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
```

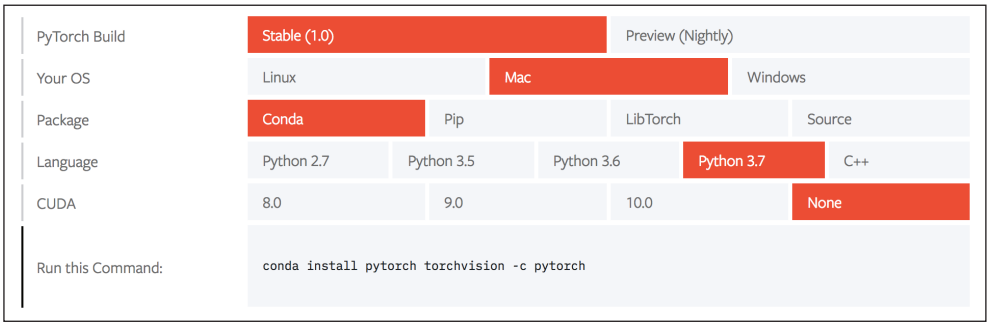
Должно появиться приглашение Python с упоминанием Anaconda:

```
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.

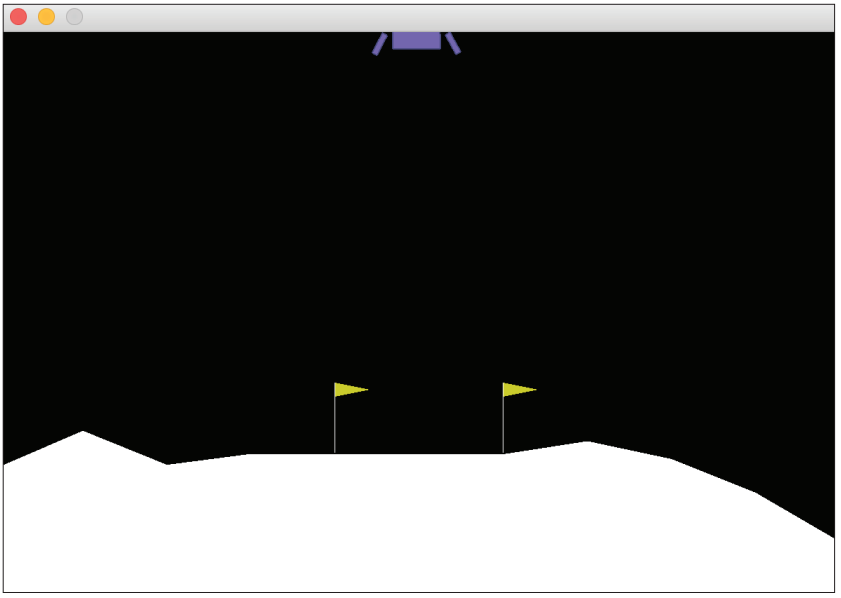
Взять, к примеру, среды Box2d. Чтобы можно было экспериментировать с ними, нужно сначала установить зависимости `Box2d`. Тут тоже есть два способа:

```
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:

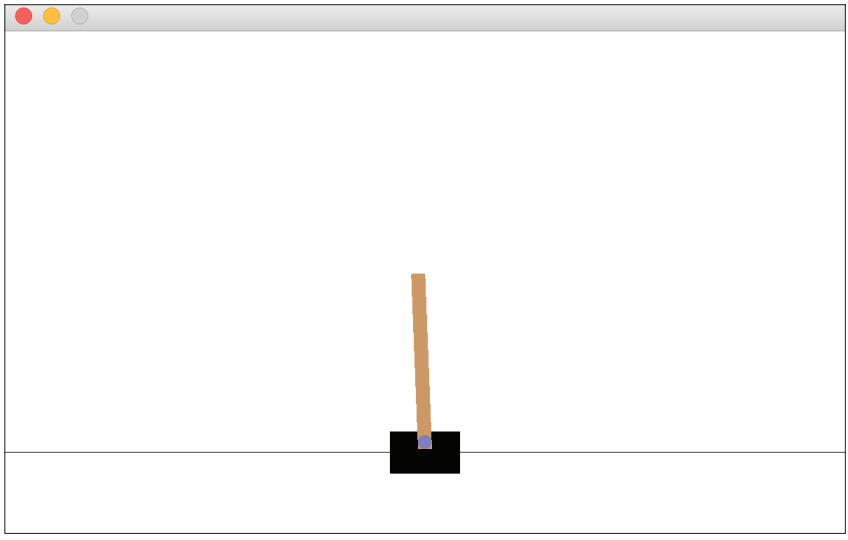
```
>>> env.reset()
array([-0.00153354, 0.01961605, -0.03912845, -0.01850426])
```

   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:

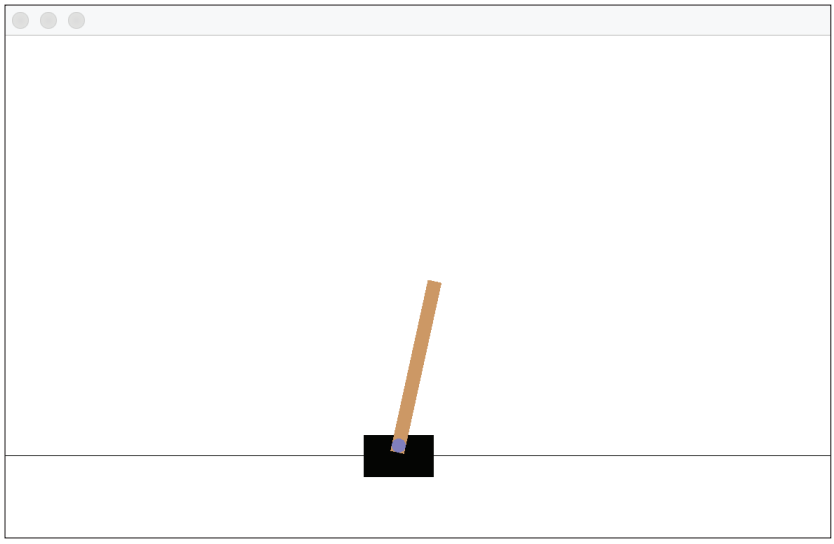


5. Теперь войдем в цикл `while` и позволим агенту сделать столько ходов, сколько он сможет:

```
>>> 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.

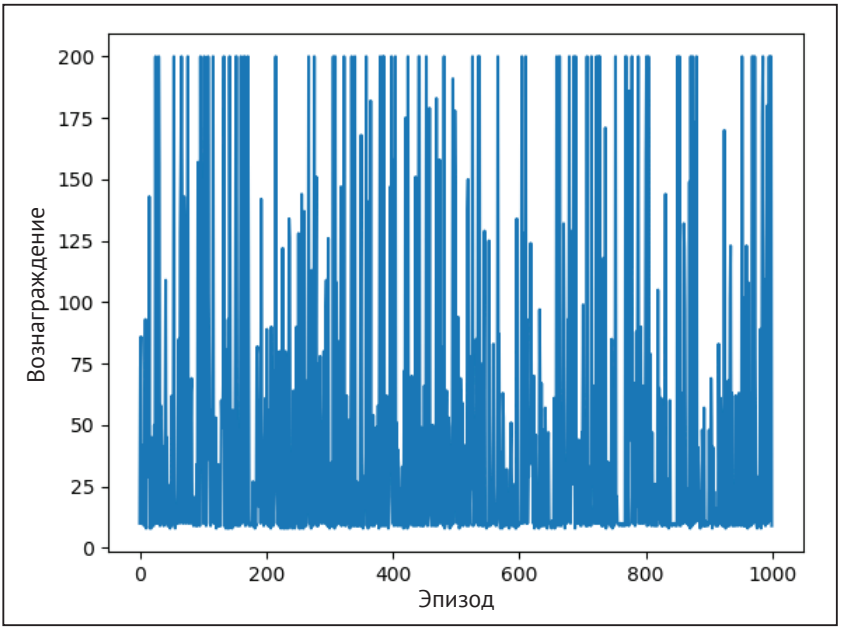
Мы также заметили еще одну интересную вещь: стратегия, обученная методом случайного поиска, оказалась лучше случайного выбора действий. Это потому, что при выборе случайного линейного отображения учитываются наблюдения. Имея больше информации об окружающей среде, мы можем принимать более осмысленные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

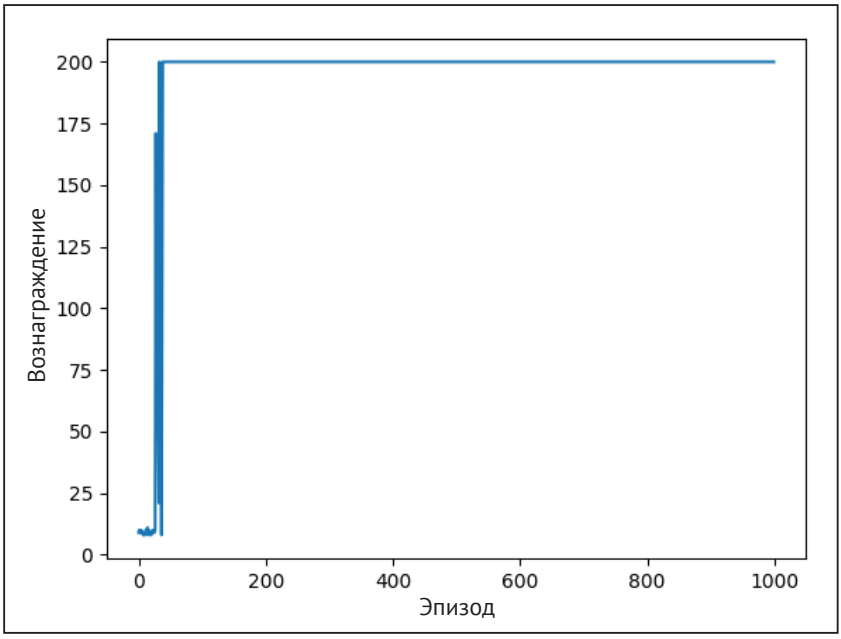
Вознаграждение от эпизода к эпизоду увеличивается. Уже в первых 100 эпизодах оно достигает 200 и остается на этом уровне. Среднее полное вознаграждение тоже выглядит обнадеживающе:

```
>>> 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
```

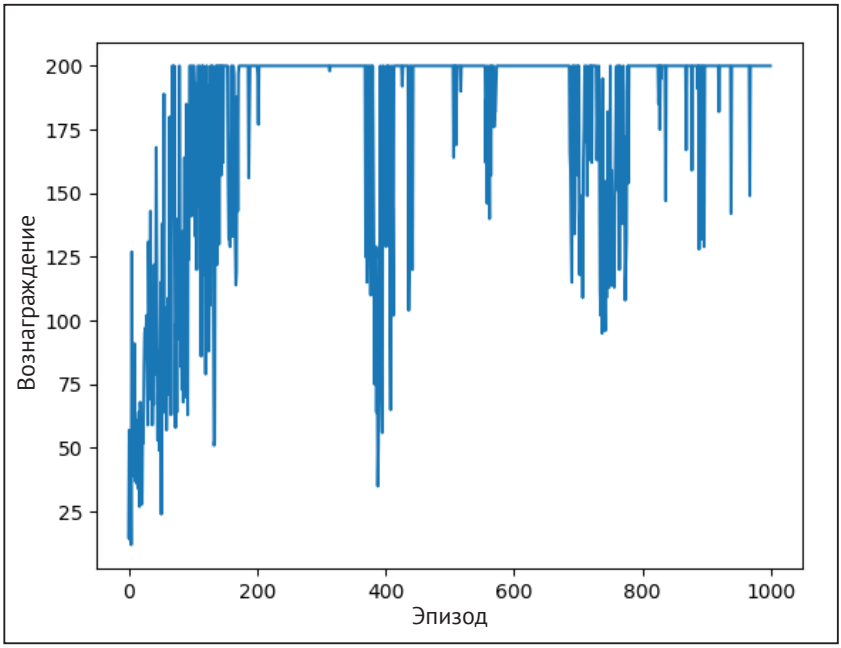
6. Вычислим среднее полное вознаграждение, полученное в алгоритме градиента стратегии:

```
>>> 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.