100
>>> n_state = capital_max + 1
```

Вознаграждения задаются так:

```
>>> rewards = torch.zeros(n_state)
>>> rewards[-1] = 1
>>> print(rewards)
tensor([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1.])
```

Предположим, что орел выпадает в 40 % случаев:

```
>>> head_prob = 0.4
```

Поместим эти параметры в словарь:

```
>>> env = {'capital_max': capital_max,
...        'head_prob': head_prob,
...        'rewards': rewards,
...        'n_state': n_state}
```

4. Теперь напишем функцию, которая вычисляет оптимальные ценности, применяя алгоритм итерации по ценности.

```
>>> def value_iteration(env, gamma, threshold):
...     """
...     Решает задачу о подбрасывании монеты с помощью алгоритма
...     итерации по ценности
...     @param env: окружающая среда
```

```
...     @param gamma: коэффициент обесценивания
...     @param threshold: оценивание заканчивается, когда ценности всех
...                       состояний будут меньше этого значения
...     @return: ценности при следовании оптимальной стратегии для
...              данной среды
...     """
...     head_prob = env['head_prob']
...     n_state = env['n_state']
...     capital_max = env['capital_max']
...     V = torch.zeros(n_state)
...     while True:
...         V_temp = torch.zeros(n_state)
...         for state in range(1, capital_max):
...             v_actions = torch.zeros(min(state, capital_max - state) + 1)
...             for action in range(1, min(state, capital_max - state) + 1):
...                 v_actions[action] += head_prob * (
                       rewards[state + action] + gamma * V[state + action])
...                 v_actions[action] += (1 - head_prob) * (
                       rewards[state - action] + gamma * V[state - action])
...             V_temp[state] = torch.max(v_actions)
...         max_delta = torch.max(torch.abs(V - V_temp))
...         V = V_temp.clone()
...         if max_delta <= threshold:
...             break
...     return V
```

Нам нужно только вычислить ценности состояний от 1 до 99, поскольку для состояний 0 и 100+ они равны 0. В состоянии s возможны действия от 1 до $min(s, 100 - s)$. Это следует иметь в виду при решении уравнения оптимальности Беллмана.

5. Напишем функцию, которая вычисляет оптимальную стратегию по оптимальным ценностям.

```
>>> def extract_optimal_policy(env, V_optimal, gamma):
...     """
...     Строит оптимальную стратегию по оптимальным ценностям
...     @param env: окружающая среда
...     @param V_optimal: оптимальные ценности
...     @param gamma: коэффициент обесценивания
...     @return: оптимальная стратегия
...     """
...     head_prob = env['head_prob']
...     n_state = env['n_state']
...     capital_max = env['capital_max']
...     optimal_policy = torch.zeros(capital_max).int()
...     for state in range(1, capital_max):
...         v_actions = torch.zeros(n_state)
...         for action in range(1, min(state, capital_max - state) + 1):
...             v_actions[action] += head_prob * (
                    rewards[state + action] +
                    gamma * V_optimal[state + action])
```

```
...             v_actions[action] += (1 - head_prob) *
                    (rewards[state - action] +
                     gamma * V_optimal[state - action])
...         optimal_policy[state] = torch.argmax(v_actions)
...     return optimal_policy
```

6. И наконец, вызовем эту функцию, передав ей среду, коэффициент обесценивания и порог сходимости, чтобы вычислить оптимальные ценности и оптимальную стратегию. И замерим, сколько времени ушло на решение этого МППР, а потом сравним со временем работы алгоритма итерации по стратегиям.

```
>>> import time
>>> start_time = time.time()
>>> V_optimal = value_iteration(env, gamma, threshold)
>>> optimal_policy = extract_optimal_policy(env, V_optimal, gamma)
>>> print("Для решения методом итерации по ценности понадобилось
          {:.3f}c".format(time.time() - start_time))
Для решения методом итерации по ценности понадобилось 4.717 с
```

Мы решили задачу методом итерации по ценности за 4.717 секунды.

7. Посмотрим, какие получились оптимальные ценности и стратегия:

```
>>> print('Оптимальные ценности:\n{}'.format(V_optimal))
>>> print('Оптимальная стратегия:\n{}'.format(optimal_policy))
```

8. Можно построить график зависимости оптимальной ценности от состояния:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(V_optimal[:100].numpy())
>>> plt.title('Оптимальные ценности состояний')
>>> plt.xlabel('Капитал')
>>> plt.ylabel('Ценность')
>>> plt.show()
```

А теперь решим ту же задачу методом итерации по стратегиям.

9. Сначала напишем функцию `policy_evaluation`, которая вычисляет ценности при следовании заданной стратегии.

```
>>> def policy_evaluation(env, policy, gamma, threshold):
...     """
...     Оценивает стратегию
...     @param env: окружающая среда
...     @param policy: тензор стратегии, содержащий действия, предпринимаемые
...                    в каждом состоянии
...     @param gamma: коэффициент обесценивания
...     @param threshold: оценивание заканчивается, когда ценности всех
...                       состояний будут меньше этого значения
...     @return: ценности при следовании данной стратегии
...     """
...     head_prob = env['head_prob']
...     n_state = env['n_state']
```

```
...     capital_max = env['capital_max']
...     V = torch.zeros(n_state)
...     while True:
...         V_temp = torch.zeros(n_state)
...         for state in range(1, capital_max):
...             action = policy[state].item()
...             V_temp[state] += head_prob * (
                      rewards[state + action] + gamma * V[state + action])
...             V_temp[state] += (1 - head_prob) * (
                      rewards[state - action] + gamma * V[state - action])
...         max_delta = torch.max(torch.abs(V - V_temp))
...         V = V_temp.clone()
...         if max_delta <= threshold:
...             break
...     return V
```

10. Далее реализуем основную часть алгоритма итерации по стратегиям – улучшение стратегии.

```
>>> def policy_improvement(env, V, gamma):
...     """
...     Строит улучшенную стратегию на основе ценностей
...     @param env: окружающая среда
...     @param V: ценности состояний
...     @param gamma: коэффициент обесценивания
...     @return: стратегия
...     """
...     head_prob = env['head_prob']
...     n_state = env['n_state']
...     capital_max = env['capital_max']
...     policy = torch.zeros(n_state).int()
...     for state in range(1, capital_max):
...         v_actions = torch.zeros(min(state, capital_max - state) + 1)
...         for action in range(1, min(state, capital_max - state) + 1):
...             v_actions[action] += head_prob * (
                       rewards[state + action] + gamma * V[state + action])
...             v_actions[action] += (1 - head_prob) * (
                       rewards[state - action] + gamma * V[state - action])
...         policy[state] = torch.argmax(v_actions)
...     return policy
```

11. Имея оба компонента, мы можем написать главную функцию:

```
>>> def policy_iteration(env, gamma, threshold):
...     """
...     Решает задачу о подбрасывании монеты с помощью алгоритма
...     итерации по стратегиям
...     @param env: окружающая среда
...     @param gamma: коэффициент обесценивания
...     @param threshold: оценивание заканчивается, когда ценности всех
...                       состояний будут меньше этого значения
...     @return: оптимальные ценности и оптимальная стратегия для
...              данной среды
```

```
...     """
...     n_state = env['n_state']
...     policy = torch.zeros(n_state).int()
...     while True:
...         V = policy_evaluation(env, policy, gamma, threshold)
...         policy_improved = policy_improvement(env, V, gamma)
...         if torch.equal(policy_improved, policy):
...             return V, policy_improved
...         policy = policy_improved
```

12. И наконец, вызовем эту функцию, передав ей среду, коэффициент обесценивания и порог сходимости, чтобы вычислить оптимальные ценности и оптимальную стратегию. И замерим, сколько времени ушло на решение этого МППР.

```
>>> start_time = time.time()
>>> V_optimal, optimal_policy = policy_iteration(env, gamma, threshold)
>>> print("Для решения методом итерации по стратегиям понадобилось
          {:.3f}c".format(time.time() - start_time))
Для решения методом итерации по стратегиям понадобилось 2.002 с
```

13. Распечатаем полученные оптимальные ценности и стратегию:

```
>>> print('Оптимальные ценности:\n{}'.format(V_optimal))
>>> print('Оптимальная стратегия:\n{}'.format(optimal_policy))
```

Как это работает

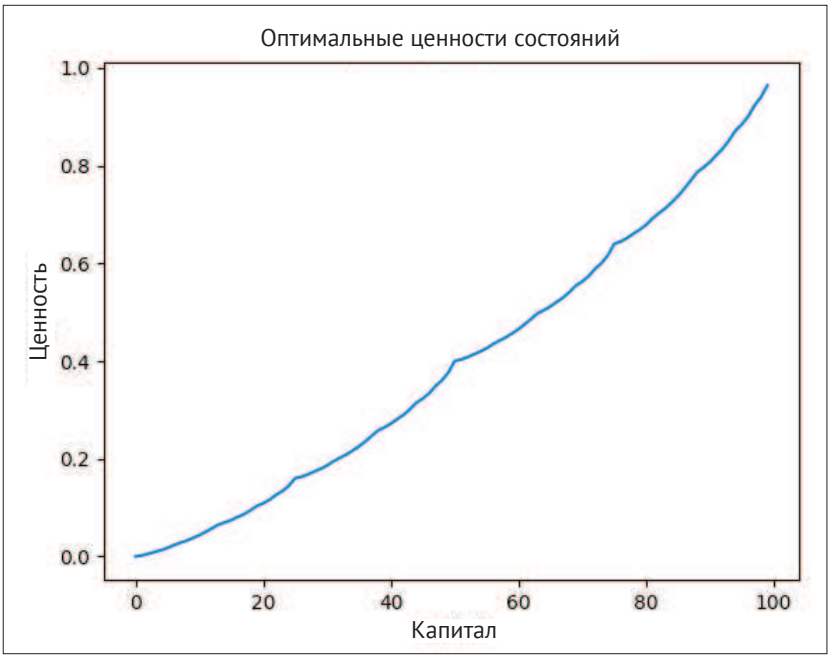
После выполнения шага 7 будут напечатаны такие оптимальные ценности:

```
Оптимальные ценности:
tensor([0.0000, 0.0021, 0.0052, 0.0092, 0.0129, 0.0174, 0.0231, 0.0278, 0.0323,
        0.0377, 0.0435, 0.0504, 0.0577, 0.0652, 0.0695, 0.0744, 0.0807, 0.0866,
        0.0942, 0.1031, 0.1087, 0.1160, 0.1259, 0.1336, 0.1441, 0.1600, 0.1631,
        0.1677, 0.1738, 0.1794, 0.1861, 0.1946, 0.2017, 0.2084, 0.2165, 0.2252,
        0.2355, 0.2465, 0.2579, 0.2643, 0.2716, 0.2810, 0.2899, 0.3013, 0.3147,
        0.3230, 0.3339, 0.3488, 0.3604, 0.3762, 0.4000, 0.4031, 0.4077, 0.4138,
        0.4194, 0.4261, 0.4346, 0.4417, 0.4484, 0.4565, 0.4652, 0.4755, 0.4865,
        0.4979, 0.5043, 0.5116, 0.5210, 0.5299, 0.5413, 0.5547, 0.5630, 0.5740,
        0.5888, 0.6004, 0.6162, 0.6400, 0.6446, 0.6516, 0.6608, 0.6690, 0.6791,
        0.6919, 0.7026, 0.7126, 0.7248, 0.7378, 0.7533, 0.7697, 0.7868, 0.7965,
        0.8075, 0.8215, 0.8349, 0.8520, 0.8721, 0.8845, 0.9009, 0.9232, 0.9406,
        0.9643, 0.0000])
```

И такая оптимальная стратегия:

```
Оптимальная стратегия:
tensor([  0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
         18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 22, 29, 30, 31, 32, 33,  9, 35,
         36, 37, 38, 11, 40,  9, 42, 43, 44,  5,  4,  3,  2,  1, 50,  1,  2, 47,
          4,  5, 44,  7,  8,  9, 10, 11, 38, 12, 36, 35, 34, 17, 32, 19, 30,  4,
          3,  2, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11,
         10,  9,  8,  7,  6,  5,  4,  3,  2, 1], dtype=torch.int32)
```

На шаге 8 будет построен такой график оптимальных ценностей состояний.



Как видим, по мере увеличения капитала (состояния) растет и оценка вознаграждения (ценность состояния), что естественно.

На шаге 9 мы проделали примерно то же, что в рецепте «Решение МППР с помощью алгоритма итерации по ценности», только окружающая среда другая.

На шаге 10 функция улучшения стратегии строит новую стратегию по переданным ценностям, применяя уравнение оптимальности Беллмана.

На шаге 12 мы видим, что с помощью метода итерации по стратегиям задачу удалось решить за 2.002 секунды – примерно в два раза меньше, чем методом итерации по ценности.

На шаге 13 печатаются оптимальные ценности:

```
Оптимальные ценности:
tensor([0.0000, 0.0021, 0.0052, 0.0092, 0.0129, 0.0174, 0.0231, 0.0278, 0.0323,
        0.0377, 0.0435, 0.0504, 0.0577, 0.0652, 0.0695, 0.0744, 0.0807, 0.0866,
        0.0942, 0.1031, 0.1087, 0.1160, 0.1259, 0.1336, 0.1441, 0.1600, 0.1631,
        0.1677, 0.1738, 0.1794, 0.1861, 0.1946, 0.2017, 0.2084, 0.2165, 0.2252,
        0.2355, 0.2465, 0.2579, 0.2643, 0.2716, 0.2810, 0.2899, 0.3013, 0.3147,
        0.3230, 0.3339, 0.3488, 0.3604, 0.3762, 0.4000, 0.4031, 0.4077, 0.4138,
        0.4194, 0.4261, 0.4346, 0.4417, 0.4484, 0.4565, 0.4652, 0.4755, 0.4865,
        0.4979, 0.5043, 0.5116, 0.5210, 0.5299, 0.5413, 0.5547, 0.5630, 0.5740,
        0.5888, 0.6004, 0.6162, 0.6400, 0.6446, 0.6516, 0.6608, 0.6690, 0.6791,
        0.6919, 0.7026, 0.7126, 0.7248, 0.7378, 0.7533, 0.7697, 0.7868, 0.7965,
        0.8075, 0.8215, 0.8349, 0.8520, 0.8721, 0.8845, 0.9009, 0.9232, 0.9406,
        0.9643, 0.0000])
```

А также оптимальная стратегия:

```
Оптимальная стратегия:
tensor([  0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
         18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 22, 29, 30, 31, 32, 33,  9, 35,
         36, 37, 38, 11, 40,  9, 42, 43, 44,  5,  4,  3,  2,  1, 50,  1,  2, 47,
          4,  5, 44,  7,  8,  9, 10, 11, 38, 12, 36, 35, 34, 17, 32, 19, 30,  4,
          3,  2, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11,
         10,  9,  8,  7,  6,  5,  4,  3,  2, 1], dtype=torch.int32)
```

Результаты, полученные обоими методами, совпадают.

Мы решили игровую задачу двумя методами: итерацией по ценности и итерацией по стратегиям. Самое сложное в задаче обучения с подкреплением – сформулировать ее в виде МППР. В нашем случае стратегия заключается в том, чтобы перейти от текущего капитала (состояния) к новому капиталу, делая ставки (действия). Оптимальная стратегия максимизирует вероятность выиграть игру (вознаграждение +1).

Интересно также отметить, как в нашем примере определяются вероятности переходов состояний для уравнения Беллмана. Выполнение действия a в состоянии s (при наличии капитала s поставить a долларов) может иметь два исхода:

- переход в состояние $s + a$, если выпадет орел. Вероятность такого перехода равна вероятности выпадения орла;
- переход в состояние $s - a$, если выпадет решка. Вероятность такого перехода равна вероятности выпадения решки.

Это очень похоже на окружающую среду FrozenLake, когда агент переходит в намеченную ячейку только с определенной вероятностью.

Мы также убедились, что в данном случае алгоритм итерации по стратегиям сходится быстрее итерации по ценности. Это связано с тем, что количество возможных действий может достигать 50 – больше, чем 4 действия в среде FrozenLake. Для МППР с большим количеством действий алгоритм итерации по стратегиям оказывается эффективнее.

Это еще не все

Интересно посмотреть, действительно ли оптимальная стратегия работает. Будем действовать, как настоящие игроки, и сыграем 10 000 эпизодов. Мы хотим сравнить оптимальную стратегию с двумя другими: консервативной (ставить 1 доллар в каждом раунде) и случайной (ставить случайную сумму).

1. Первым делом определим все три вышеупомянутые стратегии.
   Сначала – оптимальную:

   ```
   >>> def optimal_strategy(capital):
   ...     return optimal_policy[capital].item()
   ```

   Затем – консервативную:

   ```
   >>> def conservative_strategy(capital):
   ...     return 1
   ```

И наконец, случайную:

```
>>> def random_strategy(capital):
...     return torch.randint(1, capital + 1, (1,)).item()
```

2. Определим функцию, которая выполняет один эпизод и сообщает, как он закончился.

```
>>> def run_episode(head_prob, capital, policy):
...     while capital > 0:
...         bet = policy(capital)
...         if torch.rand(1).item() < head_prob:
...             capital += bet
...             if capital >= 100:
...                 return 1
...             else:
...                 capital -= bet
...     return 0
```

3. Зададим начальный капитал (50 долларов) и количество эпизодов (10 000):

```
>>> capital = 50
>>> n_episode = 10000
```

4. Выполним 10 000 эпизодов и запомним, сколько раз мы выиграли:

```
>>> n_win_random = 0
>>> n_win_conservative = 0
>>> n_win_optimal = 0
>>> for episode in range(n_episode):
...     n_win_random += run_episode(head_prob, capital, random_strategy)
...     n_win_conservative += run_episode(
                    head_prob, capital, conservative_strategy)
...     n_win_optimal += run_episode(head_prob, capital, optimal_strategy)
```

5. Напечатаем вероятности выигрыша для всех трех стратегий:

```
>>> print('Средняя вероятность выигрыша при случайной стратегии: {}'
          .format(n_win_random/n_episode))
Средняя вероятность выигрыша при случайной стратегии: 0.2251
>>> print('Средняя вероятность выигрыша при консервативной стратегии: {}'
          .format(n_win_conservative/n_episode))
Средняя вероятность выигрыша при консервативной стратегии: 0.0
>>> print('Средняя вероятность выигрыша при оптимальной стратегии: {}'
          .format(n_win_optimal/n_episode))
Средняя вероятность выигрыша при оптимальной стратегии: 0.3947
```

Нет сомнений, что наша оптимальная стратегия лучше!

Глава 3

Применение методов Монте-Карло для численного оценивания

В предыдущей главе мы занимались марковскими процессами принятия решений (МППР) и применением к ним методов **динамического программирования** (**ДП**). У основанных на модели методах, к которым относится и ДП, есть ряд недостатков. Требуется полное знание окружающей среды, т. е. матрицы переходов и матрицы вознаграждений. Кроме того, они плохо масштабируются, особенно на среды с большим количеством состояний.

В этой главе мы продолжим путешествие, рассмотрим безмодельные методы **Монте-Карло** (**МК**), которые не требуют априорных знаний об окружающей среде и гораздо лучше масштабируются, чем ДП. Для начала оценим значение числа π методом Монте-Карло. Затем поговорим о том, как использовать метод Монте-Карло для предсказания ценностей состояний и пар состояние–действие, применяя усреднение дохода, полученного при первом посещении или при всех посещениях состояния. Мы обучим агента играть в блэкджек. Будет также рассмотрено управление методом Монте-Карло с применением ε-жадной стратегии и взвешенной выборки по значимости.

В этой главе приводятся следующие рецепты:

- вычисление π методом Монте-Карло;
- оценивание стратегии методом Монте-Карло;
- предсказание методом Монте-Карло в игре блэкджек;
- управление методом Монте-Карло с единой стратегией;
- разработка управления методом Монте-Карло с ε-жадной стратегией;
- управление методом Монте-Карло с разделенной стратегией;
- разработка управления методом Монте-Карло со взвешенной выборкой по значимости.

Вычисление π методом Монте-Карло

Начнем с простого проекта: оценивание числа π методом Монте-Карло, лежащим в основе безмодельных алгоритмов обучения с подкреплением.

Методом Монте-Карло называется любой метод, в котором для решения задачи используется случайность. Алгоритм много раз производит **случайную выборку**, смотрит, какая ее часть обладает определенными свойствами, а затем выполняет численное оценивание.

Применим метод Монте-Карло, чтобы вычислить приближенное значение числа π. Поместим много случайно выбранных точек в квадрат со стороной 2 ($-1 < x < 1$, $-1 < y < 1$) и посчитаем, сколько из них попало в круг единичного радиуса. Мы знаем, что площадь квадрата равна

$$C = 2^2 = 4,$$

а площадь круга равна

$$S = \pi * 1^2 = \pi.$$

Разделив площадь круга на площадь квадрата, получим:

$$S/C = \pi/4.$$

S/C можно оценить как долю точек, попавших внутри круга. Соответственно, оценка π будет в четыре раза больше.

Как это делается

Воспользуемся методом МК для получения оценки числа π.

1. Импортируем необходимые модули: PyTorch, `math` (там определено значение π) и `matplotlib` (для размещения случайных точек внутри квадрата).

```
>>> import torch
>>> import math
>>> import matplotlib.pyplot as plt
```

2. Случайно сгенерируем 1000 точек внутри квадрата, для которых $-1 < x < 1$ и $-1 < y < 1$:

```
>>> n_point = 1000
>>> points = torch.rand((n_point, 2)) * 2 - 1
```

3. Инициализируем счетчик точек, попавших внутрь единичного круга, и список для их хранения:

```
>>> n_point_circle = 0
>>> points_circle = []
```

4. Для каждой случайной точки вычислим расстояние до начала координат. Точка попадает внутрь (или на границу) круга, если это расстояние меньше или равно 1:

```
>>> for point in points:
...     r = torch.sqrt(point[0] ** 2 + point[1] ** 2)
...     if r <= 1:
...         points_circle.append(point)
...         n_point_circle += 1
```

5. Преобразуем список в тензор:

```
>>> points_circle = torch.stack(points_circle)
```

6. Нанесем все случайные точки на график, изобразив те из них, что оказались внутри круга, другим цветом:

```
>>> plt.plot(points[:, 0].numpy(), points[:, 1].numpy(), 'y.')
>>> plt.plot(points_circle[:, 0].numpy(), points_circle[:,1].numpy(), 'c.')
```

7. Нарисуем, что получилось:

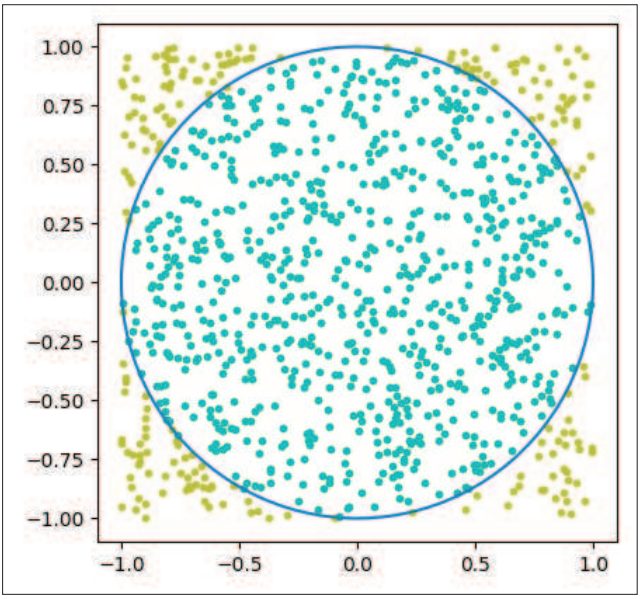
```
>>> i = torch.linspace(0, 2 * math.pi)
>>> plt.plot(torch.cos(i).numpy(), torch.sin(i).numpy())
>>> plt.axes().set_aspect('equal')
>>> plt.show()
```

8. И наконец, вычислим значение π:

```
>>> pi_estimated = 4 * (n_point_circle / n_point)
>>> print('Оценка значения pi:', pi_estimated)
```

Как это работает

На шаге 5 будет нарисована такая картинка – точки случайно разбросаны по всему квадрату, часть из них попала в круг.



Метод Монте-Карло работает в силу **закона больших чисел**, согласно которому среднее в большом количестве испытаний сходится к математическому ожиданию. В нашем случае величина `4 * (n_point_circle / n_point)` сходится к истинному значению π, когда количество сгенерированных случайных точек стремится к бесконечности.

На шаге 8 печатается вычисленная оценка π:

```
Оценка значения pi: 3.156
```

Приближенное значение π, вычисленное методом Монте-Карло, довольно близко к истинному (3.14159...).

Это еще не все

Мы можем улучшить оценку, увеличив количество итераций, например до 10 000. На каждой итерации будем случайно генерировать точку в квадрате и смотреть, попала ли она внутрь круга; оценивать π будем по ходу дела, вычисляя текущую долю точек в круге.

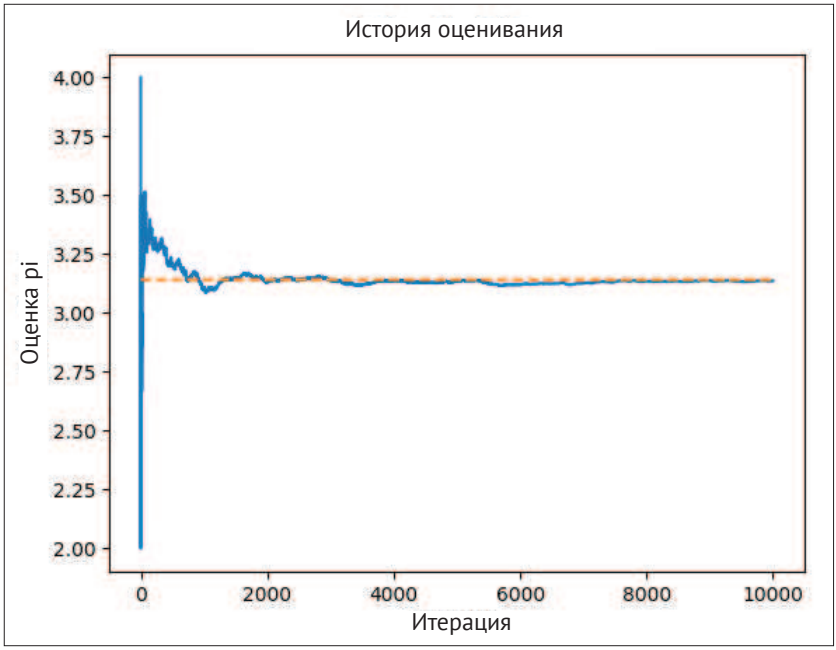
Затем изобразим на графике всю историю оценивания и истинное значение π. Все это реализовано в следующей функции.

```
>>> def estimate_pi_mc(n_iteration):
...     n_point_circle = 0
...     pi_iteration = []
...     for i in range(1, n_iteration+1):
...         point = torch.rand(2) * 2 - 1
...         r = torch.sqrt(point[0] ** 2 + point[1] ** 2)
...         if r <= 1:
...             n_point_circle += 1
...         pi_iteration.append(4 * (n_point_circle / i))
...     plt.plot(pi_iteration)
...     plt.plot([math.pi] * n_iteration, '--')
...     plt.xlabel('Итерация')
...     plt.ylabel('Оценка pi')
...     plt.title('История оценивания')
...     plt.show()
...     print('Оценка значения pi:', pi_iteration[-1])
```

Вызовем эту функцию, задав 10 000 итераций:

```
>>> estimate_pi_mc(10000)
Оценка значения pi: 3.1364
```

Результат показан на рисунке ниже.



Видно, что чем больше итераций, тем ближе оценка π к истинному значению. Конечно, имеют место флуктуации. Но с ростом числа итераций они сглаживаются.

См. также

О других интересных применениях метода Монте-Карло можно прочитать в следующих статьях:

- Playing games such as Go, Havannah, and Battleship, with MC tree search, which searches for the best move (https://en.wikipedia.org/wiki/Monte_Carlo_tree_search);
- Assessing investments and portfolio (https://en.wikipedia.org/wiki/Monte_Carlo_methods_in_finance);
- Studying biological systems with MC simulations (https://en.wikipedia.org/wiki/Bayesian_inference_in_phylogeny).

Оценивание стратегии методом Монте-Карло

В главе 2 мы с помощью ДП оценивали стратегию, т. е. вычисляли для нее функцию ценности состояний. Этот метод неплохо работает, но имеет некоторые ограничения. И главное – для него необходимо иметь полную информацию об окружающей среде, в т. ч. знать матрицу переходов и матрицу вознаграждений. Но на практике матрица переходов заранее неизвестна. Алгоритмы обучения с подкреплением, нуждающиеся в известном МППР, называются **основанными на** модели. А алгоритмы, которым не нужна априорная информация о переходах и вознаграждениях, называются **безмодельными**. Метод Монте-Карло относится к числу безмодельных.

В этом рецепте мы вычислим функцию ценности с помощью метода Монте-Карло. В качестве примера снова возьмем окружающую среду FrozenLake, но будем предполагать, что к матрицам переходов и вознаграждений нет доступа. Напомним, что **доходом** процесса называется полное вознаграждение, полученное в долгосрочной перспективе:

$$G_t = \sum_k \gamma^k R_{t+k+1}.$$

В стратегии оценивания методом МК в качестве оценки функции ценности вместо **ожидаемого дохода** (как в ДП) используется **эмпирический средний доход**. Существует два способа такого оценивания: **метод МК первого посещения**, когда усредняются только доходы, полученные при **первом посещении** состояния s в эпизоде, и **всех посещений**, когда усредняются доходы, полученные после всех посещений s. Очевидно, что метод первого посещения требует гораздо меньше вычислений, поэтому применяется чаще.

Как это делается

Для поиска оптимальной стратегии в среде FrozenLake применим метод МК первого посещения.

1. Импортируем библиотеки PyTorch и Gym и создадим экземпляр окружающей среды FrozenLake:

```
>>> import torch
>>> import gym
>>> env = gym.make("FrozenLake-v0")
```

2. Для оценивания стратегии методом Монте-Карло сначала нужно определить функцию, которая выполняет один эпизод взаимодействия со средой FrozenLake, следуя заданной стратегии, и возвращает вознаграждение и состояние на каждом шаге.

```
>>> def run_episode(env, policy):
...     state = env.reset()
...     rewards = []
...     states = [state]
...     is_done = False
...     while not is_done:
```

```
...         action = policy[state].item()
...         state, reward, is_done, info = env.step(action)
...         states.append(state)
...         rewards.append(reward)
...         if is_done:
...             break
...     states = torch.tensor(states)
...     rewards = torch.tensor(rewards)
...     return states, rewards
```

Подчеркнем, что в методах Монте-Карло нужно хранить все состояния и вознаграждения в них, потому что у нас нет доступа к информации о среде: вероятностям переходов и матрице вознаграждений.

3. Далее определим функцию, которая оценивает заданную стратегию методом МК первого посещения.

```
>>> def mc_prediction_first_visit(env, policy, gamma, n_episode):
...     n_state = policy.shape[0]
...     V = torch.zeros(n_state)
...     N = torch.zeros(n_state)
...     for episode in range(n_episode):
...         states_t, rewards_t = run_episode(env, policy)
...         return_t = 0
...         first_visit = torch.zeros(n_state)
...         G = torch.zeros(n_state)
...         for state_t, reward_t in zip(reversed(states_t)[1:],
                                         reversed(rewards_t)):
...             return_t = gamma * return_t + reward_t
...             G[state_t] = return_t
...             first_visit[state_t] = 1
...         for state in range(n_state):
...             if first_visit[state] > 0:
...                 V[state] += G[state]
...                 N[state] += 1
...     for state in range(n_state):
...         if N[state] > 0:
...             V[state] = V[state] / N[state]
...     return V
```

4. Зададим коэффициент обесценивания 1, чтобы упростить вычисления, и имитируем 10 000 эпизодов:

```
>>> gamma = 1
>>> n_episode = 10000
```

5. Возьмем оптимальную стратегию, вычисленную в предыдущей главе, и подадим ее на вход функции МК первого посещения наряду с другими параметрами:

```
>>> optimal_policy = torch.tensor([0., 3., 3., 3., 0., 3., 2., 3.,
3., 1., 0., 3., 3., 2., 1., 3.])
>>> value = mc_prediction_first_visit(env, optimal_policy, gamma,
n_episode)
```

```
>>> print('Функция ценности, вычисленная методом МК первого посещения:\n',
          value)
Функция ценности, вычисленная методом МК первого посещения:
tensor([0.7463, 0.5004, 0.4938, 0.4602, 0.7463, 0.0000, 0.3914,
0.0000, 0.7463, 0.7469, 0.6797, 0.0000, 0.0000, 0.8038, 0.8911,
0.0000])
```

Мы только что вычислили функцию ценности оптимальной стратегии методом МК первого посещения.

Как это работает

На шаге 3 выполняются следующие действия:

- прогоняется `n_episode` эпизодов;
- для каждого эпизода вычисляются доходы, полученные при первом посещении каждого состояния;
- для каждого состояния вычисляется среднее доходов, полученных при его первом посещении, по всем эпизодам.

Как видим, если предсказание основано на МК, то необязательно знать полную модель среды. На самом деле в большинстве практических ситуаций матрицы переходов и вознаграждений заранее неизвестны или их очень трудно получить. Представьте только, сколько состояний возможно при игре в шахматы или го, а также количество возможных действий – вычислить матрицы переходов и вознаграждений едва ли возможно. Идея безмодельного обучения с подкреплением в том и заключается, чтобы получать опыт во взаимодействии с окружающей средой.

В данном случае мы имели дело только с тем, что поддается наблюдению, т. е. с новыми состояниями и полученными для них вознаграждениями на каждом шаге, после чего делали предсказание методом Монте-Карло. Заметим, что чем больше эпизодов имитировано, тем точнее предсказания. Построив график функции ценности после каждого эпизода, мы увидим, как он сходится со временем, – так же, как в задаче оценивания числа π.

Это еще не все

Давайте также оценим функцию ценности оптимальной стратегии методом МК всех посещений.

1. Определим функцию, которая оценивает заданную стратегию методом МК всех посещений:

```
>>> def mc_prediction_every_visit(env, policy, gamma, n_episode):
...     n_state = policy.shape[0]
...     V = torch.zeros(n_state)
...     N = torch.zeros(n_state)
...     G = torch.zeros(n_state)
...     for episode in range(n_episode):
...         states_t, rewards_t = run_episode(env, policy)
...         return_t = 0
```

```
...         for state_t, reward_t in zip(reversed(states_t)[1:],
                                         reversed(rewards_t)):
...             return_t = gamma * return_t + reward_t
...             G[state_t] += return_t
...             N[state_t] += 1
...     for state in range(n_state):
...         if N[state] > 0:
...             V[state] = G[state] / N[state]
...     return V
```

Как и в методе МК первого посещения, эта функция выполняет следующие действия:

- прогоняется `n_episode` эпизодов;
- для каждого эпизода вычисляются доходы, полученные при каждом посещении каждого состояния;
- для каждого состояния вычисляется среднее всех полученных в нем доходов по всем эпизодам.

2. Вычисляем функцию ценности, вызвав только что написанную функцию, которой передается стратегия и другие параметры:

```
>>> value = mc_prediction_every_visit(env, optimal_policy, gamma,
n_episode)
```

3. Выводим результат:

```
>>> print('Функция ценности, вычисленная методом МК всех посещений:\n', value)
'Функция ценности, вычисленная методом МК всех посещений:
tensor([0.6221, 0.4322, 0.3903, 0.3578, 0.6246, 0.0000, 0.3520,
0.0000, 0.6428, 0.6759, 0.6323, 0.0000, 0.0000, 0.7624, 0.8801,
0.0000])
```

Предсказание методом Монте-Карло в игре блэкджек

В этом рецепте мы будем играть в карточную игру блэкджек (или «двадцать одно») и оценим стратегию, которая, на наш взгляд, может работать хорошо. Мы ближе познакомимся с предсказанием методом Монте-Карло и подготовим почву для нахождения оптимальной стратегии путем применения управления Монте-Карло в следующих рецептах.

В блэкджеке цель состоит в том, чтобы набрать как можно больше очков, но не более 21. Валет, дама и король стоят по 10 очков, а карты от 2 до 10 оцениваются по номинальному достоинству. Туз может стоить 1 или 11 очков, в последнем случае туз называется **играющим**. Игрок играет против сдающего. Вначале обоим участникам сдается по две карты, и одна из карт сдающего открывается. Игрок может попросить дополнительные карты (сказав **еще**) или остановиться (сказав **хватит**). Когда игрок останавливается, сдающий берет из колоды карты до тех пор, пока их сумма не окажется больше или равна 17. Если сумма карт у игрока превысит 21 (**перебор**), игрок проигрывает. В противном случае, если сумма карт у сдающего превысит 21, игрок выигрывает.

Если ни один участник не перебрал, то выигрывает тот, кто набрал больше очков, а если очков поровну, то игра завершается вничью. В Gym окружающая среда Blackjack описывается следующим образом:

- каждый эпизод представляет собой МППР, в начале которого оба участника получают две карты, и одна карта сдающего открыта;
- эпизод заканчивается, когда какой-то участник выигрывает или игра завершается вничью. В конце эпизода начисляется вознаграждение +1, если выиграл игрок, –1 – если игрок проиграл, и 0 – в случае ничьей;
- в каждом раунде у игрока есть два действия: еще (1) – получить еще карту и хватит (0) – больше не брать карт.

Сначала попробуем простую стратегию – брать карты, пока сумма очков меньше 18 (или 19, или 20 – как хотите).

Как это делается

Приступим к имитации окружающей среды Blackjack, попутно объяснив ее состояния и действия.

1. Импортируем библиотеки PyTorch и Gym и создадим экземпляр окружающей среды Blackjack:

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

2. Переведем среду в исходное состояние:

```
>>> env.reset()
(20, 5, False)
```

   Возвращаются три переменные, определяющие состояние:
   - количество очков у игрока (в этом примере 20);
   - количество очков у сдающего (в этом примере 5);
   - признак наличия играющего туза у игрока (в этом примере `False`).

   Туз называется играющим, если его можно посчитать как 11 очков без перебора. Если у игрока нет туза или есть, но его зачет привел бы к перебору, то третья переменная состояния равна `False`.
   Рассмотрим следующий эпизод:

```
>>> env.reset()
(18, 6, True)
```

   18 очков и `True` означают, что у игрока на руках играющий туз и семерка, причем туз посчитан как 11 очков.

3. На примере некоторых действий покажем, как работает среда Blackjack. Сначала попросим еще карту, поскольку наличие играющего туза дает нам определенную гибкость:

```
>>> env.step(1)
((20, 6, True), 0, False, {})
```

Возвращаются три переменные состояния (20, 6, `True`), вознаграждение (пока 0) и признак завершения эпизода (пока `False`).
После этого мы перестаем брать карты:

```
>>> env.step(0)
((20, 6, True), 1, True, {})
```

В этом эпизоде мы выиграли, поскольку вознаграждение равно 1 и эпизод завершился. Но после того как игрок сказал «хватит», к действиям приступает сдающий.

4. Бывает, что мы проигрываем, например:

```
>>> env.reset()
(15, 10, False)
>>> env.step(1)
((25, 10, False), -1, True, {})
```

Перейдем теперь к предсказанию ценности для простой стратегии, когда мы перестаем брать карты, набрав 18 очков.

5. Как всегда, сначала нужно написать функцию, которая имитирует эпизод Blackjack при следовании простой стратегии.

```
>>> def run_episode(env, hold_score):
...     state = env.reset()
...     rewards = []
...     states = [state]
...     is_done = False
...     while not is_done:
...         action = 1 if state[0] < hold_score else 0
...         state, reward, is_done, info = env.step(action)
...         states.append(state)
...         rewards.append(reward)
...         if is_done:
...             break
...     return states, rewards
```

6. Затем определим функцию, которая оценивает простую стратегию игры в блэкджек методом МК первого посещения:

```
>>> from collections import defaultdict
>>> def mc_prediction_first_visit(env, hold_score, gamma, n_episode):
...     V = defaultdict(float)
...     N = defaultdict(int)
...     for episode in range(n_episode):
...         states_t, rewards_t = run_episode(env, hold_score)
...         return_t = 0
...         G = {}
...         for state_t, reward_t in zip(states_t[1::-1], rewards_t[::-1]):
...             return_t = gamma * return_t + reward_t
...             G[state_t] = return_t
...         for state, return_t in G.items():
...             if state[0] <= 21:
...                 V[state] += return_t
...                 N[state] += 1
```

```
...     for state in V:
...         V[state] = V[state] / N[state]
...     return V
```

7. Зададим параметры: порог остановки (`hold_score`) 18, коэффициент обесценивания 1, количество эпизодов 500 000:

```
>>> hold_score = 18
>>> gamma = 1
>>> n_episode = 500000
```

8. Выполним предсказание МК с такими параметрами:

```
>>> value = mc_prediction_first_visit(env, hold_score, gamma, n_episode)
```

Распечатаем получившуюся функцию ценности:

```
>>> print('Функция ценности, вычисленная методом МК первого посещения:\n',
value)
```

В результате будут напечатаны ценности всех возможных состояний. И напоследок напечатаем количество состояний:

```
>>> print('Количество состояний:', len(value))
Количество состояний: 280
```

Как видим, всего их 280.

Как это работает

На шаге 4 мы набрали больше 21 очка и, стало быть, проиграли. Повторим, что в среде Blackjack состояние – кортеж из трех элементов. Первый элемент – количество очков, набранных игроком, второй – взятая из колоды карта достоинством от 1 до 10, третий – признак наличия играющего туза.

На шаге 5 агент говорит «еще» или «хватит», исходя из текущего количества очков. Если оно меньше `hold_score`, он берет еще карту, иначе останавливается. Как положено в методе Монте-Карло, мы запоминаем состояния и вознаграждения на всех шагах.

После выполнения шага 8 мы получим такой результат:

```
Функция ценности, вычисленная методом МК первого посещения:
defaultdict(<class 'float'>, {(20, 6, False): 0.6923485653560042, (17, 5,
False): -0.24390243902439024, (16, 5, False): -0.19118165784832453, (20,
10, False): 0.4326379146490474, (20, 7, False): 0.7686220540168588, (16, 6,
False): -0.19249478804725503,
.....
.....
(5, 9, False): -0.20612244897959184, (12, 7, True): 0.058823529411764705,
(6, 4, False): -0.26582278481012656, (4, 8, False): -0.14937759336099585,
(4, 3, False): -0.1680327868852459, (4, 9, False): -0.20276497695852536,
(4, 4, False): -0.3201754385964912, (12, 8, True): 0.11057692307692307})
```

Мы показали, насколько эффективно можно вычислить функцию ценности 280 состояний в среде Blackjack с помощью предсказания методом МК. Функция предсказания, написанная на шаге 6, выполняет следующие действия:

- прогоняет `n_episode` эпизодов, следуя простой стратегии;
- для каждого эпизода вычисляет доходы при первом посещении каждого состояния;
- для каждого состояния вычисляет ценность, усредняя доходы при первом посещении по всем эпизодам.

Отметим, что мы игнорируем состояния, в которых игрок набрал больше 21 очка, т. к. знаем, что вознаграждение в них равно –1.

Модель окружающей среды Blackjack (ее матрицы переходов и вознаграждений) заранее неизвестна. Да и получить вероятности переходов между двумя состояниями было бы затруднительно. Размер матрицы переходов был бы равен 280 * 280 * 2, так что объем вычислений оказался бы очень велик. В решении методом МК нам нужно было лишь имитировать достаточно много эпизодов, для каждого из них вычислить доход и соответственно обновить функцию ценности.

В следующий раз, когда будете играть в блэкджек, придерживаясь простой стратегии (остановиться, набрав заранее заданное количество очков), посмотрите, помогут ли предсказанные ценности состояний делать ставки.

Это еще не все

Поскольку в этом случае состояний много, воспринимать их ценности трудно. Чтобы наглядно представить функцию ценности, можно построить трехмерный график. Состояние в данном случае трехмерное, причем третья координата может принимать все два значения (есть играющий туз или нет). Поэтому можно разделить график на две части: одна – для состояний с играющим тузом, другая – для остальных. В каждом случае по оси *x* откладывается количество очков, набранное игроком, по оси *y* – открытая карта сдающего, а по оси *z* – ценность.

Для построения графиков выполним следующие действия.

1. Импортируем необходимые модули из библиотеки matplotlib:

```
>>> import matplotlib
>>> import matplotlib.pyplot as plt
>>> from mpl_toolkits.mplot3d import Axes3D
```

2. Напишем вспомогательную функцию для построения трехмерного графика:

```
>>> def plot_surface(X, Y, Z, title):
...     fig = plt.figure(figsize=(20, 10))
...     ax = fig.add_subplot(111, projection='3d')
...     surf = ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
...     cmap=matplotlib.cm.coolwarm, vmin=-1.0, vmax=1.0)
...     ax.set_xlabel('Очки игрока')
...     ax.set_ylabel('Открытая карта сдающего')
...     ax.set_zlabel('Ценность')
...     ax.set_title(title)
...     ax.view_init(ax.elev, -120)
...     fig.colorbar(surf)
...     plt.show()
```

3. Затем напишем функцию, которая строит массивы, изображаемые по каждому из трех измерений, и вызовем `plot_surface` для визуализации функции ценности с играющим тузом и без него:

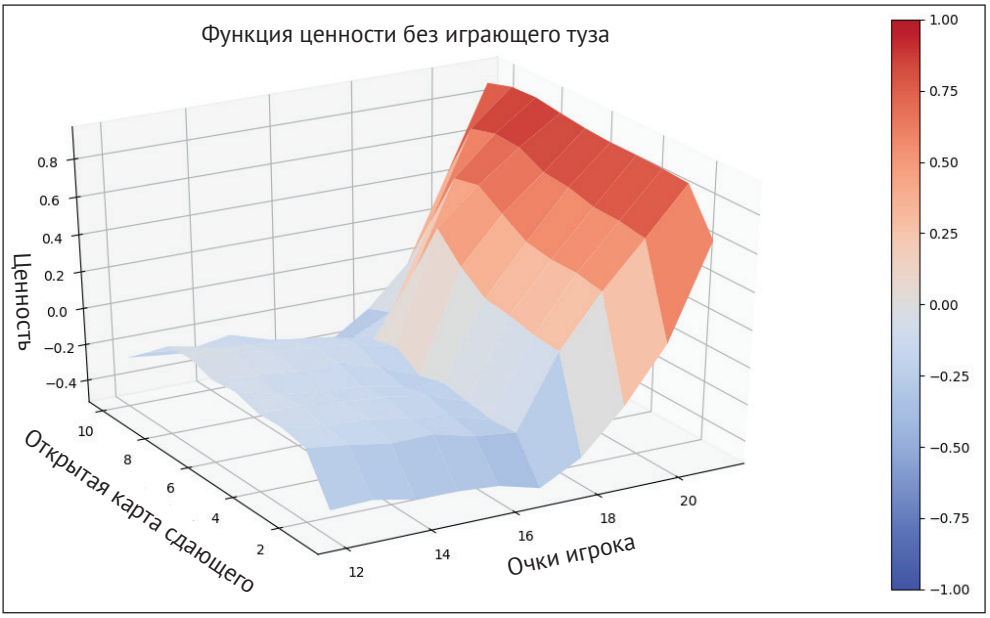
```
>>> def plot_blackjack_value(V):
...     player_sum_range = range(12, 22)
...     dealer_show_range = range(1, 11)
...     X, Y = torch.meshgrid([torch.tensor(player_sum_range),
                               torch.tensor(dealer_show_range)])
...     values_to_plot = torch.zeros((len(player_sum_range),
                                      len(dealer_show_range), 2))
...     for i, player in enumerate(player_sum_range):
...         for j, dealer in enumerate(dealer_show_range):
...             for k, ace in enumerate([False, True]):
...                 values_to_plot[i, j, k] = V[(player, dealer, ace)]
...     plot_surface(X, Y, values_to_plot[:,:,0].numpy(),
                     "Функция ценности без играющего туза")
...     plot_surface(X, Y, values_to_plot[:,:,1].numpy(),
                     "Функция ценности с играющим тузом")
```

   Нас интересуют только состояния, в которых игрок набрал больше 11 очков, так что в тензоре `values_to_plot` хранятся только такие ценности.

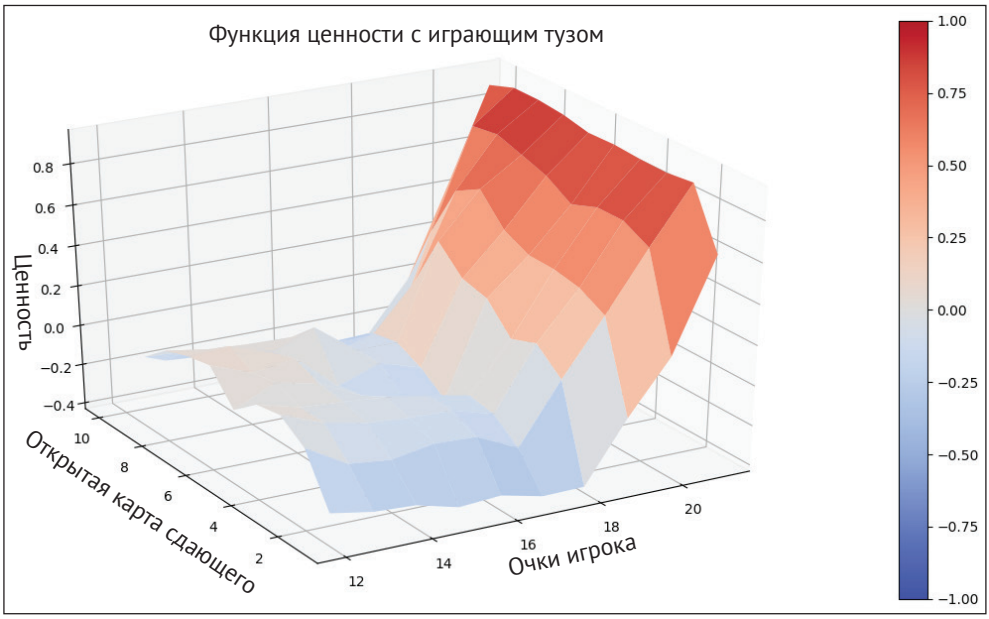
4. И наконец, вызываем функцию `plot_blackjack_value`:

```
>>> plot_blackjack_value(value)
```

Ниже показан график функции ценности состояний без играющего туза.



А вот как выглядит график функции ценности состояний с играющим тузом.



Попробуйте поэкспериментировать со значением параметра `hold_score` и посмотрите, как при этом меняется функция ценности.

См. также

Если вы ранее не сталкивались с окружающей средой Blackjack, можете почерпнуть дополнительные сведения о ней из исходного кода по адресу https://github.com/openai/gym/blob/master/gym/envs/toy_text/blackjack.py.

Иногда проще прочитать сам код, чем его описание на естественном языке.

Управление методом Монте-Карло с единой стратегией

В предыдущем рецепте мы предсказали ценность стратегии, при которой агент останавливается, набрав 18 очков. Это простая стратегия, которой легко следовать, но, очевидно, не оптимальная. В этом рецепте мы будем искать оптимальную стратегию игры в блэкджек с помощью управления методом Монте-Карло.

Предсказание методом Монте-Карло используется, чтобы оценить ценность при следовании заданной стратегии, а **управление методом Монте-Карло**

(**управление МК**) – чтобы найти оптимальную стратегию. Есть две основные категории управления МК: с единой стратегией и с разделенной стратегией. Методы с **единой стратегией** (on-policy) обучаются оптимальной стратегии, выполняя эту стратегию, а затем оценивая и улучшая ее, тогда как методы с **разделенной стратегией** (off-policy) обучаются оптимальной стратегии на данных, сгенерированных другой стратегией. Принцип работы управления МК с единой стратегией очень похож на алгоритм итерации по стратегиям в динамическом программировании, поскольку состоит из двух этапов: оценивания и улучшения.

- На этапе оценивания вычисляется не функция ценности (она же функция **ценности состояний**, или **полезности**), а функция **ценности действий**. Эта функция чаще называется **Q-функцией** и определяет ценность пары состояние–действие (s, a), когда при следовании заданной стратегии в состоянии s предпринимается действие a. Как и раньше, оценивание можно производить для первого посещения или для всех посещений.
- На этапе улучшения стратегия обновляется путем сопоставления каждому состоянию оптимального действия:

  $\pi(s) = \text{argmax}_a Q(s, a)$.

Оптимальная стратегия получается, если чередовать эти два этапа на протяжении большого числа итераций.

Как это делается

Давайте найдем оптимальную стратегию игры в блэкджек с помощью управления МК с единой стратегией.

1. Импортируем необходимые модули из библиотеки matplotlib:

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

2. Напишем функцию, которая выполняет один эпизод, предпринимая действия, продиктованные Q-функцией. Это этап улучшения.

```
>>> def run_episode(env, Q, n_action):
...     """
...     Выполняет эпизод, руководствуясь заданной Q-функцией
...     @param env: имя окружающей среды OpenAI Gym
...     @param Q: Q-функция
...     @param n_action: пространство действий
...     @return: результирующие состояния, действия и вознаграждения для
...              всего эпизода
...     """
...     state = env.reset()
...     rewards = []
...     actions = []
...     states = []
```

```
...     is_done = False
...     action = torch.randint(0, n_action, [1]).item()
...     while not is_done:
...         actions.append(action)
...         states.append(state)
...         state, reward, is_done, info = env.step(action)
...         rewards.append(reward)
...         if is_done:
...             break
...         action = torch.argmax(Q[state]).item()
...     return states, actions, rewards
```

3. Теперь реализуем алгоритм управления МК с единой стратегией.

```
>>> from collections import defaultdict
>>> def mc_control_on_policy(env, gamma, n_episode):
...     """
...     Находит оптимальную стратегию методом управления МК с единой
...     стратегией
...     @param env: имя окружающей среды OpenAI Gym
...     @param gamma: коэффициент обесценивания
...     @param n_episode: количество эпизодов
...     @return: оптимальная Q-функция и оптимальная стратегия
...     """
...     n_action = env.action_space.n
...     G_sum = defaultdict(float)
...     N = defaultdict(int)
...     Q = defaultdict(lambda: torch.empty(env.action_space.n))
...     for episode in range(n_episode):
...         states_t, actions_t, rewards_t = run_episode(env, Q, n_action)
...         return_t = 0
...         G = {}
...         for state_t, action_t, reward_t in zip(states_t[::-1],
                    actions_t[::-1], rewards_t[::-1]):
...             return_t = gamma * return_t + reward_t
...             G[(state_t, action_t)] = return_t
...         for state_action, return_t in G.items():
...             state, action = state_action
...             if state[0] <= 21:
...                 G_sum[state_action] += return_t
...                 N[state_action] += 1
...                 Q[state][action] = G_sum[state_action] / N[state_action]
...     policy = {}
...     for state, actions in Q.items():
...         policy[state] = torch.argmax(actions).item()
...     return Q, policy
```

4. Зададим коэффициент обесценивания 1 и количество эпизодов 500 000:

```
>>> gamma = 1
>>> n_episode = 500000
```

5. Выполним алгоритм управления МК с единой стратегией, чтобы найти оптимальные Q-функцию и стратегию:

```
>>> optimal_Q, optimal_policy = mc_control_on_policy(env, gamma, n_episode)
>>> print(optimal_policy)
```

6. Вычислим функцию ценности для оптимальной стратегии и распечатаем ее:

```
>>> optimal_value = defaultdict(float)
>>> for state, action_values in optimal_Q.items():
...     optimal_value[state] = torch.max(action_values).item()
>>> print(optimal_value)
```

7. Визуализируем функцию ценности, вызвав функции `plot_blackjack_value` и `plot_surface`, разработанные в предыдущем рецепте:

```
>>> plot_blackjack_value(optimal_value)
```

Как это работает

В этом рецепте мы применили к игре в блэкджек управление МК с единой стратегией, исследуя различные начальные действия. Цель – оптимизация стратегии – достигается путем чередования этапов оценивания и улучшения в каждом имитированном эпизоде.

На шаге 2 мы выполняем эпизод и выбираем действия, диктуемые Q-функцией:

- инициализируем эпизод;
- предпринимаем случайное действие в качестве исследовательского старта;
- последующие действия выбираем на основе текущей Q-функции, т. е. $a' = \text{argmax}_a Q(s, a)$;
- запоминаем состояния, действия и вознаграждения на всех шагах эпизода для использования на этапе оценивания.

Важно отметить, что первое действие выбирается случайным образом, потому что алгоритм управления МК сходится к оптимальному решению только при таком условии. Выполнение эпизода в алгоритме МК, начиная со случайного действия, называется **исследовательским стартом**.

Первое действие выбирается случайно, для того чтобы стратегия сошлась к оптимальному решению. В противном случае некоторые состояния никогда не были бы посещены, ценности соответствующих пар состояние–действие не были бы оптимизированы, и в итоге стратегия получилась бы неоптимальной.

Шаг 2 представляет собой этап улучшения, а шаг 3 – управление МК, на нем выполняются следующие действия:

- Q-функция инициализируется случайными небольшими значениями;
- выполняется `n_episode` эпизодов;
- для каждого эпизода выполняется улучшение стратегии и вычисляются состояния, действия и вознаграждения. Затем производится оценивание стратегии с помощью предсказания методом МК первого посещения на основе только что полученных состояний, действий и вознаграждений; при этом обновляется Q-функция;
- в итоге мы получаем оптимальную Q-функцию и оптимальную стратегию, заключающуюся в том, чтобы выбирать в каждом состоянии наилучшее действие, диктуемое оптимальной Q-функцией.

На каждой итерации стратегия выбирает действие жадно, т. е. сообразуясь с текущей функцией ценности действий Q ($\pi(s) = \text{argmax}_a Q(s, a)$). В результате удается получить оптимальную стратегию, хотя начинали мы со случайной.

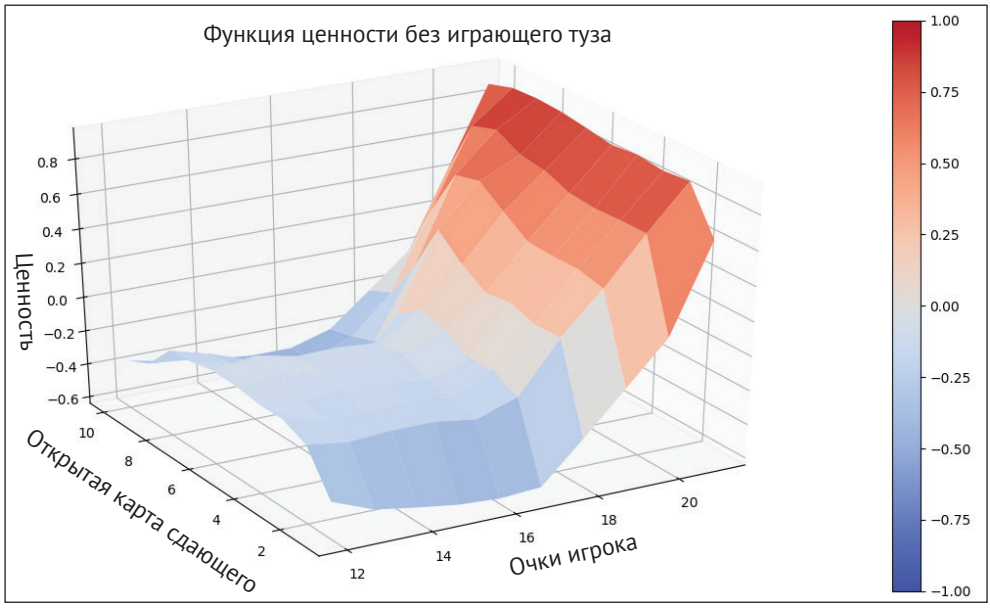
На шаге 5 распечатывается следующая оптимальная стратегия:

```
{(16, 8, True): 1, (11, 2, False): 1, (15, 5, True): 1, (14, 9, False): 1,
(11, 6, False): 1, (20, 3, False): 0, (9, 6, False): 0, (12, 9, False): 0,
(21, 2, True): 0, (16, 10, False): 1, (17, 5, False): 0, (13, 10, False):
1, (12, 10, False): 1, (14, 10, False): 0, (10, 2, False): 1, (20, 4,
False): 0, (11, 4, False): 1, (16, 9, False): 0, (10, 8,
  ......
  ......
1, (18, 6, True): 0, (12, 2, True): 1, (8, 3, False): 1, (13, 3, True): 0,
(4, 7, False): 1, (18, 8, True): 0, (6, 5, False): 1, (17, 6, True): 0,
(19, 9, True): 0, (4, 4, False): 0, (14, 5, True): 1, (12, 6, True): 0, (4,
9, False): 1, (13, 4, True): 1, (4, 8, False): 1, (14, 3, True): 1, (12, 4,
True): 1, (4, 6, False): 0, (12, 5, True): 0, (4, 2, False): 1, (4, 3,
False): 1, (5, 4, False): 1, (4, 1, False): 0}
```

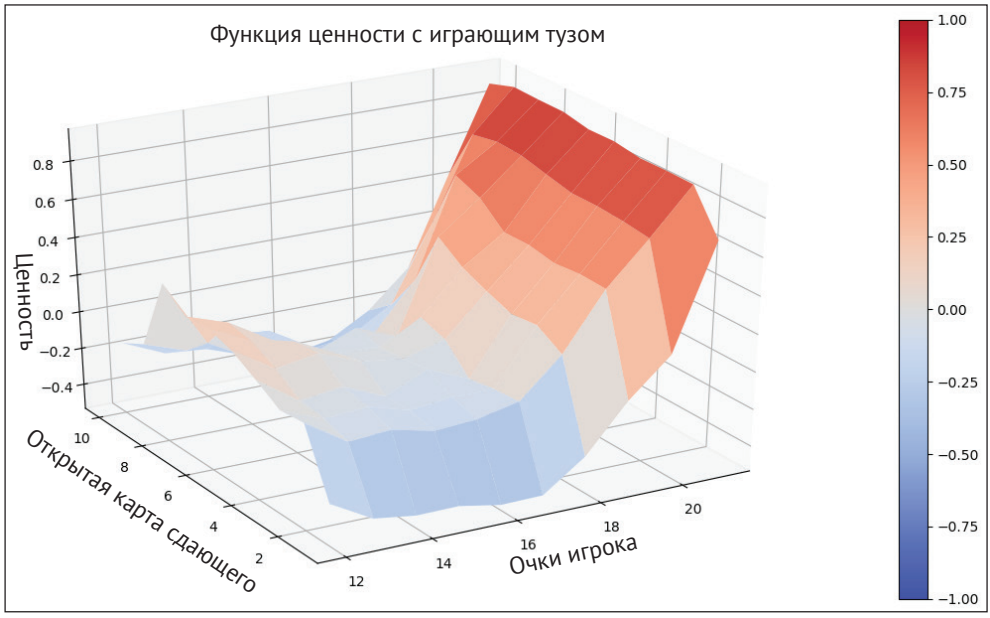
На шаге 6 распечатывается окончательная оптимальная стратегия:

```
{(21, 8, False): 0.9262458682060242, (11, 8, False): 0.16684606671333313,
(16, 10, False): -0.4662476181983948, (16, 10, True): -0.3643564283847809,
(14, 8, False): -0.2743947207927704, (13, 10, False): -0.3887477219104767,
(12, 9, False): -0.22795115411281586
  ......
  ......
(4, 3, False): -0.18421052396297455, (4, 8, False): -0.16806723177433014,
(13, 2, True): 0.05485232174396515, (5, 5, False): -0.09459459781646729,
(5, 8, False): -0.3690987229347229, (20, 2, True): 0.6965699195861816, (17,
2, True): -0.09696969389915466, (12, 2, True): 0.0517241396009922}
```

На шаге 7 строится график ценности состояний без играющего туза:



И график ценности состояний с играющим тузом:



Это еще не все

Интересно, действительно ли оптимальная стратегия лучше простой. Для ответа на этот вопрос сымитируем по 100 000 эпизодов игры в блэкджек для оптимальной и для простой стратегии, а затем сравним шансы на выигрыш.

1. Сначала определим простую стратегию, при которой игрок говорит «хватит», набрав 18 очков или больше.

```
>>> hold_score = 18
>>> hold_policy = {}
>>> player_sum_range = range(2, 22)
>>> for player in range(2, 22):
...     for dealer in range(1, 11):
...         action = 1 if player < hold_score else 0
...         hold_policy[(player, dealer, False)] = action
...         hold_policy[(player, dealer, True)] = action
```

2. Затем определим функцию-обертку, которая выполняет один эпизод, следуя заданной стратегии, и возвращает полное вознаграждение.

```
>>> def simulate_episode(env, policy):
...     state = env.reset()
...     is_done = False
...     while not is_done:
...         action = policy[state]
...         state, reward, is_done, info = env.step(action)
...         if is_done:
...             return reward
```

3. Зададим количество эпизодов (100 000) и инициализируем счетчики выигрышей и проигрышей:

```
>>> n_episode = 100000
>>> n_win_optimal = 0
>>> n_win_simple = 0
>>> n_lose_optimal = 0
>>> n_lose_simple = 0
```

4. Выполним 100 000 эпизодов, подсчитывая попутно выигрыши и проигрыши:

```
>>> for _ in range(n_episode):
...     reward = simulate_episode(env, optimal_policy)
...     if reward == 1:
...         n_win_optimal += 1
...     elif reward == -1:
...         n_lose_optimal += 1
...     reward = simulate_episode(env, hold_policy)
...     if reward == 1:
...         n_win_simple += 1
...     elif reward == -1:
...         n_lose_simple += 1
```

5. И напечатаем полученные результаты:

```
>>> print('Вероятность выигрыша при простой стратегии: {}'.
format(n_win_simple/n_episode))
'Вероятность выигрыша при простой стратегии: 0.39923
>>> print('Вероятность выигрыша при оптимальной стратегии: {}'.
format(n_win_optimal/n_episode))
Вероятность выигрыша при оптимальной стратегии: 0.41281
```

При следовании оптимальной стратегии шанс выиграть составляет 41.28 %, а при следовании простой – 39.92 %. Посчитаем также вероятность проигрыша:

```
>>> print('Вероятность проигрыша при простой стратегии:{}'
.format(n_lose_simple/n_episode))
Вероятность проигрыша при простой стратегии: 0.51024
>>> print('Вероятность проигрыша при оптимальной стратегии:{}'
.format(n_lose_optimal/n_episode))
Вероятность проигрыша при простой стратегии: 0.493
```

С другой стороны, шанс проиграть при следовании оптимальной стратегии составляет 49.3 %, а при следовании простой – 51.02 %. Без сомнения, оптимальная стратегия выигрывает по всем статьям!

Разработка управления методом Монте-Карло с ε-жадной стратегией

В предыдущем рецепте мы искали оптимальную стратегию методом управления МК – когда жадно выбирается действие с наибольшей ценностью пары состояние–действие. Но жадный выбор в начальных эпизодах не гарантирует нахождения оптимального решения. Если ограничиться только тем, что кажется наилучшим решением сейчас, игнорируя проблему в целом, то можно застрять в локальном оптимуме, так и не найдя глобального. Чтобы этого не произошло, применяется ε-жадная стратегия.

В управлении МК с ε-жадной стратегией мы не пытаемся всегда выбирать наилучшее из известных действий, а иногда выбираем случайное действие. Как следует из названия, у алгоритма есть два аспекта:

- эпсилон: параметр ε, принимающий значения от 0 до 1, определяет вероятность, с которой выбирается случайное действие:

  $\pi(s, a) = \varepsilon/|A|$,

  где |A| – количество возможных действий;
- жадность: с вероятностью 1 – ε выбирается действие, для которого ценность пары состояние–действие максимальна:

  $\pi(s, a) = 1 - \varepsilon + \varepsilon/|A|$.

ε-жадная стратегия в большинстве случаев выбирает наилучшее из известных действий, но время от времени исследует другие действия.

Как это делается

Займемся имитацией окружающей среды Blackjack с помощью ε-жадной стратегии.

1. Импортируем необходимые модули и создадим экземпляр среды Blackjack:

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

2. Напишем функцию, которая выполняет один эпизод, следуя ε-жадной стратегии.

```
>>> def run_episode(env, Q, epsilon, n_action):
...     """
...     Выполняет эпизод, следуя ε-жадной стратегии
...     @param env: имя окружающей среды OpenAI Gym
...     @param Q: Q-функция
...     @param epsilon: компромисс между исследованием и использованием
...     @param n_action: пространство действий
...     @return: результирующие состояния, действия и вознаграждения для
...              всего эпизода
...     """
...     state = env.reset()
...     rewards = []
...     actions = []
...     states = []
...     is_done = False
...     while not is_done:
...         probs = torch.ones(n_action) * epsilon / n_action
...         best_action = torch.argmax(Q[state]).item()
...         probs[best_action] += 1.0 - epsilon
...         action = torch.multinomial(probs, 1).item()
...         actions.append(action)
...         states.append(state)
...         state, reward, is_done, info = env.step(action)
...         rewards.append(reward)
...         if is_done:
...             break
...     return states, actions, rewards
```

3. Теперь реализуем управление МК с ε-жадной единой стратегией:

```
>>> from collections import defaultdict
>>> def mc_control_epsilon_greedy(env, gamma, n_episode, epsilon):
...     """
...     Строит оптимальную ε-жадную стратегию методом управления МК
...     с единой стратегией
...     @param env: имя окружающей среды OpenAI Gym
...     @param gamma: коэффициент обесценивания
...     @param n_episode: количество эпизодов
...     @param epsilon: компромисс между исследованием и использованием
...     @return: оптимальные Q-функция и стратегия
...     """
...     n_action = env.action_space.n
...     G_sum = defaultdict(float)
...     N = defaultdict(int)
...     Q = defaultdict(lambda: torch.empty(n_action))
...     for episode in range(n_episode):
```

```
...         states_t, actions_t, rewards_t =
                    run_episode(env, Q, epsilon, n_action)
...         return_t = 0
...         G = {}
...         for state_t, action_t, reward_t in zip(states_t[::-1],
                                actions_t[::-1], rewards_t[::-1]):
...             return_t = gamma * return_t + reward_t
...             G[(state_t, action_t)] = return_t
...         for state_action, return_t in G.items():
...             state, action = state_action
...             if state[0] <= 21:
...                 G_sum[state_action] += return_t
...                 N[state_action] += 1
...                 Q[state][action] = G_sum[state_action] / N[state_action]
...     policy = {}
...     for state, actions in Q.items():
...         policy[state] = torch.argmax(actions).item()
...     return Q, policy
```

4. Зададим коэффициент обесценивания 1, ε = 0.1 и количество эпизодов 500 000:

```
>>> gamma = 1
>>> n_episode = 500000
>>> epsilon = 0.1
```

5. Выполним алгоритм управления МК с ε-жадной стратегией, чтобы найти оптимальные Q-функцию и стратегию:

```
>>> optimal_Q, optimal_policy = mc_control_epsilon_greedy(env,
gamma, n_episode, epsilon)
```

Распечатать оптимальные ценности и построить графики вы можете сами, воспользовавшись написанными ранее функциями `plot_blackjack_value` и `plot_surface`.

6. И наконец, посмотрим, как работает ε-жадный метод. Снова сымитируем 100 000 эпизодов игры в блэкджек с оптимальной стратегией, найденной описанным выше способом, и подсчитаем вероятности выигрыша и проигрыша.

```
>>> n_episode = 100000
>>> n_win_optimal = 0
>>> n_lose_optimal = 0
>>> for _ in range(n_episode):
...     reward = simulate_episode(env, optimal_policy)
...     if reward == 1:
...         n_win_optimal += 1
...     elif reward == -1:
...         n_lose_optimal += 1
```

Здесь повторно используется функция `simulate_episode` из предыдущего рецепта.

Как это работает

В этом рецепте мы применили к игре в блэкджек управление МК с единой ε-жадной стратегией.

На шаге 2, следуя ε-жадной стратегии, выполняется один эпизод и производятся следующие действия:

- эпизод инициализируется;
- вычисляются вероятности каждого действия; лучшее согласно текущей Q-функции действие выбирается с вероятностью $1 - \varepsilon + \varepsilon/|A|$, в противном случае выбирается случайное действие;
- сохраняются состояния, действия и вознаграждения на всех шагах эпизода, чтобы использовать их на этапе оценивания.

ε-жадная стратегия лучше жадного метода поиска, поскольку использует лучшее действие лишь с вероятностью $1 - \varepsilon + \varepsilon/|A|$, но при этом допускает исследование других действий с вероятностью $\varepsilon/|A|$. Гиперпараметр ε представляет компромисс между использованием и исследованием. Если он равен 0, то алгоритм становится чисто жадным, а если 1, то каждое действие выбирается с равной вероятностью, т. е. алгоритм вырождается в случайное исследование.

Значение ε следует подбирать экспериментально, нет никакого универсального рецепта для любой ситуации. Тем не менее, как правило, начинают со значения 0.1, 0.2 или 0.3. Другой подход – взять значение побольше (скажем, 0.5 или 0.7) и постепенно уменьшать его (скажем, умножая на 0.999 в каждом эпизоде). Тогда вначале стратегия будет уделять больше внимания исследованию, а в конце станет использовать хорошие действия.

Наконец, после шага 6 мы усредняем результаты по 100 000 эпизодов и печатаем вероятность выигрыша:

```
>>> print('Вероятность выигрыша при оптимальной стратегии: {}'.
format(n_win_optimal/n_episode))
Вероятность выигрыша при оптимальной стратегии: 0.42436
```

Оптимальная стратегия, найденная ε-жадным методом, выигрывает с вероятностью 42.44 %, это больше, чем в случае стратегии без исследования (41.28 %).

И еще напечатаем вероятность проигрыша:

```
>>> print('Вероятность проигрыша при оптимальной стратегии:{}'
.format(n_lose_optimal/n_episode))
Вероятность проигрыша при простой стратегии: 0.48048
```

Как видим, при ε-жадной стратегии шансы на проигрыш меньше, чем при стратегии без исследования (48.05 % и 49.3 % соответственно).

Управление методом Монте-Карло с разделенной стратегией

Еще один подход к решению МППР методами Монте-Карло – управление с разделенной стратегией, которое мы изучим в этом рецепте.

Метод **с разделенной стратегией** оптимизирует **целевую стратегию** π на данных, сгенерированных другой, **поведенческой стратегией** b. Целевая стратегия только использует лучшие действия, тогда как поведенческая служит для исследования. Это означает, что целевая стратегия жадная относительно своей текущей Q-функции, а поведенческая генерирует поведение, чтобы у целевой стратегии были данные, на которых можно обучаться. Поведенческая стратегия может быть любой, при условии что все действия во всех состояниях выбираются с ненулевой вероятностью, это гарантирует, что стратегия может исследовать все возможности.

Поскольку в методе с разделенной стратегией мы имеем дело с двумя разными стратегиями, в эпизодах, встречающихся в обеих стратегиях, можно использовать только **общие** шаги. Это означает, что мы начинаем с последнего шага, для которого действие, предпринятое согласно поведенческой стратегии, отличается от действия, предпринятого согласно жадной стратегии. А чтобы обучить целевую стратегию с помощью другой стратегии, применяется **выборка по значимости** – техника, которая часто используется для оценивания математического ожидания некоторого распределения при наличии выборок из другого распределения. Взвешенная значимость пары состояние–действие вычисляется следующим образом:

$$w_t = \sum_{k=t} [\pi(a_k|s_k)/b(a_k|s_k),$$

где $\pi(a_k|s_k)$ – вероятность выбора действия a_k в состоянии s_k при следовании целевой стратегии; $b(a_k|s_k)$ – вероятность того же при следовании поведенческой стратегии, а вес w_t – сумма отношений этих вероятностей по всем шагам, начиная с t-го и до конца эпизода. Вес w_t применяется к доходу на шаге t.

Как это делается

Для нахождения оптимальной стратегии игры в блэкджек методом управления МК с разделенной стратегией выполним следующие действия.

1. Импортируем необходимые модули и создадим экземпляр среды Blackjack:

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

2. Сначала определим поведенческую стратегию, которая случайным образом с одинаковой вероятностью выбирает действия.

```
>>> def gen_random_policy(n_action):
...     probs = torch.ones(n_action) / n_action
...     def policy_function(state):
...         return probs
...     return policy_function
>>> random_policy = gen_random_policy(env.action_space.n)
```

   Поведенческая стратегия может быть любой, при условии что все действия во всех состояниях выбираются с ненулевой вероятностью.

3. Затем напишем функцию, которая выполняет один эпизод и выбирает действия, следуя поведенческой стратегии:

```
>>> def run_episode(env, behavior_policy):
...     """
...     Выполняет один эпизод, следуя заданной поведенческой стратегии
...     @param env: имя окружающей среды OpenAI Gym
...     @param behavior_policy: поведенческая стратегия
...     @return: результирующие состояния, действия и вознаграждения для
...              всего эпизода
...     """
...     state = env.reset()
...     rewards = []
...     actions = []
...     states = []
...     is_done = False
...     while not is_done:
...         probs = behavior_policy(state)
...         action = torch.multinomial(probs, 1).item()
...         actions.append(action)
...         states.append(state)
...         state, reward, is_done, info = env.step(action)
...         rewards.append(reward)
...         if is_done:
...             break
...     return states, actions, rewards
```

Здесь запоминаются состояния, действия и вознаграждения на всех шагах эпизода, позже они будут использоваться в качестве обучающих данных для целевой стратегии.

4. Теперь реализуем алгоритм управления МК с разделенной стратегией:

```
>>> from collections import defaultdict
>>> def mc_control_off_policy(env, gamma, n_episode, behavior_policy):
...     """
...     Строит оптимальную стратегию методом управления МК
...     с разделенной стратегией
...     @param env: имя окружающей среды OpenAI Gym
...     @param gamma: коэффициент обесценивания
...     @param n_episode: количество эпизодов
...     @param behavior_policy: поведенческая стратегия
...     @return: оптимальные Q-функция и стратегия
...     """
...     n_action = env.action_space.n
...     G_sum = defaultdict(float)
...     N = defaultdict(int)
...     Q = defaultdict(lambda: torch.empty(n_action))
...     for episode in range(n_episode):
...         W = {}
...         w = 1
...         states_t, actions_t, rewards_t = run_episode(env, behavior_policy)
...         return_t = 0
```

```
...         G = {}
...         for state_t, action_t, reward_t in zip(states_t[::-1],
                                actions_t[::-1], rewards_t[::-1]):
...             return_t = gamma * return_t + reward_t
...             G[(state_t, action_t)] = return_t
...             if action_t != torch.argmax(Q[state_t]).item():
...                 break
...             w *= 1./ behavior_policy(state_t)[action_t]
...         for state_action, return_t in G.items():
...             state, action = state_action
...             if state[0] <= 21:
...                 G_sum[state_action] += return_t * W[state_action]
...             N[state_action] += 1
...             Q[state][action] = G_sum[state_action] / N[state_action]
...     policy = {}
...     for state, actions in Q.items():
...         policy[state] = torch.argmax(actions).item()
...     return Q, policy
```

5. Зададим коэффициент обесценивания 1 и количество эпизодов 500 000:

```
>>> gamma = 1
>>> n_episode = 500000
```

6. Выполним алгоритм управления МК с разделенной стратегией, чтобы найти оптимальные Q-функцию и стратегию:

```
>>> optimal_Q, optimal_policy = mc_control_off_policy(env, gamma,
n_episode, random_policy)
```

Как это работает

В этом рецепте мы применили управление МК с разделенной стратегией к игре блэкджек.

На шаге 1 выполняются следующие действия:

- Q-функция инициализируется случайными малыми значениями;
- прогоняется `n_episode` эпизодов:
- для каждого эпизода выполняется поведенческая стратегия, которая генерирует состояния, действия и вознаграждения. Затем выполняется оценивание целевой стратегии методом МК первого посещения на основе **общих** шагов и Q-функция обновляется с учетом взвешенного дохода;
- в конце получается оптимальная Q-функция и оптимальная стратегия, которая в каждом состоянии выбирает наилучшее действие в соответствии с этой Q-функцией.

Алгоритм обучает целевую стратегию, наблюдая за поведением другого агента и используя опыт, накопленный при следовании иной стратегии. Целевая стратегия оптимизируется жадно, тогда как поведенческая продолжает исследовать различные варианты. Производится усреднение доходов, полученных при следовании поведенческой стратегии с коэффициентами выборки

по значимости относительно целевой стратегии. Возможно, вы недоумеваете, почему при вычислении веса w_t величина $\pi(a_k|s_k)$ всегда равна 1. Напомним, что мы рассматриваем только общие шаги, сделанные при следовании поведенческой стратегии и предположительно целевой стратегии, а целевая стратегия всегда жадная. Поэтому $\pi(a|s)$ всегда равно 1.

Это еще не все

Метод МК можно реализовать инкрементно. При прохождении эпизода, вместо того чтобы сохранять доход и коэффициенты выборки по значимости для каждого первого посещения пары состояние–действие, мы можем вычислять Q-функцию динамически. В случае неинкрементного подхода Q-функция вычисляется в конце, когда известны все доходы в n эпизодах:

$$V_n = \left(\sum_{k=1}^{n} w_k R_k\right)/n.$$

В случае же инкрементного подхода Q-функция обновляется на каждом шаге эпизода по формуле

$$v_{n+1} = V_n + w_{n+1}(R_{n+1} - V_n)/(n+1).$$

Инкрементный подход более эффективен, потому что позволяет уменьшить потребление памяти и лучше масштабируется. Ниже показана его реализация.

```
>>> def mc_control_off_policy_incremental(env, gamma, n_episode, behavior_policy):
...     n_action = env.action_space.n
...     N = defaultdict(int)
...     Q = defaultdict(lambda: torch.empty(n_action))
...     for episode in range(n_episode):
...         W = 1.
...         states_t, actions_t, rewards_t = run_episode(env, behavior_policy)
...         return_t = 0.
...         for state_t, action_t, reward_t in
                    zip(states_t[::-1], actions_t[::-1], rewards_t[::-1]):
...             return_t = gamma * return_t + reward_t
...             N[(state_t, action_t)] += 1
...             Q[state_t][action_t] += (W / N[(state_t, action_t)])
                        * (return_t - Q[state_t][action_t])
...             if action_t != torch.argmax(Q[state_t]).item():
...                 break
...             W *= 1./ behavior_policy(state_t)[action_t]
...     policy = {}
...     for state, actions in Q.items():
...         policy[state] = torch.argmax(actions).item()
...     return Q, policy
```

Для нахождения оптимальной стратегии вызовем эту функцию:

```
>>> optimal_Q, optimal_policy = mc_control_off_policy_incremental(env,
gamma, n_episode, random_policy)
```

См. также

Подробное объяснение выборки по значимости см. в документе по адресу https://statweb.stanford.edu/~owen/mc/Ch-var-is.pdf.

Разработка управления методом Монте-Карло со взвешенной выборкой по значимости

В предыдущем рецепте мы просто усредняли доходы, полученные при следовании поведенческой стратегии с коэффициентами выборки по значимости относительно целевой стратегии. Эта техника носит название **обыкновенной выборки по значимости**. Известно, что у нее высокая дисперсия, поэтому обычно предпочтение отдается взвешенной выборке по значимости, о которой мы и поговорим в этом рецепте.

Взвешенная выборка по значимости отличается от обыкновенной тем, как усредняются доходы. Вместо простого усреднения производится взвешенное:

$$V_n = \left(\sum_{k=1}^{n} w_k R_k\right) / \sum_{k=1}^{n} w_k.$$

Зачастую дисперсия оказывается гораздо ниже, чем в обыкновенной версии. Попробовав применить обыкновенную выборку по значимости к среде Blackjack, вы обнаружите, что результаты разных прогонов сильно различаются.

Как это делается

Чтобы применить управление МК с разделенной стратегией и взвешенной выборкой по значимости, выполним следующие действия.

1. Импортируем необходимые модули и создадим экземпляр среды Blackjack:

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

2. Сначала определим поведенческую стратегию, которая случайным образом с одинаковой вероятностью выбирает действия.

```
>>> random_policy = gen_random_policy(env.action_space.n)
```

3. Затем воспользуемся написанной ранее функцией `run_episode`, которая выполняет один эпизод, выбирая действия в соответствии с поведенческой стратегией.

4. Теперь реализуем алгоритм управления МК с разделенной стратегией и взвешенной выборкой по значимости:

```
>>> from collections import defaultdict
>>> def mc_control_off_policy_weighted(env, gamma, n_episode,
behavior_policy):
```

```
...     """
...     Строит оптимальную стратегию методом управления МК
...     с разделенной стратегией и взвешенной выборкой по значимости
...     @param env: имя окружающей среды OpenAI Gym
...     @param gamma: коэффициент обесценивания
...     @param n_episode: количество эпизодов
...     @param behavior_policy: поведенческая стратегия
...     @return: оптимальные Q-функция и стратегия
...     """
...     n_action = env.action_space.n
...     N = defaultdict(float)
...     Q = defaultdict(lambda: torch.empty(n_action))
...     for episode in range(n_episode):
...         W = 1.
...         states_t, actions_t, rewards_t = run_episode(env, behavior_policy)
...         return_t = 0.
...         for state_t, action_t, reward_t in zip(states_t[::-1],
                                actions_t[::-1], rewards_t[::-1]):
...             return_t = gamma * return_t + reward_t
...             N[(state_t, action_t)] += W
...             Q[state_t][action_t] += (W / N[(state_t, action_t)])
                            * (return_t - Q[state_t][action_t])
...             if action_t != torch.argmax(Q[state_t]).item():
...                 break
...             W *= 1./ behavior_policy(state_t)[action_t]
...     policy = {}
...     for state, actions in Q.items():
...         policy[state] = torch.argmax(actions).item()
...     return Q, policy
```

Это инкрементный вариант управления МК.

5. Зададим коэффициент обесценивания 1 и количество эпизодов 500 000:

```
>>> gamma = 1
>>> n_episode = 500000
```

6. Выполним управление МК с разделенной стратегией, задав поведенческую стратегию `random_policy`, и найдем оптимальные Q-функцию и стратегию:

```
>>> optimal_Q, optimal_policy = mc_control_off_policy_weighted(env,
gamma, n_episode, random_policy)
```

Как это работает

Мы только что применили к окружающей среде Blackjack алгоритм управления МК с разделенной стратегией и взвешенной выборкой по значимости. Это очень похоже на обыкновенную выборку по значимости, но вместо умножения доходов на коэффициенты и усреднения результатов мы масштабируем доходы с помощью взвешенного среднего. На практике взвешенная выборка по значимости имеет гораздо меньшую дисперсию, чем обыкновенная, поэтому является более предпочтительной.

Это еще не все

Наконец, давайте сымитируем несколько эпизодов и посмотрим, каковы шансы на выигрыш и проигрыш при следовании полученной оптимальной стратегии.

Воспользуемся функцией `simulate_episode`, разработанной в рецепте «Управление методом Монте-Карло с единой стратегией», и выполним 100 000 эпизодов.

```
>>> n_episode = 100000
>>> n_win_optimal = 0
>>> n_lose_optimal = 0
>>> for _ in range(n_episode):
...     reward = simulate_episode(env, optimal_policy)
...     if reward == 1:
...         n_win_optimal += 1
...     elif reward == -1:
...         n_lose_optimal += 1
```

И распечатаем результаты:

```
>>> print('Вероятность выигрыша при оптимальной стратегии: {}'.
format(n_win_optimal/n_episode))
'Вероятность выигрыша при оптимальной стратегии: 0.43072
>>> print('Вероятность проигрыша при оптимальной стратегии: {}'.
format(n_lose_optimal/n_episode))
Вероятность проигрыша при оптимальной стратегии: 0.47756
```

См. также

Доказательство того, что взвешенная выборка по значимости действительно лучше обыкновенной, можно найти в следующих работах:

- *Hesterberg T. C.* Advances in importance sampling. Ph. D. Dissertation, Statistics Department, Stanford University, 1988;
- *Casella G., Robert C. P.* Post-processing accept-reject samples: recycling and rescaling // Journal of Computational and Graphical Statistics, 7 (2): 139–157, 1988;
- *Precup D., Sutton R. S., Singh S.* Eligibility traces for off-policy policy evaluation. In: Proceedings of the 17th International Conference on Machine Learning, p. 759–766, 2000.

Глава 4

TD-обучение и Q-обучение

В предыдущей главе мы решали МППР методом Монте-Карло, который относится к семейству безмодельных алгоритмов, не имеющих априорной информации об окружающей среде. Однако в обучении методом МК функция ценности и Q-функция обычно обновляются только в конце эпизода. Это плохо, поскольку бывают очень длинные и даже бесконечные процессы. В этой главе мы рассмотрим метод обучения на основе временных различий (TD-обучения), который решает эту проблему. В алгоритме TD-обучения ценности действий обновляются на каждом временном шаге эпизода, что значительно повышает эффективность обучения.

Мы начнем эту главу с обсуждения окружающих сред Cliff Walking (блуждание на краю обрыва) и Windy Gridworld (ветреный сеточный мир), на которых продемонстрируем обсуждаемые методы TD-обучения. Следуя пошаговым инструкциям, читатель получит практический опыт применения Q-обучения для управления с разделенной стратегией и алгоритма SARSA для управления с единой стратегией. Мы также разберем интересную задачу о поездке на такси и покажем, как она решается обоими методами. И напоследок обсудим алгоритм двойного Q-обучения.

В этой главе приводятся следующие рецепты:

- подготовка окружающей среды Cliff Walking;
- реализация алгоритма Q-обучения;
- подготовка окружающей среды Windy Gridworld;
- реализация алгоритма SARSA;
- решение задачи о такси методом Q-обучения;
- решение задачи о такси методом SARSA;
- реализация алгоритма двойного Q-обучения.

Подготовка окружающей среды Cliff Walking

В первом рецепте мы познакомимся со средой Cliff Walking, на которой будем опробовать описываемые далее TD-методы.

Cliff Walking – типичная окружающая среда Gym с длинными эпизодами без гарантии завершения. Это задача на сетке 4×12. На каждом шаге агент делает

ход вверх, вправо, вниз или влево. Вначале агент находится в левом нижнем углу, а для успешного завершения эпизода должен перейти в правый нижний угол. Все остальные ячейки в последней строке – обрыв, при попадании в них агент возвращается на исходную позицию, но эпизод продолжается. За каждый шаг агенту начисляется вознаграждение –1, а за падение с обрыва – вознаграждение –100.

Подготовка

Для начала найдем имя окружающей среды Cliff Walking в таблице по адресу https://github.com/openai/gym/wiki/Table-of-environments. Она называется `CliffWalking-v0`, пространство наблюдений в ней представлено целыми числами от 0 (левый верхний угол) до 47 (правый нижний угол), а действий всего четыре (вверх = 0, вправо = 1, вниз = 2, влево = 3).

Как это делается

Для имитации среды Cliff Walking выполним следующие действия.

1. Импортируем библиотеку Gym и создадим экземпляр окружающей среды Cliff Walking:

```
>>> import gym
>>> env = gym.make("CliffWalking-v0")
>>> n_state = env.observation_space.n
>>> print(n_state)
48
>>> n_action = env.action_space.n
>>> print(n_action)
4
```

2. Приведем окружающую среду в исходное состояние:

```
>>> env.reset()
0
```

   Агент начинает работу в состоянии 36, в левом нижнем углу.

3. Нарисуем окружающую среду:

```
>>> env.render()
```

4. Попробуем сделать ход вниз, несмотря на то что такой ход невозможен.

```
>>> new_state, reward, is_done, info = env.step(2)
>>> env.render()
o o o o o o o o o o o o
o o o o o o o o o o o o
o o o o o o o o o o o o
x C C C C C C C C C C T
```

   Агент остается на месте. Напечатаем, что получилось:

```
>>> print(new_state)
36
```

```
>>> print(reward)
-1
```

Этот ход, как и любой другой, приносит вознаграждение –1:

```
>>> print(is_done)
False
```

Эпизод не завершен, т. к. агент еще не достиг цели:

```
>>> print(info)
{'prob': 1.0}
```

Это означает, что новое положение однозначно определяется ходом. Теперь сделаем ход вверх – это возможно:

```
>>> new_state, reward, is_done, info = env.step(0)
>>> env.render()
o o o o o o o o o o o o
o o o o o o o o o o o o
x o o o o o o o o o o o
o C C C C C C C C C T
```

Напечатаем, что получилось:

```
>>> print(new_state)
24
```

Агент сдвинулся вверх:

```
>>> print(reward)
-1
```

И это принесло вознаграждение –1.

5. Теперь попробуем сделать ход вправо и вниз:

```
>>> new_state, reward, is_done, info = env.step(1)
>>> new_state, reward, is_done, info = env.step(2)
>>> env.render()
o o o o o o o o o o o o
o o o o o o o o o o o o
o o o o o o o o o o o o
x C C C C C C C C C T
```

Агент сорвался с обрыва, поэтому был возвращен в исходную точку и заработал вознаграждение –100:

```
>>> print(new_state)
36
>>> print(reward)
-100
>>> print(is_done)
False
```

6. Наконец, пройдем по кратчайшему пути к цели:

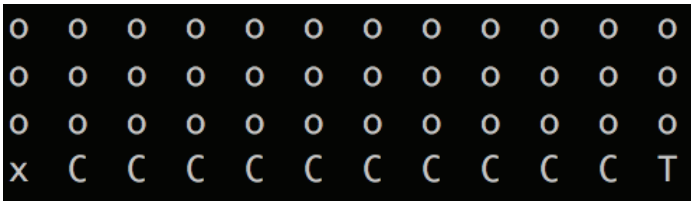
```
>>> new_state, reward, is_done, info = env.step(0)
>>> for _ in range(11):
```

```
...     env.step(1)
>>> new_state, reward, is_done, info = env.step(2)
>>> env.render()
o o o o o o o o o o o o
o o o o o o o o o o o o
o o o o o o o o o o o o
o C C C C C C C C C x
>>> print(new_state)
47
>>> print(reward)
-1
>>> print(is_done)
True
```

Как это работает

На шаге 1 мы импортировали библиотеку Gym и создали экземпляр окружающей среды Cliff Walking. Затем на шаге 2 сбросили среду в исходное состояние.

На шаге 3 мы нарисовали окружающую среду – матрицу 4×12, которая представляет сетку. Буквой x обозначено начальное положение агента, буквой T – конечная ячейка, буквами C – обрыв, а буквами o – обычные ячейки:



На шагах 4, 5, 6 мы сделали несколько ходов и понаблюдали за результатами и полученными вознаграждениями.

Понятно, что эпизод в среде Cliff Walking может длиться очень долго и даже бесконечно, поскольку падение с обрыва возвращает агента в начальное положение, но не завершает игру. Чем раньше достигнута цель, тем лучше, поскольку каждый результат уменьшает вознаграждение – на –1 или на –100. В следующем рецепте мы покажем, как решить задачу о блуждании на краю обрыва с помощью TD-метода.

Реализация алгоритма Q-обучения

Обучение на основе временных различий (TD-обучение), как и обучение методом Монте-Карло, является безмодельным алгоритмом. Напомним, что при обучении методом МК Q-функция обновляется в конце эпизода (как в режиме первого посещения, так и в режиме всех посещений). Главное достоинство TD-обучения состоит в том, что Q-функция обновляется на каждом шаге эпизода.

В этом рецепте мы рассмотрим популярный TD-метод – **Q-обучение**. Это алгоритм обучения с разделенной стратегией. В нем Q-функция обновляется по следующей формуле:

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

где s' – состояние, в которое переходит среда после выполнения действия a в состоянии s, r – начисленное вознаграждение, α – скорость обучения, γ – коэффициент обесценивания. Член $\max_{a'} Q(s', a')$ означает, что поведенческая стратегия жадная, т. е. для генерирования обучающих данных выбирается действие с наибольшим значением Q-функции в состоянии s'. В методе Q-обучения действия выбираются в соответствии с ε-жадной стратегией.

Как это делается

Реализуем алгоритм Q-обучения для взаимодействия со средой Cliff Walking.

1. Импортируем библиотеки PyTorch и Gym и создадим экземпляр окружающей среды Cliff Walking:

```
>>> import torch
>>> import gym
>>> env = gym.make("CliffWalking-v0")
>>> from collections import defaultdict
```

2. Определим ε-жадную стратегию:

```
>>> def gen_epsilon_greedy_policy(n_action, epsilon):
...     def policy_function(state, Q):
...         probs = torch.ones(n_action) * epsilon / n_action
...         best_action = torch.argmax(Q[state]).item()
...         probs[best_action] += 1.0 - epsilon
...         action = torch.multinomial(probs, 1).item()
...         return action
...     return policy_function
```

3. Напишем функцию, выполняющую Q-обучение:

```
>>> def q_learning(env, gamma, n_episode, alpha):
...     """
...     Строит оптимальную стратегию методом Q-обучения с разделенной
        стратегией
...     @param env: имя окружающей среды OpenAI Gym
...     @param gamma: коэффициент обесценивания
...     @param n_episode: количество эпизодов
...     @return: оптимальные Q-функция и стратегия
...     """
...     n_action = env.action_space.n
...     Q = defaultdict(lambda: torch.zeros(n_action))
...     for episode in range(n_episode):
...         state = env.reset()
...         is_done = False
...         while not is_done:
```

```
...             action = epsilon_greedy_policy(state, Q)
...             next_state, reward, is_done, info = env.step(action)
...             td_delta = reward + gamma * torch.max(Q[next_state])
                           - Q[state][action]
...             Q[state][action] += alpha * td_delta
...             if is_done:
...                 break
...             state = next_state
...     policy = {}
...     for state, actions in Q.items():
...         policy[state] = torch.argmax(actions).item()
...     return Q, policy
```

4. Зададим коэффициент обесценивания 1, скорость обучения 0.4, ε = 0.1 и выполним 500 эпизодов:

```
>>> gamma = 1
>>> n_episode = 500
>>> alpha = 0.4
>>> epsilon = 0.1
```

5. Создадим экземпляр ε-жадной стратегии:

```
>>> epsilon_greedy_policy =
gen_epsilon_greedy_policy(env.action_space.n, epsilon)
```

6. Выполним Q-обучение с заданными выше параметрами и распечатаем оптимальную стратегию:

```
>>> optimal_Q, optimal_policy = q_learning(env, gamma, n_episode, alpha)
>>> print('Оптимальная стратегия:\n', optimal_policy)
Оптимальная стратегия:
{36: 0, 24: 1, 25: 1, 13: 1, 12: 2, 0: 3, 1: 1, 14: 2, 2: 1, 26:
1, 15: 1, 27: 1, 28: 1, 16: 2, 4: 2, 3: 1, 29: 1, 17: 1, 5: 0, 30:
1, 18: 1, 6: 1, 19: 1, 7: 1, 31: 1, 32: 1, 20: 2, 8: 1, 33: 1, 21:
1, 9: 1, 34: 1, 22: 2, 10: 2, 23: 2, 11: 2, 35: 2, 47: 3}
```

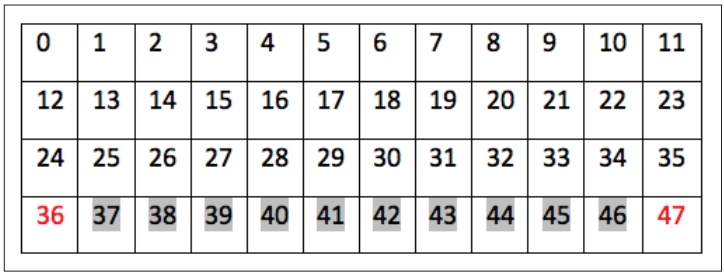
Как это работает

На шаге 2 ε-жадная стратегия принимает параметр ε от 0 до 1 и количество возможных действий |A|. С вероятностью ε/|A| действие выбирается произвольное действие, а с вероятностью 1 – ε + ε/|A| действие – с наибольшей ценностью пары состояние–действие.

На шаге 3 выполняется Q-обучение:

- инициализируем таблицу значений Q-функции нулями;
- в каждом эпизоде агент выбирает действие, следуя ε-жадной стратегии. И после каждого шага Q-функция обновляется;
- выполняем `n_episode` эпизодов;
- получаем оптимальную стратегию на основе оптимальной Q-функции.

Оптимальная стратегия, распечатанная на шаге 6, показывает, что из начального состояния 36 агент делает шаг вверх в состояние 24, затем идет вправо до состояния 35 и, делая шаг вниз, достигает цели:



Как видим, алгоритм Q-обучения оптимизирует Q-функцию, обучаясь на опыте, сгенерированном другой стратегией. Это очень похоже на управление методом МК с разделенной стратегией. Разница в том, что Q-функция обновляется после каждого шага, а не в конце эпизода. Это лучше работает в окружающих средах, где эпизоды длинные и ждать, пока эпизод завершится, неэффективно. На каждом шаге Q-обучения (или любого другого TD-метода) мы получаем новую информацию об окружающей среде и используем эту информацию для немедленного обновления ценностей. В нашем примере для нахождения оптимальной стратегии понадобилось всего 500 эпизодов.

Это еще не все

Для нахождения оптимальной стратегии понадобилось около 50 эпизодов. Мы можем нанести на график длину каждого эпизода. Также можно графически представить, как изменялась величина вознаграждения.

1. Определим два списка: для хранения длин эпизодов и полученных в них вознаграждений:

```
>>> length_episode = [0] * n_episode
>>> total_reward_episode = [0] * n_episode
```

2. В процессе обучения будем сохранять длину эпизода и вознаграждение в нем. Ниже приведена модифицированная функция `q_learning`:

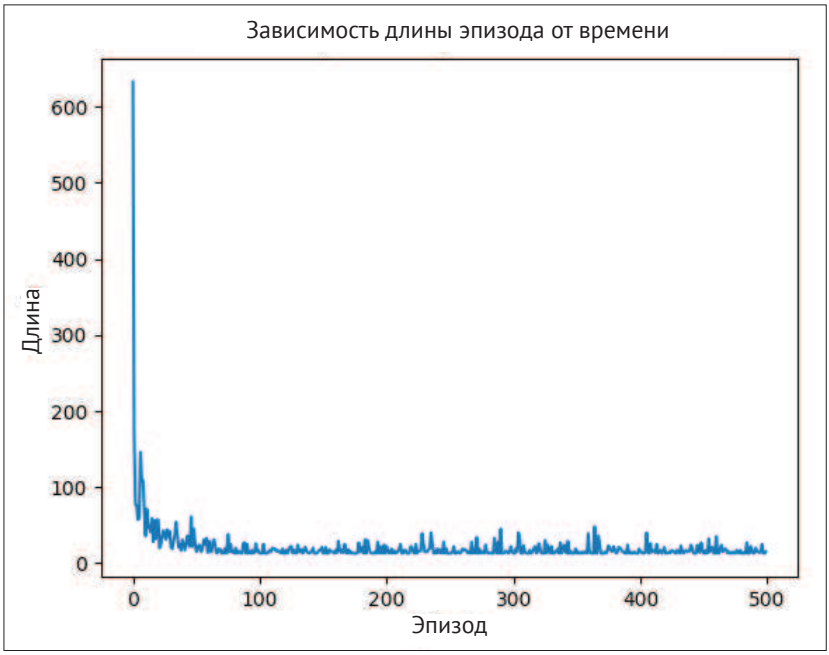
```
>>> def q_learning(env, gamma, n_episode, alpha):
...     n_action = env.action_space.n
...     Q = defaultdict(lambda: torch.zeros(n_action))
...     for episode in range(n_episode):
...         state = env.reset()
...         is_done = False
...         while not is_done:
...             action = epsilon_greedy_policy(state, Q)
...             next_state, reward, is_done, info = env.step(action)
...             td_delta = reward + gamma * torch.max(Q[next_state])
                                  - Q[state][action]
...             Q[state][action] += alpha * td_delta
...             length_episode[episode] += 1
...             total_reward_episode[episode] += reward
...             if is_done:
...                 break
```

```
...             state = next_state
...     policy = {}
...     for state, actions in Q.items():
...         policy[state] = torch.argmax(actions).item()
...     return Q, policy
```

3. Построим график зависимости длины эпизода от времени:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(length_episode)
>>> plt.title('Зависимость длины эпизода от времени')
>>> plt.xlabel('Эпизод')
>>> plt.ylabel('Длина')
>>> plt.show()
```

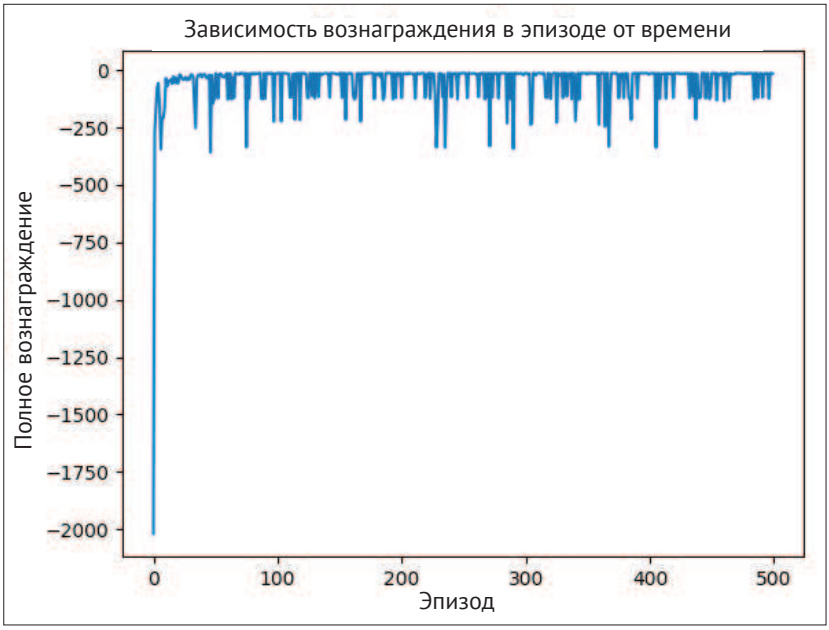
Вот как он выглядит:



4. Теперь построим график зависимости вознаграждения от времени:

```
>>> plt.plot(total_reward_episode)
>>> plt.title('Зависимость вознаграждения в эпизоде от времени')
>>> plt.xlabel('Эпизод')
>>> plt.ylabel('Полное вознаграждение')
>>> plt.show()
```

Вот как выглядит результат:

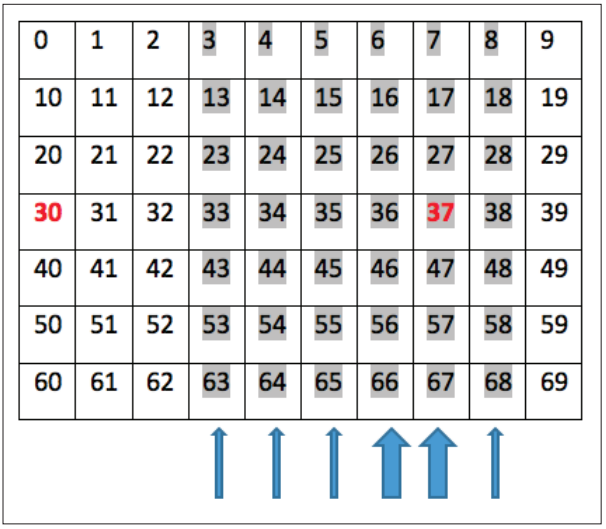


Если уменьшить значение ε, то флуктуации станут слабее – это эффект случайного исследования в ε-жадной стратегии.

Подготовка окружающей среды Windy Gridworld

В предыдущем рецепте мы взаимодействовали со сравнительно простой окружающей средой и легко нашли оптимальную стратегию. Сейчас займемся более сложной окружающей средой Windy Gridworld, тоже на сетке, в которой действует внешняя сила, сдувающая агента из некоторых ячеек. А в следующем рецепте мы применим TD-метод для поиска оптимальной стратегии в этой среде.

Среда Windy Gridworld развернута на сетке 7×10, изображенной на рисунке ниже:



Агент может двигаться вверх, вправо, вниз и влево. Первоначально он находится в ячейке 30, а цель – попасть в ячейку 37, в этот момент эпизод заканчивается. За каждый сделанный шаг агенту начисляется вознаграждение –1.

Сложность этой среды в том, что в столбцах с 4 по 9 дует ветер. Если агент оказывается в любой ячейке из этих столбцов, то его будет сдувать вверх. Сила ветра в седьмом и восьмом столбцах равна 1, а в четвертом, пятом, шестом и девятом – 2. Например, если агент попытается сделать шаг вправо из состояния 43, то окажется в состоянии 34. Сделав шаг влево из состояния 48, агент окажется в состоянии 37. Сделав шаг вверх из состояния 67, он окажется в состоянии 37, потому что ветер сдует его еще на две ячейки вверх. Если же он сделает шаг вниз из состояния 27, то окажется в состоянии 17, потому что ветер сдувает на две ячейки вверх, пресекая попытку спуститься.

> В настоящее время окружающая среда Windy Gridworld не включена в комплект Gym. Мы реализуем ее, взяв за основу код среды Cliff Walking, находящийся по адресу https://github.com/openai/gym/blob/master/gym/envs/toy_text/cliffwalking.py.

Как это делается

Реализуем среду Windy Gridworld:

1. Импортируем необходимые модули: NumPy и класс `discrete` из библиотеки Gym.

```
>>> import numpy as np
>>> import sys
>>> from gym.envs.toy_text import discrete
```

2. Определим четыре действия:

```
>>> UP = 0
>>> RIGHT = 1
>>> DOWN = 2
>>> LEFT = 3
```

3. Напишем метод `__init__` класса `WindyGridworldEnv`:

```
>>> class WindyGridworldEnv(discrete.DiscreteEnv):
...     def __init__(self):
...         self.shape = (7, 10)
...         nS = self.shape[0] * self.shape[1]
...         nA = 4
...         # Столбцы, в которых дует ветер
...         winds = np.zeros(self.shape)
...         winds[:,[3,4,5,8]] = 1
...         winds[:,[6,7]] = 2
...         self.goal = (3, 7)
...         # Задать вероятности переходов и вознаграждения
...         P = {}
...         for s in range(nS):
...             position = np.unravel_index(s, self.shape)
...             P[s] = {a: [] for a in range(nA)}
...             P[s][UP] = self._calculate_transition_prob(
                                    position, [-1, 0], winds)
...             P[s][RIGHT] = self._calculate_transition_prob(
                                    position, [0, 1], winds)
...             P[s][DOWN] = self._calculate_transition_prob(
                                    position, [1, 0], winds)
...             P[s][LEFT] = self._calculate_transition_prob(
                                    position, [0, -1], winds)
...         # Задать начальное состояние
...         # Агент всегда начинает в ячейке (3, 0)
...         isd = np.zeros(nS)
...         isd[np.ravel_multi_index((3,0), self.shape)] = 1.0
...         super(WindyGridworldEnv, self).__init__(nS, nA, P, isd)
```

Здесь определяются пространство наблюдений, ветреные участки и сила ветра, матрицы переходов и вознаграждений и начальное состояние.

4. Определим метод `_calculate_transition_prob`, который возвращает результат действия: вероятность (она всегда равна 1), новое состояние, вознаграждение (всегда равно –1) и признак завершения эпизода.

```
...     def _calculate_transition_prob(self, current, delta, winds):
...         """
...         Определяет результат действия. Вероятность перехода всегда
...         равна 1.0.
...         @param current: (row, col), текущая позиция в сетке
...         @param delta: изменение позиции при переходе
...         @param winds: эффект ветра
...         @return: (1.0, new_state, reward, is_done)
...         """
```

```
...         new_position = np.array(current) + np.array(delta)
                    + np.array([-1, 0]) * winds[tuple(current)]
...         new_position = self._limit_coordinates(new_position).astype(int)
...         new_state = np.ravel_multi_index(tuple(new_position), self.shape)
...         is_done = tuple(new_position) == self.goal
...         return [(1.0, new_state, -1.0, is_done)]
```

Здесь вычисляется новое положение агента, зная текущее, ход и эффект ветра. При этом гарантируется, что новое положение не выйдет за пределы сетки. И в конце проверяется, достиг ли агент конечной цели.

5. Определим метод `_limit_coordinates`, который предотвращает выход агента за пределы сеточного мира:

```
...     def _limit_coordinates(self, coord):
...         coord[0] = min(coord[0], self.shape[0] - 1)
...         coord[0] = max(coord[0], 0)
...         coord[1] = min(coord[1], self.shape[1] - 1)
...         coord[1] = max(coord[1], 0)
...         return coord
```

6. Еще добавим метод `render`, который отображает сетку и агента:

```
...     def render(self):
...         outfile = sys.stdout
...         for s in range(self.nS):
...             position = np.unravel_index(s, self.shape)
...             if self.s == s:
...                 output = " x "
...             elif position == self.goal:
...                 output = " T "
...             else:
...                 output = " o "
...             if position[1] == 0:
...                 output = output.lstrip()
...             if position[1] == self.shape[1] - 1:
...                 output = output.rstrip()
...                 output += "\n"
...             outfile.write(output)
...         outfile.write("\n")
```

Здесь `X` обозначает текущее положение агента, `T` – конечную ячейку, а `o` – все прочие ячейки.

Теперь выполним несколько взаимодействий с окружающей средой Windy Gridworld.

1. Создадим экземпляр среды Windy Gridworld:

```
>>> env = WindyGridworldEnv()
```

2. Сбросим среду в исходное состояние:

```
>>> env.reset()
>>> env.render()
o o o o o o o o o o
```

```
o o o o o o o o o o
o o o o o o o o o o
x o o o o o o T o o
o o o o o o o o o o
o o o o o o o o o o
o o o o o o o o o o
```

Агент начинает работу в состоянии 30.

3. Сделаем ход вправо:

```
>>> print(env.step(1))
>>> env.render()
(31, -1.0, False, {'prob': 1.0})
o o o o o o o o o o
o o o o o o o o o o
o o o o o o o o o o
o x o o o o o T o o
o o o o o o o o o o
o o o o o o o o o o
o o o o o o o o o o
```

Агент оказывается в состоянии 31 и получает вознаграждение –1.

4. Сделаем два хода вправо:

```
>>> print(env.step(1))
>>> print(env.step(1))
>>> env.render()
(32, -1.0, False, {'prob': 1.0})
(33, -1.0, False, {'prob': 1.0})
o o o o o o o o o o
o o o o o o o o o o
o o o o o o o o o o
o o o x o o o T o o
o o o o o o o o o o
o o o o o o o o o o
o o o o o o o o o o
```

5. И еще один ход вправо:

```
>>> print(env.step(1))
>>> env.render()
(24, -1.0, False, {'prob': 1.0})
o o o o o o o o o o
o o o o o o o o o o
o o o o x o o o o o
o o o o o o o T o o
o o o o o o o o o o
o o o o o o o o o o
o o o o o o o o o o
```

Ветер сдул агента на одну ячейку вверх, так что он оказался в состоянии 24.

Продолжайте, пока не дойдете до цели.

Как это работает

Мы только что разработали окружающую среду, похожую на Cliff Walking. Разница между ними в том, что в среде Windy Gridworld есть еще ветер, дующий вверх. Каждое действие приносит агенту вознаграждение –1. Поэтому требуется добраться до цели как можно раньше. В следующем рецепте мы решим задачу, применив еще один TD-метод управления.

Реализация алгоритма SARSA

Напомним, что Q-обучение – алгоритм TD-обучения с разделенной стратегией. В этом рецепте мы решим МППР методом TD-обучения с единой стратегией – **SARSA** (State-Action-Reward-State-Action).

Как и Q-обучение, SARSA завязан на ценности пар состояние–действие. Обновление Q-функции производится по следующей формуле:

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

Здесь s' – состояние, в которое переходит среда после выбора агентом действия a в состоянии s; r – полученное при этом вознаграждение; α – скорость обучения, γ – коэффициент обесценивания. Напомним, что в Q-обучении для обновления значения Q-функции применяется жадная поведенческая стратегия $\max_{a'} Q(s', a')$. В SARSA для обновления значения Q-функции мы просто выбираем следующее действие a', следуя ε-жадной стратегии. Поэтому SARSA является алгоритмом с единой стратегией.

Как это делается

Найдем оптимальную стратегию взаимодействия со средой Windy Gridworld методом SARSA.

1. Импортируем библиотеку PyTorch и модуль `WindyGridworldEnv` (в предположении, что последний находится в файле `windy_gridworld.py`) и создадим экземпляр окружающей среды Windy Gridworld:

```
>>> import torch
>>> from windy_gridworld import WindyGridworldEnv
>>> env = WindyGridworldEnv()
```

2. Определим ε-жадную поведенческую стратегию:

```
>>> def gen_epsilon_greedy_policy(n_action, epsilon):
...     def policy_function(state, Q):
...         probs = torch.ones(n_action) * epsilon / n_action
...         best_action = torch.argmax(Q[state]).item()
...         probs[best_action] += 1.0 - epsilon
...         action = torch.multinomial(probs, 1).item()
...         return action
...     return policy_function
```

3. Зададим количество эпизодов и инициализируем два списка для хранения длин эпизодов и полученных в них вознаграждений:

```
>>> n_episode = 500
>>> length_episode = [0] * n_episode
>>> total_reward_episode = [0] * n_episode
```

4. Определим функцию, реализующую алгоритм SARSA:

```
>>> from collections import defaultdict
>>> def sarsa(env, gamma, n_episode, alpha):
...     """
...     Строит оптимальную стратегию методом SARSA с единой стратегией
...     @param env: имя окружающей среды OpenAI Gym
...     @param gamma: коэффициент обесценивания
...     @param n_episode: количество эпизодов
...     @return: оптимальные Q-функция и стратегия
...     """
...     n_action = env.action_space.n
...     Q = defaultdict(lambda: torch.zeros(n_action))
...     for episode in range(n_episode):
...         state = env.reset()
...         is_done = False
...         action = epsilon_greedy_policy(state, Q)
...         while not is_done:
...             next_state, reward, is_done, info = env.step(action)
...             next_action = epsilon_greedy_policy(next_state, Q)
...             td_delta = reward + gamma * Q[next_state][next_action]
                                - Q[state][action]
...             Q[state][action] += alpha * td_delta
...             length_episode[episode] += 1
...             total_reward_episode[episode] += reward
...             if is_done:
...                 break
...             state = next_state
...             action = next_action
...     policy = {}
...     for state, actions in Q.items():
...         policy[state] = torch.argmax(actions).item()
...     return Q, policy
```

5. Зададим коэффициент обесценивания 1, скорость обучения 0.4 и ε = 0.1:

```
>>> gamma = 1
>>> alpha = 0.4
>>> epsilon = 0.1
```

6. Создадим экземпляр ε-жадной стратегии:

```
>>> epsilon_greedy_policy =
gen_epsilon_greedy_policy(env.action_space.n, epsilon)
```

7. Выполним алгоритм SARSA с параметрами, определенными выше, и распечатаем оптимальную стратегию:

```
>>> optimal_Q, optimal_policy = sarsa(env, gamma, n_episode, alpha)
>>> print('Оптимальная стратегия:\n', optimal_policy)
Оптимальная стратегия:
{30: 2, 31: 1, 32: 1, 40: 1, 50: 2, 60: 1, 61: 1, 51: 1, 41: 1,
42: 1, 20: 1, 21: 1, 62: 1, 63: 2, 52: 1, 53: 1, 43: 1, 22: 1, 11:
1, 10: 1, 0: 1, 33: 1, 23: 1, 12: 1, 13: 1, 2: 1, 1: 1, 3: 1, 24:
1, 4: 1, 5: 1, 6: 1, 14: 1, 7: 1, 8: 1, 9: 2, 19: 2, 18: 2, 29: 2,
28: 1, 17: 2, 39: 2, 38: 1, 27: 0, 49: 3, 48: 3, 37: 3, 34: 1, 59:
2, 58: 3, 47: 2, 26: 1, 44: 1, 15: 1, 69: 3, 68: 1, 57: 2, 36: 1,
25: 1, 54: 2, 16: 1, 35: 1, 45: 1}
```

Как это работает

На шаге 4 функция `sarsa` выполняет следующие действия:

- инициализирует нулями таблицу значений Q-функции;
- в каждом эпизоде позволяет агенту следовать ε-жадной стратегии при выборе действия. И на каждом шаге обновляет Q-функцию по формуле $Q(s, a) = Q(s, a) + \alpha(r + \gamma Q(s', a') - Q(s, a))$, в которой a' выбирается, следуя ε-жадной стратегии. Затем выполняет новое действие a' в новом состоянии s';
- прогоняет `n_episode` эпизодов;
- строит оптимальную стратегию на основе оптимальной Q-функции.

Как видим, алгоритм SARSA оптимизирует Q-функцию, выбирая действие в соответствии с той же ε-жадной стратегией. Это очень похоже на метод управления МК с единой стратегией. Разница в том, что Q-функция обновляется небольшими приращениями на каждом шаге, а не в конце всего эпизода. Это считается преимуществом в окружающих средах с длинными эпизодами, когда ждать завершения эпизода неэффективно. На каждом шаге SARSA мы получаем новую информацию о среде и сразу же используем ее для обновления ценностей. В нашем примере для нахождения оптимальной стратегии понадобилось всего 500 эпизодов.

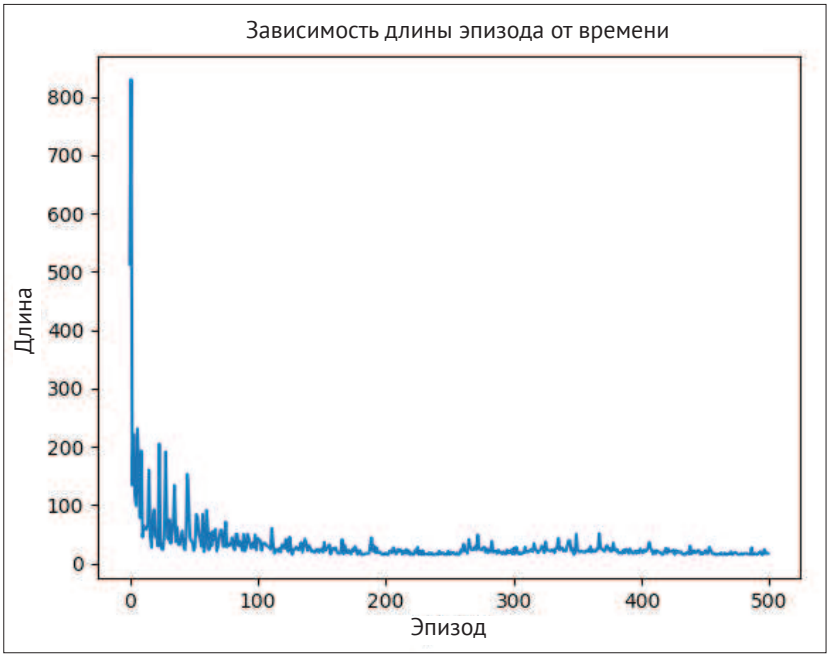
Это еще не все

На самом деле для получения оптимальной стратегии хватило примерно 200 эпизодов. Чтобы убедиться в этом, построим графики длин и полных вознаграждений для каждого эпизода.

1. Нарисуем график зависимости длины эпизода от времени:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(length_episode)
>>> plt.title('Зависимость длины эпизода от времени')
>>> plt.xlabel('Эпизод')
>>> plt.ylabel('Длина')
>>> plt.show()
```

Получается такой график:

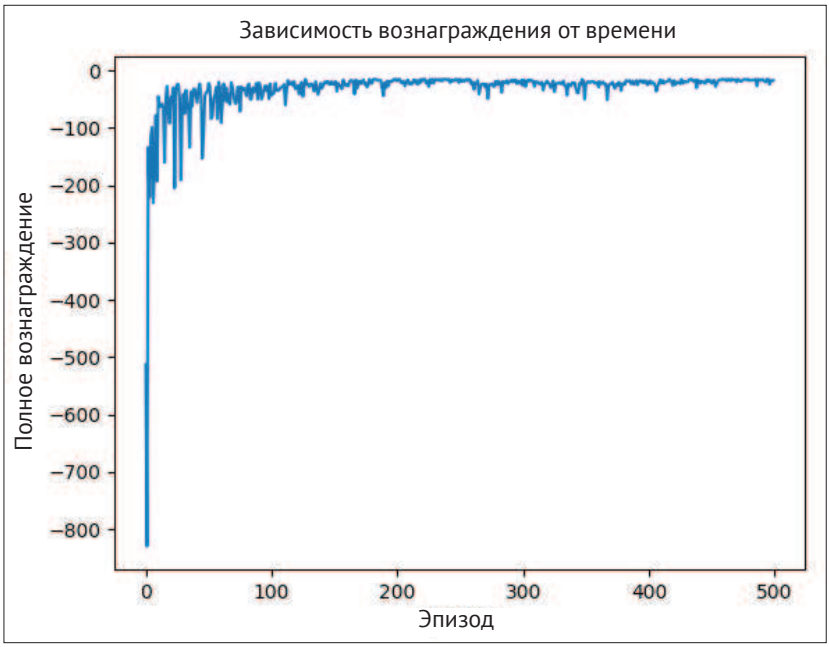


Как видим, длина эпизода стабилизируется после 200 эпизодов. А небольшие флуктуации связаны со случайным исследованием в ε-жадной стратегии.

2. Нарисуем график зависимости вознаграждения от времени:

```
>>> plt.plot(total_reward_episode)
>>> plt.title('Зависимость вознаграждения от времени')
>>> plt.xlabel('Эпизод')
>>> plt.ylabel('Полное вознаграждение')
>>> plt.show()
```

Результат показан на рисунке ниже.

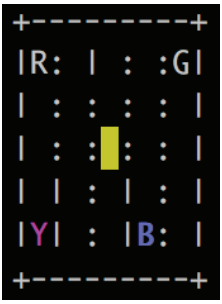


Чем меньше значение ε, тем слабее флуктуации, являющиеся результатом случайного исследования в ε-жадной стратегии.

В следующих двух рецептах мы воспользуемся обоими изученными методами для взаимодействия с окружающей средой посложнее, где количество состояний и действий больше. Начнем с алгоритма Q-обучения.

Решение задачи о такси методом Q-обучения

Задача о такси (https://gym.openai.com/envs/Taxi-v2/) – еще одна популярная задача на сетке. Агент существует в сеточном мире размера 5×5 и играет роль водителя такси, который должен посадить пассажира в какой-то ячейке и высадить в месте назначения. Рассмотрим пример:



Цвета интерпретируются следующим образом.

- **Желтый:** начальное положение такси. Оно выбирается случайным образом в каждом эпизоде.
- **Синий:** положение пассажира. Случайным образом выбирается в каждом эпизоде.
- **Фиолетовый:** место назначения. Случайным образом выбирается в каждом эпизоде.
- **Зеленый:** положение такси, везущего пассажира.

Буквами R, Y, B, G обозначены те ячейки, в которых разрешены посадка и высадка. Одна из них совпадает с местом нахождения пассажира, другая – с местом назначения.

Такси может совершать шесть детерминированных действий:

- **0:** двигаться на юг;
- **1:** двигаться на север;
- **2:** двигаться на восток;
- **3:** двигаться на запад;
- **4:** посадить пассажира;
- **5:** высадить пассажира.

Вертикальные черточки между ячейками означают, что в этом направлении движение запрещено.

Вознаграждение на каждом шаге равно –1 со следующими исключениями:

- **+20:** пассажир доставлен в место назначения. На этом эпизод заканчивается;
- **–10:** попытка посадить или высадить пассажира в неразрешенном месте (не совпадающем ни с одной из ячеек R, Y, B, G).

Заметим также, что пространство наблюдений гораздо больше, чем 25 (5*5), поскольку нужно учитывать место посадки и высадки пассажира, а также занято такси или свободно. Следовательно, размер пространства наблюдений равен 25 * 5 (4 возможных места посадки пассажира плюс признак того, что он уже сидит в машине) * 4 (места высадки) = 500.

Подготовка

Прежде всего найдем имя окружающей среды Taxi в таблице по адресу https://github.com/openai/gym/wiki/Table-of-environments. Она называется Taxi-v2, и попутно мы узнаем, что наблюдения представлены числами от 0 до 499, а возможных действий четыре (вверх = 0, вправо = 1, вниз = 2, влево = 3).

Как это делается

Для имитации окружающей среды Taxi выполним следующим действия.

1. Импортируем библиотеку Gym и создадим экземпляр среды Taxi:

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

```
>>> n_state = env.observation_space.n
>>> print(n_state)
 500
>>> n_action = env.action_space.n
>>> print(n_action)
 6
```

2. Сбросим среду в исходное состояние:

```
>>> env.reset()
 262
```

3. Нарисуем среду:

```
>>> env.render()
```

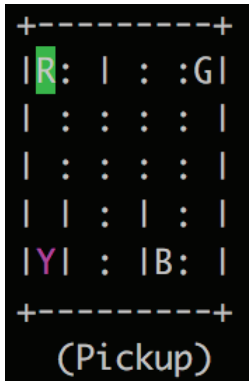
Появится уже знакомая матрица размера 5×5.

Пассажир находится в ячейке R, а место назначения – в ячейке Y. На вашем компьютере картинка может быть другой, потому что начальное состояние генерируется случайным образом.

4. Теперь отправимся за пассажиром, который находится от нас в трех ячейках к западу и двух к северу (сделайте поправку на то, что видите у себя на экране), и посадим его в машину. Затем снова нарисуем окружающую среду:

```
>>> print(env.step(3))
(242, -1, False, {'prob': 1.0})
>>> print(env.step(3))
(222, -1, False, {'prob': 1.0})
>>> print(env.step(3))
(202, -1, False, {'prob': 1.0})
>>> print(env.step(1))
(102, -1, False, {'prob': 1.0})
>>> print(env.step(1))
(2, -1, False, {'prob': 1.0})
>>> print(env.step(4))
(18, -1, False, {'prob': 1.0})
Render the environment:
>>> env.render()
```

5. Матрица на экране обновится следующим образом (у вас может быть по-другому):



   Такси перекрасилось в зеленый цвет.

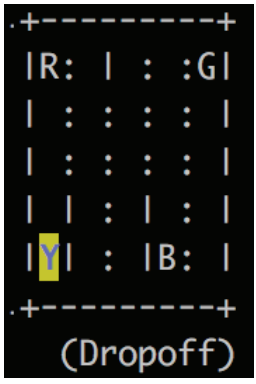
6. Теперь поедем к месту назначения – четыре ячейки на юг – и высадим пассажира:

```
>>> print(env.step(0))
(118, -1, False, {'prob': 1.0})
>>> print(env.step(0))
(218, -1, False, {'prob': 1.0})
>>> print(env.step(0))
(318, -1, False, {'prob': 1.0})
>>> print(env.step(0))
(418, -1, False, {'prob': 1.0})
>>> print(env.step(5))
(410, 20, True, {'prob': 1.0})
```

   В итоге получаем вознаграждение +20, и эпизод заканчивается:

```
>>> env.render()
```

   Обновленная матрица выглядит так:



Теперь применим к задаче о такси метод Q-обучения.

1. Импортируем библиотеку PyTorch:

```
>>> import torch
```

2. Повторно используем функцию ε-жадной стратегии `gen_epsilon_greedy_policy` из рецепта «Реализация алгоритма Q-обучения».

3. Зададим количество эпизодов и инициализируем два списка для хранения длин эпизодов и полученных в них вознаграждений:

```
>>> n_episode = 1000
>>> length_episode = [0] * n_episode
>>> total_reward_episode = [0] * n_episode
```

4. В качестве функции, выполняющей Q-обучение, используем функцию `q_learning` из рецепта «Реализация алгоритма Q-обучения».

5. Зададим остальные параметры: коэффициент обесценивания, скорость обучения и ε, а затем создадим экземпляр ε-жадной стратегии:

```
>>> gamma = 1
>>> alpha = 0.4
>>> epsilon = 0.1
>>> epsilon_greedy_policy =
gen_epsilon_greedy_policy(env.action_space.n, epsilon)
```

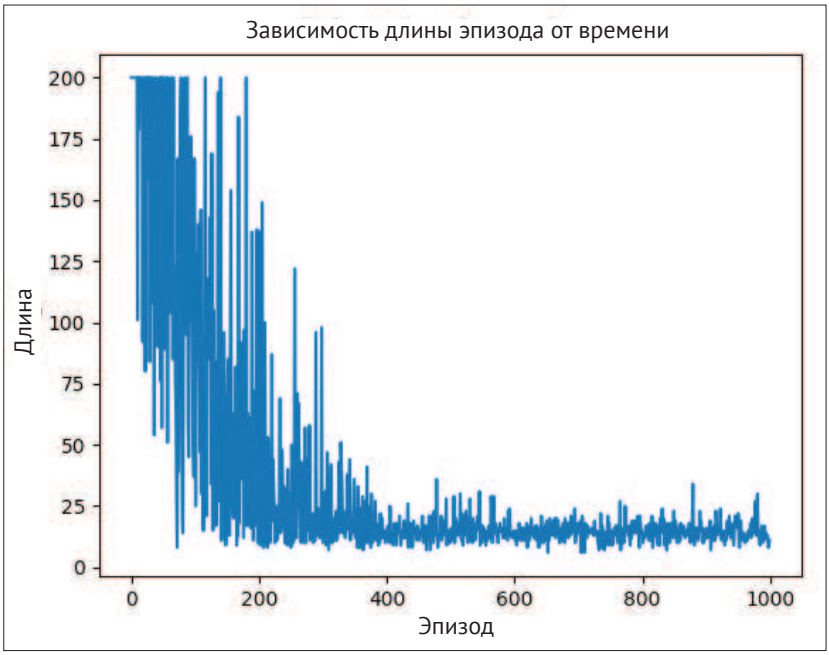
6. И наконец, выполним Q-обучение и найдем оптимальную стратегию в задаче о такси:

```
>>> optimal_Q, optimal_policy = q_learning(env, gamma, n_episode, alpha)
```

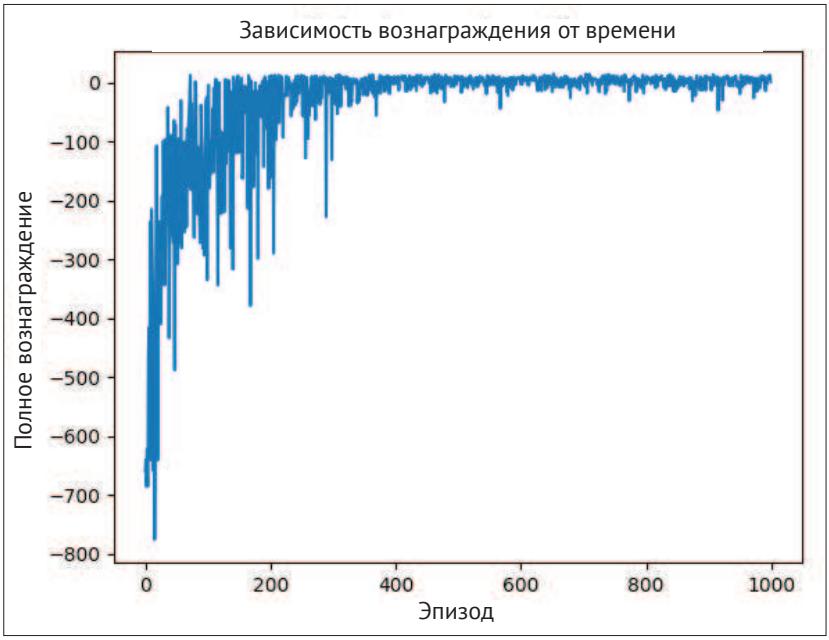
Как это работает

В этом рецепте мы решили задачу о такси методом Q-обучения с разделенной стратегией.

После шага 6 можно построить графики длины эпизода и полученного вознаграждения и убедиться, что модель сходится. График зависимости длины эпизода от времени выглядит следующим образом:



А график зависимости вознаграждения от времени – так:



Как видим, стабилизация начинается после 400 эпизодов.

Taxi – довольно сложная сеточная окружающая среда с 500 дискретными состояниями и 6 возможными действиями. Алгоритм Q-обучения оптимизирует Q-функцию на каждом шаге эпизода, обучаясь на опыте, генерируемом жадной стратегией. Мы получаем информацию об окружающей среде в процессе обучения и сразу же используем ее для обновления ценностей, следуя ε-жадной стратегии.

Решение задачи о такси методом SARSA

В этом рецепте мы решим задачу о такси методом SARSA и настроим гиперпараметры, применив алгоритм поиска на сетке.

Начнем с набора гиперпараметров SARSA, подразумеваемого по умолчанию. Они выбраны из интуитивных сообщений и на основе предшествующего опыта. Впоследствии мы подберем наилучшие значения.

Как это делается

Применим алгоритм SARSA к взаимодействию со средой Taxi.

1. Импортируем библиотеки PyTorch и gym и создадим экземпляр окружающей среды Taxi:

```
>>> import torch
>>> import gym
>>> env = gym.make('Taxi-v2')
```

2. Повторно используем функцию ε-жадной поведенческой стратегии `gen_epsilon_greedy_policy` из рецепта «Реализация алгоритма SARSA».

3. Зададим количество эпизодов и инициализируем два списка для хранения длин эпизодов и полученных в них вознаграждений:

```
>>> n_episode = 1000
>>> length_episode = [0] * n_episode
>>> total_reward_episode = [0] * n_episode
```

4. В качестве функции, выполняющей SARSA, используем функцию `sarsa` из рецепта «Реализация алгоритма SARSA».

5. Зададим коэффициент обесценивания 1, скорость обучения 0.4 и ε = 0.1:

```
>>> gamma = 1
>>> alpha = 0.4
>>> epsilon = 0.01
```

6. Создадим экземпляр ε-жадной стратегии:

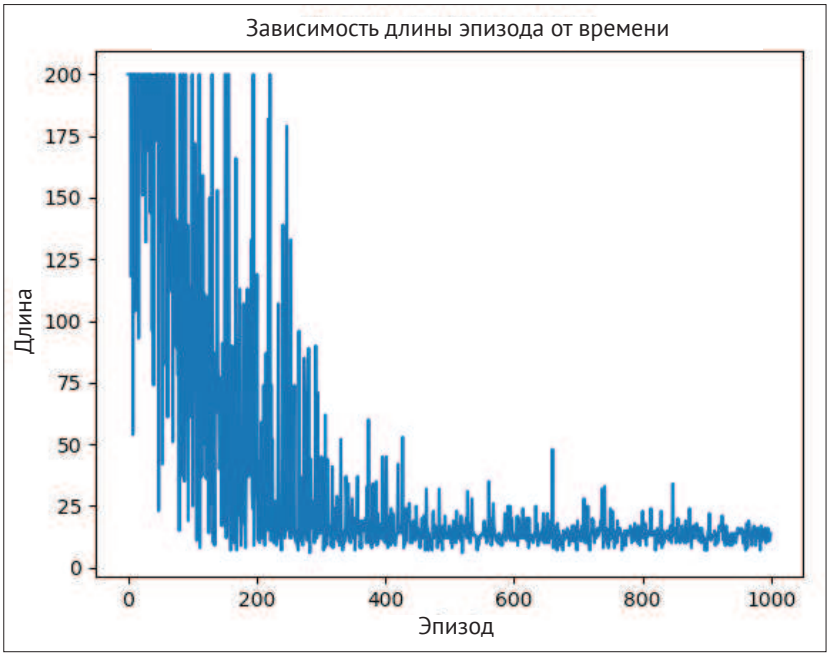
```
>>> epsilon_greedy_policy =
gen_epsilon_greedy_policy(env.action_space.n, epsilon)
```

7. И наконец, выполним алгоритм SARSA с заданными выше параметрами:

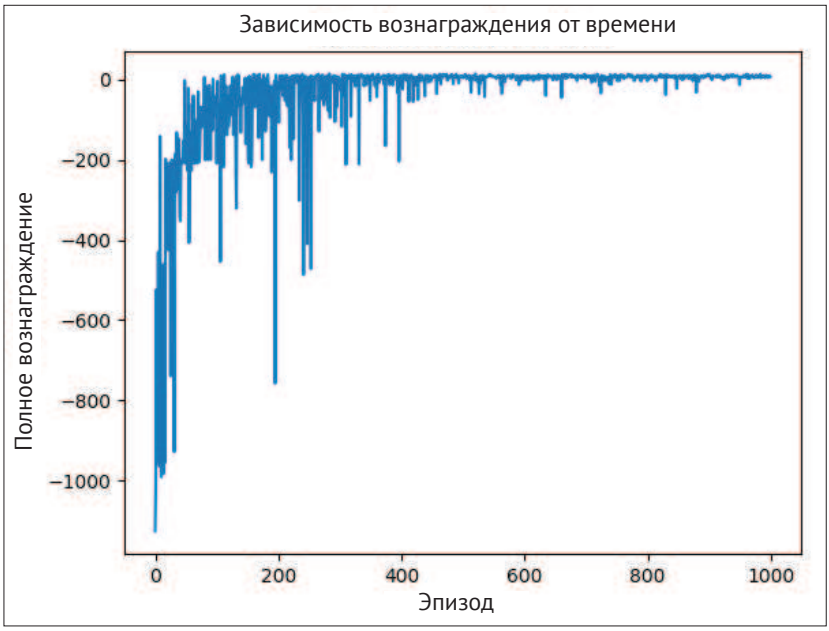
```
>>> optimal_Q, optimal_policy = sarsa(env, gamma, n_episode, alpha)
```

Как это работает

После шага 7 можно построить графики длины эпизода и полученного вознаграждения и убедиться, что модель сходится. График зависимости длины эпизода от времени выглядит следующим образом:



А график зависимости вознаграждения от времени – так:



Описанный алгоритм SARSA работает неплохо, но есть возможность его улучшить. Позже мы воспользуемся поиском на сетке, чтобы найти наилучший набор гиперпараметров.

Это еще не все

Поиск на сетке – это программный способ найти наилучший набор гиперпараметров в обучении с подкреплением. Качество набора измеряется по следующим показателям:

- среднее полное вознаграждение в нескольких первых эпизодах: мы хотим, чтобы вознаграждение стало большим как можно быстрее;
- средняя длина нескольких первых эпизодов: мы хотим, чтобы такси довозило пассажира до места назначения как можно быстрее;
- среднее вознаграждение на одном временном шаге в нескольких первых эпизодах: мы хотим, чтобы это вознаграждение достигло максимума как можно быстрее.

Реализуем заявленный алгоритм.

1. Далее будем использовать три потенциальных значения альфа, [0.4, 0.5, 0.6], и три потенциальных значения эпсилон, [0.1, 0.03, 0.01]. И будем рассматривать только первые 500 эпизодов.

```
>>> alpha_options = [0.4, 0.5, 0.6]
>>> epsilon_options = [0.1, 0.03, 0.01]
>>> n_episode = 500
```

2. Выполним поиск на сетке, обучив алгоритм SARSA с каждым набором гиперпараметров и оценив его качество.

```
>>> for alpha in alpha_options:
...     for epsilon in epsilon_options:
...     length_episode = [0] * n_episode
...     total_reward_episode = [0] * n_episode
...     sarsa(env, gamma, n_episode, alpha)
...     reward_per_step = [reward/float(step) for reward, step in zip(
                total_reward_episode, length_episode)]
...     print('alpha: {}, epsilon: {}'.format(alpha, epsilon))
...     print('Среднее вознаграждение в {} эпизодах: {}'.format(
                n_episode, sum(total_reward_episode) / n_episode))
...     print('Средняя длина {} эпизодов: {}'.format(
                n_episode, sum(length_episode) / n_episode))
...     print('Среднее вознаграждение на одном шаге в {} эпизодах:{}\n'.format(
                n_episode, sum(reward_per_step) / n_episode))
```

Выполнив этот код, мы получим такие результаты:

```
alpha: 0.4, epsilon: 0.1
Среднее вознаграждение в 500 эпизодах: -75.442
Средняя длина 500 эпизодов: 57.682
Среднее вознаграждение на одном шаге в 500 эпизодах: -0.32510755063660324
alpha: 0.4, epsilon: 0.03
Среднее вознаграждение в 500 эпизодах: -73.378
Средняя длина 500 эпизодов: 56.53
Среднее вознаграждение на одном шаге в 500 эпизодах: -0.2761201410280632
alpha: 0.4, epsilon: 0.01
Среднее вознаграждение в 500 эпизодах: -78.722
Средняя длина 500 эпизодов: 59.366
Среднее вознаграждение на одном шаге в 500 эпизодах: -0.3561815084186654
alpha: 0.5, epsilon: 0.1
Среднее вознаграждение в 500 эпизодах: -72.026
Средняя длина 500 эпизодов: 55.592
Среднее вознаграждение на одном шаге в 500 эпизодах: -0.25355404831497264
alpha: 0.5, epsilon: 0.03
Среднее вознаграждение в 500 эпизодах: -67.562
Средняя длина 500 эпизодов: 52.706
Среднее вознаграждение на одном шаге в 500 эпизодах: -0.20602525679639022
alpha: 0.5, epsilon: 0.01
Среднее вознаграждение в 500 эпизодах: -75.252
Средняя длина 500 эпизодов: 56.73
Среднее вознаграждение на одном шаге в 500 эпизодах: -0.2588407558703358
alpha: 0.6, epsilon: 0.1
Среднее вознаграждение в 500 эпизодах: -62.568
Средняя длина 500 эпизодов: 49.488
Среднее вознаграждение на одном шаге в 500 эпизодах: -0.1700284221229244
alpha: 0.6, epsilon: 0.03
Среднее вознаграждение в 500 эпизодах: -68.56
Средняя длина 500 эпизодов: 52.804
Среднее вознаграждение на одном шаге в 500 эпизодах: -0.24794191768600077
alpha: 0.6, epsilon: 0.01
```

```
Среднее вознаграждение в 500 эпизодах: -63.468
Средняя длина 500 эпизодов: 49.752
Среднее вознаграждение на одном шаге в 500 эпизодах: -0.14350124172091722
```

Как видим, в данном случае наилучшим стал набор гиперпараметров alpha: 0.6, epsilon: 0.01, при котором достигнуты наибольшее вознаграждение на одном шаге, наибольшее среднее вознаграждение и наименьшая длина эпизода.

Реализация алгоритма двойного Q-обучения

И напоследок мы в этой главе реализуем алгоритм двойного Q-обучения.

Q-обучение – эффективный и популярный TD-алгоритм обучения с подкреплением. Но иногда он работает плохо, главным образом из-за жадной компоненты $\max_{a'}Q(s', a')$. Он может завышать оценки ценности действий, что приводит к неудовлетворительным результатам. Алгоритм двойного Q-обучения призван преодолеть этот недостаток посредством использования двух Q-функций, которые мы обозначим *Q1* и *Q2*. На каждом шаге обновляется одна случайно выбранная Q-функция. Если выбрана *Q1*, то обновление производится по формуле:

$$a^* = \text{argmax}_a Q1(s', a);$$
$$Q1(s, a) = Q1(s, a) + \alpha(r + \gamma Q2(s', a^*) - Q1(s, a)),$$

а если *Q2*, то по формуле:

$$a^* = \text{argmax}_a Q2(s', a);$$
$$Q2(s, a) = Q2(s, a) + \alpha(r + \gamma Q1(s', a^*) - Q2(s, a)).$$

Это значит, что в обновлении каждой Q-функции участвует другая, при этом применяется жадный поиск, что уменьшает степень завышения оценки ценности действий по сравнению с одной Q-функцией.

Как это делается

Реализуем алгоритм двойного Q-обучения для окружающей среды Taxi.

1. Импортируем необходимые библиотеки и создадим экземпляр среды Taxi:

```
>>> import torch
>>> import gym
>>> env = gym.make('Taxi-v2')
```

2. Повторно используем функцию ε-жадной стратегии `gen_epsilon_greedy_policy` из рецепта «Реализация алгоритма Q-обучения».

3. Зададим количество эпизодов и инициализируем два списка для хранения длин эпизодов и полученных в них вознаграждений:

```
>>> n_episode = 3000
>>> length_episode = [0] * n_episode
>>> total_reward_episode = [0] * n_episode
```

Мы имитируем 3000 эпизодов, поскольку для сходимости алгоритма двойного Q-обучения эпизодов нужно больше.

4. Определим функцию, которая выполняет двойное Q-обучение:

```
>>> def double_q_learning(env, gamma, n_episode, alpha):
...     """
...     Строит оптимальную стратегию методом двойного Q-обучения с
...     разделенной стратегией
...     @param env: имя окружающей среды OpenAI Gym
...     @param gamma: коэффициент обесценивания
...     @param n_episode: количество эпизодов
...     @return: оптимальные Q-функция и стратегия
...     """
...     n_action = env.action_space.n
...     n_state = env.observation_space.n
...     Q1 = torch.zeros(n_state, n_action)
...     Q2 = torch.zeros(n_state, n_action)
...     for episode in range(n_episode):
...         state = env.reset()
...         is_done = False
...         while not is_done:
...             action = epsilon_greedy_policy(state, Q1 + Q2)
...             next_state, reward, is_done, info = env.step(action)
...             if (torch.rand(1).item() < 0.5):
...                 best_next_action = torch.argmax(Q1[next_state])
...                 td_delta = reward + gamma * Q2[next_state][best_next_action]
                                  - Q1[state][action]
...                 Q1[state][action] += alpha * td_delta
...             else:
...                 best_next_action = torch.argmax(Q2[next_state])
...                 td_delta = reward + gamma * Q1[next_state][best_next_action]
                                  - Q2[state][action]
...                 Q2[state][action] += alpha * td_delta
...             length_episode[episode] += 1
...             total_reward_episode[episode] += reward
...             if is_done:
...                 break
...             state = next_state
...     policy = {}
...     Q = Q1 + Q2
...     for state in range(n_state):
...         policy[state] = torch.argmax(Q[state]).item()
...     return Q, policy
```

5. Зададим остальные параметры: коэффициент обесценивания, скорость обучения и ε, а затем создадим экземпляр ε-жадной стратегии:

```
>>> gamma = 1
>>> alpha = 0.4
>>> epsilon = 0.1
>>> epsilon_greedy_policy =
gen_epsilon_greedy_policy(env.action_space.n, epsilon)
```

6. И наконец, выполним двойное Q-обучение и найдем оптимальную стратегию в задаче о такси:

```
>>> optimal_Q, optimal_policy = double_q_learning(env, gamma,
n_episode, alpha)
```

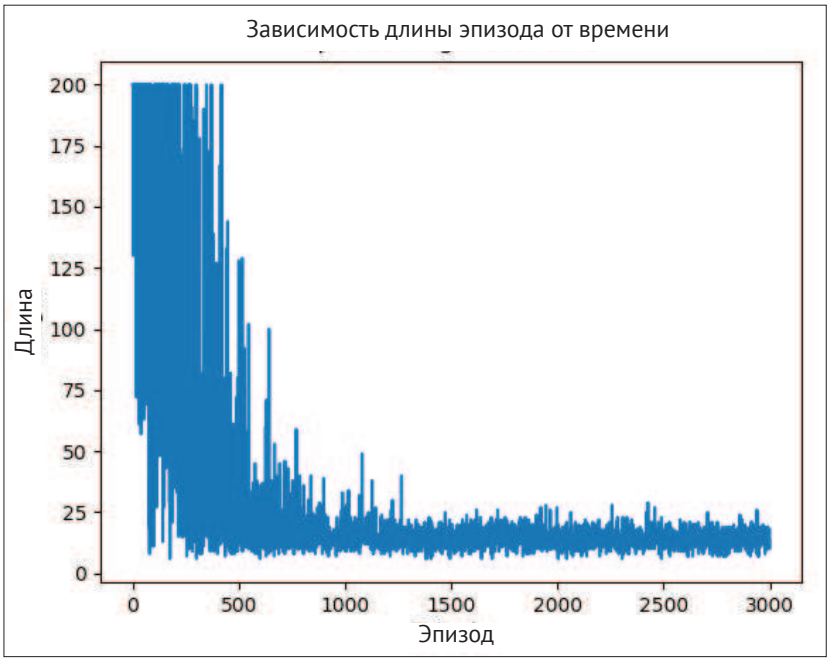
Как это работает

В этом рецепте мы решили задачу о такси методом двойного Q-обучения.

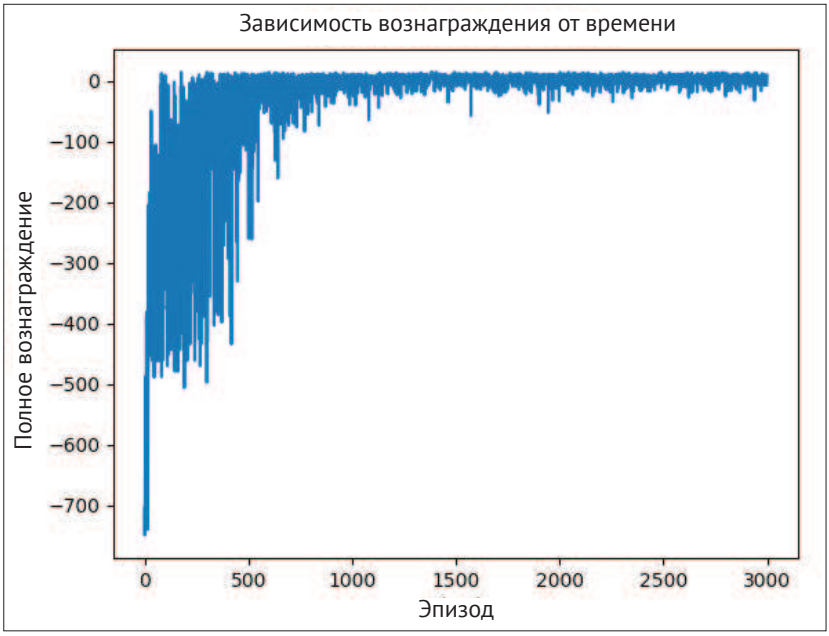
Функция на шаге 4 выполняет следующие действия:

- инициализирует нулями две таблицы значений Q-функций;
- на каждом шаге эпизода случайным образом выбирает для обновления одну Q-функцию и дает агенту возможность, следуя ε-жадной стратегии, выбрать, какое действие предпринять, и обновить выбранную Q-функцию с помощью другой;
- выполняет `n_episode` эпизодов;
- строит оптимальную стратегию, суммируя (или усредняя) обе Q-функции.

После шага 6 можно построить графики длины эпизода и полученного вознаграждения и убедиться, что модель сходится. График зависимости длины эпизода от времени выглядит следующим образом:



А график зависимости вознаграждения от времени – так:



Алгоритм двойного Q-обучения преодолевает потенциальный недостаток простого Q-обучения в сложных окружающих средах. Он случайным образом чередует и обновляет две Q-функции, что предотвращает завышение оценок ценностей действий, свойственное одной Q-функции. Однако возможно занижение оценки, поскольку на разных шагах обновляются разные Q-функции. Поэтому для нахождения оптимальных ценностей действий требуется больше эпизодов.

См. также

Теоретическое обоснование двойного Q-обучения см. в оригинальной статье Хадо ван Хассельта, опубликованной в журнале «Advances in Neural Information Processing Systems» 23 (NIPS 2010), 2613-2621, 2010 (https://papers.nips.cc/paper/3964-double-q-learning).

Глава 5

Решение задачи о многоруком бандите

Алгоритмы многоруких бандитов являются, пожалуй, одними из самых популярных в обучении с подкреплением. Мы начнем эту главу с того, что создадим многорукого бандита и поэкспериментируем со случайными стратегиями. Далее нас будут интересовать четыре стратегии решения задачи о многоруком бандите: ε-жадная, исследование с функцией softmax, верхняя доверительная граница и выборка Томпсона. На их примере мы увидим различные подходы к дилемме исследования–использования. Мы также рассмотрим задачу о рекламе в интернете и покажем, как ее решить с помощью алгоритма многорукого бандита. Наконец, мы решим задачу о контекстной рекламе с помощью алгоритма контекстуальных бандитов, который позволяет принимать более обоснованные решения при оптимизации показа объявлений.

В этой главе приводятся следующие рецепты:

- создание окружающей среды с многоруким бандитом;
- решение задачи о многоруком бандите с помощью ε-жадной стратегии;
- решение задачи о многоруком бандите с помощью softmax-исследования;
- решение задачи о многоруком бандите с помощью алгоритма верхней доверительной границы;
- решение задачи о рекламе в интернете с помощью алгоритма многорукого бандита;
- решение задачи о многоруком бандите с помощью выборки Томпсона;
- решение задачи о рекламе в интернете с помощью контекстуальных бандитов.

Создание окружающей среды с многоруким бандитом

Задача о многоруком бандите – одна из самых простых задач обучения с подкреплением. Для ее описания проще всего представить игральный автомат с несколькими рычагами (руками), причем все они выдают разные выигрыши с разными вероятностями. Наша цель – найти наилучший рычаг, ко-

торый дает максимальный доход (и впоследствии только его и использовать). Начнем с простой постановки, когда величина и вероятность выигрыша для каждого рычага фиксированы. Сначала создадим окружающую среду и решим ее с помощью алгоритма со случайной стратегией.

Как это делается

Для создания окружающей среды с многоруким бандитом выполним следующий код.

```
>>> import torch
>>> class BanditEnv():
...     """
...     Окружающая среда с многоруким бандитом
...     payout_list:
...         Список вероятностей выигрышей отдельных рычагов
...     reward_list:
...         Список величин выигрышей
...     """
...     def __init__(self, payout_list, reward_list):
...         self.payout_list = payout_list
...         self.reward_list = reward_list
...
...     def step(self, action):
...         if torch.rand(1).item() < self.payout_list[action]:
...             return self.reward_list[action]
...         return 0
```

Метод `step` выполняет действие и возвращает величину вознаграждения в случае выигрыша, в противном случае 0.

Теперь решим задачу о многоруком бандите, применяя случайную стратегию.

1. Определим вероятности и величины выигрыша для трехрукого бандита и создадим экземпляр окружающей среды:

   ```
   >>> bandit_payout = [0.1, 0.15, 0.3]
   >>> bandit_reward = [4, 3, 1]
   >>> bandit_env = BanditEnv(bandit_payout, bandit_reward)
   ```

   Например, выбор рычага 0 приносит вознаграждение 4 с вероятностью 10 %.

2. Зададим количество эпизодов и определим списки для хранения следующих данных: полное вознаграждение, полученное при выборе каждого рычага, сколько раз выбирался каждый рычаг и среднее вознаграждение для каждого рычага.

   ```
   >>> n_episode = 100000
   >>> n_action = len(bandit_payout)
   >>> action_count = [0 for _ in range(n_action)]
   >>> action_total_reward = [0 for _ in range(n_action)]
   >>> action_avg_reward = [[] for action in range(n_action)]
   ```

3. Определим стратегию, которая выбирает рычаг случайным образом:

```
>>> def random_policy():
...     action = torch.multinomial(torch.ones(n_action), 1).item()
...     return action
```

4. Прогоним 100 000 эпизодов. После каждого эпизода будем обновлять статистику рычагов:

```
>>> for episode in range(n_episode):
...     action = random_policy()
...     reward = bandit_env.step(action)
...     action_count[action] += 1
...     action_total_reward[action] += reward
...     for a in range(n_action):
...         if action_count[a]:
...             action_avg_reward[a].append(
                    action_total_reward[a] / action_count[a])
...         else:
...             action_avg_reward[a].append(0)
```

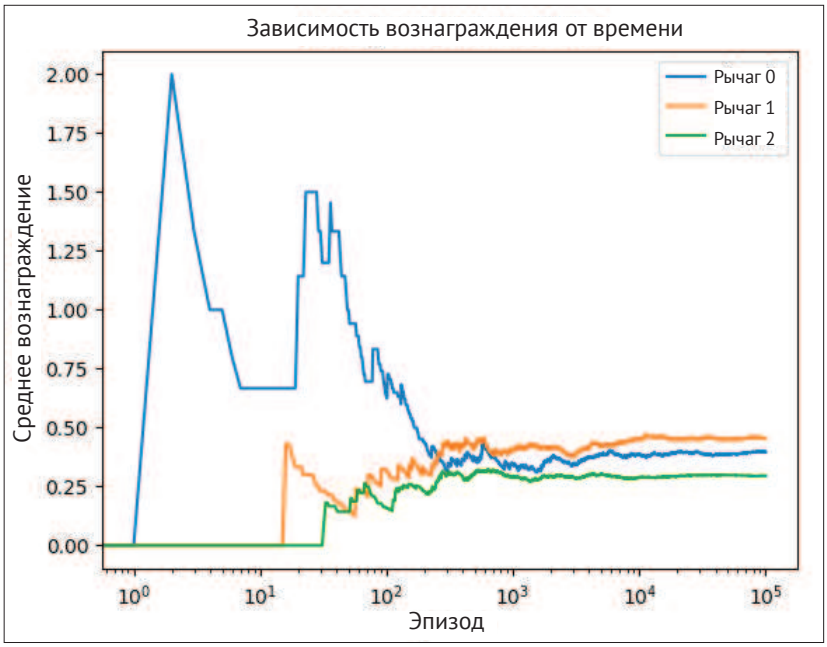
5. По завершении построим график зависимости среднего вознаграждения от времени:

```
>>> import matplotlib.pyplot as plt
>>> for action in range(n_action):
...     plt.plot(action_avg_reward[action])
>>> plt.legend(['Рычаг {}'.format(action) for action in range(n_action)])
>>> plt.title('Зависимость вознаграждения от времени')
>>> plt.xscale('log')
>>> plt.xlabel('Эпизод')
>>> plt.ylabel('Среднее вознаграждение')
>>> plt.show()
```

Как это работает

В только что рассмотренном примере есть игральный автомат с тремя рычагами. Каждый рычаг выдает выигрыш (вознаграждение) разной величины с разной вероятностью. В каждом эпизоде мы случайным образом выбираем, за какой рычаг потянуть (какое действие выполнить), и получаем выигрыш с некоторой вероятностью.

После выполнения кода на шаге 5 будет построен такой график:



Рычаг 1 дает наибольшее среднее вознаграждение. И, как видим, вознаграждение стабилизируется после примерно 10 000 эпизодов.

Это наивное решение, поскольку мы только и делаем, что исследуем рычаги. Более осмысленные стратегии мы разработаем в следующих рецептах.

Решение задачи о многоруком бандите с помощью ε-жадной стратегии

Лучших результатов можно достичь, если сочетать исследование с использованием. В этом рецепте мы воспользуемся хорошо известной ε-жадной стратегией.

ε-жадная стратегия решения задачи о многоруком бандите большую часть времени использует лучшее из найденных действий, но иногда исследует другие действия. Точнее, вероятность исследования равна ε, а вероятность использования – 1 – ε, где ε – число от 0 до 1.

- **Эпсилон:** каждое действие выбирается с вероятностью

  $\pi(s, a) = \varepsilon/|A|$,

  где |A| – количество возможных действий.
- **Жадная:** предпочтение отдается действию с максимальной ценностью пары состояние–действие, а вероятность его выбора увеличивается на $1 - \varepsilon$:

  $\pi(s, a) = 1 - \varepsilon + \varepsilon/|A|$.

Как это делается

Для решения задачи о многоруком бандите выполняются следующие действия.

1. Импортируем библиотеку PyTorch и окружающую среду, разработанную в предыдущем рецепте (предполагается, что код класса `BanditEnv` находится в файле `multi_armed_bandit.py`):

```
>>> import torch
>>> from multi_armed_bandit import BanditEnv
```

2. Определим вероятности и величины выигрыша для трехрукого бандита и создадим экземпляр окружающей среды:

```
>>> bandit_payout = [0.1, 0.15, 0.3]
>>> bandit_reward = [4, 3, 1]
>>> bandit_env = BanditEnv(bandit_payout, bandit_reward)
```

3. Зададим количество эпизодов и определим списки для хранения следующих данных: полное вознаграждение, полученное при выборе каждого рычага, сколько раз выбирался каждый рычаг и среднее вознаграждение для каждого рычага.

```
>>> n_episode = 100000
>>> n_action = len(bandit_payout)
>>> action_count = [0 for _ in range(n_action)]
>>> action_total_reward = [0 for _ in range(n_action)]
>>> action_avg_reward = [[] for action in range(n_action)]
```

4. Определим функцию ε-жадной стратегии, зададим значение ε и создадим экземпляр стратегии:

```
>>> def gen_epsilon_greedy_policy(n_action, epsilon):
...     def policy_function(Q):
...         probs = torch.ones(n_action) * epsilon / n_action
...         best_action = torch.argmax(Q).item()
...         probs[best_action] += 1.0 - epsilon
```

```
...         action = torch.multinomial(probs, 1).item()
...         return action
...     return policy_function
>>> epsilon = 0.2
>>> epsilon_greedy_policy = gen_epsilon_greedy_policy(n_action, epsilon)
```

5. Инициализируем Q-функцию, которая будет возвращать среднее вознаграждение для каждого рычага:

```
>>> Q = torch.zeros(n_action)
```

   Эту Q-функцию мы будем обновлять по ходу обучения.

6. Прогоним 100 000 эпизодов. После каждого эпизода будем обновлять статистику рычагов:

```
>>> for episode in range(n_episode):
...     action = epsilon_greedy_policy(Q)
...     reward = bandit_env.step(action)
...     action_count[action] += 1
...     action_total_reward[action] += reward
...     Q[action] = action_total_reward[action] / action_count[action]
...     for a in range(n_action):
...         if action_count[a]:
...             action_avg_reward[a].append(
                        action_total_reward[a] / action_count[a])
...         else:
...             action_avg_reward[a].append(0)
```

7. По завершении построим график зависимости среднего вознаграждения от времени:

```
>>> import matplotlib.pyplot as plt
>>> for action in range(n_action):
...     plt.plot(action_avg_reward[action])
>>> plt.legend(['Рычаг {}'.format(action) for action in range(n_action)])
>>> plt.title('Зависимость вознаграждения от времени')
>>> plt.xscale('log')
>>> plt.xlabel('Эпизод')
>>> plt.ylabel('Среднее вознаграждение')
>>> plt.show()
```

Как это работает

Как и в других МППР, ε-жадная стратегия выбирает наилучший рычаг с вероятностью 1 – ε, а случайное действие – с вероятностью ε. Величина ε определяет компромисс между исследованием и использованием.

График, построенный на шаге 7, выглядит следующим образом:

Рычаг 1 дает наибольшее среднее вознаграждение. Среднее вознаграждение стабилизируется после примерно 1000 эпизодов.

Это еще не все

Возникает вопрос, действительно ли ε-жадная стратегия превосходит случайную. Во-первых, при следовании ε-жадной стратегии сходимость наступает быстрее, а во-вторых, как легко убедиться, величина среднего вознаграждения выше.

Действительно, усредним вознаграждение по всем эпизодам:

```
>>> print(sum(action_total_reward) / n_episode)
0.43718
```

При 100 000 эпизодов средний выигрыш при ε-жадной стратегии составляет 0.43718. Такое же вычисление для случайной стратегии дает 0.37902.

Решение задачи о многоруком бандите с помощью softmax-исследования

В этом рецепте мы применим к задаче о многоруком бандите алгоритм softmax-исследования и посмотрим, как он отличается от ε-жадной стратегии.

Мы видели, что при следовании ε-жадной стратегии не лучший рычаг слу-

чайно выбирается с вероятностью ε/|A|. Все такие рычаги трактуются одинаково, независимо от ценности, возвращаемой Q-функцией. Лучший рычаг также выбирается с фиксированной вероятностью, не зависящей от его ценности. В случае **softmax-исследования** рычаг выбирается с вероятностью, основанной на **softmax-распределении** значений Q-функции. Эта вероятность вычисляется по формуле:

$$P(a) = \frac{\exp(Q(a)/\tau)}{\sum_i^{|A|} \exp(Q(i)/\tau)},$$

где τ – температурный коэффициент, задающий степень случайности исследования. Чем больше τ, тем ближе исследование к случайному; чем меньше τ, тем с большей вероятностью выбирается лучший рычаг.

Как это делается

Решим задачу о многоруком бандите с помощью алгоритма softmax-исследования.

1. Импортируем библиотеку PyTorch и окружающую среду, разработанную в рецепте «Создание окружающей среды с многоруким бандитом» (предполагается, что код класса `BanditEnv` находится в файле `multi_armed_bandit.py`):

```
>>> import torch
>>> from multi_armed_bandit import BanditEnv
```

2. Определим вероятности и величины выигрыша для трехрукого бандита и создадим экземпляр окружающей среды:

```
>>> bandit_payout = [0.1, 0.15, 0.3]
>>> bandit_reward = [4, 3, 1]
>>> bandit_env = BanditEnv(bandit_payout, bandit_reward)
```

3. Зададим количество эпизодов и определим списки для хранения следующих данных: полное вознаграждение, полученное при выборе каждого рычага, сколько раз выбирался каждый рычаг и среднее вознаграждение для каждого рычага.

```
>>> n_episode = 100000
>>> n_action = len(bandit_payout)
>>> action_count = [0 for _ in range(n_action)]
>>> action_total_reward = [0 for _ in range(n_action)]
>>> action_avg_reward = [[] for action in range(n_action)]
```

4. Определим функцию softmax-исследования, зададим значение τ и создадим экземпляр стратегии:

```
>>> def gen_softmax_exploration_policy(tau):
...     def policy_function(Q):
...         probs = torch.exp(Q / tau)
...         probs = probs / torch.sum(probs)
...         action = torch.multinomial(probs, 1).item()
```

```
...         return action
...     return policy_function
>>> tau = 0.1
>>> softmax_exploration_policy = gen_softmax_exploration_policy(tau)
```

5. Инициализируем Q-функцию, которая будет возвращать среднее вознаграждение для каждого рычага:

```
>>> Q = torch.zeros(n_action)
```

   Эту Q-функцию мы будем обновлять по ходу обучения.

6. Прогоним 100 000 эпизодов. После каждого эпизода будем обновлять статистику рычагов:

```
>>> for episode in range(n_episode):
...     action = softmax_exploration_policy(Q)
...     reward = bandit_env.step(action)
...     action_count[action] += 1
...     action_total_reward[action] += reward
...     Q[action] = action_total_reward[action] / action_count[action]
...     for a in range(n_action):
...         if action_count[a]:
...             action_avg_reward[a].append(
                            action_total_reward[a] / action_count[a])
...         else:
...             action_avg_reward[a].append(0)
```

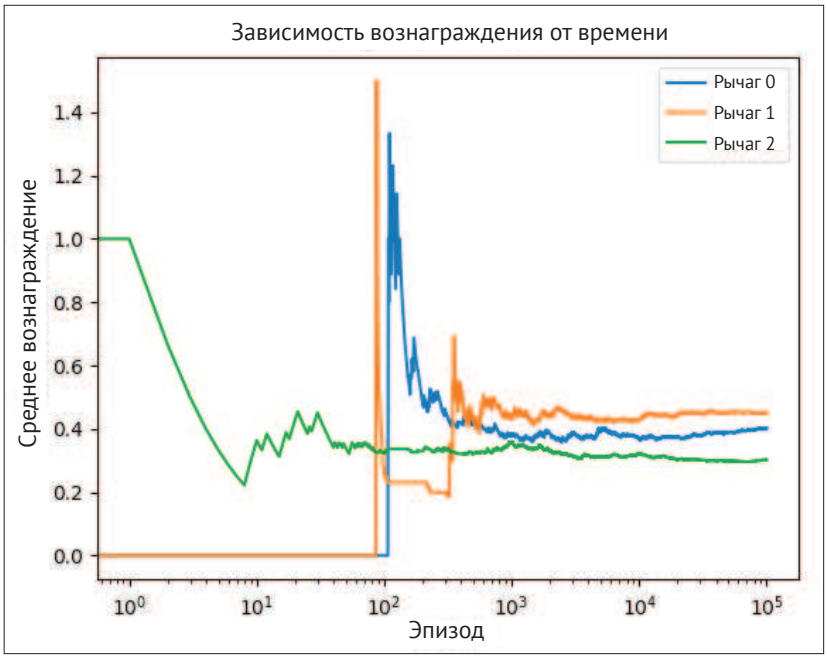
7. По завершении построим график зависимости среднего вознаграждения от времени:

```
>>> import matplotlib.pyplot as plt
>>> for action in range(n_action):
...     plt.plot(action_avg_reward[action])
>>> plt.legend(['Рычаг {}'.format(action) for action in range(n_action)])
>>> plt.title('Зависимость вознаграждения от времени')
>>> plt.xscale('log')
>>> plt.xlabel('Эпизод')
>>> plt.ylabel('Среднее вознаграждение')
>>> plt.show()
```

Как это работает

При следовании стратегии softmax-исследования дилемма исследования–использования решается с помощью функции softmax, построенной по значениям Q-функции. Вместо фиксирования вероятностей выбора лучшего и случайного рычага эта стратегия вычисляет вероятности, пользуясь softmax-распределением с температурным коэффициентом τ. Чем больше τ, тем больше внимания уделяется исследованию.

На шаге 7 будет построен следующий график:



Рычаг 1 дает наибольшее среднее вознаграждение. Среднее вознаграждение стабилизируется после примерно 800 эпизодов.

Решение задачи о многоруком бандите с помощью алгоритма верхней доверительной границы

В двух предыдущих рецептах мы исследовали случайные действия в задаче о многоруком бандите с фиксированными вероятностями, как в ε-жадной стратегии, или с вероятностями, основанными на значениях Q-функции, как в алгоритме softmax-исследования. В обоих случаях вероятности выбора случайных действий не изменяются со временем. В идеале мы хотели бы уменьшить объем исследования по мере того, как обучение приближается к концу. В этом рецепте используется алгоритм **верхней доверительной границы**, как раз и предназначенный для этой цели.

В основе алгоритма **верхней доверительной границы** (upper confidence bound – **UCB**) лежит идея доверительного интервала. Вообще говоря, доверительный интервал – это диапазон, внутри которого находится истинное значение. В алгоритме UCB доверительным интервалом рычага является диапазон, в котором находится среднее вознаграждение, полученное при выборе этого рычага. Интервал имеет вид [нижняя доверительная граница, верхняя доверительная граница], но для оценки потенциала рычага мы будем использовать только верхнюю. UCB вычисляется по формуле

$$UCB(a) = Q(a) + \sqrt{2\log(t)/N(a)},$$

где t – количество эпизодов, а $N(a)$ – сколько раз в этих t эпизодах выбирался рычаг a. По мере прогресса обучения доверительный интервал сужается и становится все более точным. Выбирать следует рычаг с наибольшей UCB.

Как это делается

Решим задачу о многоруком бандите с помощью алгоритма UCB.

1. Импортируем библиотеку PyTorch и окружающую среду, разработанную в рецепте «Создание окружающей среды с многоруким бандитом» (предполагается, что код класса `BanditEnv` находится в файле `multi_armed_bandit.py`):

```
>>> import torch
>>> from multi_armed_bandit import BanditEnv
```

2. Определим вероятности и величины выигрыша для трехрукого бандита и создадим экземпляр окружающей среды:

```
>>> bandit_payout = [0.1, 0.15, 0.3]
>>> bandit_reward = [4, 3, 1]
>>> bandit_env = BanditEnv(bandit_payout, bandit_reward)
```

3. Зададим количество эпизодов и определим списки для хранения следующих данных: полное вознаграждение, полученное при выборе каждого рычага, сколько раз выбирался каждый рычаг и среднее вознаграждение для каждого рычага.

```
>>> n_episode = 100000
>>> n_action = len(bandit_payout)
>>> action_count = torch.tensor([0. for _ in range(n_action)])
>>> action_total_reward = [0 for _ in range(n_action)]
>>> action_avg_reward = [[] for action in range(n_action)]
```

4. Определим функцию стратегии UCB, которая вычисляет лучший рычаг по формуле верхней доверительной границы:

```
>>> def upper_confidence_bound(Q, action_count, t):
...     ucb = torch.sqrt((2 * torch.log(torch.tensor(float(t))))
                                          / action_count) + Q
...     return torch.argmax(ucb)
```

5. Инициализируем Q-функцию, которая будет возвращать среднее вознаграждение для каждого рычага:

```
>>> Q = torch.empty(n_action)
```

   Эту Q-функцию мы будем обновлять по ходу обучения.

6. Прогоним 100 000 эпизодов со стратегией UCB. После каждого эпизода будем обновлять статистику рычагов:

```
>>> for episode in range(n_episode):
...     action = upper_confidence_bound(Q, action_count, episode)
...     reward = bandit_env.step(action)
...     action_count[action] += 1
...     action_total_reward[action] += reward
...     Q[action] = action_total_reward[action] / action_count[action]
...     for a in range(n_action):
...         if action_count[a]:
...             action_avg_reward[a].append(
                        action_total_reward[a] / action_count[a])
...         else:
...             action_avg_reward[a].append(0)
```

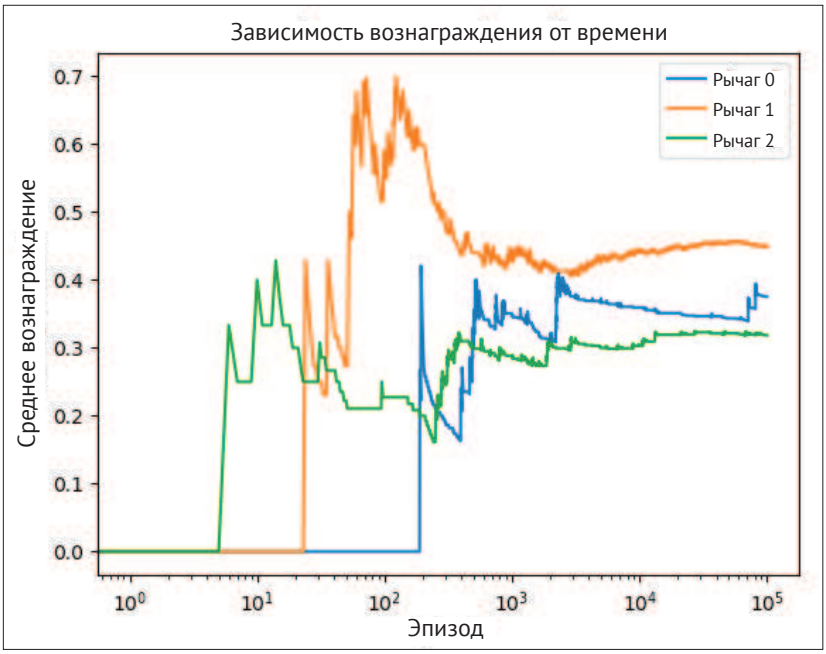
7. По завершении построим график зависимости среднего вознаграждения от времени:

```
>>> import matplotlib.pyplot as plt
>>> for action in range(n_action):
...     plt.plot(action_avg_reward[action])
>>> plt.legend(['Рычаг {}'.format(action) for action in range(n_action)])
>>> plt.title('Зависимость вознаграждения от времени')
>>> plt.xscale('log')
>>> plt.xlabel('Эпизод')
>>> plt.ylabel('Среднее вознаграждение')
>>> plt.show()
```

Как это работает

В этом рецепте мы решили задачу о многоруком бандите с помощью алгоритма UCB. Он адаптивно решает дилемму исследования–использования, изменяя вероятности по мере продвижения обучения. Если информации о действии еще мало, его доверительный интервал относительно широкий, поэтому неопределенность, сопровождающая выбор такого действия, высока. Чем в большем количестве эпизодов выбиралось действие, тем уже доверительный интервал. В таком случае можно с большей определенностью сказать, стоит выбирать действие или нет. В конечном итоге алгоритм UCB в каждом эпизоде выбирает рычаг с наибольшим значением UCB, и степень уверенности со временем возрастает.

На шаге 7 будет построен следующий график:



Рычаг 1 дает наибольшее среднее вознаграждение.

Это еще не все

Возникает вопрос, действительно ли алгоритм UCB превосходит ε-жадную стратегию. Вычислим среднее вознаграждение по всем эпизодам:

```
>>> print(sum(action_total_reward) / n_episode)
0.44605
```

В 100 000 эпизодов UCB приносит среднее вознаграждение 0.44605, это больше, чем при следовании ε-жадной стратегии – 0.43718.

См. также

Если вы хотите почитать о доверительных интервалах, обратитесь к статье по адресу http://www.stat.yale.edu/Courses/1997-98/101/confint.htm.

Решение задачи о рекламе в интернете с помощью алгоритма многорукого бандита

Представьте, что вы рекламщик и хотите оптимизировать показ объявлений на сайте.

- Есть три возможных цвета фона рекламных объявлений: красный, зеленый и синий. Какой цвет обеспечивает наибольшую кликабельность (CTR)?
- Есть три способа подачи рекламы: *узнайте…*, *бесплатно…* и *попробуйте*. При каком из них кликабельность максимальна?

Для каждого посетителя требуется выбрать объявление, которое максимизирует CTR на протяжении длительного времени. Как решить эту задачу?

Возможно, вы подумали про A/B-тестирование, когда весь трафик случайным образом разбивается на группы, каждому объявлению назначается своя группа, а затем выбирается объявление с наибольшим CTR за весь период наблюдения. Но по существу это чистое исследование, к тому же обычно мы не знаем, как долго продлится наблюдение, и можем потерять заметную долю потенциальных кликов. Кроме того, при A/B-тестировании предполагается, что неизвестный CTR объявления со временем остается неизменным. В противном случае A/B-тестирование нужно периодически повторять.

Подход на основе многорукого бандита, безусловно, лучше A/B-тестирования. Рычагами здесь являются объявления, а вознаграждение, выплачиваемое при выборе рычага, равно 1 (клик) или 0 (нет клика).

Попробуем применить к этой задаче алгоритм UCB.

Как это делается

Для применения алгоритма UCB к задаче о рекламе в интернете поступим следующим образом.

1. Импортируем библиотеку PyTorch и окружающую среду, разработанную в рецепте «Создание окружающей среды с многоруким бандитом» (предполагается, что код класса `BanditEnv` находится в файле `multi_armed_bandit.py`):

```
>>> import torch
>>> from multi_armed_bandit import BanditEnv
```

2. Определим вероятности и величины выигрыша для трехрукого бандита и создадим экземпляр окружающей среды:

```
>>> bandit_payout = [0.01, 0.015, 0.03]
>>> bandit_reward = [1, 1, 1]
>>> bandit_env = BanditEnv(bandit_payout, bandit_reward)
```

   Здесь CTR для объявления 0 равен 1 %, для объявления 1 – 1.5 %, а для объявления 2 – 3 %.

3. Зададим количество эпизодов и определим списки для хранения следующих данных: полное вознаграждение, полученное при выборе каждого рычага, сколько раз выбирался каждый рычаг и среднее вознаграждение для каждого рычага.

```
>>> n_episode = 100000
>>> n_action = len(bandit_payout)
>>> action_count = torch.tensor([0. for _ in range(n_action)])
```

```
>>> action_total_reward = [0 for _ in range(n_action)]
>>> action_avg_reward = [[] for action in range(n_action)]
```

4. Определим функцию стратегии UCB, которая вычисляет лучший рычаг по формуле верхней доверительной границы:

```
>>> def upper_confidence_bound(Q, action_count, t):
...     ucb = torch.sqrt((2 * torch.log(
                    torch.tensor(float(t)))) / action_count) + Q
...     return torch.argmax(ucb)
```

5. Инициализируем Q-функцию, которая будет возвращать среднее вознаграждение для каждого рычага:

```
>>> Q = torch.empty(n_action)
```

   Эту Q-функцию мы будем обновлять по ходу обучения.

6. Прогоним 100 000 эпизодов со стратегией UCB. После каждого эпизода будем обновлять статистику рычагов:

```
>>> for episode in range(n_episode):
...     action = upper_confidence_bound(Q, action_count, episode)
...     reward = bandit_env.step(action)
...     action_count[action] += 1
...     action_total_reward[action] += reward
...     Q[action] = action_total_reward[action] / action_count[action]
...     for a in range(n_action):
...         if action_count[a]:
...             action_avg_reward[a].append(
                    action_total_reward[a] / action_count[a])
...         else:
...             action_avg_reward[a].append(0)
```

7. По завершении построим график зависимости среднего вознаграждения от времени:

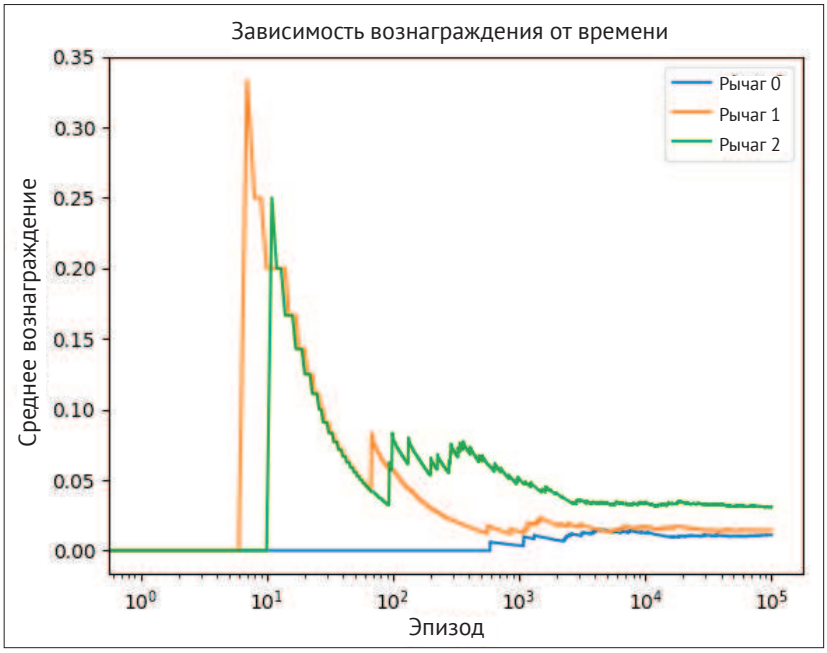
```
>>> import matplotlib.pyplot as plt
>>> for action in range(n_action):
...     plt.plot(action_avg_reward[action])
>>> plt.legend(['Рычаг {}'.format(action) for action in range(n_action)])
>>> plt.title('Зависимость вознаграждения от времени')
>>> plt.xscale('log')
>>> plt.xlabel('Эпизод')
>>> plt.ylabel('Среднее вознаграждение')
>>> plt.show()
```

Как это работает

В этом рецепте мы решили задачу оптимизации показа рекламных объявлений, рассмотрев ее как задачу о многоруком бандите. Мы преодолели трудности, свойственные подходу на основе A/B-тестирования. Для решения задачи о многоруком бандите мы применили алгоритм UCB; вознаграждение за выбор каждого рычага равно 1 или 0. Вместо чистого исследования без какой-либо связи между действием и вознаграждением UCB (и другие алгоритмы,

например ε-жадный и softmax-исследование) динамически переключаются между исследованием и использованием по мере необходимости. Если для объявления еще мало данных, то доверительный интервал относительно широкий, поэтому неопределенность, сопровождающая его показ, высока. Чем в большем количестве эпизодов выбиралось объявление, тем уже доверительный интервал и тем меньше неопределенность.

На шаге 7 будет построен следующий график:



Наибольшая кликабельность (среднее вознаграждение) предсказана для объявления 2, и алгоритм сходится.

Итак, мы нашли, что оптимально показывать объявление 2. И чем раньше мы установим этот факт, тем лучше, поскольку число потенциально потерянных кликов будет меньше. В данном примере превосходство объявления 2 стало очевидно после примерно 100 эпизодов.

Решение задачи о многоруком бандите с помощью выборки Томпсона

В этом рецепте мы разрешим дилемму исследования–использования в задаче о рекламе в интернете, применив другой алгоритм – выборку Томпсона. Как мы увидим, он сильно отличается от рассмотренных выше алгоритмов.

Выборку Томпсона (Thompson sampling – **TS**) называют еще байесовским бандитом, потому что этот алгоритм основан на байесовском подходе, т. е. для него характерны следующие особенности:

- это вероятностный алгоритм;
- он вычисляет априорное распределение для каждого рычага и производит выборку ценности действия из этого распределения;
- затем выбирается рычаг с наибольшей ценностью и наблюдается полученное вознаграждение;
- наконец, априорное распределение обновляется с учетом результата наблюдения. Эта процедура называется **байесовским обновлением**.

В нашей задаче оптимизации вознаграждение за нажатие каждого рычага равно 1 или 0. В качестве априорного можно взять **бета-распределение**, которое служит для описания случайных величин, значения которых ограничены конечным интервалом. Бета-распределение двухпараметрическое, с параметрами α и β. Величина α говорит, сколько раз мы получили вознаграждение 1, а β – сколько раз получено вознаграждение 0.

Чтобы вы лучше освоились, рассмотрим несколько бета-распределений, прежде чем приступать к реализации алгоритма TS.

Как это делается

Для исследования бета-распределения выполним следующие действия.

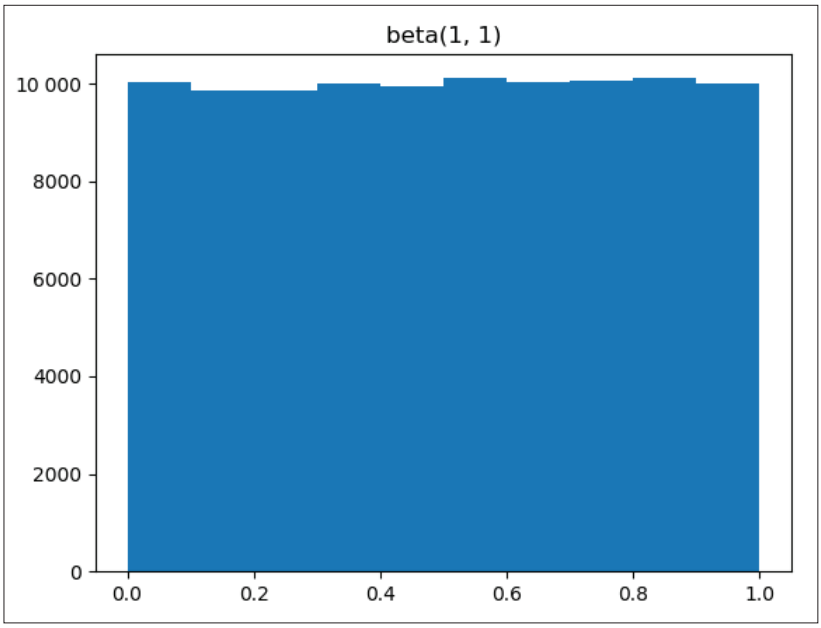
1. Импортируем PyTorch, а также библиотеку matplotlib, чтобы наглядно показать форму распределений.

```
>>> import torch
>>> import matplotlib.pyplot as plt
```

2. Для начала рассмотрим форму распределения с параметрами $\alpha = 1$ и $\beta = 1$:

```
>>> beta1 = torch.distributions.beta.Beta(1, 1)
>>> samples1 = [beta1.sample() for _ in range(100000)]
>>> plt.hist(samples1, range=[0, 1], bins=10)
>>> plt.title('beta(1, 1)')
>>> plt.show()
```

Будет нарисован такой график:

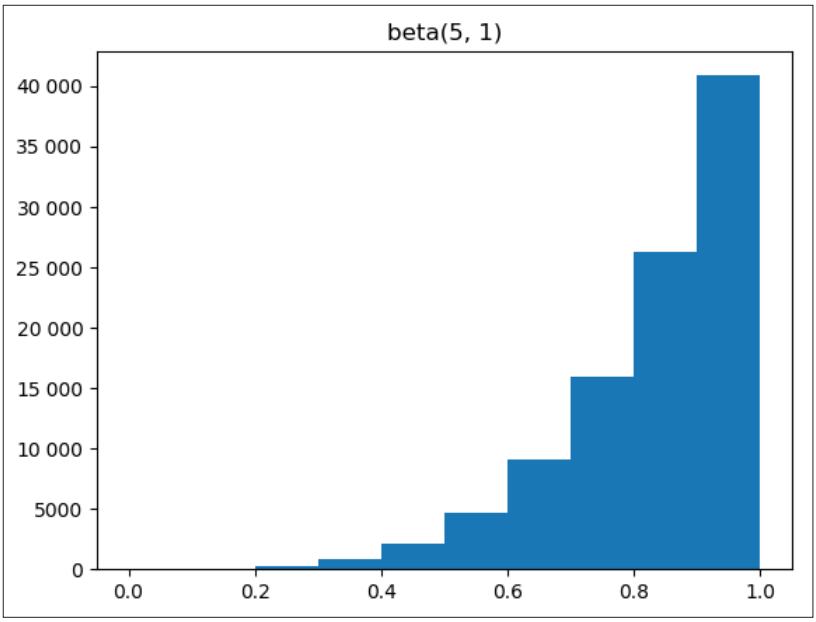


Очевидно, что при $\alpha = 1$ и $\beta = 1$ распределение не дает никакой информации о том, где в интервале от 0 до 1 находится истинное значение. Мы получаем просто равномерное распределение.

3. Теперь посмотрим, как выглядит бета-распределение при $\alpha = 5$ и $\beta = 1$:

```
>>> beta2 = torch.distributions.beta.Beta(5, 1)
>>> samples2 = [beta2.sample() for _ in range(100000)]
>>> plt.hist(samples2, range=[0, 1], bins=10)
>>> plt.title('beta(5, 1)')
>>> plt.show()
```

График получится таким:

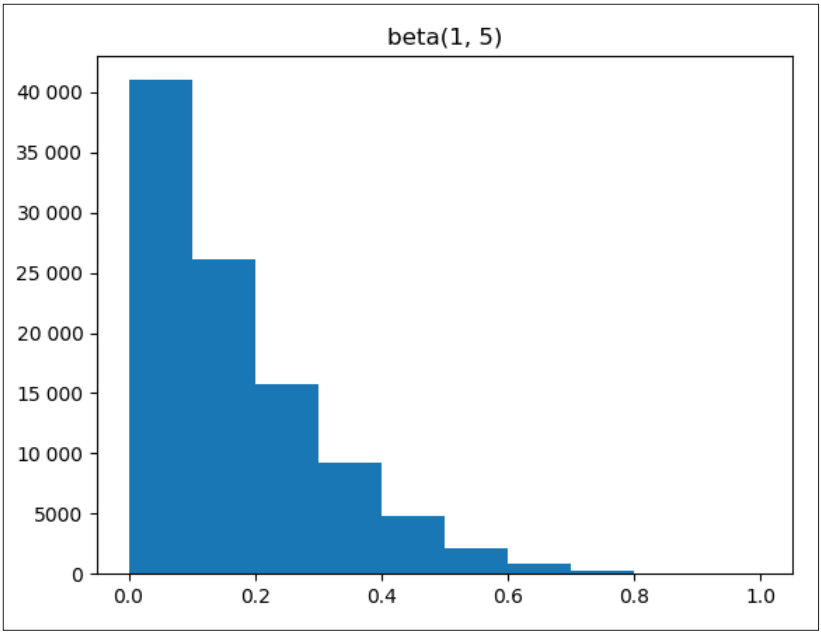


Параметры $\alpha = 5$ и $\beta = 1$ означают, что в четырех испытаниях получено 4 вознаграждения, равных 1, подряд. Распределение явно смещено в сторону 1.

4. Теперь пусть $\alpha = 1$ и $\beta = 5$:

```
>>> beta3 = torch.distributions.beta.Beta(1, 5)
>>> samples3= [beta3.sample() for _ in range(100000)]
>>> plt.hist(samples3, range=[0, 1], bins=10)
>>> plt.title('beta(1, 5)')
>>> plt.show()
```

Вот как выглядит график:

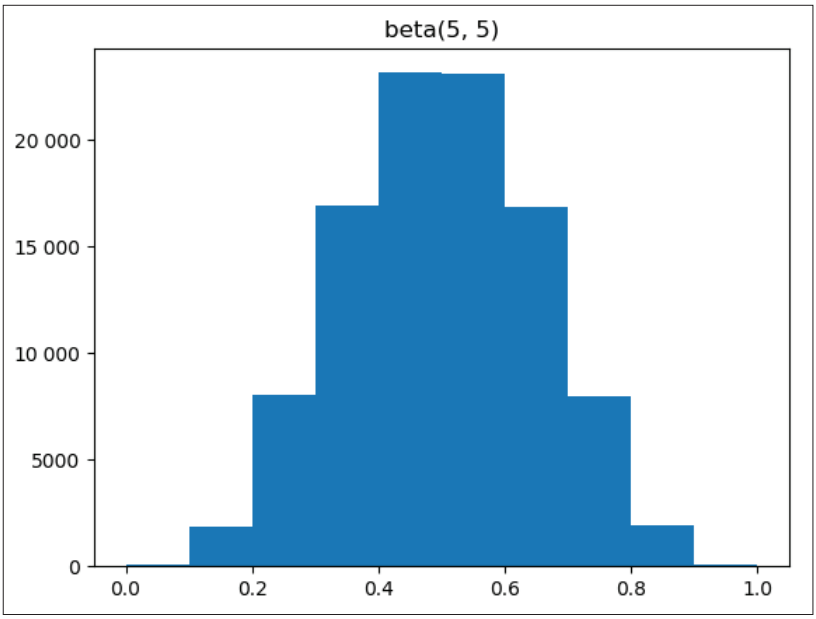


Параметры $\alpha = 1$ и $\beta = 5$ означают, что в четырех испытаниях получено 4 вознаграждения, равных 0, подряд. Распределение явно смещено в сторону 0.

5. И наконец, рассмотрим ситуацию $\alpha = 5$ и $\beta = 5$:

```
>>> beta4 = torch.distributions.beta.Beta(5, 5)
>>> samples4= [beta4.sample() for _ in range(100000)]
>>> plt.hist(samples4, range=[0, 1], bins=10)
>>> plt.title('beta(5, 5)')
>>> plt.show()
```

Теперь график выглядит так:



Параметры $\alpha = 5$ и $\beta = 5$ означают, что в восьми испытаниях наблюдалось поровну кликов и отсутствия кликов. Распределение концентрируется в окрестности средней точки 0.5.

Теперь решим задачу о рекламе в интернете с помощью алгоритма выборки Томпсона.

1. Импортируем окружающую среду, разработанную в рецепте «Создание окружающей среды с многоруким бандитом» (предполагается, что код класса `BanditEnv` находится в файле `multi_armed_bandit.py`):

```
>>> import torch
>>> from multi_armed_bandit import BanditEnv
```

2. Определим вероятности и величины выигрыша для трехрукого бандита (три объявления-кандидата) и создадим экземпляр окружающей среды:

```
>>> bandit_payout = [0.01, 0.015, 0.03]
>>> bandit_reward = [1, 1, 1]
>>> bandit_env = BanditEnv(bandit_payout, bandit_reward)
```

3. Зададим количество эпизодов и определим списки для хранения следующих данных: полное вознаграждение, полученное при выборе каждого рычага, сколько раз выбирался каждый рычаг и среднее вознаграждение для каждого рычага.

```
>>> n_episode = 100000
>>> n_action = len(bandit_payout)
>>> action_count = torch.tensor([0. for _ in range(n_action)])
```

```
>>> action_total_reward = [0 for _ in range(n_action)]
>>> action_avg_reward = [[] for action in range(n_action)]
```

4. Определим функцию TS, которая производит выборку из бета-распределения для каждого рычага, а затем возвращает рычаг с наибольшей ценностью.

```
>>> def thompson_sampling(alpha, beta):
...     prior_values = torch.distributions.beta.Beta(alpha, beta).sample()
...     return torch.argmax(prior_values)
```

5. Инициализируем α и β для каждого рычага:

```
>>> alpha = torch.ones(n_action)
>>> beta = torch.ones(n_action)
```

   Заметим, что начальные параметры всех бета-распределений $\alpha = \beta = 1$.

6. Прогоним 100 000 эпизодов, применяя алгоритм TS. В каждом эпизоде будем обновлять α и β каждого рычага с учетом наблюдаемого вознаграждения.

```
>>> for episode in range(n_episode):
...     action = thompson_sampling(alpha, beta)
...     reward = bandit_env.step(action)
...     action_count[action] += 1
...     action_total_reward[action] += reward
...     if reward > 0:
...         alpha[action] += 1
...     else:
...         beta[action] += 1
...     for a in range(n_action):
...         if action_count[a]:
...             action_avg_reward[a].append(
                    action_total_reward[a] / action_count[a])
...         else:
...             action_avg_reward[a].append(0)
```

7. По завершении построим график зависимости среднего вознаграждения от времени:

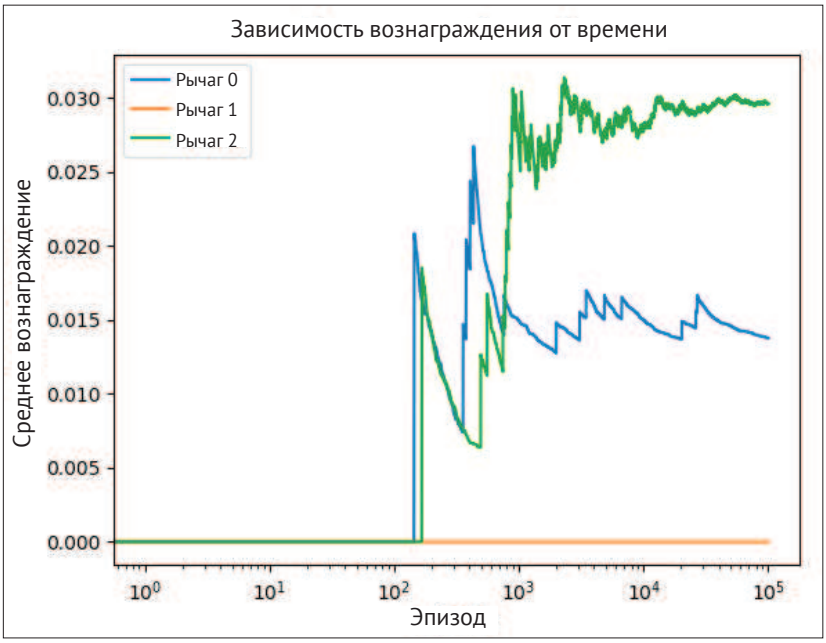
```
>>> import matplotlib.pyplot as plt
>>> for action in range(n_action):
...     plt.plot(action_avg_reward[action])
>>> plt.legend(['Рычаг {}'.format(action) for action in range(n_action)])
>>> plt.title('Зависимость вознаграждения от времени')
>>> plt.xscale('log')
>>> plt.xlabel('Эпизод')
>>> plt.ylabel('Среднее вознаграждение')
>>> plt.show()
```

Как это работает

В этом рецепте мы решили задачу о показе рекламы в интернете с помощью алгоритма TS. Основное отличие TS от трех других подходов – применение байесовской оптимизации. Сначала вычисляется априорное распределение

для каждого рычага, а затем из каждого распределения производится случайная выборка. После этого выбирается рычаг с наибольшей ценностью, а наблюдаемый результат используется для обновления априорного распределения. Стратегия TS является стохастической и жадной. Если вероятность клика по объявлению высока, то соответствующее ему бета-распределение смещено в сторону 1, поэтому случайно выбранное значение будет ближе к 1.

На шаге 7 будет построен следующий график:



Наибольшая кликабельность (среднее вознаграждение) предсказана для объявления 2.

См. также

Желающие почитать о бета-распределении могут обратиться к следующим источникам:

- https://www.itl.nist.gov/div898/handbook/eda/section3/eda366h.htm;
- http://varianceexplained.org/statistics/beta_distribution_and_baseball/.

Решение задачи о рекламе в интернете с помощью контекстуальных бандитов

Вы, наверное, обратили внимание, что при оптимизации показа объявлений мы интересовались только самим объявлением и игнорировали всю остальную информацию, например о пользователе и веб-странице, хотя она также может

влиять на то, будет переход по ссылке или нет. В этом рецепте мы поговорим о том, как учесть дополнительную информацию, и решим задачу методом контекстуальных бандитов.

До сих пор в задаче о многоруких бандитах не участвовало понятие состояния, и в этом состоит глубокое отличие от МППР. Нам доступно несколько действий, и вознаграждение начисляется в зависимости от выбранного действия. **Контекстуальные бандиты** – это обобщение многоруких бандитов путем добавления состояния. Состояние дает описание окружающей среды и помогает агенту принимать более обоснованные решения. В примере интернет-рекламы состоянием может быть пол пользователя (два значения – мужской и женский), возрастная группа пользователя (например, четыре значения) или категория страницы (скажем, спорт, финансы или новости). Интуитивно кажется, что пользователи из определенной демографической группы более склонны щелкать по объявлениям, размещенным на определенных страницах.

Понять идею контекстуальных бандитов нетрудно. Многорукий бандит – это один игральный автомат с несколькими рычагами, тогда как контекстуальный бандит – это множество таких автоматов. Каждый автомат представляет состояние с несколькими действиями (рычагами). Цель обучения – найти наилучший рычаг (действие) для каждого автомата (состояния).

Для простоты рассмотрим пример интернет-рекламы с двумя состояниями.

Как это делается

Решим задачу о контекстуальном бандите в области интернет-рекламы с помощью алгоритма UCB.

1. Импортируем библиотеку PyTorch и окружающую среду, разработанную в рецепте «Создание окружающей среды с многоруким бандитом» (предполагается, что код класса `BanditEnv` находится в файле `multi_armed_bandit.py`):

```
>>> import torch
>>> from multi_armed_bandit import BanditEnv
```

2. Определим вероятности и величины выигрыша для двух трехруких бандитов:

```
>>> bandit_payout_machines = [
...     [0.01, 0.015, 0.03],
...     [0.025, 0.01, 0.015]
... ]
>>> bandit_reward_machines = [
...     [1, 1, 1],
...     [1, 1, 1]
... ]
```

   Здесь истинный CTR объявления 0 равен 1 %, объявления 1 – 1.5 %, объявления 2 – 3 % для первого состояния и [2.5%, 1%, 1.5%] для второго состояния.

   Количество игральных автоматов в нашем случае равно двум:

```
>>> n_machine = len(bandit_payout_machines)
```

Создадим список бандитов, имея информацию о выплатах:

```
>>> bandit_env_machines = [BanditEnv(bandit_payout, bandit_reward)
...     for bandit_payout, bandit_reward in
...     zip(bandit_payout_machines, bandit_reward_machines)]
```

3. Зададим количество эпизодов и определим списки для хранения следующих данных: полное вознаграждение, полученное при выборе каждого рычага в каждом состоянии, сколько раз выбирался каждый рычаг в каждом состоянии и среднее вознаграждение для каждого рычага в каждом состоянии.

```
>>> n_episode = 100000
>>> n_action = len(bandit_payout_machines[0])
>>> action_count = torch.zeros(n_machine, n_action)
>>> action_total_reward = torch.zeros(n_machine, n_action)
>>> action_avg_reward = [[[] for action in range(n_action)] for _
in range(n_machine)]
```

4. Определим функцию стратегии UCB, которая вычисляет наилучший рычаг по формуле верхней доверительной границы:

```
>>> def upper_confidence_bound(Q, action_count, t):
...     ucb = torch.sqrt((2 * torch.log(
                torch.tensor(float(t)))) / action_count) + Q
...     return torch.argmax(ucb)
```

5. Инициализируем Q-функцию, которая будет возвращать среднее вознаграждение для каждого рычага в каждом состоянии:

```
>>> Q_machines = torch.empty(n_machine, n_action)
```

Эту Q-функцию мы будем обновлять по ходу обучения.

6. Прогоним 100 000 эпизодов, применяя алгоритм UCB. После каждого эпизода будем обновлять статистику каждого рычага в каждом состоянии.

```
>>> for episode in range(n_episode):
...     state = torch.randint(0, n_machine, (1,)).item()
...     action = upper_confidence_bound(
                Q_machines[state], action_count[state], episode)
...     reward = bandit_env_machines[state].step(action)
...     action_count[state][action] += 1
...     action_total_reward[state][action] += reward
...     Q_machines[state][action] = action_total_reward[state][action]
                                  / action_count[state][action]
...     for a in range(n_action):
...         if action_count[state][a]:
...             action_avg_reward[state][a].append(
                        action_total_reward[state][a]
                        / action_count[state][a])
...         else:
...             action_avg_reward[state][a].append(0)
```

7. По завершении построим график зависимости среднего вознаграждения от времени для каждого состояния:

```
>>> import matplotlib.pyplot as plt
>>>     for state in range(n_machine):
...     for action in range(n_action):
...         plt.plot(action_avg_reward[state][action])
...     plt.legend(['Рычаг {}'.format(action) for action in range(n_action)])
...     plt.xscale('log')
...     plt.title('Зависимость вознаграждения от времени для состояния{}'.
format(state))
...     plt.xlabel('Эпизод')
...     plt.ylabel('Среднее вознаграждение')
...     plt.show()
```

Как это работает

В этом рецепте мы решили задачу оптимизации интернет-рекламы с помощью алгоритма UCB для контекстуальных бандитов.

На шаге 7 будут построены показанные ниже графики.

Для первого состояния:

И для второго состояния:

В первом состоянии лучшим является объявление 2, для него предсказан наибольший CTR. Во втором состоянии лучшим является объявление 0. В обоих случаях это согласуется с истинным CTR.

Контекстуальный бандит представляет собой множество многоруких бандитов, каждый из которых соответствует одному состоянию окружающей среды. Состояние дает описание среды, и это помогает агенту принимать более обоснованные решения. В нашем примере мужчины перейдут по рекламной ссылке с большей вероятностью, чем женщины. Мы использовали два игральных автомата, чтобы включить оба состояния, и искали, какой рычаг выгоднее потянуть в каждом состоянии.

Следует отметить, что контекстуальные бандиты все равно отличаются от МППР, хотя и включают концепцию состояния. Во-первых, состояние контекстуального бандита не определяется предыдущими действиями или состояниями, а просто является наблюдением за окружающей средой. Во-вторых, не существует задержанного или обесцененного вознаграждения, потому что эпизод для контекстуального бандита всегда состоит из одного шага. Однако по сравнению с многорукими бандитами контекстуальные все же ближе к МППР, т. к. действия обусловлены состояниями окружающей среды. Можно сказать, что контекстуальные бандиты занимают промежуточное положение между многорукими бандитами и обучением с подкреплением на основе МППР.

Глава 6

Масштабирование с помощью аппроксимации функций

До сих пор в методах Монте-Карло и TD-методах мы представляли функцию ценности в виде таблицы. TD-методы позволяют обновлять Q-функцию динамически на протяжении эпизода, и это считается улучшением по сравнению с методами МК. Однако TD-методы по-прежнему плохо масштабируются на задачи с большим числом состояний или действий. Обучение в таких условиях заняло бы слишком много времени.

В этой главе мы будем рассматривать аппроксимацию функций, это позволит решить проблему масштабируемости TD-методов. Начнем с описания окружающей среды Mountain Car (машина на горе). Разработанную линейную оценку функции мы включим в алгоритмы Q-обучения и SARSA. Затем улучшим алгоритм Q-обучения, добавив буфер воспроизведения опыта, и поэкспериментируем с использованием нейронных сетей в качестве оценки функции. И наконец, покажем, как применить полученные в этой главе знания к решению задачи о балансировании стержня.

В этой главе приводятся следующие рецепты:

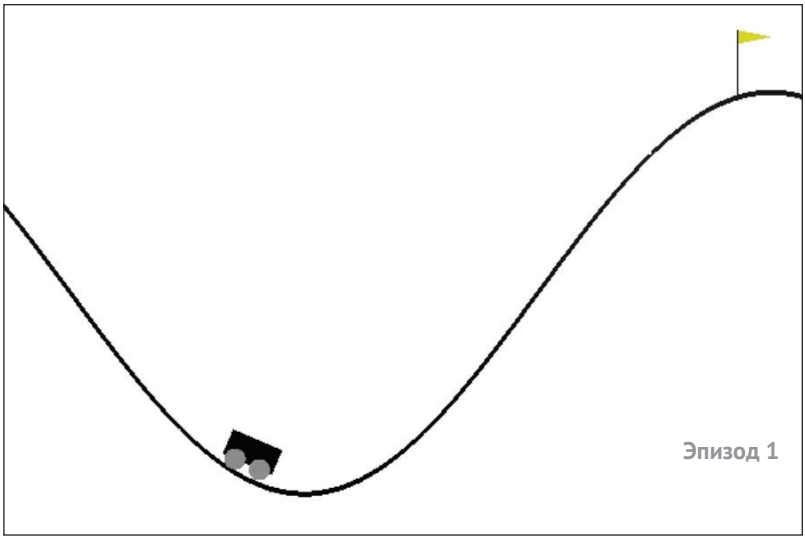
- подготовка окружающей среды Mountain Car;
- оценивание Q-функций посредством аппроксимации методом градиентного спуска;
- реализация Q-обучения с линейной аппроксимацией функций;
- реализация SARSA с линейной аппроксимацией функций;
- пакетная обработка с применением буфера воспроизведения опыта;
- реализация Q-обучения с аппроксимацией функций нейронной сетью;
- решение задачи о балансировании стержня с помощью аппроксимации функций.

Подготовка окружающей среды Mountain Car

TD-метод может обучать Q-функцию на каждом шаге эпизода, но он не масштабируется. Например, количество состояний в шахматах порядка 10^{40}, а в игре го – 10^{70}. К тому же обучиться ценностям состояний в непрерывном случае TD-метод вообще не способен. Поэтому такие задачи приходится решать, прибегая к аппроксимации функций, т. е. аппроксимации пространства состояний набором признаков.

В этом рецепте мы познакомимся с окружающей средой Mountain Car, а в последующих решим эту задачу с помощью аппроксимации функций.

Mountain Car (https://gym.openai.com/envs/MountainCar-v0/) – типичная окружающая среда Gym с непрерывными состояниями. Как показано на рисунке ниже, цель заключается в том, чтобы привести автомобиль на вершину горы.



Автомобиль может находиться в диапазоне от –1.2 (слева) до 0.6 (справа), а цель (желтый флажок) находится в точке с абсциссой 0.5. Двигатель автомобиля недостаточно мощный, чтобы преодолеть весь подъем самостоятельно, поэтому необходимо отъехать назад и разогнаться на спуске. На каждом шаге доступно три дискретных действия:

- газ, задний ход (0);
- по инерции (1);
- газ, передний ход (2).

И существует два состояния среды:

- позиция автомобиля: непрерывно изменяется от –1.2 до 0.6;
- скорость автомобиля: непрерывно изменяется от –0.07 до 0.07.

На каждом шаге начисляется вознаграждение –1, пока автомобиль не достигнет цели (позиции 0.5).

Эпизод заканчивается, когда будет достигнута цель (это понятно) или после 200 шагов.

Подготовка

Для начала найдем имя окружающей среды Mountain Car в таблице по адресу https://github.com/openai/gym/wiki/Table-of-environments. Она называется `MountainCar-v0`, пространство наблюдений в ней представлено числами с плавающей точкой, а действий всего три (влево = 0, по инерции = 1, вправо = 2).

Как это делается

Для имитации среды Mountain Car выполним следующие действия.

1. Импортируем библиотеку Gym и создадим экземпляр окружающей среды Mountain Car:

```
>>> import gym
>>> env = gym.envs.make("MountainCar-v0")
>>> n_action = env.action_space.n
>>> print(n_action)
3
```

2. Приведем окружающую среду в исходное состояние:

```
>>> env.reset()
array([-0.52354759, 0. ])
```

   Автомобиль начинает движение в состоянии `[-0.52354759, 0.]`, т. е. находится в районе точки –0.5 и имеет скорость 0. На вашем компьютере начальное положение может быть другим, т. к. это случайное число в диапазоне от –0.6 до –0.4.

3. Сначала попробуем наивный подход: просто будем давить на газ в надежде, что сможем доехать до вершины:

```
>>> is_done = False
>>> while not is_done:
...     next_state, reward, is_done, info = env.step(2)
...     print(next_state, reward, is_done)
...     env.render()
>>> env.render()
[-0.49286453  0.00077561] -1.0 False
[-0.4913191   0.00154543] -1.0 False
[-0.48901538  0.00230371] -1.0 False
[-0.48597058  0.0030448 ] -1.0 False
......
......
[-0.29239555 -0.0046231 ] -1.0 False
[-0.29761694 -0.00522139] -1.0 False
[-0.30340632 -0.00578938] -1.0 True
```

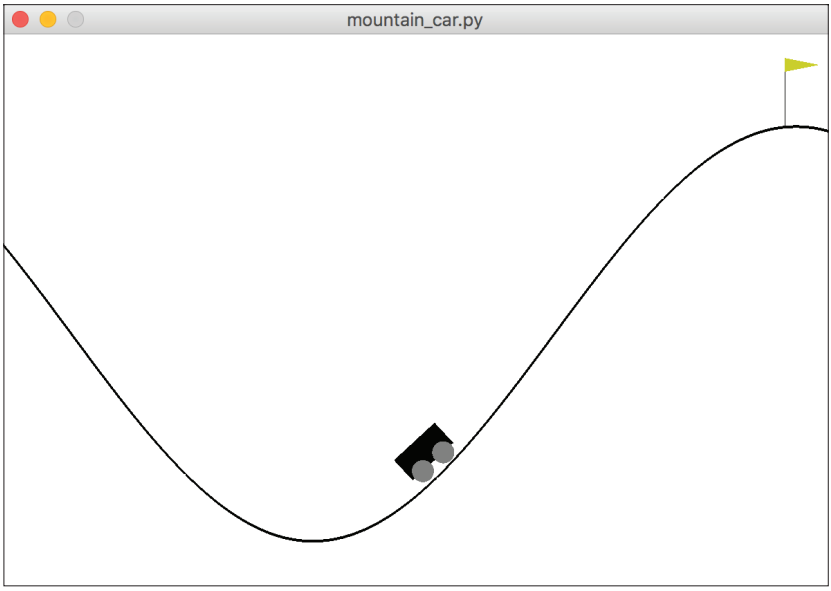
4. Закроем окружающую среду:

```
env.close()
```

Как это работает

На шаге 3 состояние (позиция и скорость) изменяется, и вознаграждение всякий раз равно –1.

На видео будет видно, что машина движется то вправо, то влево, но вершины горы не достигает.



Итак, задача о машине на горе не так проста, как казалось. Необходимо двигаться взад-вперед, чтобы разогнаться. А переменные состояния непрерывные, т. е. метод поиска в таблице с последующим ее обновлением (как в TD-методе) не годится. В следующем рецепте мы решим задачу с помощью аппроксимации функций.

Оценивание Q-функций посредством аппроксимации методом градиентного спуска

Начиная с этого рецепта мы будем разрабатывать алгоритмы аппроксимации для взаимодействия со средами, в которых переменные состояния непрерывны. Начнем с линейной аппроксимации Q-функций и метода градиентного спуска.

Основная идея аппроксимации функций – воспользоваться набором **признаков** для оценки значений Q-функции. Это очень полезно, когда пространство состояний велико и таблица Q-функции могла бы оказаться чрезмерно большой. Есть несколько способов отобразить признаки на значения Q-функции, например линейная аппроксимация (аппроксимация линейными комбинациями признаков) и нейронные сети. В случайно линейной аппрокси-

мации функция ценности состояний записывается в виде взвешенной суммы признаков:

$$V(s) = \theta_1 F_1(s) + \theta_2 F_2(s) + \ldots + \theta_n F_n(s),$$

где $F_1(s), F_2(s), \ldots, F_n(s)$ – набор признаков, зависящих от состояния s, а $\theta_1, \theta_2, \ldots, \theta_n$ – веса этих признаков. Это равенство можно записать также в векторном виде $V(s) = \theta F(s)$.

При описании TD-метода мы вычисляли будущие состояния по формуле

$$V(s_t) := V(s_t) + \alpha[r + \gamma V(s_{t+1}) - V(s_t)],$$

где r – вознаграждение, начисляемое при переходе из соединения s_t в состояние s_{t+1}, α – скорость обучения, γ – коэффициент обесценивания. Обозначив TD-ошибку буквой δ, будем иметь:

$$\delta = r + \gamma V(s_{t+1}) - V(s_t);$$
$$V(s_t) := V(s_t) + \alpha\delta.$$

Это и есть уравнения градиентного спуска. Таким образом, цель обучения – найти оптимальные веса θ для аппроксимации функции ценности состояний $V(s)$ для всех возможных действий. Функция потерь, которую мы пытаемся минимизировать, в этом случае такая же, как в задаче регрессии, т. е. среднеквадратическая ошибка между истинным значением и его оценкой. После каждого шага эпизода мы имеем новую оценку истинной ценности состояния и изменяем веса θ, приближаясь к оптимальным значениям.

Следует также обратить внимание на набор признаков $F(s)$. Хорошим считается набор признаков, улавливающий динамику изменения состояния. Обычно набор признаков генерируется с помощью нормальных распределений с разными параметрами – средним и стандартным отклонением.

Как это делается

Разработаем линейный аппроксиматор Q-функции.

1. Импортируем необходимые пакеты:

```
>>> import torch
>>> from torch.autograd import Variable
>>> import math
```

   Модуль `Variable` обертывает тензоры и поддерживает обратное распространение.

2. Реализуем метод `__init__` класса линейной оценки `Estimator`:

```
>>> class Estimator():
...     def __init__(self, n_feat, n_state, n_action, lr=0.05):
...         self.w, self.b = self.get_gaussian_wb(n_feat, n_state)
...         self.n_feat = n_feat
...         self.models = []
...         self.optimizers = []
```

```
...         self.criterion = torch.nn.MSELoss()
...         for _ in range(n_action):
...             model = torch.nn.Linear(n_feat, 1)
...             self.models.append(model)
...             optimizer = torch.optim.SGD(model.parameters(), lr)
...             self.optimizers.append(optimizer)
```

Он принимает три параметра: количество признаков `n_feat`, количество состояний и количество действий. Сначала генерируется набор коэффициентов `w` и `b` для функций $F(s)$, их значения выбираются из нормальных распределений, которые будут определены ниже. Затем инициализируется `n_action` линейных моделей, каждая из которых соответствует одному действию, и соответственно `n_action` оптимизаторов. Для линейных моделей мы будем использовать модуль `Linear` из PyTorch. Он принимает `n_feat` признаков и возвращает предсказанную ценность состояния для действия. Вместе с каждой линейной моделью инициализируется также оптимизатор методом стохастического градиентного спуска. Скорость обучения каждого оптимизатора равна 0.05. В качестве функции потерь берется среднеквадратическая ошибка.

3. Далее определим метод `get_gaussian_wb`, который генерирует набор коэффициентов `w` и `b` для линейной функции $F(s)$:

```
>>> def get_gaussian_wb(self, n_feat, n_state, sigma=.2):
...     """
...     Генерирует коэффициенты признаков, выбирая их из нормального
...     распределения
...     @param n_feat: количество признаков
...     @param n_state: количество состояний
...     @param sigma: параметр ядра
...     @return: коэффициенты признаков
...     """
...     torch.manual_seed(0)
...     w = torch.randn((n_state, n_feat)) * 1.0 / sigma
...     b = torch.rand(n_feat) * 2.0 * math.pi
...     return w, b
```

Матрица коэффициентов `w` имеет размер `n_feat` × `n_state`, а ее элементы выбираются из нормального распределения с дисперсией `sigma`; смещения `b` представлены списком `n_feat` значений, выбираемых из равномерного распределения на отрезке $[0, 2\pi]$.

Отметим, что очень важно выбирать конкретное начальное значение генератора случайных чисел (`torch.manual_seed(0)`), так чтобы при разных прогонах состояние отображалось на один и тот же набор признаков.

4. Теперь напишем функцию, которая, зная `w` и `b`, отображает пространство состояний в пространство признаков.

```
>>> def get_feature(self, s):
...     """
...     Генерирует признаки по входному состоянию
...     @param s: входное состояние
```

```
...     @return: признаки
...     """
...     features = (2.0 / self.n_feat) ** .5 * torch.cos(
            torch.matmul(torch.tensor(s).float(), self.w) + self.b)
...     return features
```

Признак, соответствующий состоянию *s*, вычисляется по формуле

$$F(s) = \sqrt{2/n\_feat}\cos(w * s + b).$$

Косинус в этой формуле гарантирует, что признак попадает в диапазон [–1, 1], каким бы ни было входное состояние.

5. Модель и процедура генерации признаков определены, и теперь можно реализовать метод обучения, который обновляет линейные модели, получив новое наблюдение.

```
>>>     def update(self, s, a, y):
...         """
...         Обновляет веса линейной оценки на основе переданного обучающего
...         примера
...         @param s: состояние
...         @param a: действие
...         @param y: целевое значение
...         """
...         features = Variable(self.get_feature(s))
...         y_pred = self.models[a](features)
...         loss = self.criterion(y_pred, Variable(torch.Tensor([y])))
...         self.optimizers[a].zero_grad()
...         loss.backward()
...         self.optimizers[a].step()
```

Получив обучающий пример, функция сначала переводит состояние в пространство признаков, вызывая метод `get_feature`. Полученные признаки подаются на вход текущей линейной модели действия `a`. По предсказанному результату и целевому значению вычисляется потеря и градиенты. Затем веса θ обновляются с помощью обратного распределения ошибки.

6. Следующая операция – предсказание ценности состояния для каждого действия с помощью текущих моделей:

```
>>>     def predict(self, s):
...         """
...         Вычисляет значения Q-функции от состояния, применяя
...         обученную модель
...         @param s: входное состояние
...         @return: ценности состояния
...         """
...         features = self.get_feature(s)
...         with torch.no_grad():
...             return torch.tensor([model(features)
                                     for model in self.models])
```

С классом `Estimator` мы закончили.

7. Теперь проверим его работу на тестовых данных. Сначала создадим объект `Estimator`, который отображает 2-мерное пространство состояний в 10-мерное пространство признаков и работает с одним действием:

```
>>> estimator = Estimator(10, 2, 1)
```

8. Сгенерируем признаки для состояния [0.5, 0.1]:

```
>>> s1 = [0.5, 0.1]
>>> print(estimator.get_feature(s1))
tensor([ 0.3163, -0.4467, -0.0450, -0.1490, 0.2393, -0.4181,
-0.4426, 0.3074, -0.4451, 0.1808])
```

   Как видим, получился 10-мерный вектор признаков.

9. Обучим оцениватель на списке состояний и их ценностей (в данном примере действие только одно):

```
>>> s_list = [[1, 2], [2, 2], [3, 4], [2, 3], [2, 1]]
>>> target_list = [1, 1.5, 2, 2, 1.5]
>>> for s, target in zip(s_list, target_list):
...     feature = estimator.get_feature(s)
...     estimator.update(s, 0, target)
```

10. И наконец, воспользуемся обученной линейной моделью для предсказания ценности новых состояний:

```
>>> print(estimator.predict([0.5, 0.1]))
tensor([0.6172])
>>> print(estimator.predict([2, 3]))
tensor([0.8733])
```

Для состояния [0.5, 0.1] предсказана ценность 0.5847, а для состояния [2, 3] – ценность 0.7969.

Как это работает

Метод аппроксимации функции заменяет ценности состояний более компактной моделью, чем таблица точных значений в TD-методе. Сначала пространство состояний отображается в пространство признаков, а затем значения Q-функции оцениваются с помощью модели регрессии. Обучение производится с учителем. В качестве модели регрессии может использоваться линейная модель или нейронная сеть. В этом рецепте мы разработали оцениватель на основе линейной регрессии. Он генерирует признаки в виде коэффициентов, выбираемых из нормального распределения. Веса линейной модели обновляются при поступлении обучающих данных методом градиентного спуска. Затем с помощью обученной модели предсказываются значения Q-функции.

Аппроксимация функций многократно уменьшает количество состояний в тех случаях, когда оно слишком велико для применения TD-метода. И что еще важнее, она допускает обобщение на ранее не встречавшиеся состояния, поскольку ценности состояний аппроксимированы функциями оценки, обученными на известных входных состояниях.

См. также

Читателям, незнакомым с линейной регрессией или методом градиентного спуска, рекомендуем следующие источники:

- https://towardsdatascience.com/step-by-step-tutorial-on-linearregression-with-stochastic-gradient-descent-1d35b088a843;
- https://machinelearningmastery.com/simple-linear-regression-tutorialfor-machine-learning/.

Реализация Q-обучения с линейной аппроксимацией функций

В предыдущем рецепте мы разработали оценку методом линейной регрессии. Теперь воспользуемся ей в алгоритме Q-обучения.

Выше мы видели, что Q-обучение – это алгоритм с разделенной стратегией, в котором Q-функция обновляется по формуле

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

Здесь s' – состояние, в которое окружающая среда переходит из состояния s после действия a, r – полученное при этом вознаграждение, α – скорость обучения, γ – коэффициент обесценивания. Член $\max_{a'} Q(s', a')$ означает, что стратегия жадная, т. е. для генерации обучающих данных выбирается действие в состоянии s' с наибольшим значением Q-функции. В алгоритме Q-обучения действия выбираются с помощью ε-жадной стратегии. В Q-обучении с аппроксимацией функций имеется похожий член ошибки:

$$\delta = r + \gamma V(s_{t+1}) - V(s_t) = r + \gamma \max_{a'} V(s') - V(s_t).$$

Цель обучения – минимизировать член ошибки, сведя его к нулю, т. е. оценка $V(s_t)$ должна удовлетворять уравнению

$$V(s_t) = r + \gamma \max_{a'} V(s').$$

Теперь задача сводится к тому, чтобы найти оптимальные веса θ в формуле $V(s) = \theta F(s)$, при которых достигается наилучшая аппроксимация функции ценности состояний $V(s)$ для всех возможных действий. Функция потерь, которую мы пытаемся минимизировать в этом случае, такая же, как в задаче регрессии, т. е. среднеквадратическая ошибка между истинным значением и его оценкой.

Как это делается

Реализуем алгоритм Q-обучения с аппроксимацией функций, пользуясь линейным оценивателем – классом `Estimator` из файла `linear_estimator.py`, который был создан в предыдущем рецепте.

1. Импортируем необходимые модули и создадим экземпляр окружающей среды Mountain Car:

```
>>> import gym
>>> import torch
>>> from linear_estimator import Estimator
>>> env = gym.envs.make("MountainCar-v0")
```

2. Определим ε-жадную стратегию:

```
>>> def gen_epsilon_greedy_policy(estimator, epsilon, n_action):
...     def policy_function(state):
...         probs = torch.ones(n_action) * epsilon / n_action
...         q_values = estimator.predict(state)
...         best_action = torch.argmax(q_values).item()
...         probs[best_action] += 1.0 - epsilon
...        action = torch.multinomial(probs, 1).item()
...         return action
...     return policy_function
```

Функции передается параметр ε, принимающий значения от 0 до 1, количество возможных действий |A| и оцениватель, используемый для предсказания ценностей состояний. Каждое действие выбирается с вероятностью ε/|A|, а лучшее из известных действий (с наибольшей ценностью пары состояние–действие) с вероятностью 1 – ε + ε/|A|.

3. Определим функцию, выполняющую Q-обучение с аппроксимацией функций.

```
>>> def q_learning(env, estimator, n_episode, gamma=1.0,
                   epsilon=0.1, epsilon_decay=.99):
... """
...     Алгоритм Q-обучения с аппроксимацией функций
...     @param env: окружающая среда Gym
...     @param estimator: объект класса Estimator
...     @param n_episode: количество эпизодов
...     @param gamma: коэффициент обесценивания
...     @param epsilon: параметр ε-жадной стратегии
...     @param epsilon_decay: коэффициент затухания epsilon
...     """
...     for episode in range(n_episode):
...     policy = gen_epsilon_greedy_policy(estimator,
                epsilon * epsilon_decay ** episode, n_action)
...     state = env.reset()
...     is_done = False
...     while not is_done:
...         action = policy(state)
...         next_state, reward, is_done, _ = env.step(action)
...         q_values_next = estimator.predict(next_state)
...         td_target = reward + gamma * torch.max(q_values_next)
...         estimator.update(state, action, td_target)
...         total_reward_episode[episode] += reward
...
...         if is_done:
...             break
...         state = next_state
```

Функция `q_learning()` выполняет следующие действия:

- в каждом эпизоде создает ε-жадную стратегию, причем ε затухает с коэффициентом 0.99 (если в первом эпизоде ε было равно 0.1, то во втором будет равно 0.099);
- выполняет эпизод: на каждом шаге выбирается действие a, следуя ε-жадной стратегии; значения Q-функции для нового состояния вычисляются с использованием текущего оценивателя, затем целевое значение $V(s_t) = r + \gamma \max_{a'} V(s')$ используется для обучения оценивателя;
- прогоняет `n_episode` эпизодов и запоминает полные вознаграждения в каждом эпизоде.

4. Задаем количество признаков 200, скорость обучения 0.03 и создаем оцениватель с такими параметрами:

```
>>> n_state = env.observation_space.shape[0]
>>> n_action = env.action_space.n
>>> n_feature = 200
>>> lr = 0.03
>>> estimator = Estimator(n_feature, n_state, n_action, lr)
```

5. Выполняем Q-обучение с аппроксимацией функций на 300 эпизодах, запоминая полные вознаграждения в каждом эпизоде.

```
>>> n_episode = 300
>>> total_reward_episode = [0] * n_episode
>>> q_learning(env, estimator, n_episode, epsilon=0.1)
```

6. Строим график зависимости вознаграждения в эпизоде от времени:

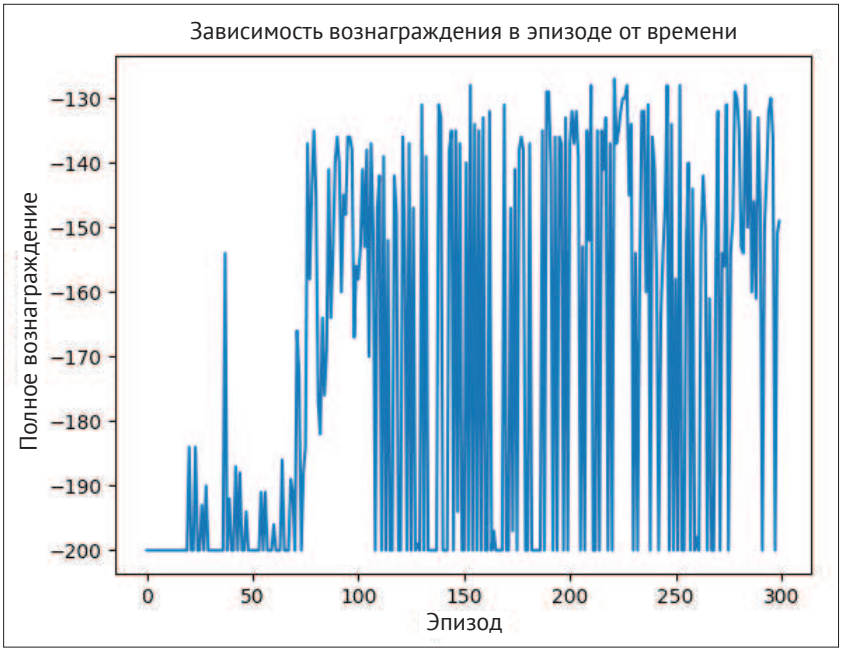
```
>>> import matplotlib.pyplot as plt
>>> plt.plot(total_reward_episode)
>>> plt.title('Зависимость вознаграждения в эпизоде от времени')
>>> plt.xlabel(Эпизод')
>>> plt.ylabel('Полное вознаграждение')
>>> plt.show()
```

Как это работает

Как видим, алгоритм Q-обучения с аппроксимацией функций пытается найти оптимальные веса в моделях аппроксимации, стремясь наилучшим образом оценить значения Q-функции. Он похож на табличное Q-обучение в том смысле, что оба генерируют обучающие данные с помощью другой стратегии. Он больше подходит для окружающих сред с большим пространством состояний, когда Q-функция аппроксимируется набором моделей регрессии и латентными признаками, в то время как при табличном Q-обучении для обновления ценностей применяется точный поиск в таблице. Тот факт, что модели регрессии обновляются после каждого шага эпизода, сближает Q-обучение с аппроксимацией функций и табличное Q-обучение.

После того как модель обучена, нам остается только применить модели регрессии для предсказания ценностей пар состояние–действие для всех воз-

можных действий и выбрать действие с максимальной ценностью в данном состоянии. График, построенный на шаге 6, выглядит следующим образом:



Мы видим, что после первых 25 итераций в большинстве эпизодов автомобилю, чтобы добраться до вершины горы, нужно от 130 до 160 шагов.

Реализация SARSA с линейной аппроксимацией функций

В предыдущем рецепте мы решили задачу о машине на горе с помощью алгоритма Q-обучения с разделенной стратегией. Теперь проделаем то же самое с помощью алгоритма SARSA (разумеется, его версии с аппроксимацией функций).

В алгоритме SARSA Q-функция обновляется по формуле:

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

Здесь s' – состояние, в которое окружающая среда переходит из состояния s после действия a, r – полученное при этом вознаграждение, α – скорость обучения, γ – коэффициент обесценивания. Следующее действие a' выбирается, следуя той же ε-жадной стратегии, которая используется для обновления Q. И это действие выполняется на следующем шаге. Поэтому член ошибки в SARSA с аппроксимацией функций выглядит так:

$$\delta = r + \gamma V(s_{t+1}) - V(s_t) = r + \gamma V(s', a') - V(s_t).$$

Цель обучения – минимизировать член ошибки, сведя его к нулю, т. е. оценка $V(s_t)$ должна удовлетворять уравнению

$$V(s_t) = r + \gamma V(s', a').$$

Теперь задача сводится к тому, чтобы найти оптимальные веса θ в формуле $V(s) = \theta F(s)$, при которых достигается наилучшая аппроксимация функции ценности состояний $V(s)$ для всех возможных действий. Функция потерь, которую мы пытаемся минимизировать в этом случае, такая же, как в задаче регрессии, т. е. среднеквадратическая ошибка между истинным значением и его оценкой.

Как это делается

Реализуем алгоритм Q-обучения с аппроксимацией функций, пользуясь линейным оценивателем – классом `Estimator` из файла `linear_estimator.py`, который был создан в рецепте «Оценивание Q-функций посредством аппроксимации методом градиентного спуска».

1. Импортируем необходимые модули и создадим экземпляр окружающей среды Mountain Car:

```
>>> import gym
>>> import torch
>>> from linear_estimator import Estimator
>>> env = gym.envs.make("MountainCar-v0")
```

2. Воспользуемся той же функцией ε-жадной стратегии, которая была написана в предыдущем рецепте.

3. Определим функцию, выполняющую алгоритм SARSA с аппроксимацией функций.

```
>>> def sarsa(env, estimator, n_episode, gamma=1.0,
epsilon=0.1, epsilon_decay=.99):
... """
...     Алгоритм SARSA с аппроксимацией функций
...     @param env: окружающая среда Gym
...     @param estimator: объект класса Estimator
...     @param n_episode: количество эпизодов
...     @param gamma: коэффициент обесценивания
...     @param epsilon: параметр ε-жадной стратегии
...     @param epsilon_decay: коэффициент затухания epsilon
...     """
...     for episode in range(n_episode):
...         policy = gen_epsilon_greedy_policy(estimator,
                            epsilon * epsilon_decay ** episode,
                            env.action_space.n)
...         state = env.reset()
...         action = policy(state)
...         is_done = False
...
...         while not is_done:
...             next_state, reward, done, _ = env.step(action)
```

```
...             q_values_next = estimator.predict(next_state)
...             next_action = policy(next_state)
...             td_target = reward + gamma * q_values_next[next_action]
...             estimator.update(state, action, td_target)
...             total_reward_episode[episode] += reward
...
...             if done:
...                 break
...             state = next_state
...             action = next_action
```

Функция `sarsa()` выполняет следующие действия:

- в каждом эпизоде создает ε-жадную стратегию, причем ε затухает с коэффициентом 0.99;
- выполняет эпизод: на каждом шаге выбирается действие a, следуя ε-жадной стратегии; в новом состоянии выбирается действие в соответствии с той же ε-жадной стратегией, значения Q-функции для нового состояния вычисляются с использованием текущего оценивателя, затем целевое значение $V(s_t) = r + \gamma \max_{a'} V(s', a')$ используется для обучения оценивателя;
- прогоняет `n_episode` эпизодов и запоминает полные вознаграждения в каждом эпизоде.

4. Зададим количество признаков 200, скорость обучения 0.03 и создадим оцениватель с такими параметрами:

```
>>> n_state = env.observation_space.shape[0]
>>> n_action = env.action_space.n
>>> n_feature = 200
>>> lr = 0.03
>>> estimator = Estimator(n_feature, n_state, n_action, lr)
```

5. Выполним алгоритм SARSA с аппроксимацией функций на 300 эпизодах, запоминая полные вознаграждения в каждом эпизоде.

```
>>> n_episode = 300
>>> total_reward_episode = [0] * n_episode
>>> sarsa(env, estimator, n_episode, epsilon=0.1)
```

6. Построим график зависимости вознаграждения в эпизоде от времени:

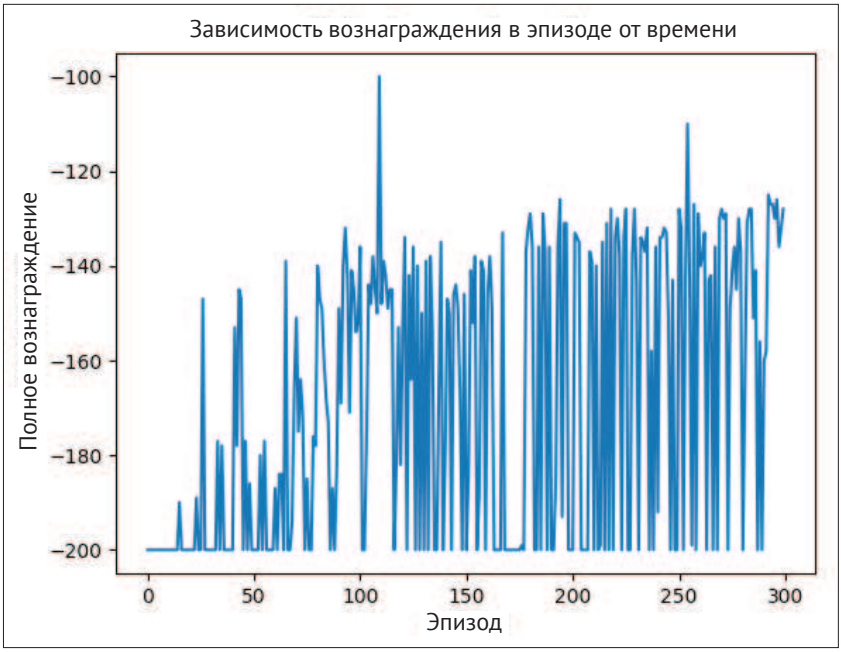
```
>>> import matplotlib.pyplot as plt
>>> plt.plot(total_reward_episode)
>>> plt.title('Зависимость вознаграждения в эпизоде от времени')
>>> plt.xlabel(Эпизод')
>>> plt.ylabel('Полное вознаграждение')
>>> plt.show()
```

Как это работает

SARSA с аппроксимацией функций пытается найти оптимальные веса в моделях аппроксимации, стремясь наилучшим образом оценить значения Q-функции. Он оптимизирует оцениватель, выбирая действия в соответствии с той же

стратегией, в противоположность обучению на опыте, сгенерированном другой стратегией, как в Q-обучении.

После того как модель обучена, нам остается только применить модели регрессии для предсказания ценностей пар состояние–действие для всех возможных действий и выбрать действие с максимальной ценностью в данном состоянии. График, построенный на шаге 6, выглядит следующим образом:



Мы видим, что после первых 25 итераций в большинстве эпизодов автомобилю, чтобы добраться до вершины горы, нужно от 130 до 160 шагов.

Пакетная обработка с применением буфера воспроизведения опыта

В двух предыдущих рецептах мы реализовали два алгоритма обучения с аппроксимацией функций: с разделенной и с единой стратегией. В этом рецепте мы улучшим качество Q-обучения с разделенной стратегией, включив буфер воспроизведения опыта.

Воспроизведение опыта означает, что мы сохраняем опыт агента, накопленный на протяжении эпизода, вместо того чтобы выполнять Q-обучение. Этап обучения с воспроизведением опыта превращается в два этапа: накопление опыта и обновление моделей на основе полученного опыта по завершении эпизода. Точнее, опыт (который также называют буфером, или памятью) включает прошлое состояние, выбранное в нем действие, полученное вознаграждение и следующее состояние для каждого шага эпизода.

На этапе обучения из буфера опыта случайным образом выбирается сколько-то примеров, которые используются для обучения моделей. Воспроизведение опыта может стабилизировать процесс обучения, обеспечив набор слабо коррелированных примеров, а значит, повысится и эффективность обучения.

Как это делается

Применим воспроизведение опыта к Q-обучению с аппроксимацией функций, воспользовавшись линейным оценивателем – классом `Estimator` из файла `linear_estimator.py`, который был создан в рецепте «Оценивание Q-функций посредством аппроксимации методом градиентного спуска».

1. Импортируем необходимые модули и создадим экземпляр окружающей среды Mountain Car:

```
>>> import gym
>>> import torch
>>> from linear_estimator import Estimator
>>> from collections import deque
>>> import random
>>> env = gym.envs.make("MountainCar-v0")
```

2. Воспользуемся той же функцией ε-жадной стратегии, которая была написана в рецепте «Реализация Q-обучения с применением линейной аппроксимации функций».

3. Зададим количество признаков 200, скорость обучения 0.03 и создадим оцениватель с такими параметрами:

```
>>> n_state = env.observation_space.shape[0]
>>> n_action = env.action_space.n
>>> n_feature = 200
>>> lr = 0.03
>>> estimator = Estimator(n_feature, n_state, n_action, lr)
```

4. Определим буфер для хранения опыта:

```
>>> memory = deque(maxlen=400)
```

   Новые примеры будут добавляться в конец очереди, а старые – удаляться из начала, когда количество элементов в очереди превысит 400.

5. Определим функцию, выполняющую Q-обучение с аппроксимацией функций и воспроизведением опыта.

```
>>> def q_learning(env, estimator, n_episode, replay_size,
gamma=1.0, epsilon=0.1, epsilon_decay=.99):
...     """
...     Алгоритм Q-обучения с аппроксимацией функций и воспроизведением опыта
...     @param env: окружающая среда Gym
...     @param estimator: объект класса Estimator
...     @param replay_size: сколько примеров использовать при каждом
...               обновлении модели
...     @param n_episode: количество эпизодов
...     @param gamma: коэффициент обесценивания
```

```
...     @param epsilon: параметр ε-жадной стратегии
...     @param epsilon_decay: коэффициент затухания epsilon
...     """
...     for episode in range(n_episode):
...     policy = gen_epsilon_greedy_policy(estimator,
                       epsilon * epsilon_decay ** episode,
                       n_action)
...     state = env.reset()
...     is_done = False
...     while not is_done:
...         action = policy(state)
...         next_state, reward, is_done, _ = env.step(action)
...         total_reward_episode[episode] += reward
...         if is_done:
...             break
...
...         q_values_next = estimator.predict(next_state)
...         td_target = reward + gamma * torch.max(q_values_next)
...         memory.append((state, action, td_target))
...         state = next_state
...
...     replay_data = random.sample(memory,
                    min(replay_size, len(memory)))
...     for state, action, td_target in replay_data:
...         estimator.update(state, action, td_target)
```

Эта функция выполняет следующие действия:

- в каждом эпизоде создает ε-жадную стратегию, причем ε затухает с коэффициентом 0.99 (если в первом эпизоде ε было равно 0.1, то во втором будет равно 0.099);
- выполняет эпизод: на каждом шаге выбирается действие a, следуя ε-жадной стратегии; значения Q-функции для нового состояния вычисляются с использованием текущего оценивателя, затем вычисляется целевое значение $V(s_t) = r + \gamma \max_{a'} V(s')$ и в буфере воспроизведения сохраняется кортеж, состоящий из состояния, действия и целевого значения;
- после каждого эпизода из буфера случайным образом выбирается `replay_size` примеров, которые используются для обучения оценивателя;
- прогоняет `n_episode` эпизодов и запоминает полные вознаграждения в каждом эпизоде.

6. Выполним Q-обучение с буфером воспроизведения на 1000 эпизодов:

```
>>> n_episode = 1000
```

Больше эпизодов нужно просто потому, что модели недостаточно обучены, поэтому агент выполняет больше случайных шагов в ранних эпизодах.

Зададим размер буфера воспроизведения 190:

```
>>> replay_size = 190
```

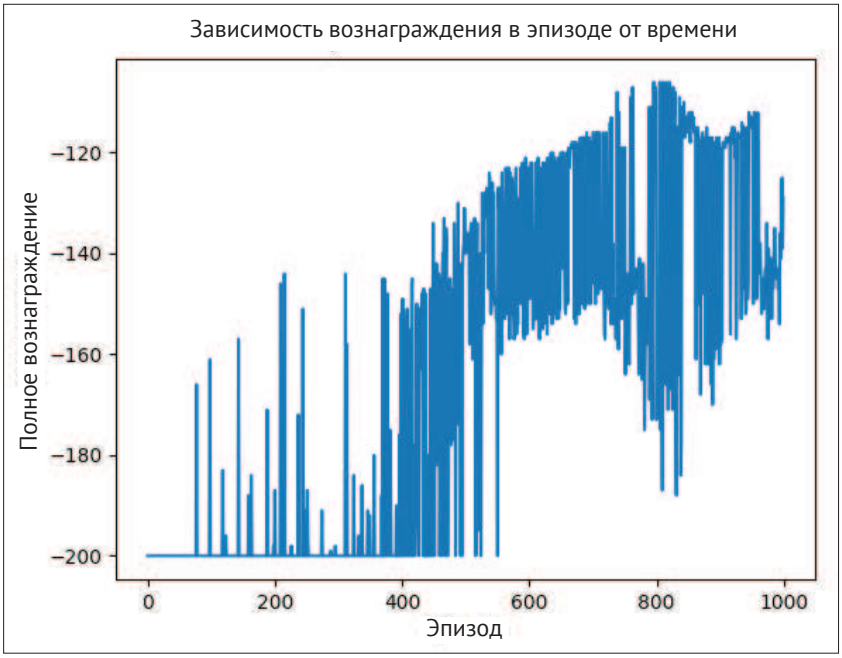
Будем запоминать полные вознаграждения в каждом эпизоде:

```
>>> total_reward_episode = [0] * n_episode
>>> q_learning(env, estimator, n_episode, replay_size, epsilon=0.1)
```

7. Построим график зависимости вознаграждения в эпизоде от времени:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(total_reward_episode)
>>> plt.title('Зависимость вознаграждения в эпизоде от времени')
>>> plt.xlabel('Эпизод')
>>> plt.ylabel('Полное вознаграждение')
>>> plt.show()
```

Построенный график выглядит следующим образом:



Мы видим, что при наличии буфера воспроизведения Q-обучение становится гораздо устойчивее. После первых 500 итераций вознаграждения в большинстве эпизодов колеблются в диапазоне от –160 до –120.

Как это работает

В этом рецепте мы решили задачу о машине на горе с помощью Q-обучения с аппроксимацией функций и буфером воспроизведения опыта. Новое решение превосходит Q-обучение без буфера воспроизведения, поскольку, вместо того чтобы сразу приниматься за обучение оценивателя, мы сначала накапливаем в буфере данные, наблюдавшиеся в эпизодах, а затем производим слу-

чайные выборки пакетов примеров из буфера и на них обучаем оцениватель. В результате примеры во входном наборе данных меньше зависят друг от друга, поэтому обучение оказывается более устойчивым и эффективным.

Реализация Q-обучения с аппроксимацией функций нейронной сетью

Выше мы уже говорили, что для аппроксимации можно использовать и нейронные сети. В этом рецепте мы решим задачу о машине на горе, применив Q-обучение с аппроксимацией нейронной сетью.

Цель аппроксимации функций – взять набор признаков и оценить Q-функцию с помощью модели регрессии. Используя для оценивания нейронную сеть, мы повышаем гибкость регрессии (благодаря нескольким слоям сети) и вводим нелинейность (нелинейную функцию активации в скрытых слоях). В остальных чертах Q-обучение производится примерно так же, как в случае линейной аппроксимации. Для обучения сети используется метод градиентного спуска. Конечная цель обучения – найти оптимальные веса сети, при которых достигается наилучшая аппроксимация функции ценности состояний $V(s)$ для всех возможных действий. В качестве минимизируемой функции потерь по-прежнему выступает среднеквадратическая ошибка между истинным значением и его оценкой.

Как это делается

Начнем с реализации оценивателя на основе нейронной сети. Мы воспользуемся многими частями линейного оценивателя, разработанного в рецепте «Оценивание Q-функций посредством аппроксимации методом градиентного спуска». Различие лишь в том, что входной и выходной слои соединены со скрытым слоем с функцией активации ReLU (блок линейной ректификации). Поэтому изменить нужно только метод `__init__`:

```
>>> class Estimator():
...     def __init__(self, n_feat, n_state, n_action, lr=0.05):
...     self.w, self.b = self.get_gaussian_wb(n_feat, n_state)
...     self.n_feat = n_feat
...     self.models = []
...     self.optimizers = []
...     self.criterion = torch.nn.MSELoss()
...     for _ in range(n_action):
...         model = torch.nn.Sequential(
...                     torch.nn.Linear(n_feat, n_hidden),
...                     torch.nn.ReLU(),
...                     torch.nn.Linear(n_hidden, 1)
...         )
...         self.models.append(model)
...         optimizer = torch.optim.Adam(model.parameters(), lr)
...         self.optimizers.append(optimizer)
```

Как видим, скрытый слой содержит n_hidden блоков и наделен функцией активации ReLU, torch.nn.ReLU(), а за ним следует выходной слой, порождающий оценку.

Остальные части класса Estimator с нейронной сетью такие же, как в линейном оценивателе. Можете скопировать их из файла nn_estimator.py.

Теперь можно перейти собственно к Q-обучению с буфером воспроизведения опыта.

1. Импортируем необходимые модули, в т. ч. только что написанный класс оценивателя на основе нейронной сети Estimator из файла nn_estimator.py, и создадим экземпляр окружающей среды Mountain Car:

```
>>> import gym
>>> import torch
>>> from nn_estimator import Estimator
>>> from collections import deque
>>> import random
>>> env = gym.envs.make("MountainCar-v0")
```

2. Воспользуемся той же функцией ε-жадной стратегии, которая была написана в рецепте «Реализация Q-обучения с применением линейной аппроксимации функций».

3. Зададим количество признаков 200, скорость обучения 0.001, размер скрытого слоя 50 и создадим оцениватель с такими параметрами:

```
>>> n_state = env.observation_space.shape[0]
>>> n_action = env.action_space.n
>>> n_feature = 200
>>> n_hidden = 50
>>> lr = 0.001
>>> estimator = Estimator(n_feature, n_state, n_action, n_hidden, lr)
```

4. Определим буфер для хранения опыта:

```
>>> memory = deque(maxlen=300)
```

   Новые примеры будут добавляться в конец очереди, а старые – удаляться из начала, когда количество элементов в очереди превысит 300.

5. Воспользуемся той же функцией обучения q_learning, которая была разработана в предыдущем рецепте. Она выполняет Q-обучение с аппроксимацией функций и буфером воспроизведения опыта.

6. Выполним Q-обучение с буфером воспроизведения на 1000 эпизодов и зададим размер выборки из буфера 200:

```
>>> n_episode = 1000
>>> replay_size = 200
```

   Будем запоминать полное вознаграждение в каждом эпизоде:

```
>>> total_reward_episode = [0] * n_episode
>>> q_learning(env, estimator, n_episode, replay_size, epsilon=0.1)
```

7. Построим график зависимости вознаграждения в эпизоде от времени:

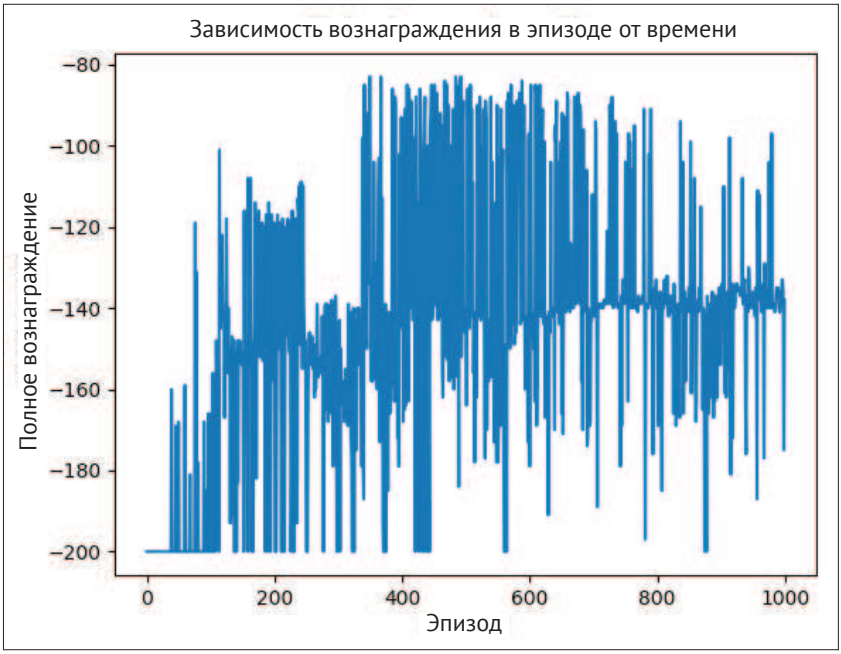
```
>>> import matplotlib.pyplot as plt
```

```
>>> plt.plot(total_reward_episode)
>>> plt.title('Зависимость вознаграждения в эпизоде от времени')
>>> plt.xlabel(Эпизод')
>>> plt.ylabel('Полное вознаграждение')
>>> plt.show()
```

Как это работает

Аппроксимация функций нейронными сетями очень похожа на линейную аппроксимацию, только вместо простой линейной функции для отображения признаков на целевые значения применяется нейронная сеть. В остальном алгоритм такой же, но благодаря сложной архитектуре нейронной сети и нелинейной функции активации он обладает большей гибкостью, а значит, и лучшей предсказательной способностью.

Построенный на шаге 7 график выглядит следующим образом:



Мы видим, что качество Q-обучения с нейронной сетью выше, чем при использовании линейной функции. После первых 500 итераций вознаграждения в большинстве эпизодов колеблются в диапазоне от –140 до –85.

См. также

Читателям, желающим освежить знания о нейронных сетях, рекомендуем следующие источники:

- https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html;
- https://www.cs.toronto.edu/~jlucas/teaching/csc411/lectures/tut5_handout.pdf.

Решение задачи о балансировании стержня с помощью аппроксимации функций

В этом бонусном рецепте мы решим задачу о балансировании стержня (CartPole), применив аппроксимацию функций.

В главе 1 мы уже решали эту задачу с помощью случайного поиска, а также алгоритмов восхождения на вершину и градиентной стратегии. А сейчас мы применим к ней знания, полученные в этой главе.

Как это делается

Продемонстрируем решение с применением аппроксимации функций нейронной сетью без буфера воспроизведения опыта.

1. Импортируем необходимые модули, в т. ч. только что написанный класс оценивателя на основе нейронной сети `Estimator` из файла `nn_estimator.py`, и создадим экземпляр окружающей среды CartPole:

```
>>> import gym
>>> import torch
>>> from nn_estimator import Estimator
>>> env = gym.envs.make("CartPole-v0")
```

2. Воспользуемся той же функцией ε-жадной стратегии, которая была написана в рецепте «Реализация Q-обучения с применением линейной аппроксимации функций».

3. Зададим количество признаков 400 (отметим, что пространство состояний в окружающей среде CartPole 4-мерное), скорость обучения 0.01, размер скрытого слоя 100 и создадим оцениватель с такими параметрами:

```
>>> n_state = env.observation_space.shape[0]
>>> n_action = env.action_space.n
>>> n_feature = 400
>>> n_hidden = 100
>>> lr = 0.01
>>> estimator = Estimator(n_feature, n_state, n_action, n_hidden, lr)
```

4. Воспользуемся той же функцией обучения `q_learning`, которая была разработана в рецепте «Реализация Q-обучения с применением линейной аппроксимации функций». Она выполняет Q-обучение с аппроксимацией функций.
5. Выполним Q-обучение с аппроксимацией функций на 1000 эпизодов и будем запоминать полное вознаграждение в каждом эпизоде:

```
>>> n_episode = 1000
>>> total_reward_episode = [0] * n_episode
>>> q_learning(env, estimator, n_episode, epsilon=0.1)
```

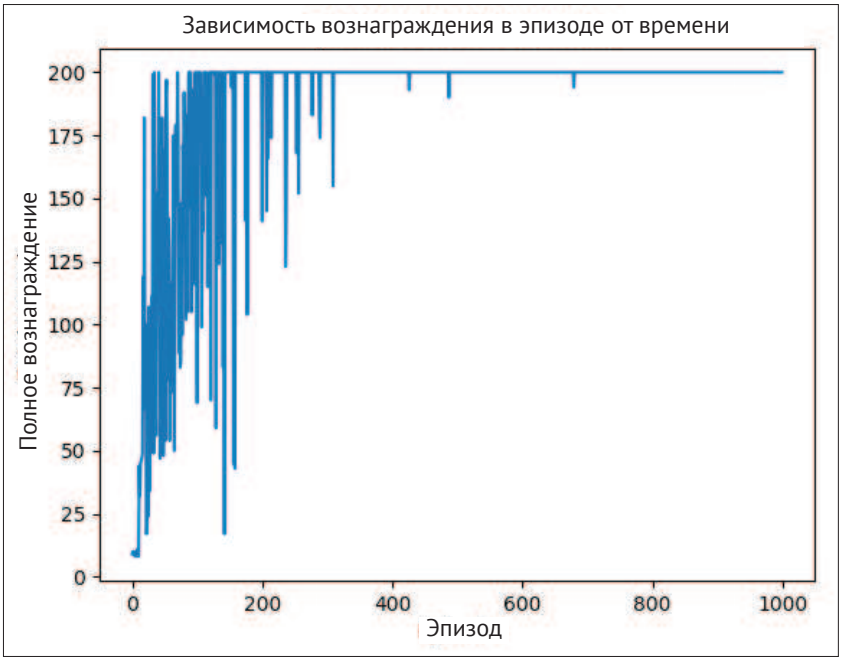
6. И наконец, построим график зависимости вознаграждения в эпизоде от времени:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(total_reward_episode)
>>> plt.title('Зависимость вознаграждения в эпизоде от времени')
>>> plt.xlabel(Эпизод')
>>> plt.ylabel('Полное вознаграждение')
>>> plt.show()
```

Как это работает

Мы решили задачу CartPole, применив алгоритм с аппроксимацией функций нейронной сетью, разработанный в предыдущем рецепте. Отметим, что пространство наблюдений в этой среде 4-мерное, т. е. его размерность в два раза больше, чем в среде Mountain Car, поэтому мы, руководствуясь интуицией, вдвое увеличили количество признаков и размер скрытого слоя. Поэкспериментируйте с алгоритмом SARSA и Q-обучением с буфером воспроизведения и посмотрите, какой из них окажется лучше.

Построенный на шаге 6 график выглядит следующим образом:



Мы видим, что после первых 300 итераций вознаграждение в большинстве эпизодов максимально и равно +200.

Глава 7

Глубокие Q-сети в действии

Глубокое Q-обучение, в котором используются глубокие Q-сети, считается самой современной технологией обучения с подкреплением. В этой главе мы разработаем различные модели на основе глубоких Q-сетей и применим их к решению нескольких задач ОП. Начнем с простых Q-сетей, а затем дополним их воспроизведением опыта. Чтобы повысить робастность, мы введем дополнительную целевую сеть и покажем, как настраивается глубокая Q-сеть. Мы также поэкспериментируем с дуэльными Q-сетями и увидим, чем их функция ценности отличается от других типов глубоких Q-сетей. В двух последних рецептах мы решим трудные задачи, связанные с играми Atari, включив сверточную нейронную сеть в глубокую Q-сеть.

В этой главе приводятся следующие рецепты:

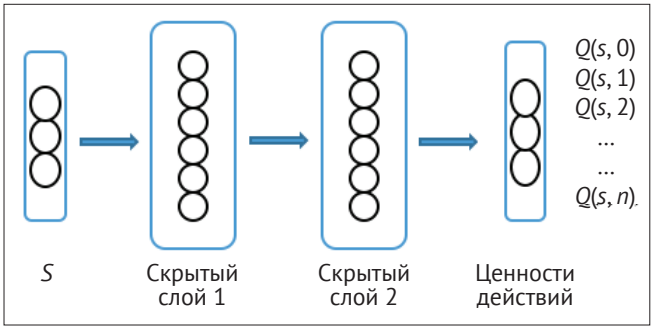
- реализация глубоких Q-сетей;
- улучшение DQN с помощью воспроизведения опыта;
- реализация алгоритма Double DQN;
- настройка гиперпараметров алгоритма Double DQN для среды CartPole;
- реализация алгоритма Dueling DQN;
- применение DQN к играм Atari;
- использование сверточных нейронных сетей в играх Atari.

Реализация глубоких Q-сетей

Напомним, что применение **аппроксимации функций** позволяет заменить пространство состояний набором признаков, порождаемых по исходным состояниям. **Глубокие Q-сети** (Deep Q-Network – DQN) очень похожи на аппроксимацию нейронными сетями, но нейронная сеть в них используется для прямого отображения состояний на ценности действий без использования промежуточных сгенерированных признаков.

В глубоком Q-обучении нейронная сеть обучается выводить значения $Q(s, a)$ для каждого действия, зная входное состояние s. Действие агента a выбирается

на основе порождаемых значений $Q(s, a)$, следуя ε-жадной стратегии. Архитектура DQN с двумя скрытыми слоями показана на рисунке ниже:



Напомним, что Q-обучение – это алгоритм обучения с разделенной стратегией, в котором Q-функция обновляется по формуле:

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

где s' – состояние, в которое переходит среда из состояния s после действия a, r – полученное при этом вознаграждение, α – скорость обучения, γ – коэффициент обесценивания. Наличие члена $\max_{a'} Q(s', a')$ означает, что поведенческая стратегия жадная, т. е. для генерации обучающих данных выбирается действие в состоянии s' с наибольшим значением Q-функции. DQN обучается минимизировать член ошибки

$$\delta = r + \gamma \max_{a'} Q(s') - Q(s).$$

Таким образом, наша цель – найти оптимальную модель сети, при которой достигается наилучшая аппроксимация функции ценности состояний $Q(s, a)$ для всех возможных действий. Минимизируемая функция потерь, как и в задаче регрессии, – среднеквадратическая ошибка между истинным значением и его оценкой.

Разработаем модель DQN для решения задачи о машине на горе (https://gym.openai.com/envs/MountainCar-v0/).

Как это делается

Реализуем глубокое Q-обучение с помощью сети DQN.

1. Импортируем необходимые пакеты:

```
>>> import gym
>>> import torch
>>> from torch.autograd import Variable
>>> import random
```

   Модуль `Variable` обертывает тензор и поддерживает обратное распространение.

2. Начнем с метода `__init__` класса DQN:

```
>>> class DQN():
...     def __init__(self, n_state, n_action, n_hidden=50, lr=0.05):
...         self.criterion = torch.nn.MSELoss()
...         self.model = torch.nn.Sequential(
...             torch.nn.Linear(n_state, n_hidden),
...             torch.nn.ReLU(),
...             torch.nn.Linear(n_hidden, n_action)
...     )
...     self.optimizer = torch.optim.Adam(self.model.parameters(), lr)
```

3. Теперь напишем метод обучения, который обновляет нейронную сеть, получив новый пример.

```
>>>     def update(self, s, y):
...         """
...         Обновляет веса DQN, получив обучающий пример
...         @param s: состояние
...         @param y: целевое значение
...         """
...         y_pred = self.model(torch.Tensor(s))
...         loss = self.criterion(y_pred, Variable(torch.Tensor(y)))
...         self.optimizer.zero_grad()
...         loss.backward()
...         self.optimizer.step()
```

4. Далее следует функция предсказания ценности состояния:

```
>>>     def predict(self, s):
...         """
...         Вычисляет значения Q-функции состояния для всех действий,
...         применяя обученную модель
...         @param s: входное состояние
...         @return: значения Q для всех действий
...         """
...         with torch.no_grad():
...             return self.model(torch.Tensor(s))
```

   С классом `DQN` – все! Теперь можно заняться алгоритмом обучения.

5. Создадим экземпляр окружающей среды Mountain Car:

```
>>> env = gym.envs.make("MountainCar-v0")
```

6. Определим ε-жадную стратегию:

```
>>> def gen_epsilon_greedy_policy(estimator, epsilon, n_action):
...     def policy_function(state):
...         if random.random() < epsilon:
...             return random.randint(0, n_action - 1)
...         else:
...             q_values = estimator.predict(state)
...         return torch.argmax(q_values).item()
...     return policy_function
```

7. Определим алгоритм Q-обучения с применением DQN:

```
>>> def q_learning(env, estimator, n_episode, gamma=1.0,
epsilon=0.1, epsilon_decay=.99):
...     """
...     Глубокое Q-обучение с применением DQN
...     @param env: имя окружающей среды Gym
...     @param estimator: объект класса Estimator
...     @param n_episode: количество эпизодов
...     @param gamma: коэффициент обесценивания
...     @param epsilon: параметр ε-жадной стратегии
...     @param epsilon_decay: коэффициент затухания epsilon
...     """
...     for episode in range(n_episode):
...         policy = gen_epsilon_greedy_policy(estimator, epsilon, n_action)
...         state = env.reset()
...         is_done = False
...         while not is_done:
...             action = policy(state)
...             next_state, reward, is_done, _ = env.step(action)
...             total_reward_episode[episode] += reward
...             modified_reward = next_state[0] + 0.5
...             if next_state[0] >= 0.5:
...                 modified_reward += 100
...             elif next_state[0] >= 0.25:
...                 modified_reward += 20
...             elif next_state[0] >= 0.1:
...                 modified_reward += 10
...             elif next_state[0] >= 0:
...                 modified_reward += 5
...
...             q_values = estimator.predict(state).tolist()
...
...             if is_done:
...                 q_values[action] = modified_reward
...                 estimator.update(state, q_values)
...                 break
...             q_values_next = estimator.predict(next_state)
...             q_values[action] = modified_reward + gamma *
                        torch.max(q_values_next).item()
...             estimator.update(state, q_values)
...             state = next_state
...         print('Эпизод: {}, полное вознаграждение: {}, epsilon:{}'.
                format(episode,
                       total_reward_episode[episode], epsilon))
...         epsilon = max(epsilon * epsilon_decay, 0.01)
```

8. Зададим размер скрытого слоя и скорость обучения и создадим экземпляр класса `DQN` с такими параметрами:

```
>>> n_state = env.observation_space.shape[0]
>>> n_action = env.action_space.n
>>> n_hidden = 50
```

```
>>> lr = 0.001
>>> dqn = DQN(n_state, n_action, n_hidden, lr)
```

9. Выполним глубокое Q-обучение, применяя только что разработанный алгоритм DQN, на 1000 эпизодов и будем запоминать полные вознаграждения (до обесценивания) в каждом эпизоде:

```
>>> n_episode = 1000
>>> total_reward_episode = [0] * n_episode
>>> q_learning(env, dqn, n_episode, gamma=.99, epsilon=.3)
Эпизод: 0, полное вознаграждение: -200.0, epsilon: 0.3
Эпизод: 1, полное вознаграждение: -200.0, epsilon: 0.297
Эпизод: 2, полное вознаграждение: -200.0, epsilon: 0.29402999999999996
.....
.....
Эпизод: 993, полное вознаграждение: -177.0, epsilon: 0.01
Эпизод: 994, полное вознаграждение: -200.0, epsilon: 0.01
Эпизод: 995, полное вознаграждение: -172.0, epsilon: 0.01
Эпизод: 996, полное вознаграждение: -200.0, epsilon: 0.01
Эпизод: 997, полное вознаграждение: -200.0, epsilon: 0.01
Эпизод: 998, полное вознаграждение: -173.0, epsilon: 0.01
Эпизод: 999, полное вознаграждение: -200.0, epsilon: 0.01
```

10. Построим график зависимости вознаграждения в эпизоде от времени:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(total_reward_episode)
>>> plt.title('Зависимость вознаграждения в эпизоде от времени')
>>> plt.xlabel('Эпизод')
>>> plt.ylabel('Полное вознаграждение')
>>> plt.show()
```

Как это работает

Метод `__init__` класса `DQN` (шаг 2) принимает четыре параметра: количество состояний во входном слое, количество действий в выходном слое, количество скрытых блоков (в данном примере скрытый слой всего один) и скорость обучения. Он инициализирует нейронную сеть с одним скрытым слоем и функцией активации ReLU. Сеть имеет `n_state` входов и порождает `n_action` выходов – предсказанные ценности состояний при выборе каждого действия. В качестве оптимизатора используется алгоритм Adam, а в качестве функции потерь – среднеквадратическая ошибка.

На шаге 3 производится обновление сети: зная исходное состояние и соответствующее ему целевое значение, метод вычисляет потерю и градиенты. Затем модель нейронной сети обновляется методом обратного распространения.

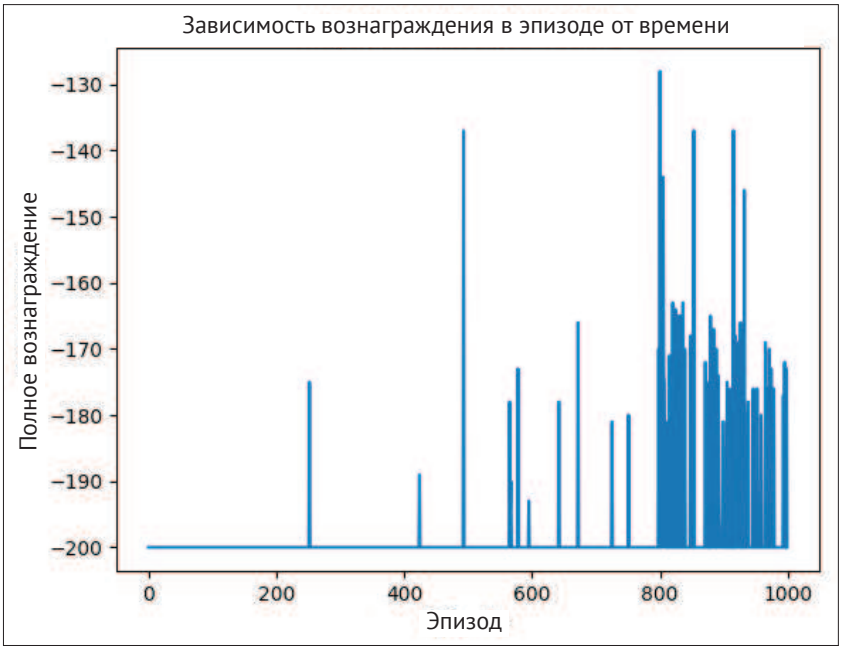
На шаге 7 функция глубокого Q-обучения выполняет следующие действия:

- в каждом эпизоде создает ε-жадную стратегию, причем ε затухает с коэффициентом 0.99 (если в первом эпизоде ε было равно 0.1, то во втором будет равно 0.099). Задается также нижний предел ε, равный 0.01;

- выполняет эпизод: на каждом шаге выбирается действие a в состоянии s, следуя ε-жадной стратегии; значения Q-функции `q_values` в предыдущем состоянии вычисляются с использованием метода `predict` класса `DQN`;
- вычисляет значения Q-функции `q_values_next` для нового состояния s', затем вычисляет целевое значение, обновляя старые значения `q_values` для действия по формуле $Q(s, a) = r + \gamma \max_{a'} Q(s', a')$;
- использует пример $(s, Q(s))$ для обучения нейронной сети. Отметим, что $Q(s)$ включает ценности для всех действий;
- прогоняет `n_episode` эпизодов и запоминает полные вознаграждения в каждом эпизоде.

Обратите внимание, что при обучении модели используется модифицированная версия вознаграждения. Оно зависит от положения автомобиля, поскольку мы хотим добраться до вершины, расположенной в точке с абсциссой +0.5. Поэтому мы используем пороговую систему поощрений с порогами +0.5, +0.25, +0.1 и 0 – чем больше порог, тем выше вознаграждение. При такой системе поощрения предпочтение отдается позициям, которые ближе к цели, поэтому скорость обучения заметно больше по сравнению с исходным постоянным вознаграждением –1 на каждом шаге.

Наконец, график, построенный на шаге 10, выглядит следующим образом:



Мы видим, что в последних 200 эпизодах автомобиль добирается вершины горы за 170–180 шагов.

Глубокое Q-обучение использует для аппроксимации ценности состояний более прямолинейную модель, нейронную сеть, отказываясь от набора промежуточных искусственных признаков. На одном шаге, когда среда переходит

из старого состояния в новое после выбора агентом некоторого действия и начисляет за это вознаграждение, обучение DQN состоит из следующих стадий:

- использовать модель нейронной сети для оценки значений Q старого состояния;
- использовать модель нейронной сети для оценки значений Q нового состояния;
- обновить целевое значение Q для действия с учетом вознаграждения и новых значений Q по формуле $r + \gamma \, max_{a'} \, Q(s', a')$;
- заметим, что в заключительном состоянии целевое значение Q принимается равным r;
- обучить нейронную сеть, приняв в качестве входа старое состояние, а в качестве выхода целевые значения Q.

Веса сети обновляются методом градиентного спуска. Обученная сеть может предсказывать значения Q для заданного состояния.

DQN значительно уменьшает количество обучаемых состояний, и это замечательно, потому что обучение на миллионах состояний с помощью TD-методов неосуществимо. Кроме того, этот алгоритм напрямую отображает входное состояние на значения Q-функции, обходясь без дополнительных функций для генерации искусственных признаков.

См. также

Читателям, незнакомым с методом оптимизации на основе градиентного спуска Adam, рекомендуем следующие источники:

- https://towardsdatascience.com/adam-latest-trends-in-deep-learningoptimization-6be9a291375c;
- https://arxiv.org/abs/1412.6980.

Улучшение DQN с помощью воспроизведения опыта

Аппроксимация значений Q-функции нейронной сетью, обучаемой на одном примере за раз, ведет себя не очень устойчиво. Напомним, что раньше для повышения устойчивости мы включили буфер воспроизведения опыта. А сейчас применим эту идею к DQN.

Идея воспроизведения опыта заключается в том, чтобы сохранять в памяти опыт агента (старое состояние, новое состояние, действие и вознаграждение), накопленный на протяжении эпизодов в сеансе обучения. Набрав достаточно опыта, мы производим случайную выборку пакета примеров из буфера в памяти и используем его для обучения нейронной сети. Обучение с буфером воспроизведения состоит из двух этапов: накопление опыта и обновление модели на случайной выборке из прошлого опыта. Выборка должна быть случайной, потому что иначе модель обучалась бы на самом недавнем опыте, и нейронная сеть могла бы застрять в локальном минимуме.

Ниже мы разработаем DQN с буфером воспроизведения для решения задачи о машине на горе.

Как это делается

1. Импортируем необходимые модули и создадим экземпляр окружающей среды Mountain Car.

```
>>> import gym
>>> import torch
>>> from collections import deque
>>> import random
>>> from torch.autograd import Variable
>>> env = gym.envs.make("MountainCar-v0")
```

2. Для реализации буфера воспроизведения добавим в класс `DQN` метод `replay`:

```
>>> def replay(self, memory, replay_size, gamma):
...     """
...     Воспроизведение опыта
...     @param memory: буфер воспроизведения опыта
...     @param replay_size: сколько примеров использовать при каждом
...                обновлении модели
...     @param gamma: коэффициент обесценивания
...     """
...     if len(memory) >= replay_size:
...         replay_data = random.sample(memory, replay_size)
...         states = []
...         td_targets = []
...         for state, action, next_state, reward, is_done in replay_data:
...             states.append(state)
...             q_values = self.predict(state).tolist()
...             if is_done:
...                 q_values[action] = reward
...             else:
...                 q_values_next = self.predict(next_state)
...                 q_values[action] = reward + gamma *
                        torch.max(q_values_next).item()
...             td_targets.append(q_values)
...
...         self.update(states, td_targets)
```

   Больше в классе ничего не меняется.

3. Мы воспользуемся функцией `gen_epsilon_greedy_policy`, разработанной в рецепте «Реализация глубоких Q-сетей», и не будем здесь повторять ее код.

4. Зададим форму нейронной сети – размеры входного, выходного и скрытого слоев, скорость обучения 0.001 и создадим экземпляр класса `DQN`:

```
>>> n_state = env.observation_space.shape[0]
>>> n_action = env.action_space.n
>>> n_hidden = 50
>>> lr = 0.001
>>> dqn = DQN(n_state, n_action, n_hidden, lr)
```

5. Определим буфер для хранения опыта:

```
>>> memory = deque(maxlen=10000)
```

Новые примеры будут добавляться в конец очереди, а старые – удаляться из начала, когда количество элементов в очереди превысит 10 000.

6. Определим функцию, выполняющую глубокое Q-обучение с воспроизведением опыта.

```
>>> def q_learning(env, estimator, n_episode, replay_size,
gamma=1.0, epsilon=0.1, epsilon_decay=.99):
...     """
...     Глубокое Q-обучение методом DQN с воспроизведением опыта
...     @param env: имя окружающей среды Gym
...     @param estimator: объект класса Estimator
...     @param replay_size: сколько примеров использовать при каждом
...               обновлении модели
...     @param n_episode: количество эпизодов
...     @param gamma: коэффициент обесценивания
...     @param epsilon: параметр ε-жадной стратегии
...     @param epsilon_decay: коэффициент затухания epsilon
...     """
...     for episode in range(n_episode):
...         policy = gen_epsilon_greedy_policy(estimator, epsilon, n_action)
...         state = env.reset()
...         is_done = False
...         while not is_done:
...             action = policy(state)
...             next_state, reward, is_done, _ = env.step(action)
...             total_reward_episode[episode] += reward
...             modified_reward = next_state[0] + 0.5
...             if next_state[0] >= 0.5:
...                 modified_reward += 100
...             elif next_state[0] >= 0.25:
...                 modified_reward += 20
...             elif next_state[0] >= 0.1:
...                 modified_reward += 10
...             elif next_state[0] >= 0:
...                 modified_reward += 5
...             memory.append((state, action, next_state,
                        modified_reward, is_done))
...             if is_done:
...                 break
...             estimator.replay(memory, replay_size, gamma)
...             state = next_state
...         print('Эпизод: {}, полное вознаграждение: {}, epsilon:{}'.
                format(episode,
                       total_reward_episode[episode], epsilon))
...         epsilon = max(epsilon * epsilon_decay, 0.01)
```

7. Выполним глубокое Q-обучение с буфером воспроизведения на 600 эпизодов:

```
>>> n_episode = 600
```

Зададим размер выборки из буфера воспроизведения на каждом шаге:

```
>>> replay_size = 20
```

Будем запоминать полные вознаграждения в каждом эпизоде:

```
>>> total_reward_episode = [0] * n_episode
>>> q_learning(env, dqn, n_episode, replay_size, gamma=.9, epsilon=.3)
```

8. Построим график зависимости вознаграждения в эпизоде от времени:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(total_reward_episode)
>>> plt.title('Зависимость вознаграждения в эпизоде от времени')
>>> plt.xlabel('Эпизод')
>>> plt.ylabel('Полное вознаграждение')
>>> plt.show()
```

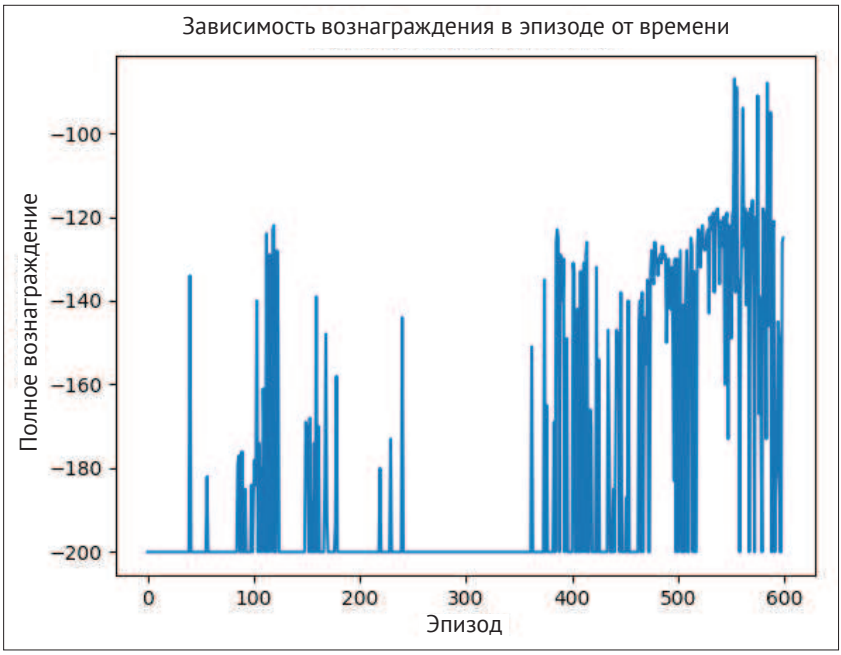
Как это работает

На шаге 2 функция `replay` сначала производит случайную выборку размера `replay_size` из буфера воспроизведения опыта. Затем каждый элемент выборки преобразуется в обучающий пример, состоящий из входного состояния и выходного целевого значения. После этого получившийся пакет используется для обновления нейронной сети.

На шаге 6 функция глубокого Q-обучения с воспроизведением опыта выполняет следующие действия:

- в каждом эпизоде создает ε-жадную стратегию, причем ε затухает с коэффициентом 0.99;
- выполняет эпизод: на каждом шаге выбирается действие *a*, следуя ε-жадной стратегии, опыт (старое состояние, действие, новое состояние, вознаграждение) сохраняется в памяти;
- на каждом шаге запомненный опыт используется для обучения нейронной сети, при условии что в буфере достаточно обучающих примером для выборки;
- прогоняет `n_episode` эпизодов и запоминает полные вознаграждения в каждом эпизоде.

График, построенный на шаге 8, выглядит следующим образом:



Мы видим, что в большинстве из последних 200 эпизодов автомобиль добирается до вершины горы за 120–160 шагов.

В глубоком Q-обучении с воспроизведением опыта мы сохраняем опыт агента на каждом шаге и для обучения DQN производим случайные выборки из прошлого опыта. Обучение в этом случае состоит из двух этапов: накопление опыта и обновление модели на основе этого опыта. Точнее, опыт (который также называют **буфером**, или **памятью**) включает прошлое состояние, выбранное в нем действие, полученное вознаграждение и следующее состояние для каждого шага эпизода. Воспроизведение опыта может стабилизировать процесс обучения, обеспечив набор слабо коррелированных примеров, а значит, повысится и эффективность обучения.

Реализация алгоритма Double DQN

В разработанных выше алгоритмах глубокого Q-обучения для вычисления предсказанных и целевых значений используется одна и та же нейронная сеть. Это может стать причиной расходимости, потому что целевые значения все время изменяются, а предсказания должны следовать за ними по пятам. В этом рецепте мы разработаем новый алгоритм, в котором будет две нейронные сети вместо одной.

В алгоритме **двойной DQN** (Double DQN) для оценивания целевых значений используется отдельная сеть, которая имеет такую же структуру, как сеть для

предсказания. Но ее веса изменяются только после каждых T эпизодов (здесь T – настраиваемый гиперпараметр). Обновление сводится просто к копированию весов предсказательной сети. Таким образом, целевая функция в течение некоторого времени остается неизменной, что повышает устойчивость процесса обучения.

Математически обучение двойной DQN означает минимизацию следующего члена ошибки:

$$\delta = r + \gamma \max_{a'} Q_T(s') - Q(s),$$

где s' – состояние, в которое переходит среда из состояния s после действия a, r – полученное при этом вознаграждение, α – скорость обучения, γ – коэффициент обесценивания, Q_T – функция ценности для целевой сети, а Q – для предсказательной сети.

Теперь решим задачу о машине на горе методом Double DQN.

Как это делается

1. Импортируем необходимые модули и создадим экземпляр окружающей среды Mountain Car.

```
>>> import gym
>>> import torch
>>> from collections import deque
>>> import random
>>> import copy
>>> from torch.autograd import Variable
>>> env = gym.envs.make("MountainCar-v0")
```

2. Прежде чем включать буфер воспроизведения в целевую сеть, инициализируем ее в методе `__init__` класса `DQN`:

```
>>> class DQN():
...      def __init__(self, n_state, n_action, n_hidden=50, lr=0.05):
...         self.criterion = torch.nn.MSELoss()
...         self.model = torch.nn.Sequential(
...                     torch.nn.Linear(n_state, n_hidden),
...                     torch.nn.ReLU(),
...                     torch.nn.Linear(n_hidden, n_action)
...         )
...     self.optimizer = torch.optim.Adam(self.model.parameters(), lr)
...     self.model_target = copy.deepcopy(self.model)
```

   Целевая сеть имеет такую же структуру, как предсказательная.

3. Добавим вычисление ценностей с помощью целевой сети:

```
>>>     def target_predict(self, s):
...         """
...         Вычисляет значения Q-функции состояния для всех действий
...         с помощью целевой сети
...         @param s: входное состояние
```

```
...         @return: целевые ценности состояния для всех действий
...         """
...         with torch.no_grad():
...             return self.model_target(torch.Tensor(s))
```

4. Добавим метод для синхронизации весов целевой и предсказательной сетей:

```
>>>     def copy_target(self):
...         self.model_target.load_state_dict(self.model.state_dict())
```

5. Для вычисления целевой ценности будем использовать целевую, а не предсказательную сеть:

```
>>>     def replay(self, memory, replay_size, gamma):
...         """
...         Буфер воспроизведения совместно с целевой сетью
...         @param memory: буфер воспроизведения опыта
...         @param replay_size: сколько примеров использовать при каждом
...               обновлении модели
...         @param gamma: коэффициент обесценивания
...         """
...         if len(memory) >= replay_size:
...             replay_data = random.sample(memory, replay_size)
...             states = []
...             td_targets = []
...             for state, action, next_state, reward, is_done in replay_data:
...                 states.append(state)
...                 q_values = self.predict(state).tolist()
...                 if is_done:
...                     q_values[action] = reward
...                 else:
...                     q_values_next = self.target_predict(next_state).detach()
...                     q_values[action] = reward + gamma *
                                torch.max(q_values_next).item()
...
...                 td_targets.append(q_values)
...
...             self.update(states, td_targets)
```

Больше в классе `DQN` ничего не меняется.

6. Мы воспользуемся функцией `gen_epsilon_greedy_policy`, разработанной в рецепте «Реализация глубоких Q-сетей», и не будем здесь повторять ее код.

7. Зададим форму нейронной сети – размеры входного, выходного и скрытого слоев, скорость обучения 0.01 и создадим экземпляр класса `DQN`:

```
>>> n_state = env.observation_space.shape[0]
>>> n_action = env.action_space.n
>>> n_hidden = 50
>>> lr = 0.01
>>> dqn = DQN(n_state, n_action, n_hidden, lr)
```

8. Определим буфер для хранения опыта:

```
>>> memory = deque(maxlen=10000)
```

Новые примеры будут добавляться в конец очереди, а старые – удаляться из начала, когда количество элементов в очереди превысит 10 000.

9. Определим функцию, выполняющую глубокое Q-обучение методом Double DQN:

```
>>> def q_learning(env, estimator, n_episode, replay_size,
target_update=10, gamma=1.0, epsilon=0.1, epsilon_decay=.99):
...     """
...     Глубокое Q-обучение методом Double DQN с воспроизведением опыта
...     @param env: имя окружающей среды Gym
...     @param estimator: объект класса DQN
...     @param replay_size: сколько примеров использовать при каждом
...               обновлении модели
...     @param target_update: через сколько эпизодов обновлять целевую сеть
...     @param n_episode: количество эпизодов
...     @param gamma: коэффициент обесценивания
...     @param epsilon: параметр ε-жадной стратегии
...     @param epsilon_decay: коэффициент затухания epsilon
...     """
...     for episode in range(n_episode):
...         if episode % target_update == 0:
...             estimator.copy_target()
...         policy = gen_epsilon_greedy_policy(estimator, epsilon, n_action)
...         state = env.reset()
...         is_done = False
...         while not is_done:
...             action = policy(state)
...             next_state, reward, is_done, _ = env.step(action)
...             total_reward_episode[episode] += reward
...             modified_reward = next_state[0] + 0.5
...             if next_state[0] >= 0.5:
...                 modified_reward += 100
...             elif next_state[0] >= 0.25:
...                 modified_reward += 20
...             elif next_state[0] >= 0.1:
...                 modified_reward += 10
...             elif next_state[0] >= 0:
...                 modified_reward += 5
...             memory.append((state, action, next_state,
                            modified_reward, is_done))
...             if is_done:
...                 break
...             estimator.replay(memory, replay_size, gamma)
...             state = next_state
...     print('Эпизод: {}, полное вознаграждение: {}, epsilon:{}'.
              format(episode, total_reward_episode[episode],epsilon))
...     epsilon = max(epsilon * epsilon_decay, 0.01)
```

10. Выполним глубокое Q-обучение методом Double DQN на 1000 эпизодов:

```
>>> n_episode = 600
```

Зададим размер выборки из буфера воспроизведения на каждом шаге:

```
>>> replay_size = 20
```

Целевая сеть обновляется после каждых 10 эпизодов:

```
>>> target_update = 10
```

Будем запоминать полные вознаграждения в каждом эпизоде:

```
>>> total_reward_episode = [0] * n_episode
>>> q_learning(env, dqn, n_episode, replay_size, gamma=.9, epsilon=.3)

Эпизод: 0, полное вознаграждение: -200.0, epsilon: 1
Эпизод: 1, полное вознаграждение: -200.0, epsilon: 0.99
Эпизод: 2, полное вознаграждение: -200.0, epsilon: 0.9801
……
……
Эпизод: 991, полное вознаграждение: -151.0, epsilon: 0.01
Эпизод: 992, полное вознаграждение: -200.0, epsilon: 0.01
Эпизод: 993, полное вознаграждение: -158.0, epsilon: 0.01
Эпизод: 994, полное вознаграждение: -160.0, epsilon: 0.01
Эпизод: 995, полное вознаграждение: -200.0, epsilon: 0.01
Эпизод: 996, полное вознаграждение: -200.0, epsilon: 0.01
Эпизод: 997, полное вознаграждение: -200.0, epsilon: 0.01
Эпизод: 998, полное вознаграждение: -151.0, epsilon: 0.01
Эпизод: 999, полное вознаграждение: -200.0, epsilon: 0.01
```

11. Построим график зависимости вознаграждения в эпизоде от времени:

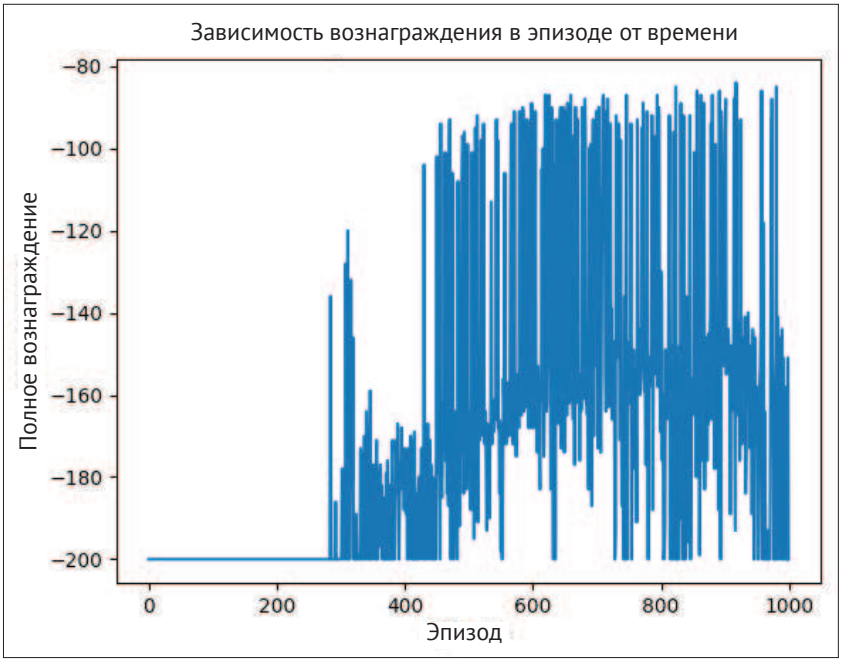
```
>>> import matplotlib.pyplot as plt
>>> plt.plot(total_reward_episode)
>>> plt.title('Зависимость вознаграждения в эпизоде от времени')
>>> plt.xlabel('Эпизод')
>>> plt.ylabel('Полное вознаграждение')
>>> plt.show()
```

Как это работает

На шаге 5 функция `replay` сначала производит случайную выборку размера `replay_size` из буфера воспроизведения опыта. Затем каждый элемент выборки преобразуется в обучающий пример, состоящий из входного состояния и выходного целевого значения. После этого получившийся пакет используется для обновления нейронной сети.

Шаг 9 самый важный в алгоритме Double DQN: на нем для вычисления целевых значений используется другая сеть, которая обновляется не после каждого эпизода. Прочие части функции `replay` такие же, как в глубоком Q-обучении с воспроизведением опыта.

График, построенный на шаге 11, выглядит следующим образом:



Мы видим, что после первых 400 эпизодов автомобиль добирается до вершины горы в большинстве случаев за 80–160 шагов.

В глубоком Q-обучении с двойной DQN используются две разные сети: для предсказания и для вычисления целевых значений. Первая служит для предсказания значений Q-функции, а вторая – для вычисления устойчивых целевых значений Q-функции. С некоторым интервалом (скажем, каждые 10 эпизодов или каждые 1500 шагов обучения) обе сети синхронизируются. Поскольку целевые значения в течение некоторого времени остаются неизменными, данные, на которых обучается предсказательная сеть, более стабильны. Полученные результаты показывают, что метод Double DQN превосходит обычный DQN с одной сетью.

Настройка гиперпараметров алгоритма Double DQN для среды CartPole

В этом рецепте мы решим задачу о балансировании стержня методом Double DQN. Мы покажем, как настроить его гиперпараметры для достижения наилучших результатов.

Для настройки гиперпараметров применим технику **поиска на сетке** – будем исследовать различные комбинации значений и выберем ту, при которой

среднее качество максимально. Начать можно с грубой сетки, а затем постепенно измельчать ее. И не забывайте фиксировать начальные значения следующих генераторов случайных чисел, чтобы обеспечить воспроизводимость результатов:

- генератор случайных чисел, встроенный в окружающую среду Gym;
- генератор случайных чисел ε-жадной стратегии;
- начальные веса нейронной сети в PyTorch.

Как это делается

Для решения задачи о балансировании стержня выполним следующие действия.

1. Импортируем необходимые модули и создадим экземпляр окружающей среды CartPole.

```
>>> import gym
>>> import torch
>>> from collections import deque
>>> import random
>>> import copy
>>> from torch.autograd import Variable
>>> env = gym.envs.make("CartPole-v0")
```

2. Воспользуемся классом `DQN`, разработанным в предыдущем рецепте.

3. Воспользуемся функцией `gen_epsilon_greedy_policy`, разработанной в рецепте «Реализация глубоких Q-сетей», и не будем здесь повторять ее код.

4. Определим функцию, выполняющую глубокое Q-обучение методом Double DQN:

```
>>> def q_learning(env, estimator, n_episode, replay_size,
target_update=10, gamma=1.0, epsilon=0.1, epsilon_decay=.99):
...     """
...     Глубокое Q-обучение методом Double DQN с воспроизведением опыта
...     @param env: имя окружающей среды Gym
...     @param estimator: объект класса DQN
...     @param replay_size: сколько примеров использовать при каждом
...               обновлении модели
...     @param target_update: через сколько эпизодов обновлять целевую сеть
...     @param n_episode: количество эпизодов
...     @param gamma: коэффициент обесценивания
...     @param epsilon: параметр ε-жадной стратегии
...     @param epsilon_decay: коэффициент затухания epsilon
...     """
...     for episode in range(n_episode):
...         if episode % target_update == 0:
...             estimator.copy_target()
...         policy = gen_epsilon_greedy_policy(estimator, epsilon, n_action)
...         state = env.reset()
...         is_done = False
```

```
...         while not is_done:
...             action = policy(state)
...             next_state, reward, is_done, _ = env.step(action)
...             total_reward_episode[episode] += reward
...             memory.append((state, action, next_state, reward, is_done))
...             if is_done:
...                 break
...             estimator.replay(memory, replay_size, gamma)
...             state = next_state
...         epsilon = max(epsilon * epsilon_decay, 0.01)
```

5. Зададим форму нейронной сети – размеры входного, выходного и скрытого слоев, скорость обучения 0.01, общее количество эпизодов и количество эпизодов, на которых оценивается качество:

```
>>> n_state = env.observation_space.shape[0]
>>> n_action = env.action_space.n
>>> n_episode = 600
>>> last_episode = 200
```

6. Определим гиперпараметры, настраиваемые с помощью поиска на сетке:

```
>>> n_hidden_options = [30, 40]
>>> lr_options = [0.001, 0.003]
>>> replay_size_options = [20, 25]
>>> target_update_options = [30, 35]
```

7. И наконец, произведем поиск на сетке: на каждой итерации будем создавать экземпляр класса `DQN` с текущими гиперпараметрами и обучать его на 600 эпизодах. Затем оценим качество, усреднив полные вознаграждения по последним 200 эпизодам.

```
>>> for n_hidden in n_hidden_options:
...     for lr in lr_options:
...         for replay_size in replay_size_options:
...             for target_update in target_update_options:
...                 env.seed(1)
...                 random.seed(1)
...                 torch.manual_seed(1)
...                 dqn = DQN(n_state, n_action, n_hidden, lr)
...                 memory = deque(maxlen=10000)
...                 total_reward_episode = [0] * n_episode
...                 q_learning(env, dqn, n_episode, replay_size,
                        target_update, gamma=.9, epsilon=1)
...                 print(n_hidden, lr, replay_size, target_update,
                        sum(total_reward_episode[-last_episode:])/last_episode)
```

Как это работает

После шага 7 печатаются следующие результаты поиска на сетке:

```
30 0.001 20 30 143.15
30 0.001 20 35 156.165
```

```
30 0.001 25 30 180.575
30 0.001 25 35 192.765
30 0.003 20 30 187.435
30 0.003 20 35 122.42
30 0.003 25 30 169.32
30 0.003 25 35 172.65
40 0.001 20 30 136.64
40 0.001 20 35 160.08
40 0.001 25 30 141.955
40 0.001 25 35 122.915
40 0.003 20 30 143.855
40 0.003 20 35 178.52
40 0.003 25 30 125.52
40 0.003 25 35 178.85
```

Как видим, наилучшее среднее вознаграждение, 192.77, получено при такой комбинации гиперпараметров: `n_hidden=30`, `lr=0.001`, `replay_size=25`, `target_update=35`.

Можете еще поэкспериментировать с гиперпараметрами – возможно, вам удастся найти лучшую модель DQN.

В этом рецепте мы решили задачу о балансировании стержня методом Double DQN. Мы применили поиск на сетке для настройки гиперпараметров: размера скрытого слоя, скорости обучения, размера пакета, выбираемого из буфера воспроизведения опыта, и частоты обновления целевой сети. Существуют и другие гиперпараметры: количество эпизодов, начальное значение ε, коэффициент затухания ε, – их тоже можно исследовать. Но во всех экспериментах фиксируйте начальные значения генераторов случайных чисел, чтобы результаты были воспроизводимы и допускали сравнение. Качество модели DQN измеряется как среднее полное вознаграждение в нескольких последних эпизодах.

Реализация алгоритма Dueling DQN

В этом рецепте мы разработаем еще один продвинутый алгоритм DQN – **Dueling DQN** (**DDQN**), или **дуэльные DQN**. Мы покажем, как вычисление значений Q-функции разбивается в DDQN на две части.

В DDQN Q-функция вычисляется по формуле:

$$Q(s,a) = V(s) + A(s,a) - \frac{1}{|A|}\sum_{a=1}^{|A|} A(s,a),$$

где $V(s)$ – функция ценности состояний, вычисляющая ценность нахождения в состоянии s, а $A(s, a)$ – зависящая от состояния функция преимущества, которая оценивает, насколько действие a в состоянии s лучше всех остальных действий в том же состоянии. Разделив функции ценности и преимущества, мы сможем учесть тот факт, что в процессе обучения агент необязательно смотрит на ценность и преимущество одновременно. Иными словами, агент, обучаемый с помощью алгоритма DDQN, может по своему выбору оптимизировать обе функции или какую-то одну из них.

Как это делается

Решим задачу о машине на горе методом DDQN.

1. Импортируем необходимые модули и создадим экземпляр окружающей среды Mountain Car.

```
>>> import gym
>>> import torch
>>> from collections import deque
>>> import random
>>> from torch.autograd import Variable
>>> import torch.nn as nn
>>> env = gym.envs.make("MountainCar-v0")
```

2. Определим модель DDQN:

```
>>> class DuelingModel(nn.Module):
...     def __init__(self, n_input, n_output, n_hidden):
...         super(DuelingModel, self).__init__()
...         self.adv1 = nn.Linear(n_input, n_hidden)
...         self.adv2 = nn.Linear(n_hidden, n_output)
...         self.val1 = nn.Linear(n_input, n_hidden)
...         self.val2 = nn.Linear(n_hidden, 1)
...
...     def forward(self, x):
...         adv = nn.functional.relu(self.adv1(x))
...         adv = self.adv2(adv)
...         val = nn.functional.relu(self.val1(x))
...         val = self.val2(val)
...         return val + adv - adv.mean()
```

3. Воспользуемся моделью DDQN в классе `DQN`:

```
>>> class DQN():
...     def __init__(self, n_state, n_action, n_hidden=50, lr=0.05):
...         self.criterion = torch.nn.MSELoss()
...         self.model = DuelingModel(n_state, n_action, n_hidden)
...         self.optimizer = torch.optim.Adam(self.model.parameters(), lr)
```

   Больше в классе `DQN` ничего не меняется.

4. Воспользуемся функцией `gen_epsilon_greedy_policy`, разработанной в рецепте «Реализация глубоких Q-сетей», и не будем здесь повторять ее код.

5. Воспользуемся функцией `q_learning`, разработанной в рецепте «Улучшение DQN с помощью воспроизведения опыта», и не будем здесь повторять ее код.

6. Зададим форму нейронной сети – размеры входного, выходного и скрытого слоев, скорость обучения 0.001 и создадим экземпляр класса `DQN`:

```
>>> n_state = env.observation_space.shape[0]
>>> n_action = env.action_space.n
>>> n_hidden = 50
```

```
>>> lr = 0.001
>>> dqn = DQN(n_state, n_action, n_hidden, lr)
```

7. Определим буфер для хранения опыта:

```
>>> memory = deque(maxlen=10000)
```

Новые примеры будут добавляться в конец очереди, а старые – удаляться из начала, когда количество элементов в очереди превысит 10 000.

8. Будем выполнять глубокое Q-обучение методом DDQN на 600 эпизодах:

```
>>> n_episode = 600
```

Зададим размер выборки из буфера воспроизведения на каждом шаге:

```
>>> replay_size = 20
```

Будем запоминать полные вознаграждения в каждом эпизоде:

```
>>> total_reward_episode = [0] * n_episode
>>> q_learning(env, dqn, n_episode, replay_size, gamma=.9,
epsilon=.3)
```

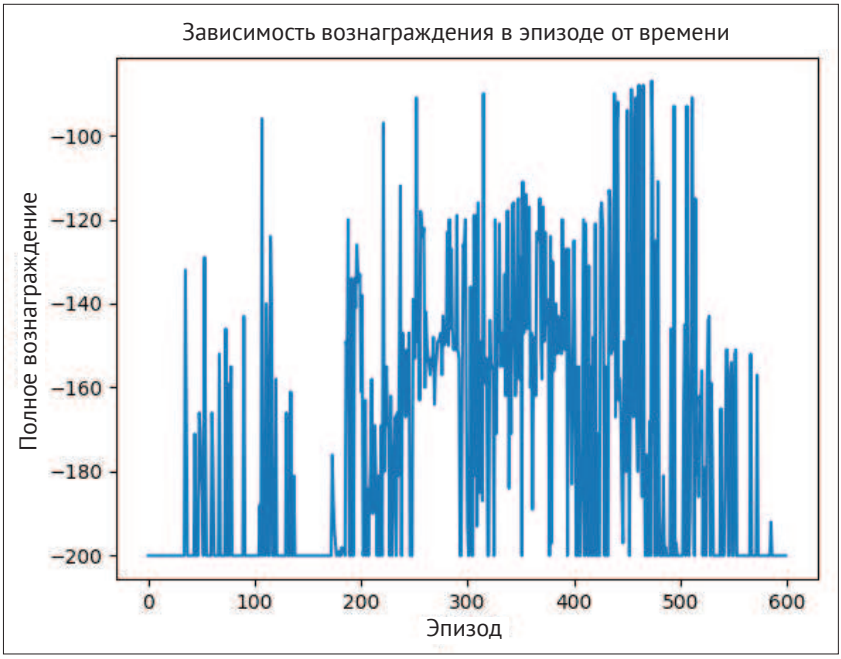
9. Построим график зависимости вознаграждения в эпизоде от времени:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(total_reward_episode)
>>> plt.title('Зависимость вознаграждения в эпизоде от времени')
>>> plt.xlabel(Эпизод')
>>> plt.ylabel('Полное вознаграждение')
>>> plt.show()
```

Как это работает

Шаг 2 – основная часть алгоритма Dueling DQN. Он состоит из двух частей: преимущество действия (`adv`) и ценность состояния (`val`). В примере используется только один скрытый слой.

График, построенный на шаге 9, выглядит следующим образом:

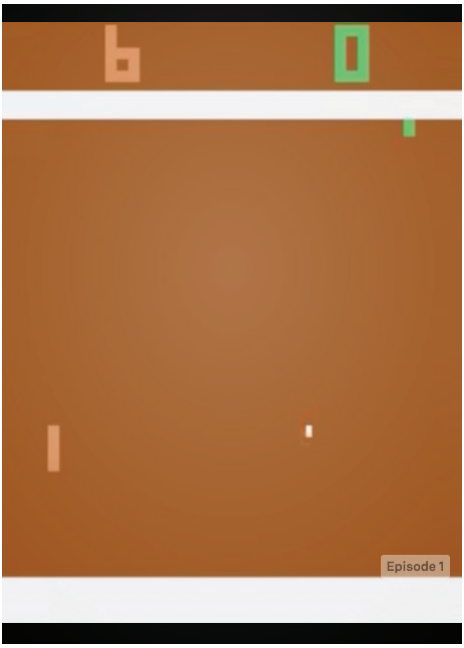


В DDQN предсказанное значение Q-функции есть сумма двух слагаемых: ценности состояния и преимущества действия. Первое оценивает, насколько хорошо находиться в некотором состоянии, второе – насколько лучше выбрать данное действие по сравнению со всеми остальными. Эти два слагаемых вычисляются по отдельности и объединяются в последнем слое сети DQN. Напомним, что в традиционных DQN обновляется значение Q только для данного действия в некотором состоянии. В DDQN ценность состояния обновляется для всех действий (не только данного), а также вычисляется преимущество действия. Поэтому сети DDQN считаются более робастными.

Применение DQN к играм Atari

До сих пор мы имели дело с довольно простыми задачами, когда применять DQN – все равно, что стрелять из пушек по воробьям. В этом и следующем рецептах мы воспользуемся сетью DQN для решения игр Atari, представляющих собой гораздо более сложные задачи.

В этом рецепте мы в качестве примера возьмем окружающую среду Pong (https://gym.openai.com/envs/Pong-v0/), которая эмулирует одноименную игру для Atari 2600, в которой агент играет в настольный теннис против партнера. В этой среде наблюдением является RGB-изображение на экране (см. снимок экрана ниже).



Это матрица, имеющая форму (210, 160, 3), которая соответствует изображению размера 210×160 с тремя RGB-каналами.

Агент (справа) может двигаться вверх и вниз и должен отбивать мяч. Если он промахивается, то соперник (слева) получает 1 очко. Если же промахивается соперник, то 1 очко получает агент. Выигрывает тот, кто первым наберет 21 очко. У агента имеется шесть возможных действий:

- **0**: **NOOP**: агент стоит на месте;
- **1**: **FIRE**: бессмысленное действие;
- **2**: **RIGHT**: агент смещается вверх;
- **3**: **LEFT**: агент смещается вниз;
- **4**: **RIGHTFIRE**: то же, что 2;
- **5**: **LEFTFIRE**: то же, что 5.

Каждое действие повторяется на протяжении *k* кадров (*k* может быть равно 2, 3, 4 или 16 в зависимости от варианта окружающей среды Pong). Вознаграждение начисляется следующим образом:

- –1: агент промахнулся по мячу;
- 1: соперник промахнулся по мячу;
- 0: в остальных случаях.

Пространство наблюдений в игре Pong имеет размер *210 * 160 * 3*, это намного больше, чем все, с чем нам приходилось иметь дело раньше. Поэтому мы предварительно уменьшим размер изображений до *84 * 84* и преобразуем их в полутоновые.

Как это делается

Для взаимодействия с окружающей средой Pong выполним следующие действия.

1. Импортируем необходимые модули и создадим экземпляр окружающей среды Pong.

```
>>> import gym
>>> import torch
>>> import random
>>> env = gym.envs.make("PongDeterministic-v4")
```

   В этом варианте Pong действие детерминированное и повторяется на протяжении 16 кадров.

2. Изучим пространство наблюдений и пространство действий:

```
>>> state_shape = env.observation_space.shape
>>> n_action = env.action_space.n
>>> print(state_shape)
(210, 160, 3)
>>> print(n_action)
6
>>> print(env.unwrapped.get_action_meanings())
['NOOP', 'FIRE', 'RIGHT', 'LEFT', 'RIGHTFIRE', 'LEFTFIRE']
```

3. Зададим три действия:

```
>>> ACTIONS = [0, 2, 3]
>>> n_action = 3
```

   Это следующие действия: оставаться на месте, сместиться вверх, сместиться вниз.

4. Предпримем несколько случайных действий и нарисуем экран:

```
>>> env.reset()
>>> is_done = False
>>> while not is_done:
...     action = ACTIONS[random.randint(0, n_action - 1)]
...     obs, reward, is_done, _ = env.step(action)
...     print(reward, is_done)
...     env.render()
0.0 False
0.0 False
0.0 False
……
……
0.0 False
0.0 False
0.0 False
-1.0 True
```

   Вы увидите, как два игрока играют в настольный теннис, хотя агент все время проигрывает.

5. Напишем функцию, которая уменьшает размер изображения и преобразует его в полутоновое:

```
>>> import torchvision.transforms as T
>>> from PIL import Image
>>> image_size = 84
>>> transform = T.Compose([T.ToPILImage(),
...                        T.Grayscale(num_output_channels=1),
...                        T.Resize((image_size, image_size),
                                    interpolation=Image.CUBIC),
...                        T.ToTensor(),
...                       ])
```

Это мы создали преобразователь, изменяющий размер на 84 * 84:

```
>>> def get_state(obs):
... state = obs.transpose((2, 0, 1))
... state = torch.from_numpy(state)
... state = transform(state)
... return state
```

А эта функция преобразует измененное изображение в тензор формы (1, 84, 84):

```
>>> state = get_state(obs)
>>> print(state.shape)
torch.Size([1, 84, 84])
```

Теперь можно приступить к взаимодействию с окружающей средой, применяя алгоритм Double DQN.

1. На этот раз будем использовать нейронную сеть с двумя скрытыми слоями, поскольку во входном слое приблизительно 21 000 блоков.

```
>>> from collections import deque
>>> import copy
>>> from torch.autograd import Variable
>>> class DQN():
...     def __init__(self, n_state, n_action, n_hidden, lr=0.05):
...         self.criterion = torch.nn.MSELoss()
...         self.model = torch.nn.Sequential(
...             torch.nn.Linear(n_state, n_hidden[0]),
...             torch.nn.ReLU(),
...             torch.nn.Linear(n_hidden[0], n_hidden[1]),
...             torch.nn.ReLU(),
...             torch.nn.Linear(n_hidden[1], n_action)
...         )
...         self.model_target = copy.deepcopy(self.model)
...         self.optimizer = torch.optim.Adam(self.model.parameters(), lr)
```

2. В остальном класс `DQN` почти такой же, как в рецепте «Реализация алгоритма Double DQN», с одним мелким отличием в методе `replay`:

```
>>>     def replay(self, memory, replay_size, gamma):
...         """
...         Воспроизведение опыта с целевой сетью
```

```
...         @param memory: буфер воспроизведения опыта
...         @param replay_size: количество выбираемых из буфера примеров при
...                             каждом обновлении модели
...         @param gamma: коэффициент обесценивания
...         """
...         if len(memory) >= replay_size:
...             replay_data = random.sample(memory, replay_size)
...             states = []
...             td_targets = []
...             for state, action, next_state, reward, is_done in replay_data:
...                 states.append(state.tolist())
...                 q_values = self.predict(state).tolist()
...                 if is_done:
...                     q_values[action] = reward
...                 else:
...                     q_values_next = self.target_predict(next_state).detach()
...                     q_values[action] = reward + gamma *
                            torch.max(q_values_next).item()
...                 td_targets.append(q_values)
...             self.update(states, td_targets)
```

3. Воспользуемся функцией `gen_epsilon_greedy_policy`, разработанной в рецепте «Реализация глубоких Q-сетей», и не будем здесь повторять ее код.

4. Напишем функцию глубокого Q-обучения методом Double DQN:

```
>>> def q_learning(env, estimator, n_episode, replay_size,
target_update=10, gamma=1.0, epsilon=0.1, epsilon_decay=.99):
...     """
...     Глубокое Q-обучение с применением DDQN и буфера воспроизведения опыта
...     @param env: имя окружающей среды Gym
...     @param estimator: объекта класса DQN
...     @param replay_size: сколько примеров использовать при каждом
...                обновлении модели
...     @param target_update: через сколько эпизодов обновлять целевую сеть
...     @param n_episode: количество эпизодов
...     @param gamma: коэффициент обесценивания
...     @param epsilon: параметр ε-жадной стратегии
...     @param epsilon_decay: коэффициент затухания epsilon
...     """
...     for episode in range(n_episode):
...         if episode % target_update == 0:
...             estimator.copy_target()
...             policy = gen_epsilon_greedy_policy(estimator, epsilon, n_action)
...             obs = env.reset()
...             state = get_state(obs).view(image_size * image_size)[0]
...             is_done = False
...             while not is_done:
...                 action = policy(state)
...                 next_obs, reward, is_done, _ = env.step(ACTIONS[action])
...                 total_reward_episode[episode] += reward
...                 next_state = get_state(obs).view(image_size * image_size)
...                 memory.append((state, action, next_state, reward, is_done))
```

```
...                 if is_done:
...                     break
...                 estimator.replay(memory, replay_size, gamma)
...                 state = next_state
...             print('Эпизод: {}, полное вознаграждение: {}, epsilon:{}'.
                    format(episode, total_reward_episode[episode], epsilon))
...             epsilon = max(epsilon * epsilon_decay, 0.01)
```

Наблюдение размера [210, 160, 3] будет преобразовано в матрицу полутонового изображения меньшего размера [84, 84] и сериализовано для подачи на вход нейронной сети.

5. Теперь зададим форму нейронной сети: размеры входного и скрытых слоев:

```
>>> n_state = image_size * image_size
>>> n_hidden = [200, 50]
```

Зададим также гиперпараметры:

```
>>> n_episode = 1000
>>> lr = 0.003
>>> replay_size = 32
>>> target_update = 10
```

И создадим объект класса `DQN`:

```
>>> dqn = DQN(n_state, n_action, n_hidden, lr)
```

6. Определим буфер для хранения опыта:

```
>>> memory = deque(maxlen=10000)
```

7. И наконец, выполним глубокое Q-обучение и будем запоминать полные вознаграждения в каждом эпизоде:

```
>>> total_reward_episode = [0] * n_episode
>>> q_learning(env, dqn, n_episode, replay_size, target_update,
gamma=.9, epsilon=1)
```

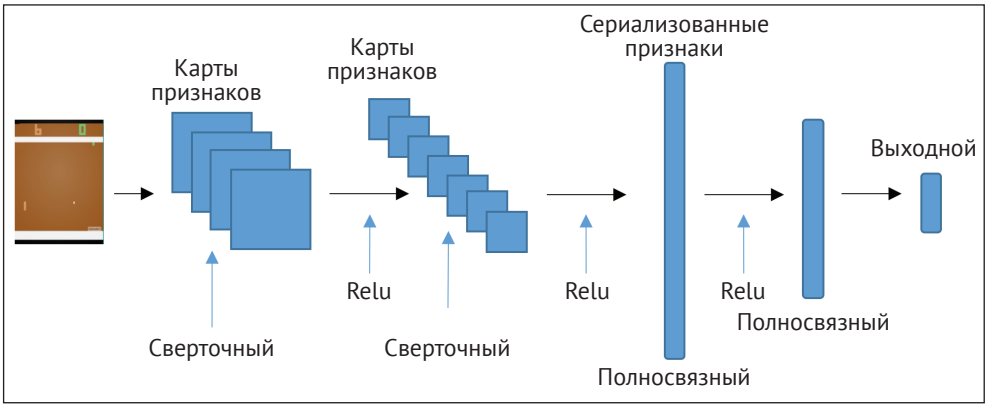
Как это работает

Наблюдение в среде Pong гораздо сложнее, чем в средах, с которыми мы работали ранее. Это трехканальное изображение размера 210×160. Поэтому мы предварительно преобразуем его в полутоновое и уменьшаем размер до 84×84, а затем сериализуем, т. е. преобразуем в одномерный массив, чтобы подать на вход полносвязной нейронной сети. Входной слой состоит примерно из 6000 блоков, и, чтобы справиться с такой сложностью, мы включили в сеть два скрытых слоя.

Использование сверточных нейронных сетей в играх Atari

В предыдущем рецепте все наблюдаемые изображения в среде Pong преобразовывались в полутоновые, а затем в одномерный массив, подаваемый на вход полносвязной нейронной сети. Такая сериализация изображения может привести к потере информации. А почему бы не подавать на вход само изображение? В этом рецепте мы включим в модель DQN **сверточную нейронную сеть** (**CHC**).

CHC – одна из лучших архитектур нейронных сетей для работы с изображениями. Ее сверточные слои эффективно выделяют из изображения признаки, которые затем передаются полносвязным слоям. На рисунке ниже приведен пример CHC с двумя сверточными слоями.



Легко понять, что если просто «распрямить» изображение в вектор, то мы потеряем часть информации о местоположении мяча и игроков, а такого рода информация очень важна для обучения модели. В сверточных слоях CHC эта информация представлена картами признаков, порождаемыми несколькими фильтрами.

Мы по-прежнему уменьшаем размер изображения с 210×160 до 84×84, но сохраняем все три RGB-канала, не преобразуя их в линейный массив.

Как это делается

Решим задачу об игре Pong с помощью комбинации DQN и CHC.

1. Импортируем необходимые модули и создадим экземпляр окружающей среды Pong.

```
>>> import gym
>>> import torch
>>> import random
```

```
>>> from collections import deque
>>> import copy
>>> from torch.autograd import Variable
>>> import torch.nn as nn
>>> import torch.nn.functional as F
>>> env = gym.envs.make("PongDeterministic-v4")
```

2. Зададим три действия:

```
>>> ACTIONS = [0, 2, 3]
>>> n_action = 3
```

Это следующие действия: оставаться на месте, сместиться вверх, сместиться вниз.

3. Напишем функцию, которая уменьшает размер изображения:

```
>>> import torchvision.transforms as T
>>> from PIL import Image
>>> image_size = 84
>>> transform = T.Compose([T.ToPILImage(),
...                        T.Resize((image_size, image_size),
                                    interpolation=Image.CUBIC),
...                        T.ToTensor()])
```

Далее определим функцию, которая уменьшает изображение и преобразует его в тензор формы (3, 84, 84):

```
>>> def get_state(obs):
...     state = obs.transpose((2, 0, 1))
...     state = torch.from_numpy(state)
...     state = transform(state).unsqueeze(0)
...     return state
```

4. Теперь можно приступить к взаимодействию со средой Pong, для чего нужно разработать модель CNC:

```
>>> class CNNModel(nn.Module):
...     def __init__(self, n_channel, n_action):
...         super(CNNModel, self).__init__()
...         self.conv1 = nn.Conv2d(in_channels=n_channel,
                    out_channels=32, kernel_size=8, stride=4)
...         self.conv2 = nn.Conv2d(32, 64, 4, stride=2)
...         self.conv3 = nn.Conv2d(64, 64, 3, stride=1)
...         self.fc = torch.nn.Linear(7 * 7 * 64, 512)
...         self.out = torch.nn.Linear(512, n_action)
...
...     def forward(self, x):
...         x = F.relu(self.conv1(x))
...         x = F.relu(self.conv2(x))
...         x = F.relu(self.conv3(x))
...         x = x.view(x.size(0), -1)
...         x = F.relu(self.fc(x))
...         output = self.out(x)
...         return output
```

5. Воспользуемся только что созданной моделью CHC в нашей модели DQN:

```
>>> class DQN():
...     def __init__(self, n_channel, n_action, lr=0.05):
...         self.criterion = torch.nn.MSELoss()
...         self.model = CNNModel(n_channel, n_action)
...         self.model_target = copy.deepcopy(self.model)
...         self.optimizer = torch.optim.Adam(self.model.parameters(), lr)
```

6. В остальном класс `DQN` почти такой же, как в рецепте «Реализация алгоритма Double DQN», с одним мелким отличием в методе `replay`:

```
>>> def replay(self, memory, replay_size, gamma):
...     """
...     Воспроизведение опыта с целевой сетью
...     @param estimator: буфер воспроизведения опыта
...     @param replay_size: сколько примеров использовать при каждом
...               обновлении модели
...     @param gamma: коэффициент обесценивания
...     """
...     if len(memory) >= replay_size:
...         replay_data = random.sample(memory, replay_size)
...         states = []
...         td_targets = []
...         for state, action, next_state, reward, is_done in replay_data:
...             states.append(state.tolist()[0])
...             q_values = self.predict(state).tolist()[0]
...             if is_done:
...                 q_values[action] = reward
...             else:
...                 q_values_next = self.target_predict(next_state).detach()
...                 q_values[action] = reward + gamma *
                            torch.max(q_values_next).item()
...             td_targets.append(q_values)
...     self.update(states, td_targets)
```

7. Воспользуемся функцией `gen_epsilon_greedy_policy`, разработанной в рецепте «Реализация глубоких Q-сетей», и не будем здесь повторять ее код.

8. Напишем функцию глубокого Q-обучения методом Double DQN:

```
>>> def q_learning(env, estimator, n_episode, replay_size,
target_update=10, gamma=1.0, epsilon=0.1, epsilon_decay=.99):
...     """
...     Глубокое Q-обучение методом Double DQN с воспроизведением опыта
...     @param env: имя окружающей среды Gym
...     @param estimator: объект класса DQN
...     @param replay_size: сколько примеров использовать при каждом
...               обновлении модели
...     @param target_update: через сколько эпизодов обновлять целевую сеть
...     @param n_episode: количество эпизодов
...     @param gamma: коэффициент обесценивания
...     @param epsilon: параметр ε-жадной стратегии
...     @param epsilon_decay: коэффициент затухания epsilon
```

```
...     """
...     for episode in range(n_episode):
...         if episode % target_update == 0:
...             estimator.copy_target()
...         policy = gen_epsilon_greedy_policy(estimator, epsilon, n_action)
...         obs = env.reset()
...         state = get_state(obs)
...         is_done = False
...         while not is_done:
...             action = policy(state)
...             next_obs, reward, is_done, _ = env.step(ACTIONS[action])
...             total_reward_episode[episode] += reward
...             next_state = get_state(obs)
...             memory.append((state, action, next_state, reward, is_done))
...             if is_done:
...                 break
...             estimator.replay(memory, replay_size, gamma)
...             state = next_state
...         print('Эпизод: {}, полное вознаграждение: {}, epsilon: {}'.
                format(episode, total_reward_episode[episode], epsilon))
...         epsilon = max(epsilon * epsilon_decay, 0.01)
```

9. Зададим гиперпараметры:

```
>>> n_episode = 1000
>>> lr = 0.00025
>>> replay_size = 32
>>> target_update = 10
```

   И создадим объект класса `DQN`:

```
>>> dqn = DQN(3, n_action, lr)
```

10. Определим буфер для хранения опыта:

```
>>> memory = deque(maxlen=100000)
```

11. И наконец, выполним глубокое Q-обучение и будем запоминать полные вознаграждения в каждом эпизоде:

```
>>> total_reward_episode = [0] * n_episode
>>> q_learning(env, dqn, n_episode, replay_size, target_update,
gamma=.9, epsilon=1)
```

Как это работает

Функция обработки изображений на шаге 3 сначала уменьшает размер изображения по каждому каналу до 84×84, а затем преобразует результат в тензор формы (3, 84, 84) для подачи на вход нейронной сети.

Модель CNC, созданная на шаге 4, состоит из трех сверточных слоев с функцией активации ReLU. Карты признаков, порожденные последним сверточным слоем, сериализуются и подаются на вход полносвязного скрытого слоя с 512 блоками, за которым следует выходной слой.

Впервые включение СНС в DQN было описано в статье компании DeepMind «Playing Atari with Deep Reinforcement Learning» (https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf). Модель принимает на входе пиксели изображения и выводит оценки будущих вознаграждений. Она хорошо работает и для других игровых сред Atari, в которых наблюдениями являются изображения на экране игры. Сверточные слои играют роли эффективных иерархических экстракторов признаков. Они могут обучиться представлениям признаков на исходных данных изображений, а затем подать признаки на вход полносвязных слоев для обучения стратегии успешного управления.

Имейте в виду, что процесс обучения в предыдущем примере обычно занимает пару дней даже при наличии GPU и около 90 часов на машине с 4-ядерным процессором Intel i7 2.9 ГГц.

См. также

Читателям, незнакомым с СНС, рекомендуем следующие источники:

- *Yuxi (Hayden) Liu and Saransh Mehta.* Hands-On Deep Learning Architectures with Python: глава 4 «Архитектура СНС» (издательство Packt Publishing);
- *Yuxi (Hayden) Liu and Pablo Maldonado.* R Deep Learning Projects: глава 1 «Распознавание рукописных цифр с помощью сверточных нейронных сетей» и глава 2 «Распознавание дорожных знаков беспилотными транспортными средствами» (издательство Packt Publishing).

Глава 8

Реализация методов градиента стратегии и оптимизация стратегии

Эта глава посвящена методам градиента стратегии – одной из самых популярных техник обучения с подкреплением в последние годы. Мы начнем с реализации основополагающего алгоритма REINFORCE, а затем рассмотрим его усовершенствованный вариант – REINFORCE с базой. Мы также реализуем несколько вариантов более мощного алгоритма исполнитель–критик и применим его к задачам о балансировании стержня и о блуждании на краю обрыва. Будет рассмотрена окружающая среда с непрерывным пространством действий, для решения которой мы обратимся к нормальному распределению. В конце главы мы методом перекрестной энтропии обучим агента решать задачу о балансировании стержня.

В этой главе приводятся следующие рецепты:

- реализация алгоритма REINFORCE;
- реализация алгоритма REINFORCE с базой;
- реализация алгоритма исполнитель–критик;
- решение задачи о блуждании на краю обрыва с помощью алгоритма исполнитель–критик;
- подготовка непрерывной окружающей среды Mountain Car;
- решение непрерывной задачи о блуждании на краю обрыва методом A2C;
- решение задачи о балансировании стержня методом перекрестной энтропии.

Реализация алгоритма REINFORCE

Из недавних публикаций следует, что методы градиента стратегии становятся все более популярными. Цель обучения в этом случае – оптимизировать распределение вероятностей действий, так чтобы в данном состоянии выбор дей-

ствия, приносящего большее вознаграждение, был более вероятен. В первом рецепте этой главы мы рассмотрим алгоритм REINFORCE, лежащий в основе всех методов градиента стратегии.

Алгоритм **REINFORCE** называют также методом градиента стратегии Монте-Карло, поскольку он оптимизирует стратегию, применяя метод Монте-Карло. Точнее, он собирает примеры траекторий в одном эпизоде, следуя текущей стратегии, и использует их для обучения параметров стратегии θ. Целевая функция в методе градиентов стратегии такова:

$$J(\theta) = E\left[\sum_{t=0}^{T-1} r_{t+1}\right] = \sum_{t=0}^{T-1} P(s_t, a_t) * r_{t+1}.$$

Ее градиент можно записать в виде:

$$\nabla J(\theta) = \sum_{t=0}^{T-1} \nabla \log \pi(a_t|s_t) * G_t.$$

Здесь G_t – доход, т. е. полное обесцененное вознаграждение, полученное до момента t, а $\pi(a_t|s_t)$ – стохастическая стратегия, определяющая вероятности различных действий в данном состоянии. Поскольку обновление стратегии производится после завершения всего эпизода и сбора всех примеров, REINFORCE является алгоритмом с разделенной стратегией.

Вычислив градиенты стратегии, мы применяем обратное распространение для обновления параметров стратегии. После того как стратегия обновлена, мы выполняем эпизод, собираем набор примеров и используем их для следующего обновления параметров стратегии.

Теперь реализуем алгоритм REINFORCE для взаимодействия с окружающей средой CartPole (https://gym.openai.com/envs/CartPole-v0/).

Как это делается

1. Импортируем необходимые пакеты и создадим экземпляр окружающей среды CartPole:

```
>>> import gym
>>> import torch
>>> import torch.nn as nn
>>> env = gym.make('CartPole-v0')
```

2. Сначала напишем метод `__init__` класса `PolicyNetwork`, который аппроксимирует стратегию нейронной сетью:

```
>>> class PolicyNetwork():
...     def __init__(self, n_state, n_action, n_hidden=50, lr=0.001):
...         self.model = nn.Sequential(
...             nn.Linear(n_state, n_hidden),
...             nn.ReLU(),
...             nn.Linear(n_hidden, n_action),
...             nn.Softmax(),
...         )
...         self.optimizer = torch.optim.Adam(self.model.parameters(), lr)
```

3. Добавим метод predict, который вычисляет оценку стратегии:

```
>>>     def predict(self, s):
...         """
...         Вычисляет вероятности действий в состоянии s,
...         применяя обученную модель
...         @param s: входное состояние
...         @return: предсказанная стратегия
...         """
...         return self.model(torch.Tensor(s))
```

4. Теперь напишем метод, который обновляет нейронную сеть на основе собранных в эпизоде примеров:

```
>>>     def update(self, returns, log_probs):
...         """
...         Обновляет веса сети стратегии на основе обучающих примеров
...         @param returns: доход (накопительное вознаграждение) на каждом
...         шаге эпизода
...         @param log_probs: логарифмы вероятностей на каждом шаге
...         """
...         policy_gradient = []
...         for log_prob, Gt in zip(log_probs, returns):
...             policy_gradient.append(-log_prob * Gt)
...
...             loss = torch.stack(policy_gradient).sum()
...             self.optimizer.zero_grad()
...             loss.backward()
...             self.optimizer.step()
```

5. Последний метод класса PolicyNetwork – get_action, он выбирает действие в данном состоянии на основе предсказанной стратегии.

```
>>>     def get_action(self, s):
...         """
...         Предсказывает стратегию, выбирает действие и вычисляет логарифм
...         его вероятности
...         @param s: входное состояние
...         @return: выбранное действие и логарифм его вероятности
...         """
...         probs = self.predict(s)
...         action = torch.multinomial(probs, 1).item()
...         log_prob = torch.log(probs[action])
...         return action, log_prob
```

Логарифм вероятности действия станет частью обучающего примера. С классом PolicyNetwork – все.

6. Теперь перейдем к алгоритму REINFORCE с сетевой моделью стратегии.

```
>>> def reinforce(env, estimator, n_episode, gamma=1.0):
...     """
...     Алгоритм REINFORCE
...     @param env: имя окружающей среды Gym
...     @param estimator: сеть, аппроксимирующая стратегию
```

```
...     @param n_episode: количество эпизодов
...     @param gamma: коэффициент обесценивания
...     """
...     for episode in range(n_episode):
...         log_probs = []
...         rewards = []
...         state = env.reset()
...         while True:
...             action, log_prob = estimator.get_action(state)
...             next_state, reward, is_done, _ = env.step(action)
...             total_reward_episode[episode] += reward
...             log_probs.append(log_prob)
...             rewards.append(reward)
...
...             if is_done:
...                 returns = []
...                 Gt = 0
...                 pw = 0
...                 for reward in rewards[::-1]:
...                     Gt += gamma ** pw * reward
...                     pw += 1
...                     returns.append(Gt)
...                 returns = returns[::-1]
...                 returns = torch.tensor(returns)
...                 returns = (returns - returns.mean()) / (
...                         returns.std() + 1e-9)
...                 estimator.update(returns, log_probs)
...                 print('Эпизод: {}, полное вознаграждение: {}'.
                        format(episode, total_reward_episode[episode]))
...                 break
...
...             state = next_state
```

7. Зададим форму сети стратегии (размеры входного, выходного и скрытого слоев) и скорость обучения, затем создадим экземпляр класса `PolicyNetwork`:

```
>>> n_state = env.observation_space.shape[0]
>>> n_action = env.action_space.n
>>> n_hidden = 128
>>> lr = 0.003
>>> policy_net = PolicyNetwork(n_state, n_action, n_hidden, lr)
```

   Коэффициент обесценивания положим равным 0.9:

```
>>> gamma = 0.9
```

8. Выполним обучение методом REINFORCE с только что разработанной стратегией с нейронной сетью на 500 эпизодах и будем сохранять полные вознаграждения в каждом эпизоде:

```
>>> n_episode = 500
>>> total_reward_episode = [0] * n_episode
>>> reinforce(env, policy_net, n_episode, gamma)
```

9. Построим график зависимости вознаграждения в эпизоде от времени:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(total_reward_episode)
>>> plt.title('Зависимость вознаграждения в эпизоде от времени')
>>> plt.xlabel('Эпизод')
>>> plt.ylabel('Полное вознаграждение')
>>> plt.show()
```

Как это работает

На шаге 2 мы для простоты создали нейронную сеть с одним скрытым слоем. На вход сети подается состояние, затем следует скрытый слой, и на выходе выдаются вероятности выбора различных действий. Поэтому в качестве функции активации в выходном слое используется softmax.

На шаге 4 обновляются параметры сети: пользуясь всеми данными, собранными в эпизоде, в т. ч. доходами и логарифмами вероятностей, мы вычисляем градиенты стратегии, а затем применяем обратное распространение, чтобы обновить параметры стратегии.

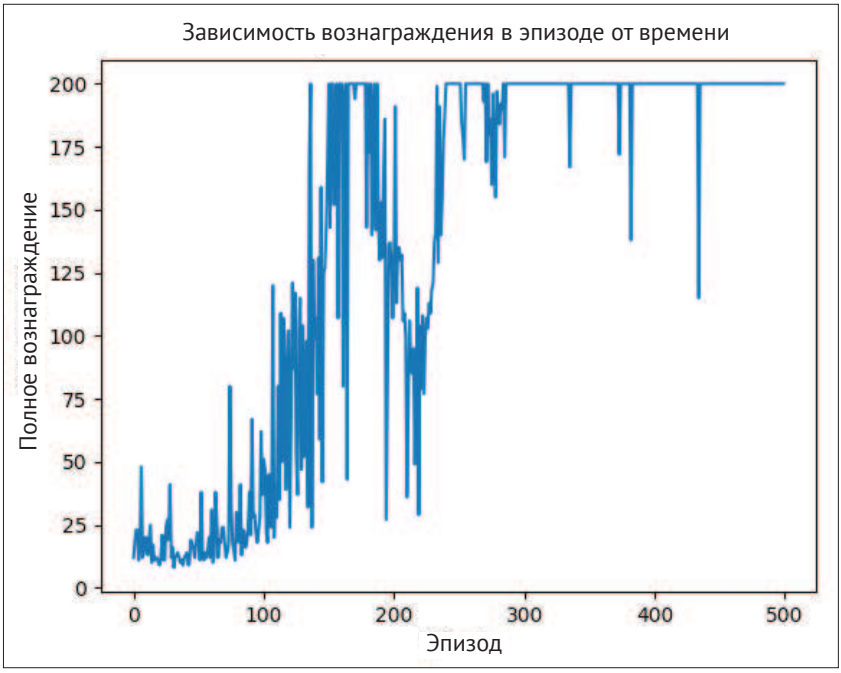
На шаге 6 алгоритма REINFORCE выполняются следующие действия:

- прогоняется один эпизод: на каждом шаге выбирается действие, следуя текущей стратегии, сохраняются вознаграждение и логарифм вероятности;
- по завершении эпизода вычисляется обесцененный доход на каждом шаге; полученные доходы нормируются путем вычитания среднего и деления на стандартное отклонение;
- зная доходы и логарифмы вероятностей, вычисляются градиенты стратегии, а затем обновляются параметры стратегии. Кроме того, отображается полное вознаграждение в каждом эпизоде;
- все вышеперечисленные шаги повторяются в `n_episode` эпизодах.

На шаге 8 печатаются такие сообщения:

```
Эпизод: 0, полное вознаграждение: 12.0
Эпизод: 1, полное вознаграждение: 18.0
Эпизод: 2, полное вознаграждение: 23.0
Эпизод: 3, полное вознаграждение: 23.0
Эпизод: 4, полное вознаграждение: 11.0
.....
.....
Эпизод: 495, полное вознаграждение: 200.0
Эпизод: 496, полное вознаграждение: 200.0
Эпизод: 497, полное вознаграждение: 200.0
Эпизод: 498, полное вознаграждение: 200.0
Эпизод: 499, полное вознаграждение: 200.0
```

График, построенный на шаге 9, выглядит следующим образом:



Мы видим, что в большинстве из последних 200 эпизодов вознаграждение достигает максимального значения +200.

REINFORCE – это семейство методов градиента стратегии, в которых параметры стратегии обновляются по формуле

$$\Delta\theta = \alpha \sum_{t=0}^{T-1} \nabla \log \pi(a_t|s_t) * G_t,$$

где α – скорость обучения, $\pi(a_t|s_t)$ – отображение действий на их вероятности, а G_t – доход, т. е. накопительное обесцененное вознаграждение на шаге эпизода t. Поскольку набор обучающих примеров строится только после завершения эпизода, REINFORCE является алгоритмом с разделенной стратегией. Процесс обучения можно свести к следующим шагам.

1. Случайным образом инициализировать параметры стратегии θ.
2. Выполнить эпизод, выбирая действия в соответствии с текущей стратегией.
3. На каждом шаге сохранять логарифм вероятности выбранного действия, а также полученное вознаграждение.
4. Вычислить доход на каждом шаге.
5. Вычислить градиенты стратегии, зная логарифмы вероятностей и доходы, и обновить параметры стратегии θ методом обратного распространения.
6. Повторять шаги 2–5.

Еще раз повторим, что поскольку алгоритму REINFORCE необходимы полные траектории, генерируемые стохастической стратегией, он относится к семейству методов Монте-Карло.

См. также

Вывести формулу градиента стратегии непросто. Для этого используется формула градиента логарифма (log derivative trick). Интересующиеся могут обратиться к статье по адресу http://www.1-4-5.net/~dmm/ml/log_derivative_trick.pdf.

Реализация алгоритма REINFORCE с базой

В алгоритме REINFORCE эпизод прогоняется до конца, и только потом собранные данные используются для обновления стратегии. Однако в силу стохастичности стратегии в одном и том же состоянии в разных эпизодах могут выбираться разные действия. Это может запутать обучение, потому что один пример требует увеличить вероятность выбора некоторого действия, а другой – уменьшить ее. Для решения этой проблемы высокой дисперсии разработан вариант алгоритма – REINFORCE с базой, который мы и рассмотрим в этом рецепте.

В алгоритме REINFORCE с базой мы вычитаем базовую ценность состояния из дохода G. Поэтому при обновлении градиента используется функция преимущества A, определенная следующим образом:

$$A_t = G_t - V(s_t);$$

$$\Delta J(\theta) = \sum_{t=0}^{T-1} \nabla \log \pi(a_t|s_t) * A_t.$$

Здесь $V(s)$ – функция, оценивающая ценность состояний. Обычно используется либо линейная функция, либо нейронная сеть. Введение базовой ценности позволяет откалибровать вознаграждения относительно среднего действия в данном состоянии.

Мы разработаем алгоритм REINFORCE с базой, используя две нейронные сети – одну для стратегии, другую для оценивания ценности, – и применим его к окружающей среде CartPole.

Как это делается

1. Импортируем необходимые пакеты и создадим экземпляр окружающей среды CartPole:

```
>>> import gym
>>> import torch
>>> import torch.nn as nn
>>> from torch.autograd import Variable
>>> env = gym.make('CartPole-v0')
```

2. Часть, относящаяся к сети стратегии, в основном такая же, как в классе `PolicyNetwork`, написанном в предыдущем рецепте. Напомним, что в методе `update` используется функция преимущества:

```
>>> def update(self, advantages, log_probs):
...     """
...     Обновляет веса сети стратегии на основе обучающих примеров
...     @param advantages: преимущества на каждом шаге эпизода
...     @param log_probs: логарифмы вероятностей на каждом шаге
...     """
...     policy_gradient = []
...     for log_prob, Gt in zip(log_probs, advantages):
...         policy_gradient.append(-log_prob * Gt)
...
...         loss = torch.stack(policy_gradient).sum()
...         self.optimizer.zero_grad()
...         loss.backward()
...         self.optimizer.step()
```

3. В качестве сети ценности мы будем использовать сеть регрессии с одним скрытым слоем:

```
>>> class ValueNetwork():
...     def __init__(self, n_state, n_hidden=50, lr=0.05):
...         self.criterion = torch.nn.MSELoss()
...         self.model = torch.nn.Sequential(
...         torch.nn.Linear(n_state, n_hidden),
...         torch.nn.ReLU(),
...         torch.nn.Linear(n_hidden, 1)
...         )
...         self.optimizer = torch.optim.Adam(self.model.parameters(), lr)
```

При ее обучении ставится цель аппроксимировать ценности состояний, поэтому в роли функции потерь выступает среднеквадратическая ошибка.

Метод `update` обучает модель регрессии на множестве входных состояний и целевых ценностей, разумеется, методом обратного распространения:

```
...     def update(self, s, y):
...         """
...         Обновить веса DQN, получив обучающий пример
...         @param s: состояния
...         @param y: целевые ценности
...         """
...         y_pred = self.model(torch.Tensor(s))
...         loss = self.criterion(y_pred, Variable(torch.Tensor(y)))
...         self.optimizer.zero_grad()
...         loss.backward()
...         self.optimizer.step()
```

А метод `predict` оценивает ценность состояния:

```
...     def predict(self, s):
...         """
```

```
...         Вычисляет значения Q-функции состояния для всех действий,
...         применяя обученную модель
...         @param s: входное состояние
...         @return: значения Q-функции состояния для всех действий
...         """
...         with torch.no_grad():
...             return self.model(torch.Tensor(s))
```

4. Теперь можно перейти к реализации алгоритма REINFORCE с базой с моделью, включающей сети стратегии и ценности:

```
>>> def reinforce(env, estimator_policy, estimator_value,
n_episode, gamma=1.0):
...     """
...     Алгоритм REINFORCE с базой
...     @param env: имя окружающей среды Gym
...     @param estimator: сеть стратегии
...     @param estimator_value: сеть ценности
...     @param n_episode: количество эпизодов
...     @param gamma: коэффициент обесценивания
...     """
...     for episode in range(n_episode):
...         log_probs = []
...         states = []
...         rewards = []
...         state = env.reset()
...         while True:
...             states.append(state)
...             action, log_prob = estimator_policy.get_action(state)
...             next_state, reward, is_done, _ = env.step(action)
...             total_reward_episode[episode] += reward
...             log_probs.append(log_prob)
...             rewards.append(reward)
...
...             if is_done:
...                 Gt = 0
...                 pw = 0
...                 returns = []
...                 for t in range(len(states)-1, -1, -1):
...                     Gt += gamma ** pw * rewards[t]
...                     pw += 1
...                     returns.append(Gt)
...                 returns = returns[::-1]
...                 returns = torch.tensor(returns)
...                 baseline_values = estimator_value.predict(states)
...                 advantages = returns - baseline_values
...                 estimator_value.update(states, returns)
...                 estimator_policy.update(advantages, log_probs)
...                 print('Эпизод: {}, полное вознаграждение: {}'.
                        format(episode, total_reward_episode[episode]))
...                 break
...             state = next_state
```

5. Зададим форму сети стратегии (размеры входного, выходного и скрытого слоев) и скорость обучения, затем создадим экземпляр класса `PolicyNetwork`:

```
>>> n_state = env.observation_space.shape[0]
>>> n_action = env.action_space.n
>>> n_hidden_p = 64
>>> lr_p = 0.003
>>> policy_net = PolicyNetwork(n_state, n_action, n_hidden_p, lr_p)
```

   Для сети ценности также зададим размер и создадим экземпляр класса:

```
>>> n_hidden_v = 64
>>> lr_v = 0.003
>>> value_net = ValueNetwork(n_state, n_hidden_v, lr_v)
```

   Коэффициент обесценивания положим равным 0.9:

```
>>> gamma = 0.9
```

6. Выполним обучение методом REINFORCE с базой на 2000 эпизодов и будем сохранять полные вознаграждения в каждом эпизоде:

```
>>> n_episode = 2000
>>> total_reward_episode = [0] * n_episode
>>> reinforce(env, policy_net, value_net, n_episode, gamma)
```

7. Построим график зависимости вознаграждения в эпизоде от времени:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(total_reward_episode)
>>> plt.title('Зависимость вознаграждения в эпизоде от времени')
>>> plt.xlabel('Эпизод')
>>> plt.ylabel('Полное вознаграждение')
>>> plt.show()
```

Как это работает

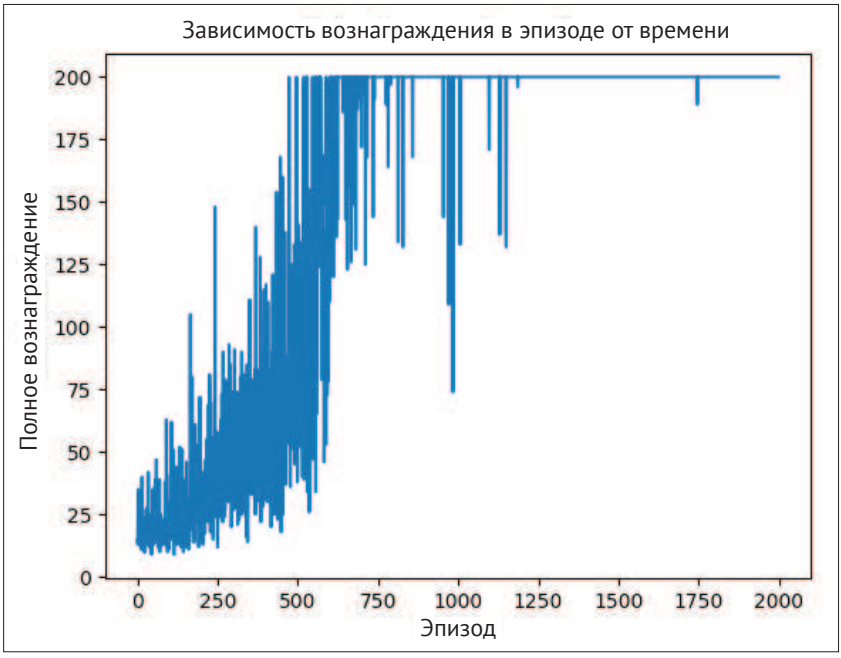
Алгоритм REINFORCE относится к семейству методов Монте-Карло, поскольку для обучения сети стратегии ему нужна полная траектория. Однако в разных эпизодах при одной и той же стохастической стратегии могут выбираться разные действия. Чтобы уменьшить дисперсию, мы вычитаем ценность состояния из дохода. Получающаяся функция преимущества измеряет вознаграждение относительно среднего действия, оно и используется при обновлении градиента.

На шаге 4 алгоритма REINFORCE с базой выполняются следующие действия:

- прогоняется один эпизод: на каждом шаге сохраняются состояние, вознаграждение и логарифм вероятности;
- по завершении эпизода вычисляется обесцененный доход на каждом шаге; с помощью сети ценности вычисляется оценка базы; вычисляется преимущество путем вычитания базы из дохода;

- зная преимущества и логарифмы вероятностей, вычисляются градиенты стратегии, после чего обновляются сети стратегии и ценности. Кроме того, отображается полное вознаграждение в каждом эпизоде;
- все вышеперечисленные шаги повторяются в `n_episode` эпизодах.

График, построенный на шаге 7, выглядит следующим образом:



Видно, что результаты стабилизируются после примерно 1200 эпизодов. Благодаря базе мы смогли откалибровать вознаграждения и уменьшить дисперсию оценок градиента.

Реализация алгоритма исполнитель–критик

В алгоритме REINFORCE с базой есть два компонента: модель стратегии и функция ценности. Их можно объединить, потому что цель обучения функции ценности заключается в том, чтобы обновить сеть стратегии. Именно так и поступает алгоритм **исполнитель–критик**, который мы разработаем в этом рецепте.

Сеть в алгоритме исполнитель–критик состоит из двух частей.

- **Исполнитель.** Принимает входное состояние и выводит вероятности действий. По сути дела, он обучает оптимальную стратегию, обновляя модель с помощью информации, предоставляемой критиком.
- **Критик.** Оценивает, насколько хорошо оказаться во входном состоянии, вычисляя функции ценности. Ценность состояния подсказывает исполнителю, что он должен подправить.

У этих двух компонентов общие входной и скрытый слои сети, поскольку при такой архитектуре их обучение происходит более эффективно, чем если бы сети были полностью разделены. Соответственно, функция потерь представляет собой сумму двух слагаемых: отрицательного логарифмического правдоподобия действия, которое оценивает качество исполнителя, и среднеквадратической ошибки между оценкой и вычисленным значением дохода (измеряет качество критика).

Более распространенная версия алгоритма исполнитель–критик называется **Advantage Actor-Critic** (**A2C**). Из самого названия следует, что критик вычисляет не ценность состояния, а преимущество, как в алгоритме REINFORCE с базой. Иначе говоря, он оценивает, насколько некоторое действие в данном состоянии лучше других действий. Это позволяет уменьшить дисперсию в сети стратегии.

Как это делается

Реализуем алгоритм исполнитель–критик для окружающей среды CartPole.

1. Импортируем необходимые пакеты и создадим экземпляр окружающей среды CartPole:

```
>>> import gym
>>> import torch
>>> import torch.nn as nn
>>> import torch.nn.functional as F
>>> env = gym.make('CartPole-v0')
```

2. Сначала реализуем модель нейронной сети исполнитель–критик:

```
>>> class ActorCriticModel(nn.Module):
...     def __init__(self, n_input, n_output, n_hidden):
...         super(ActorCriticModel, self).__init__()
...         self.fc = nn.Linear(n_input, n_hidden)
...         self.action = nn.Linear(n_hidden, n_output)
...         self.value = nn.Linear(n_hidden, 1)
...
...     def forward(self, x):
...         x = torch.Tensor(x)
...         x = F.relu(self.fc(x))
...         action_probs = F.softmax(self.action(x), dim=-1)
...         state_values = self.value(x)
...         return action_probs, state_values
```

3. Напишем метод `__init__` класса `PolicyNetwork` с использованием модели исполнитель–критик:

```
>>> class PolicyNetwork():
...     def __init__(self, n_state, n_action, n_hidden=50, lr=0.001):
...         self.model = ActorCriticModel(n_state, n_action, n_hidden)
...         self.optimizer = torch.optim.Adam(self.model.parameters(), lr)
...         self.scheduler = torch.optim.lr_scheduler.StepLR(
                    self.optimizer, step_size=10, gamma=0.9)
```

Отметим, что мы воспользовались понижателем скорости обучения, который позволяет динамически уменьшать ее в процессе обучения.

4. Добавим метод `predict`, который вычисляет оценки вероятностей действий и ценность состояния:

```
>>>     def predict(self, s):
...         """
...         Вычисляет выход, применяя модель исполнитель-критик
...         @param s: входное состояние
...         @return: вероятности действий, ценность состояния
...         """
...         return self.model(torch.Tensor(s))
```

5. Напишем метод `training`, который обновляет нейронную сеть с учетом примеров, собранных в эпизоде:

```
>>>     def update(self, returns, log_probs, state_values):
...         """
...         Обновляет веса сети исполнитель-критик на основе переданных
...         обучающих примеров
...         @param returns: доход (накопительное вознаграждение) на
...                         каждом шаге эпизода
...         @param log_probs: логарифм вероятности на каждом шаге
...         @param state_values: ценности состояний на каждом шаге
...         """
...         loss = 0
...         for log_prob, value, Gt in zip(log_probs, state_values, returns):
...             advantage = Gt - value.item()
...             policy_loss = -log_prob * advantage
...             value_loss = F.smooth_l1_loss(value, Gt)
...             loss += policy_loss + value_loss
...         self.optimizer.zero_grad()
...         loss.backward()
...         self.optimizer.step()
```

6. Последний метод класса `PolicyNetwork` – `get_action`, он выбирает действие в данном состоянии на основе предсказанной стратегии.

```
>>>     def get_action(self, s):
...         """
...         Предсказывает стратегию, выбирает действие и вычисляет логарифм
...         его вероятности
...         @param s: входное состояние
...         @return: выбранное действие и логарифм его вероятности
...         """
...         action_probs, state_value = self.predict(s)
...         action = torch.multinomial(action_probs, 1).item()
...         log_prob = torch.log(action_probs[action])
...         return action, log_prob, state_value
```

Он также возвращает логарифм вероятности выбранного действия и оценку ценности состояния.

С классом `PolicyNetwork` – все.

7. Теперь можно перейти к разработке главной функции обучения модели исполнитель–критик.

```
>>> def actor_critic(env, estimator, n_episode, gamma=1.0):
...     """
...     Алгоритм исполнитель-критик
...     @param env: имя окружающей среды Gym
...     @param estimator: сеть стратегии
...     @param n_episode: количество эпизодов
...     @param gamma: коэффициент обесценивания
...     """
...     for episode in range(n_episode):
...         log_probs = []
...         rewards = []
...         state_values = []
...         state = env.reset()
...         while True:
...             action, log_prob, state_value = estimator.get_action(state)
...             next_state, reward, is_done, _ = env.step(action)
...             total_reward_episode[episode] += reward
...             log_probs.append(log_prob)
...             state_values.append(state_value)
...             rewards.append(reward)
...
...             if is_done:
...                 returns = []
...                 Gt = 0
...                 pw = 0
...                 for reward in rewards[::-1]:
...                     Gt += gamma ** pw * reward
...                     pw += 1
...                     returns.append(Gt)
...                 returns = returns[::-1]
...                 returns = torch.tensor(returns)
...                 returns = (returns - returns.mean()) /
                                (returns.std() + 1e-9)
...                 estimator.update(returns, log_probs, state_values)
...                 print('Эпизод: {}, полное вознаграждение: {}'.format(
                        episode, total_reward_episode[episode]))
...                 if total_reward_episode[episode] >= 195:
...                     estimator.scheduler.step()
...                 break
...
...             state = next_state
```

8. Зададим форму сети стратегии (размеры входного, выходного и скрытого слоев) и скорость обучения, затем создадим экземпляр класса `PolicyNetwork`:

```
>>> n_state = env.observation_space.shape[0]
>>> n_action = env.action_space.n
>>> n_hidden = 128
```

```
>>> lr = 0.03
>>> policy_net = PolicyNetwork(n_state, n_action, n_hidden, lr)
```

Коэффициент обесценивания положим равным 0.9:

```
>>> gamma = 0.9
```

9. Выполним обучение методом исполнитель–критик с только разработанной сетью стратегии на 1000 эпизодов и будем сохранять полные вознаграждения в каждом эпизоде:

```
>>> n_episode = 1000
>>> total_reward_episode = [0] * n_episode
>>> actor_critic(env, policy_net, n_episode, gamma)
```

10. Построим график зависимости вознаграждения в эпизоде от времени:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(total_reward_episode)
>>> plt.title('Зависимость вознаграждения в эпизоде от времени')
>>> plt.xlabel('Эпизод')
>>> plt.ylabel('Полное вознаграждение')
>>> plt.show()
```

Как это работает

На шаге 2 видно, что исполнитель и критик разделяют общие параметры входного и скрытого слоев. Выходной слой исполнителя порождает вероятности выбора действий, а выходной слой критика – оценку ценности входного состояния.

На шаге 5 вычисляется величина преимущества и ее отрицательное логарифмическое правдоподобие. Функция потерь в алгоритме исполнитель–критик – это сумма отрицательного логарифмического правдоподобия преимущества и среднеквадратической ошибки между доходом и оценкой ценности состояния. Отметим, что мы воспользовались функцией `smooth_l1_loss`, которая равна квадрату разности, если абсолютная величина ошибки меньше 1, а в противном случае равна абсолютной величине.

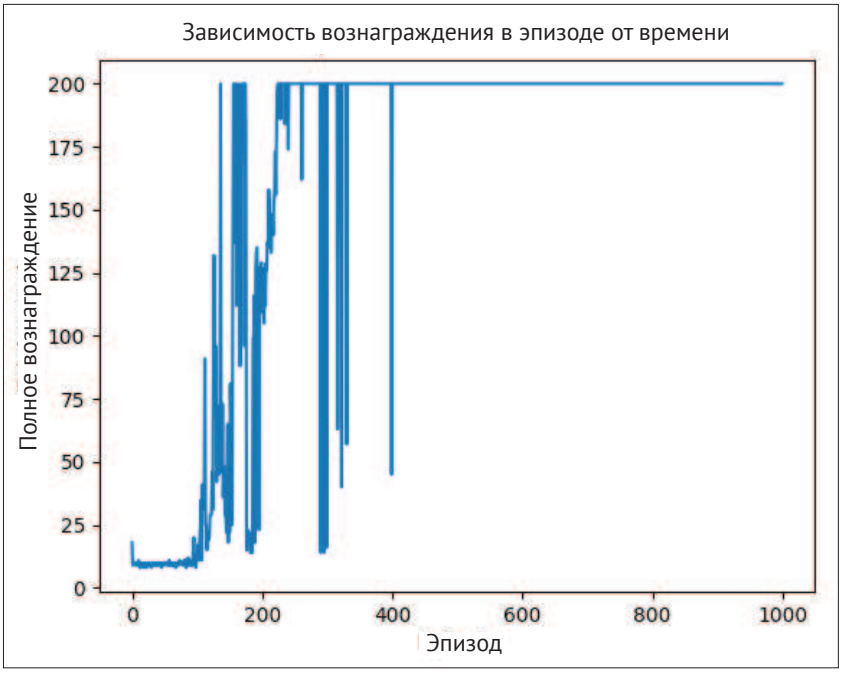
На шаге 7 функция обучения модели исполнитель–критик выполняет следующие действия:

- прогоняется один эпизод: на каждом шаге выбирается действие, следуя текущей оценке стратегии, а также сохраняются вознаграждение, логарифм вероятности и оценка ценности состояния;
- по завершении эпизода вычисляется обесцененный доход на каждом шаге; результирующие доходы нормируются путем вычитания среднего и деления на стандартное отклонение;
- на основе доходов, логарифмов вероятностей и ценностей состояний обновляются параметры стратегии. Кроме того, отображается полное вознаграждение в каждом эпизоде;
- если полное вознаграждение в эпизоде больше +195, то скорость обучения немного уменьшается;
- все вышеперечисленные шаги повторяются в `n_episode` эпизодах.

Ниже показаны сообщения, напечатанные после обучения на шаге 9:

```
Эпизод: 0, полное вознаграждение: 18.0
Эпизод: 1, полное вознаграждение: 9.0
Эпизод: 2, полное вознаграждение: 9.0
Эпизод: 3, полное вознаграждение: 10.0
Эпизод: 4, полное вознаграждение: 10.0
...
...
Эпизод: 995, полное вознаграждение: 200.0
Эпизод: 996, полное вознаграждение: 200.0
Эпизод: 997, полное вознаграждение: 200.0
Эпизод: 998, полное вознаграждение: 200.0
Эпизод: 999, полное вознаграждение: 200.0
```

График, построенный на шаге 10, выглядит следующим образом:



Как видим, после примерно 400 эпизодов вознаграждение устойчиво достигает максимального значения +200. В алгоритме исполнитель–критик обучение распадается на два компонента: исполнитель и критик. Критик в алгоритме A2C вычисляет, насколько хорошим является действие в некотором состоянии, и это подсказывает исполнителю, как реагировать. Преимущество пары состояние–действие вычисляется по формуле $A(s, a) = Q(s, a) - V(s)$, т. е. ценность состояния вычитается из значения Q-функции. Исполнитель оценивает вероятности действия, руководствуясь подсказками критика. Включение преимущества способствует уменьшению дисперсии, поэтому A2C считается

более устойчивым, чем стандартный алгоритм исполнитель–критик. Пример окружающей среды CartPole показывает, что результаты A2C стабилизируются после обучения на нескольких сотнях эпизодов и превосходят результаты REINFORCE с базой.

Решение задачи о блуждании на краю обрыва с помощью алгоритма исполнитель–критик

В этом рецепте мы решим более сложную задачу о блуждании на краю обрыва с помощью алгоритма A2C.

Cliff Walking – типичная окружающая среда Gym с длинными эпизодами без гарантии завершения. Это задача на сетке 4×12. На каждом шаге агент делает ход вверх, вправо, вниз или влево. Вначале агент находится в левом нижнем углу, а для успешного завершения эпизода должен перейти в правый нижний угол. Все остальные ячейки в последней строке – обрыв, при попадании в них агент возвращается на исходную позицию, но эпизод продолжается. За каждый шаг агенту начисляется вознаграждение –1, а за падение с обрыва – вознаграждение –100.

Состояние – целое число от 0 до 47, описывающее местоположение агента, как показано на рисунке ниже.

0	1	2	3	4	5	6	7	8	9	10	11
12	13	14	15	16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44	45	46	47

Никакого внутреннего смысла у числа, описывающего состояние, нет. Например, 30 не означает, что соответствующее состояние в 3 раза чем-то отличается от состояния 10. Поэтому, перед тем как передавать состояние сети стратегии, преобразуем его в унитарный вектор.

Как это делается

1. Импортируем необходимые пакеты и создадим экземпляр окружающей среды Cliff Walking:

```
>>> import gym
>>> import torch
>>> import torch.nn as nn
>>> import torch.nn.functional as F
>>> env = gym.make('CliffWalking-v0')
```

2. Поскольку размерность состояния равна 48, будем использовать более сложную нейронную сеть исполнитель–критик с двумя скрытыми слоями:

```
>>> class ActorCriticModel(nn.Module):
...     def __init__(self, n_input, n_output, n_hidden):
...         super(ActorCriticModel, self).__init__()
...         self.fc1 = nn.Linear(n_input, n_hidden[0])
...         self.fc2 = nn.Linear(n_hidden[0], n_hidden[1])
...         self.action = nn.Linear(n_hidden[1], n_output)
...         self.value = nn.Linear(n_hidden[1], 1)
...
...     def forward(self, x):
...         x = torch.Tensor(x)
...         x = F.relu(self.fc1(x))
...         x = F.relu(self.fc2(x))
...         action_probs = F.softmax(self.action(x), dim=-1)
...         state_values = self.value(x)
...         return action_probs, state_values
```

Как и раньше, исполнитель и критик разделяют общие параметры входного и скрытых слоев.

3. Класс `PolicyNetwork` ничем не отличается от разработанного в рецепте «Реализация алгоритма исполнитель–критик».

4. Напишем главную функцию обучения модели исполнитель–критик. Она почти такая же, как в предыдущем рецепте, только добавляется преобразование состояния в унитарный вектор:

```
>>> def actor_critic(env, estimator, n_episode, gamma=1.0):
...     """
...     Алгоритм исполнитель-критик
...     @param env: имя окружающей среды Gym
...     @param estimator: сеть стратегии
...     @param n_episode: количество эпизодов
...     @param gamma: коэффициент обесценивания
...     """
...     for episode in range(n_episode):
...         log_probs = []
...         rewards = []
...         state_values = []
...         state = env.reset()
...         while True:
...             one_hot_state = [0] * 48
...             one_hot_state[state] = 1
...             action, log_prob, state_value =
                    estimator.get_action(one_hot_state)
...             next_state, reward, is_done, _ = env.step(action)
...             total_reward_episode[episode] += reward
...             log_probs.append(log_prob)
...             state_values.append(state_value)
...             rewards.append(reward)
```

```
...
...             if is_done:
...                 returns = []
...                 Gt = 0
...                 pw = 0
...                 for reward in rewards[::-1]:
...                     Gt += gamma ** pw * reward
...                     pw += 1
...                     returns.append(Gt)
...                 returns = returns[::-1]
...                 returns = torch.tensor(returns)
...                 returns = (returns - returns.mean()) /
                            (returns.std() + 1e-9)
...                 estimator.update(returns, log_probs, state_values)
...                 print('Эпизод: {}, полное вознаграждение: {}'.format(
                        episode, total_reward_episode[episode]))
...                 if total_reward_episode[episode] >= -14:
...                     estimator.scheduler.step()
...                 break
...
...             state = next_state
```

5. Зададим форму сети стратегии (размеры входного, выходного и скрытого слоев) и скорость обучения, затем создадим экземпляр класса `PolicyNetwork`:

```
>>> n_state = 48
>>> n_action = env.action_space.n
>>> n_hidden = [128, 32]
>>> lr = 0.03
>>> policy_net = PolicyNetwork(n_state, n_action, n_hidden, lr)
```

   Коэффициент обесценивания положим равным 0.9:

```
>>> gamma = 0.9
```

6. Выполним обучение методом исполнитель–критик с только что разработанной сетью стратегии на 1000 эпизодов и будем сохранять полные вознаграждения в каждом эпизоде:

```
>>> n_episode = 1000
>>> total_reward_episode = [0] * n_episode
>>> actor_critic(env, policy_net, n_episode, gamma)
```

7. Построим график зависимости вознаграждения в эпизоде от времени, начиная с 100-го эпизода:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(range(100, n_episode), total_reward_episode[100:])
>>> plt.title('Зависимость вознаграждения в эпизоде от времени')
>>> plt.xlabel('Эпизод')
>>> plt.ylabel('Полное вознаграждение')
>>> plt.show()
```

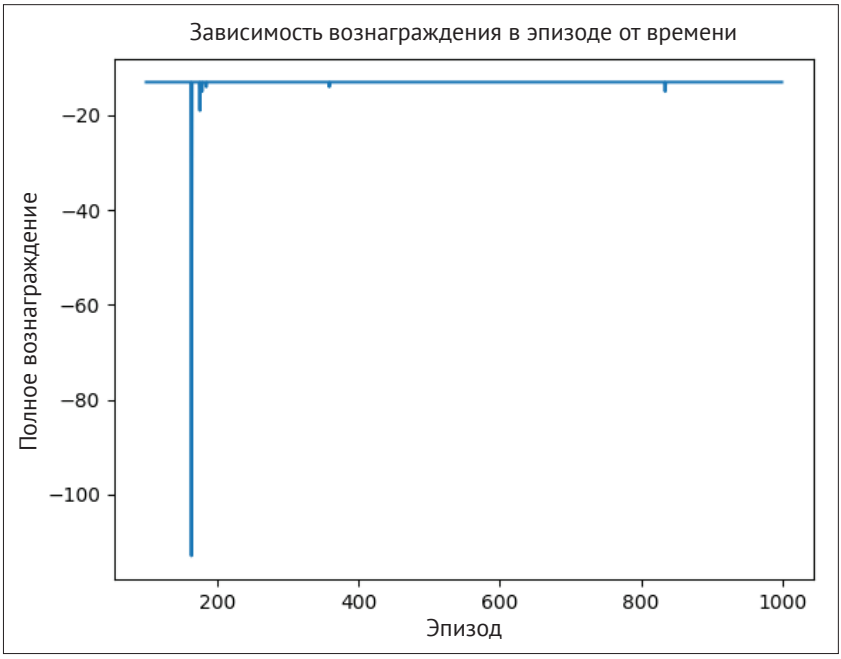
Как это работает

На шаге 4 мы немного уменьшаем скорость обучения, если полное вознаграждение в эпизоде больше –14. Максимально достижимое вознаграждение равно –13, для этого нужно пройти по маршруту 36-24-25-26-27-28-29-30-31-32-33-34-35-47.

Ниже показаны сообщения, напечатанные после обучения на шаге 6:

```
Эпизод: 0, полное вознаграждение: -85355
Эпизод: 1, полное вознаграждение: -3103
Эпизод: 2, полное вознаграждение: -1002
Эпизод: 3, полное вознаграждение: -240
Эпизод: 4, полное вознаграждение: -118
...
...
Эпизод: 995, полное вознаграждение: -13
Эпизод: 996, полное вознаграждение: -13
Эпизод: 997, полное вознаграждение: -13
Эпизод: 998, полное вознаграждение: -13
Эпизод: 999, полное вознаграждение: -13
```

График, построенный на шаге 7, выглядит следующим образом:



Как видим, после примерно 180 эпизодов вознаграждение в большинстве эпизодов достигает максимального значения –13.

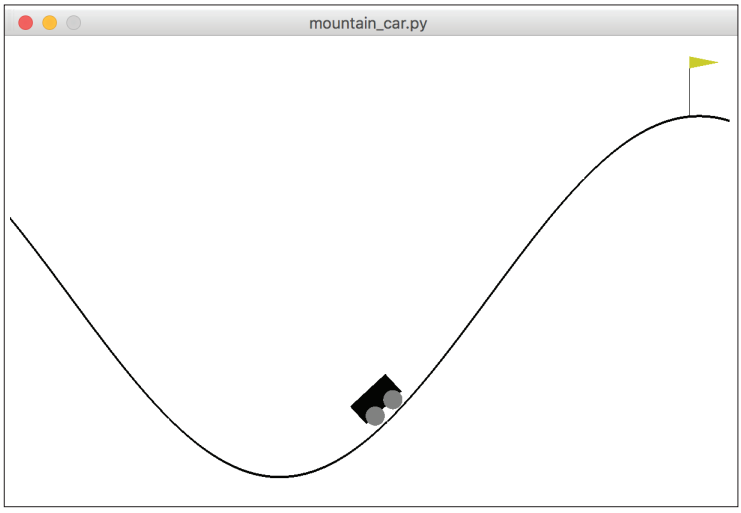
В этом рецепте мы решили задачу о блуждании на краю обрыва, применив алгоритм A2C. Поскольку число от 0 до 47, представляющее положение агента

на сетке 4×12, не имеет никакого внутреннего смысла, мы сначала преобразовали его в унитарный вектор длины 48. Чтобы справиться с 48-мерным входом, мы немного усложнили нейронную сеть, добавив еще один скрытый слой. Как выяснилось, A2C в нашем примере ведет себя устойчиво и находит оптимальную стратегию.

Подготовка непрерывной окружающей среды Mountain Car

До сих пор мы работали с окружающими средами, в которых действия принимают дискретные значения, скажем 0 и 1 для представления действий «вверх» и «вниз». В этом рецепте мы займемся средой Mountain Car с непрерывными действиями.

Continuous Mountain Car (https://github.com/openai/gym/wiki/MountainCarContinuous-v0) – это вариант среды Mountain Car с непрерывными действиями, принимающими значения от –1 до 1 (см. рисунок ниже). Цель – доехать на машине до вершины горы, расположенной справа.



Автомобиль может находиться в диапазоне от –1.2 (слева) до 0.6 (справа), а цель (желтый флажок) находится в точке с абсциссой 0.5. Двигатель автомобиля недостаточно мощный, чтобы преодолеть весь подъем самостоятельно, поэтому необходимо отъехать назад и разогнаться на спуске. Действие представлено числом с плавающей точкой, описывающей силу, толкающую автомобиль влево, если значение принадлежит диапазону от –1 до 0, и вправо, если оно принадлежит диапазону от 0 до 1.

Существует два состояния среды:

- позиция автомобиля: непрерывно изменяется от –1.2 до 0.6;
- скорость автомобиля: непрерывно изменяется от –0.07 до 0.07.

В начальном состоянии позиция находится в диапазоне от –0.6 до –0.4, а скорость равна 0. На каждом шаге начисляется вознаграждение $-a^2$, где a – выбранное действие. За достижение цели начисляется дополнительное вознаграждение +100. Таким образом, все шаги, кроме последнего, штрафуются, и чем их больше, тем меньше будет итоговое вознаграждение. Эпизод заканчивается, когда автомобиль доберется до вершины, или после 1000 шагов.

Как это делается

Для имитации непрерывной окружающей среды Mountain Car выполним следующие действия.

1. Импортируем необходимые пакеты и создадим экземпляр окружающей среды Mountain Car:

```
>>> import gym
>>> import torch
>>> env = gym.envs.make("MountainCarContinuous-v0")
```

2. Изучим пространство действий:

```
>>> print(env.action_space.low[0])
-1.0
>>> print(env.action_space.high[0])
1.0
```

3. Приведем окружающую среду в исходное состояние:

```
>>> env.reset()
array([-0.56756635, 0. ])
```

   Автомобиль начинает движение в состоянии `[-0.56756635, 0.]`, т. е. находится в районе точки –0.56 и имеет скорость 0. На вашем компьютере начальное положение может быть другим, т. к. это случайное число в диапазоне от –0.6 до –0.4.

4. Сначала попробуем наивный подход: будем выбирать случайное действие от –1 до 1:

```
>>> is_done = False
>>> while not is_done:
...     random_action = torch.rand(1) * 2 - 1
...     next_state, reward, is_done, info = env.step(random_action)
...     print(next_state, reward, is_done)
...     env.render()
>>> env.render()
[-0.5657432   0.00182313] -0.09924464356736849 False
[-0.5622848   0.00345837] -0.07744002014160288 False
[-0.55754507  0.00473979] -0.04372991690837722 False
......
......
```

Состояние (позиция и скорость) изменяется в соответствии с действием, и на каждом шаге начисляется вознаграждение $-a^2$.

На видео показано, что машина движется то вправо, то влево.

Как это работает

Понятно, что непрерывная среда Mountain Car – довольно трудная задача, куда труднее первоначальной, где было всего три возможных действия. Мы должны двигаться взад-вперед, меняя направление и прилагая необходимую силу, чтобы разогнать автомобиль до нужной скорости. А поскольку пространство действий непрерывно, поиск в таблице с последующим обновлением (как в TD-методе DQN) не подходит. В следующем рецепте мы решим эту задачу, применив непрерывный вариант алгоритма A2C.

Решение непрерывной задачи о блуждании на краю обрыва методом A2C

В этом рецепте мы решим задачу о блуждании на краю обрыва с помощью алгоритма A2C, естественно, его непрерывной версии. И заодно увидим, чем она отличается от дискретной.

Ранее, при работе в средах с дискретными действиями, мы выбирали действия на основе оценок вероятностей. Но как смоделировать непрерывное управление, если невозможно произвести такую выборку из бесконечного количества действий? Мы можем прибегнуть к нормальному распределению. Предположим, что ценности действий имеют нормальное распределение:

$$\pi(a|s) = N(\mu, \sigma),$$

где среднее μ и стандартное отклонение σ вычисляются сетью стратегии. Тогда можно будет выбирать действия из такого распределения с текущими значениями среднего и стандартного отклонения. Функция потерь в непрерывном A2C аналогична той, что мы раньше использовали в дискретном случае, т. е. является суммой отрицательного логарифмического правдоподобия, вычисленного по вероятностям нормально распределенных действий и преимуществу, и ошибки регрессии между фактически полученным доходом и оценкой ценностей состояний.

Отметим, что нормальное распределение используется для описания одномерных действий, а если пространство действий k-мерное, то нужно будет взять k нормальных распределений. В непрерывной среде Mountain Car пространство действий одномерное. Основная сложность применения A2C к непрерывному управлению заключается в построении сети стратегии, которая вычисляет параметры нормального распределения.

Как это делается

1. Импортируем необходимые пакеты и создадим экземпляр окружающей среды Mountain Car:

```
>>> import gym
>>> import torch
>>> import torch.nn as nn
```

```
>>> import torch.nn.functional as F
>>> env = gym.make('MountainCarContinuous-v0')
```

2. Сначала реализуем нейронную сеть исполнитель–критик:

```
>>> class ActorCriticModel(nn.Module):
...     def __init__(self, n_input, n_output, n_hidden):
...         super(ActorCriticModel, self).__init__()
...         self.fc = nn.Linear(n_input, n_hidden)
...         self.mu = nn.Linear(n_hidden, n_output)
...         self.sigma = nn.Linear(n_hidden, n_output)
...         self.value = nn.Linear(n_hidden, 1)
...         self.distribution = torch.distributions.Normal
...
...     def forward(self, x):
...         x = F.relu(self.fc(x))
...         mu = 2 * torch.tanh(self.mu(x))
...         sigma = F.softplus(self.sigma(x)) + 1e-5
...         dist = self.distribution(mu.view(1, ).data, sigma.view(1, ).data)
...         value = self.value(x)
...         return dist, value
```

3. Напишем метод `__init__` класса `PolicyNetwork` с использованием только что разработанной модели исполнитель–критик:

```
>>> class PolicyNetwork():
...     def __init__(self, n_state, n_action, n_hidden=50, lr=0.001):
...         self.model = ActorCriticModel(n_state, n_action, n_hidden)
...         self.optimizer = torch.optim.Adam(self.model.parameters(), lr)
```

4. Добавим метод `predict`, который вычисляет оценки вероятностей действий и ценность состояния:

```
>>>     def predict(self, s):
...         """
...         Вычисляет выход с использованием непрерывной модели
...         исполнитель-критик
...         @param s: входное состояние
...         @return: выборка из нормального распределения, ценность состояния
...         """
...         self.model.training = False
...         return self.model(torch.Tensor(s))
```

5. Напишем метод `training`, который обновляет сеть стратегии на основе примеров, собранных в эпизоде. Воспользуемся методом `update`, разработанным в рецепте «Реализация алгоритма исполнитель–критик», и не будем здесь повторять его код.

6. И последний метод класса `PolicyNetwork` – `get_action`, он производит для заданного состояния выборку действия из нормального распределения с оцененными параметрами:

```
>>>     def get_action(self, s):
...         """
...         Оценивает стратегию и выбирает действие, вычисляет логарифм его
...         вероятности
```

```
...         @param s: входное состояние
...         @return: выбранное действие, логарифм вероятности,
...                  предсказанная ценность состояния
...         """
...         dist, state_value = self.predict(s)
...         action = dist.sample().numpy()
...         log_prob = dist.log_prob(action[0])
...         return action, log_prob, state_value
```

Метод возвращает логарифм вероятности выбранного действия и оценку ценности состояния.

С классом `PolicyNetwork` для непрерывного управления – все.

Теперь можно перейти к главной функции обучения модели исполнитель–критик.

```
>>> def actor_critic(env, estimator, n_episode, gamma=1.0):
...     """
...     Непрерывный алгоритм исполнитель-критик
...     @param env: имя окружающей среды Gym
...     @param estimator: сеть стратегии
...     @param n_episode: количество эпизодов
...     @param gamma: коэффициент обесценивания
...     """
...     for episode in range(n_episode):
...         log_probs = []
...         rewards = []
...         state_values = []
...         state = env.reset()
...         while True:
...             state = scale_state(state)
...             action, log_prob, state_value = estimator.get_action(state)
...             action = action.clip(env.action_space.low[0],
...                                  env.action_space.high[0])
...             next_state, reward, is_done, _ = env.step(action)
...             total_reward_episode[episode] += reward
...             log_probs.append(log_prob)
...             state_values.append(state_value)
...             rewards.append(reward)
...             if is_done:
...                 returns = []
...                 Gt = 0
...                 pw = 0
...                 for reward in rewards[::-1]:
...                     Gt += gamma ** pw * reward
...                     pw += 1
...                     returns.append(Gt)
...                 returns = returns[::-1]
...                 returns = torch.tensor(returns)
...                 returns = (returns - returns.mean()) /
                                (returns.std() + 1e-9)
...                 estimator.update(returns, log_probs, state_values)
...                 print('Эпизод: {}, полное вознаграждение: {}'.
                        format(episode, total_reward_episode[episode]))
...                 break
...             state = next_state
```

7. Функция scale_state нормирует (стандартизует) входы, чтобы ускорить сходимость модели. Сначала случайным образом генерируется 10 000 наблюдений, на которых обучается нормировщик:

```
>>> import sklearn.preprocessing
>>> import numpy as np
>>> state_space_samples = np.array(
...     [env.observation_space.sample() for x in range(10000)])
>>> scaler = sklearn.preprocessing.StandardScaler()
>>> scaler.fit(state_space_samples)
```

   Обученный нормировщик используется в функции scale_state для преобразования новых данных:

```
>>> def scale_state(state):
... scaled = scaler.transform([state])
... return scaled[0]
```

8. Зададим форму сети стратегии (размеры входного, выходного и скрытого слоев) и скорость обучения, затем создадим экземпляр класса PolicyNetwork:

```
>>> n_state = env.observation_space.shape[0]
>>> n_action = 1
>>> n_hidden = 128
>>> lr = 0.0003
>>> policy_net = PolicyNetwork(n_state, n_action, n_hidden, lr)
```

   Коэффициент обесценивания положим равным 0.9:

```
>>> gamma = 0.9
```

9. Выполним обучение непрерывным методом исполнитель–критик с только что разработанной сетью стратегии на 200 эпизодах и будем сохранять полные вознаграждения в каждом эпизоде:

```
>>> n_episode = 200
>>> total_reward_episode = [0] * n_episode
>>> actor_critic(env, policy_net, n_episode, gamma)
```

10. Построим график зависимости вознаграждения в эпизоде от времени:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(total_reward_episode)
>>> plt.title('Зависимость вознаграждения в эпизоде от времени')
>>> plt.xlabel('Эпизод')
>>> plt.ylabel('Полное вознаграждение')
>>> plt.show()
```

Как это работает

В этом рецепте мы использовали алгоритм гауссова A2C для решения непрерывной задачи о машине на горе.

На шаге использована сеть с одним скрытым слоем. Выходной слой порождает три элемента: среднее и стандартное отклонение нормального распределения и ценность состояния. Среднее распределения приводится к диапазону

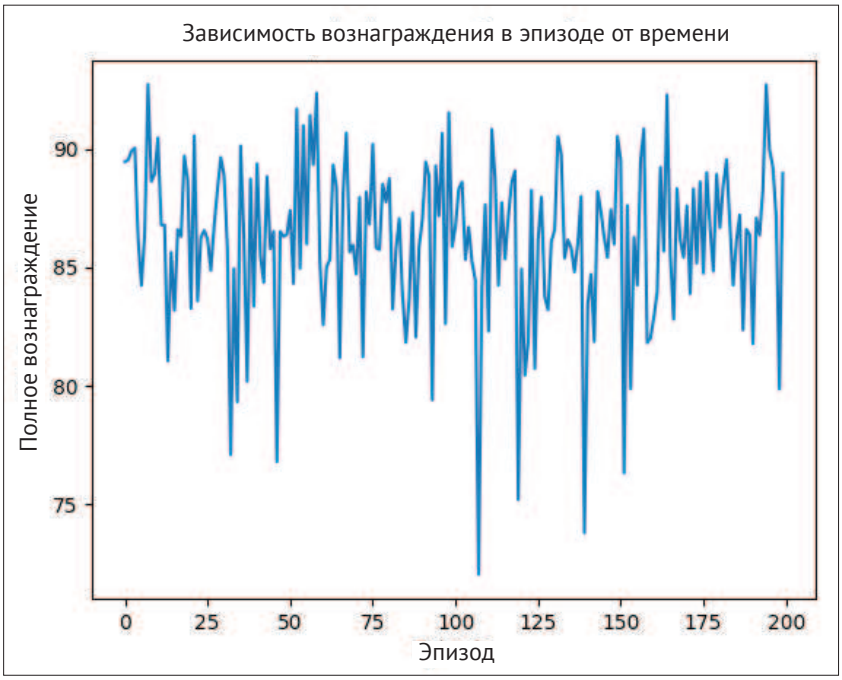
[–1, 1] (или [–2, 2] в нашем примере) с помощью функции активации tanh. Для получения стандартного отклонения мы использовали в качестве функции активации softplus, чтобы результат был гарантированно положительным. Сеть возвращает текущее нормальное распределение (исполнитель) и оценку ценности состояния (критик).

Функция обучения для модели исполнитель–критик, написанная на шаге 7, очень похожа на созданную в рецепте «Реализация алгоритма исполнитель–критик». Мы добавили обрезку выбранного действия, чтобы оно попадало в диапазон [–1, 1]. Что делает функция `scale_state`, мы объясним чуть ниже.

Ниже показаны сообщения, напечатанные после обучения на шаге 9:

```
Эпизод: 0, полное вознаграждение: 89.46417524456328
Эпизод: 1, полное вознаграждение: 89.54226159679301
Эпизод: 2, полное вознаграждение: 89.91828341346695
Эпизод: 3, полное вознаграждение: 90.04199470314816
Эпизод: 4, полное вознаграждение: 86.23157467747066
...
...
Эпизод: 194, полное вознаграждение: 92.71676277432059
Эпизод: 195, полное вознаграждение: 89.97484988523927
Эпизод: 196, полное вознаграждение: 89.26063135086025
Эпизод: 197, полное вознаграждение: 87.19460382302674
Эпизод: 198, полное вознаграждение: 79.86081433777699
Эпизод: 199, полное вознаграждение: 88.98075638481279
```

График, построенный на шаге 10, выглядит следующим образом:



В требованиях к среде (см. https://github.com/openai/gym/wiki/MountainCarContinuous-v0) говорится, что задача считается решенной, если вознаграждение превысило +90. В нескольких эпизодах мы достигли этой цели.

В непрерывном алгоритме A2C предполагается, что распределение по каждому измерению пространства действий нормальное. Среднее и стандартное отклонение выдает выходной слой сети стратегии. Он же формирует оценку ценности состояний. Действие (или набор действий) – результат выборки из нормального распределения (или нескольких таких распределений). Функция потерь аналогична той, что использовалась в дискретной версии, – это сумма отрицательного логарифмического правдоподобия, вычисленного на основе вероятностей нормально распределенных действий и величины преимущества, и ошибки регрессии между фактически полученным доходом и оценкой ценностей состояний.

Это еще не все

До сих пор мы моделировали стратегию стохастически, производя выборку действий из распределения или вычисляя их вероятности. Напоследок мы кратко обсудим алгоритм **детерминированного градиента стратегии** (Deterministic Policy Gradient – **DPG**), когда стратегия моделируется как цепочка детерминированных решений. Мы рассматриваем детерминированную стратегию как частный случай стохастической, когда входные состояния непосредственно отображаются на действия, а не на вероятности действий. В алгоритме DPG используются две нейронные сети:

- **сеть исполнитель–критик.** Она очень похожа на аналогичную сеть в алгоритме A2C, только детерминированна. Эта сеть предсказывает ценности состояния и действия;
- **целевая сеть исполнитель–критик.** Это копия сети исполнитель–критик, которая создается не на каждом шаге, а периодически, чтобы стабилизировать обучение. Очевидно, что нам не нужна постоянно движущаяся мишень, так что эта сеть на некоторое время фиксирует цель для обучения.

Как видим, в алгоритме DPG нового немного, это просто комбинация A2C с механизмом задержки цели. Попробуйте самостоятельно реализовать этот алгоритм и воспользоваться им для взаимодействия с непрерывной средой Mountain Car.

См. также

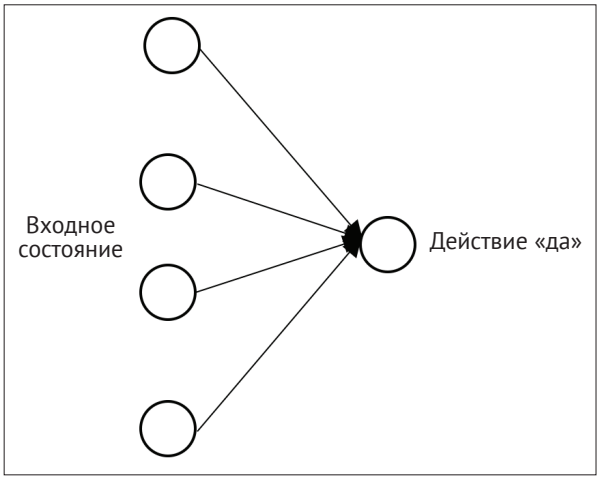
Для тех, кто незнаком с функцией активации softplus или хочет больше узнать об алгоритме DPG, рекомендуем следующие источники:

- **Softplus:** https://en.wikipedia.org/wiki/Rectifier_(neural_networks);
- оригинальная статья по DPG: https://hal.inria.fr/file/index/docid/938992/filename/dpg-icml2014.pdf.

Решение задачи о балансировании стержня методом перекрестной энтропии

В этом последнем, бонусном рецепте мы реализуем простой, но эффективный алгоритм решения задачи в среде CartPole. Он основан на перекрестной энтропии и напрямую отображает входные состояния на выходное действие. Он более прямолинейный, чем прочие алгоритмы градиента стратегии, описанные в этой главе.

Мы применили несколько алгоритмов градиента стратегии к решению задачи о балансировании стержня. В них используются нетривиальные архитектуры нейронных сетей и функции потерь – для такой простой задачи это, пожалуй, перебор. А почему бы не предсказывать действия в заданных состояниях непосредственно? Идея проста: смоделировать отображение состояния на действие и обучать его ТОЛЬКО на самом успешном прошлом опыте. Нас интересует, каким должно быть правильное действие. В данном случае целевой функцией является перекрестная энтропия между фактическим и предсказанным действиями. В среде CartPole возможных действий два: переместить тележку влево или вправо. Для простоты переформулируем проблему как задачу бинарной классификации, которая схематично представлена на рисунке ниже.



Как это делается

1. Импортируем необходимые пакеты и создадим экземпляр окружающей среды CartPole:

```
>>> import gym
>>> import torch
>>> import torch.nn as nn
>>> from torch.autograd import Variable
>>> env = gym.make('CartPole-v0')
```

2. Сначала определим класс для оценивания действий.

```
>>> class Estimator():
...     def __init__(self, n_state, lr=0.001):
...         self.model = nn.Sequential(
...             nn.Linear(n_state, 1),
...             nn.Sigmoid()
...         )
...         self.criterion = torch.nn.BCELoss()
...         self.optimizer = torch.optim.Adam(self.model.parameters(), lr)
...
...     def predict(self, s):
...         return self.model(torch.Tensor(s))
...
...     def update(self, s, y):
...         """
...         Обновляет веса модели на основе обучающих примеров
...         """
...         y_pred = self.predict(s)
...         loss = self.criterion(y_pred, Variable(torch.Tensor(y)))
...         self.optimizer.zero_grad()
...         loss.backward()
...         self.optimizer.step()
```

3. Напишем главную функцию обучения для алгоритма перекрестной энтропии.

```
>>> def cross_entropy(env, estimator, n_episode, n_samples):
...     """
...     Алгоритм перекрестной энтропии для обучения стратегии
...     @param env: имя окружающей среды Gym
...     @param estimator: бинарный оцениватель
...     @param n_episode: количество эпизодов
...     @param n_samples: количество обучающих примеров
...     """
...     experience = []
...     for episode in range(n_episode):
...         rewards = 0
...         actions = []
...         states = []
...         state = env.reset()
...         while True:
...             action = env.action_space.sample()
...             states.append(state)
...             actions.append(action)
...             next_state, reward, is_done, _ = env.step(action)
...             rewards += reward
...             if is_done:
...                 for state, action in zip(states, actions):
...                     experience.append((rewards, state, action))
...                 break
...             state = next_state
...
```

```
...     experience = sorted(experience, key=lambda x: x[0], reverse=True)
...     select_experience = experience[:n_samples]
...     train_states = [exp[1] for exp in select_experience]
...     train_actions = [exp[2] for exp in select_experience]
...
...     for _ in range(100):
...         estimator.update(train_states, train_actions
```

4. Зададим размер входного слоя сети оценивателя действий и скорость обучения:

```
>>> n_state = env.observation_space.shape[0]
>>> lr = 0.01
```

   И создадим экземпляр класса `Estimator`:

```
>>> estimator = Estimator(n_state, lr)
```

5. Сгенерируем 5000 случайных эпизодов и выберем лучшие 10 000 пар (состояние, действие), на которых будем обучать оцениватель:

```
>>> n_episode = 5000
>>> n_samples = 10000
>>> cross_entropy(env, estimator, n_episode, n_samples)
```

6. Протестируем обученную модель на 100 эпизодах и запомним полные вознаграждения:

```
>>> n_episode = 100
>>> total_reward_episode = [0] * n_episode
>>> for episode in range(n_episode):
...     state = env.reset()
...     is_done = False
...     while not is_done:
...         action = 1 if estimator.predict(state).item() >= 0.5 else 0
...         next_state, reward, is_done, _ = env.step(action)
...         total_reward_episode[episode] += reward
...         state = next_state
```

7. Построим график зависимости вознаграждения в эпизоде от времени:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(total_reward_episode)
>>> plt.title('Зависимость вознаграждения в эпизоде от времени')
>>> plt.xlabel('Эпизод')
>>> plt.ylabel('Полное вознаграждение')
>>> plt.show()
```

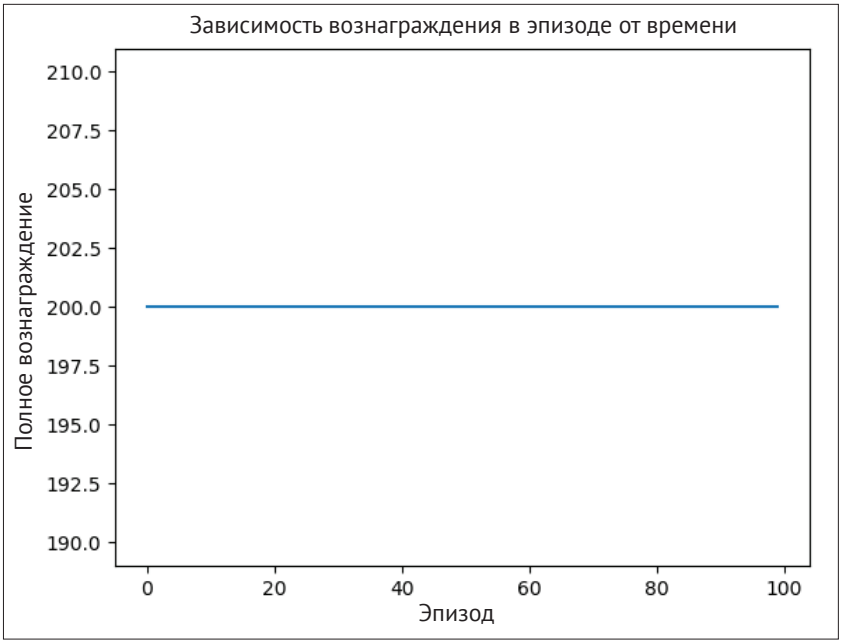
Как это работает

Сеть оценивателя действий на шаге 2 состоит из двух слоев – входного и выходного с сигмоидной функцией активации и перекрестной энтропией в качестве функции потерь.

На шаге 3 обучается модель перекрестной энтропии. Точнее, в каждом эпизоде мы предпринимаем случайные действия, получаем вознаграждения и за-

поминаем состояния и действия. Сгенерировав `n_episode` эпизодов, мы отбираем `n_samples` самых успешных (с максимальным полным вознаграждением) и конструируем из них обучающие примеры в виде пар (состояние, действие). Затем на этом наборе производится 100 итераций обучения оценивателя.

График, построенный на шаге 7, выглядит следующим образом:



Как видим во всех тестовых эпизодах, получено вознаграждение +200!

Метод перекрестной энтропии очень прост, но полезен в простых окружающих средах. Он напрямую моделирует связь между входными состояниями и выходными действиями. Задача управления переформулируется как задача классификации, в которой нужно предсказывать правильное действие из множества возможных. Хитрость в том, чтобы обучаться на подходящем опыте, который говорит модели, какое действие в данном состоянии принесет наибольшее вознаграждение.

Глава 9

Кульминационный проект – применение DQN к игре Flappy Bird

В этой, последней главе мы займемся финальным проектом – применением обучения с подкреплением к игре Flappy Bird. Все, чему мы научились в книге, найдет воплощение в интеллектуальном игровом боте. Мы построим **глубокую Q-сеть** (**DQN**), настроим параметры модели и развернем ее. А потом посмотрим, как долго птица сможет оставаться в воздухе.

Наш кульминационный проект будет последовательно изложен в следующих рецептах:

- подготовка игровой среды;
- построение глубокой Q-сети для игры Flappy Bird;
- обучение и настройка сети;
- развертывание модели и игра.

Подготовка игровой среды

Чтобы DQN-сеть могла играть в Flappy Bird, нужно сначала подготовить окружающую среду.

Для имитации Flappy Bird мы воспользуемся пакетом Pygame (https://www.pygame.org), который содержит модули, предназначенные для разработки видеоигр, а также графические и звуковые библиотеки. Для установки Pygame выполните команду

```
pip install pygame
```

Flappy Bird – знаменитая игра для мобильных устройств, ее создал Донг Нгуен. Можете поиграть в нее с помощью клавиатуры на сайте https://flappybird.io/. Цель игры – оставаться в воздухе как можно дольше. Игра заканчивается, когда птица касается пола или трубы. Таким образом, птица должна взмахи-

вать крыльями в определенные моменты времени, чтобы обогнуть случайно расставленные трубы и не упасть на землю. Возможных действий всего два – махать и не махать крыльями. В игровой окружающей среде за каждый временной шаг начисляется вознаграждение +1, с двумя исключениями:

- –1 в случае столкновения;
- +1, когда птица пролетает в просвет между двумя трубами. В оригинальной игре Flappy Bird счет зависел от количества преодоленных просветов.

Подготовка

Скачайте ресурсы игровой среды с сайта https://github.com/yanpanlau/Keras-FlappyBird/tree/master/assets/sprites. Для простоты мы будем использовать только изображения в каталоге `sprites`. Точнее, нам понадобятся следующие изображения:

- `background-black.png`: фон на экране;
- `base.png`: изображение поля;
- `pipe-green.png`: изображение труб, от которых птица должна держаться подальше;
- `redbird-downflap.png`: изображение птицы с опущенными крыльями;
- `redbird-midflap.png`: изображение птицы с горизонтально расположенными крыльями;
- `redbird-upflap.png`: изображение птицы с поднятыми крыльями.

Если хотите, можете скачать еще и звуковые файлы, чтобы было интереснее играть.

Как это делается

Подготовим окружающую среду для игры Flappy Bird с использованием пакета Pygame.

1. Начнем со служебной функции, которая загружает изображения и преобразует их в нужный формат.

```
>>> from pygame.image import load
>>> from pygame.surfarray import pixels_alpha
>>> from pygame.transform import rotate
>>> def load_images(sprites_path):
...     base_image = load(sprites_path + 'base.png').convert_alpha()
...     background_image = load(sprites_path + 'background-black.png').convert()
...     pipe_images = [rotate(load(sprites_path +
                'pipe-green.png').convert_alpha(), 180),
...                       load(sprites_path +
                'pipe-green.png').convert_alpha()]
...     bird_images = [load(sprites_path +
                'redbird-upflap.png').convert_alpha(),
...                       load(sprites_path +
                'redbird-midflap.png').convert_alpha(),
...                       load(sprites_path +
```

```
                'redbird-downflap.png').convert_alpha()]
...     bird_hitmask = [pixels_alpha(image).astype(bool)
                        for image in bird_images]
...     pipe_hitmask = [pixels_alpha(image).astype(bool)
                        for image in pipe_images]
...     return base_image, background_image, pipe_images,
               bird_images, bird_hitmask, pipe_hitmask
```

2. Импортируем необходимые пакеты:

```
>>> from itertools import cycle
>>> from random import randint
>>> import pygame
```

3. Инициализируем игру и таймер и зададим частоту обновления экрана 30 кадров/с:

```
>>> pygame.init()
>>> fps_clock = pygame.time.Clock()
>>> fps = 30
```

4. Зададим размер экрана, создадим экран и поместим на него надпись:

```
>>> screen_width = 288
>>> screen_height = 512
>>> screen = pygame.display.set_mode((screen_width, screen_height))
>>> pygame.display.set_caption('Flappy Bird')
```

5. Загрузим необходимые изображения (из папки `sprites`):

```
>>> base_image, background_image, pipe_images, bird_images,
bird_hitmask, pipe_hitmask = load_images('sprites/')
```

6. Получим параметры игры, в т. ч. размер птицы и труб, и зададим вертикальный просвет между трубами 100:

```
>>> bird_width = bird_images[0].get_width()
>>> bird_height = bird_images[0].get_height()
>>> pipe_width = pipe_images[0].get_width()
>>> pipe_height = pipe_images[0].get_height()
>>> pipe_gap_size = 100
```

7. Птица взмахивает крыльями в порядке вверх, горизонтально, вниз, горизонтально, вверх и т. д.

```
>>> bird_index_gen = cycle([0, 1, 2, 1])
```

   Это нужно только для того, чтобы игра смотрелась веселее.

8. Итак, все константы определены и можно перейти к методу `__init__` класса `FlappyBird`:

```
>>> class FlappyBird(object):
...     def __init__(self):
...         self.pipe_vel_x = -4
...         self.min_velocity_y = -8
...         self.max_velocity_y = 10
...         self.downward_speed = 1
```

```
...         self.upward_speed = -9
...         self.cur_velocity_y = 0
...         self.iter = self.bird_index = self.score = 0
...         self.bird_x = int(screen_width / 5)
...         self.bird_y = int((screen_height - bird_height) / 2)
...         self.base_x = 0
...         self.base_y = screen_height * 0.79
...         self.base_shift = base_image.get_width() -
                    background_image.get_width()
...         self.pipes = [self.gen_random_pipe(screen_width),
                    self.gen_random_pipe(screen_width * 1.5)]
...         self.is_flapped = False
```

9. Определим метод `gen_random_pipe`, который создает пару труб (верхнюю и нижнюю) в заданной горизонтальной и случайно выбранной вертикальной позициях.

```
>>>     def gen_random_pipe(self, x):
...         gap_y = randint(2, 10) * 10 + int(self.base_y * 0.2)
...         return {"x_upper": x,
...                 "y_upper": gap_y - pipe_height,
...                 "x_lower": x,
...                 "y_lower": gap_y + pipe_gap_size}
```

Верхняя труба начинается в точке с ординатой `gap_y - pipe_height`, а нижняя – в точке с ординатой `gap_y + pipe_gap_size`.

10. Напишем метод `check_collision`, который возвращает `True`, если птица упала на землю или столкнулась с трубой.

```
>>>     def check_collision(self):
...         if bird_height + self.bird_y >= self.base_y - 1:
...             return True
...         bird_rect = pygame.Rect(self.bird_x, self.bird_y,
                                    bird_width, bird_height)
...         for pipe in self.pipes:
...             pipe_boxes = [pygame.Rect(pipe["x_upper"],
                    pipe["y_upper"], pipe_width, pipe_height),
...                           pygame.Rect(pipe["x_lower"],
                    pipe["y_lower"], pipe_width, pipe_height)]
...             # Проверить, пересекается ли ограничивающий птицу прямоугольник с
...             # с прямоугольниками, ограничивающими трубы
...             if bird_rect.collidelist(pipe_boxes) == -1:
...                 return False
...             for i in range(2):
...                 cropped_bbox = bird_rect.clip(pipe_boxes[i])
...                 x1 = cropped_bbox.x - bird_rect.x
...                 y1 = cropped_bbox.y - bird_rect.y
...                 x2 = cropped_bbox.x - pipe_boxes[i].x
...                 y2 = cropped_bbox.y - pipe_boxes[i].y
...                 for x in range(cropped_bbox.width):
...                     for y in range(cropped_bbox.height):
...                         if bird_hitmask[self.bird_index][x1+x, y1+y]
                                and pipe_hitmask[i][x2+x, y2+y]:
```

```
...                             return True
...         return False
```

11. Последний и самый важный метод – `next_step`, он выполняет действие и возвращает обновленный кадр игры, полученное вознаграждение и признак завершения эпизода.

```
>>>     def next_step(self, action):
...         pygame.event.pump()
...         reward = 0.1
...         if action == 1:
...             self.cur_velocity_y = self.upward_speed
...             self.is_flapped = True
...         # Обновить счет
...         bird_center_x = self.bird_x + bird_width / 2
...         for pipe in self.pipes:
...             pipe_center_x = pipe["x_upper"] + pipe_width / 2
...             if pipe_center_x < bird_center_x < pipe_center_x + 5:
...                 self.score += 1
...                 reward = 1
...                 break
...         # Обновить индекс изображения птицы и номер итерации
...         if (self.iter + 1) % 3 == 0:
...             self.bird_index = next(bird_index_gen)
...         self.iter = (self.iter + 1) % fps
...         self.base_x = -((-self.base_x + 100) % self.base_shift)
...         # Обновить позицию птицы
...         if self.cur_velocity_y < self.max_velocity_y
                        and not self.is_flapped:
...             self.cur_velocity_y += self.downward_speed
...             self.is_flapped = False
...             self.bird_y += min(self.cur_velocity_y,
                        self.bird_y - self.cur_velocity_y - bird_height)
...         if self.bird_y < 0:
...             self.bird_y = 0
...         # Обновить положение труб
...         for pipe in self.pipes:
...             pipe["x_upper"] += self.pipe_vel_x
...             pipe["x_lower"] += self.pipe_vel_x
...         # Добавить новую трубу, перед тем как первая уйдет за левый
...         # край экрана
...         if 0 < self.pipes[0]["x_lower"] < 5:
...             self.pipes.append(self.gen_random_pipe(screen_width + 10))
...         # удалить первую трубу, если она не видна на экране
...         if self.pipes[0]["x_lower"] < -pipe_width:
...             self.pipes.pop(0)
...         if self.check_collision():
...             is_done = True
...             reward = -1
...             self.__init__()
...         else:
```

```
...             is_done = False
...         # Нарисовать спрайты
...         screen.blit(background_image, (0, 0))
...         screen.blit(base_image, (self.base_x, self.base_y))
...         screen.blit(bird_images[self.bird_index],
                        (self.bird_x, self.bird_y))
...         for pipe in self.pipes:
...             screen.blit(pipe_images[0], (pipe["x_upper"], pipe["y_upper"]))
...             screen.blit(pipe_images[1], (pipe["x_lower"], pipe["y_lower"]))
...         image = pygame.surfarray.array3d(pygame.display.get_surface())
...         pygame.display.update()
...         fps_clock.tick(fps)
...         return image, reward, is_done
```

И на этом с окружающей средой Flappy Bird покончено.

Как это работает

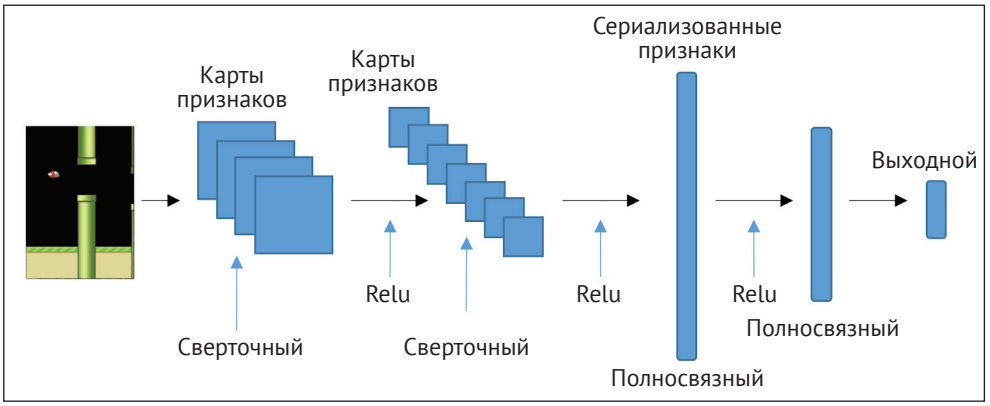
На шаге 8 мы определили скорость движения труб (на 4 единицы влево за единицу времени), минимальную и максимальную вертикальную скорость птицы при движении вверх и вниз (–8 и 10), ускорение в направлении вверх и вниз (–9 и 1), вертикальную скорость по умолчанию (0), начальный индекс изображения птицы (0), начальный счет, начальную горизонтальную и вертикальную позицию птицы, положение земли, а также координаты труб, которые случайно генерируются методом `gen_random_pipe`.

На шаге 11 определено вознаграждение по умолчанию на каждом шаге, равное +0.1. Если действием является взмах крыльями, то мы увеличиваем вертикальную скорость, прибавив ускорение движения вверх. Затем проверяется, проскочила ли птица между трубами. Если да, то счет в игре увеличивается на 1, а за шаг начисляется вознаграждение +1. Мы обновляем позицию птицы, индекс ее изображения и позиции труб. Новая пара труб генерируется, если старая вот-вот выйдет за левый край экрана, а когда это происходит, старая пара труб стирается с экрана. Если птица столкнулась с трубой, эпизод заканчивается, начисляется вознаграждение –1, и игра сбрасывается в исходное состояние. И напоследок обновляется кадр на экране.

Построение глубокой Q-сети для игры Flappy Bird

Подготовив окружающую среду Flappy Bird, приступим к построению DQN-модели.

Как мы видели, на каждом шаге возвращается изображение на экране после совершения действия. Для работы с изображениями лучше всего приспособлены сверточные нейронные сети (СНС). Сверточные слои СНС эффективно выделяют из изображений признаки, которые затем передаются полносвязным слоям. Мы будем использовать СНС с тремя сверточными слоями и одним полносвязным скрытым слоем. Архитектура сети показана на рисунке ниже.



Как это делается

Построим DQN-модель на основе CHC.

1. Импортируем необходимые модули:

```
>>> import torch
>>> import torch.nn as nn
>>> import torch.nn.functional as F
>>> import numpy as np
>>> import random
```

2. Начнем с модели CHC.

```
>>> class DQNModel(nn.Module):
...     def __init__(self, n_action=2):
...         super(DQNModel, self).__init__()
...         self.conv1 = nn.Conv2d(4, 32, kernel_size=8, stride=4)
...         self.conv2 = nn.Conv2d(32, 64, 4, stride=2)
...         self.conv3 = nn.Conv2d(64, 64, 3, stride=1)
...         self.fc = nn.Linear(7 * 7 * 64, 512)
...         self.out = nn.Linear(512, n_action)
...         self._create_weights()
...
...     def _create_weights(self):
...         for m in self.modules():
...             if isinstance(m, nn.Conv2d) or isinstance(m, nn.Linear):
...                 nn.init.uniform(m.weight, -0.01, 0.01)
...                 nn.init.constant_(m.bias, 0)
...
...     def forward(self, x):
...         x = F.relu(self.conv1(x))
...         x = F.relu(self.conv2(x))
...         x = F.relu(self.conv3(x))
...         x = x.view(x.size(0), -1)
...         x = F.relu(self.fc(x))
...         output = self.out(x)
...         return output
```

3. Затем создадим DQN с воспроизведением опыта, воспользовавшись только что построенной моделью CHC:

```
>>> class DQN():
...     def __init__(self, n_action, lr=1e-6):
...         self.criterion = torch.nn.MSELoss()
...         self.model = DQNModel(n_action)
...         self.optimizer = torch.optim.Adam(self.model.parameters(), lr)
```

4. Метод `predict` оценивает выходные значения Q-функции в заданном состоянии:

```
>>>     def predict(self, s):
...         """
...         Вычисляет значения Q-функции для всех действий в заданном
...         состоянии, применяя обученную модель
...         @param s: входное состояние
...         @return: значения Q для всех действий в состоянии s
...         """
...         return self.model(torch.Tensor(s))
```

5. Метод `update` обновляет веса нейронной сети на основе обучающего примера и возвращает текущую потерю:

```
>>>     def update(self, y_predict, y_target):
...         """
...         Обновляет веса DQN на основе обучающего примера
...         @param y_predict: предсказанное значение
...         @param y_target: целевое значение
...         @return: потеря
...         """
...         loss = self.criterion(y_predict, y_target)
...         self.optimizer.zero_grad()
...         loss.backward()
...         self.optimizer.step()
...         return loss
```

6. И последняя часть класса `DQN` – метод `replay`, который воспроизводит опыт из буфера.

```
>>>     def replay(self, memory, replay_size, gamma):
...         """
...         Воспроизведение опыта
...         @param memory: буфер воспроизведения опыта
...         @param replay_size: сколько примеров использовать при каждом
...                   обновлении модели
...         @param gamma: коэффициент обесценивания
...         @return: потеря
...         """
...         if len(memory) >= replay_size:
...             replay_data = random.sample(memory, replay_size)
...             state_batch, action_batch, next_state_batch,
                    reward_batch, done_batch = zip(*replay_data)
...             state_batch = torch.cat(
```

```
                    tuple(state for state in state_batch))
...             next_state_batch = torch.cat(
                    tuple(state for state in next_state_batch))
...             q_values_batch = self.predict(state_batch)
...             q_values_next_batch = self.predict(next_state_batch)
...             reward_batch = torch.from_numpy(np.array(
                    reward_batch, dtype=np.float32)[:, None])
...             action_batch = torch.from_numpy(
...                 np.array([[1, 0] if action == 0 else [0, 1]
                    for action in action_batch], dtype=np.float32))
...             q_value = torch.sum(
                    q_values_batch * action_batch, dim=1)
...             td_targets = torch.cat(
...                 tuple(reward if terminal else reward +
                            gamma * torch.max(prediction) for
                            reward, terminal, prediction
...                     in zip(reward_batch, done_batch,
                            q_values_next_batch)))
...             loss = self.update(q_value, td_targets)
...             return loss
```

С классом `DQN` мы тоже разобрались. В следующем рецепте мы обучим модель `DQN`.

Как это работает

На шаге 2 мы собрали сеть DQN на основе CHC. Нейронная сеть содержит три сверточных слоя с разными конфигурациями. Функцией активации в каждом слое является ReLU. Карта признаков, созданная последним сверточным слоем, сериализуется и передается полносвязному слою с 512 блоками, за которым следует выходной слой.

Отметим также, что мы задали границы начальных случайных значений весов и нулевое смещение, чтобы ускорить сходимость модели.

На шаге 6 выполняется обучение с буфером воспроизведения опыта. Если опыта достаточно, то производится случайная выборка размера `replay_size`, используемая для порождения обучающего набора. Каждый элемент выборки преобразуется в обучающий пример, состоящий из предсказанной и целевой ценности заданного входного состояния. Целевые ценности вычисляются следующим образом:

- обновить целевое значение Q-функции для действия, используя вознаграждение и новые значения Q, по формуле $r + \gamma max_{a'} Q(s', a')$;
- если состояние заключительное, то целевое значение Q принимается равным r.

И наконец, мы обновляем нейронную сеть, применив сформированный пакет обучающих примеров.

Обучение и настройка сети

В этом рецепте мы обучим модель DQN играть в Flappy Bird.

На каждом шаге обучения мы выбираем действие, следуя ε-жадной стратегии: с вероятностью ε выбирается случайное действие – махать или не махать крыльями, а с вероятностью 1 – ε – действие с наибольшей известной на данный момент ценностью. На каждом шаге значение ε корректируется, так что в начале обучения исследованию отдается большее предпочтение, чем в конце, когда модель DQN становится более зрелой.

Мы видели, что наблюдением на каждом шаге является двумерное изображение на экране. Эти изображения необходимо преобразовать в состояния. Одно изображение не дает достаточно информации, чтобы направить агента в нужном направлении. Поэтому состояние складывается из четырех последовательных изображений. Сначала мы уменьшим изображение до нужного размера, а затем конкатенируем его с предыдущими.

Как это делается

1. Импортируем необходимые модули:

```
>>> import random
>>> import torch
>>> from collections import deque
```

2. Реализуем ε-жадную стратегию:

```
>>> def gen_epsilon_greedy_policy(estimator, epsilon, n_action):
...     def policy_function(state):
...         if random.random() < epsilon:
...             return random.randint(0, n_action - 1)
...         else:
...             q_values = estimator.predict(state)
...         return torch.argmax(q_values).item()
...     return policy_function
```

3. Зададим размер предобработанного изображения, размер пакета, скорость обучения, коэффициент обесценивания, количество действий, начальное и конечное значения ε, количество итераций и размер буфера воспроизведения:

```
>>> image_size = 84
>>> batch_size = 32
>>> lr = 1e-6
>>> gamma = 0.99
>>> init_epsilon = 0.1
>>> final_epsilon = 1e-4
>>> n_iter = 2000000
>>> memory_size = 50000
>>> n_action = 2
```

Будем периодически сохранять обученную модель, потому что процесс обучения долгий:

```
>>> saved_path = 'trained_models'
```

Не забудьте создать папку с именем `trained_models`.

4. Зададим начальное значение генератора случайных чисел, чтобы можно было воспроизвести результаты:

```
>>> torch.manual_seed(123)
```

5. Создадим модель DQN:

```
>>> estimator = DQN(n_action)
```

и буфер воспроизведения опыта:

```
>>> memory = deque(maxlen=memory_size)
```

Новые примеры будут добавляться в конец очереди, а старые удаляться, когда общее число примеров превысит 50 000.

6. Инициализируем окружающую среду Flappy Bird:

```
>>> env = FlappyBird()
```

Получим начальное изображение:

```
>>> image, reward, is_done = env.next_step(0)
```

7. Уменьшим исходное изображение до размера `image_size * image_size`:

```
>>> import cv2
>>> import numpy as np
>>> def pre_processing(image, width, height):
...     image = cv2.cvtColor(cv2.resize(image,
                (width, height)), cv2.COLOR_BGR2GRAY)
...     _, image = cv2.threshold(image, 1, 255, cv2.THRESH_BINARY)
...     return image[None, :, :].astype(np.float32)
```

Если пакет `cv2` не установлен, сделайте это командой

```
pip install opencv-python
```

Подвергнем изображение предварительной обработке:

```
>>> image = pre_processing(image[:screen_width, :int(env.base_y)],
image_size, image_size)
```

8. Теперь сконструируем состояние – результат конкатенации четырех изображений. Пока у нас имеется только первый кадр, поэтому повторим его четыре раза:

```
>>> image = torch.from_numpy(image)
>>> state = torch.cat(tuple(image for _ in range(4)))[None, :, :, :]
```

9. Выполним `n_iter` итераций цикла обучения:

```
>>> for iter in range(n_iter):
...     epsilon = final_epsilon + (n_iter - iter)
                * (init_epsilon - final_epsilon) / n_iter
```

```
...     policy = gen_epsilon_greedy_policy(
                    estimator, epsilon, n_action)
...     action = policy(state)
...     next_image, reward, is_done = env.next_step(action)
...     next_image = pre_processing(next_image[
                :screen_width, :int(env.base_y)], image_size, image_size)
...     next_image = torch.from_numpy(next_image)
...     next_state = torch.cat((
                state[0, 1:, :, :], next_image))[None, :, :, :]
...     memory.append([state, action, next_state, reward, is_done])
...     loss = estimator.replay(memory, batch_size, gamma)
...     state = next_state
...     print("Итерация: {}/{}, Действие: {},
               Потеря: {}, Epsilon {}, Вознаграждение: {}".format(
...             iter + 1, n_iter, action, loss, epsilon, reward))
...     if iter+1 % 10000 == 0:
...         torch.save(estimator.model, "{}/{}".format(saved_path, iter+1))
```

В результате выполнения этой части кода будут напечатаны такие сообщения:

```
Итерация: 1/2000000, Действие: 0, Потеря: None, Epsilon 0.1, Вознаграждение: 0.1
Итерация: 2/2000000, Действие: 0, Потеря: None, Epsilon
 0.09999995005000001, Вознаграждение: 0.1
Итерация: 3/2000000, Действие: 0, Потеря: None, Epsilon 0.0999999001,
 Вознаграждение: 0.1
Итерация: 4/2000000, Действие: 0, Потеря: None, Epsilon
 0.09999985015, Вознаграждение: 0.1
...
...
Итерация: 201/2000000, Действие: 1, Потеря: 0.040504034608602524,
 Epsilon 0.09999001000000002, Вознаграждение: 0.1
Итерация: 202/2000000, Действие: 1, Потеря: 0.010011588223278522,
 Epsilon 0.09998996005, Вознаграждение: 0.1
Итерация: 203/2000000, Действие: 1, Потеря: 0.07097195833921432,
 Epsilon 0.09998991010000001, Вознаграждение: 0.1
Итерация: 204/2000000, Действие: 1, Потеря: 0.040418840944767,
 Epsilon 0.09998986015000001, Вознаграждение: 0.1
Итерация: 205/2000000, Действие: 1, Потеря: 0.00999421812593937,
 Epsilon 0.09998981020000001, Вознаграждение: 0.1
```

Обучение занимает довольно длительное время. Конечно, его можно ускорить, воспользовавшись GPU.

10. И наконец, сохраним последнюю обученную модель:

```
>>> torch.save(estimator.model, "{}/final".format(saved_path))
```

Как это работает

На шаге 9 выполняются следующие действия:

- немного уменьшается ε и создается ε-жадная стратегия;
- следуя ε-жадной стратегии, выбирается действие;

- результирующее изображение подвергается предварительной обработке, после чего путем конкатенации четырех последовательных изображений конструируется новое состояние;
- сохраняется опыт, полученный на данной итерации, – состояния, действие, следующее состояние, полученное вознаграждение, признак окончания эпизода;
- модель обновляется с помощью буфера воспроизведения;
- печатается состояние обучения и обновляется состояние;
- обученная модель периодически сохраняется, чтобы не переобучать ее с самого начала в случае сбоя.

Развертывание модели и игра

Итак, DQN-модель обучена, применим ее к игре Flappy Bird. Сделать это просто. Нужно лишь выбирать на каждом шаге действие с наибольшей ценностью. Мы сыграем несколько игр и посмотрим на результаты. Не забывайте подвергать изображение на экране предварительной обработке и конструировать состояние.

Как это делается

Протестируем DQN-модель на нескольких новых эпизодах.

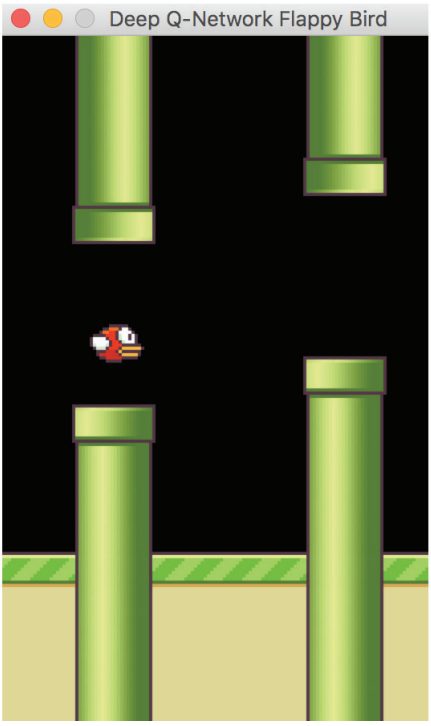
1. Сначала загрузим финальную модель:

```
>>> model = torch.load("{}/final".format(saved_path))
```

2. Прогоним 100 эпизодов и в каждом выполним показанный ниже код:

```
>>> n_episode = 100
>>> for episode in range(n_episode):
...     env = FlappyBird()
...     image, reward, is_done = env.next_step(0)
...     image = pre_processing(image[:screen_width,
                :int(env.base_y)], image_size, image_size)
...     image = torch.from_numpy(image)
...     state = torch.cat(tuple(image for _ in range(4)))[None, :, :, :]
...     while True:
...         prediction = model(state)[0]
...         action = torch.argmax(prediction).item()
...         next_image, reward, is_done = env.next_step(action)
...         if is_done:
...             break
...         next_image = pre_processing(next_image[:screen_width,
                :int(env.base_y)], image_size, image_size)
...         next_image = torch.from_numpy(next_image)
...         next_state = torch.cat((state[0, 1:, :, :],
                next_image))[None, :, :, :]
...         state = next_state
```

Если все нормально, то мы увидим, как птица пролетает между трубами, как показано на рисунке ниже:



Как это работает

На шаге 2 в каждом эпизоде выполняются следующие действия:

- инициализируется окружающая среда Flappy Bird;
- наблюдается начальное изображение, и на его основе генерируется состояние;
- для этого состояния с помощью модели вычисляются значения Q-функции и выбирается действие с наибольшей ценностью;
- наблюдается новое изображение и определяется, закончился эпизод или еще нет;
- если эпизод продолжается, вычисляется новое состояние, которое становится текущим;
- эти действия повторяются до завершения эпизода.

Предметный указатель

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38;
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.

Юси (Хэйден) Лю

Обучение с подкреплением на PyTorch: сборник рецептов

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Слинкин А. А.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура PT Serif. Печать офсетная.
Усл. печ. л. 22,91. Тираж 200 экз.

Отпечатано в ПАО «Т8 Издательские Технологии»
109316, Москва, Волгоградский проспект, д. 42, корпус 5

Веб-сайт издательства: **www.dmkpress.com**