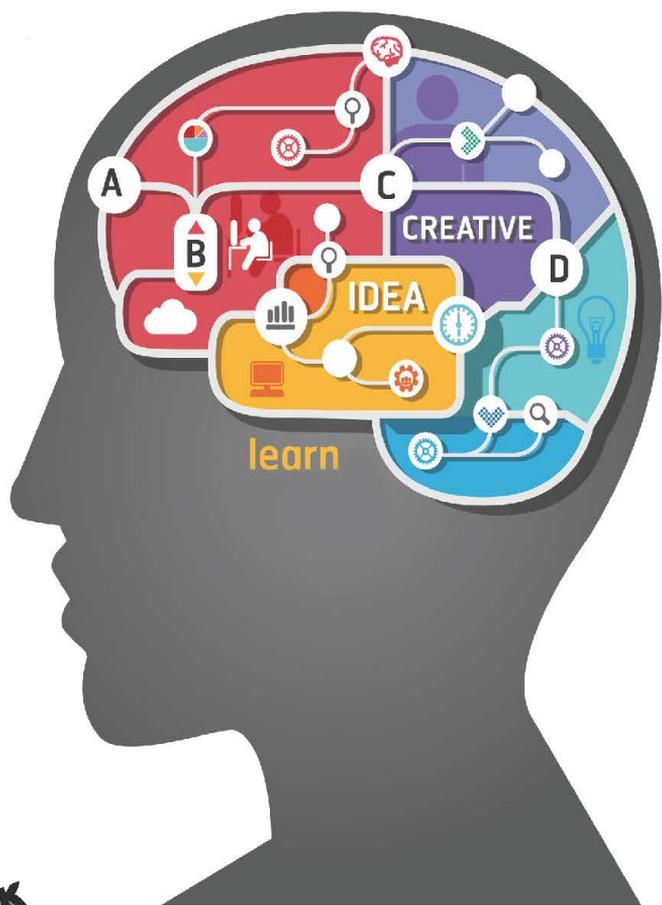


Алгоритмы обучения с подкреплением на Python



Андреа Лонца

Алгоритмы обучения с подкреплением на Python

Андреа Лонца

Алгоритмы обучения с подкреплением на Python

Reinforcement Learning Algorithms with Python

Learn, understand, and develop smart
algorithms for addressing AI challenges

Andrea Lonza



BIRMINGHAM - MUMBAI

Алгоритмы обучения с подкреплением на Python

Описание и разработка алгоритмов
искусственного интеллекта

Андреа Лонца



Москва, 2020

УДК 004.85
ББК 32.971.3
Л76

Лонца А.

Л76 Алгоритмы обучения с подкреплением на Python / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2020. – 286 с.: ил.

ISBN 978-5-97060-855-5

Эта книга поможет читателю овладеть алгоритмами обучения с подкреплением (ОП) и научиться реализовывать их при создании самообучающихся агентов.

В первой части рассматриваются различные элементы ОП, сфера его применения, инструменты, необходимые для работы в среде ОП. Вторая и третья части посвящены непосредственно алгоритмам. В числе прочего автор показывает, как сочетать Q-обучение с нейронными сетями для решения сложных задач, описывает методы градиента стратегии, TRPO и PPO, позволяющие повысить производительность и устойчивость, а также детерминированные алгоритмы DDPG и TD3. Читатель узнает о том, как работает техника подражательного обучения, познакомится с алгоритмами исследования на базе верхней доверительной границы (UCB и UCB1) и метаалгоритмом ESBAS.

Издание предназначено для тех, кто интересуется исследованиями в области искусственного интеллекта, применяет в работе глубокое обучение или хочет освоить обучение с подкреплением с нуля. Обязательное условие – владение языком Python на рабочем уровне.

УДК 004.85
ББК 32.971.3

First published in the English language under the title 'Reinforcement Learning Algorithms with Python. Russian language edition copyright © 2020 by DMK Press. All rights reserved.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN 978-1-78913-111-6 (англ.)
ISBN 978-5-97060-855-5 (рус.)

Copyright © Packt Publishing 2019
© Оформление, издание, перевод,
ДМК Пресс, 2020

*Спасибо, папа и мама, что открыли мне свет,
именуемый жизнью, и всегда были рядом.*

Фред, ты крутой чувак.

Ты всегда понуждаешь меня сделать больше.

Спасибо, брат!

Содержание

Об авторе	12
Предисловие	13
Часть I. АЛГОРИТМЫ И ОКРУЖАЮЩИЕ СРЕДЫ	18
Глава 1. Ландшафт обучения с подкреплением	19
Введение в ОП.....	20
Сравнение ОП и обучения с учителем.....	22
История ОП.....	23
Глубокое обучение.....	25
Элементы ОП.....	26
Стратегия.....	26
Функция ценности.....	28
Вознаграждение.....	29
Модель.....	30
Применение ОП.....	30
Игры.....	30
Робототехника и индустрия 4.0.....	31
Машинное обучение.....	32
Экономика и финансы.....	32
Здравоохранение.....	32
Интеллектуальные транспортные системы.....	33
Оптимизация энергопотребления и умные сети электроснабжения.....	33
Резюме.....	33
Вопросы.....	33
Для дальнейшего чтения.....	34
Глава 2. Реализация цикла ОП и OpenAI Gym	35
Настройка окружающей среды.....	36
Установка OpenAI Gym.....	36
Установка Roboschool.....	37
OpenAI Gym и цикл ОП.....	37
Разработка цикла ОП.....	38
Привыкаем к пространствам.....	41
Разработка моделей МО с помощью TensorFlow.....	42
Тензоры.....	43
Создание графа.....	45
Простой пример линейной регрессии.....	46

Введение в TensorBoard.....	49
Типы окружающих сред ОП.....	51
Зачем нужны различные среды?.....	51
Окружающие среды с открытым исходным кодом.....	52
Резюме.....	54
Вопросы.....	55
Для дальнейшего чтения.....	55

Глава 3. Решение задач методом динамического программирования..... 56

МППР.....	56
Стратегия.....	58
Доход.....	58
Функции ценности.....	59
Уравнение Беллмана.....	60
Классификация алгоритмов ОП.....	61
Безмодельные алгоритмы.....	62
Алгоритмы ОП, основанные на модели.....	63
Разнообразие алгоритмов.....	64
Динамическое программирование.....	64
Оценивание и улучшение стратегии.....	65
Итерация по стратегиям.....	66
Итерация по ценности.....	70
Резюме.....	72
Вопросы.....	73
Для дальнейшего чтения.....	73

Часть II. БЕЗМОДЕЛЬНЫЕ АЛГОРИТМЫ ОП..... 74

Глава 4. Применение Q-обучения и алгоритма SARSA..... 75

Обучение без модели.....	76
Порядок действий.....	76
Оценивание стратегии.....	77
Проблема исследования.....	77
TD-обучение.....	78
TD-обновление.....	79
Улучшение стратегии.....	79
Сравнение методов Монте-Карло и TD-методов.....	79
SARSA.....	80
Алгоритм.....	80
Применение SARSA к игре Taxi-v2.....	81
Q-обучение.....	86
Теория.....	86
Алгоритм.....	87
Применение Q-обучения к игре Taxi-v2.....	87
Сравнение SARSA и Q-обучения.....	89

Резюме.....	91
Вопросы.....	92
Глава 5. Глубокая Q-сеть.....	93
Глубокие нейронные сети и Q-обучение	93
Аппроксимация функций	94
Q-обучение с нейронными сетями	95
Неустойчивость глубокого Q-обучения	96
DQN.....	97
Решение.....	97
Алгоритм DQN	98
Архитектура модели.....	101
Применение DQN к игре Pong	102
Игры Atari	102
Предварительная обработка	103
Реализация DQN	105
Результаты.....	112
Вариации на тему DQN.....	113
Double DQN	114
Dueling DQN	117
<i>n</i> -шаговый DQN	118
Резюме.....	120
Вопросы.....	120
Для дальнейшего чтения.....	121
Глава 6. Стохастическая оптимизация и градиенты	
стратегии.....	122
Методы градиента стратегии.....	122
Градиент стратегии	123
Теорема о градиенте стратегии.....	124
Вычисление градиента.....	125
Стратегия	126
Алгоритм ГС с единой стратегией.....	127
Устройство алгоритма REINFORCE.....	127
Реализация REINFORCE.....	129
Посадка космического корабля с помощью алгоритма REINFORCE	132
REINFORCE с базой	134
Реализация REINFORCE с базой.....	136
Обучение алгоритма исполнитель–критик.....	137
Как критик помогает обучаться исполнителю	137
<i>n</i> -шаговая модель AC.....	138
Реализация AC.....	139
Посадка космического корабля с помощью алгоритма AC	141
Дополнительные улучшения AC и полезные советы	142
Резюме.....	143
Вопросы.....	143
Для дальнейшего чтения.....	143

Глава 7. Реализация TRPO и PPO	144
Roboschool	144
Управление непрерывной системой.....	145
Метод естественного градиента стратегии	148
Интуитивное описание NPG	149
Немного математики	150
Осложнения в методе естественного градиента	152
Оптимизация стратегии в доверительной области	152
Алгоритм TRPO	153
Реализация алгоритма TRPO	156
Применение TRPO	160
Проксимальная оптимизация стратегии.....	163
Краткое описание	163
Алгоритм PPO	163
Реализация PPO	164
Применение PPO	166
Резюме.....	168
Вопросы.....	168
Для дальнейшего чтения.....	169
Глава 8. Применения алгоритмов DDPG и TD3	170
Сочетание оптимизации градиента стратегии с Q-обучением	170
Детерминированный градиент стратегии.....	171
Алгоритм DDPG	174
Реализация DDPG	176
Применение DDPG к среде VipedalWalker-v2.....	180
Алгоритм TD3	182
Проблема смещения оценки в сторону завышения.....	182
Уменьшение дисперсии	184
Применение TD3 к среде VipedalWalker-v2.....	186
Резюме.....	187
Вопросы.....	188
Для дальнейшего чтения.....	188
Часть III. ЗА ПРЕДЕЛАМИ БЕЗМОДЕЛЬНЫХ АЛГОРИТМОВ	189
Глава 9. ОП на основе модели	190
Методы на основе модели.....	190
Общая картина обучения на основе модели	191
Достоинства и недостатки	195
Сочетание безмодельного и основанного на модели обучения	196
Полезная комбинация.....	196
Построение модели из изображений.....	198
Применение алгоритма ME-TRPO к задаче об обратном маятнике	199

Принцип работы ME-TRPO	200
Реализация ME-TRPO	200
Эксперименты в среде RoboSchool.....	204
Резюме.....	206
Вопросы.....	207
Для дальнейшего чтения.....	207
Глава 10. Подражательное обучение и алгоритм DAgger	208
Технические требования.....	208
Установка Flappy Bird	209
Подход на основе подражания	209
Пример: помощник водителя.....	210
Сравнение подражательного обучения и обучения с подкреплением.....	211
Роль эксперта в подражательном обучении	211
Структура IL	212
Игра Flappy Bird	214
Порядок взаимодействия с окружающей средой.....	215
Алгоритм агрегирования набора данных	216
Алгоритм DAgger	217
Реализация DAgger	217
Анализ результатов игры в Flappy Bird	221
Обратное обучение с подкреплением.....	222
Резюме.....	223
Вопросы.....	223
Для дальнейшего чтения.....	224
Глава 11. Оптимизация методом черного ящика	225
За рамками ОП.....	225
Краткий обзор ОП.....	226
Альтернатива	226
Основы эволюционных алгоритмов	227
Генетические алгоритмы	230
Эволюционные стратегии.....	230
Масштабируемые эволюционные стратегии.....	232
Основной принцип.....	233
Масштабируемая реализация.....	234
Применение масштабируемой ЭС к среде LunarLander	239
Резюме.....	241
Вопросы.....	241
Для дальнейшего чтения.....	242
Глава 12. Разработка алгоритма ESBAS	243
Исследование и использование.....	244
Задача о многооруком бандите	245
Подходы к исследованию.....	246
ϵ -жадная стратегия	246

Алгоритм UCB	247
Сложность исследования	248
Алгоритм ESBAS.....	249
Что такое выбор алгоритма	249
ESBAS изнутри	250
Реализация.....	252
Тестирование в среде Acrobot.....	255
Резюме.....	257
Вопросы.....	258
Для дальнейшего чтения.....	258
Глава 13. Практические подходы к решению проблем ОП.....	259
Рекомендуемые практики глубокого ОП.....	259
Выбор подходящего алгоритма	260
От простого к сложному.....	261
Проблемы глубокого ОП.....	263
Устойчивость и воспроизводимость результатов	263
Эффективность	264
Обобщаемость.....	265
Передовые методы	266
ОП без учителя.....	266
Перенос обучения.....	268
ОП в реальном мире	270
Лицом к лицу с реальным миром.....	270
Преодоление разрыва между имитационной моделью и реальным миром	271
Создание собственной окружающей среды.....	272
Будущее ОП и его влияние на общество	272
Резюме.....	273
Вопросы.....	274
Для дальнейшего чтения.....	274
Ответы на вопросы.....	275
Предметный указатель	281

Об авторе

Андреа Лонца занимается глубоким обучением, одержим искусственным интеллектом и страстью создавать машины, действующие «разумно». Знания в области обучения с подкреплением, обработки естественного языка и компьютерного зрения приобрел в ходе работы над проектами по машинному обучению в университете и в промышленности. Также участвовал в нескольких конкурсах Kaggle и достигал высоких результатов. Всегда ищет интересные задачи и любит доказывать, на что способен.

О РЕЦЕНЗЕНТЕ

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.

Предисловие

Обучение с подкреплением (ОП) – популярное и многообещающее направление искусственного интеллекта, в рамках которого изучается построение моделей и агентов, способных находить идеальное поведение в условиях изменяющихся требований. Эта книга поможет вам овладеть алгоритмами ОП и научиться реализовывать их при создании самообучающихся агентов.

Книга начинается с введения в инструменты, библиотеки и процедуру установки, необходимые для работы в среде ОП, затем рассматриваются различные элементы ОП и применение методов, основанных на понятии ценности, в частности алгоритмов Q-обучения и SARSA. Вы научитесь сочетать Q-обучение с нейронными сетями для решения сложных задач. Рассматриваются также методы градиента стратегии, TRPO и PPO, позволяющие повысить производительность и устойчивость, а затем детерминированные алгоритмы DDPG и TD3. Объясняется, как работает техника подражательного обучения и как алгоритм Dagger позволяет обучить агента летать. Вы узнаете об эволюционных стратегиях и оптимизации методом черного ящика. И наконец познакомитесь с алгоритмами исследования на базе верхней доверительной границы (UCB и UCB1) и метаалгоритмом ESBAS.

Прочитав книгу до конца, вы научитесь применять основные алгоритмы ОП для решения реальных задач и станете членом сообщества ОП.

ПРЕДПОЛАГАЕМАЯ АУДИТОРИЯ

Если вы занимаетесь исследованиями в области ИИ, применяете в работе глубокое обучение или просто хотите изучить ОП с нуля, то эта книга для вас. Она будет полезна и тем, кто хочет узнать о последних достижениях в этой области. Необходимо владение языком Python на рабочем уровне.

СТРУКТУРА КНИГИ

В главе 1 «Ландшафт обучения с подкреплением» описываются проблемы, которые ОП с успехом решает, и приложения, в которых алгоритмы ОП уже нашли применение. Здесь же рассматриваются инструменты, библиотеки и процедуры их установки и настройки, необходимые для выполнения обсуждаемых в книге проектов.

В главе 2 «Реализация цикла ОП и OpenAI Gym» описывается главный цикл алгоритмов ОП, инструментарий для разработки алгоритмов и различные типы сред. Мы разработаем с помощью интерфейса к OpenAI Gym агента, который будет играть в игру CartPole, совершая случайные действия. Мы также научимся использовать OpenAI Gym для работы в других средах.

Глава 3 «Решение задач методом динамического программирования» содержит введение в основные идеи, терминологию и подходы, применяемые в ОП.

Вы узнаете о главных составных частях ОП и составите общее представление о том, как алгоритмы ОП применяются к решению задач. Вы также узнаете о различиях между основанными на модели и безмодельными алгоритмами и о классификации алгоритмов обучения с подкреплением. Мы применим динамическое программирование для решения игры FrozenLake.

В главе 4 «Применения Q-обучения и алгоритма SARSA» речь пойдет о методах на основе ценности, в частности Q-обучении и SARSA, двух алгоритмах, которые отличаются от динамического программирования и хорошо масштабируются на задачи большого размера. Чтобы лучше разобраться в этих алгоритмах, мы применим их к игре FrozenLake и сравним результаты с динамическим программированием.

В главе 5 «Глубокие Q-сети» описывается использование нейронных сетей и, в частности, **сверточных нейронных сетей (СНС)** к Q-обучению. Вы узнаете, почему сочетание Q-обучения и нейронных сетей дает поразительные результаты и как это открывает путь к гораздо более широкому кругу задач. Кроме того, мы применим глубокую Q-сеть к игре Atari, воспользовавшись интерфейсом к OpenAI Gym.

В главе 6 «Стохастическая оптимизация и градиенты стратегии» вводится новое семейство безмодельных алгоритмов: методы градиента стратегии. Мы узнаем о различии между методами градиента стратегии и методами на основе ценности, а также об их сильных и слабых сторонах. Затем реализуем алгоритмы REINFORCE и исполнитель–критик для решения игры LunarLander.

В главе 7 «Реализация TRPO и PPO» предлагается модификация методов градиента стратегии с использованием новых механизмов контроля над улучшением стратегии. Эти механизмы позволяют улучшить устойчивость и сходимость алгоритмов градиента стратегии. В частности, мы опишем и реализуем два основных метода градиента стратегии, в которых используются эти подходы: TRPO и PPO. Они будут реализованы в семействе сред с непрерывным пространством действий RoboSchool.

В главе 8 «Применения алгоритмов DDPG и TD3» вводится новая категория алгоритмов с детерминированной стратегией, которые сочетают идею градиента стратегии с Q-обучением. Вы узнаете об идее и реализации двух глубоких детерминированных алгоритмов, DDPG и TD3, в новой окружающей среде.

В главе 9 «ОП на основе модели» иллюстрируются алгоритмы ОП, которые обучаются модели окружающей среды для планирования будущих действий, т. е. обучения стратегии. Вы узнаете, как они работают, в чем их плюсы и почему они предпочтительны во многих ситуациях. Чтобы лучше разобраться в них, мы реализуем основанный на модели алгоритм в среде RoboSchool.

В главе 10 «Подражательное обучение и алгоритм DAgger» объясняется, как работает подражательное обучение и как его можно адаптировать к конкретной задаче. Будет рассмотрен самый известный алгоритм подражательного обучения, DAgger. Ради лучшего освоения мы реализуем его и тем ускорим процесс обучения агента в игре FlappyBird.

В главе 11 «Оптимизация методом черного ящика» изучаются эволюционные алгоритмы – класс алгоритмов оптимизации методом черного ящика, которые не опираются на обратное распространение. Интерес к этим алгоритмам растет, потому что они, во-первых, быстро обучаются, а во-вторых, легко

распараллеливаются на сотни и тысячи процессорных ядер. В этой главе подводится теоретический фундамент под эти алгоритмы и приводится практическая реализация на примере алгоритма эволюционной стратегии.

В главе 12 «Разработка алгоритма ESBAS» описывается важная дилемма исследования–использования, специфичная для ОП. Она демонстрируется на примере задачи о многоруких бандитах и решается методами верхней доверительной границы: UCB и UCB1. Затем мы узнаем о проблеме выбора алгоритма и разработаем метаалгоритм ESBAS, в котором UCB1 используется для выбора наиболее подходящего алгоритма ОП в конкретной ситуации.

В главе 13 «Практические подходы к решению проблем ОП» мы рассмотрим основные проблемы, возникающие в этой области, и объясним, как их преодолевать. Вы узнаете о некоторых трудностях применения ОП к реальным задачам, о будущем глубокого ОП и о его влиянии на общество.

Что необходимо для чтения этой книги

Необходимо владение языком Python на рабочем уровне. Знакомство с ОП и различными инструментами, используемыми в этой области, также было бы полезно.

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, адреса в Twitter, например: «В этой книге используется версия Python 3.7, но должны работать все версии начиная с 3.5. Предполагается также, что пакеты `numpy` и `matplotlib` уже установлены».

Отдельно стоящие фрагменты кода набраны так:

```
import gym

# создать окружающую среду
env = gym.make("CartPole-v1")
# привести среду в исходное состояние перед началом работы
env.reset()

# повторить 10 раз
for i in range(10):
    # предпринять случайное действие
    env.step(env.action_space.sample())
    # отрисовать игру
    env.render()

# закрыть среду
env.close()
```

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

```
$ git clone https://github.com/pybox2d/pybox2d
$ cd pybox2d
$ pip install -e .
```

Новые термины, важные слова и слова на экране набраны **полужирным шрифтом**. Также выделяются элементы интерфейса, например пункты меню и поля в диалоговых окнах, например: «В **обучении с подкреплением (ОП)** алгоритм называется агентом, он обучается на данных, поступающих от окружающей среды».

 Предупреждения и важные замечания оформлены так.

 Советы и рекомендации выглядят так.

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.

Часть I

АЛГОРИТМЫ И ОКРУЖАЮЩИЕ СРЕДЫ

Эта часть представляет собой введение в обучение с подкреплением. В ней закладывается теоретический фундамент и подготавливается почва для последующих глав. Эта часть состоит из следующих глав:

- глава 1 «Ландшафт обучения с подкреплением»;
- глава 2 «Реализация цикла ОП и OpenAI Gym»;
- глава 3 «Решение задач методами динамического программирования».

Глава 1

Ландшафт обучения с подкреплением

Люди и животные обучаются методом проб и ошибок. Этот процесс основан на механизмах вознаграждения в ответ на то или иное поведение. Его цель – посредством многократного повторения побудить к выбору таких действий, которые порождают положительные отклики, и отвлечь от действий, порождающих отрицательные отклики. Посредством механизма проб и ошибок мы обучаемся взаимодействовать с людьми и окружающим нас миром, преследовать сложные осмысленные цели, а не просто стремиться к получению немедленного удовольствия.

Обучение на опыте и взаимодействии принципиально важно. Представьте, что вы должны научиться играть в футбол, только глядя на то, как играют другие. Если, основываясь на таком опыте, вы выйдете на поле, то, скорее всего, будете играть отвратительно.

Это было продемонстрировано в середине XX века в известной работе Ричарда Хелда (Richard Held) и Алана Хейна (Alan Hein) 1963 года, в которой два котенка с рождения были посажены в некое подобие карусели. Один котенок мог двигаться свободно (был активен), а второй не мог производить никаких движений и перемещался пассивно – его возил первый. Впоследствии только у котенка, который мог свободно двигаться, развилось восприятие глубины и двигательные навыки. Это было доказано отсутствием у пассивного котенка мигательного рефлекса на приближающиеся объекты. Таким образом, в этом довольно жестоком эксперименте было продемонстрировано, что для обучения животного необходимо физическое взаимодействие с окружающей средой, а не только зрительная стимуляция.

В основе **обучения с подкреплением (ОП)** лежит идея об активном взаимодействии со средой методом проб и ошибок, рассмотренная у обучающихся людей и животных. В частности, в ОП агент постепенно обучается в процессе взаимодействия с окружающим миром. Это позволяет наделять компьютер рудиментарными навыками обучения, взяв за образец поведение человека.

Эта книга целиком посвящена обучению с подкреплением. Ее цель – помочь вам разобраться в данной области на практических примерах. В начальных главах мы познакомимся с базовыми понятиями обучения с подкреплением, а затем приступим к разработке первых алгоритмов. Постепенно мы будем создавать все более сложные и мощные алгоритмы для решения более интересных

и интригующих задач. Вы увидите, что обучение с подкреплением – очень широкий предмет и что в нем существует много алгоритмов, предназначенных для решения разных задач разными способами. Однако мы постараемся дать простое, но полное описание всех идей, а также ясные и практически полезные реализации алгоритмов.

В этой главе мы познакомимся с фундаментальными концепциями ОП, узнаем о различиях между имеющимися подходами и о таких ключевых понятиях, как функция ценности, вознаграждение и модель окружающей среды. Попутно мы расскажем об истории ОП и ее приложениях.

В этой главе рассматриваются следующие вопросы:

- введение в ОП;
- элементы ОП;
- приложения ОП.

ВВЕДЕНИЕ В ОП

ОП – часть машинного обучения, в которой изучается последовательное принятие решений для достижения поставленной цели. Задача ОП состоит из **агента**, принимающего решения, и физического или виртуального мира, с которым агент взаимодействует, – **окружающей среды**. Взаимодействие агента с окружающей средой сводится к **действиям**, имеющим некоторые последствия. В результате агент получает от среды обратную связь в форме нового **состояния** и **вознаграждения**. Оба этих сигнала – последствия действия, предпринятого агентом. Точнее, вознаграждение показывает, насколько хорошим или плохим было действие, а состояние – это текущее представление агента и окружающей среды. Этот цикл показан на рис. 1.1.

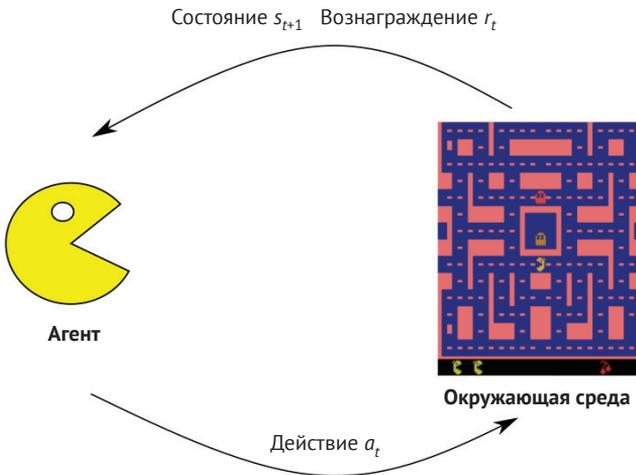


Рис. 1.1

На этом рисунке агент представлен значком пакмана¹, который в зависимости от текущего состояния окружающей среды выбирает следующее действие. Его поведение влияет на окружающую среду, а именно на его положение и положения врагов. В ответ на поведение агента среда возвращает новое состояние и вознаграждение. Этот цикл повторяется, пока игра не завершится.

Конечная цель агента – максимизировать полное вознаграждение, полученное за все время существования. Введем обозначения: если a_t – действие в момент t , а r_t – вознаграждение в момент t , то агент предпринимает действия a_0, a_1, \dots, a_t , стремясь максимизировать сумму $\sum_{i=0}^t r_i$.

Чтобы максимизировать полное вознаграждение, агент должен обучиться наилучшему поведению в каждой ситуации. Для этого агенту необходимо произвести оптимизацию на длинном горизонте, принимая во внимание каждое действие. В окружающей среде с большим количеством дискретных или непрерывных состояний и действий обучение затруднено тем, что агент вынужден учитывать все возможные ситуации. Мало того, вознаграждение может поступать очень редко и запаздывать во времени, что дополнительно усложняет процесс обучения.

Чтобы привести пример задачи ОП и заодно объяснить, в чем сложность разреженного вознаграждения, рассмотрим хорошо известную сказку о Гензеле и Гретель. Родители завели их в лес, чтобы там оставить, но Гензель, знавший об их намерении, захватил с собой из дому ломоть хлеба и оставил дорожку из хлебных крошек, которая должна была привести его и сестру обратно домой. В системе ОП агентами являются Гензель и Гретель, а окружающей средой – лес. За каждую встреченную на обратном пути крошку агент получает вознаграждение +1, а за выход к дому – вознаграждение +10. В этом случае чем чаще хлебная дорожка, тем легче детям будет найти дорогу домой, поскольку для перехода от одной крошки к другой нужно исследовать сравнительно небольшую область. К сожалению, в реальном мире разреженные вознаграждения встречаются куда чаще плотных.

Важная характеристика ОП заключается в возможности достижения результата, даже когда окружающая среда динамическая, неопределенная и недетерминированная. Ниже приведены примеры реальных проблем, которые можно поставить как задачи ОП.

- Беспилотные автомобили – популярная идея, которая с трудом поддается ОП. Дело в том, что при движении по дороге нужно принимать во внимание много факторов (пешеходы, другие автомобили, велосипеды, светофоры и т. д.), а среда в высшей степени неопределенная. В данном случае беспилотный автомобиль – это агент, который может воздействовать на руль, акселератор и тормоза. Средой же является окружающий мир. Очевидно, что агент не может воспринять весь мир вокруг себя целиком, ему доступна лишь ограниченная информация от датчиков (например, камеры, радара и системы навигации). Цель беспилотного автомобиля – добраться до места назначения за минимальное время, соблюдая правила дорожного движения и не нанеся никакого вреда. Сле-

¹ Персонаж старой компьютерной игры PacMan. – Прим. перев.

довательно, агент может получить отрицательное вознаграждение, если случится какое-то неприемлемое событие, а величина положительного пропорциональна времени поездки.

- В шахматах цель состоит в том, чтобы поставить мат противнику. В терминах ОП игрок является агентом, а текущая позиция на доске – окружающей средой. Агенту разрешается переставлять фигуры согласно правилам. В итоге окружающая среда возвращает положительное вознаграждение, если агент выиграл, и отрицательное, если проиграл. Во всех остальных случаях вознаграждение нулевое, а следующим состоянием является позиция на доске после хода противника. В отличие от примера беспилотного автомобиля, здесь состояние агента совпадает с состоянием окружающей среды. Иными словами, агент знает все об окружающей среде.

Сравнение ОП и обучения с учителем

ОП и обучение с учителем – похожие, но все же различные парадигмы обучения на данных. Многие задачи можно решить как с помощью ОП, так и обучения с учителем, но в большинстве случаев эти методы предназначены для решения разных задач.

Цель обучения с учителем – научиться обобщать, располагая лишь фиксированным набором данных с ограниченным количеством примеров. Каждый пример состоит из входа и желательного выхода (или метки), так что реакция на выбор агента следует незамедлительно.

Напротив, в ОП акцент ставится на последовательных действиях, которые можно предпринять в конкретной ситуации. В данном случае единственное, что дает учитель, – сигнал вознаграждения. Какое действие правильно при данных условиях, неизвестно, в отличие от обучения с учителем.

ОП можно рассматривать как более общую и полную систему обучения. Перечислим основные характеристики ОП:

- вознаграждение может быть плотным, разреженным или приходиться с большим запаздыванием. Во многих случаях вознаграждение становится известно только в конце задания (например, в шахматах);
- задача последовательная и зависящая от времени – выбранные действия могут влиять на следующие действия, которые, в свою очередь, влияют на возможные вознаграждения и состояния;
- агент должен совершать действия, которые с высокой вероятностью приведут к достижению цели (использование), но в то же время должен пробовать иные действия, чтобы другие части окружающей среды не остались неисследованными (исследование). Эту двойственность называют дилеммой (или компромиссом) исследования–использования, она призвана решить трудную проблему поиска баланса между исследованием и использованием окружающей среды. Она важна также и потому, что, в отличие от обучения с учителем, агент ОП может влиять на окружающую среду, т. к. вправе собирать новые данные, коль скоро считает это полезным;
- окружающая среда стохастическая и недетерминированная, и агент должен учитывать это в процессе обучения и для предсказания следующего

действия. Мы увидим, что многие компоненты ОП можно спроектировать так, что они будут выдавать либо только одно детерминированное значение, либо диапазон значений, каждому из которых сопоставлена вероятность.

Третий вид обучения – **обучение без учителя** – применяется для выявления закономерностей в данных при отсутствии какой-либо информации от учителя. Сжатие данных, кластеризация и порождающие модели – все это примеры обучения без учителя. Его можно инкорпорировать в систему ОП, чтобы исследовать окружающую среду и получать информацию о ней. Комбинация обучения без учителя и ОП называется **ОП без учителя**. В этом случае никакое вознаграждение не присуждается, а агент может генерировать внутренние мотивы, побуждающие отдать предпочтение новым ситуациям, которые можно исследовать.



Отметим, что задачи, связанные с беспилотными автомобилями, можно решать и как задачи обучения с учителем, но результаты получаются неудовлетворительные. Основная проблема в том, что распределение данных, с которым агент сталкивается на практике, может сильно отличаться от того, что он видел в процессе обучения.

История ОП

Первое математическое обоснование ОП появилось в 1960–1970-е годы в области оптимального управления. Тем самым была решена задача минимизации некоторой меры поведения динамической системы, изменяющейся во времени. Этот метод подразумевал решение системы уравнений при известной динамике системы. Тогда было сформулировано ключевое понятие **марковского процесса принятия решений (МППР)**. Оно легло в основу общего подхода к моделированию принятия решений в стохастических условиях. В эти годы был разработан метод решения задач оптимального управления, получивший название **динамического программирования (ДП)**. Смысл ДП в том, чтобы разбить сложную задачу на несколько более простых с целью решить систему уравнений МППР.

Отметим, что ДП упрощает поиск оптимального управления лишь для систем с известной динамикой, ни о каком обучении речи не идет. Кроме того, ему свойственна проблема **проклятия размерности**, поскольку требования к вычислительным ресурсам экспоненциально возрастают по мере роста количества состояний.

Но, как заметили Ричард Саттон и Эндрю Бартон, хотя эти методы не подразумевают обучения, мы все равно должны считать методы решения задач оптимального управления, в т. ч. ДП, методами ОП.

В 1980-х годах наконец-то появилась концепция обучения на основе предсказаний в соседние моменты времени – метод **обучения на основе временных различий (temporal difference learning – TD-обучение)**. Открытие TD-обучения принесло с собой новое семейство эффективных алгоритмов, которые будут рассмотрены в этой книге.

Первые задачи, решенные с помощью TD-обучения, были настолько малы, что их можно было представить с помощью таблиц или массивов. Соответ-

ствующие методы получили название **табличных методов**, они часто находят оптимальное решение, но не масштабируемы. Вообще, во многих задачах ОП пространство состояний огромно, поэтому табличные методы к ним неприменимы. В таких случаях используется аппроксимация функций, позволяющая найти хорошее приближенное решение с меньшими вычислительными затратами.

Включение в ОП аппроксимации функций и, в частности, искусственных нейронных сетей (в т. ч. глубоких) – дело нетривиальное, но, как показано на целом ряде примеров, иногда это позволяет достичь поразительных результатов. Сочетание глубокого обучения с ОП называют **глубоким обучением с подкреплением (глубоким ОП)**, оно обрело широкую популярность, после того как алгоритм **глубокой Q-сети (DQN)** в 2015 году продемонстрировал умение играть в игры компании Atari на уровне, превышающем возможности человека (для обучения использовались только изображения на экране). Еще одно впечатляющее достижение глубокого ОП – программа AlphaGo, которая в 2017 году стала первой программой, обыгравшей в го 18-кратного чемпиона мира Ли Седоля. Эти прорывные достижения не только показали, что программа может работать лучше человека в многомерных пространствах (воспринимая те же самые изображения, что и человек), но и что она может вести себя различными интересными способами. Примером может служить неожиданное решение, найденное системой глубокого ОП в аркадной игре Atari Breakout, в которой игрок должен уничтожить все кирпичи, как показано на рис. 1.2. Агент обнаружил, что, пробив туннель слева от кирпичей и направляя шарик в эту сторону, он сможет уничтожить гораздо больше кирпичей и тем самым увеличить свой счет всего одним ходом.



Рис. 1.2

Есть еще много интересных примеров того, как агенты демонстрировали изобретательное поведение или стратегию, неизвестную людям, в частности ход, сделанный AlphaGo в игре против Ли Седоля. С точки зрения человека, он выглядел бессмысленно, но в итоге позволил AlphaGo победить (он даже получил название – **ход 37**).

В наши дни при работе с многомерным пространством состояний или действий применение глубоких нейронных сетей для аппроксимации функций

кажется чуть ли не очевидным выбором. Глубокое ОП применялось и к более трудным задачам, например: оптимизация энергопотребления в центрах обработки данных, беспилотные автомобили, оптимизация многопериодического портфеля ценных бумаг, робототехника и многое другое.

Глубокое обучение

Теперь можно задаться вопросом: почему глубокое обучение в сочетании с ОП дает такие замечательные результаты? Главным образом потому, что глубокое обучение способно справляться с пространством состояний очень высокой размерности. До изобретения глубокого ОП пространства состояний приходилось разбивать на более простые представления, называемые **признаками**. Их было трудно проектировать, и иногда эта задача была подвластна только узким специалистам. Теперь же, пользуясь глубокими нейронными сетями, в частности **сверточными нейронными сетями (СНС)** или **рекуррентными нейронными сетями (РНС)**, ОП-система может обучиться различным уровням абстракции непосредственно на исходных пикселях или последовательных данных (например, текстах на естественном языке). Такая конфигурация показана на рис. 1.3.

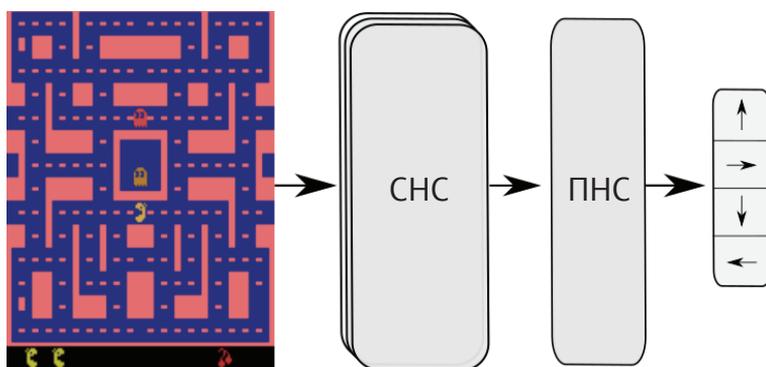


Рис. 1.3

Кроме того, задачи глубокого ОП теперь можно решать от начала до конца в едином процессе. Раньше алгоритм ОП состоял из двух разных конвейеров: один для восприятия, второй для принятия решений. Теперь же эти процессы объединены и обучаются вместе: от исходных пикселей к выбору действия. Например, пакмана на рис. 1.3 можно обучить с помощью СНС, которая обрабатывает визуальный компонент, и **полносвязной нейронной сети (ПНС)**, которая преобразует выход СНС в действие.

В настоящее время глубокое ОП – чрезвычайно актуальная тема. И главная причина заключается в том, что глубокое ОП считается технологией, которая позволит создавать машины с высоким уровнем интеллекта. Доказательством служит тот факт, что две широко известные компании в области ИИ, DeepMind и OpenAI, ведут активные исследования в области ОП.

Но, несмотря на несомненные достижения глубокого ОП, многое еще предстоит сделать. Некоторые из открытых вопросов перечислены ниже:

- система глубокого ОП обучается слишком медленно по сравнению с человеком;
- перенос обучения в ОП остается нерешенной проблемой;
- функцию вознаграждения трудно придумать и определить;
- агенты ОП с трудом обучаются в таких сложных и динамических средах, как реальный мир.

Тем не менее фронт исследований в этой области быстро расширяется, и компании начинают включать ОП в свои продукты.

ЭЛЕМЕНТЫ ОП

Как мы знаем, агент взаимодействует с окружающей средой посредством действий. Это заставляет среду изменяться и начислять агенту вознаграждение, пропорциональное качеству действий. Методом проб и ошибок агент постепенно обучается находить наилучшее действие в каждой ситуации, стремясь в итоге получить как можно большее полное вознаграждение. В системе ОП выбор действия в конкретном состоянии производится с помощью **стратегии**, а полное вознаграждение, достижимое при старте из некоторого состояния, называется **функцией ценности**. Короче говоря, если агент хочет вести себя оптимально, то в каждой ситуации стратегия должна выбирать такое действие, которое переводит агента в следующее состояние с наибольшей ценностью. Теперь рассмотрим эти основополагающие понятия более пристально.

Стратегия

Стратегия определяет, как агент выбирает действие в данном состоянии. Выбирается такое действие, которое максимизирует полное вознаграждение, достижимое из данного состояния, а не действие, приносящее наибольшее немедленное вознаграждение. Преследуется долгосрочная цель агента. Например, если машине нужно доехать до места назначения 30 км, но хода осталось только на 10 км, а до ближайших заправок 1 км и 60 км, то стратегия выберет заправку на первой АТС (в одном километре), чтобы не остаться без топлива посередине дороги. Это решение не оптимально в ближайшей перспективе, поскольку на заправку уйдет время, но оно необходимо для достижения конечной цели.

На рис. 1.4 показан простой пример – агент, перемещающийся по сетке 4×4, должен добраться до звезды, избегая спиралей. Действия, рекомендуемые стратегией, обозначены стрелкой в направлении хода. Слева на рис. 1.4 показана случайная начальная стратегия, а справа – окончательная оптимальная стратегия. Если имеются два одинаково хороших действия, то агент выбирает любое из них произвольно.

Существует важное различие между стохастическими и детерминированными стратегиями. Детерминированная стратегия однозначно подсказывает, какое действие предпринять, а стохастическая сообщает вероятности каждого

действия. Идея вероятности действия полезна, потому что учитывает динамичность окружающей среды и помогает исследовать ее.

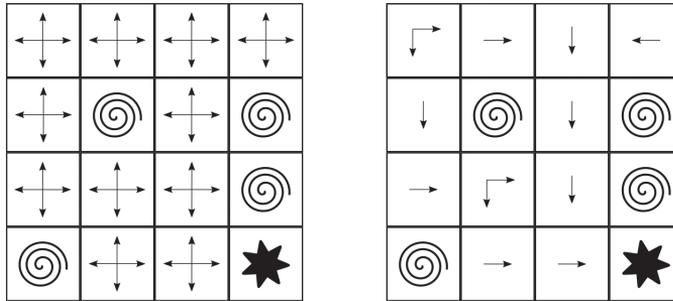


Рис. 1.4

Одна из классификаций алгоритмов ОП основана на том, как стратегии улучшаются в процессе обучения. В более простом случае стратегия, которая воздействует на окружающую среду, похожа на ту, что улучшается во время обучения. По-другому можно сказать, что стратегия обучается на тех же данных, которые генерирует. Такие алгоритмы называются алгоритмами с **единой стратегией** (on-policy algorithm). Напротив, в алгоритмах с **разделенной стратегией** (off-policy algorithm) присутствуют две стратегии: одна воздействует на среду, а другая обучается, но фактически не используется. Первая называется **поведенческой стратегией**, вторая – **целевой стратегией**. Цель поведенческой стратегии – взаимодействовать со средой и собирать о ней информацию, чтобы улучшить **пассивную** целевую стратегию. Как мы увидим в последующих главах, алгоритмы с разделенной стратегией менее устойчивы и их труднее проектировать, чем алгоритмы с единой стратегией, зато у них более высокая выборочная эффективность, т. е. для обучения нужно меньше данных.

Чтобы лучше уяснить себе обе концепции, представьте человека, который хочет обучиться новому навыку. Если он ведет себя как алгоритм с единой стратегией, то всякий раз, испробовав новую последовательность действий, будет изменять свои представления и поведение в соответствии с полученным полным вознаграждением. Наоборот, если человек ведет себя как алгоритм с разделенной стратегией, то может обучиться (целевая стратегия), глядя на старую видеозапись собственных действий (поведенческая стратегия) во время применения того же навыка, т. е. может использовать прежний опыт для самосовершенствования.

Методами градиента стратегии называется семейство алгоритмов ОП, которые обучаются параметрической стратегии (как глубокая нейронная сеть) непосредственно на данных о градиенте качества по стратегии. У таких алгоритмов много достоинств, в т. ч. способность работать с непрерывными действиями и исследовать окружающую среду с разными уровнями детализации. Мы будем подробно рассматривать их в главах 6 «Стохастическая оптимизация и градиенты стратегии», 7 «Реализация TRPO и PPO» и 8 «Применения алгоритмов DDPG и TD3».

Функция ценности

Функция ценности представляет качество состояния в долгосрочной перспективе. Это полное вознаграждение, ожидаемое в будущем, если агент стартует из данного состояния. Если вознаграждение измеряет немедленное качество действия, то функция ценности – его качество на протяженном отрезке времени. Из того, что вознаграждение велико, еще не следует, что будет велика ценность, и наоборот – если вознаграждение мало, это еще не значит, что мала ценность состояния.

Функция ценности может зависеть только от состояния или от пары состояние–действие. В первом случае она называется **функцией ценности состояний**, а во втором – **функцией ценности действий**.

На рис. 1.5 показаны окончательные ценности состояний (слева) и соответствующая оптимальная стратегия (справа).

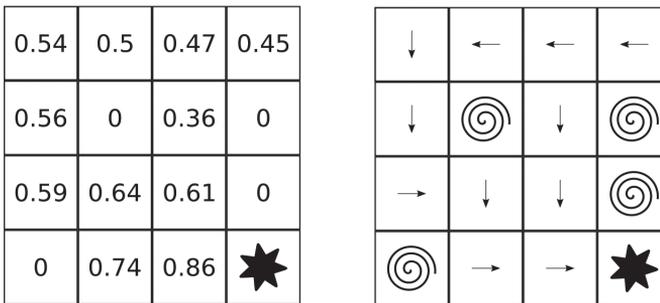


Рис. 1.5

На том же примере сетки, который использовался для иллюстрации понятия стратегии, мы можем показать, что такое функция ценности состояний. Прежде всего можно предположить, что в любой ситуации назначается вознаграждение 0, кроме случая, когда агент доходит до клетки со звездой, где получает вознаграждение +1. Далее предположим, что сильный ветер сдувает агента в другом направлении с вероятностью 0.33. Тогда ценности состояний будут такими, как в левой части рис. 1.5. Оптимальная стратегия будет выбирать действия, которые переводят агента в следующее состояние с максимальной ценностью, как показано в правой части рис. 1.5.

Методы ценности действий (или методы, основанные на функции ценности) – еще одно большое семейство алгоритмов ОП. Они сводятся к обучению функции ценности действий и использованию ее для выбора действий. Начиная с главы 3 мы многое расскажем об этих алгоритмах. Отметим, что из-за стремления взять лучшее из обоих миров в некоторых методах градиента стратегии для обучения хорошей стратегии используется функция ценности. Такие методы называются **методами исполнитель–критик**. На рис. 1.6 показаны все три основных семейства алгоритмов ОП.



Рис. 1.6

Вознаграждение

На каждом временном шаге, т. е. после каждого хода агента, окружающая среда посылает ему число, показывающее, насколько хорошим было действие. Это число называется **вознаграждением**. Как мы уже упоминали, конечная цель агента – максимизировать полное вознаграждение, полученное за время взаимодействия со средой.

В литературе считается, что вознаграждение – часть окружающей среды, но в реальности это не совсем так. Вознаграждение может порождать и сам агент, но только не та его часть, которая принимает решения. Именно по этой причине, а заодно чтобы упростить формулировки, вознаграждение всегда исходит от окружающей среды.

Вознаграждение – единственный сигнал учителя, попадающий в цикл ОП, и важно проектировать вознаграждение правильно, чтобы получить агента с хорошим поведением. Если вознаграждение в чем-то дефектно, то агент может обнаружить эти дефекты и выработать неправильное поведение. Например, *Coast Runners* – игра, цель которой – первым прийти к финишу гонок на водных катерах. В процессе гонки катера получают вознаграждение за поражение мишеней. В компании OpenAI обучили играть агента ОП. Обнаружилось, что вместо стремления как можно быстрее прийти к финишу обучаемый катер мчался по кругу, тараня возрождающиеся мишени и не обращая внимания на периодические поломки и пожары на борту. То есть катер нашел способ максимизировать полное вознаграждение, действуя не так, как ожидалось. Такое поведение стало результатом неверного баланса между краткосрочным и долгосрочным вознаграждениями.

Вознаграждение может выдаваться с разной частотой, зависящей от окружающей среды. Частое вознаграждение называют **плотным**; если же оно выдается лишь несколько раз за игру или только в самом конце, то говорят о **разреженном вознаграждении**. В таком случае агенту бывает очень трудно получить вознаграждение и найти оптимальные действия.

Подражательное обучение и **обратное ОП** – две эффективные техники обучения в условиях отсутствия вознаграждения от окружающей среды. В подражательном обучении для отображения состояний на действия исполь-

зуется демонстрация работы эксперта. А в обратном ОП функция вознаграждения выводится из оптимального поведения эксперта. То и другое – предмет главы 10.

Модель

Модель – необязательный компонент агента, т. е. она не требуется для нахождения стратегии взаимодействия со средой. Модель детально описывает поведение окружающей среды, а именно предсказывает следующее состояние и вознаграждение для данных состояния и действия. Если модель известна, то для взаимодействия с ней и рекомендации будущих действий можно использовать алгоритмы планирования. Например, в средах с дискретными действиями потенциальные траектории можно смоделировать, применяя поиск с заглядыванием вперед (скажем, поиск по дереву методом Монте-Карло).

Модель окружающей среды может быть либо задана заранее, либо обучена посредством взаимодействия с ней. Если среда сложная, то имеет смысл аппроксимировать ее глубокой нейронной сетью. Алгоритмы ОП, в которых используется уже известная или обученная модель среды, называются **методами, основанными на модели**. Они являются противоположностью безмодельным методам и подробно рассматриваются в главе 9.

ПРИМЕНЕНИЕ ОП

ОП применяется в самых разных областях, например: робототехника, финансы, здравоохранение, интеллектуальные транспортные системы. В общем случае применения можно объединить в три большие группы: автоматические машины (автономные транспортные средства, «умные» сети электроснабжения, робототехника и т. п.), оптимизация процессов (плановое техническое обслуживание, цепочки поставок, планирование процессов) и контроль (например, обнаружение отказов и контроль качества).

Поначалу ОП применялось только к простым задачам, но глубокое ОП открыло дорогу к проблемам совсем другого уровня сложности. В настоящее время глубокое ОП демонстрирует весьма многообещающие результаты. К сожалению, многие из этих прорывов ограничиваются исследовательскими приложениями или играми, и зачастую не так-то просто перекинуть мост между чисто исследовательскими приложениями и промышленными задачами. Но, несмотря на это, все больше компаний движется в направлении включения ОП в свои отрасли промышленности и продукты.

Рассмотрим основные области, в которых ОП уже внедряется или видна потенциальная выгода от него.

Игры

Игры – идеальный испытательный стенд для ОП, потому что создаются специально для того, чтобы бросить вызов человеческим возможностям. Чтобы пройти игру до конца, необходим человеческий мозг (память, способность рас-

суждать и координация движений). Поэтому компьютер, способный играть на уровне человека или лучше, обязан обладать теми же качествами. Кроме того, игру зачастую легко воспроизвести, поэтому ее нетрудно смоделировать на компьютере. Видеоигры оказались очень трудны для компьютера из-за частичной наблюдаемости (т. е. в каждый момент времени видна только часть игры) и гигантского пространства поиска (компьютер не может смоделировать все возможные конфигурации).

Прорыв в играх произошел, когда в 2015 году программа AlphaGo обыграла Ли Седоля в древнюю игру го. Это случилось вопреки всем предсказаниям. В то время считалось, что компьютер не сможет обыграть профессионального игрока в го еще по крайней мере 10 лет. В AlphaGo использовалось ОП в сочетании с обучением с учителем на играх, сыгранных профессионалами-людьми. Спустя несколько лет после матча следующая версия программы, AlphaGo Zero, разгромила AlphaGo со счетом 100:0. AlphaGo Zero научилась играть в го всего за три дня игр с самой собой.



Игра с собой – весьма эффективный способ обучения алгоритма, потому что не нужен никакой противник. К тому же в игре с собой можно выработать дополнительные навыки, которые иначе остались бы нераскрытыми.

Чтобы уловить хаотичность и непрерывную природу реального мира, группу из пяти нейронных сетей, получившую название OpenAI Five, научили играть в *DOTA 2*, стратегическую игру реального времени, в которой две команды (по пять игроков) играют друг против друга. Сложность обучения в этом случае обусловлена длительными временными горизонтами (в среднем игра продолжается 45 минут и состоит из тысяч действий), частичной наблюдаемостью (каждый игрок видит только небольшую область вокруг себя) и непрерывными пространствами действий и наблюдений очень высокой размерности. В 2018 году OpenAI Five сыграла против игроков в *DOTA 2* на конкурсе The International. Она проиграла матч, но продемонстрировала прирожденные способности к сотрудничеству и выстраиванию стратегии. Затем, 13 апреля 2019 года, OpenAI Five официально победила чемпионов мира по этой игре и стала первым представителем искусственного интеллекта, превзошедшим профессиональную киберспортивную команду.

Робототехника и индустрия 4.0

ОП в промышленной робототехнике – область чрезвычайно активных исследований, поскольку является естественным внедрением этой парадигмы в практику. Потенциал интеллектуальных промышленных роботов и выгоды от их использования велики и многообразны. ОП наделяет индустрию 4.0 (или, как говорят, четвертую промышленную революцию) интеллектуальными устройствами, системами и роботами, которые могут выполнять очень сложные операции, действуя рациональным образом. Системы, умеющие прогнозировать необходимость технического обслуживания, диагностировать себя в режиме реального времени и управлять производственной деятельностью, можно объединить в единый комплекс, добиваясь улучшенного управления и более высокой производительности труда.

Машинное обучение

Гибкость ОП позволяет применять его не только для решения автономных задач, но и как своего рода метод точной настройки алгоритмов обучения с учителем. Во многих задачах **обработки естественных языков (ОЕЯ)** и компьютерного зрения подлежащая оптимизации функция не дифференцируема, поэтому для настройки параметров нейронной сети необходима вспомогательная дифференцируемая функция потерь. Но расхождение между двумя функциями потерь может снизить качество конечной системы. Один из способов решения этой проблемы состоит в том, чтобы сначала обучить систему, применяя методы обучения с учителем со вспомогательной функцией потерь, а затем использовать ОП для окончательной оптимизации сети относительно конечной метрики. Например, этот процесс можно с пользой применить в таких областях, как машинный перевод и вопросно-ответные системы, где метрики, по которым оценивается результат, сложны и недифференцируемы.

Кроме того, ОП может решать такие задачи ОЕЯ, как построение диалоговых систем и порождение текстов. Компьютерное зрение, локализация, анализ движения, визуальный контроль и визуальное слежение – все эти системы можно обучать методами ОП.

Глубокое обучение позволяет решить трудную задачу ручного конструирования признаков, оставляя человеку проектирование архитектуры нейронной сети. Это трудоемкая операция, смысл которой – наилучшим образом объединить несколько частей. Так почему бы не автоматизировать ее? Вообще-то можно. **Проектирование нейронной архитектуры** (neural architecture design – NAD) – подход, в котором ОП используется для проектирования архитектуры глубоких нейронных сетей (ГНС). Вычислительно он обходится очень дорого, но все же позволяет создавать архитектуры ГНС, не уступающие лучшим решениям в области классификации изображений.

Экономика и финансы

Управление бизнесом – еще одно естественное применение ОП. Оно успешно использовалось в интернет-рекламе, когда целью было максимизировать выручку от показа объявлений с платой за клик, касающихся рекомендаций по выбору продуктов, в сфере управления отношениями с клиентами и в маркетинге. А в области финансов ОП применялось для решения задач ценообразования опционов и многопериодической оптимизации.

Здравоохранение

В здравоохранении ОП используется для диагностики и лечения. С его помощью можно построить исходную оценку для снабженного искусственным интеллектом помощника врачей и медсестер. В частности, ОП помогает подготавливать индивидуальные прогрессивные назначения для пациентов – этот процесс называется режимом динамического лечения. Другие примеры применения ОП в здравоохранении – персонализированный контроль уровня глюкозы в крови и персонализированное лечение сепсиса и СПИДа.

Интеллектуальные транспортные системы

В этой области ОП применяется для разработки и улучшения всех типов транспортных систем: интеллектуальные сети для управления пробками (например, управление светофорами), наблюдение за дорожным движением, безопасность (прогнозирование столкновений) и беспилотные автомобили.

Оптимизация энергопотребления и умные сети электроснабжения

Оба направления – основа интеллектуальной генерации, распределения и потребления электричества. В системы принятия решений и контроля можно внедрить методы ОП, чтобы обеспечить динамическую реакцию на изменение условий в окружающей среде. ОП можно также использовать для регулирования спроса на электроэнергию в ответ на динамическое ценообразование или для сокращения потребления.

РЕЗЮМЕ

ОП – целеустремленный подход к принятию решения. От других парадигм он отличается прямым взаимодействием с окружающей средой и механизмом отложенного вознаграждения. Сочетание ОП с глубоким обучением очень полезно в задачах с многомерными пространствами состояний и с перцептивными входами. Понятия стратегии и функции ценности являются основными в ОП, поскольку говорят о том, какое действие предпринять, и о качестве состояний окружающей среды. В ОП модель окружающей среды необязательна, но может дать дополнительную информацию, а значит, улучшить качество стратегии.

Познакомившись с ключевыми концепциями, мы в последующих главах перейдем к самим алгоритмам ОП. Но сначала, прямо в следующей главе, зожим основы для разработки алгоритмов с помощью библиотек OpenAI и TensorFlow.

Вопросы

1. Что такое ОП?
2. Какова конечная цель агента?
3. Каковы основные различия между ОП и обучением с учителем?
4. Какие преимущества дает сочетание ОП с глубоким обучением?
5. Откуда берет начало термин «подкрепление»?
6. В чем разница между стратегией и функциями ценности?
7. Можно ли обучить модель окружающей среды посредством взаимодействия с ней?

Для дальнейшего чтения

- Пример неправильной функции вознаграждения см. по адресу <https://blog.openai.com/faulty-reward-functions/>.
- Для получения дополнительных сведений о глубоком ОП см. страницу по адресу <http://karpathy.github.io/2016/05/31/rl/>.

Глава 2

Реализация цикла ОП и OpenAI Gym

В каждом проекте машинного обучения алгоритм обучается на правилах и инструкциях, содержащихся в обучающем наборе данных, стремясь как можно лучше решить задачу. В **обучении с подкреплением (ОП)** алгоритм называется агентом, и обучается он на данных, предоставляемых окружающей средой. Здесь окружающая среда является непрерывным источником информации, который возвращает данные в соответствии с действиями агента. А поскольку данные, возвращаемые средой, потенциально бесконечны, имеется много концептуальных и практических различий между ситуациями, возникающими в процессе обучения. Впрочем, для целей этой главы важно лишь подчеркнуть тот факт, что различные среды предлагают не только различные задачи, но и различные типы входных и выходных данных, а также сигналов вознаграждения, но при этом в любом случае от алгоритма требуется умение адаптироваться. Например, робот может получать информацию о своем состоянии как в виде потока визуальных данных, например от RGB-камеры, так и от дискретных внутренних датчиков.

В этой главе мы настроим среду, необходимую для программирования алгоритмов ОП, и напишем свой первый алгоритм. Это простой алгоритм для игры в CartPole (балансирование стержня, шарнирно закрепленного на тележке), но на его примере мы получим полезный эталон для совершенствования основного цикла ОП, после чего можно будет переходить к более сложным алгоритмам. Кроме того, поскольку в последующих главах нам придется часто программировать глубокие нейронные сети, мы напомним основы библиотеки TensorFlow и познакомимся с инструментом визуализации TensorBoard.

Почти все среды, встречающиеся в этой книге, основаны на интерфейсе **Gym**, исходный код которого открыт компанией OpenAI. Поэтому мы рассмотрим его и воспользуемся некоторыми из встроенных в него окружающих сред. Затем, прежде чем переходить к более основательному изучению алгоритмов ОП, мы перечислим ряд окружающих сред с открытым исходным кодом и объясним, в чем состоят различия между ними и каковы сильные стороны каждой среды. Таким образом, вы сможете составить практическое представление о многообразии проблем, решаемых с помощью ОП.

В этой главе рассматриваются следующие вопросы:

- настройка окружающей среды;
- OpenAI Gym и цикл ОП;

- TensorFlow;
- TensorBoard;
- типы окружающих сред ОП.

НАСТРОЙКА ОКРУЖАЮЩЕЙ СРЕДЫ

Перечислим три главных элемента, необходимых для создания алгоритмов глубокого ОП.

- **Язык программирования.** Python – первый кандидат для разработки алгоритмов машинного обучения благодаря своей простоте и богатству написанных на нем сторонних библиотек.
- **Библиотека глубокого обучения.** В этой книге используется библиотека TensorFlow, поскольку, как мы увидим ниже, она масштабируемая, гибкая и весьма выразительная. Но есть и много других библиотек, например PyTorch и Caffe.
- **Окружающая среда.** В этой книге нам встретится много разных окружающих сред, на примере которых демонстрируется, как решать различные задачи, и объясняются сильные стороны алгоритмов ОП.

В этой книге используется версия Python 3.7, но должны работать все версии начиная с 3.5. Предполагается также, что уже установлены пакеты `numpy` и `matplotlib`.

Если библиотека TensorFlow еще не установлена, скачайте ее с сайта или установите, выполнив в окне терминала команду

```
$ pip install tensorflow
```

Если ваш компьютер оснащен графическим процессором (GPU), то можете вместо этого выполнить команду

```
$ pip install tensorflow-gpu
```

Инструкции по установке и упражнения, относящиеся к этой главе, можно найти в репозитории GitHub по адресу <https://github.com/PacktPublishing/Reinforcement-Learning-Algorithms-with-Python>.

А теперь посмотрим, как устанавливаются окружающие среды.

Установка OpenAI Gym

OpenAI Gym предлагает общий интерфейс, а также широкий спектр окружающих сред. Для установки выполните следующие команды.

В OS X:

```
$ brew install cmake boost boost-python sdl2 swig wget
```

В Ubuntu 16.04:

```
$ apt-get install -y python-pygame python3-opengl zlib1g-dev libjpeg-dev  
patchelf cmake swig libboost-all-dev libsdl2-dev libosmesa6-dev xvfb ffmpeg
```

В Ubuntu 18.04:

```
$ sudo apt install -y python3-dev zlib1g-dev libjpeg-dev cmake swig pythonpyglet
python3-opengl libboost-all-dev libsdl2-dev libosmesa6-dev patchelf
ffmpeg xvfb
```

Затем выполните такие команды:

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

Для некоторых окружающих сред Gym нужно установить еще `pybox2d`:

```
$ git clone https://github.com/pybox2d/pybox2d
$ cd pybox2d
$ pip install -e .
```

Установка Roboschool

И еще одна интересующая нас среда – эмулятор роботов Roboschool. Устанавливается он просто, но если возникнут ошибки, скачайте код из репозитория GitHub:

```
$ pip install roboschool
```

OPENAI GYM И ЦИКЛ ОП

Поскольку в ОП агент и окружающая среда должны взаимодействовать друг с другом, на ум сразу же приходит Земля – физический мир, в котором мы живем. Увы, в настоящее время она используется лишь в нескольких случаях. Проблемы в современных алгоритмах проистекают из очень большого количества взаимодействий со средой, которые должен выполнить агент, чтобы обучиться хорошему поведению. Может потребоваться сотни, тысячи и даже миллионы действий, а это занимает слишком много времени. Одно из решений – воспользоваться имитированными окружающими средами в начале процесса обучения и лишь в конце произвести точную настройку в реальном мире. Это намного лучше, чем обучаться непосредственно в окружающем мире, но все равно необходимы медленные взаимодействия с реальной средой. Однако во многих случаях задачу можно полностью имитировать. Для исследования и реализации алгоритмов ОП идеальным испытательным стендом являются игры и эмуляторы роботов, поскольку для решения соответствующих задач необходимы способности к планированию, выработке стратегии и долгосрочная память. К тому же в играх имеется ясная система вознаграждения, и их можно полностью имитировать в искусственной окружающей среде (на компьютере), что ускоряет взаимодействие, а значит, и процесс обучения. Поэтому для демонстрации алгоритмов ОП в этой книге используются в основном видеоигры и эмуляторы роботов.

OpenAI Gym, комплект инструментов с открытым исходным кодом для разработки и исследования алгоритмов ОП, создан с целью предоставить единый интерфейс к окружающим средам, не ограничивая при этом количество и разнообразие возможных сред. В их число входят игры для Atari 2600, задачи с непрерывным управлением, классические задачи теории управления, целеустремленные задачи для эмуляторов роботов и простые текстовые игры. Благодаря общности интерфейса Gym многие сторонние компании используют его для создания собственных сред.

Разработка цикла ОП

Базовый цикл ОП показан в следующем ниже фрагменте кода. В нем модель ОП играет на протяжении 10 ходов и на каждом ходе рисует состояние игры.

```
import gym

# создать окружающую среду
env = gym.make("CartPole-v1")
# привести среду в исходное состояние перед началом
env.reset()

# повторять 10 раз
for i in range(10):
    # предпринять случайное действие
    env.step(env.action_space.sample())
    # нарисовать состояние игры
    env.render()

# закрыть окружающую среду
env.close()
```

Получается такая картинка:

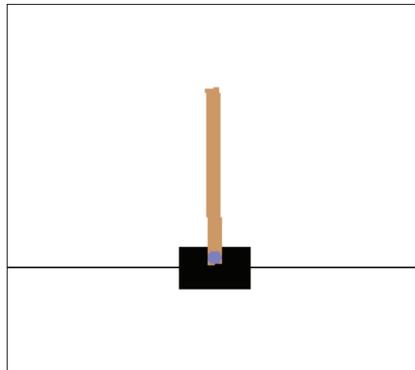


Рис. 2.1 ❖ Изображение игры CartPole

Рассмотрим этот код внимательнее. Вначале создается новая окружающая среда с именем CartPole-v1 – классическая игра, используемая при исследовании задач теории управления. Но прежде чем использовать, ее необходимо

инициализировать, обратившись к методу `reset()`. Затем цикл повторяется 10 раз. На каждой итерации метод `env.action_space.sample()` случайно выбирает действие, затем оно выполняется в окружающей среде методом `env.step()`, после чего результат – текущее состояние игры – отображается методом `render()`, как показано на рис. 2.1. В конце окружающей среда закрывается методом `env.close()`.

❗ Не пугайтесь, если этот код выдаст предупреждения о нерекондуемом коде. Это просто означает, что какие-то функции были изменены. Программа все равно будет работать правильно.

Этот цикл одинаков для любой среды, построенной на базе интерфейса Gym, но пока что агент только выбирает случайные действия, не получая никакой обратной связи, а без этого никакое обучение с подкреплением невозможно.

❗ В ОП термины **состояние** и **наблюдение** часто употребляются как синонимы, но на самом деле это не одно и то же. В состоянии закодирована вся информация, относящаяся к окружающей среде. А наблюдение подразумевает только часть истинного состояния среды, видимую агенту, например системе восприятия робота. Для простоты в OpenAI Gym всегда используется термин «наблюдение».

На рис. 2.2 ниже показан поток управления в цикле.



Рис. 2.2 ❖ Базовый цикл ОП в интерфейсе OpenAI Gym. Окружающая среда возвращает следующее состояние, вознаграждение, флаг завершения (`done`) и некоторую дополнительную информацию

В действительности метод `step()` возвращает четыре переменные, содержащие информацию о взаимодействии с окружающей средой. На рис. 2.2 показан цикл взаимодействия агента со средой, а также переменные, которыми они обмениваются: **Observation**, **Reward**, **Done** и **Info**. **Observation** – объект, представляющий новое наблюдение (или состояние) окружающей среды. **Reward** – число с плавающей точкой, равное вознаграждению, полученному за последнее действие. **Done** – булево значение, используемое в эпизодических задачах, т. е. задачах, в которых количество взаимодействий ограничено. Если `done` равно `True`, то эпизод завершился и среду следует переустановить. Например, `done` равно `True`, когда задача выполнена до конца или агент «умер». А объект **Info** содержит словарь с дополнительной информацией о среде, но обычно он не используется.

Для тех, кто не знает, скажем, что CartPole – игра, в которой требуется балансировать стержень, шарнирно закрепленный на горизонтально движущейся тележке. Вознаграждение +1 назначается за каждый шаг, на котором стержень стоит прямо. Эпизод заканчивается, когда стержень наклонился слишком сильно или оставался в равновесии в течение 200 шагов (за что было начислено максимальное полное вознаграждение 200).

Теперь можно написать более полный алгоритм, который играет 10 игр и печатает полное вознаграждение в каждой игре.

```
import gym

# создать и инициализировать окружающую среду
env = gym.make("CartPole-v1")
env.reset()

# сыграть 10 игр
for i in range(10):
    # инициализировать переменные
    done = False
    game_gew = 0

    while not done:
        # выбрать случайное действие
        action = env.action_space.sample()
        # выполнить один шаг взаимодействия с окружающей средой
        new_obs, rew, done, info = env.step(action)
        game_gew += rew
        # если завершено, напечатать полное вознаграждение в игре и сбросить среду
        if done:
            print('Эпизод %d завершен, Вознаграждение:%d' % (i, game_gew))
            env.reset()
```

Результат будет выглядеть примерно так:

```
Эпизод: 0, Вознаграждение:13
Эпизод: 1, Вознаграждение:16
Эпизод: 2, Вознаграждение:23
Эпизод: 3, Вознаграждение:17
Эпизод: 4, Вознаграждение:30
Эпизод: 5, Вознаграждение:18
Эпизод: 6, Вознаграждение:14
Эпизод: 7, Вознаграждение:28
Эпизод: 8, Вознаграждение:22
Эпизод: 9, Вознаграждение:16
```

В следующей таблице показаны результаты метода `step()` для последних четырех действий:

Observation	Reward	Done	Info
[-0.05356921, -0.38150626, 0.12529277, 0.9449761]	1.0	False	{}
[-0.06119933, -0.57807287, 0.14419229, 1.27425449]	1.0	False	{}
[-0.07276079, -0.38505429, 0.16967738, 1.02997704]	1.0	False	{}
[-0.08046188, -0.58197758, 0.19027692, 1.37076617]	1.0	False	{}
[-0.09210143, -0.3896757, 0.21769224, 1.14312384]	1.0	True	{}

Заметим, что наблюдения окружающей среды закодированы в виде массива 1×4 , что вознаграждение, как и ожидалось, всегда равно 1 и что флаг `done` равен `True` только в последней строке, соответствующей завершению игры. Кроме того, словарь **Info** в данном случае пуст.

В следующих главах мы создадим агентов, которые будут играть в `CartPole`, предпринимая более осмысленные действия, зависящие от текущего положения стержня.

Привыкаем к пространствам

В `OpenAI Gym` действия и наблюдения в основном являются экземплярами класса `Discrete` или `Box`. Эти два класса представляют разные пространства. Класс `Box` представляет n -мерный массив, а `Discrete` – пространство, допускающее фиксированный диапазон неотрицательных чисел. В таблице выше мы уже видели, что одно наблюдение `CartPole` кодируется четырьмя числами с плавающей точкой, т. е. экземпляром класса `Box`. Мы можем узнать тип и размерность пространства наблюдений, распечатав переменную `env.observation_space`:

```
import gym

env = gym.make('CartPole-v1')
print(env.observation_space)
```

И действительно, результат вполне ожидаем:

```
>> Box(4,)
```

 В этой книге тексту, напечатанному функцией `print()`, предшествует префикс `>>`.

Точно так же можно узнать размерность пространства действий:

```
print(env.action_space)
```

В ответ будет напечатано:

```
>> Discrete(2)
```

`Discrete(2)` означает, что действие может принимать два значения: 0 или 1. Действительно, воспользовавшись выборочной функцией в предыдущем примере, мы будем получать 0 или 1 (в игре `CartPole` это означает перемещение влево или вправо):

```
print(env.action_space.sample())
>> 0
print(env.action_space.sample())
>> 1
```

Атрибуты экземпляра `low` и `high` возвращают минимальное и максимальное допустимые значения в пространстве `Box`:

```
print(env.observation_space.low)
>> [-4.8000002e+00 -3.4028235e+38 -4.1887903e-01 -3.4028235e+38]
print(env.observation_space.high)
>> [4.8000002e+00 3.4028235e+38 4.1887903e-01 3.4028235e+38]
```

РАЗРАБОТКА МОДЕЛЕЙ МО С ПОМОЩЬЮ TENSORFLOW

TensorFlow – это библиотека машинного обучения (МО), которая берет на себя численные вычисления с высокой производительностью. Своей популярностью она обязана высокому качеству и подробной документации, способности создавать масштабируемые модели в производственной среде и наличию удобного интерфейса с GPU и тензорными процессорами (TPU).

Над TensorFlow надстроено много высокоуровневых API, упрощающих разработку и развертывание моделей МО, в т. ч. Keras, Eager Execution и Estimators. Эти API очень полезны во многих контекстах, но для разработки алгоритмов ОП мы будем пользоваться только низкоуровневыми API.

Не откладывая в долгий ящик, приступим к работе с **TensorFlow**. В следующих строках вычисляется сумма констант a и b , созданных методом `tf.constant()`:

```
import tensorflow as tf

# создать две константы: a и b
a = tf.constant(4)
b = tf.constant(3)

# выполнить вычисление
c = a + b

# создать сеанс
session = tf.Session()

# выполнить сеанс. В нем вычисляется сумма
res = session.run(c)
print(res)
```

Отличительной особенностью TensorFlow является тот факт, что все вычисления выражаются в виде графа вычислений, который сначала необходимо определить, а затем выполнить. И лишь после выполнения графа становятся доступны результаты вычислений. В примере выше после операции $c = a + b$ переменная c еще не содержит никакого значения. Действительно, если распечатать c до создания сеанса, то получим:

```
>> Tensor("add:0", shape=(), dtype=int32)
```

Это класс переменной c , а не результат сложения.

Вычисление должно быть произведено внутри сеанса, созданного методом `tf.Session()`. Для этого методу `run` сеанса следует передать операцию. Таким образом, чтобы вычислить граф и, стало быть, сумму a и b , мы должны создать сеанс и передать c ему на вход:

```
session = tf.Session()
res = session.run(c)
print(res)
```

```
>> 7
```



Если вы пользуетесь Jupyter-блокнотом, то не забудьте инициализировать граф, вызвав метод `tf.reset_default_graph()`.

Тензоры

Переменные в TensorFlow представляются в виде тензоров – массивов с произвольным числом измерений. Существует три типа тензоров: `tf.Variable`, `tf.constant` и `tf.placeholder`. Все они, кроме `tf.Variable`, неизменяемы.

Чтобы узнать форму тензора, мы пишем:

```
# константа
a = tf.constant(1)
print(a.shape)
>> ()

# массив из пяти элементов
b = tf.constant([1,2,3,4,5])
print(b.shape)
>> (5,)
```

К элементам тензора легко обратиться с помощью механизмов, похожих на встроенные в Python:

```
a = tf.constant([1,2,3,4,5])
first_three_elem = a[:3]
fourth_elem = a[3]

sess = tf.Session()
print(sess.run(first_three_elem))

>> array([1,2,3])

print(sess.run(fourth_elem))
>> 4
```

Константа

Как мы уже видели, константа – неизменяемая разновидность тензора, которую легко создать методом `tf.constant`:

```
a = tf.constant([1.0, 1.1, 2.1, 3.1], dtype=tf.float32, name='a_const')
print(a)
>> Tensor("a_const:0", shape=(4,), dtype=float32)
```

Местозаполнитель

Местозаполнитель (placeholder) – это тензор, который получает значение во время выполнения. Обычно местозаполнители используются в качестве входов модели. На каждый вход графа вычислений на этапе выполнения подаются данные с помощью факультативного аргумента `feed_dict`, который позволяет вызывающей стороне переопределить значения тензоров в графе. В следующем фрагменте местозаполнитель заменяется значением `[[0.1,0.2,0.3]]`:

```
import tensorflow as tf

a = tf.placeholder(shape=(1,3), dtype=tf.float32)
b = tf.constant([[10,10,10]], dtype=tf.float32)

c = a + b

sess = tf.Session()
```

```
res = sess.run(c, feed_dict={a:[[0.1,0.2,0.3]])
print(res)
```

```
>> [[10.1 10.2 10.3]]
```

Если размер по первому измерению входа неизвестен на этапе создания графа, то TensorFlow может взять все хлопоты на себя. Просто передайте в качестве значения None:

```
import tensorflow as tf
import numpy as np

# NB: размер по первому измерению равен 'None', т. е. длина может быть любой
a = tf.placeholder(shape=(None,3), dtype=tf.float32)
b = tf.placeholder(shape=(None,3), dtype=tf.float32)

c = a + b
print(a)
```

```
>> Tensor("Placeholder:0", shape=(?, 3), dtype=float32)
```

```
sess = tf.Session()
print(sess.run(c, feed_dict={a:[[0.1,0.2,0.3]], b:[[10,10,10]]}))
```

```
>> [[10.1 10.2 10.3]]
```

```
v_a = np.array([[1,2,3],[4,5,6]])
v_b = np.array([[6,5,4],[3,2,1]])
print(sess.run(c, feed_dict={a:v_a, b:v_b}))
```

```
>> [[7. 7. 7.]
      [7. 7. 7.]]
```

Эта возможность полезна, когда количество обучающих примеров заранее неизвестно.

Переменная

Переменная – это изменяемый тензор, который можно обучить с помощью оптимизатора. Например, могут существовать свободные переменные, определяющие веса и смещения нейронной сети.

Создадим две переменные, одна из которых инициализирована известными значениями, а вторая – случайными:

```
import tensorflow as tf
import numpy as np

# переменная, инициализированная случайными значениями
var = tf.get_variable("first_variable", shape=[1,3], dtype=tf.float32)

# переменная, инициализированная константами
init_val = np.array([4,5])
var2 = tf.get_variable("second_variable", shape=[1,2], dtype=tf.int32,
initializer=tf.constant_initializer(init_val))

# создать сеанс
sess = tf.Session()
```

```
# инициализировать все переменные
sess.run(tf.global_variables_initializer())

print(sess.run(var))
>> [[ 0.93119466 -1.0498083 -0.2198658 ]]

print(sess.run(var2))
>> [[4 5]]
```

Переменные остаются неинициализированными, пока не вызван метод `global_variables_initializer()`.

Все созданные до сих пор переменные помечаются как `trainable`, т. е. граф может модифицировать их, например, после операции оптимизации. Переменные можно создать и как необучаемые:

```
var2 = tf.get_variable("variable", shape=[1,2], trainable=False,
dtype=tf.int32)
```

Ниже показан простой способ получить все переменные:

```
print(tf.global_variables())
>> [<tf.Variable 'first_variable:0' shape=(1, 3) dtype=float32_ref>,
<tf.Variable 'second_variable:0' shape=(1, 2) dtype=int32_ref>]
```

Создание графа

Граф представляет низкоуровневые вычисления в терминах зависимостей между операциями. В TensorFlow сначала определяется граф, а затем создается сеанс, в контексте которого выполняются все включенные в граф операции.

Создание, вычисление и оптимизация графа в TensorFlow устроены так, что обеспечивается высокая степень параллелизма, распределенное выполнение и переносимость. Все эти свойства очень важны для построения моделей машинного обучения.

Чтобы вы могли составить представление о внутренней структуре графа, порождаемого TensorFlow, напишем программу, которая создает граф, показанный на рис. 2.3.

```
import tensorflow as tf
import numpy as np

const1 = tf.constant(3.0, name='constant1')

var = tf.get_variable("variable1", shape=[1,2], dtype=tf.float32)
var2 = tf.get_variable("variable2", shape=[1,2], trainable=False, dtype=tf.float32)

op1 = const1 * var
op2 = op1 + var2
op3 = tf.reduce_mean(op2)

sess = tf.Session()
sess.run(tf.global_variables_initializer())
sess.run(op3)
```

Вот как выглядит получающийся граф:

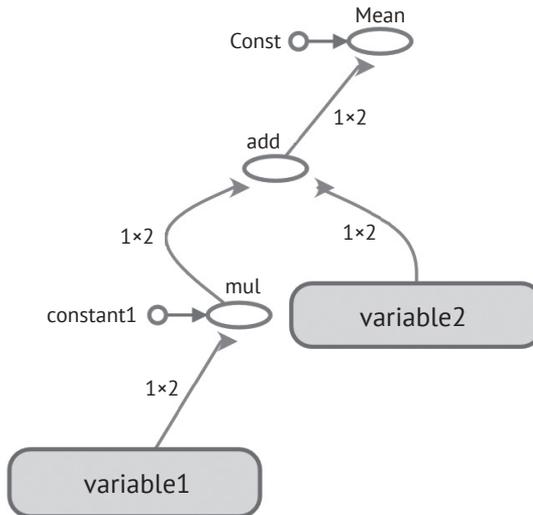


Рис. 2.3 ❖ Пример графа вычислений

Простой пример линейной регрессии

Чтобы переварить все введенные понятия, создадим простую модель линейной регрессии. Сначала импортируем все библиотеки и зададим конкретные начальные значения для генераторов случайных чисел, входящих в NumPy и TensorFlow (чтобы результаты были воспроизводимы):

```
import tensorflow as tf
import numpy as np
from datetime import datetime

np.random.seed(10)
tf.set_random_seed(10)
```

Затем можно создать синтетический набор данных, содержащий 100 примеров, показанный на рис. 2.4.

Так как это линейная регрессия, то $y = W * X + b$, где W и b – какие-то числа. Положим $W = 0.5$ и $b = 1.4$. Добавим также нормально распределенный случайный шум:

```
W, b = 0.5, 1.4
# создать набор данных, содержащий 100 примеров
X = np.linspace(0,100, num=100)

# добавить случайный шум в метки y
y = np.random.normal(loc=W * X + b, scale=2.0, size=len(X))
```

Следующий шаг – создать местозаполнители для входа и выхода, а также переменные, содержащие вес и смещение линейной модели. В процессе обучения эти переменные будут оптимизированы, чтобы как можно лучше аппроксимировать истинные вес и смещение.

```
# создать местозаполнители
x_ph = tf.placeholder(shape=[None,], dtype=tf.float32)
y_ph = tf.placeholder(shape=[None,], dtype=tf.float32)

# создать переменные
v_weight = tf.get_variable("weight", shape=[1], dtype=tf.float32)
v_bias = tf.get_variable("bias", shape=[1], dtype=tf.float32)
```

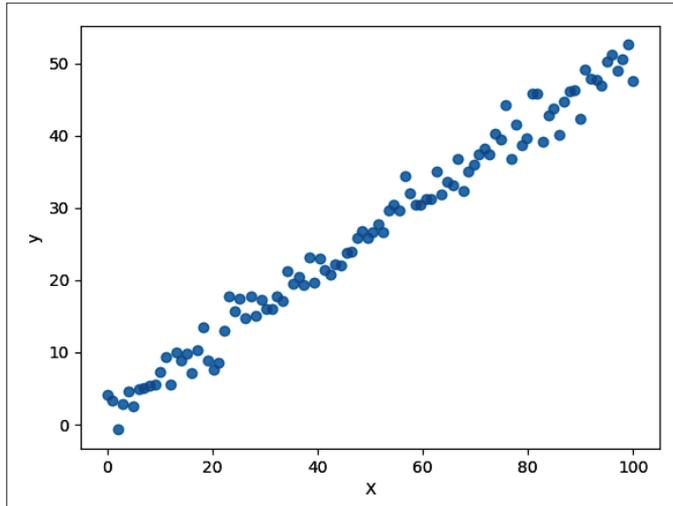


Рис. 2.4 ❖ Набор данных для примера линейной регрессии

Далее строим граф вычислений, определяющий линейную операцию и потерю, в качестве которой возьмем **среднеквадратическую ошибку (СКО)**:

```
# вычисление линейной функции
out = v_weight * x_ph + v_bias

# вычислить среднеквадратическую ошибку
loss = tf.reduce_mean((out - y_ph)**2)
```

Теперь можно создать экземпляр оптимизатора и вызвать его метод `minimize()`, чтобы минимизировать СКО-потерю. Метод `minimize()` сначала вычисляет градиенты переменных (`v_weight` и `v_bias`), а затем с их помощью обновляет переменные:

```
opt = tf.train.AdamOptimizer(0.4).minimize(loss)
```

Создадим сеанс и инициализируем переменные:

```
session = tf.Session()
session.run(tf.global_variables_initializer())
```

Для обучения несколько раз прогоним оптимизатор, подавая в граф набор данных. Чтобы следить за состоянием модели, СКО и переменными модели (весом и смещением), будем через каждые 40 итераций (эпох) распечатывать их значения:

```
# цикл обучения параметров
for ep in range(210):
    # прогнать оптимизатор и получить потерю
    train_loss, _ = session.run([loss, opt], feed_dict={x_ph:X, y_ph:y})

    # печатать номер эпохи и потерю
    if ep % 40 == 0:
        print('Эпоха: %3d, SKO: %.4f, W: %.3f, b: %.3f' % (ep, train_loss,
            session.run(v_weight), session.run(v_bias)))
```

В конце напечатаем окончательные значения переменных:

```
print('Окончательный вес: %.3f, смещение: %.3f' % (session.run(v_weight),
    session.run(v_bias)))
```

В результате будет напечатано что-то типа:

```
>> Эпоха: 0, SKO: 4617.4390, вес: 1.295, смещение: -0.407
Эпоха: 40, SKO: 5.3334, вес: 0.496, смещение: -0.727
Эпоха: 80, SKO: 4.5894, вес: 0.529, смещение: -0.012
Эпоха: 120, SKO: 4.1029, вес: 0.512, смещение: 0.608
Эпоха: 160, SKO: 3.8552, вес: 0.506, смещение: 1.092
Эпоха: 200, SKO: 3.7597, вес: 0.501, смещение: 1.418
Окончательный вес: 0.500, смещение: 1.473
```

В процессе обучения видно, что потеря убывает и стремится к ненулевому значению (приблизительно 3.71). Это связано со случайным шумом, который мешает SKO принять идеальное значение 0.

Что касается вычисленных веса и смещения (0.500 и 1.473), то, как и следовало ожидать, они очень близки к значениям, которые использовались при построении набора данных. Синяя линия на рис. 2.5 – предсказание, выданное обученной моделью, а точками обозначены обучающие примеры.

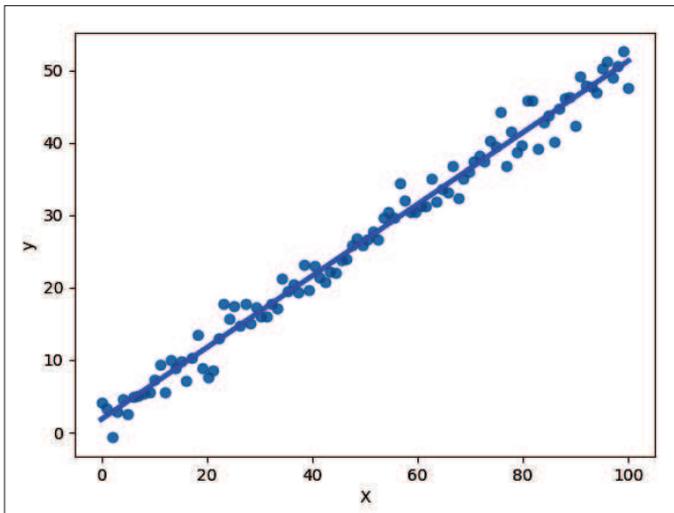


Рис. 2.5 ❖ Предсказание модели линейной регрессии

ВВЕДЕНИЕ В TENSORBOARD

Следить, как изменяются переменные в процессе обучения модели, утомительно. Например, в программе линейной регрессии мы следили за потерей и параметрами модели, печатая их через каждые 40 эпох. Но вместе со сложностью алгоритма растет и количество переменных и показателей, за которыми надо бы следить. По счастью, на помощь приходит TensorBoard.

TensorBoard – это комплект средств визуализации, которые можно использовать для построения графиков показателей, визуализации графов TensorFlow и дополнительной информации. Типичный экран TensorBoard показан на рис. 2.6.

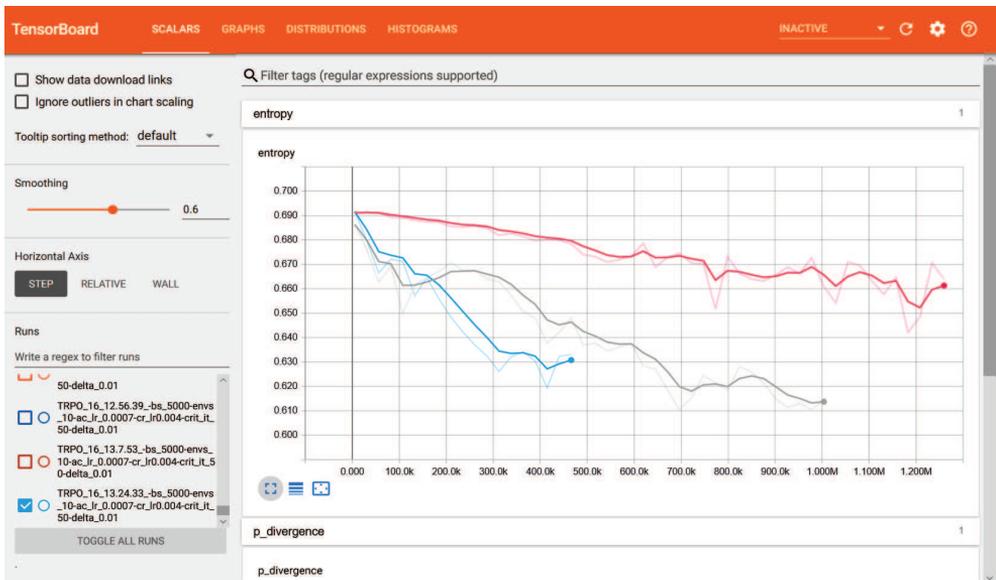


Рис. 2.6 ❖ Страница TensorBoard

Интегрировать TensorBoard с TensorFlow довольно просто, нужно лишь добавить в код несколько строчек. В частности, для визуализации изменения СКО-потери со временем и мониторинга веса и смещения нашей модели линейной регрессии нужно сначала присоединить тензор потери `loss` к `tf.summary.scalar()`, а параметры модели – к `tf.summary.histogram()`. Приведенный ниже фрагмент следует вставить перед вызовом оптимизатора:

```
tf.summary.scalar('MSEloss', loss)
tf.summary.histogram('model_weight', v_weight)
tf.summary.histogram('model_bias', v_bias)
```

Чтобы упростить процесс и представить единую сводку, мы можем объединить все три величины:

```
all_summary = tf.summary.merge_all()
```

Сейчас необходимо создать экземпляр `FileWriter`, который будет записывать сводную информацию в файл:

```
now = datetime.now()
clock_time = "{}_{}.{}.{}".format(now.day, now.hour, now.minute, now.second)
file_writer = tf.summary.FileWriter('log_dir/'+clock_time,
tf.get_default_graph())
```

В первых двух строчках создается уникальное имя файла на основе текущих даты и времени. В третьей строчке путь к файлу и граф TensorFlow передаются объекту `FileWriter`. Вторым параметром необязателен и представляет подлежащий визуализации граф.

И последнее изменение – заменить в цикле обучения строку `train_loss, _ = session.run(..)` такой:

```
train_loss, _, train_summary = session.run([loss, opt, all_summary],
feed_dict={x_ph:X, y_ph:y})
file_writer.add_summary(train_summary, ep)
```

Сначала `all_summary` вычисляется в текущем сеансе, а затем результат добавляется в `file_writer` для записи в файл. Эта процедура вычислит все три сводки, которые мы перед этим объединили, и запишет их в журнальный файл. Затем `TensorBoard` прочтет данные из файла и визуализирует обе гистограммы и граф вычислений.

Не забудьте в конце закрыть `file_writer`:

```
file_writer.close()
```

Наконец, мы можем открыть `TensorBoard`, для чего перейдем в рабочий каталог и введем в терминале такую команду:

```
$ tensorboard --logdir=log_dir
```

Эта команда создает веб-сервер, который прослушивает порт 6006. Чтобы запустить `TensorBoard`, необходимо перейти по показанной ссылке (рис. 2.7).

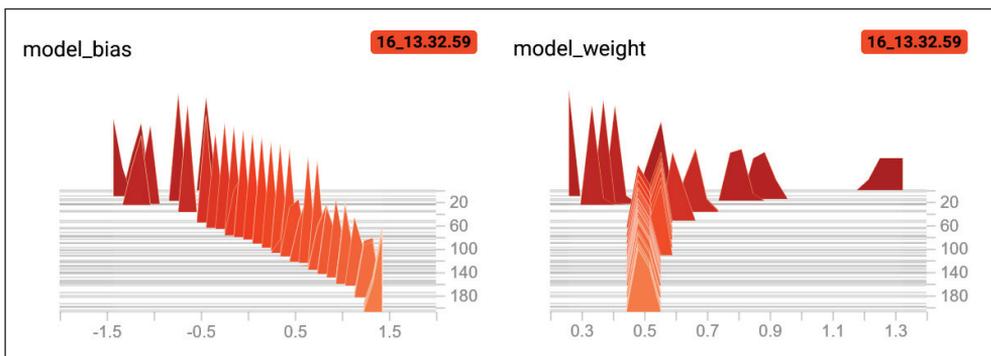


Рис. 2.7 ❖ Гистограмма параметров модели линейной регрессии

Теперь мы можем походить по `TensorBoard`, щелкая по вкладкам в верхней части страницы, чтобы посмотреть на графики, гистограммы и сам граф. На

предыдущем – и на последующих – снимке экрана показаны некоторые результаты визуализации. Графики и граф интерактивны, так что потратьте некоторое время, чтобы понять, как с ними работать. Загляните также в официальную документацию по TensorBoard (https://www.tensorflow.org/guide/summaries_and_tensorboard), чтобы познакомиться с дополнительными возможностями.

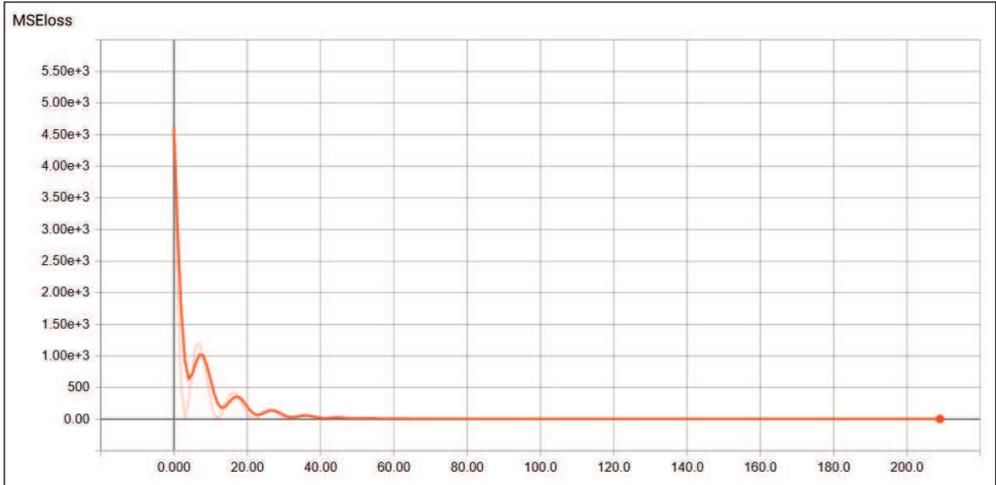


Рис. 2.8 ❖ График СКО-потери

Типы окружающих сред ОП

Окружающие среды, как и размеченные наборы данных в обучении с учителем, – неотъемлемая часть ОП, поскольку они определяют, чему надо обучиться, а равно диктуют выбор алгоритмов. В этом разделе мы рассмотрим основные различия между типами окружающих сред и перечислим наиболее важные среды с открытым исходным кодом.

Зачем нужны различные среды?

Если в реальных приложениях выбор окружающей среды продиктован решаемой задачей, то в исследовательских – обычно внутренними свойствами среды. В последнем случае конечная цель – не столько обучить агента на конкретной задаче, сколько продемонстрировать какие-то связанные с задачей возможности.

Например, если целью является создание многоагентного алгоритма ОП, то в среде должно быть по меньшей мере два агента, способных взаимодействовать друг с другом, и от задачи это не зависит. С другой стороны, если нужно создать пожизненного обучаемого (агента, который непрерывно создает все более трудные задачи и обучается их решению, применяя знания, полученные при решении предыдущих, более простых задач), то основное качество,

которым должна обладать среда, – способность адаптации к новым ситуациям и реалистичной предметной области.

Помимо задач, окружающие среды могут отличаться и другими характеристиками, например сложностью, пространством наблюдений, пространством действий и функцией вознаграждения.

- **Сложность.** Спектр окружающих сред широк: от балансирования стержня до манипуляций физическими предметами с помощью роботизированной руки. Чтобы продемонстрировать способность алгоритма справляться с большим пространством состояний, имитирующим сложность реального мира, можно выбрать более сложную окружающую среду. С другой стороны, если нужно продемонстрировать лишь некоторые конкретные качества, то достаточно среды попроще.
- **Пространство наблюдений.** Как мы уже видели, пространство наблюдений может варьироваться от полного состояния окружающей среды до частичного наблюдения, доступного системе восприятия, например изображения, состоящего из строк пикселей.
- **Пространство действий.** Среды с большим непрерывным пространством состояний вынуждают агента иметь дело с вещественными векторами, тогда как в случае дискретных действий обучиться проще, поскольку количество действий ограничено.
- **Функция вознаграждения.** Среды с большим объемом исследования и отложенным вознаграждением, например игра Montezuma's revenge (Мечь Монтесумы), с трудом поддаются решению. Как ни странно, лишь немногие алгоритмы способны достичь уровня человека. Поэтому такие среды используются как испытательный стенд для алгоритмов, предназначенных для решения проблемы исследования.

Окружающие среды с открытым исходным кодом

Как спроектировать окружающую среду, удовлетворяющую поставленным требованиям? По счастью, существует много сред с открытым исходным кодом, предназначенных для решения конкретной проблемы или класса проблем. Например, среда CoinRun, показанная на рис. 2.9, была создана для измерения способности алгоритма к обобщению.

Перечислим несколько основных сред с открытым исходным кодом. Они созданы разными командами и компаниями, но почти все совместимы с интерфейсом OpenAI Gym.

- **Gym Atari** (<https://gym.openai.com/envs/#atari>). Включает игры для Atari 2600, в которых входными данными служат изображения на экране. Полезны для измерения качества алгоритмов ОП на широком спектре игр с одним и тем же пространством наблюдений.
- **Gym Classic control** (https://gym.openai.com/envs/#classic_control). Классические игры, которые можно использовать для оценки и отладки алгоритма.
- **Gym MuJoCo** (<https://gym.openai.com/envs/#mujoco>). Включает непрерывные задачи управления (например, Ant и HalfCheetah), построенные поверх MuJoCo, физического движка с платной лицензией (бесплатная лицензия доступна студентам).

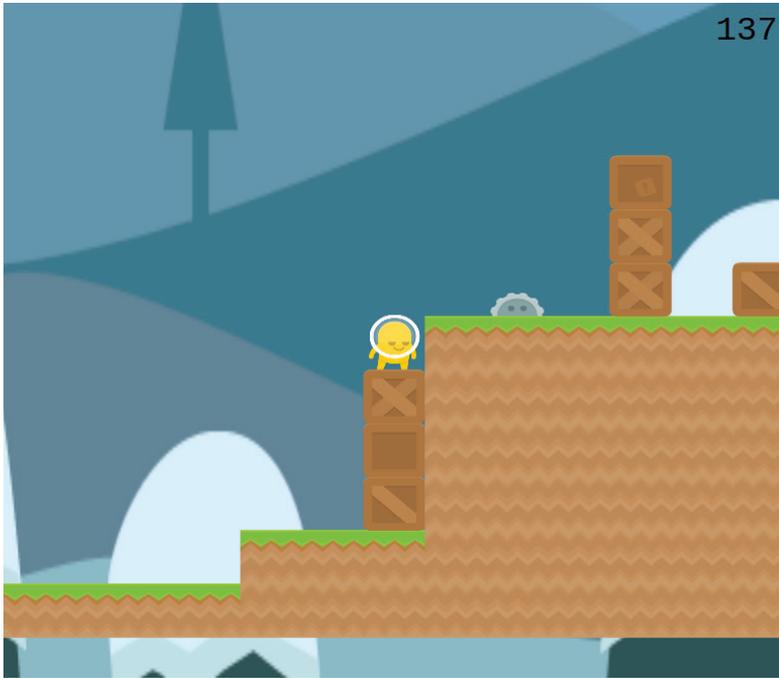


Рис. 2.9 ❖ Окружающая среда CoinRun

- **MalmoEnv** (<https://github.com/Microsoft/malmo>). Среда, построенная поверх Minecraft.
- **Pommerman** (<https://github.com/MultiAgentLearning/playground>). Отличная среда для обучения многоагентных алгоритмов. Игра Pommerman – вариант знаменитой игры Bomberman.
- **Roboschool** (<https://github.com/openai/roboschool>). Среда эмуляции роботов, интегрированная с OpenAI Gym. Включает копию окружающей среды MuJoCo, показанной на рис. 2.10, две интерактивные среды для повышения надежности агента и одну многопользовательскую среду.
- **Duckietown** (<https://github.com/duckietown/gym-duckietown>). Эмулятор беспилотного автомобиля с различными картами местности и препятствиями.
- **PLE** (<https://github.com/ntasfi/PyGame-Learning-Environment>). Включает много аркадных игр, в т. ч. Monster Kong, FlappyBird и Snake.
- **Unity ML-Agents** (<https://github.com/Unity-Technologies/ml-agents>). Среда, построенные поверх Unity с реалистичными физическими законами. ML-agents допускает большую свободу и позволяет создавать собственные окружающие среды с использованием Unity.
- **CoinRun** (<https://github.com/openai/coinrun>). Среда, предназначенная для решения проблемы переобучения в ОП. Генерирует различные среды для обучения и тестирования.
- **DeepMind Lab** (<https://github.com/deepmind/lab>). Комплект трехмерных сред для решения задач навигации и сборки пазлов.

- **DeepMind PySC2** (<https://github.com/deepmind/pysc2>). Среда для обучения сложной игре, StarCraft II.

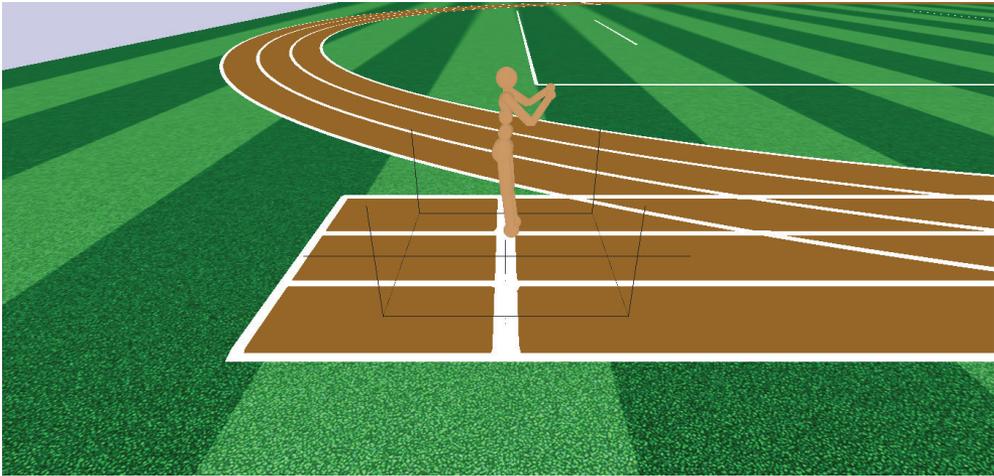


Рис. 2.10 ❖ Окружающая среда Roboschool

РЕЗЮМЕ

Мы надеемся, что в этой главе вы узнали все, что нужно знать об инструментах и компонентах для построения алгоритмов ОП. Мы настроили среду разработки на Python и написали первый алгоритм с применением окружающей среды OpenAI Gym. Поскольку в большинстве передовых алгоритмов ОП используется глубокое обучение, мы познакомились с библиотекой TensorFlow, которая будет постоянно встречаться в книге. Применение TensorFlow ускоряет разработку алгоритмов глубокого ОП, поскольку библиотека берет на себя наиболее сложные части обучения глубоких нейронных сетей, в т. ч. алгоритм обратного распространения. Кроме того, в комплекте с TensorFlow идет TensorBoard – инструмент визуализации, который используется для мониторинга и отладки алгоритмов.

Поскольку в следующих главах мы будем использовать много окружающих сред, важно четко представлять себе, чем они различаются. Сейчас вы уже знаете достаточно, чтобы выбрать подходящую среду для своего проекта, но имейте в виду, что приведенный нами список неполон, могут существовать и другие среды, которые лучше отвечают вашей задаче.

В следующих главах мы научимся разрабатывать алгоритмы ОП. Точнее, в главе 3 будут представлены алгоритмы, которые можно использовать в простых задачах, когда окружающая среда полностью известна. Затем мы перейдем к более сложным случаям.

Вопросы

1. Что выводит функция `step()` в Gym?
2. Как выбрать случайное действие с помощью интерфейса OpenAI Gym?
3. В чем главное различие между классами `Box` и `Discrete`?
4. Для чего в ОП используются библиотеки глубокого обучения?
5. Что такое тензор?
6. Что можно визуализировать в TensorBoard?
7. Какую из упомянутых в этой главе окружающих сред вы стали бы использовать для создания беспилотного автомобиля?

Для дальнейшего чтения

- Официальное руководство по TensorFlow находится по адресу https://www.tensorflow.org/guide/low_level_intro.
- Официальное руководство по TensorBoard находится по адресу https://www.tensorflow.org/guide/summaries_and_tensorboard.

Глава 3

Решение задач методом динамического программирования

У этой главы несколько целей. Мы познакомимся со многими вопросами, важными для понимания проблем, стоящих перед обучением с подкреплением, а также с первыми алгоритмами, предложенными для их решения. Если ранее мы говорили об **обучении с подкреплением (ОП)** в общих чертах, не уточняя технические детали, то теперь формализуем постановку задачи и разработаем алгоритм для решения простой игры.

Проблему ОП можно поставить как **марковский процесс принятия решений (МППР)** – это формализует основные элементы ОП, в т. ч. функции ценности и ожидаемое вознаграждение. Затем, пользуясь математическим аппаратом этой теории, можно приступить к созданию алгоритмов ОП. Алгоритмы различаются способом соединения компонентов и исходными предположениями.

Как мы увидим в этой главе, алгоритмы ОП можно разбить на три категории. Они частично перекрываются, потому что некоторые алгоритмы обладают свойствами, характерными для нескольких категорий. Разъяснив фундаментальные понятия, мы построим алгоритм первого типа – динамического программирования, который применим к задачам, в которых имеется полная информация об окружающей среде.

В этой главе рассматриваются следующие вопросы:

- МППР;
- классификация алгоритмов ОП;
- динамическое программирование.

МППР

МППР описывает способ последовательного принятия решений, когда выбранное действие влияет на следующие состояния и результаты. Процессы МППР достаточно общие и гибкие и позволяют корректно поставить задачу обучения достижению цели посредством взаимодействий, т. е. именно ту задачу, которую решает ОП.

МППР представляет собой четверку (S, A, P, R) , где:

- S – конечное пространство состояний;
- A – конечное пространство действий;
- P – функция переходов, определяющая вероятность перейти в состояние s' из состояния s посредством действия a . В равенстве $P(s', s, a) = p(s'|s, a)$ функция переходов равна условной вероятности s' при заданных s и a ;
- R – функция вознаграждения, которая определяет величину вознаграждения за переход из состояния s' в состояние s посредством действия a .

На рис. 3.1 приведен пример МППР. Стрелками представлены переходы между состояниями, в начале каждой стрелки проставлена вероятность перехода по ней, а в середине – вознаграждение за этот переход. Сумма вероятностей всех переходов из любого состояния должна быть равна 1. В данном примере конечное состояние представлено квадратиком (состояние S_5). Для простоты в этом МППР только одно действие.

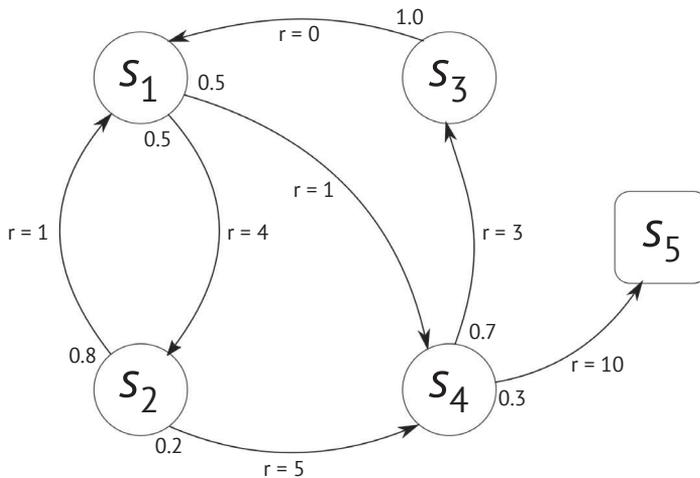


Рис. 3.1 ❖ Пример МППР
с пятью состояниями и одним действием

Этот МППР определяет последовательность дискретных временных шагов, образующих траекторию состояний и действий $(S_0, A_0, S_1, A_1, \dots)$, где состояния подчиняются динамике МППР, т. е. функции переходов $p(s'|s, a)$. Таким образом, функция переходов полностью характеризует динамику окружающей среды.

По определению, функция переходов и функция вознаграждения определяются только текущим состоянием и не зависят от последовательности ранее посещенных состояний. Это так называемое **марковское свойство** означает, что процесс не обладает памятью – будущее состояние зависит только от текущего, но не от истории. Следовательно, состояние содержит всю доступную информацию. Система, обладающая таким свойством, называется **полностью наблюдаемой**.

Во многих приложениях ОП марковское свойство отсутствует, но на практике мы можем обойти эту проблему, предположив, что это все-таки МППР, и используя конечную последовательность предыдущих состояний (конечную

историю): $S_t, S_{t-1}, S_{t-2}, \dots, S_{t-k}$. Такая система называется **частично наблюдаемой**, а состояния называются **наблюдениями**. Мы будем использовать эту стратегию в играх Atari, где входными данными для агента являются строки пикселей. Марковское свойство здесь не выполняется, потому что кадр статический и не несет информации о скорости или направлении движения объектов. Но эту информацию можно получить, рассмотрев три или четыре последовательных кадра (правда, она все равно будет приближенной).

Конечная цель МППР – найти стратегию π , которая доставляет максимум полному вознаграждению $\sum_{t=0}^{\infty} R_{\pi}(s_t, s_{t+1})$, где R_{π} – вознаграждение, полученное на каждом шаге при следовании стратегии π . Считается, что решение МППР найдено, если стратегия выбирает лучшее из возможных действий в каждом состоянии. Такая стратегия называется **оптимальной**.

Стратегия

Стратегия определяет, какое действие предпринять в данной ситуации. Стратегии бывают детерминированными и стохастическими.

Детерминированная стратегия обозначается $a_t = \mu(s_t)$, а стохастическая – $a_t \sim \pi(\cdot|s_t)$, где знак тильды (\sim) обозначает распределение вероятностей. Стохастические стратегии используются, когда имеет смысл рассматривать распределение вероятностей действий, например когда разумно было бы включить в систему действие, приносящее шум.

Стохастические стратегии бывают категориальными и гауссовыми. Первый случай похож на задачу классификации, для выбора действия применяется функция softmax по категориям. Во втором случае действия выбираются из нормального распределения, заданного средним и стандартным отклонением (или дисперсией). Эти параметры могут быть также функциями от состояния.

При использовании параметрических стратегий параметры обозначаются буквой θ . Например, детерминированная стратегия обозначается $\mu_{\theta}(s_t)$.



Все три термина – стратегия, сторона, принимающая решения, и агент – обозначают одно и то же, и в этой книге мы будем употреблять их как синонимы.

Доход

Последовательность состояний и действий МППР ($S_0, A_0, S_1, A_1, \dots$) называется **траекторией**, или **разверткой** (rollout), и обозначается τ . Каждой траектории соответствует последовательность вознаграждений, начисляемых за действия. **Доходом** называется некоторая функция от этих вознаграждений, которая в простейшем случае равна сумме:

$$G(\tau) = r_0 + r_1 + r_2 + \dots = \sum_{t=0}^{\infty} r_t. \quad (3.1)$$

Доход следует рассматривать отдельно для траекторий с конечным и бесконечным горизонтом. Это различие существенно, потому что в случае, когда взаимодействие с окружающей средой никогда не прекращается, приведенная

выше сумма бесконечна. Это нехорошо, поскольку мы не получаем от дохода никакой информации. Такого рода задачи называются непрерывными, и для них доход нужно определять по-другому. Самое лучшее решение – придавать больший вес краткосрочным вознаграждениям и меньший – тем, что случатся в отдаленном будущем. Для этого выбирается значение от 0 до 1, которое называется **коэффициентом обесценивания** и обозначается λ . Тогда доход G определяется следующим образом:

$$G(\tau) = r_0 + \lambda r_1 + \lambda^2 r_2 + \dots = \sum_{t=0}^{\infty} \lambda^t r_t. \quad (3.2)$$

Можно считать, что эта формула отдает предпочтение действиям, которые ближе по времени к текущему моменту, по сравнению с теми, что произойдут еще нескоро. Рассмотрим пример – вы выиграли в лотерею и думаете, когда забрать выигрыш. Надо полагать, что вы предпочли бы сделать это через несколько дней, а не через несколько лет. Коэффициент λ как раз и определяет, сколько времени вы готовы ждать получения выигрыша. Если $\lambda = 1$, значит, вам все равно, когда забрать выигрыш, а если $\lambda = 0$, то вы хотели бы получить его немедленно.

Если траектория имеет конечный горизонт, т. е. заканчивается естественным образом, то задача называется эпизодической (слово «эпизод» употребляется как синоним «траектории»). Для эпизодических задач формула (3.1) работает, но все равно иногда желательно использовать ее вариант с коэффициентом обесценивания:

$$G(\tau) = r_0 + \lambda r_1 + \lambda^2 r_2 + \dots = \sum_{t=0}^k \lambda^t r_t. \quad (3.3)$$

Если горизонт конечный, но длинный, то коэффициент обесценивания повышает устойчивость алгоритма, поскольку отдаленные по времени вознаграждения учитываются лишь частично. На практике коэффициент обесценивания выбирается между 0.9 и 0.999.

Тривиальное, но весьма полезное преобразование формулы (3.3) позволяет выразить доход в момент t через доход в момент $t + 1$:

$$G_t(\tau) = r_t + \lambda G_{t+1}(\tau). \quad (3.4)$$

Упрощая нотацию, получаем

$$G_t = r_t + \lambda G_{t+1}. \quad (3.5)$$

Теперь, воспользовавшись понятием дохода, мы можем определить цель ОП: найти оптимальную стратегию π , которая максимизирует ожидаемый доход, т. е. найти $\operatorname{argmax}_{\pi} E_{\pi}[G(\tau)]$, где $E_{\pi}[\cdot]$ – математическое ожидание случайной величины.

Функции ценности

Доход дает полезное представление о ценности траектории, но ничего не говорит о качестве отдельных посещенных состояний. Такой показатель качества

важен, потому что стратегия могла бы использовать его для выбора следующего наилучшего действия. Стратегия должна была бы просто выбирать действие, которое ведет в состояние с наибольшим показателем качества. Именно это и делает **функция ценности (состояний)**: она оценивает **качество** в терминах ожидаемого дохода при старте из данного состояния и следовании стратегии. Формально функция ценности определяется следующим образом:

$$V_{\pi}(s) = E_{\pi}[G|s_0 = s] = E_{\pi}\left[\sum_{t=0}^k \lambda^t r_t | s_0 = s\right].$$

По аналогии с функцией ценности состояний **функция ценности действий** определяется как ожидаемый доход при старте из данного состояния при условии данного первого действия:

$$Q_{\pi}(s, a) = E_{\pi}[G|s_0 = s, a_0 = a] = E_{\pi}\left[\sum_{t=0}^k \lambda^t r_t | s_0 = s, a_0 = a\right].$$

Функцию ценности и функцию ценности действий называют также **V-функцией** и **Q-функцией** соответственно. Они тесно связаны друг с другом, поскольку функцию ценности можно определить через функцию ценности действий:

$$V_{\pi}(s) = E_{\pi}[Q_{\pi}(s, a)].$$

Зная оптимальную Q^* , можно найти оптимальную функцию ценности:

$$V^*(s) = \max_a Q^*(s, a),$$

поскольку $a^*(s) = \operatorname{argmax}_a Q^*(s, a)$.

Уравнение Беллмана

Функции **V** и **Q** можно оценить, прогнав несколько раз траектории, следующие стратегии π , а затем усреднив полученные значения. Эта техника эффективна и используется во многих контекстах, но она обходится слишком дорого, если учесть, что для вычисления дохода нужно знать вознаграждения на всей траектории.

По счастью, существует уравнение Беллмана, которое определяет функции ценности и ценности действий рекурсивно, что дает возможность оценивать их, зная оценки последующих состояний. Для этого в уравнении Беллмана используется вознаграждение, полученное в текущем состоянии, и ценность следующего за ним состояния. Мы уже видели рекуррентную формулу дохода (3.5) и можем применить ее к ценности состояния:

$$\begin{aligned} V_{\pi}(s) &= E_{\pi}[G_t | s_0 = s] = E_{\pi}[r_t + \gamma G_{t+1} | s_0 = s] \\ &= E_{\pi}[r_t + \gamma V_{\pi}(s_{t+1}) | s_t = s, a_t \sim \pi(s_t)]. \end{aligned} \quad (3.6)$$

Аналогично уравнение Беллмана можно применить к функции ценности действий:

$$\begin{aligned}
 Q_{\pi}(s, a) &= E_{\pi}[G_t | s_t = s, a_t = a] \\
 &= E_{\pi}[r_t + \gamma G_{t+1} | s_t = s, a_t = a] \\
 &= E_{\pi}[r_t + \gamma Q_{\pi}(s_{t+1}, a_{t+1}) | s_t = s, a_t = a].
 \end{aligned}
 \tag{3.7}$$

В формулах (3.6) и (3.7) в обновлении V_{π} и Q_{π} участвуют только ценности последующих состояний, так что нет необходимости раскручивать траекторию до конца, как в исходном определении.

КЛАССИФИКАЦИЯ АЛГОРИТМОВ ОП

Прежде чем приступить к разработке первого алгоритма ОП, который решает уравнение Беллмана, дадим общий, но вместе с тем подробный обзор алгоритмов ОП. Это необходимо, потому что различия между ними могут вызвать путаницу. Определяя, какой алгоритм лучше всего подходит для решения конкретной задачи, нужно учитывать много факторов. Из этого обзора можно будет составить общее представление об ОП, тогда в последующих главах, где алгоритмы подробно рассматриваются с теоретической и практической точек зрения, мы уже будем иметь перед глазами цель и понимать место алгоритма в общей картине.

Первое различие – алгоритмы, основанные на модели, и безмодельные алгоритмы. В первом случае необходима модель окружающей среды, во втором случае без нее можно обойтись. Модель окружающей среды ценна тем, что несет весьма полезную информацию, которую можно использовать при нахождении желаемых стратегий. Однако в большинстве случаев получить такую модель невозможно. Например, смоделировать игру в крестики-нолики легко, но построить модель волн на море очень трудно. Впрочем, безмодельные алгоритмы способны обучаться без всяких предположений о среде. Классификация алгоритмов ОП показана на рис. 3.2.

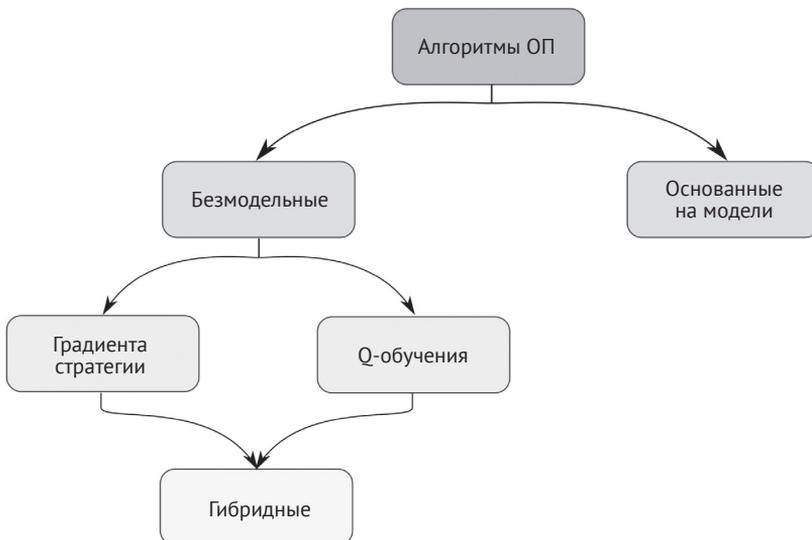


Рис. 3.2 ❖ Классификация алгоритмов ОП

Здесь мы видим различие между основанными на модели и безмодельными алгоритмами, а также два широко известных типа безмодельных алгоритмов: градиента стратегии и на основе ценности. В последующих главах мы увидим, что эти алгоритмы могут комбинироваться.

Безмодельные алгоритмы

Безмодельные алгоритмы прогоняют траектории, следуя заданной стратегии, чтобы приобрести опыт и улучшить агента. В таких алгоритмах имеется три шага, которые повторяются до тех пор, пока не будет найдена хорошая стратегия.

1. Генерация новых примеров путем следования стратегии в окружающей среде. Траектории прогоняются до достижения конечного состояния или на протяжении фиксированного количества шагов.
2. Оценка дохода.
3. Улучшение стратегии с помощью собранных примеров и повторная оценка.

Эти три шага лежат в основе всех алгоритмов такого типа, но конкретный алгоритм зависит от того, как выполняется каждый шаг. Примерами могут служить алгоритмы градиента стратегии и алгоритмы на основе ценности. Выглядят они совершенно по-разному, но основаны на сходных принципах и описанной трехшаговой процедуре.

Алгоритмы на основе ценности

В алгоритмах на основе ценности, или **алгоритмах функции ценности**, используется парадигма, очень похожая на ту, что мы видели в предыдущем разделе. То есть применяется уравнение Беллмана, чтобы обучить Q-функцию, которая, в свою очередь, используется для обучения стратегии. В наиболее часто встречающейся постановке для аппроксимации функции применяется глубокая нейронная сеть и другие хитрости, позволяющие справиться с высокой дисперсией и общей неустойчивостью. С некоторой точки зрения, алгоритмы на основе ценности ближе к алгоритмам регрессии с учителем.

Как правило, это алгоритмы с разделенной стратегией, т. е. не требуется оптимизировать именно ту стратегию, которая использовалась для генерации данных. Это означает, что такие методы могут обучаться на прошлом опыте, поскольку выборочные данные можно хранить в буфере воспроизведения. Способность использовать предыдущие выборки повышает выборочную эффективность функции ценности по сравнению с другими безмодельными алгоритмами.

Алгоритмы градиента стратегии

Другое семейство безмодельных алгоритмов составляют алгоритмы градиента стратегии (или методы оптимизации стратегии). Для них характерна более прямая и очевидная интерпретация проблемы ОП, поскольку они непосредственно обучают параметрическую стратегию путем обновления параметров в направлении улучшения. В основе лежит принцип ОП, согласно которому за хорошие действия следует поощрять (увеличивая градиент стратегии), а за плохие – наказывать.

В отличие от алгоритмов функции ценности, в большинстве алгоритмов оптимизации стратегии требуются данные единой стратегии, что снижает их выборочную эффективность. Методы оптимизации стратегии могут демонстрировать неустойчивость из-за того, что движение в направлении наискорейшего подъема для поверхностей с большой кривизной может завести слишком далеко в данном направлении – в область плохих решений. Чтобы справиться с этой проблемой, было предложено много подходов, например оптимизация стратегии только в доверительной области или оптимизация суррогатной обрзанной функции с целью ограничить изменения стратегии.

Главное преимущество методов градиента стратегии заключается в том, что они применимы к окружающим средам с непрерывной областью действий. Для алгоритмов функций ценности эта проблема очень трудна, поскольку они обучают Q -функцию на дискретных парах состояний и действий.

АЛГОРИТМЫ ИСПОЛНИТЕЛЬ–КРИТИК – это алгоритмы градиента стратегии с единой стратегией, которые одновременно обучают функцию ценности (обычно Q -функцию), называемую критиком, давать отзывы на стратегию исполнителя. Представьте, что вы, исполнитель, собрались поехать в супермаркет новой дорогой. К несчастью, не успели вы добраться до места назначения, как звонит шеф и требует срочно вернуться на работу. Поскольку до супермаркета вы так и не добрались, то не знаете, действительно ли новый маршрут быстрее старого. Но если вы доехали до какого-то знакомого места, то можете оценить время от него до супермаркета и решить, что предпочтительнее: новый маршрут или старый. Именно такую оценку и производит критик. И таким образом удается улучшить исполнителя, даже если он не достиг конечной цели.

Сочетание исполнителя и критика оказалось очень эффективным и часто применяется в алгоритмах градиента стратегии. Эту технику можно комбинировать и с другими идеями в области оптимизации стратегии, в т. ч. с идеей доверительной области.

Гибридные алгоритмы

Достоинства алгоритмов функций ценности и градиента стратегии можно объединить, создав гибридный алгоритм, обладающий более высокой выборочной эффективностью и устойчивостью.

В гибридных алгоритмах Q -функции и градиенты стратегии взаимно дополняют и усиливают друг друга. В них оценивается ожидаемая Q -функция детерминированных действий с целью улучшить стратегию.



Имейте в виду, что поскольку алгоритмы исполнитель–критик обучают и используют функцию ценности, они классифицируются как алгоритмы градиента стратегии, а не как гибридные. Связано это с тем, что целевая функция в них такая же, как в методах градиента стратегии. Функция ценности обновляется, только чтобы получить дополнительную информацию.

Алгоритмы ОП, основанные на модели

Наличие модели окружающей среды означает, что можно предсказать переходы состояний и вознаграждения для каждой пары состояние–действие (не

взаимодействуя с реальной средой). Как уже было сказано, модель известна лишь в немногих случаях, но уж если она известна, то может быть использована разными способами. Самое очевидное применение модели – планирование будущих действий. Речь о том, чтобы спланировать будущие ходы, когда последствия действий уже известны. Например, если мы точно знаем ходы противника, то можем наперед продумать все свои действия, перед тем как выбрать первое из них. Правда, планирование может оказаться весьма накладным и далеко не тривиальным процессом.

Модель можно также обучить на взаимодействиях с окружающей средой путем анализа последствий действий (в терминах состояний и вознаграждений). Это решение не всегда наилучшее, потому что в реальном мире обучение модели может оказаться очень дорогим. Кроме того, если модель усвоит только грубое приближение к окружающей среде, то результаты могут оказаться катастрофическими.

Модель, известную заранее или обученную, можно использовать для планирования и улучшения стратегии и включить в различные фазы алгоритма ОП. Из хорошо известных примеров ОП, основанного на модели, назовем чистое планирование, встроенное планирование для улучшения стратегии и генерации выборки из аппроксимированной модели.

Семейство алгоритмов, в которых модель используется для оценивания функции ценности, называется **динамическим программированием (ДП)**. Мы будем изучать его ниже в этой главе.

Разнообразие алгоритмов

Зачем нужно так много типов алгоритмов ОП? Потому что ни один не является самым лучшим во всех ситуациях. Каждый алгоритм проектируется под конкретные потребности и принимает во внимание различные факторы. Наиболее значимые различия – устойчивость, выборочная эффективность и время обучения. Далее в книге мы постепенно проясним этот вопрос, но, вообще говоря, алгоритмы градиента стратегии более устойчивы и надежны, чем алгоритмы функции ценности. С другой стороны, методы функции ценности обладают лучшей выборочной эффективностью, поскольку это методы с разделенной стратегией и потому могут использовать предшествующий опыт. В свою очередь, алгоритмы, основанные на модели, лучше алгоритмов Q-обучения с точки зрения выборочной эффективности, но гораздо дороже с вычислительной точки зрения и работают медленнее.

Помимо описанных выше, существуют и другие компромиссы, которые следует учитывать при проектировании и развертывании алгоритма (например, простота использования и надежность), так что это вовсе не тривиальные процессы.

ДИНАМИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

ДП – общая алгоритмическая парадигма, смысл которой в том, чтобы разбить задачу на меньшие перекрывающиеся подзадачи, а затем найти решение исходной задачи, объединяя решения подзадач.

ДП находит применение в обучении с подкреплением и является одним из самых простых подходов. Оно подходит для вычисления оптимальной стратегии, поскольку имеется точная модель окружающей среды.

ДП – важная веха в истории алгоритмов ОП, одновременно заложившая фундамент для следующего поколения алгоритмов, но само оно очень дорого обходится с вычислительной точки зрения. ДП может работать только с МППР, имеющим не слишком много состояний и действий, поскольку требуется обновлять ценности всех состояний (или пар состояние–действие), принимая во внимание все остальные возможные состояния. Кроме того, в алгоритмах ДП функция ценности представлена массивом или таблицей. Это эффективный и быстрый способ хранения информации, т. к. никакая информация не теряется, но требуется память для размещения больших таблиц. Поскольку в алгоритмах ДП для хранения функций ценности используются таблицы, они еще называются алгоритмами табличного обучения. Противоположностью является приближенное обучение, в котором используются аппроксимации функций ценности представлениями фиксированного размера, например искусственной нейронной сетью.

В ДП применяется **бутстрэппинг**, т. е. оценка ценности состояния улучшается в результате использования ожидаемой ценности следующих состояний. Как мы уже видели, бутстрэппинг встречается в уравнении Беллмана. И действительно, в ДП уравнения Беллмана (3.6) и (3.7) применяются для оценки V^* и (или) Q^* . Это делается следующим образом:

$$V^*(s) = \max_a E[r_t + \gamma V^*(s_{t+1}) | s_t = s, a_t = a].$$

Или, если воспользоваться Q-функцией,

$$QV^*(s, a) = E[r_t + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1}) | s_t = s, a_t = a].$$

После того как оптимальная функция ценности и функция ценности состояний найдены, можно найти оптимальную стратегию, просто выбирая те действия, которые доставляют максимум математическому ожиданию.

Оценивание и улучшение стратегии

Чтобы найти оптимальную стратегию, нужно сначала найти оптимальную функцию ценности. Это делается с помощью итеративной процедуры **оценивания стратегии** – строится последовательность $\{V_0, \dots, V_k\}$ все более точных приближений к функции ценности для стратегии π , для чего используется функция переходов состояний из модели, математическое ожидание следующего состояния и непосредственное вознаграждение. Точнее, последовательные приближения к функции ценности вычисляются с помощью уравнения Беллмана:

$$\begin{aligned} V_{k+1}(s) &= E_{\pi} [r_t + \gamma V_k(s_{t+1}) | s_t = s] \\ &= \sum_a \pi(s, a) \sum_{s', r} p(s' | s, a) [r + \gamma V_k(s')]. \end{aligned} \quad (3.8)$$

Эта последовательность сходится к оптимальному значению при $k \rightarrow \infty$. На рис. 3.3 показаны обновления $V_{k+1}(s_t)$ с использованием ценностей последующих состояний.

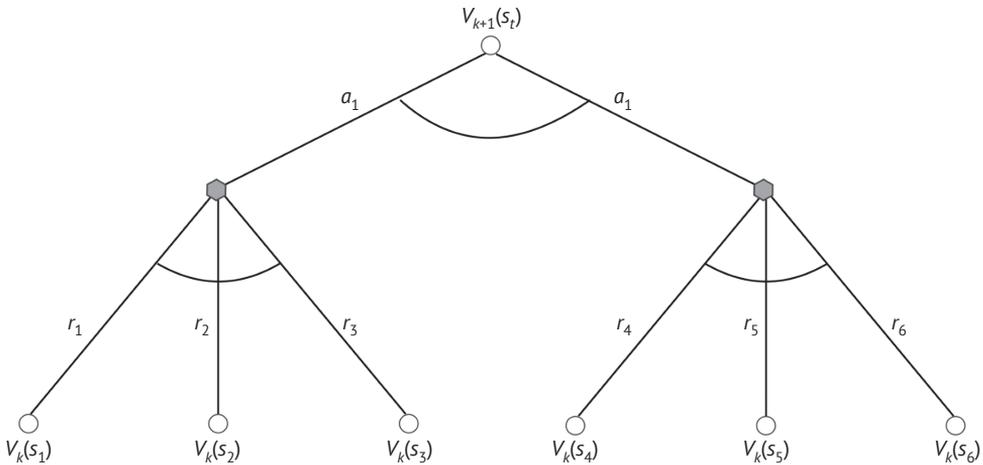


Рис. 3.3 ❖ Обновление по формуле (3.8)

Формула (3.8) обновления функции ценности применима, только если известны функция переходов состояний p и функция вознаграждения r для каждого состояния и действия, т. е. полностью известна модель окружающей среды.

Заметим, что первое суммирование по действиям в формуле (3.8) необходимо для стохастических стратегий, потому что стратегия дает вероятность каждого действия. Для простоты мы, начиная с этого момента, будем рассматривать только детерминированные стратегии.

После того как функция ценности улучшена, ее можно использовать для нахождения улучшенной стратегии. Эта процедура называется **улучшением стратегии** и описывается следующей формулой обновления:

$$\pi' = \operatorname{argmax}_a Q_{\pi}(s, a) = \operatorname{argmax}_a \sum_{s', r} p(s'|s, a)[r + \gamma V_{\pi}(s')]. \quad (3.9)$$

Здесь создается стратегия π' по функции ценности V_{π} исходной стратегии π . Можно формально доказать, что новая стратегия π' всегда лучше π и что стратегия оптимальна тогда и только тогда, когда оптимальна функция ценности. Комбинация оценивания стратегии с ее улучшением дает два алгоритма вычисления оптимальной стратегии: **итерацию по стратегиям** и **итерацию по ценности**. В обоих случаях оценивание стратегии используется для монотонного улучшения функции ценности, а улучшение стратегии – для вычисления новой стратегии. Единственная разница заключается в том, что в алгоритме итерации по стратегиям две эти фазы выполняются циклически, а в алгоритме итерации по ценности – в одном обновлении.

Итерация по стратегиям

Алгоритм итерации по стратегиям в цикле сначала производит оценивание стратегии, т. е. обновляет V_{π} в рамках текущей стратегии π по формуле (3.8), а затем улучшение стратегии по формуле (3.9), т. е. вычисляет π' , пользуясь улучшенной функцией ценности V_{π} .

Ниже приведен псевдокод алгоритма.

инициализировать $V_\pi(s)$ и $\pi(s)$ для каждого состояния s

while π не устойчива:

> оценивание стратегии

while V_π не устойчива:

для каждого состояния s :

$$V_\pi(s) = \sum_{s',r} p(s'|s, \pi(a)) [r + \gamma V_\pi(s')]$$

> улучшение стратегии

для каждого состояния s :

$$\pi = \operatorname{argmax}_a \sum_{s',r} p(s'|s, a) [r + \gamma V_\pi(s')]$$

После инициализации во внешнем цикле производится оценивание и улучшение стратегии, пока стратегия не стабилизируется. На каждой итерации оценивается стратегия, найденная на предыдущем шаге улучшения, на котором, в свою очередь, используется оценка функции ценности.

Применение итерации по стратегиям к игре FrozenLake

Чтобы уложить в сознании идеи, стоящие за алгоритмом итерации по стратегиям, применим его к игре FrozenLake¹. Здесь окружающей средой является сетка 4×4. С помощью четырех действий, соответствующих направлениям (0 – влево, 1 – вниз, 2 – вправо, 3 – вверх), агент должен добраться до противоположной стороны сетки, не упав в полыньи. Ходы не определены жестко, т. е. агент может двигаться в любом направлении. За достижение цели начисляется вознаграждение +1. На рис. 3.4 показана карта игры. S обозначает начальную позицию, звездочка – конечную позицию, а спирали – полыньи.

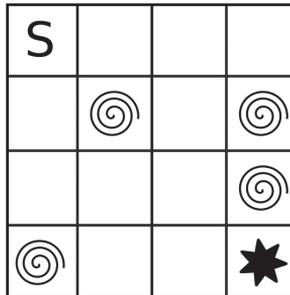


Рис. 3.4 ❖ Карта игры FrozenLake

Посмотрим, как решить эту задачу, располагая всеми необходимыми средствами.



Весь код, приведенный в этой главе, имеется в репозитории этой книги на GitHub по адресу <https://github.com/PacktPublishing/Reinforcement-Learning-Algorithms-with-Python>.

¹ Замерзшее озеро. – Прим. перев.

Сначала создадим окружающую среду, инициализируем функцию ценности и стратегию:

```
env = gym.make('FrozenLake-v0')
env = env.unwrapped
nA = env.action_space.n
nS = env.observation_space.n
V = np.zeros(nS)
policy = np.zeros(nS)
```

Затем нужно создать главный цикл, в котором на каждой итерации выполняется один шаг оценивания стратегии и один шаг улучшения стратегии. Цикл завершается, когда стратегия стабилизируется. Ниже приведен соответствующий код.

```
policy_stable = False
it = 0
while not policy_stable:
    policy_evaluation(V, policy)
    policy_stable = policy_improvement(V, policy)
    it += 1
```

В конце печатаем количество выполненных итераций, функцию ценности, стратегию и счет в нескольких тестовых играх:

```
print('Сошелся после %i итераций по стратегиям'%(it))
run_episodes(env, V, policy)
print(V.reshape((4,4)))
print(policy.reshape((4,4)))
```

Перед тем как определить функцию `policy_evaluation`, напишем функцию, которая вычисляет ожидаемую ценность действия; она понадобится и в функции `policy_improvement`:

```
def eval_state_action(V, s, a, gamma=0.99):
    return np.sum([p * (rew + gamma*V[next_s]) for p, next_s, rew, _ in env.P[s][a]])
```

Здесь `env.P` – словарь, содержащий всю информацию о динамике окружающей среды. `gamma` – коэффициент обесценивания, для простых и умеренно трудных задач берется стандартное значение 0.99; чем выше коэффициент, тем труднее агенту предсказать ценность состояния, поскольку он вынужден заглядывать дальше в будущее.

Затем определим функцию `policy_evaluation`, которая должна производить вычисления по формуле (3.8) для каждого состояния, следуя текущей стратегии, пока результаты не стабилизируются. Поскольку стратегия детерминированная, вычисляем только одно действие:

```
def policy_evaluation(V, policy, eps=0.0001):
    while True:
        delta = 0
        for s in range(nS):
            old_v = V[s]
            V[s] = eval_state_action(V, s, policy[s])
            delta = max(delta, np.abs(old_v - V[s]))
        if delta < eps:
            break
```

Считается, что функция ценности стабилизировалась, если δ меньше пороговой величины ϵ . Как только это условие выполнено, цикл `while` прекращается.

Функция `policy_improvement` принимает функцию ценности и стратегию и перебирает все состояния, обновляя стратегию на основе новой функции ценности:

```
def policy_improvement(V, policy):
    policy_stable = True
    for s in range(nS):
        old_a = policy[s]
        policy[s] = np.argmax([eval_state_action(V, s, a) for a in range(nA)])
        if old_a != policy[s]:
            policy_stable = False
    return policy_stable
```

`policy_improvement(V, policy)` возвращает `False`, если стратегия изменилась. Это означает, что она еще не стабилизировалась.

И напоследок прогоняем несколько игр, чтобы протестировать новую стратегию, и печатаем количество выигрышей.

```
def run_episodes(env, V, policy, num_games=100):
    tot_rew = 0
    state = env.reset()
    for _ in range(num_games):
        done = False
        while not done:
            next_state, reward, done, _ = env.step(policy[state])
            state = next_state
            tot_rew += reward
        if done:
            state = env.reset()
    print('Выиграно %i из %i игр!'%(tot_rew, num_games))
```

Вот и все.

Алгоритм сходится за 7 итераций и выигрывает приблизительно в 85 % игр.

Стратегия, найденная нашей программой, показана слева на рис. 3.5. Как видим, агент делает ходы в странных направлениях, но они продиктованы динамикой окружающей среды. Справа на рис. 3.5 показаны окончательные ценности состояний.

←	↑	↑	↑	0.54	0.5	0.47	0.45
←	⊙	←	⊙	0.56	0	0.36	0
↑	↓	←	⊙	0.59	0.64	0.61	0
⊙	→	↓	★	0	0.74	0.86	★

Рис. 3.5 ❖ Результаты в игре FrozenLake.

Оптимальная стратегия показана слева, а оптимальные ценности состояний – справа

Итерация по ценности

Итерация по ценности – еще один алгоритм динамического программирования для поиска оптимальных ценностей в МППР, но, в отличие от итерации по стратегиям, когда оценивание и улучшение стратегии выполняются в цикле по стратегиям, в алгоритме итерации по ценности оба действия объединены в одно обновление. Точнее, для обновления ценности состояния сразу выбирается наилучшее действие:

$$V_{k+1}(s) = \max_a \sum_{s',r} p(s'|s,a)[r + \gamma V_k(s')]. \quad (3.10)$$

Псевдокод итерации по ценности даже проще, чем итерации по стратегиям:

инициализировать $V(s)$ для каждого состояния s

while V не устойчива:

> итерация по ценности

для каждого состояния s :

$$V(s) = \max_a \sum_{s',r} p(s'|s,a)[r + \gamma V(s')]$$

> вычислить оптимальную стратегию:

$$\pi = \operatorname{argmax}_a \sum_{s',r} p(s'|s,a)[r + \gamma V(s)]$$

Единственное различие – другая формула обновления оценки ценности и отсутствие собственно цикла итерации по стратегиям. Найденная оптимальная стратегия описывается следующей формулой:

$$\pi^* = \operatorname{argmax}_a \sum_{s',r} p(s'|s,a)[r + \gamma V^*(s)]. \quad (3.11)$$

Применение итерации по ценности к игре FrozenLake

Теперь мы можем применить алгоритм итерации по ценности к игре FrozenLake и сравнить оба подхода: сойдутся ли они к одной и той же стратегии и функции ценности.

Как и раньше, определим функцию `eval_state_action` для оценивания ценности пары состояние–действие:

```
def eval_state_action(V, s, a, gamma=0.99):
    return np.sum([p * (rew + gamma*V[next_s]) for p, next_s, rew, _ in
env.P[s][a]])
```

Затем напишем тело алгоритма итерации по ценности:

```
def value_iteration(eps=0.0001):
    V = np.zeros(nS)
    it = 0
    while True:
        delta = 0
        # обновить ценность каждого состояния
        for s in range(nS):
            old_v = V[s]
            # формула (3.10)
```

```

    V[s] = np.max([eval_state_action(V, s, a) for a in range(nA)])
    delta = max(delta, np.abs(old_v - V[s]))
# если стабилизировалась, выйти из цикла
if delta < eps:
    break
else:
    print('Итерация:', it, ' дельта:', np.round(delta,5))
    it += 1
return V

```

Цикл выполняется, пока функция ценности не стабилизируется (т. е. дельта не станет меньше порога `eps`), и на каждой итерации обновляется ценность каждого состояния по формуле (3.10).

Как и в алгоритме итерации по стратегиям, функция `run_episodes` прогоняет несколько игр, чтобы протестировать стратегию. Единственная разница в том, что в данном случае стратегия определяется одновременно с выполнением `run_episodes` (в случае итерации по стратегиям мы заранее определяли действие для каждого состояния).

```

def run_episodes(env, V, num_games=100):
    tot_rew = 0
    state = env.reset()

    for _ in range(num_games):
        done = False

        while not done:
            # выбрать наилучшее действие, пользуясь функцией ценности
            # формула (3.11)
            action = np.argmax([eval_state_action(V, state, a) for a in range(nA)])
            next_state, reward, done, _ = env.step(action)
            state = next_state
            tot_rew += reward
            if done:
                state = env.reset()
        print('Выиграно %i из %i игр!'%(tot_rew, num_games))

```

Наконец, создадим окружающую среду, развернем ее, выполним алгоритм итерации по ценности и прогоним несколько тестовых игр:

```

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

nA = env.action_space.n
nS = env.observation_space.n

V = value_iteration(eps=0.0001)
run_episodes(env, V, 100)
print(V.reshape((4,4)))

```

Будет напечатано что-то вроде:

```

Итерация: 0 дельта: 0.33333
Итерация: 1 дельта: 0.1463
Итерация: 2 дельта: 0.10854
...

```

Итерация: 128 дельта: 0.00011

Итерация: 129 дельта: 0.00011

Итерация: 130 дельта: 0.0001

Выиграно 86 из 100 игр!

```
[[0.54083394 0.49722378 0.46884941 0.45487071]
 [0.55739213 0. 0.35755091 0. ]
 [0.5909355 0.64245898 0.61466487 0. ]
 [0. 0.74129273 0.86262154 0. ]]
```

Алгоритм итерации по ценности сошелся после 130 итераций. Найденные функция ценности и стратегия такие же, как для алгоритма итерации по стратегиям.

РЕЗЮМЕ

Задачу ОП можно формализовать в виде МППР, что дает абстрактный каркас для обучения целенаправленных агентов. МППР определяется как множество состояний, действий, вознаграждений и вероятностей переходов, а решить МППР значит найти стратегию, которая максимизирует ожидаемое вознаграждение в каждом состоянии. Для МППР обязательно выполнение марковского свойства, благодаря которому будущие состояния зависят только от текущего, но не от его истории.

Пользуясь определением МППР, мы сформулировали понятия стратегии, функции дохода, ожидаемого дохода, функции ценности действий и функции ценности. Последние две можно определить в терминах ценностей последующих состояний, а соответствующие уравнения называются уравнениями Беллмана. Эти уравнения полезны, потому что предлагают итеративный способ вычисления функций ценности. После этого оптимальную функцию ценности можно использовать для нахождения оптимальной стратегии.

Алгоритмы ОП бывают основанными на модели и безмодельными. В первом случае требуется модель окружающей среды для планирования следующих действий, тогда как во втором модель не нужна, а алгоритм обучается прямо на взаимодействиях с окружающей средой. Безмодельные алгоритмы можно далее отнести к двум категориям: градиента стратегии и функции ценности. Алгоритмы градиента стратегии обучаются непосредственно по стратегии методом градиентного подъема и обычно относятся к семейству алгоритмов с единой стратегией. Алгоритмы функции ценности чаще являются алгоритмами с разделенной стратегией и для нахождения стратегии обучают функцию ценности действий или функцию ценности. Методы обоих типов можно объединить, получив лучшее из обоих миров.

ДП – первое рассмотренное нами семейство алгоритмов на основе модели. Оно используется, когда известна полная модель окружающей среды, а количество состояний и действий не слишком велико. В алгоритмах ДП применяется бутстрэппинг для оценивания ценности состояния, а для обучения оптимальной стратегии используется два процесса: оценивание и улучшение стратегии. На этапе оценивания стратегии вычисляется функция ценности состояний для произвольной стратегии, а на этапе улучшения стратегия улучшается с помощью функции ценности состояний, найденной в процессе оценивания.

Путем сочетания процессов оценивания и улучшения стратегии можно построить два алгоритма: итерации по стратегиям и итерации по ценности. Основное различие между ними заключается в том, что в первом случае оценивание и улучшение стратегии производятся итеративно, а во втором – оба процесса объединены в одном обновлении.

Хотя ДП страдает от проклятия размерности (сложность экспоненциально возрастает вместе с увеличением числа состояний), идеи оценивания стратегии и итерации по стратегиям лежат в основе почти всех алгоритмов ОП, хотя и в более общей форме.

Еще один недостаток ДП – необходимость иметь точную модель окружающей среды. Это ограничивает его применимость во многих задачах.

В следующей главе мы увидим, как V -функции и Q -функции используются для обучения стратегии в задачах, где модель неизвестна, а выборка производится прямо из окружающей среды.

Вопросы

1. Что такое МППР?
2. Что такое стохастическая стратегия?
3. Как можно определить функцию дохода в терминах дохода на следующем временном шаге?
4. Почему так важно уравнение Беллмана?
5. Какие факторы ограничивают применимость алгоритмов ДП?
6. Что такое оценивание стратегии?
7. В чем различие между итерацией по стратегиям и итерацией по ценности?

Для дальнейшего чтения

- *Саттон Р. С., Барто Э. Дж.* Обучение с подкреплением. Главы 3 и 4.

Часть II

БЕЗМОДЕЛЬНЫЕ АЛГОРИТМЫ ОП

В этой части мы познакомимся с безмодельными алгоритмами ОП, методами, основанными на ценности, и методами градиента стратегии. Заодно мы разработаем ряд современных алгоритмов.

Глава 4

Применение Q-обучения и алгоритма SARSA

Алгоритмы динамического программирования (ДП) эффективны для решения задач обучения с подкреплением (ОП), но опираются на два сильных предположения. Во-первых, должна быть известна модель окружающей среды, а во-вторых, пространство состояний должно быть достаточно мало, чтобы не сказывалось проклятие размерности.

В этой главе мы разработаем класс алгоритмов, для которых первое предположение не нужно. Кроме того, эти алгоритмы не подвержены проклятию размерности. Они обучаются непосредственно на взаимодействии со средой, оценивая функцию ценности на основе большого объема данных о доходе, и, в отличие от ДП, не вычисляют математическое ожидание ценности состояний с помощью модели. В этой новой постановке мы поговорим об опыте как способе обучения функций ценности. Мы рассмотрим, какие проблемы возникают при обучении стратегии, опираясь только на взаимодействия со средой, и какие существуют способы их преодоления. После краткого введения мы познакомимся с обучением на основе **временных различий** (temporal difference – **TD**) – действенным способом обучения оптимальных стратегий на опыте. В TD-обучении заимствованы идеи алгоритмов ДП, но используется только информация, полученная в процессе взаимодействия с окружающей средой. Мы рассмотрим два алгоритма TD-обучения: SARSA и Q-обучение. Они очень похожи и оба гарантируют сходимость в табличном случае, но вместе с тем имеют интересные различия, в которых стоит разобраться. Основным является алгоритм Q-обучения; в сочетании с другими техниками он используется во многих современных алгоритмах ОП, как станет ясно в последующих главах.

Чтобы лучше разобраться в TD-обучении и перейти от теории к практике, мы реализуем алгоритмы Q-обучения и SARSA на примере новой игры, а затем поговорим о различии между ними с точки зрения качества и применения.

В этой главе рассматриваются следующие вопросы:

- обучение без модели;
- TD-обучение;
- SARSA;
- применение SARSA к игре Taxi-v2;
- Q-обучение;
- применение Q-обучения к игре Taxi-v2.

ОБУЧЕНИЕ БЕЗ МОДЕЛИ

По определению, функцией ценности стратегии называется математическое ожидание дохода (т. е. сумма обесцененных доходов) этой стратегии при старте из заданного состояния:

$$V_{\pi}(s) = E_{\pi}[G | s_0 = s].$$

Как было показано в главе 3, алгоритмы ДП обновляют ценности состояний, вычисляя математическое ожидание ценностей всех последующих состояний:

$$V_{k+1}(s) = E_{\pi}[r_t + \gamma V_k(s_{t+1}) | s_t = s] = \sum_a \pi(s, a) \sum_{s', r} p(s' | s, a) [r + \gamma V_k(s')].$$

К сожалению, для вычисления функции ценности необходимо знать вероятности переходов состояний. Для получения этих вероятностей в алгоритмах ДП используется модель окружающей среды. Но возникает вопрос: что делать, если эта модель недоступна. В таком случае лучше всего собрать необходимую информацию путем прямого взаимодействия со средой. Если организовать это правильно, то все будет работать, поскольку многократная выборка из среды позволяет аппроксимировать математическое ожидание и получить хорошую оценку функции ценности.

Порядок действий



Первым делом нужно уточнить, как производится выборка из окружающей среды и как взаимодействовать с ней, чтобы получить пригодную для использования информацию о динамике.

Проще всего выполнять текущую стратегию до конца эпизода. В итоге должна получиться траектория типа изображенной на рис. 4.1. Когда эпизод завершится, можно будет вычислить значения дохода в каждом состоянии путем обратного распространения вверх суммы доходов r_t, \dots, r_{t+n} . Повторив этот процесс несколько раз (т. е. пройдя по нескольким траекториям) для каждого состояния, мы получим несколько значений дохода. Усреднив их, получим ожидаемый доход в каждом состоянии. Вычисленный таким образом ожидаемый доход – это приближенная функция ценности. Чем больше траекторий исследовано, тем больше будет наблюдаемых доходов, и, по закону больших чисел, результат усреднения оценок сходится к математическому ожиданию.

Рис. 4.1 ❖ Траектория, начинающаяся из некоторого состояния

Как и в случае ДП, алгоритмы, которые обучают стратегию на прямом взаимодействии с окружающей средой, опираются на концепции оценивания и улучшения стратегии. Оценивание стратегии заключается в оценивании функции ценности стратегии, а на этапе улучшения полученные таким образом оценки используются для улучшения стратегии.

Оценивание стратегии

Как мы только что видели, использовать реальный опыт для оценивания функции ценности просто. Нужно только следовать стратегии при взаимодействии с окружающей средой, пока не будет достигнуто конечное состояние, а затем вычислить доход и усреднить по выборке доходов, как показано в формуле (4.1):

$$V(s_t) = \frac{1}{N} \sum_{i=0}^N (G_t^i). \quad (4.1)$$

Таким образом, ожидаемый доход в некотором состоянии можно аппроксимировать опытными данными, усреднив по выборочным эпизодам, начинающимся в этом состоянии. Методы, в которых функция ценности оценивается по формуле (4.1), называются **методами Монте-Карло**. Если посещать все пары состояние–действие и прогнать достаточно много траекторий, то методы Монте-Карло гарантируют сходимость к оптимальной стратегии.

Проблема исследования

Как гарантировать, что будут выбраны все действия в каждом состоянии? И почему это так важно? Сначала ответим на второй вопрос, а затем покажем, как можно (по крайней мере, теоретически) организовать исследование окружающей среды таким образом, чтобы посетить каждое возможное состояние.

Зачем исследовать?

Траектории выбираются, следуя стратегии, которая может быть стохастической или детерминированной. В случае детерминированной стратегии при каждой выборке траектории будут посещаться одни и те же состояния, так что во время обновления функции ценности будет учитываться только ограниченное подмножество состояний. Это существенно ограничивает наши знания об окружающей среде. Все равно что учитель, который никогда не меняет своего мнения о предмете, вы так и увязнете в болоте этих идей, не имея возможности узнать о других.

Таким образом, исследование среды необходимо, если мы хотим добиться хороших результатов, оно гарантирует, что мимо нашего внимания не пройдут лучшие стратегии.

С другой стороны, если стратегия устроена так, что только и занята исследованием среды, пренебрегая тем, что уже успела узнать, то найти хорошую стратегию будет очень трудно, пожалуй, даже невозможно. Поиск баланса между исследованием и использованием (т. е. поведением согласно лучшей из найденных к настоящему моменту стратегий) называется дилеммой исследования–использования и подробно рассматривается в главе 12.

Как исследовать

Очень эффективный образ действий в подобных ситуациях называется ϵ -жадным исследованием. Заключается он в том, что агент должен действовать случайно (выбрать произвольное действие) с вероятностью ϵ и жадно (выбрать

лучшее действие) с вероятностью $1 - \varepsilon$. Например, если $\varepsilon = 0.8$, то в среднем из каждых 10 действий восемь выбираются случайно.

Чтобы избежать слишком активного исследования на поздних этапах обучения, когда агент уже уверен в своих знаниях, ε можно уменьшать со временем. Такой подход называется **затуханием** ε . В подобном варианте первоначально стохастическая стратегия постепенно сходится к детерминированной и, хочется надеяться, оптимальной стратегии.

Существует много других способов исследования (например, больцмановское исследование), которые дают более точный результат, но при этом довольно сложны. Для целей этой главы ε -жадное исследование вполне подходит.

TD-ОБУЧЕНИЕ

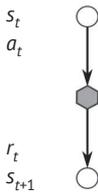


Рис. 4.2 ❖ Обновление в методе одношагового обучения с бутстрэппингом

Методы Монте-Карло – действенный способ обучаться непосредственно на выборке из окружающей среды, но у них есть крупный недостаток – необходимо иметь полную траекторию. Мы должны дойти до конца эпизода и только потом обновлять ценности состояний. Поэтому нужно понять, что произойдет, если траектория бесконечная или просто слишком длинная. Ответ разочаровывает – результаты получатся ужасными. А решение мы уже видели в алгоритмах ДП, где ценности состояний обновляются на

каждом шаге, не дожидаясь конца эпизода. Вместо того чтобы использовать полный доход, накопленный на всей траектории, мы ограничиваемся непосредственным вознаграждением и оценкой ценности следующего состояния. Наглядный пример такого обновления приведен на рис. 4.2, где показана часть траектории, задействованная на одном шаге обучения. Эта техника называется **бутстрэппингом** и полезна не только для длинных, потенциально бесконечных эпизодов, но и для эпизодов любой протяженности. Первая причина заключается в том, что при таком подходе уменьшается дисперсия ожидаемого дохода, поскольку ценности состояний зависят только от следующего непосредственного вознаграждения, а не от всех вознаграждений на траектории. Вторая причина в том, что процесс обучения имеет место на каждом шаге, т. е. такие алгоритмы обучаются в онлайн-режиме. Подобное обучение называется одношаговым. Напротив, методы Монте-Карло офлайн-овые, т. к. в них используется информация, доступная только после завершения эпизода. Методы, обучающиеся в онлайн-режиме с применением бутстрэппинга, называются методами TD-обучения.

TD-обучение можно рассматривать как сочетание методов Монте-Карло с ДП, поскольку в них используется идея выборки, заимствованная у первых, и идея бутстрэппинга, заимствованная у вторых. TD-обучение широко применяется во всех алгоритмах ОП и составляет ядро многих из них. Все представленные в этой главе алгоритмы (точнее, SARSA и Q-обучение) – одношаговые табличные безмодельные (т. е. обходящиеся без модели окружающей среды) TD-методы.

TD-обновление

Из предыдущей главы мы знаем, что

$$V_{\pi}(s) = E_{\pi}[G_t | s_t = s]. \quad (4.2)$$

В обновлении методом Монте-Карло эта величина оценивается эмпирически путем усреднения доходов на нескольких полных траекториях. Разворачивая эту формулу, получаем:

$$\begin{aligned} E_{\pi}[G_t | s_t = s] &= E_{\pi}[r_t + \gamma G_{t+1} | s_t = s] \\ &= E_{\pi}[r_t + \gamma V_{\pi}(s_{t+1}) | s_t = s]. \end{aligned} \quad (4.3)$$

Это уравнение аппроксимируется TD-алгоритмами. Разница в том, что TD-алгоритмы оценивают математическое ожидание, а не вычисляют его. Оценка производится так же, как в методах Монте-Карло, путем усреднения:

$$E_{\pi}[r_t + \gamma V_{\pi}(s_{t+1}) | s_t = s] \approx \frac{1}{N} \sum_{i=0}^N \pi[r_t^i + \gamma V_{\pi}(s_{t+1}^i) | s_t = s].$$

На практике вместо вычисления среднего TD-обновление производится путем улучшения ценности состояния на небольшую величину в направлении оптимального значения:

$$V(s_t) \leftarrow V(s_t) + \alpha[r + \gamma V(s_{t+1}) - V(s_t)]. \quad (4.4)$$

Постоянная α определяет, насколько сильно ценность состояния должна изменяться при каждом обновлении. Если $\alpha = 0$, то ценность состояния не изменится вовсе. Если же $\alpha = 1$, то ценность состояния будет равна $r + \gamma V(s_{t+1})$ (эта величина называется **TD-целью**), так что старая ценность оказывается полностью забыта. На практике нам такие крайности не нужны, и обычно α выбирается в диапазоне от 0.5 до 0.001.

Улучшение стратегии

TD-обучение сходится к оптимальной стратегии, при условии что вероятность выбора каждого состояния больше нуля. Чтобы удовлетворить этому требованию, TD-методы должны исследовать окружающую среду, как было показано в предыдущем разделе. Для исследования можно применить ε -жадную стратегию. Она гарантирует, что будут выбираться как жадные, так и случайные действия, чтобы обеспечить и исследование среды, и использование уже накопленных знаний.

Сравнение методов Монте-Карло и TD-методов

У обоих методов – Монте-Карло и TD – есть важная общая черта: они сходятся к оптимальному решению в табличных задачах (когда ценности состояния хранятся в таблице или массиве), при условии что используется стратегия с исследованием. Однако они различаются способом обновления функции цен-

ности. Вообще говоря, у TD-обучения по сравнению с методами Монте-Карло меньше дисперсия, но больше смещение. Кроме того, TD-методы в общем случае обучаются быстрее и потому предпочтительнее методов Монте-Карло.

SARSA

До сих пор мы представляли TD-обучение как общий способ оценивания функции ценности для заданной стратегии. На практике использовать TD в таком виде невозможно, потому что ему не хватает главной компоненты – улучшения стратегии. И SARSA, и Q-обучение – одношаговые табличные TD-алгоритмы, которые не только оценивают функции ценности, но и оптимизируют стратегию, и это можно использовать в самых разных задачах ОП. В этом разделе мы воспользуемся алгоритмом SARSA для обучения оптимальной стратегии для заданного МППР. Затем мы познакомимся с Q-обучением.

Смысл TD-обучения в том, чтобы оценить ценность состояния. Задумайтесь об этом. Как в данном состоянии выбрать действие с наибольшей ценностью следующего состояния? Выше мы говорили, что нужно выбирать действие, которое переведет агента в состояние с наибольшей ценностью. Однако без модели окружающей среды, которая дает список возможных следующих состояний, мы не можем знать, какое действие переведет агента в такое состояние. Алгоритм SARSA вместо обучения функции ценности обучает и применяет функцию ценности действий Q . Значение $Q(s, a)$ равно ценности состояния s , если выбрано действие a .

Алгоритм

В принципе, все наблюдения, относящиеся к TD-обновлению, остаются справедливы и для SARSA. Применив их к определению Q-функции, получаем обновление в алгоритме SARSA:

$$V_{\pi}(s) = E_{\pi}[G_t | s_t = s]. \quad (4.5)$$

Здесь α – коэффициент, определяющий, насколько сильно изменилась ценность действия, а γ – коэффициент обесценивания в диапазоне от 0 до 1, который позволяет придать меньшую важность решениям, принимаемым в далеком будущем (действия в краткосрочной перспективе важнее, чем в отдаленной). На рис. 4.3 приведена наглядная интерпретация обновления SARSA.

Само название SARSA означает, что обновление основано на состоянии s_t (State), действии a_t (Action), вознаграждении r_t (Reward), следующем состоянии s_{t+1} (State) и, наконец, следующем действии a_{t+1} (Action).

SARSA – алгоритм с единой стратегией, т. е. стратегия, использованная для накопления опыта путем взаимодействия с окружающей средой (поведенческая стратегия), – это именно та стратегия, которая обнов-

s_t
 a_t
 r_t
 s_{t+1}
 a_{t+1}



Рис. 4.3 ❖ Обновление в алгоритме SARSA

ляется. Подобный характер алгоритма связан с тем, что текущая стратегия используется для выбора следующего действия a_{t+1} , необходимого для вычисления оценки $Q(s_{t+1}, a_{t+1})$, и с предположением, что при следующем действии агент будет следовать той же стратегии (т. е. предпримет действие a_{t+1}).

Алгоритмы с единой стратегией обычно проще алгоритмов с разделенной стратегией, но они не так эффективны и, как правило, требуют больше данных для обучения. Несмотря на это, SARSA гарантированно сходится к оптимальной стратегии, если посещает каждое состояние бесконечное число раз и стратегия со временем становится детерминированной. На практике используется ε -жадная стратегия с затуханием, т. е. ε стремится к нулю или близкому к нулю значению. Ниже приведен псевдокод алгоритма SARSA. В нем мы использовали ε -жадную стратегию, но могли бы взять любую обеспечивающую исследование.

инициализировать $Q(s, a)$ для каждой пары состояние–действие
 выбрать $\alpha \in (0, 1]$, $\gamma \in (0, 1]$

для N эпизодов:

$s_t = env_start()$

$a_t = \varepsilon greedy(Q, s_t)$

while s_t не является заключительным состоянием:

$r_t, s_{t+1} = env(a_t)$ # $env()$ совершает один шаг взаимодействия со средой

$a_{t+1} = \varepsilon greedy(Q, s_{t+1})$

$Q(s_t, a_t) \leftarrow (Q, s_{t+1}) + \alpha[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$

$s_t = s_{t+1}$

$a_t = a_{t+1}$

Здесь функция $\varepsilon greedy()$ реализует ε -жадную стратегию. Заметим, что в SARSA выполняется именно то действие, которое было выбрано и использовано на предыдущем шаге для обновления ценности пары состояние–действие.

ПРИМЕНЕНИЕ SARSA К ИГРЕ TAXI-V2

После теоретического введения в TD-обучение и конкретно в алгоритм SARSA мы наконец можем реализовать SARSA и решить интересующую нас задачу. Выше мы уже видели, что SARSA применим к средам с неизвестной моделью и динамикой, но поскольку это табличный алгоритм с ограничениями по масштабируемости, воспользоваться им можно только для окружающей среды с небольшими дискретными пространствами состояний и действий. Поэтому мы выбрали окружающую среду Gym под названием Taxi-v2, которая удовлетворяет всем требованиям и является хорошим испытательным стендом для такого рода алгоритмов.

Игра Taxi-v2 была придумана для изучения иерархического обучения с подкреплением (тип алгоритмов ОП, в которых создается иерархия стратегий и каждая решает некоторую подзадачу). Цель игры – взять пассажира и высадить его точно в указанном месте. За успешную поездку начисляется вознаграждение +20, а за посадку или высадку в неразрешенном месте – штраф –10. Кроме того, на каждом временном шаге теряется один балл. На рис. 4.4 показан пример игры.

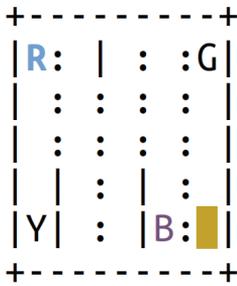


Рис. 4.4 ❖ Начальное состояние окружающей среды Taxi-v2

Существует шесть разрешенных ходов, соответствующих четырем направлениям, посадке и высадке. Символ `:` представляет пустую позицию, символ `|` – стену, сквозь которую такси не может проехать, а буквы `R`, `G`, `Y`, `B` – четыре пункта. Такси, обозначенное желтым прямоугольником, должно посадить пассажира в пункте голубого цвета и высадить в пункте фиолетового цвета.

Реализация довольно проста и следует псевдокоду, показанному в предыдущем разделе. Ниже приведен весь код с пояснениями. Он имеется также в репозитории данной книги на GitHub.

Сначала напишем главную функцию `SARSA(...)`, которая выполняет большую часть работы. Затем можно будет реализовать две вспомогательные функции, решающие простые, но важные задачи.

Алгоритму SARSA нужна окружающая среда и несколько гиперпараметров, передаваемых в качестве аргументов:

- скорость обучения lr , которую мы выше обозначали α и которая определяет объем обучения при каждом обновлении;
- `num_episodes` – количество эпизодов до завершения алгоритма;
- `eps` – начальное значение параметра случайности ϵ -жадной стратегии;
- `gamma` – коэффициент обесценивания, снижающий важность действий в отдаленном будущем;
- `eps_decay` – коэффициент линейного затухания `eps` при переходе от одного эпизода к другому.

Вот первые строки кода:

```
def SARSA(env, lr=0.01, num_episodes=10000, eps=0.3, gamma=0.95, eps_decay=0.00005):
    nA = env.action_space.n
    nS = env.observation_space.n
    test_rewards = []
    Q = np.zeros((nS, nA))
    games_reward = []
```

Здесь инициализируются переменные. `nA` и `nS` – соответственно количество действий и наблюдений окружающей среды. `Q` – матрица, содержащая значения `Q`-функции для каждой пары состояние–действие, а `test_rewards` и `games_rewards` – списки, в которых будет храниться информация о результатах игры.

Далее реализуем главный цикл, в котором обучается `Q`-функция:

```
for ep in range(num_episodes):
    state = env.reset()
    done = False
    tot_rew = 0

    if eps > 0.01:
        eps -= eps_decay

    action = eps_greedy(Q, state, eps)
```

Во второй строчке среда сбрасывается в начальное состояние в каждом эпизоде и сохраняется ее текущее состояние. В третьей строчке инициализируется

булева переменная, которая примет значение True, когда среда окажется в заключительном состоянии. В следующих двух строчках переменная eps обновляется, пока не окажется меньше 0.01. Этот порог позволяет продолжать исследование окружающей среды даже в отдаленной перспективе. В последней строчке выбирается ϵ -жадное действие, исходя из текущего состояния и матрицы Q. Эту функцию мы определим позже.

Позабывшись об инициализации в начале каждого эпизода и выбрав первое действие, мы можем войти в цикл, продолжающийся до завершения эпизода (игры). В следующем коде производится выборка из окружающей среды и обновление Q-функции по формуле (4.5).

```
while not done:
    next_state, rew, done, _ = env.step(action) # один шаг взаимодействия со средой
    next_action = eps_greedy(Q, next_state, eps)
    Q[state][action] = Q[state][action] + lr*(rew +
gamma*Q[next_state][next_action] - Q[state][action]) # формула (4.5)
    state = next_state
    action = next_action
    tot_rew += rew
    if done:
        games_reward.append(tot_rew)
```

В переменной done хранится булево значение, показывающее, продолжает ли еще агент взаимодействовать со средой (строка 2). Таким образом, эпизод продолжается, пока done равно False (строка 1). Затем, как обычно, функция env.step возвращает следующее состояние, вознаграждение, флаг done и информационную строку. В следующей строчке eps_greedy выбирает следующее действие, исходя из next_state и значений в матрице Q. Вся соль алгоритма SARSA содержится в следующей строчке, где производится обновление по формуле (4.5). Помимо скорости обучения и коэффициента обесценивания, в обновлении принимают участие вознаграждение, полученное на последнем выполненном шаге, и значения в массиве Q.

Далее устанавливаются новые значения действия и состояния, полученное вознаграждение прибавляется к суммарному, и если окружающая среда оказалась в заключительном состоянии, то суммарное вознаграждение помещается в список games_reward, и внутренний цикл завершается.

В последних строчках функции SARSA мы через каждые 300 эпох прогоняем 1000 тестовых игр и печатаем информацию: номер эпохи, значение eps и усредненное вознаграждение в тестовых играх. И в самом конце возвращаем массив Q.

```
if (ep % 300) == 0:
    test_rew = run_episodes(env, Q, 1000)
    print("Эпизод:{:5d} Эпс:{:2.4f} Вознагр:{:2.4f}".format(ep, eps, test_rew))
    test_rewards.append(test_rew)
return Q
```

Теперь можно написать функцию eps_greedy, которая выбирает случайное действие из числа допустимых с вероятностью eps. Для этого нужно просто сделать выборку из равномерного распределения от 0 до 1, и если полученное число меньше eps, то выбрать случайное действие, а в противном случае выбрать жадное действие:

```
def eps_greedy(Q, s, eps=0.1):
    if np.random.uniform(0,1) < eps:
        # Выбрать случайное действие
        return np.random.randint(Q.shape[1])
    else:
        # Выбрать жадное действие
        return greedy(Q, s)
```

Для реализации жадной стратегии мы возвращаем индекс максимального значения Q в состоянии s:

```
def greedy(Q, s):
    return np.argmax(Q[s])
```

Осталось только реализовать функцию `run_episodes`, которая прогоняет несколько эпизодов для тестирования стратегии. При выборе действий применяется жадная стратегия, поскольку в процессе тестирования исследование не производится. Эта функция почти не отличается от той, что была написана в предыдущей главе для алгоритмов динамического программирования.

```
def run_episodes(env, Q, num_episodes=100, to_print=False):
    tot_rew = []
    state = env.reset()
    for _ in range(num_episodes):
        done = False
        game_rew = 0
        while not done:
            next_state, rew, done, _ = env.step(greedy(Q, state))
            state = next_state
            game_rew += rew
            if done:
                state = env.reset()
                tot_rew.append(game_rew)
    if to_print:
        print('Средний счет: %.3f из %i игр!'%(np.mean(tot_rew), num_episodes))
    else:
        return np.mean(tot_rew)
```

Отлично!

Почти все сделано. Еще нужно создать и сбросить окружающую среду и вызвать функцию SARSA, передав ей среду и все гиперпараметры:

```
if __name__ == '__main__':
    env = gym.make('Taxi-v2')
    env.reset()
    Q = SARSA(env, lr=.1, num_episodes=5000, eps=0.4, gamma=0.95, eps_decay=0.001)
```

Мы начали со значения $\text{eps} = 0.4$. Это означает, что первые действия будут случайными с вероятностью 0.4, а затем благодаря затуханию эта вероятность будет уменьшаться, пока не достигнет минимума, равного 0.01 (тот самый порог, который был задан в программе раньше).

На рис. 4.5 приведен график полного вознаграждения в тестовых играх. А на рис. 4.6 показан полный эпизод, сыгранный, следуя окончательной стратегии. Читать его следует слева направо и сверху вниз. Мы видим, что такси (сначала желтый прямоугольник, затем зеленый) движется по оптимальному маршруту в обоих направлениях.

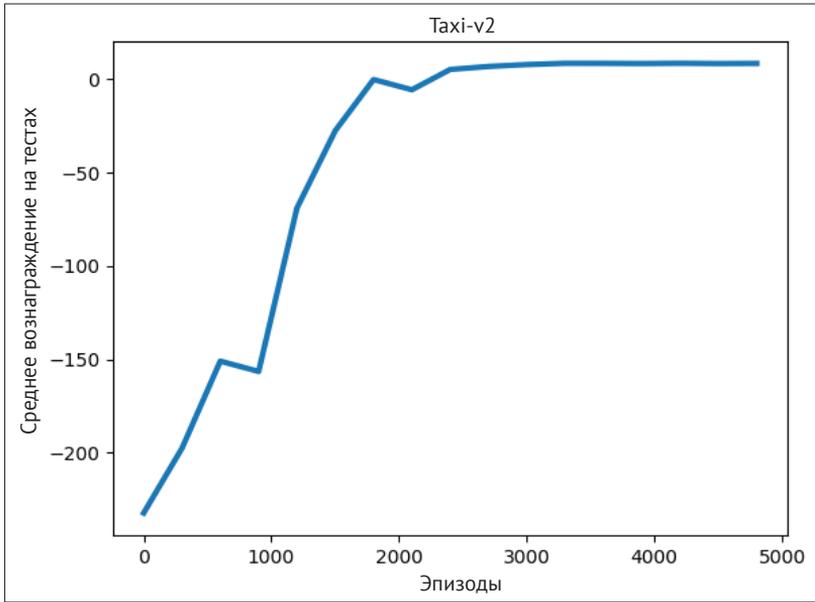


Рис. 4.5 ❖ Результаты применения алгоритма SARSA к игре Taxi-v2

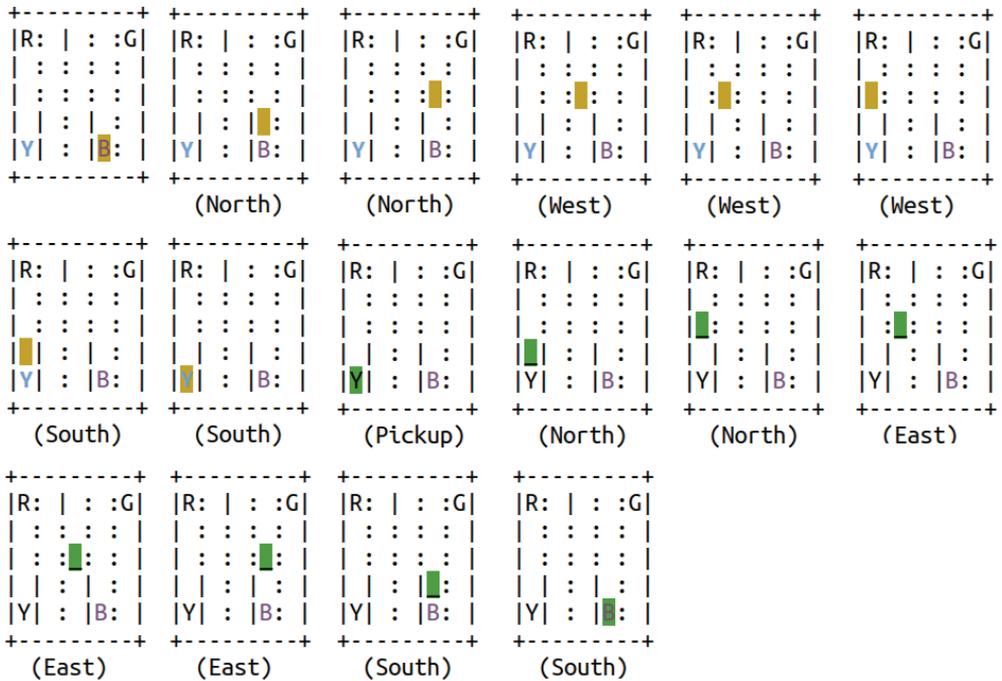


Рис. 4.6 ❖ Один эпизод игры Taxi.

Стратегия основана на значениях матрицы Q , обученной алгоритмом SARSA

Чтобы лучше прочувствовать алгоритм и его гиперпараметры, рекомендуем поэкспериментировать – изменять значения и наблюдать за результатами. Можете также попробовать экспоненциальное затухание вместо линейного. Учитесь методом проб и ошибок – как делают алгоритмы ОП.

Q-ОБУЧЕНИЕ

Q-обучение – еще один TD-алгоритм, имеющий некоторые весьма полезные отличия от SARSA. Этот алгоритм наследует от TD-обучения все характеристики одношагового обучения (т. е. возможность обучаться на каждом шаге) и способность обучаться на опыте, не имея модели окружающей среды.

Q-обучение отличается от SARSA прежде всего тем, что это алгоритм с разделенной стратегией. Напомним, что разделенная стратегия означает, что обновление производится независимо от того, какая стратегия использовалась для накопления опыта, а следовательно, алгоритмы с разделенной стратегией могут использовать прежний опыт для улучшения стратегии. Стратегия, которая применяется для взаимодействия с окружающей средой, называется поведенческой, а улучшаемая стратегия – целевой.

Ниже мы опишем самую простую версию алгоритма, применимую к табличному случаю, но она легко обобщается на аппроксимации функций, в частности на нейронные сети. В следующей главе мы реализуем более сложную версию этого алгоритма, которая способна работать с глубокими нейронными сетями и использовать предшествующий опыт для полного раскрытия потенциала алгоритмов с разделенной стратегией.

Но сначала посмотрим, как работает Q-обучение, формализуем правило обновления и представим псевдокод, объединяющий все компоненты в единое целое.

Теория

Идея Q-обучения заключается в том, чтобы аппроксимировать Q-функцию, используя оптимальную на текущий момент ценность действия. Обновление в Q-обучении очень похоже на обновление в SARSA – с тем отличием, что берется максимум ценности по всем парам состояние–действие:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)], \quad (4.6)$$

где, как всегда, α – скорость обучения, а γ – коэффициент обесценивания.

Если в SARSA обновление применяется к поведенческой стратегии (например, ε -жадной), то в алгоритме Q-обучения обновляется жадная целевая стратегия, основанная на выборе максимальной ценности действия. Если что-то осталось непонятным, взгляните на рис. 4.7. На рис. 4.3, относящемся к SARSA, оба действия a_t и a_{t+1} берутся из одной и той же стратегии, тогда как в Q-обучении действие a_{t+1} выбирается на основе максимальной ценности действия в следующем состоянии.

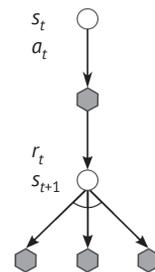


Рис. 4.7 ❖ Обновление в алгоритме Q-обучения

Поскольку обновление в Q-обучении больше не зависит от поведенческой стратегии (которая используется только для выборки из окружающей среды), то алгоритм относится к семейству с разделенной стратегией.

Алгоритм

Поскольку Q-обучение является TD-методом, необходима поведенческая стратегия, которая со временем сходится к детерминированной стратегии. Вполне подходит ε -жадная стратегия с линейным или экспоненциальным затуханием (как и для SARSA).

Напомним, что в алгоритме Q-обучения используются две стратегии:

- целевая стратегия, которая постоянно улучшается;
- поведенческая ε -жадная стратегия для взаимодействия с окружающей средой и ее исследования.

Теперь, наконец, мы можем записать псевдокод алгоритма Q-обучения.

инициализировать $Q(s, a)$ для каждой пары состояние-действие

выбрать $\alpha \in (0, 1]$, $\gamma \in (0, 1]$

для N эпизодов:

$s_t = env_start()$

while s_t не является заключительным состоянием:

$a_t = \varepsilon greedy(Q, s_t)$

$r_t, s_{t+1} = env(a_t)$ # $env()$ совершает один шаг взаимодействия со средой

$a_{t+1} = \varepsilon greedy(Q, s_{t+1})$

$Q(s_t, a_t) \leftarrow (Q, s_t) + \alpha [r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$

$s_t = s_{t+1}$

На практике α обычно выбирается из диапазона между 0.5 и 0.001, а γ – из диапазона между 0.9 и 0.999.

ПРИМЕНЕНИЕ Q-ОБУЧЕНИЯ К ИГРЕ TAXI-V2

Вообще говоря, Q-обучение можно использовать для решения тех же задач, что и SARSA, а поскольку они принадлежат одному и тому же семейству (TD-обучение), то и качество у них похожее. Но для некоторых конкретных задач один подход может оказаться предпочтительнее другого. Поэтому полезно знать, как реализуется Q-обучение.

С этой целью ниже представлено базирующееся на Q-обучении решение игры Taxi-v2 – той же окружающей среды, на которой мы экспериментировали с SARSA. Но помните, что после нескольких мелких изменений этот алгоритм можно использовать и для любой другой среды, обладающей нужными свойствами. Располагая результатами Q-обучения и SARSA для одной и той же окружающей среды, мы получим возможность сравнить их качество.

Чтобы в максимальной степени сохранить согласованность, мы оставили неизменными некоторые функции из реализации SARSA, а именно:

- $\varepsilon greedy(Q, s, \varepsilon)$ – ε -жадная стратегия, которая принимает матрицу Q , состояние s и значение ε , а возвращает действие;

- `greedy(Q, s)` – жадная стратегия, которая принимает матрицу Q и состояние s . Возвращает действие с максимальным значением Q -функции в состоянии s ;
- `run_episodes(env, Q, num_episodes, to_print)` – функция, которая прогоняет `num_episodes` игр для тестирования жадной стратегии, определяемой матрицей Q . Если `to_print` равно `True`, то печатает результаты. В любом случае возвращает среднее вознаграждение.

Реализации этих функций можно посмотреть в разделе «Применение SARSA к игре Taxi-v2» или в репозитории данной книги на GitHub по адресу <https://github.com/PacktPublishing/Reinforcement-Learning-Algorithms-with-Python>.

Главная функция, выполняющая алгоритм Q-обучения, принимает окружающую среду `env`, скорость обучения `lr` (обозначена α в формуле (4.6)), количество эпизодов обучения алгоритма `num_episodes`; начальное значение эпсилон, `eps`, в ϵ -жадной стратегии, скорость затухания `eps_decay` и коэффициент обесценивания `gamma`:

```
def Q_learning(env, lr=0.01, num_episodes=10000, eps=0.3, gamma=0.95,
              eps_decay=0.00005):
    nA = env.action_space.n
    nS = env.observation_space.n

    # Q(s,a) -> строки представляют состояния, а столбцы - действия
    Q = np.zeros((nS, nA))

    games_reward = []
    test_rewards = []
```

Здесь инициализируются переменные – размерности пространства действий и пространства наблюдений; массив Q , содержащий значения Q -функции для каждой пары состояние–действие, и пустые списки, в которых будет храниться информация о результатах игры.

Затем реализуем цикл, повторяющийся `num_episodes` раз:

```
for ep in range(num_episodes):
    state = env.reset()
    done = False
    tot_rew = 0
    if eps > 0.01:
        eps -= eps_decay
```

В начале каждой итерации (эпизода) окружающая среда сбрасывается в начальное состояние, инициализируются переменные `done` и `tot_rew` и линейно уменьшается `eps`.

Далее необходимо выполнить все временные шаги эпизода, поскольку именно на них происходит Q-обучение:

```
while not done:
    action = eps_greedy(Q, state, eps)
    next_state, rew, done, _ = env.step(action) # один шаг взаимодействия со средой

    # вычислить максимальное значение Q для следующего состояния
    Q[state][action] = Q[state][action] + lr*(rew +
gamma*np.max(Q[next_state]) - Q[state][action]) # (4.6)
```

```

state = next_state
tot_rew += rew

if done:
    games_reward.append(tot_rew)
    
```

Это главный цикл алгоритма. Поток управления стандартный:

- 1) выбирается действие, следуя ε -жадной стратегии (поведенческой);
- 2) это действие выполняется в окружающей среде, которая возвращает следующее состояние, вознаграждение и флаг done;
- 3) ценность пары состояние–действие обновляется по формуле (4.6);
- 4) значение next_state присваивается переменной state;
- 5) вознаграждение, начисленное на последнем шаге, прибавляется к полному вознаграждению за эпизод;
- 6) если это был последний шаг, то вознаграждение сохраняется в списке games_reward, и цикл завершается.

В конце через каждые 300 итераций внешнего цикла прогоняется 1000 игр для тестирования агента, печатается полезная информация и возвращается массив Q:

```

if (ep % 300) == 0:
    test_rew = run_episodes(env, Q, 1000)
    print("Эпизод:{:5d} Эпс:{:2.4f} Вознагр:{:2.4f}".format(ep, eps, test_rew))
    test_rewards.append(test_rew)
return Q
    
```

На этом все. И в качестве заключительного аккорда мы в функции main создаем окружающую среду и выполняем алгоритм:

```

if __name__ == '__main__':
    env = gym.make('Taxi-v2')
    Q = Q_learning(env, lr=.1, num_episodes=5000, eps=0.4, gamma=0.95,
eps_decay=0.001)
    
```

Алгоритм стабилизируется после примерно 3000 эпизодов, как видно на рис. 4.8. Эту картинку можно получить, нанеся на график содержимое списка test_rewards.

Как обычно, предлагаем вам настроить гиперпараметры и поэкспериментировать с реализацией, чтобы лучше прочувствовать алгоритм.

Алгоритм нашел стратегию, похожую на ту, что была найдена алгоритмом SARSA. Чтобы найти ее самостоятельно, можете нарисовать несколько эпизодов или распечатать жадную стратегию, определяемую массивом Q.

Сравнение SARSA и Q-обучения

Теперь проведем экспресс-сравнение обоих алгоритмов. На рис. 4.9 построены графики зависимости качества Q-обучения и SARSA в окружающей среде Taxi-v2 от количества эпизодов. Как видим, оба графика сходятся к одному и тому же значению (одной и той же стратегии), и скорость сходимости сравнима. Проводя такие сравнения, следует помнить, что окружающая среда и алгоритмы стохастические и могут давать разные результаты при разных прогонах. На

рис. 4.9 также видно, что в случае Q-обучения график имеет более регулярную форму, поскольку этот алгоритм устойчивее и менее чувствителен к случайным вариациям.

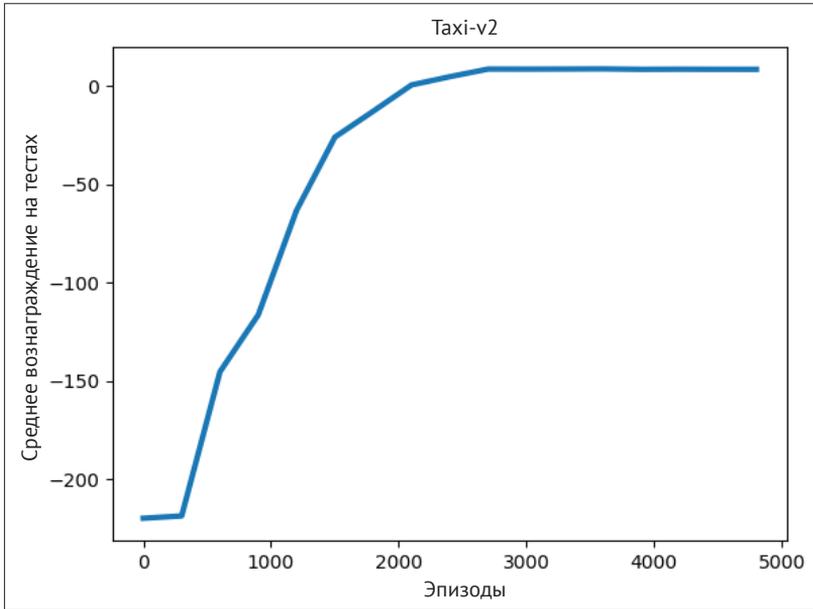


Рис. 4.8 ❖ Результаты применения алгоритма Q-обучения к игре Taxі-v2

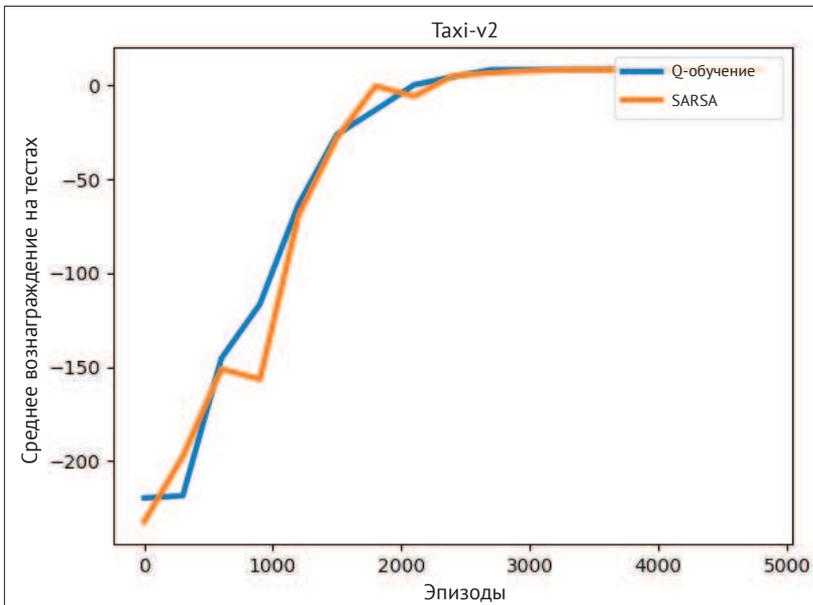


Рис. 4.9 ❖ Сравнение результатов SARSA и Q-обучения в среде Taxі-v2

Так что же, лучше использовать Q-обучение? Вообще говоря, да, и в большинстве случаев Q-обучение превосходит другие алгоритмы, но имеются среды, в которых SARSA работает лучше. Выбор между тем и другим зависит от окружающей среды и задачи.

РЕЗЮМЕ

В этой главе мы познакомились с новым семейством алгоритмов ОП, которые обучаются на опыте, полученном при прямом взаимодействии с окружающей средой. Эти методы отличаются от динамического программирования способностью к обучению функции ценности и, следовательно, стратегии, не опираясь на модель окружающей среды.

Вначале мы видели, что методы Монте-Карло дают простой способ выборки из окружающей среды, однако они должны видеть полную траекторию, перед тем как начать обучение, поэтому ко многим реальным задачам неприменимы. Для преодоления этого недостатка можно объединить методы Монте-Карло с бутстрэппингом, в результате чего получается так называемое обучение на основе временных различий (TD-обучение). Благодаря технике бутстрэппинга эти методы могут обучаться в онлайн-режиме (одношаговое обучение). Они по-прежнему сходятся к оптимальной стратегии, но дисперсия при этом уменьшается. Затем мы описали два одношаговых табличных безмодельных TD-метода: SARSA и Q-обучение. SARSA – алгоритм с единой стратегией, поскольку он обновляет ценность состояния, выбирая действие в соответствии с текущей стратегией (поведенческой). Напротив, Q-обучение – алгоритм с разделенной стратегией, потому что оценивает ценность состояния при следовании жадной стратегии, тогда как для накопления опыта использует другую стратегию. Из-за этого различия алгоритм Q-обучения оказывается намного более устойчивым и эффективным, чем SARSA.

Любой TD-метод должен исследовать окружающую среду в поисках оптимальных стратегий. За исследование среды отвечает поведенческая стратегия, которая иногда должна вести себя нежадно, например как ϵ -жадная стратегия.

Мы реализовали алгоритмы SARSA и Q-обучения и применили их к табличной игре Taxicab. Мы видели, что оба сходятся к оптимальной стратегии и показывают похожие результаты.

В силу своих свойств Q-обучение – ключевой алгоритм ОП. Кроме того, его можно приспособить к решению исключительно сложных игр с пространством состояний очень высокой размерности. Все это возможно благодаря аппроксимации функций, например глубокими нейронными сетями. В следующей главе мы разовьем эту тему и начнем знакомиться с глубокой Q-сетью, которую можно обучить играм Atari непосредственно на пиксельных изображениях.

Вопросы

1. Какое основное свойство метода Монте-Карло используется в ОП?
2. Почему методы Монте-Карло называются офлайнными?
3. Назовите две основные идеи TD-обучения.
4. В чем различие между методами Монте-Карло и TD-методами?
5. Почему для TD-обучения так важно исследование?
6. Почему Q-обучение является алгоритмом с разделенной стратегией?

Глава 5

Глубокая Q-сеть

До сих пор мы изучали и программировали алгоритмы обучения с подкреплением, которые обучают функцию ценности V для каждого состояния или функцию ценности действий Q для каждой пары состояние–действие. В этих методах каждое значение хранится и обновляется в таблице (или массиве). Такие подходы плохо масштабируются на большое число состояний и действий, поскольку размер таблицы растет экспоненциально и имеющейся памяти может не хватить.

В этой главе мы узнаем, как применение аппроксимации функций в алгоритмах обучения с подкреплением позволяет преодолеть эту трудность. Конкретно, мы сосредоточим внимание на глубоких нейронных сетях в применении к Q -обучению. В первой части главы мы объясним, как обобщить Q -обучение, применив для хранения значений функции Q аппроксимацию, и рассмотрим, с какими проблемами можно столкнуться на этом пути. Во второй части мы представим алгоритм **глубокой Q-сети** (Deep Q-network – **DQN**), который включает новые идеи и предлагает элегантное решение некоторых проблем, свойственных наивному варианту Q -обучения с использованием нейронных сетей. Мы увидим, что этот алгоритм позволяет достичь удивительных результатов для широкого круга игр, обучаясь только на пиксельных изображениях. Мы реализуем его и применим к игре Pong, что даст возможность познакомиться с его сильными и слабыми сторонами.

После того как алгоритм DQN был обнародован, другие исследователи предложили ряд вариаций, повышающих его устойчивость и эффективность. Мы кратко рассмотрим и реализуем некоторые из них, чтобы лучше понять, в чем слабость исходного варианта DQN. Мы опишем несколько идей, которые вы сами сможете воплотить в жизнь и улучшить алгоритм.

В этой главе рассматриваются следующие вопросы:

- глубокие нейронные сети и Q -обучение;
- алгоритм DQN;
- применение DQN к игре Pong;
- вариации на тему DQN.

ГЛУБОКИЕ НЕЙРОННЫЕ СЕТИ И Q-ОБУЧЕНИЕ

Как мы видели в главе 4, алгоритм Q -обучения обладает многими качествами, которые позволяют применять его к решению реальных задач. Важнейшее

его свойство – тот факт, что для обучения Q-функции используется уравнение Беллмана, согласно которому значения Q-функции обновляются с учетом ценностей последующих пар состояние–действие. Это дает алгоритму возможность обучаться на каждом шаге, не дожидаясь завершения траектории. Кроме того, ценности состояний или пар состояние–действие хранятся в справочной таблице с простым и эффективным доступом. Так устроенный алгоритм Q-обучения сходится к оптимальной стратегии, при условии что любая пара состояние–действие посещается бесконечное число раз. В этом методе используются две стратегии: нежадная поведенческая стратегия, которая накапливает опыт во взаимодействии с окружающей средой (например, ϵ -жадная), и жадная целевая стратегия, которая выбирает действие с максимальным значением Q-функции.

Поддержание табличного представления ценностей иногда противопоказано, а в некоторых случаях даже вредно. Дело в том, что в большинстве задач количество состояний и действий очень велико. Например, у изображения (даже небольшого) состояний больше, чем атомов во Вселенной. Понятно, что в такой ситуации от таблиц толку нет. Мало того что для таблицы не хватит никакой памяти, так еще лишь малая толика состояний будет посещена более одного раза, из-за чего обучение Q-функции или V-функции сталкивается с огромными трудностями. Поэтому возникает желание как-то обобщить информацию. В данном случае под обобщением понимается тот факт, что нас интересует не только точная ценность состояния $V(s)$, но также ценности похожих и близких состояний. Если некоторое состояние никогда не посещалось, мы могли бы аппроксимировать его ценностью близкого состояния. Идея обобщения исключительно важна во всех разделах машинного обучения, в т. ч. и в обучении с подкреплением.

Идея обобщения весьма плодотворна, если у агента нет полной информации об окружающей среде. В таком случае агент вынужден принимать решения, имея лишь ограниченное представление о среде, на основе **наблюдений**. Например, вообразите себе андроида, который способен только на простые взаимодействия с реальным миром. Очевидно, у него нет полной информации о Вселенной и всех ее атомах. Его восприятие, т. е. наблюдения, ограничено датчиками (например, видеокамерами). Поэтому агент-андроид должен обобщать информацию о происходящем вокруг и вести себя соответственно.

Аппроксимация функций

Обсудив основные ограничения табличных алгоритмов и уяснив, зачем алгоритмам ОП способность к обобщению, обратимся к средствам, которые позволяют снять эти ограничения и подступить к проблеме обобщения.

Пора отказаться от таблиц и представлять функции ценности с помощью аппроксиматора. Аппроксимация функций позволяет представить функцию в ограниченной области определения, располагая памятью фиксированного объема. Объем ресурсов зависит только от аппроксимирующей функции. Как всегда, выбор такой функции зависит от задачи. Примерами аппроксиматоров могут служить линейные функции, решающие деревья, алгоритм ближайших соседей, искусственные нейронные сети и т. д. Как можно ожидать, нейронные

сети предпочтительнее всего остального – не зря они так широко применяются во всех видах алгоритмов ОП. В частности, используются **глубокие нейронные сети (ГНС)**. Своей популярностью они обязаны эффективности и способности самостоятельно выявлять признаки, создавая иерархическое представление, содержащее один или несколько скрытых слоев. К тому же глубокие нейронные сети, точнее **сверточные нейронные сети (СНС)**, на удивление хорошо справляются с изображениями, как показывают недавние прорывные исследования, в особенности в задачах, обучаемых с учителем. Но хотя почти все исследования глубоких нейронных сетей проводились в контексте алгоритмов обучения с учителем, их интеграция с ОП дает весьма интересные результаты. Однако, как мы скоро увидим, это нелегко.

Q-обучение с нейронными сетями

В Q-обучении требуется обучить набор весов глубокой нейронной сети для аппроксимации Q-функции, которая параметризуется (весами нейронной сети) и записывается в виде

$$Q_{\theta}(s, a).$$

Чтобы включить глубокие нейронные сети в Q-обучение (эта комбинация называется глубоким Q-обучением), необходимо выбрать функцию потерь (или целевую функцию), подлежащую минимизации.

Как вы, наверное, помните, в табличном случае обновление в алгоритме Q-обучения выглядит следующим образом:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)].$$

Здесь s' – состояние на следующем шаге. Обновление в онлайнном режиме применяется к каждой выборке, произведенной поведенческой стратегией.

! В отличие от предыдущих глав, здесь мы для упрощения нотации обозначаем s, a состояние и действие на текущем шаге, а s', a' – на следующем шаге.

В случае нейронной сети нашей целью является оптимизация весов θ , так что Q_{θ} напоминает оптимальную Q-функцию. Но поскольку оптимальная Q-функция нам неизвестна, то мы можем лишь приближаться к ней небольшими шагами, минимизируя на каждом шаге беллмановскую ошибку $r + \gamma \max_{a'} Q(s', a') - Q(s, a)$. Этот шаг похож на то, что мы делали в табличном алгоритме Q-обучения. Однако в глубоким Q-обучении мы обновляем не единственное значение $Q(s, a)$, а берем градиент Q-функции по параметрам θ :

$$\theta \leftarrow \theta - \alpha[r + \gamma \max_{a'} Q_{\theta}(s', a') - Q_{\theta}(s, a)] \nabla_{\theta} Q_{\theta}(s, a). \quad (5.1)$$

Здесь $\nabla_{\theta} Q_{\theta}(s, a)$ – вектор частных производных Q по θ , а α – размер шага в направлении градиента – называется скоростью обучения.

На практике описанный плавный переход от табличного Q-обучения к глубокому не дает хорошей аппроксимации. Первое исправление – использовать в качестве функции потерь не беллмановскую ошибку, а **среднеквадратичскую ошибку (СКО)**. Второе исправление – перейти от онлайнного Q-обуче-

ния к пакетному. Это означает, что параметры нейронной сети обновляются после получения множественной выборки (например, мини-пакета размера больше 1 в случае обучения с учителем). В результате этих изменений мы получаем такую функцию потерь:

$$L(\theta) = E_{(s,a,r,s')}[(y_i - Q_\theta(s_i, a_i))^2]. \quad (5.2)$$

Здесь y – не настоящая функция ценности действий, которую мы вообще не используем, а целевое значение:

$$y_i = r_i + \gamma \max_{a'_i} Q_\theta(s'_i, a'_i). \quad (5.3)$$

Затем вектор параметров сети θ обновляется методом градиентного спуска по функции потерь СКО:

$$\theta = \theta - \alpha \nabla_\theta L(\theta).$$

Важно отметить, что y_i рассматриваются как константы и что градиент функции потерь не распространяется дальше.

! В предыдущей главе мы познакомились с методами Монте-Карло, поэтому сейчас хотим отметить, что их тоже можно адаптировать для работы с нейронными сетями. В этом случае в качестве y_i выступает доход G . Поскольку обновление в методах Монте-Карло несмещенное, асимптотически они лучше, чем TD, но на практике лучшие результаты дает TD-обучение.

Неустойчивость глубокого Q-обучения

Имея только что описанные функцию потерь и метод обновления, уже можно было бы разработать алгоритм глубокого Q-обучения. Увы, действительность гораздо сложнее. Если бы мы попытались реализовать такой алгоритм, то он, скорее всего, не заработал бы. Почему? Включив в алгоритм нейронные сети, мы уже не можем гарантировать улучшения. Табличное Q-обучение обладает свойствами сходимости, но про нейронную сеть этого уже не скажешь.

В книге Саттона и Барто «Обучение с подкреплением»¹ описана проблема, названная «смертельной триадой», которая возникает, когда сочетаются следующие три фактора:

- аппроксимация функции;
- бутстрэппинг (т. е. обновление, используемое в других оценках);
- обучение с разделенной стратегией (Q-обучение является алгоритмом с разделенной стратегией, поскольку обновление не зависит от стратегии, использованной для выборки).

Но это в точности основные ингредиенты алгоритма глубокого Q-обучения. Как отмечали авторы, от бутстрэппинга мы не можем отказаться, поскольку это резко увеличило бы вычислительную стоимость или эффективность по данным. Разделенная стратегия также важна для создания более интеллектуальных и мощных агентов. Ну и понятно, что без глубоких нейронных сетей

¹ Саттон Р. С., Барто Э. Дж. Обучение с подкреплением. М.: ДМК Пресс, 2020.

мы потеряем критически важный компонент. Поэтому необходимо проектировать алгоритмы так, чтобы сохранить все три компонента, но одновременно сгладить последствия смертельной триады.

Глядя на уравнения (5.2) и (5.3), может показаться, что проблема похожа на регрессию с учителем, но это не так. В обучении с учителем при выполнении стохастического градиентного спуска (СГС) мини-пакеты всегда случайно выбираются из набора данных, так чтобы гарантировать **независимость и одинаковое распределение**. В ОП опыт накапливается путем следования стратегии. А поскольку состояния последовательны и тесно связаны друг с другом, предполагать независимость и одинаковое распределение уже невозможно, в результате чего СГС теряет устойчивость.

Еще одна причина неустойчивости – нестационарность процесса Q-обучения. Из уравнений (5.2) и (5.3) видно, что целевые значения у вычисляет та самая нейронная сеть, которая обновляется. Это опасно, учитывая, что целевые значения будут также обновляться во время обучения. Это все равно что стрелять по движущейся мишени, не принимая в расчет ее перемещения. Такое поведение обусловлено исключительно способностью нейронной сети к обобщению, в табличном случае оно не составляет проблемы.

У глубокого Q-обучения пока нет убедительного теоретического обоснования, но, как мы увидим, существует алгоритм, который с помощью нескольких технических приемов улучшает свойство независимости и одинакового распределения данных и в какой-то мере сглаживает проблему движущейся мишени. Благодаря этим приемам алгоритм становится более гибким и устойчивым.

DQN

Алгоритм DQN, впервые изложенный в статье «Human-level control through deep reinforcement learning», написанной Мнихом и его коллегами из компании DeepMind, – первый масштабируемый алгоритм обучения с подкреплением, сочетающий Q-обучение с глубокими нейронными сетями. Для преодоления проблемы неустойчивости в DQN применяются две новаторские техники.

DQN оказался первым искусственным агентом, способным к обучению на самых разных трудных задачах. Более того, он обучился решению многих задач, получая на входе только строки пикселей высокой размерности и используя от начала до конца методы ОП.

Решение

Основные новации, привнесенные алгоритмом DQN, – это **буфер воспроизведения**, призванный обойти проблему корреляции данных, и отдельная *целевая сеть*, решающая проблему нестационарности.

Буфер воспроизведения

Чтобы данные на итерациях СГС были в большей степени независимыми и одинаково распределенными, в DQN был включен большой буфер воспроиз-

ведения (называемый также *воспроизведением опыта*), в котором хранится накопленный опыт. В идеале этот буфер должен содержать все переходы, имевшие место на протяжении жизни агента. При выполнении СГС случайный мини-пакет выбирается из буфера воспроизведения и используется в процедуре оптимизации. Поскольку буфер содержит разнообразный опыт, выбранный из него мини-пакет будет достаточно разнообразным, чтобы обеспечить независимость выборки. Еще одно важное свойство буфера воспроизведения – возможность повторно использовать данные, т. к. данные о переходах отбираются несколько раз. Это существенно повышает эффективность алгоритма по данным.

Целевая сеть

Проблема движущейся мишени возникает из-за того, что сеть постоянно обновляется в процессе обучения, которое модифицирует также целевые значения. Но нейронная сеть обязана обновляться, чтобы найти наилучшие ценности пар состояние–действие. Решение, примененное в DQN, – использовать две нейронные сети. Одна сеть называется *онлайновой*, она постоянно обновляется, другая – *целевой*, она обновляется через каждые N итераций (N обычно выбирается в диапазоне от 1000 до 10 000). Онлайновая сеть взаимодействует с окружающей средой, а целевая служит для предсказания целевых значений. Таким образом, на протяжении N итераций целевые значения, порождаемые целевой сетью, остаются неизменными, что препятствует распространению неустойчивости и уменьшает риск расхождения. Потенциальный недостаток заключается в том, что целевая сеть – устаревшая версия онлайновой. Тем не менее на практике преимущества часто перевешивают недостатки, и устойчивость алгоритма заметно повышается.

Алгоритм DQN

Включение буфера воспроизведения и отдельной целевой сети в алгоритм глубокого Q-обучения открыло возможность обучиться играм Atari (Space Invaders, Pong, Breakout и другим), располагая лишь изображениями, вознаграждением и сигналом завершения. DQN обучается от начала до конца с помощью комбинации сверточной и полносвязной нейронных сетей.

При обучении DQN игре в каждую из 49 игр Atari использовались одни и те же алгоритм, архитектура сети и гиперпараметры. Он оказался лучше всех предыдущих алгоритмов и на многих играх не уступал или даже превосходил профессиональных геймеров. Игры Atari далеко не тривиальны, и во многих необходимо сложное планирование. Надо признать, что в нескольких играх (например, в хорошо известной игре Montezuma's Revenge) даже DQN не смог достичь высокого уровня.

Особенность этих игр состоит в частичной наблюдаемости, поскольку агент видит только изображения. Полное состояние окружающей среды остается неизвестным. На самом деле одного изображения недостаточно, чтобы в полной мере оценить текущую ситуацию. Например, как узнать, в каком направлении движется мяч на рис. 5.1?

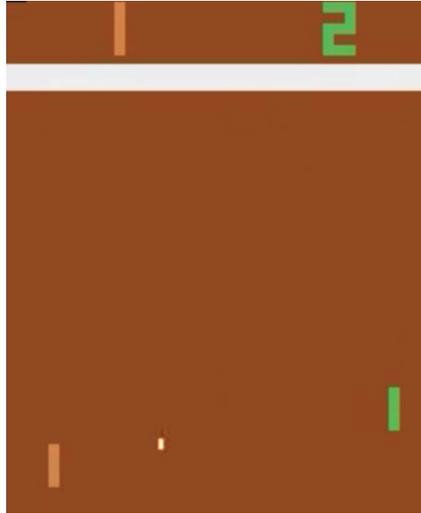


Рис. 5.1 ❖ Кадр из игры Pong

Никак – и агент тоже не знает. Чтобы разрешить эту проблему, в каждый момент времени рассматривается последовательность предыдущих наблюдений. Обычно хватает от двух до пяти последних кадров, в большинстве случаев они дают достаточно точное приближение к истинному состоянию.

Функция потерь

В процессе обучения глубокой Q-сети требуется минимизировать функцию потерь (5.2), но с использованием отдельной целевой сети \hat{Q} с весами θ' , так что функция потерь принимает вид:

$$L(\theta) = E_{(s,a,r,s')}[(r + \gamma \max_{a'} \hat{Q}_{\theta'}(s', a') - Q_{\theta}(s, a))^2]. \quad (5.4)$$

Здесь θ – вектор параметров онлайн-сети.

Оптимизация дифференцируемой функции потерь (5.4) производится нашим любимым итеративным методом – мини-пакетным градиентным спуском. Это значит, что обучающее обновление применяется к мини-пакетам, равномерно выбираемым из буфера воспроизведения. Градиент функции потерь равен

$$\nabla_{\theta} L(\theta) = E_{(s,a,r,s')}[(r + \gamma \max_{a'} \hat{Q}_{\theta'}(s', a') - Q_{\theta}(s, a)) \nabla_{\theta} Q_{\theta}(s, a)]. \quad (5.5)$$

В отличие от проблемы, свойственной глубокому Q-обучению, в алгоритме DQN процесс обучения более устойчивый. А поскольку данные в большей степени независимы и одинаково распределены, а целевая сеть (относительно) фиксирована, то алгоритм больше напоминает задачу регрессии. С другой стороны, цели по-прежнему зависят от весов сети.



Если оптимизировать функцию потерь (5.4) на каждом шаге и только на одной выборке, то получится алгоритм Q-обучения с аппроксимацией функции.

Псевдокод

Итак, мы рассказали обо всех компонентах DQN и теперь можем собрать все вместе и продемонстрировать псевдокод алгоритма, чтобы устранить все неясности (если что-то останется непонятным, не переживайте – в следующем разделе мы напишем настоящий код, и все станет на свои места).

Алгоритм DQN состоит из трех основных частей:

- сбор и сохранение данных. При сборе данных используется поведенческая стратегия (например, ε -жадная);
- оптимизация нейронной сети (выполнение СГС на мини-пакетах, выбираемых из буфера);
- обновление целевой сети.

Ниже приведен псевдокод алгоритма DQN.

Инициализировать функцию Q со случайными весами θ

Инициализировать функцию \hat{Q} со случайными весами $\theta' = \theta$

Инициализировать пустой буфер воспроизведения D

for episode = 1.. M **do**

Инициализировать окружающую среду $s \leftarrow env.reset()$

for t = 1.. T **do**

> собрать наблюдения env:

$a \leftarrow \varepsilon greedy(\phi(s))$

$s', r, d \leftarrow env(a)$

> сохранить переход в буфере воспроизведения:

$\varphi \leftarrow \phi(s), \varphi' \leftarrow \phi(s')$

$D \leftarrow D \cup (\varphi, a, r, \varphi', d)$

> обновить модель по формуле (5.4):

Выбрать случайный мини-пакет $(\varphi_j, a_j, r_j, \varphi'_j, d_j)$ из D

$$y_j = \begin{cases} r_j & \text{если } d_{j+1} = True \\ r_j + \gamma \max_{a'} \hat{Q}_{\theta'}(\phi_{j+1}, a') & \text{в противном случае} \end{cases}$$

Выполнить один шаг спуска в направлении градиента $(y_j - Q_{\theta}(\varphi_j, a_j))^2$ по θ

> Обновить целевую сеть:

Через каждые C шагов $\theta' \leftarrow \theta$ (т. е. $\hat{Q} \leftarrow Q$)

$s \leftarrow s'$

end for

end for

Здесь d – возвращаемый средой флаг, который говорит, достигла ли среда заключительного состояния. Если $d=True$, то эпизод закончился и следует восстановить начальное состояние среды. ϕ – шаг предварительной обработки, на котором понижается размерность изображений (они преобразуются в полутонные, и уменьшается размер) и последние n кадров включаются в текущую группу. Обычно берется значение n от 2 до 4. Этап предварительной обработки будет подробнее описан в следующем разделе, когда мы приступим к реализации DQN.

В DQN в динамическом буфере воспроизведения D хранится ограниченное число кадров. В оригинальной статье предложен буфер на миллион переходов, а когда он заполняется, самые старые переходы удаляются.

Все остальные части мы уже описали. Если вам интересно, почему целевое значение y_j принимает значение r_j , если $d_{j+1} = True$, объясняем: потому что после этого взаимодействия с окружающей средой не будет, так что это действительно несмещенное значение Q-функции.

Архитектура модели

До сих пор мы говорили о самом алгоритме, но не объяснили архитектуру DQN. Помимо новых идей, призванных повысить устойчивость обучения, качество алгоритма сильно зависит от архитектуры модели DQN. В оригинальной статье для всех сред Atari использовалась одна и та же архитектура, сочетающая СНС и ПНС (полносвязная нейронная сеть). Поступающие на вход изображения пропускаются через СНС, которая строит карты признаков. СНС сплошь и рядом применяются для обработки изображений, поскольку обладают инвариантностью относительно параллельных переносов и свойством разделения весов. Это означает, что для обучения сети нужно меньше весов по сравнению с глубокими нейронными сетями других типов.

На выходе модели получаются ценности пар состояние–действие, по одной для каждого действия. Таким образом, для управления агентом с пятью возможными действиями модель выведет по одному значению для каждого из них. Такая архитектура позволяет вычислить все значения Q-функции за один проход.

В СНС имеется три сверточных слоя. Каждый последующий слой включает операцию свертки с большим числом фильтров меньшей размерности, а также нелинейную функцию. Последний скрытый слой СНС полносвязный, за ним следует ректифицированная функция активации и еще один полносвязный слой, порождающий один выход для каждого действия. На рис. 5.2 схематично изображена эта архитектура.

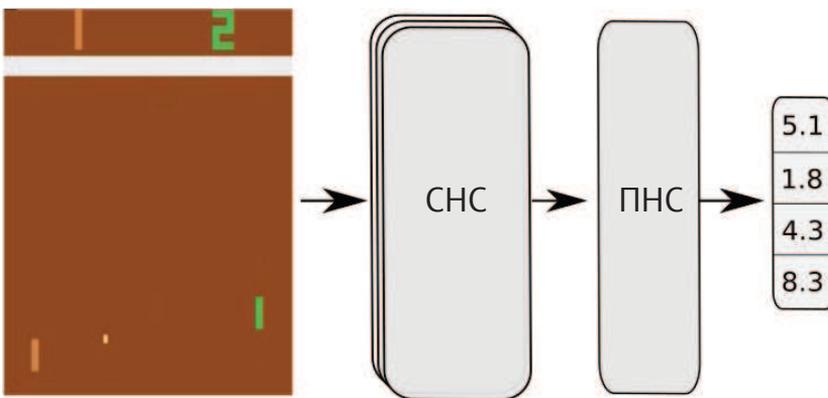


Рис. 5.2 ❖ Архитектура глубокой нейронной сети для алгоритма DQN – композиция СНС и ПНС

ПРИМЕНЕНИЕ DQN К ИГРЕ PONG

Вооружившись техническими знаниями о Q-обучении, глубоких нейронных сетях и алгоритме DQN, мы можем все это применить в деле и загрузить работой GPU. В этом разделе мы применим DQN к окружающей среде Pong из числа игр для Atari. Мы выбрали именно среду Pong, потому что она проще решается и обучение в ней занимает меньше времени, ресурсов процессора и памяти. Но если ваша машина оснащена достаточно мощным GPU, то можете применить точно такую же конфигурацию ко всем остальным играм Atari (в некоторых случаях может понадобиться небольшая настройка). По той же причине мы несколько облегчили оригинальную конфигурацию как с точки зрения мощности аппроксиматора (уменьшили количество весов), так и гиперпараметров (меньший размер буфера). Это не влияет на результаты для Pong, но может снизить качество алгоритма в других играх.

Игры Atari

Игры Atari стали стандартным испытательным стендом для алгоритмов глубокого ОП с момента появления статьи о DQN. Сначала они вошли в состав среды разработки **Arcade Learning Environment (ALE)**, а затем были обернуты стандартным интерфейсом OpenAI Gym. ALE (и Gym) включает 57 наиболее популярных видеоигр для компьютера Atari 2600, в том числе Montezuma's Revenge, Pong, Breakout и Space Invaders, показанные на рис. 5.3. Эти игры широко используются в исследованиях по ОП благодаря многомерному пространству состояний (210×160 пикселей) и разнообразию задач.



Рис. 5.3 ❖ Игры Montezuma's Revenge, Pong, Breakout и Space Invaders

Важным свойством окружающих сред Atari является их детерминированность. Это означает, что одинаковые действия всегда приводят к одинаковому результату. С точки зрения алгоритма, эта детерминированность имеет место, пока не будет использована вся история для выбора действия из стохастической стратегии.

Предварительная обработка

Кадры в играх Atari – цветные RGB-изображения размера 210×160 пикселей, так что полная размерность пространства состояний равна $210 \times 160 \times 3$. Если использовать группы из четырех соседних кадров, размерность входа составит $210 \times 160 \times 12$. Обработка данных такой размерности требует очень много вычислительных ресурсов, да и хранить много кадров в буфере воспроизведения было бы сложно. Поэтому необходима предварительная обработка для понижения размерности. В оригинальной реализации DQN применялся такой конвейер предобработки:

- RGB-изображение преобразовывалось в полутоновое;
- размер изображения уменьшался до 110×84 , а затем изображение обрезалось до размера 84×84 ;
- к текущему кадру присоединялись предыдущие три или четыре кадра;
- кадры подвергались нормализации.

Кроме того, поскольку частота кадров в игре была высокой, применялась техника пропуска последовательных кадров. Это позволяет агенту хранить и использовать для обучения меньше кадров в каждой игре, не сильно жертвуя качеством алгоритма. На практике агент выбирает действие после каждых k кадров и повторяет его для пропущенных кадров.

Кроме того, в некоторых средах агент должен нажать кнопку **Пуск**, чтобы начать игру. И еще – из-за детерминированности окружающей среды после сброса среды выполнялось несколько пустых операций, чтобы агент начинал игру в случайной позиции.

К счастью для нас, OpenAI выпустила реализацию конвейера предобработки, совместимую с интерфейсом Gym. Ее можно найти в репозитории этой книги на GitHub в файле `atari_wrappers.py`. Здесь мы приведем лишь краткое описание этой реализации:

- `NoopResetEnv(n)`: совершает n пустых операций после сброса среды, чтобы агент начинал игру в случайной позиции;
- `FireResetEnv()`: нажимает кнопку **Пуск** после сброса среды (необходимо не во всех играх);
- `MaxAndSkipEnv(skip)`: пропускает `skip` кадров, не забывая повторять действия и суммировать вознаграждения;
- `WarpFrame()`: уменьшает кадр до размера 84×84 и преобразует цветное изображение в полутоновое;
- `FrameStack(k)`: объединяет последние k кадров в группу.

Все эти функции реализованы как обертки, т. е. дают простой способ преобразовать окружающую среду, надстроив над ней новый слой. Например, чтобы масштабировать кадры в среде Pong, нужно было бы написать:

```
env = gym.make('Pong-v0')
env = ScaledFloatFrame(env)
```

Обертка должна наследовать классу `gym.Wrapper` и переопределять по крайней мере один из следующих методов: `__init__(self, env)`, `step`, `reset`, `render`, `close` или `seed`.

Мы не станем приводить реализации всех вышеперечисленных оберток, поскольку это выходит за рамки книги, но покажем, как выглядят обертки `FireResetEnv` и `WarpFrame`, чтобы вы могли составить представление о том, как они пишутся. Полный код имеется в репозитории книги на GitHub.

```
class FireResetEnv(gym.Wrapper):
    def __init__(self, env):
        """Предпринять действие после сброса среды, остающейся замороженной
        до нажатия кнопки Пуск."""
        gym.Wrapper.__init__(self, env)
        assert env.unwrapped.get_action_meanings()[1] == 'FIRE'
        assert len(env.unwrapped.get_action_meanings()) >= 3

    def reset(self, **kwargs):
        self.env.reset(**kwargs)
        obs, _, done, _ = self.env.step(1)
        if done:
            self.env.reset(**kwargs)
        obs, _, done, _ = self.env.step(2)
        if done:
            self.env.reset(**kwargs)
        return obs

    def step(self, ac):
        return self.env.step(ac)
```

Прежде всего `FireResetEnv` наследует классу `Wrapper` из `Gym`. Затем на этапе инициализации проверяется наличие действия **Пуск**, для чего среда разворачивается методом `env.unwrapped`. Переопределенный метод `reset` сначала вызывает метод `self.env.reset`, определенный в предыдущем слое, а затем выполняет действие **Пуск**, вызывая `self.env.step(1)` и зависящее от среды действие `self.env.step(2)`.

Определение `WarpFrame` аналогично:

```
class WarpFrame(gym.ObservationWrapper):
    def __init__(self, env):
        """Уменьшить размер кадра до 84x84, как описано в статье в Nature и более
        поздних работах."""
        gym.ObservationWrapper.__init__(self, env)
        self.width = 84
        self.height = 84
        self.observation_space = spaces.Box(low=0, high=255,
            shape=(self.height, self.width, 1), dtype=np.uint8)

    def observation(self, frame):
        frame = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)
        frame = cv2.resize(frame, (self.width, self.height),
            interpolation=cv2.INTER_AREA)
        return frame[:, :, None]
```

Класс `WarpFrame` наследует классу `gym.ObservationWrapper` и создает пространство `Box` со значениями от 0 до 255 и размером 84×84. Метод `observation()` преобразует цветное RGB-изображение в полутоновое и изменяет его форму и размер.

Теперь мы можем написать функцию `make_env`, которая применяет к среде все обертки:

```
def make_env(env_name, fire=True, frames_num=2, noop_num=30, skip_frames=True):
    env = gym.make(env_name)
    if skip_frames:
        env = MaxAndSkipEnv(env) # вызывается через каждые skip кадров
    if fire:
        env = FireResetEnv(env) # Пуск в начале
    env = NoopResetEnv(env, noop_max=noop_num)
    env = WarpFrame(env) # изменить форму и размер
    env = FrameStack(env, frames_num) # собрать группу из 4 кадров
    return env
```

Единственный отсутствующий шаг предобработки – масштабирование кадра. Мы займемся этим непосредственно перед тем, как подать наблюдаемый кадр на вход нейронной сети. Это связано с тем, что в классе `FrameStack` используется специальный эффективно использующий память «ленивый» массив, который будет потерян, если применять масштабирование как обертку.

Реализация DQN

Хотя DQN – довольно простой алгоритм, на этапе выбора проектных решений и реализации следует проявлять осмотрительность. Как и любой другой алгоритм глубокого ОП, его трудно отлаживать и настраивать. Но в данной книге мы предложим некоторые рекомендации и приемы, которые помогут справиться с этой задачей.

Код DQN содержит четыре основных компонента:

- глубокие нейронные сети (ГНС);
- буфер воспроизведения;
- граф вычислений;
- цикл обучения (и оценки).

Как обычно, код написан на Python с использованием TensorFlow, а для визуализации хода обучения и оценки качества алгоритма мы используем `TensorBoard`.



Весь код имеется в репозитории этой книги на GitHub. Не забудьте извлечь его оттуда. Мы опускаем реализации некоторых простых функций, чтобы не перегружать книгу.

Не откладывая в долгий ящик, приступим к реализации и для начала импортируем необходимые библиотеки:

```
import numpy as np
import tensorflow as tf
import gym
from datetime import datetime
from collections import deque
import time
import sys

from atari_wrappers import make_env
```

Библиотека `atari_wrappers` включает показанную выше функцию `make_env`.

Глубокие нейронные сети

Ниже описана архитектура ГНС (компоненты строятся последовательно):

- 1) сверточный слой, содержащий 16 фильтров размера 8×8 с четырьмя шагами и функцией активации ReLU;
- 2) сверточный слой, содержащий 32 фильтра размера 4×4 с двумя шагами и функцией активации ReLU;
- 3) сверточный слой, содержащий 32 фильтра размера 3×3 с одним шагом и функцией активации ReLU;
- 4) плотный слой, содержащий 128 блоков, с функцией активации ReLU;
- 5) плотный слой, содержащий столько блоков, сколько действий имеется в среде, с линейной функцией активации.

В функции `cnn` определены первые три сверточных слоя, а в функции `fnn` – последние два плотных слоя.

```
def cnn(x):
    x = tf.layers.conv2d(x, filters=16, kernel_size=8, strides=4,
padding='valid', activation='relu')
    x = tf.layers.conv2d(x, filters=32, kernel_size=4, strides=2,
padding='valid', activation='relu')
    return tf.layers.conv2d(x, filters=32, kernel_size=3, strides=1,
padding='valid', activation='relu')
```

```
def fnn(x, hidden_layers, output_layer, activation=tf.nn.relu,
last_activation=None):
    for l in hidden_layers:
        x = tf.layers.dense(x, units=l, activation=activation)
    return tf.layers.dense(x, units=output_layer, activation=last_activation)
```

В этом коде `hidden_layers` – список целых значений. В нашей реализации `hidden_layers=[128]`. А `output_layer` – количество действий, доступных агенту.

В функции `qnet` слои CNN и FNN связываются со слоем, который сериализует двумерный выход слоя CNN:

```
def qnet(x, hidden_layers, output_size, fnn_activation=tf.nn.relu,
last_activation=None):
    x = cnn(x)
    x = tf.layers.flatten(x)
    return fnn(x, hidden_layers, output_size, fnn_activation, last_activation)
```

Глубокая нейронная сеть полностью определена. Теперь нужно связать ее с главным графом вычислений.

Буфер воспроизведения

Буфер воспроизведения представлен классом `ExperienceBuffer`, в нем хранятся FIFO-очереди (обслуживаемые в порядке «первым пришел – первым ушел») для каждого из следующих компонент: наблюдение, вознаграждение, действие, следующее наблюдение, признак завершения. FIFO в данном случае означает, что по достижении максимальной емкости `maxlen` из очереди удаляются элементы, начиная с самых старых. В нашей реализации емкость равна `buffer_size`:

```
class ExperienceBuffer():
    def __init__(self, buffer_size):
        self.obs_buf = deque(maxlen=buffer_size)
        self.rew_buf = deque(maxlen=buffer_size)
        self.act_buf = deque(maxlen=buffer_size)
        self.obs2_buf = deque(maxlen=buffer_size)
        self.done_buf = deque(maxlen=buffer_size)

    def add(self, obs, rew, act, obs2, done):
        self.obs_buf.append(obs)
        self.rew_buf.append(rew)
        self.act_buf.append(act)
        self.obs2_buf.append(obs2)
        self.done_buf.append(done)
```

Класс ExperienceBuffer отвечает также за выборку мини-пакетов, используемых для обучения нейронной сети. Они равномерно выбираются из буфера и имеют предопределенный размер batch_size:

```
def sample_minibatch(self, batch_size):
    mb_indices = np.random.randint(len(self.obs_buf), size=batch_size)

    mb_obs = scale_frames([self.obs_buf[i] for i in mb_indices])
    mb_rew = [self.rew_buf[i] for i in mb_indices]
    mb_act = [self.act_buf[i] for i in mb_indices]
    mb_obs2 = scale_frames([self.obs2_buf[i] for i in mb_indices])
    mb_done = [self.done_buf[i] for i in mb_indices]
    return mb_obs, mb_rew, mb_act, mb_obs2, mb_done
```

Наконец, мы переопределяем метод `_len`, сообщающий длину буферов. Заметим, что поскольку все буферы одинакового размера, мы возвращаем только длину `self.obs_buf`:

```
def __len__(self):
    return len(self.obs_buf)
```

Граф вычислений и цикл обучения

Ядро алгоритма – граф вычислений и цикл обучения (и оценки) – реализовано в функции `DQN`, которая принимает в качестве аргументов имя окружающей среды и все гиперпараметры:

```
def DQN(env_name, hidden_sizes=[32], lr=1e-2, num_epochs=2000,
        buffer_size=100000, discount=0.99, update_target_net=1000, batch_size=64,
        update_freq=4, frames_num=2, min_buffer_size=5000, test_frequency=20,
        start_explor=1, end_explor=0.1, explor_steps=100000):
    env = make_env(env_name, frames_num=frames_num, skip_frames=True, noop_num=20)
    env_test = make_env(env_name, frames_num=frames_num, skip_frames=True, noop_num=20)
    env_test = gym.wrappers.Monitor(env_test,
    "VIDEOS/TEST_VIDEOS"+env_name+str(current_milli_time()),force=True,
    video_callable=lambda x: x%20==0)

    obs_dim = env.observation_space.shape
    act_dim = env.action_space.n
```

Сначала создаются две окружающие среды: одна для обучения, другая для тестирования. Класс `gym.wrappers.Monitor` – обертка `Gym`, которая сохраняет игровую окружающую среду в формате видео, а параметр `video_callable` определяет, как часто сохранять видео, а в нашем случае – через каждые 20 эпизодов.

Затем мы инициализируем граф `TensorFlow` и создаем местозаменители для наблюдений, действий и целевых ценностей. Это делается в следующих строчках:

```
tf.reset_default_graph()
obs_ph = tf.placeholder(shape=(None, obs_dim[0], obs_dim[1],
obs_dim[2]), dtype=tf.float32, name='obs')
act_ph = tf.placeholder(shape=(None,), dtype=tf.int32, name='act')
y_ph = tf.placeholder(shape=(None,), dtype=tf.float32, name='y')
```

Теперь можно создать целевую и онлайн-овую сеть, вызвав ранее определенную функцию `qnet`. Поскольку целевая сеть должна иногда обновляться, перенимая параметры онлайн-овой сети, нам понадобится операция `update_target_op`, которая присваивает каждой переменной целевой сети значение соответствующей переменной онлайн-овой сети. Это присваивание производится методом `TensorFlow assign`. А метод `tf.group` агрегирует все элементы списка `update_target` в одной операции. Вот как выглядит реализация:

```
with tf.variable_scope('target_network'):
    target_qv = qnet(obs_ph, hidden_sizes, act_dim)
    target_vars = tf.trainable_variables()

with tf.variable_scope('online_network'):
    online_qv = qnet(obs_ph, hidden_sizes, act_dim)
    train_vars = tf.trainable_variables()

update_target = [train_vars[i].assign(train_vars[i+len(target_vars)])]
for i in range(len(train_vars) - len(target_vars)):
    update_target_op = tf.group(*update_target)
```

Итак, нейронные сети и операцию обновления целевой сети мы определили, и осталось определить только функцию потерь. Она равна $(y_j - Q_\theta(\varphi_j, a_j))^2$ (или (5.5), что эквивалентно). Для нее нужны значения y_j , которые вычисляются по формуле (5.6) и передаются через местозаменитель `y_ph`, и значения `Q`-функции онлайн-овой сети $Q_\theta(\varphi_j, a_j)$. Значения `Q`-функции зависят от действия a_j , но поскольку онлайн-овая сеть выводит значение для каждого действия, то нужно найти какой-то способ извлекать только значение `Q` для a_j , отбрасывая ценности всех остальных действий. Эту операцию можно выполнить, взяв унитарный код действия a_j и затем умножив его на выход онлайн-овой сети. Например, если всего имеется пять действий и $a_j = 3$, то унитарным кодом будет `[0,0,0,1,0]`. Если на выходе онлайн-овой сети мы имеем `[3.4, 3.7, 5.4, 2.1, 1.2]`, то результатом умножения на унитарный код будет `[0, 0, 0, 5.4, 0]`. Теперь, чтобы получить значение `Q`-функции, надо просто просуммировать элементы этого вектора – получим 5.4. Все это реализовано тремя строчками кода:

```
act_onehot = tf.one_hot(act_ph, depth=act_dim)
q_values = tf.reduce_sum(act_onehot * online_qv, axis=1)
v_loss = tf.reduce_mean((y_ph - q_values)**2)
```

Для минимизации только что определенной функции потерь используется метод Adam, вариант СГС:

```
v_opt = tf.train.AdamOptimizer(lr).minimize(v_loss)
```

На этом создание графа вычислений можно считать законченным. Но перед тем как перейти к главному циклу DQN, нужно все подготовить к сохранению скаляров и гистограмм. Тогда мы впоследствии сможем визуализировать их в TensorBoard.

```
now = datetime.now()
clock_time = "{}_{}_{}.".format(now.day, now.hour, now.minute,
int(now.second))

mr_v = tf.Variable(0.0)
ml_v = tf.Variable(0.0)

tf.summary.scalar('v_loss', v_loss)
tf.summary.scalar('Q-value', tf.reduce_mean(q_values))
tf.summary.histogram('Q-values', q_values)

scalar_summary = tf.summary.merge_all()
reward_summary = tf.summary.scalar('test_rew', mr_v)
mean_loss_summary = tf.summary.scalar('mean_loss', ml_v)

hyp_str = "-lr_{}-upTN_{}-upF_{}-frms_{}".format(lr, update_target_net,
update_freq, frames_num)
file_writer =
tf.summary.FileWriter('log_dir/'+env_name+'/DQN_'+clock_time+'_'+hyp_str,
tf.get_default_graph())
```

Здесь все понятно. Вопрос могут вызвать разве что переменные `mr_v` и `ml_v`. Именно их мы хотим отслеживать в TensorBoard. Но, поскольку они не определены внутри графа вычислений, придется объявить их отдельно и присвоить значения в методе `session.run` позже. При создании объекта `FileWriter` указывается файл с уникальным именем, и объект ассоциируется с графом по умолчанию.

Теперь можно написать функцию `agent_op`, которая масштабирует наблюдение и выполняет для него все вычисления. Наблюдение уже было пропущено через конвейер предобработки (построенный с помощью оберток окружающей среды), но масштабирование мы отложили на потом:

```
def agent_op(o):
    o = scale_frames(o)
    return sess.run(online_qv, feed_dict={obs_ph:[o]})
```

Затем создается сеанс, инициализируются переменные, и окружающая среда сбрасывается в начальное состояние:

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

step_count = 0
last_update_loss = []
ep_time = current_milli_time()
batch_rew = []

obs = env.reset()
```

Следующий этап – создание буфера воспроизведения, обновление целевой сети (присваивание ей тех же параметров, что у онлайн-сети) и инициализация скорости затухания `eps_decay`. В нашем коде эpsilon затухает по тому же закону, что в оригинальной статье о DQN. Затухание линейно, а его скорость выбрана так, чтобы конечное значение `end_explor` достигалось примерно за `explor_steps` шагов. Например, если мы хотим, чтобы эpsilon уменьшился от 1.0 до 0.1 за 1000 шагов, то на каждом шаге нужно уменьшать на величину $(1 - 0.1)/1000 = 0.0009$. Все это делается в следующих строчках кода:

```
obs = env.reset()
buffer = ExperienceBuffer(buffer_size)
sess.run(update_target_op)

eps = start_explor
eps_decay = (start_explor - end_explor) / explor_steps
```

Напомним, что цикл обучения состоит из двух вложенных циклов: во внешнем обходятся эпохи, а во внутреннем реализуются переходы состояний в одной эпохе. Первая часть внутреннего цикла вполне стандартна. Выбирается действие, следуя ϵ -жадной стратегии, применяемой к онлайн-сети, производится один шаг взаимодействия со средой, новый переход добавляется в буфер, и, наконец, обновляются переменные:

```
for ep in range(num_epochs):
    g_rew = 0
    done = False

    while not done:
        act = eps_greedy(np.squeeze(agent_op(obs)), eps=eps)
        obs2, rew, done, _ = env.step(act)
        buffer.add(obs, rew, act, obs2, done)

        obs = obs2
        g_rew += rew
        step_count += 1
```

Здесь в `obs` записывается следующее наблюдение, после чего увеличивается полное `g_rew` вознаграждение, начисленное за игру.

Затем в том же цикле `eps` уменьшается, и при выполнении некоторых условий производится обучение онлайн-сети. Точнее, проверяется, что буфер достиг минимального размера и что сеть обучается один раз каждые `update_freq` шагов. Для обучения онлайн-сети сначала из буфера выбирается мини-пакет и вычисляются целевые ценности. Затем в процессе прогона сеанса минимизируется функция потерь `v_loss`, и словарь заполняется целевыми ценностями, действиями и наблюдениями из мини-пакета. Сеанс также возвращает `v_loss` и `scalar_summary` для статистики. Потом `scalar_summary` передается объекту `file_writer`, который сохраняет его в файле журнала TensorBoard. Наконец, через каждые `update_target_net` эпох обновляется целевая сеть. Вычисляется и записывается в файл журнала сводная информация о средних потерях. Все это реализовано в следующем фрагменте кода:

```

if eps > end_explor:
    eps -= eps_decay

if len(buffer) > min_buffer_size and (step_count % update_freq == 0):
    mb_obs, mb_rew, mb_act, mb_obs2, mb_done =
buffer.sample_minibatch(batch_size)
    mb_trg_qv = sess.run(target_qv, feed_dict={obs_ph:mb_obs2})
    # Вычислить целевые ценности
    y_r = q_target_values(mb_rew, mb_done, mb_trg_qv, discount)
    train_summary, train_loss, _ = sess.run([scalar_summary,
v_loss, v_opt], feed_dict={obs_ph:mb_obs, y_ph:y_r, act_ph: mb_act})

    file_writer.add_summary(train_summary, step_count)
    last_update_loss.append(train_loss)

if (len(buffer) > min_buffer_size) and (step_count %
update_target_net) == 0:
    _, train_summary = sess.run([update_target_op,
mean_loss_summary], feed_dict={ml_v:np.mean(last_update_loss)})
    file_writer.add_summary(train_summary, step_count)
    last_update_loss = []
    
```

По завершении эпохи окружающая среда сбрасывается в начальное состояние, начисленное за игру полное вознаграждение добавляется в список `batch_rew` и сбрасывается в 0. Каждые `test_frequency` эпох агент тестируется на 10 играх, и статистика дописывается в файл с помощью объекта `file_writer`. В конце обучения обе среды и `file_writer` закрываются. Код приведен ниже.

```

if done:
    obs = env.reset()
    batch_rew.append(g_rew)
    g_rew = 0

if ep % test_frequency == 0:
    test_rw = test_agent(env_test, agent_op, num_games=10)
    test_summary = sess.run(reward_summary, feed_dict={mr_v:np.mean(test_rw)})
    file_writer.add_summary(test_summary, step_count)
    print('Ep:%4d Rew:%4.2f, Eps:%2.2f -- Step:%5d -- Test:%4.2f
%4.2f' % (ep,np.mean(batch_rew), eps, step_count, np.mean(test_rw), np.std(test_rw)))
    batch_rew = []
    file_writer.close()
    env.close()
    env_test.close()
    
```

Вот и все. Теперь можно вызвать функцию `DQN`, передав ей имя окружающей среды `Гум` и все гиперпараметры:

```

if __name__ == '__main__':
    DQN('PongNoFrameskip-v4', hidden_sizes=[128], lr=2e-4,
buffer_size=100000, update_target_net=1000, batch_size=32, update_freq=2,
frames_num=2, min_buffer_size=10000)
    
```

И еще одно замечание, перед тем как перейти к анализу результатов. В качестве окружающей среды здесь использовалась не стандартная версия `Pong-v0`, а ее модифицированный вариант. Дело в том, что в стандартной версии каждое действие выполняется 2, 3 или 4 раза – сколько именно, определяется случайно

(с равномерным распределением). Но поскольку мы хотим пропускать фиксированное число кадров, то выбрали версию без встроенной возможности пропуска, `NoFrameskip`, и добавили нестандартную обертку `MaxAndSkipEnv`.

Результаты

Оценить ход выполнения алгоритма ОП – непростая задача. Самое очевидное решение – следить за конечной целью, т. е. за полным вознаграждением, начисленным на протяжении эпохи. Это хорошая метрика. Однако если использовать среднее вознаграждение в качестве критерия качества обучения, то возможен сильный шум из-за изменения весов. Это ведет к большим изменениям в распределении посещаемого состояния.

Поэтому мы оценивали алгоритм на 10 тестовых играх через каждые 20 эпох обучения и отслеживали среднее полное (необесцененного) вознаграждения, накопленного в этих играх. Кроме того, поскольку окружающая среда детерминированная, мы тестировали агента с ϵ -жадной стратегией ($\epsilon = 0.05$), чтобы оценка была надежнее. Скалярная сводка называется `test_rew`. Ее можно посмотреть в `TensorBoard`, зайдя в каталог, где хранятся журналы, и выполнив команду

```
tensorboard --logdir .
```

График, который должен быть вам уже знаком, показан на рис. 5.4. По оси x откладывается количество шагов. Как видим, график выходит на плато после линейного роста на первых 250 000 шагов и более быстрого, но тоже линейного роста на следующих 300 000 шагов:

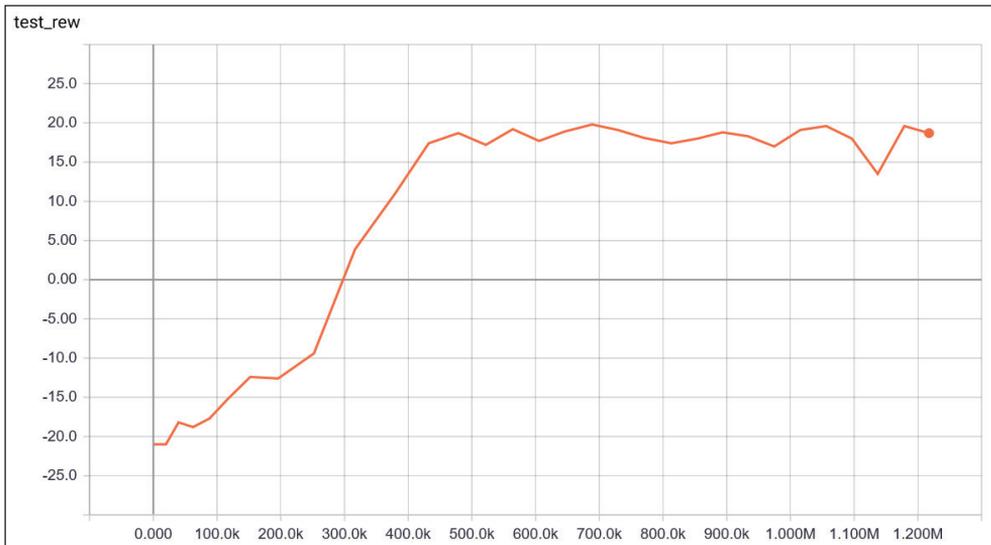


Рис. 5.4 ❖ График среднего полного вознаграждения в 10 играх. По оси x откладывается количество шагов

Pong – сравнительно простая задача. Мы обучали алгоритм примерно на 1.1 млн шагов, тогда как в оригинальной статье о DQN все алгоритмы обучались на 200 млн шагов.

Другой способ оценить алгоритм – посмотреть на вычисленные оценки ценности действий. В принципе, это полезная метрика, т. к. она отражает доверие к качеству пар состояние–действие. К сожалению, она не оптимальна, потому что некоторые алгоритмы склонны завышать оценку значений Q-функции, как мы скоро узнаем. Тем не менее мы построили график этой величины. Он показан на рис. 5.5, и, как и ожидалось, характер возрастания значений Q-функции в процессе обучения примерно такой же, как на предыдущем графике.

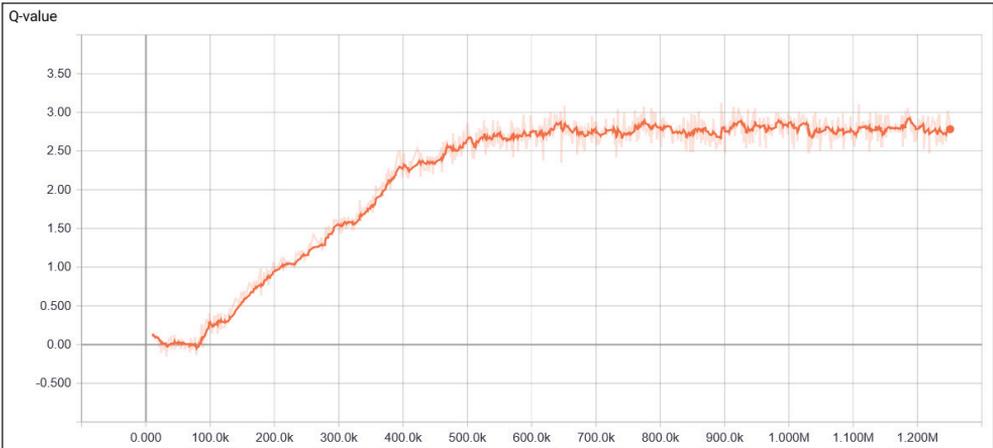


Рис. 5.5 ❖ График вычисленных в процессе обучения оценок значений Q-функции. По оси x откладывается количество шагов

Еще один важный график, изображенный на рис. 5.6, показывает, как изменяется функция потерь со временем. Он не так полезен, как в обучении с учителем, поскольку целевые значения не являются объективной истиной, но может дать неплохое представление о качестве модели.

ВАРИАЦИИ НА ТЕМУ DQN

Вдохновившись поразительными результатами DQN, многие ученые бросились изучать этот алгоритм и разработали модификации, повышающие его устойчивость, эффективность и качество. В этом разделе мы представим три таких усовершенствованных алгоритма, объясним стоящие за ними идеи и приведем реализацию. Первый из них – Double DQN, или DDQN – решает вышеупомянутую проблему завышенной оценки, свойственную DQN. Второй – Dueling DQN – отделяет представление ценности состояний от представления преимущества пары состояние–действие. Третий – *n*-шаговый DQN, это старая идея, заимствованная у TD-алгоритмов, которая занимает промежуточное положение между одношаговым обучением и обучением методом Монте-Карло.

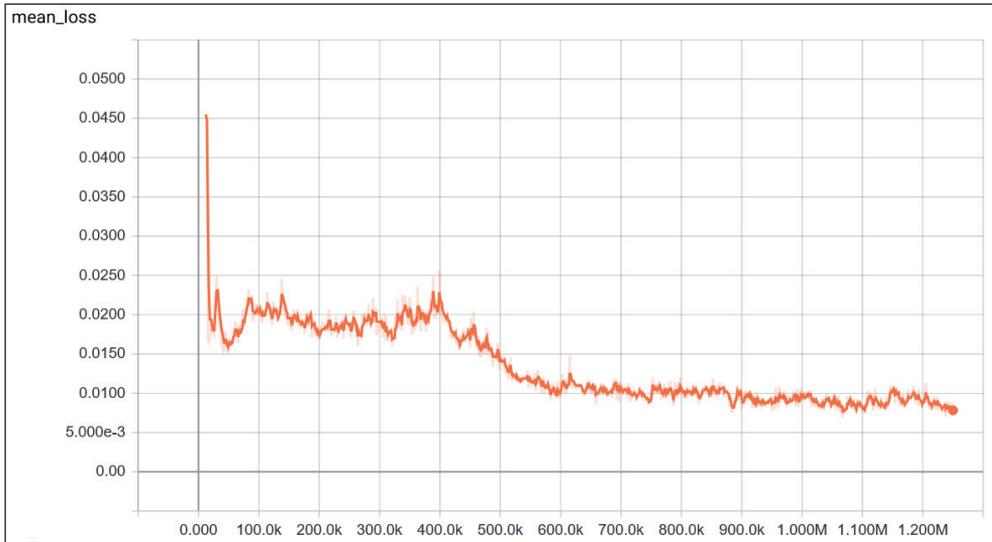


Рис. 5.6 ❖ График функции потерь

Double DQN

Завышение оценки значений Q-функции в алгоритмах Q-обучения – хорошо известная проблема. Причина – в операторе \max , который завышает истинную оценку ценности. Чтобы понять, в чем тут дело, предположим, что имеются зашумленные оценки со средним 0 и с ненулевой дисперсией, показанные на рис. 5.7. Несмотря на то что асимптотическое среднее равно 0, функция \max всегда возвращает значения, большие нуля.

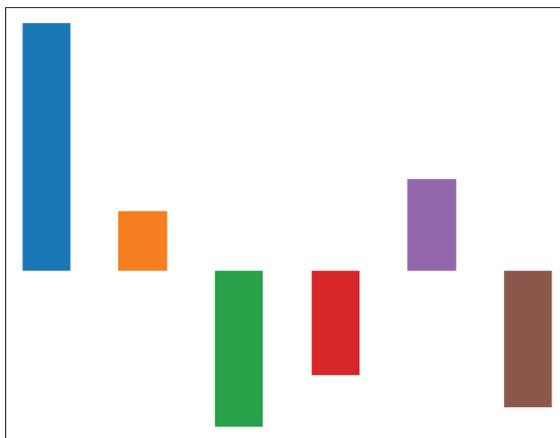


Рис. 5.7 ❖ Шесть значений, выбранных из нормального распределения со средним 0

В Q-обучении завышенная оценка не составляет особой проблемы, если завышение равномерно. Но если завышение неравномерно и ошибки для разных состояний и действий различаются, то завышение оказывает негативное влияние на алгоритм DQN и приводит к ухудшению найденной им стратегии.

Для решения этой проблемы авторы статьи «Deep Reinforcement Learning with Double Q-learning» предлагают использовать две нейронные сети: одну для выбора действий, а другую для оценивания значений Q-функции. Но вместо двух разных сетей с соответствующим увеличением сложности в статье предлагается использовать онлайнтовую сеть для выбора наилучшего действия с помощью операции \max , а целевую сеть – для вычисления значений Q-функции. При таком подходе целевая ценность U будет не такой, как в стандартном Q-обучении:

$$y = r + \gamma \max_{a'} \hat{Q}_{\theta'}(\phi', a') = r + \gamma \hat{Q}_{\theta'}(\phi', \operatorname{argmax}_{a'} \hat{Q}_{\theta'}(s', a')),$$

а такой:

$$y = r + \gamma \hat{Q}_{\theta'}(\phi', \operatorname{argmax}_{a'} Q_{\theta}(s', a')). \quad (5.7)$$

В этом варианте проблема завышения оценки теряет остроту, а алгоритм становится устойчивее.

Реализация DDQN

С точки зрения кода, единственное изменение, необходимое для реализации DDQN, – этап обучения. Нужно лишь заменить строки

```
mb_trg_qv = sess.run(target_qv, feed_dict={obs_ph:mb_obs2})
y_r = q_target_values(mb_rew, mb_done, mb_trg_qv, discount)
```

такими:

```
mb_onl_qv, mb_trg_qv = sess.run([online_qv, target_qv], feed_dict={obs_ph:mb_obs2})
y_r = double_q_target_values(mb_rew, mb_done, mb_trg_qv, mb_onl_qv, discount)
```

Здесь `double_q_target_values` – функция, которая производит вычисления по формуле (5.7) для каждого перехода в мини-пакете.

Результаты

Чтобы понять, действительно ли DQN завышает оценку значений Q-функции по сравнению с DDQN, мы построили обе кривые на одном графике (рис. 5.8). Оранжевым цветом изображены результаты DQN, синим – результаты DDQN.

А на рис. 5.9 показаны качество DDQN (синяя линия) и DQN (оранжевая линия), оцениваемых по среднему вознаграждению в тестовых играх.

Как и следовало ожидать, значения Q-функции для DDQN всегда меньше, чем для DQN, т. е. последний алгоритм действительно завышает оценки ценности. Впрочем, на качестве алгоритмов в тестовых играх это, похоже, не сказывается, т. е. завышение оценки не является существенным фактором. Однако помните, что мы тестировали алгоритм только на игре Pong. Об эффективности алгоритма не следует судить по качеству в одной лишь окружающей среде. На самом деле авторы статьи применили его ко всем 57 играм в среде разработки

ALE и сообщают, что DDQN не только дает более точные оценки ценности, но и в нескольких играх набирает значительно больше очков.

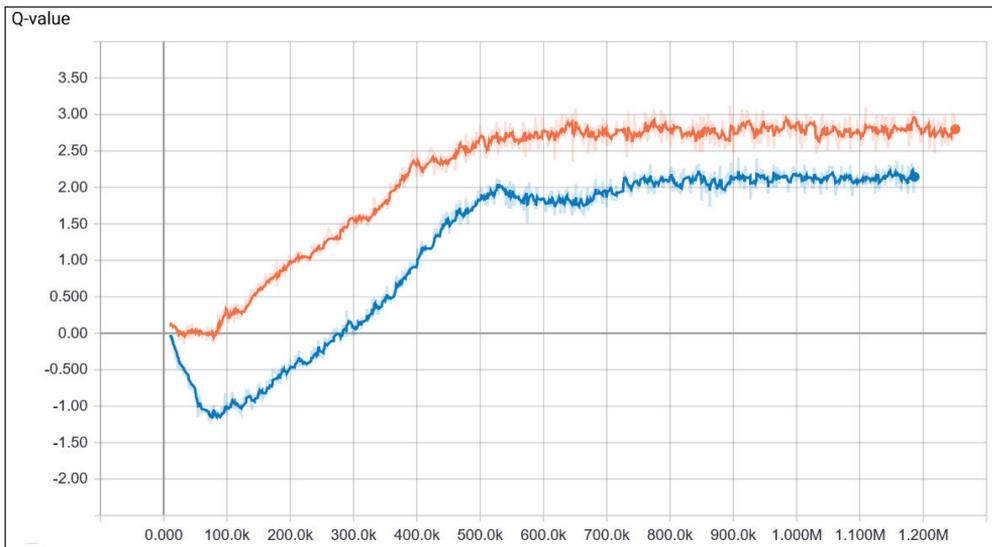


Рис. 5.8 ❖ График, демонстрирующий завышение оценок значений Q-функции в результате обучения. Синим цветом изображены результаты DDQN, оранжевым – результаты DQN. По оси x откладывается количество шагов

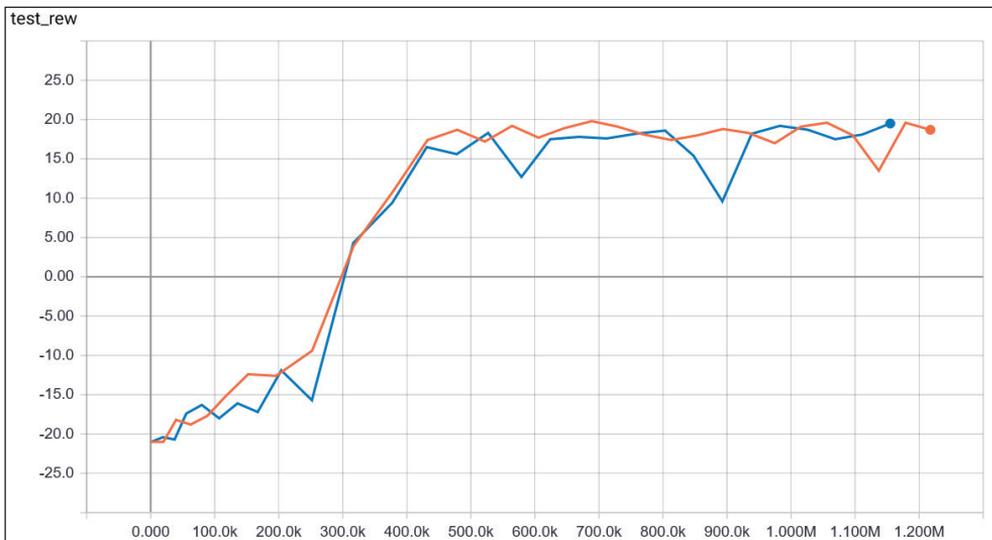


Рис. 5.9 ❖ График среднего вознаграждения в тестовых играх. Синим цветом изображены результаты DDQN, оранжевым – результаты DQN. По оси x откладывается количество шагов

Dueling DQN

В работе «Dueling Network Architectures for Deep Reinforcement Learning» (<https://arxiv.org/abs/1511.06581>) предложена новая архитектура нейронной сети с двумя оценщиками: один для функции ценности состояний, другой для функции преимущества пары состояние–действие.

Функция преимущества используется в ОП повсеместно и определяется следующим образом:

$$A(s, a) = Q(s, a) - V(s).$$

Функция преимущества говорит о том, насколько действие a в состоянии s лучше среднего по всем действиям в этом состоянии. Следовательно, если $A(s, a)$ положительно, то действие a в состоянии s лучше среднего. Напротив, если $A(s, a)$ отрицательно, то действие a в состоянии s хуже среднего.

Таким образом, оценивая функцию ценности и функцию преимущества отдельно, как предлагается в статье, мы сможем переопределить Q -функцию следующим образом:

$$Q(s, a) = V(s) + A(s, a) - \frac{1}{|A|} \sum_{a'} A(s, a'). \quad (5.8)$$

Здесь мы прибавили среднее преимущество, чтобы повысить устойчивость алгоритма DQN.

Архитектура сети Dueling DQN включает две «головы» (или потока): одну для функции ценности, другую для функции преимущества, и обе разделяют общий сверточный модуль. Авторы сообщают, что такая архитектура позволяет обучиться тому, какие состояния ценны, а какие нет, не вычисляя абсолютные ценности каждого действия в каждом состоянии. Они протестировали новую архитектуру на играх Atari и добились значительного улучшения качества в целом.

Реализация Dueling DQN

Одно из преимуществ этой архитектуры и формулы (5.8) состоит в том, что в базовый алгоритм ОП не нужно вносить никаких изменений. Изменения касаются только построения Q -сети, т. е. мы заменяем функцию `qnet` функцией `dueling_qnet`:

```
def dueling_qnet(x, hidden_layers, output_size, fnn_activation=tf.nn.relu,
last_activation=None):
    x = cnn(x)
    x = tf.layers.flatten(x)
    qf = fnn(x, hidden_layers, 1, fnn_activation, last_activation)
    aaqf = fnn(x, hidden_layers, output_size, fnn_activation, last_activation)
    return qf + aaqf - tf.reduce_mean(aaqf)
```

Создаются две нейронные сети прямого распространения: у первой только один выход (функция ценности), а у второй – столько выходов, сколько имеется действий агента (зависящая от состояния функция преимущества действия). В последней строке возвращается результат, вычисленный по формуле (5.8).

Результаты

На рис. 5.10 показаны вознаграждения, полученные в тестовых играх. Результат однозначно демонстрирует преимущества дуэльной архитектуры.

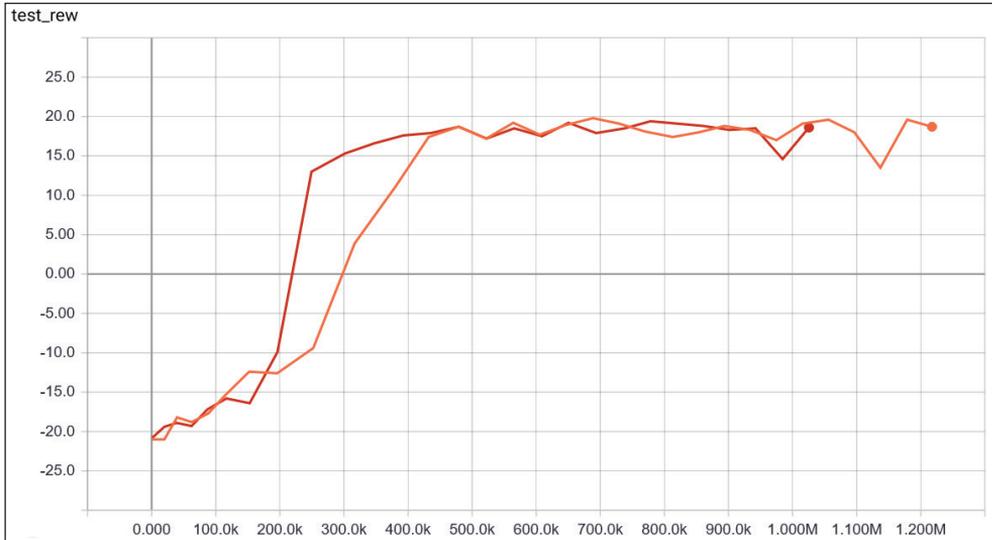


Рис. 5.10 ❖ График среднего вознаграждения в тестовых играх.

Красным цветом изображены результаты Dueling DQN, оранжевым – результаты DQN. По оси x откладывается количество шагов

n -шаговый DQN

Идея n -шагового DQN стара и связана с антагонизмом между обучением методом Монте-Карло и обучением на основе временных различий. Как было показано в главе 4, эти алгоритмы находятся на противоположных концах спектра. Алгоритм TD-обучения обучается на одном шаге, а алгоритм Монте-Карло – на всей траектории. Для TD-обучения характерна минимальная дисперсия, но и максимальное смещение, а для метода Монте-Карло – наоборот. Баланс между дисперсией и смещением можно улучшить, воспользовавшись n -шаговым доходом, т. е. доходом, вычисленным после n шагов. TD-обучение можно рассматривать как 0-шаговый доход, а метод Монте-Карло – как ∞ -шаговый доход.

Для n -шагового дохода формула обновления ценности выглядит так:

$$y_t = \sum_{t'=t}^{t'+N-1} r_{t'} + \gamma^N \max_{a'_{t+N}} \hat{Q}_{\theta'}(\phi'_{t+N}, a'_{t+N}), \quad (5.9)$$

где N – количество шагов.

N -шаговый доход можно уподобить заглядыванию на n шагов вперед, но на практике реально заглянуть в будущее невозможно, поэтому поступают ровно наоборот – вычисляют ценность, которая была n шагов назад. Такие значения доступны только в момент времени $t + n$, что вносит в процесс обучения запаздывание.

Основное преимущество этого подхода в том, что целевые ценности не так сильно смещены, а это ведет к ускорению обучения. При этом возникает важная проблема – вычисленные таким образом ценности правильны, только когда используется алгоритм обучения с единой стратегией (DQN – алгоритм с разделенной стратегией). Дело в том, что в формуле (5.9) предполагается, что стратегия, которой агент будет следовать на протяжении следующих n шагов, совпадает со стратегией, использованной при накоплении опыта. Существуют способы приспособить эту идею к случаю разделенной стратегии, но все они трудны для реализации, поэтому общая практика – ограничиться небольшими n и просто игнорировать проблему.

Реализация

Для реализации n -шагового DQN нужно внести всего несколько изменений. При выборке из буфера следует возвращать вознаграждение n шагов назад, следующее состояние n шагов назад и флаг done n шагов назад. Все это очень просто, поэтому мы не станем здесь приводить реализацию, но ее можно найти в репозитории этой книги на GitHub. Код для поддержки n -шагового дохода находится в классе `MultiStepExperienceBuffer`.

Результаты

Для алгоритмов с разделенной стратегией (таких как DQN) n -шаговое обучение хорошо работает при малых значениях n . Что касается конкретно DQN, то показано, что он хорошо работает, когда n находится между 2 и 4; это дает улучшения во многих играх Atari.

На рис. 5.11 представлены результаты нашей реализации.

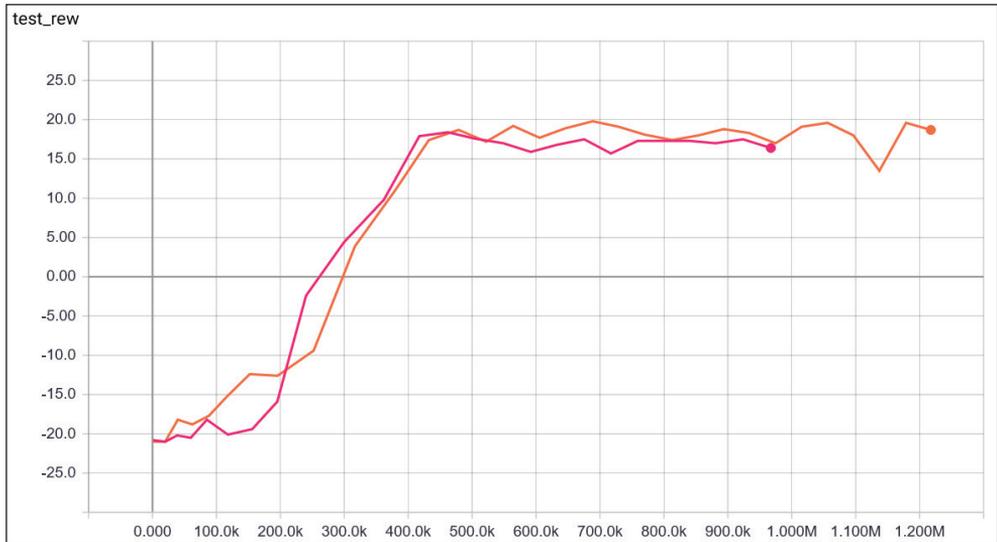


Рис. 5.11 ❖ График среднего вознаграждения в тестовых играх. Бордовым цветом изображены результаты 3-шагового DQN, оранжевым – результаты DQN. По оси x откладывается количество шагов

Мы протестировали DQN с трехшаговым доходом. Видно, что плоды обучения начинают проявляться несколько позже. Но затем обучение происходит быстрее, хотя общие формы кривых близки.

РЕЗЮМЕ

В этой главе мы продолжили изучение алгоритмов ОП и обсудили, как их можно сочетать с аппроксимацией функций, что расширяет применимость ОП. Конкретно мы описали использование глубоких нейронных сетей в качестве аппроксиматоров и показали, как при этом может возникать неустойчивость. Мы продемонстрировали, что на практике для соединения глубоких нейронных сетей с Q-обучением необходимы модификации.

Первым алгоритмом, в котором удалось сочетать глубокие нейронные сети с Q-обучением, был DQN. Он включает два ключевых компонента, призванных стабилизировать обучение и управлять решением таких сложных задач, как игры для Atari 2600. Эти два компонента – буфер воспроизведения, в котором хранится прежний опыт, и отдельная целевая сеть, обновляемая реже, чем онлайнная. Первая нужна, чтобы задействовать свойство разделенной стратегии, присущее Q-обучению, благодаря которому алгоритм может обучаться на опыте, собранном с использованием другой стратегии, и выбирать больше независимых и одинаково распределенных мини-пакетов из большого пула данных, чтобы можно было применить стохастический градиентный спуск. Вторая введена, чтобы стабилизировать целевые ценности и уменьшить нестационарность задачи.

После формального введения в алгоритм DQN мы реализовали его и протестировали на игре Pong. Мы также продемонстрировали некоторые технические особенности реализации алгоритма, в т. ч. конвейер предобработки и обертки. После публикации оригинальной статьи о DQN появилось много вариаций на эту тему, ставящих целью улучшить алгоритм и сделать его более устойчивым. Мы рассмотрели и реализовали три таких варианта: Double DQN, Dueling DQN и n -шаговый DQN. Мы применяли эти алгоритмы только к играм Atari, но в действительности они применимы ко многим реальным задачам.

В следующей главе мы познакомимся с другой категорией алгоритмов глубокого ОП – алгоритмами градиента стратегии. Это алгоритмы с единой стратегией, и, как мы увидим, они обладают некоторыми весьма важными уникальными характеристиками, которые делают их применимыми к широкому кругу задач.

Вопросы

1. Когда возникает смертельная триада?
2. Как DQN борется с неустойчивостью?
3. В чем суть проблемы движущейся мишени?
4. Как DQN удается смягчить проблему движущейся мишени?

5. Опишите процедуру оптимизации, используемую в DQN.
6. Как определяется функция преимущества пары состояние–действие?

Для дальнейшего чтения

- Подробное пособие по работе с обертками OpenAI Gym см. в статье <https://hub.packtpub.com/openai-gym-environments-wrappers-and-monitors-tutorial/>.
- Оригинальную статью «Rainbow» см. по адресу <https://arxiv.org/abs/1710.02298>.

Глава 6

Стохастическая оптимизация и градиенты стратегии

До сих пор мы разрабатывали алгоритмы обучения с подкреплением на основе ценности. Эти алгоритмы обучают функцию ценности с целью найти хорошую стратегию. И хотя они демонстрируют неплохие результаты, их применение ограничено причинами, заложенными в самом принципе работы. В этой главе мы рассмотрим новый класс алгоритмов, призванный преодолеть ограничения методов на основе ценности, – алгоритмы градиента стратегии.

Методы градиента стратегии выбирают действие, основываясь на обученной параметрической стратегии, а не на функции ценности. В этой главе мы обсудим теоретические и интуитивные соображения, лежащие в основе таких методов, а затем на этом фундаменте разработаем простейший вариант алгоритма градиента стратегии **REINFORCE**.

Из-за простоты у алгоритма REINFORCE имеются некоторые недостатки, но их можно смягчить, приложив дополнительные усилия. Поэтому мы представим две улучшенные версии REINFORCE, а именно: **REINFORCE** с базой и модели **исполнитель–критик** (actor-critic – **AC**).

В этой главе рассматриваются следующие вопросы:

- методы градиента стратегии;
- устройство алгоритма REINFORCE;
- REINFORCE с базой;
- обучение алгоритма AC.

МЕТОДЫ ГРАДИЕНТА СТРАТЕГИИ

Рассмотренные до сих пор алгоритмы по существу обучают функцию ценности $V(s)$ или функцию ценности действий $Q(s, a)$. Функция ценности определяет полное вознаграждение, которое можно получить, стартуя из данного состояния или пары состояние–действие. Зная ее, можно выбирать действие, основываясь на оценке ценности действий (или состояния).

Поэтому жадную стратегию можно определить следующим образом:

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

Методы на основе ценности в сочетании с глубокими нейронными сетями могут обучаться весьма изощренным стратегиям управления агентом в многомерном пространстве. Но, несмотря на все свои достоинства, они испытывают затруднения, когда в задаче очень много действий или пространство действий непрерывно.

В таких случаях операция поиска максимума практически неосуществима. Алгоритмы **градиента стратегии** (ГС) как раз в таких контекстах обладают невероятным потенциалом, потому что легко адаптируются к непрерывным пространствам действий.

Методы ГС принадлежит более широкому классу методов на основе стратегии, включающему эволюционные стратегии. Мы будем изучать его в главе 11 «Оптимизация методом черного ящика».

Теперь мы можем уточнить классификацию алгоритмов ОП, приведенную в главе 3 (рис. 6.1).

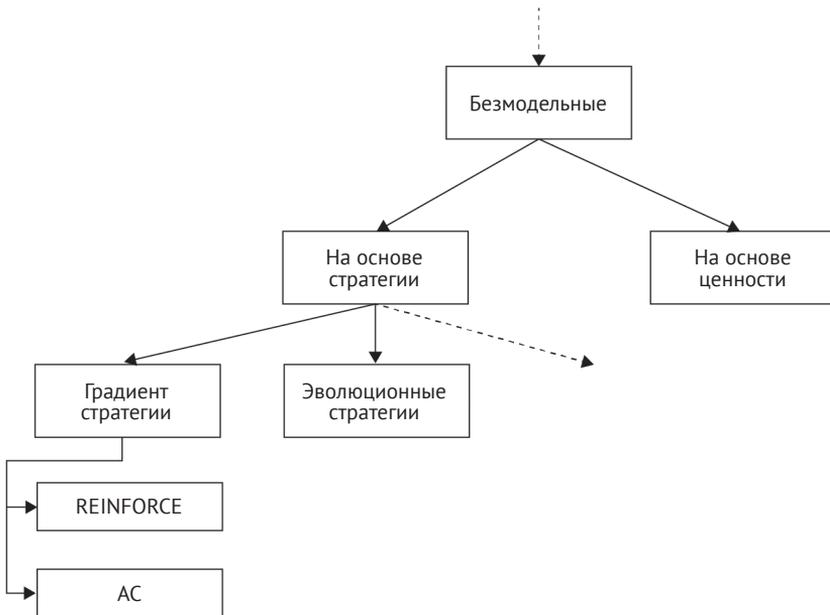


Рис. 6.1 ❖ Примерами методов градиента стратегии служат **REINFORCE** и **АС**, которые будут рассмотрены в следующих разделах

Градиент стратегии

Целью ОП является максимизация ожидаемого дохода (полного вознаграждения, с обесцениванием или без) вдоль траектории. Следовательно, целевую функцию можно записать в виде

$$J(\theta) = E_{\tau \sim \pi_{\theta}}[R(\tau)], \quad (6.1)$$

где θ – вектор параметров стратегии, например обученные веса глубокой нейронной сети.

В методах ГС для максимизации целевой функции используется ее градиент $\nabla_{\theta} J(\theta)$. Применяя метод градиентного подъема, мы можем улучшить функцию $J(\theta)$, изменяя параметры в направлении градиента, поскольку направление градиента совпадает с направлением наискорейшего роста функции.

После того как максимум найден, стратегия π_{θ} будет порождать траектории с наибольшим возможным доходом. На интуитивном уровне градиент стратегии подталкивает к выбору хороших стратегий, увеличивая их вероятность, и одновременно штрафует плохие стратегии, уменьшая их вероятность. Пользуясь формулой (6.1), мы можем записать градиент целевой функции в виде

$$\nabla_{\theta} J(\theta) = \nabla_{\theta} E_{\tau \sim \pi_{\theta}}[R(\tau)]. \quad (6.2)$$

Вспоминая определения из предыдущих глав, можно сказать, что в методах градиента стратегии для оценивания стратегии применяется оценка дохода R . А в роли улучшения стратегии выступает шаг оптимизации вектора параметров θ .

Теорема о градиенте стратегии

Глядя на формулу (6.2), мы сразу же обнаруживаем проблему, поскольку при такой формулировке градиент целевой функции зависит от распределения состояний стратегии, т. е.

$$\nabla_{\theta} J(\theta) = \nabla_{\theta} E_{\tau \sim \pi_{\theta}}[R(\tau)] = \nabla_{\theta} \sum_s d(s) \sum_a \pi_{\theta}(a|s) R(s, a). \quad (6.3)$$

Мы могли бы использовать стохастическую аппроксимацию математического ожидания, но чтобы вычислить распределение состояний $d(s)$, нам все равно нужна полная модель окружающей среды. Так что такая формулировка нам не подходит.

На помощь приходит теорема о градиенте стратегии. Она дает аналитическое выражение для вычисления градиента целевой функции по параметрам стратегии, в которое не входит производная распределения состояний. Формально теорема о градиенте стратегии позволяет выразить градиент целевой функции в виде

$$\nabla_{\theta} J(\theta) = E_{\tau \sim \pi_{\theta}}[\nabla_{\theta} \log \pi_{\theta}(\tau) R(\tau)] = E_{\pi_{\theta}}[\nabla_{\theta} \log \pi_{\theta}(a|s) Q_{\pi_{\theta}}(s, a)]. \quad (6.4)$$

Доказательство теоремы о градиенте стратегии выходит за рамки этой книги, но его можно найти в книге Саттона и Барто (<http://incompleteideas.net/book/the-book-2nd.html>) и в других источниках в сети.

Поскольку градиент целевой функции не включает производную распределения состояний, математическое ожидание можно оценить, производя выборку из стратегии. Стало быть, градиент целевой функции можно аппроксимировать следующим образом:

$$\nabla_{\theta} J(\theta) = \frac{1}{N} \sum_{i=0}^N [\nabla_{\theta} \log \pi_{\theta}(a_i|s_i) Q_{\pi_{\theta}}(s_i, a_i)]. \quad (6.5)$$

С помощью этой формулы можно выписать стохастическое обновление с градиентным подъемом:

$$\theta = \theta + \alpha \nabla_{\theta} J(\theta). \quad (6.5a)$$

Заметим, что поскольку мы хотим максимизировать целевую функцию, параметры следует изменять в направлении градиента (а не в противоположном, как в методе градиентного спуска, который описывается формулой $\theta = \theta - \alpha \nabla_{\theta} J(\theta)$).

Идея, выражаемая формулой (6.5), заключается в том, чтобы увеличить вероятность повторного выбора хороших действий в будущем, одновременно уменьшив вероятность плохих действий. Качество действий переносится скалярным выражением $Q_{\pi_{\theta}}(s_i, a_i)$, определяющим качество пары состояние–действие.

Вычисление градиента

При условии что стратегия дифференцируемая, ее градиент легко вычисляется с помощью современных программ автоматического дифференцирования.

В TensorFlow bs для этого можем определить граф вычислений и вызвать метод `tf.gradient(loss_function, variables)`, который вычисляет градиент функции потерь (`loss_function`) по обучаемым параметрам `variables`. Альтернатива – напрямую максимизировать целевую функцию, воспользовавшись методом стохастического градиентного спуска, например вызвав метод `tf.train.AdamOptimizer(lr).minimize(-objective_function)`.

Ниже показаны шаги вычисления аппроксимации в формуле (6.5), когда стратегия определена в дискретном пространстве действий размерности `env.action_space.n`:

```
pi = policy(states) # вероятности действий
onehot_action = tf.one_hot(actions, depth=env.action_space.n)
pi_log = tf.reduce_sum(onehot_action * tf.math.log(pi), axis=1)
pi_loss = -tf.reduce_mean(pi_log * Q_function(states, actions))

# вычислить градиенты pi_loss по variables
gradients = tf.gradient(pi_loss, variables)

# или оптимизировать pi_loss непосредственно методом Adam (или любым другим
# методом на основе CFC)
# pi_opt = tf.train.AdamOptimizer(lr).minimize(pi_loss)
```

Вызов `tf.one_hot` выполняет унитарное кодирование действий `actions`, т. е. для каждого действия порождает вектор-маску, содержащий 1 в позиции, соответствующей числовому значению действия, и 0 в остальных.

Затем в третьей строчке эта маска умножается на логарифм вероятности действия, и в результате получается логарифмическая вероятность действий `actions`. В четвертой строчке вычисляется потеря по формуле

$$\frac{1}{N} \sum_{i=0}^N [\log \pi_{\theta}(a_i | s_i) Q_{\pi_{\theta}}(s_i, a_i)].$$

И наконец, `tf.gradient` вычисляет градиенты `pi_loss` по параметрам `variables` в соответствии с формулой (6.5).

Стратегия

В случае когда действия дискретны и их немного, самый распространенный подход состоит в том, чтобы создать параметрическую стратегию, которая порождает числовое значение для каждого действия.

❗ Заметим, что, в отличие от алгоритма Deep Q-Network, здесь выходом стратегии являются не ценности действий $Q(s, a)$.

Затем каждое выходное значение преобразуется в вероятность. Для этого используется функция `softmax`, вычисляемая следующим образом:

$$\pi_{\theta}(a|s) = \frac{e^{z(s,a)}}{\sum_i e^{z(s,a_i)}}.$$

Значения `softmax` нормируются, так чтобы сумма была равна 1, т. е. получилось распределение вероятностей, и, следовательно, каждое значение равно вероятности выбора данного действия в состоянии s .

На рис. 6.2 показаны ценности действий, предсказанные до (диаграмма слева) и после (диаграмма справа) применения функции `softmax`. На правой диаграмме хорошо видно, что после применения `softmax` сумма значений равна 1, и при этом все значения больше 0.

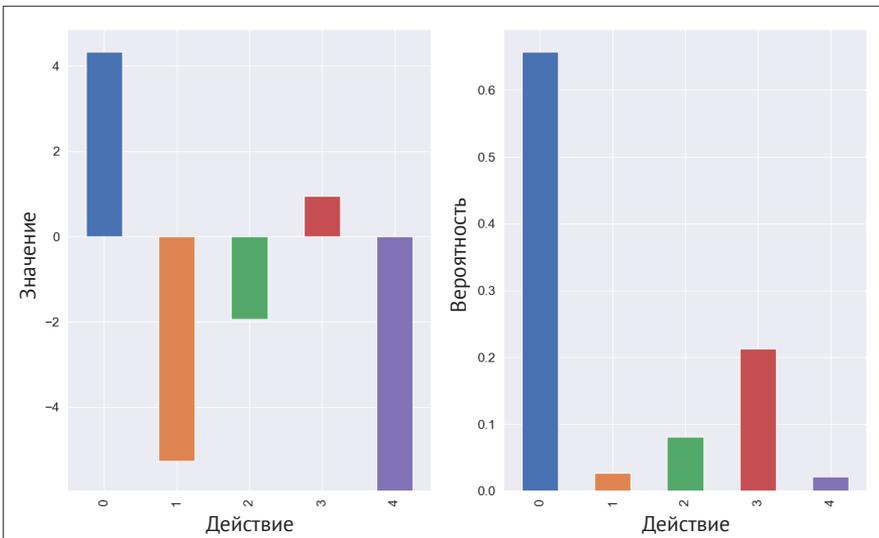


Рис. 6.2

Из правой диаграммы следует, что действия 0, 1, 2, 3, 4 будут выбраны с вероятностями, приблизительно равными 0.64, 0.02, 0.09, 0.21, 0.02 соответственно.

Чтобы использовать softmax-распределение ценностей действий, возвращаемых параметрической стратегией, мы можем взять код из раздела «Вычисление градиента» и внести в него всего одно изменение, выделенное ниже полужирным шрифтом:

```
pi = policy(states) # вероятности действий
onehot_action = tf.one_hot(actions, depth=env.action_space.n)

# вместо tf.math.log(pi)
pi_log = tf.reduce_sum(onehot_action * tf.nn.log_softmax(pi), axis=1)

pi_loss = -tf.reduce_mean(pi_log * Q_function(states, actions))
gradients = tf.gradient(pi_loss, variables)
```

Здесь мы воспользовались методом `tf.nn.log_softmax`, потому что так алгоритм получается более устойчивым, чем при вызове сначала `tf.nn.softmax`, а затем `tf.math.log`.

Преимущество стохастического распределения действия в том, что выбор действий становится принципиально случайным, что обеспечивает динамическое исследование окружающей среды. Это может показаться побочным эффектом, но вообще-то очень хорошо иметь стратегию, которая сама адаптирует уровень исследования.

В случае DQN для регулировки исследования на протяжении всего обучения нам приходилось задавать специальный параметр с линейным затуханием. Теперь же, когда исследование встроено в стратегию, нам нужно лишь подтолкнуть его, добавив один член (энтропию) в функцию потерь.

Алгоритм ГС с единой стратегией

Очень важная особенность алгоритмов градиента стратегии заключается в том, что в них используется *единая стратегия*. Это следует из формулы (6.4), в которой присутствует зависимость от текущей стратегии. В отличие от алгоритмов с разделенной стратегией, в частности DQN, в алгоритмах с единой стратегией запрещено использовать прошлый опыт.

Это означает, что весь опыт, накопленный при следовании данной стратегии, приходится отбрасывать, когда стратегия изменяется. Из-за этого у алгоритмов градиента стратегии ниже выборочная эффективность, т. е. для достижения такого же качества, как у аналогов с разделенной стратегией, им нужно больше опыта. И обобщаются они, как правило, несколько хуже.

УСТРОЙСТВО АЛГОРИТМА REINFORCE

Идею алгоритмов градиента стратегии мы уже объяснили, но осталась еще одна важная сторона. Мы пока не видели, как вычисляются ценности действий.

Вспомним формулу (6.4):

$$\nabla_{\theta} J(\theta) = E_{\pi_{\theta}} [\nabla_{\theta} \log \pi_{\theta}(a|s) Q_{\pi_{\theta}}(s, a)], \quad (6.4)$$

с помощью которой мы можем оценить градиент целевой функции, производя выборку непосредственно из опыта, накопленного в результате следования стратегии.

В ней всего два компонента: ценности $Q_{\pi_\theta}(s, a)$ и производная логарифма стратегии, которую позволяют вычислить современные библиотеки глубокого обучения (в т. ч. TensorFlow и PyTorch). Мы определили π_θ , но еще не объяснили, как оценить функцию ценности действий.

Простой способ, впервые описанный Уильямсом для алгоритма REINFORCE, – оценить доход с помощью метода **Монте-Карло (МК)**. Поэтому REINFORCE принято относить к алгоритмам Монте-Карло. Напомним, что доход МК вычисляется по выборочным траекториям при следовании заданной стратегии. Перепишем (6.4), заменив функцию ценности действий Q доходом МК G :

$$\begin{aligned}\nabla_\theta J(\theta) &= E_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) Q_{\pi_\theta}(s, a)] \\ &= E_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a_t|s_t) G_t].\end{aligned}\tag{6.6}$$

Доход G_t вычисляется по полной траектории, т. е. произвести обновление ГС можно будет только после $T - t$ шагов, где T – полное число шагов вдоль траектории. Другое следствие состоит в том, что доход МК корректно определен только в эпизодических задачах, где существует верхняя граница числа шагов (к этому заключению мы пришли еще раньше, когда изучали другие алгоритмы МК).

На практике применяется обесцененный доход в момент t , который можно также назвать *предстоящим вознаграждением* (reward to go), поскольку в нем встречаются только вознаграждения в будущие моменты времени:

$$G_t = \sum_{t'=t}^T \gamma^{t'-t} r(s_{t'}, a_{t'}).$$

Эту формулу можно переписать в виде рекуррентного соотношения:

$$G(s_t, a_t) = r(s_t, a_t) + \lambda G(s_{t+1}, a_{t+1}).$$

Реализовать ее можно, производя обработку в обратном порядке, начиная с последнего вознаграждения:

```
def discounted_rewards(rews, gamma):
    rtg = np.zeros_like(rews, dtype=np.float32)
    rtg[-1] = rews[-1]
    for i in reversed(range(len(rews)-1)):
        rtg[i] = rews[i] + gamma*rtg[i+1]
    return rtg
```

Здесь мы первым делом создаем массив NumPy и присваиваем величину последнего вознаграждения переменной `rtg`. Так делается, потому что в момент времени T $G(s_T, a_T) = r(s_T, a_T)$. Затем алгоритм вычисляет `rtg[i]`, двигаясь назад.

В главном цикле алгоритма REINFORCE выполняется несколько эпох, пока не наберется достаточно опыта, и оптимизируются параметры стратегии. Для эффективности алгоритм должен завершить хотя бы одну эпоху, прежде чем

выполнять шаг обновления (ему нужна по крайней мере одна траектория, чтобы вычислить предстоящее вознаграждение G_t). Ниже приведен псевдокод алгоритма REINFORCE:

Инициализировать π_θ со случайными весами

for эпизод 1..M **do**

Инициализировать окружающую среду $s \leftarrow env.reset()$

Инициализировать пустой буфер

> Сгенерировать несколько эпизодов

for step 1..MaxSteps **do**

> Получать опыт, взаимодействуя со средой

$a \leftarrow \pi_\theta(s)$

$s', r, d \leftarrow env(a)$

$s \leftarrow s'$

if $d == True$:

$s \leftarrow env.reset()$

> Вычислить предстоящее вознаграждение

$G(s_t, a_t) = r(s_t, a_t) + \lambda G(s_{t+1}, a_{t+1})$ # для каждого t

> Сохранить эпизод в буфере

$D \leftarrow D \cup (s_{1..T}, a_{1..T}, G_{1..T})$ # где T – длина эпизода

> шаг обновления REINFORCE с использованием всего опыта по формуле (6.5)

$$\theta \leftarrow \theta + \alpha \frac{1}{|D|} \sum_i [\nabla_\theta \log \pi_\theta(a_i | s_i) G_i^{\pi_\theta}]$$

Реализация REINFORCE

Пришло время реализовать алгоритм REINFORCE. Здесь мы ограничимся простым кодом без процедур отладки и мониторинга. Полная реализация имеется в репозитории на GitHub.

Код содержит три основные функции и один класс:

- REINFORCE(env_name, hidden_sizes, lr, num_epochs, gamma, steps_per_epoch): собственно реализация алгоритма;
- Buffer: этот класс используется для временного хранения траекторий;
- mlp(x, hidden_layer, output_size, activation, last_activation): используется для построения многослойного перцептрона в TensorFlow;
- discounted_rewards(rewards, gamma): вычисляет обесцененное предстоящее вознаграждение.

Сначала рассмотрим главную функцию REINFORCE, а потом займемся вспомогательными функциями и классом.

Функция REINFORCE состоит из двух частей. В первой части создается граф вычислений, а во второй циклически производится взаимодействие со средой и оптимизация стратегии, пока не будет выполнен критерий сходимости.

Функция REINFORCE принимает имя окружающей среды env_name, список размеров скрытых слоев hidden_sizes, скорость обучения lr, количество эпох обучения num_epochs, коэффициент обесценивания gamma и минимальное число шагов в эпохе steps_per_epoch. Формально сигнатура REINFORCE имеет вид:

```
def REINFORCE(env_name, hidden_sizes=[32], lr=5e-3, num_epochs=50,
             gamma=0.99, steps_per_epoch=100):
```

В начале REINFORCE() сбрасывается граф TensorFlow по умолчанию, создается окружающая среда, инициализируется местозаменитель и создается стратегия. Стратегия представлена полносвязным многослойным перцептроном, содержащим по одному выходу на каждое действие, с функцией активации \tanh в каждом скрытом слое. Выходами перцептрона являются ненормированные ценности действий, называемые логитами. Все это делается в следующем фрагменте:

```
def REINFORCE(env_name, hidden_sizes=[32], lr=5e-3, num_epochs=50,
             gamma=0.99, steps_per_epoch=100):

    tf.reset_default_graph()

    env = gym.make(env_name)
    obs_dim = env.observation_space.shape
    act_dim = env.action_space.n

    obs_ph = tf.placeholder(shape=(None, obs_dim[0]), dtype=tf.float32, name='obs')
    act_ph = tf.placeholder(shape=(None,), dtype=tf.int32, name='act')
    ret_ph = tf.placeholder(shape=(None,), dtype=tf.float32, name='ret')
    p_logits = mlp(obs_ph, hidden_sizes, act_dim, activation=tf.tanh)
```

Теперь можно создать операции для вычисления функции потерь и оптимизации стратегии. Этот код похож на тот, который мы уже видели в разделе «Стратегия». Единственная разница в том, что выборка действий производится методом `tf.random.multinomial` из категориального распределения действий, возвращаемого стратегией. В нашем случае эта функция выбирает всего одно действие (в зависимости от окружающей среды действий может быть и больше).

Ниже приведена реализация обновления в алгоритме REINFORCE:

```
act_multn = tf.squeeze(tf.random.multinomial(p_logits, 1))
actions_mask = tf.one_hot(act_ph, depth=act_dim)
p_log = tf.reduce_sum(actions_mask * tf.nn.log_softmax(p_logits), axis=1)
p_loss = -tf.reduce_mean(p_log*ret_ph)
p_opt = tf.train.AdamOptimizer(lr).minimize(p_loss)
```

Для действий, выбранных в процессе взаимодействия с окружающей средой, создается маска и умножается на `log_softmax`, чтобы получить $\log \pi_{\theta}(a|s)$. Затем вычисляется функция потерь. Будьте внимательны – перед `tf.reduce_mean` стоит знак минус. Мы хотим максимизировать целевую функцию. Но поскольку оптимизатору нужна подлежащая минимизации функция, мы должны передать функцию потерь. В последней строчке функция потерь ГС оптимизируется алгоритмом `AdamOptimizer`.

Теперь можно начать сеанс, сбросить глобальные переменные графа вычислений и инициализировать некоторые переменные, которые понадобятся в дальнейшем.

```
sess = tf.Session()
sess.run(tf.global_variables_initializer())
step_count = 0
train_rewards = []
train_ep_len = []
```

Далее создаем два вложенных цикла, которые во взаимодействии со средой накапливают опыт и оптимизируют стратегию, и печатаем статистические показатели:

```
for ep in range(num_epochs):
    obs = env.reset()
    buffer = Buffer(gamma)
    env_buf = []
    ep_rews = []

    while len(buffer) < steps_per_epoch:

        # выполнить стратегию
        act = sess.run(act_multn, feed_dict={obs_ph:[obs]})
        # один шаг взаимодействия со средой
        obs2, rew, done, _ = env.step(np.squeeze(act))

        env_buf.append([obs.copy(), rew, act])
        obs = obs2.copy()
        step_count += 1
        ep_rews.append(rew)

        if done:
            # сохранить в буфере полную траекторию
            buffer.store(np.array(env_buf))
            env_buf = []
            train_rewards.append(np.sum(ep_rews))
            train_ep_len.append(len(ep_rews))
            obs = env.reset()
            ep_rews = []

    obs_batch, act_batch, ret_batch = buffer.get_batch()
    # Оптимизация стратегии
    sess.run(p_opt, feed_dict={obs_ph:obs_batch, act_ph:act_batch,
ret_ph:ret_batch})

    # Напечатать статистику
    if ep % 10 == 0:
        print('Ep:%d MnRew:%.2f MxRew:%.1f EpLen:%.1f Buffer:%d --
Step:%d --' % (ep, np.mean(train_rewards), np.max(train_rewards),
np.mean(train_ep_len), len(buffer), step_count))

    train_rewards = []
    train_ep_len = []
    env.close()
```

Поток управления в обоих циклах обычный с одним исключением – взаимодействие с окружающей средой прекращается, как только траектория завершается и во временном буфере оказывается достаточно переходов.

Теперь реализуем класс Buffer, содержащий данные о траекториях:

```
class Buffer():
    def __init__(self, gamma=0.99):
        self.gamma = gamma
        self.obs = []
        self.act = []
        self.ret = []
```

```

def store(self, temp_traj):
    if len(temp_traj) > 0:
        self.obs.extend(temp_traj[:,0])
        ret = discounted_rewards(temp_traj[:,1], self.gamma)
        self.ret.extend(ret)
        self.act.extend(temp_traj[:,2])

def get_batch(self):
    return self.obs, self.act, self.ret

def __len__(self):
    assert(len(self.obs) == len(self.act) == len(self.ret))
    return len(self.obs)

```

И наконец, напишем функцию, которая создает нейронную сеть с произвольным числом скрытых слоев.

```

def mlp(x, hidden_layers, output_size, activation=tf.nn.relu, last_activation=None):
    for l in hidden_layers:
        x = tf.layers.dense(x, units=l, activation=activation)
    return tf.layers.dense(x, units=output_size, activation=last_activation)

```

Здесь `activation` – нелинейная функция, применяемая к скрытым слоям, а `last_activation` – нелинейная функция, применяемая к выходному слою.

Посадка космического корабля с помощью алгоритма REINFORCE

Алгоритм-то мы реализовали, но самое интересное еще впереди. В этом разделе мы применим REINFORCE к эпизодической окружающей среде `Gym LunarLander-v2`, которая предназначена для посадки космического корабля на Луну.

На рис. 6.3 показано начальное положение в игре и желаемое конечное положение.



Рис. 6.3

Это дискретная задача – корабль должен сесть в точке с координатами $(0,0)$, а удаление от нее штрафует. За продвижение от верхнего края экрана к нижнему начисляется положительное вознаграждение, но за включение двигателя для торможения начисляется штраф 0.3 балла в каждом кадре.

В зависимости от мягкости посадки начисляются дополнительные -100 или $+100$ баллов. Игра считается успешно пройденной, если набрано 200 баллов. Максимальное количество шагов равно 1000.

В силу последней причины мы должны будем накапливать опыт по крайней мере в течение 1000 шагов, чтобы пройти хотя бы один эпизод до конца (это значение задается гиперпараметром `steps_per_epoch`).

Мы вызываем функцию REINFORCE со следующими гиперпараметрами:

```
REINFORCE('LunarLander-v2', hidden_sizes=[64], lr=8e-3, gamma=0.99,
num_epochs=1000, steps_per_epoch=1000)
```

Анализ результатов

Чтобы лучше прочувствовать алгоритм и правильно настроить гиперпараметры, в процессе обучения отслеживалось много параметров, в т. ч. `p_loss` (потеря при следовании стратегии), `old_p_loss` (потеря до этапа оптимизации), полное вознаграждение и длина эпизодов. Мы также посмотрели несколько гистограмм. Как именно строились гистограммы в TensorBoard, вы можете узнать, заглянув в код, имеющийся в репозитории этой книги.

На рис. 6.4 показан график полного вознаграждения, усредненного по всем полным траекториям, собранным в процессе обучения.

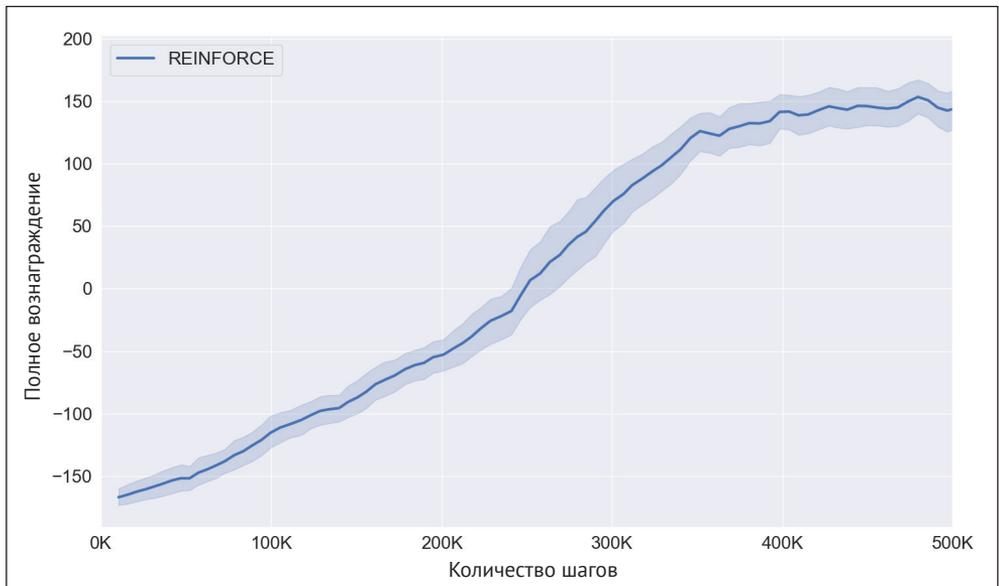


Рис. 6.4

По графику видно, что алгоритм достигает среднего количества баллов 200 или чуть меньше примерно за 500 000 шагов, т. е. чтобы научиться хорошо играть, ему нужно приблизительно 1000 полных траекторий.

При построении графика качества следует помнить, что алгоритм, скорее всего, продолжает исследование. Чтобы проверить, так ли это, нужно после-

доть за энтропией действий. Если она больше 0, значит, алгоритм не уверен, какое действие выбрать, и будет продолжать исследование – выбирать другие действия и следовать за их распределением. В данном случае после 500 000 шагов агент все еще продолжает исследование окружающей среды, как показывает график на рис. 6.5.

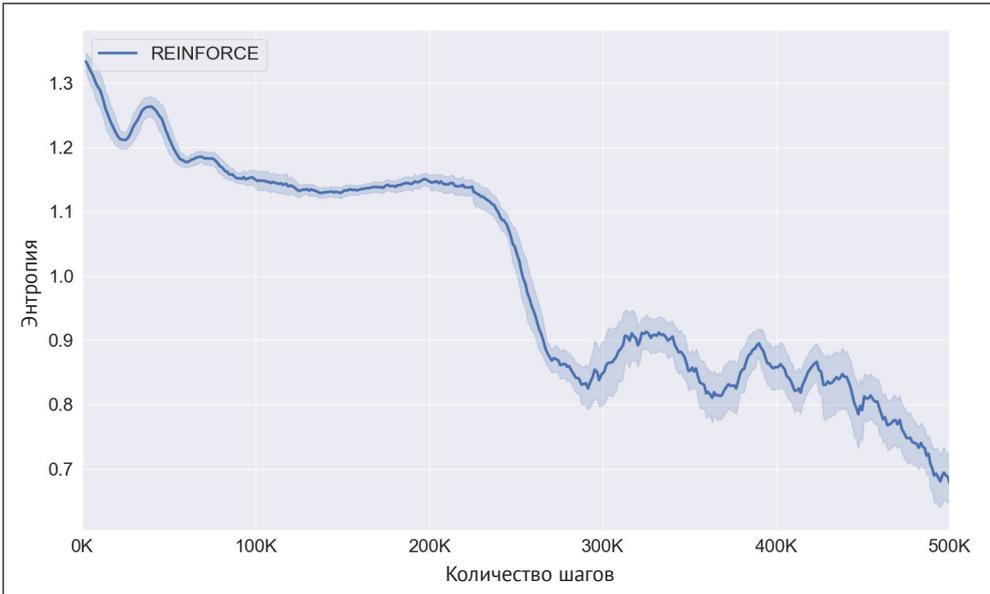


Рис. 6.5

REINFORCE с базой

У алгоритма REINFORCE есть приятное свойство – он несмещенный, поскольку доход Монте-Карло – это истинный доход на всей траектории. Однако несмещенность оценки вступает в конфликт с дисперсией, которая увеличивается по мере роста длины траектории. Почему? Из-за стохастической природы стратегии. Пройдя траекторию до конца, мы будем знать истинное вознаграждение. Но ценность, сопоставленная каждой паре состояние–действие, может быть неправильной, поскольку стратегия стохастическая, и повторное ее выполнение может завершиться другим состоянием, а следовательно, и другим вознаграждением. И чем больше количество действий в траектории, тем большая стохастичность вносится в систему, что приводит к более высокой дисперсии.

По счастью, можно включить в оценку дохода базовый уровень b и тем уменьшить дисперсию и повысить устойчивость и качество алгоритма. Алгоритмы, которые следуют этой стратегии, называются **REINFORCE с базой**, градиент целевой функции в них имеет вид

$$\nabla_{\theta} J(\theta) = E_{\pi_{\theta}}[\nabla_{\theta} \log \pi_{\theta}(a_t | s_t)(G_t - b)].$$

Этот трюк возможен, потому смещение градиентной оценки остается неизменным:

$$E[\nabla_{\theta} \log \pi_{\theta}(\tau)b] = 0.$$

Но чтобы это равенство было справедливо, базовый уровень должен быть постоянным для всех действий.

Теперь наша задача – найти подходящий базовый уровень. Проще всего вычесть средний доход:

$$b = \frac{1}{N} \sum_{n=0}^N G_n.$$

Чтобы включить это изменение в код REINFORCE, нужно только модифицировать функцию `get_batch()` в классе `Buffer`:

```
def get_batch(self):
    b_ret = self.ret - np.mean(self.ret)
    return self.obs, self.act, b_ret
```

Но хотя такая база уменьшает дисперсию, это не лучшая стратегия. Поскольку база может быть обусловлена состоянием, было бы лучше использовать оценку функции ценности:

$$\nabla_{\theta} J(\theta) = E_{\pi_{\theta}} [\nabla_{\theta} \log \pi_{\theta}(a_t | s_t)(G_t - V^{\pi_{\theta}}(s_t))].$$

Напомним, что функция ценности $V^{\pi_{\theta}}$ в среднем равна доходу, полученному при следовании стратегии π_{θ} .

При таком варианте сложность системы увеличивается, поскольку приходится придумывать, как аппроксимировать функцию ценности, но все равно этот подход применяется часто и заметно повышает качество алгоритма.

Для обучения $V^{\pi_{\theta}}(s)$ лучше всего взять нейронную сеть с оценками Монте-Карло:

$$V_w^{\pi_{\theta}}(s) = \sum_{t'=t}^T \gamma^{t'-t} r(s_{t'}, a_{t'}).$$

Здесь w – вектор параметров нейронной сети, подлежащей обучению.

Чтобы упростить обозначения, мы далее будем опускать стратегию, так что $V_w^{\pi_{\theta}}(s)$ сводится к $V_w(s)$.

Нейронная сеть обучается на данных тех же траекторий, что используют для обучения π_{θ} , дополнительное взаимодействие с окружающей средой не нужно. Оценки МК, вычисленные, например, с помощью функции `discounted_rewards(rews, gamma)`, становятся целевыми значениями, и нейронная сеть обучается, так чтобы минимизировать среднеквадратическую ошибку (СКО) – как при обучении с учителем:

$$\mathcal{L}(w) = \frac{1}{2} \sum_i (V_w(s_i) - y_i)^2.$$

Здесь w – веса нейронной сети, аппроксимирующей функцию ценности, а каждый элемент набора данных содержит состояние s_i и целевое значение

$$y_i = \sum_{t'=t}^T \gamma^{t'-t} r(s_{t'}, a_{t'}).$$

Реализация REINFORCE с базой

Функцию ценности, которую алгоритм REINFORCE с базой аппроксимирует нейронной сетью, можно реализовать, добавив всего несколько строчек к ранее написанному коду.

1. Добавить в граф вычислений нейронную сеть, операцию вычисления функции потерь, равной среднеквадратической ошибке, и процедуру оптимизации:

```
...
# место заместитель для предстоящего вознаграждения (т. е. значений y)
rtg_ph = tf.placeholder(shape=(None,), dtype=tf.float32, name='rtg')
# функция ценности, аппроксимируемая многослойным перцептроном
s_values = tf.squeeze(mlp(obs_ph, hidden_sizes, 1, activation=tf.tanh))

# функция потерь - СКО
v_loss = tf.reduce_mean((rtg_ph - s_values)**2)

# оптимизация функции ценности
v_opt = tf.train.AdamOptimizer(vf_lr).minimize(v_loss)
...
```

2. Обучить сеть `s_values` и сохранить предсказания $V_w(s_i)$, поскольку впоследствии нам нужно будет вычислять $(G_i - V_w(s_i))$. Эту операцию можно выполнить во внутреннем цикле (отличия от кода REINFORCE выделены полужирным шрифтом):

```
...
# помимо act_multn, прогнать также s_values
act, val = sess.run([act_multn, s_values], feed_dict={obs_ph:[obs]})
obs2, rew, done, _ = env.step(np.squeeze(act))

# добавить новый переход, включив предсказания ценности состояния
env_buf.append([obs.copy(), rew, act, np.squeeze(val)])
...
```

3. Извлечь из буфера пакет `rtg_batch`, содержащий целевые значения, и оптимизировать функцию ценности:

```
obs_batch, act_batch, ret_batch, rtg_batch = buffer.get_batch()
sess.run([p_opt, v_opt], feed_dict={obs_ph:obs_batch,
act_ph:act_batch, ret_ph:ret_batch, rtg_ph:rtg_batch})
```

4. Вычислить предстоящее вознаграждение (G_i) и целевые значения ($G_i - V_w(s_i)$). Это изменение вносится в класс `Buffer`. Мы должны будем создать пустой список `self.rtg` в методе инициализации класса и следующим образом модифицировать методы `store` и `get_batch`:

```
def store(self, temp_traj):
    if len(temp_traj) > 0:
        self.obs.extend(temp_traj[:,0])
```

```

rtg = discounted_rewards(temp_traj[:,1], self.gamma)
# ret = G - V
self.ret.extend(rtg - temp_traj[:,3])
self.rtg.extend(rtg)
self.act.extend(temp_traj[:,2])

def get_batch(self):
    return self.obs, self.act, self.ret, self.rtg

```

Теперь можно протестировать алгоритм REINFORCE с базой на любой окружающей среде и сравнить результаты с обычным REINFORCE.

ОБУЧЕНИЕ АЛГОРИТМА ИСПОЛНИТЕЛЬ–КРИТИК

Простой алгоритм REINFORCE обладает свойством несмещенности, но также и высокой дисперсией. Добавление базы позволяет уменьшить дисперсию, сохранив несмещенность (асимптотически алгоритм будет сходиться к локальному минимуму). Главный недостаток алгоритма REINFORCE с базой – очень медленная сходимость, т. е. большое число взаимодействий со средой.

Для ускорения обучения применяют бутстрэппинг. Мы уже неоднократно встречались с этим приемом на страницах данной книги. Он позволяет оценить доход по ценностям последующих состояний. Алгоритмы градиента стратегии, в которых используется эта техника, называются алгоритмами исполнитель–критик (actor-critic – AC). В алгоритме AC исполнителем является стратегия, а критиком – функция ценности (обычно функция ценности состояний), которая «критикует» поведение исполнителя, чтобы помочь ему быстрее обучиться. У методов AC много достоинств, но самое главное – способность к обучению в неэпизодических задачах.

Алгоритм REINFORCE непригоден для решения непрерывных задач, поскольку для вычисления предстоящего вознаграждения ему нужно знать все вознаграждения до конца траектории (а если траектория бесконечна, то у нее нет конца). Методы же AC, опирающиеся на технику бутстрэппинга, могут обучиться ценностям действий даже по неполным траекториям.

Как критик помогает обучаться исполнителю

Функция ценности действий с одношаговым бутстрэппингом определяется следующим образом:

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

где s' – следующее состояние.

Таким образом, если π_θ – исполнитель, а V_w – критик, то получается такой шаг обновления в одношаговом алгоритме AC:

$$\theta = \theta + \alpha(r_t + \gamma V_w(s'_t) - V_w(s)) \nabla_\theta \log \pi_\theta(a_t | s_t)$$

вместо обновления в алгоритме REINFORCE с базой:

$$\theta = \theta + \alpha(G_t - V_w(s)) \nabla_\theta \log \pi_\theta(a_t | s_t).$$

Обратите внимание на различие в использовании функции ценности состояний в REINFORCE и AC. В первом случае она используется только как базовый уровень и определяет ценность текущего состояния. Во втором случае функция ценности состояний используется для оценки ценности следующего состояния, и для оценки $Q(s, a)$ нужно знать только текущее вознаграждение. Таким образом, можно сказать, что одношаговый AC – полностью онлайнный инкрементный алгоритм.

***n*-шаговая модель AC**

На практике, как мы видели при рассмотрении TD-обучения, полностью онлайнный алгоритм обладает низкой дисперсией, но высоким смещением – и в этом смысле является противоположностью обучению методом Монте-Карло. Поэтому обычно предпочитают нечто среднее между тем и другим. Для этого одношаговый доход в онлайнных алгоритмах можно заменить n -шаговым.

Напомним, что мы уже реализовали n -шаговое обучение в алгоритме DQN. Единственное различие заключается в том, что DQN – алгоритм с разделенной стратегией, но теоретически n -шаговым может быть и алгоритм с единой стратегией. Мы показали, что при небольшом n качество улучшается.

Алгоритмы AC являются алгоритмами с единой стратегией, поэтому, с точки зрения улучшения качества, можно использовать сколь угодно большие значения n . Включить n шагов в AC довольно просто – одношаговый доход заменяется на $G_{t:t+n}$, а функция ценности вычисляется для состояния s_{t+n} :

$$\theta = \theta + \alpha(G_{t:t+n} + \gamma^n V_w(s_{t+n}) - V_w(s_t)) \nabla_{\theta} \log \pi_{\theta}(a_t | s_t).$$

Здесь $G_{t:t+n} = r_t + \gamma r_{t+1} + \dots + \gamma^{n-1} r_{t+n-1}$. Обратите внимание, что если s_t – заключительное состояние, то $V(s_{t+1}) = 0$.

Помимо уменьшения смещения, распространение последующих доходов в n -шаговом варианте оказывается быстрее, так что обучение более эффективно.

Интересно, что величину $G_{t:t+n} + \gamma^n V_w(s_{t+n}) - V_w(s_t)$ можно рассматривать как оценку функции преимущества. Напомним определение функции преимущества:

$$A(a_t | s_t) = Q(a_t | s_t) - V(s_t).$$

Поскольку $G_{t:t+n} + \gamma^n V_w(s_{t+n})$ является оценкой $Q_w(s_t, a_t)$, мы получаем оценку функции преимущества. Обычно эту функцию проще обучить, т. к. она выражает лишь предпочтение одному конкретному действию перед всеми остальными действиями в данном состоянии. Знать ценность этого состояния необязательно.

Что касается весов критика, то они оптимизируются с помощью какого-либо варианта метода СГС, минимизирующего среднеквадратическую ошибку

$$\mathcal{L}(w) = \frac{1}{2} \sum_i (V_w(s_i) - y_i)^2.$$

Целевые значения в этом случае вычисляются как $y_i = G_{t:t+n} + \gamma^n V_w(s_{t+n})$.

Реализация AC

Как мы видели, алгоритм AC очень похож на REINFORCE, в котором в качестве базы используется функция ценности состояний. Но для полноты картины приведем псевдокод.

Инициализировать π_θ со случайными весами

Инициализировать окружающую среду $s \leftarrow env.reset()$

for эпизод 1..M **do**

 Инициализировать пустой буфер

 > Сгенерировать несколько эпизодов

for step 1..MaxSteps **do**

 > Получать опыт, взаимодействуя со средой

$a \leftarrow \pi_\theta(s)$

$s', r, d \leftarrow env(a)$

$s \leftarrow s'$

if $d == True$:

$s \leftarrow env.reset()$

 > Вычислить предстоящее n -шаговое вознаграждение

$G_t = G_{t:t+n} + \gamma^n V_w(s_{t+n})$ # для каждого t

 > Вычислить функцию преимущества

$A_t = G_t - V_w(s_t)$ # для каждого t

 > Сохранить эпизод в буфере

$D \leftarrow D \cup (s_{1..T}, a_{1..T}, G_{1..T}, A_{1..T})$ # где T – длина эпизода

 > Шаг обновления исполнителя с использованием всего опыта в D

$\theta \leftarrow \theta + \alpha \frac{1}{|D|} \sum_i [\nabla_\theta \log \pi_\theta(a_i | s_i) A_i]$

 > Шаг обновления критика с использованием всего опыта в D

$w \leftarrow w + \alpha_w \frac{1}{|D|} \sum_i (V_w(s_i) - G_i)^2$

От REINFORCE этот алгоритм отличается вычислением n -шагового предстоящего вознаграждения, вычислением функции преимущества и несколькими изменениями в главной функции. Рассмотрим новую реализацию обесцененного вознаграждения. В отличие от предыдущей версии, оценка ценности последнего состояния `last_sv` теперь передается на вход и используется для бутстрэппинга:

```
def discounted_rewards(rews, last_sv, gamma):
```

```
    rtg = np.zeros_like(rews, dtype=np.float32)
```

```
    rtg[-1] = rews[-1] + gamma*last_sv # бутстрэппинг с оценкой ценности следующего состояния
```

```
    for i in reversed(range(len(rews)-1)):
```

```
        rtg[i] = rews[i] + gamma*rtg[i+1]
```

```
    return rtg
```

Граф вычислений не изменяется, но в главный цикл нужно внести несколько мелких, но очень важных изменений.

То, что имя функции заменено на AC, а скорость обучения критика `cg_lr` добавлена в качестве аргумента, не вызывает вопросов.

Первое реальное изменение касается сброса окружающей среды. Если в алгоритме REINFORCE мы сбрасывали среду на каждой итерации главного цикла, то в AC нужно оставить среду в том состоянии, в котором она была в конце предыдущей итерации, а сбрасывать только по достижении заключительного состояния.

Второе изменение касается бутстрэппинга функции ценности действий и вычисления вознаграждения. Напомним, что $Q(s, a) = r + \gamma V(s')$ для каждой пары состояние–действие, кроме случая, когда $V(s')$ – заключительное состояние, а в этом случае $Q(s, a) = r$. Следовательно, для бутстрэппинга нужно использовать значение 0, когда мы находимся в заключительном состоянии, $V(s')$ во всех остальных случаях. После внесения этих изменений получается такой код:

```
obs = env.reset()
ep_rews = []

for ep in range(num_epochs):
    buffer = Buffer(gamma)
    env_buf = []

    for _ in range(steps_per_env):
        act, val = sess.run([act_multn, s_values], feed_dict={obs_ph:[obs]})
        obs2, rew, done, _ = env.step(np.squeeze(act))

        env_buf.append([obs.copy(), rew, act, np.squeeze(val)])
        obs = obs2.copy()
        step_count += 1
        last_test_step += 1
        ep_rews.append(rew)

    if done:
        buffer.store(np.array(env_buf), 0)
        env_buf = []

        train_rewards.append(np.sum(ep_rews))
        train_ep_len.append(len(ep_rews))
        obs = env.reset()
        ep_rews = []

    if len(env_buf) > 0:
        last_sv = sess.run(s_values, feed_dict={obs_ph:[obs]})
        buffer.store(np.array(env_buf), last_sv)

    obs_batch, act_batch, ret_batch, rtg_batch = buffer.get_batch()
    sess.run([p_opt, v_opt], feed_dict={obs_ph:obs_batch,
act_ph:act_batch, ret_ph:ret_batch, rtg_ph:rtg_batch})
...

```

Третье изменение относится к методу store класса Buffer. Теперь мы должны иметь дело также с неполными траекториями. Выше мы видели, что оценки ценностей состояний $V(s')$ передаются в третьем аргументе функции store. Они используются для бутстрэппинга и вычисления предстоящего вознаграждения. В новой версии store мы назвали переменную, ассоциированную с ценностями состояний, last_sv и передаем ее на вход функции discounted_reward:

```
def store(self, temp_traj, last_sv):
    if len(temp_traj) > 0:
        self.obs.extend(temp_traj[:,0])
        rtg = discounted_rewards(temp_traj[:,1], last_sv, self.gamma)
        self.ret.extend(rtg - temp_traj[:,3])
        self.rtg.extend(rtg)
        self.act.extend(temp_traj[:,2])
```

Посадка космического корабля с помощью алгоритма AC

Мы применили алгоритм AC к той же окружающей среде LunarLander-v2, что использовалась при тестировании REINFORCE. Это эпизодическая игра, поэтому главные свойства AC в ней не проявляются в полной мере. Тем не менее это неплохой испытательный стенд, а вам никто не мешает взять для тестирования любую другую среду.

Мы вызываем функцию AC с такими гиперпараметрами:

```
AC('LunarLander-v2', hidden_sizes=[64], ac_lr=4e-3, cr_lr=1.5e-2,
   gamma=0.99, steps_per_epoch=100, num_epochs=8000)
```

На рис. 6.6 показан график полного вознаграждения, накопленного во всех эпохах обучения.

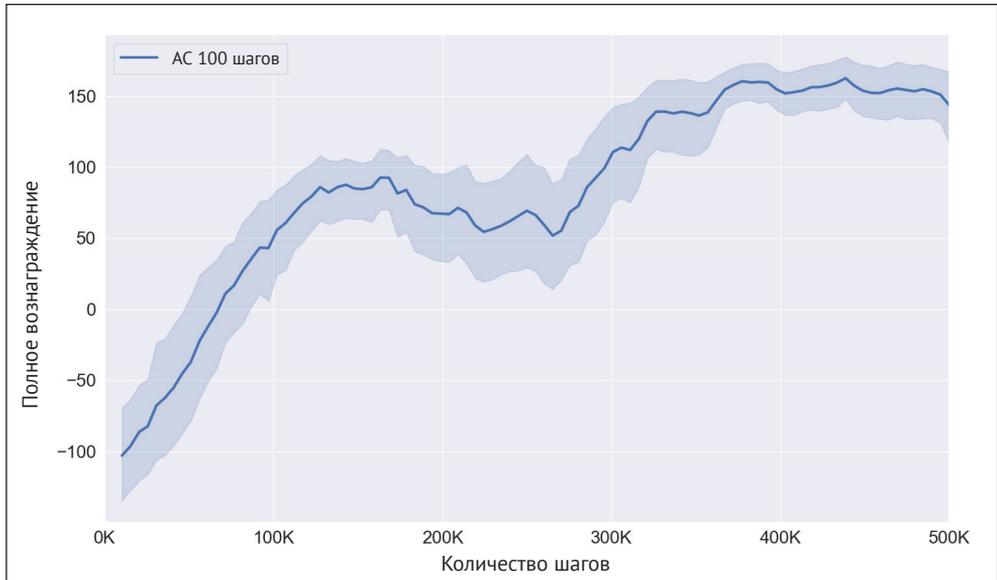


Рис. 6.6

Как видно по рис. 6.7, AC обучается быстрее REINFORCE. Однако он менее устойчив – после 200 000 шагов качество немного падает, но, к счастью, потом снова возрастает.

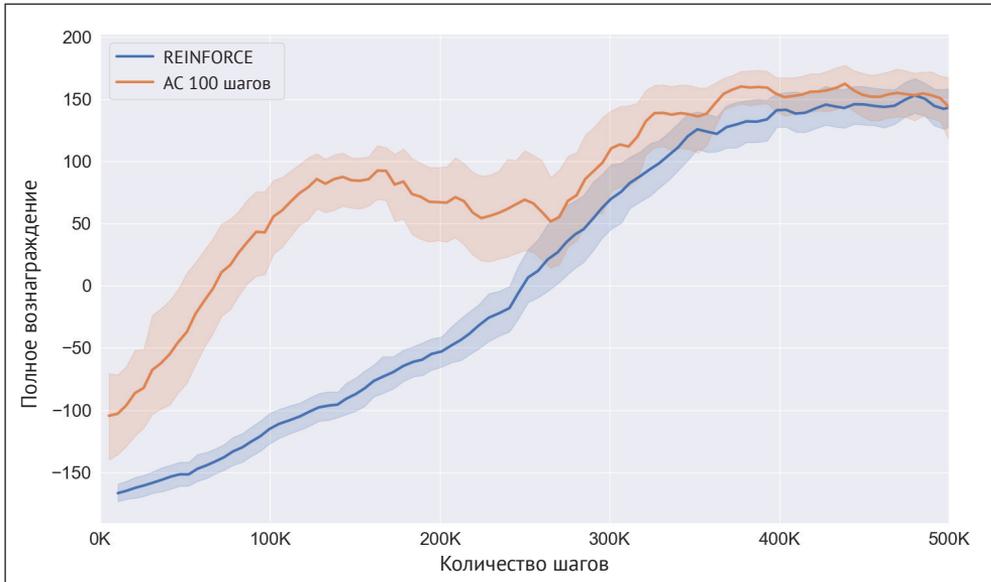


Рис. 6.7

В этой конфигурации алгоритм AC обновляет исполнителя и критика каждые 100 шагов. Теоретически можно было бы уменьшить значение `steps_per_epochs`, но на практике это обычно приводит к менее устойчивому обучению. Если, наоборот, увеличить длину эпохи, то обучение будет более устойчивым, но исполнитель обучается медленнее. Так что требуется найти хороший компромисс и подходящую скорость обучения.

Дополнительные улучшения AC и полезные советы

Существует еще несколько усовершенствований алгоритма AC, а также много приемов и хитростей, о которых стоит знать при проектировании подобных алгоритмов.

- **Проектирование архитектуры.** В нашей реализации использовано две нейронные сети: для критика и для исполнителя. Но можно спроектировать сеть, в которой основные скрытые слои общие, а различаются только выходные («головы»). Такую архитектуру труднее настроить, но в итоге она повышает эффективность алгоритма.
- **Параллельные окружающие среды.** Чтобы уменьшить дисперсию, широко практикуется накопление опыта в ходе параллельного взаимодействия с несколькими окружающими средами. В алгоритме **A3C (Asynchronous Advantage Actor-Critic)** глобальные параметры обновляются асинхронно. А его синхронная версия **A2C (Advantage Actor-Critic)** ждет завершения всех параллельных исполнителей, перед тем как приступить к обновлению глобальных параметров. Распараллеливание агентов повышает независимость опыта, полученного в результате взаимодействия с различными частями среды.

- **Размер пакета.** По сравнению с другими алгоритмами ОП (особенно с разделенной стратегией), для методов градиента стратегии и АС необходимы более крупные пакеты. Поэтому, если после настройки других гиперпараметров алгоритм не удается стабилизировать, попробуйте увеличить размер пакета.
- **Скорость обучения.** Подобрать правильную скорость обучения довольно трудно, попробуйте использовать более совершенный метод СГС-оптимизации, например Adam или RMSprop.

РЕЗЮМЕ

В этой главе мы узнали о новом классе алгоритмов обучения с подкреплением – алгоритмах градиента стратегии. Они знаменуют другой подход к решению задач ОП по сравнению с методами на основе функций ценности, рассмотренными в предыдущих главах.

Сначала мы изучили, реализовали и протестировали простой алгоритм ГС – REINFORCE, а затем добавили в него базовый уровень, чтобы уменьшить дисперсию и улучшить свойства сходимости. Алгоритмы типа исполнитель–критик не нуждаются в полной траектории. Мы применили один такой алгоритм к решению той же задачи.

Заложив прочный фундамент классических алгоритмов градиента стратегии, мы можем двигаться дальше. В следующей главе будут рассмотрены более сложные современные алгоритмы градиента стратегии, а именно **оптимизации стратегии в доверительной области** (Trust Region Policy Optimization – TRPO) и **проксимальная оптимизация стратегии** (Proximal Policy Optimization – PPO). Оба опираются на материал этой главы, но предлагают другие целевые функции, повышающие устойчивость и эффективность алгоритма.

Вопросы

1. Как алгоритмы градиента стратегии максимизируют целевую функцию?
2. В чем основная идея алгоритмов градиента стратегии?
3. Почему после включения базы в алгоритм REINFORCE он остается несмещенным?
4. К какому более широкому классу алгоритмов принадлежит REINFORCE?
5. Чем критик в методах АС отличается от функции ценности, которая используется в качестве базы в алгоритме REINFORCE?
6. Если бы вам нужно было разработать алгоритм агента, который обучается двигаться, то что бы предпочли: REINFORCE или АС?
7. Можно ли использовать n -шаговый алгоритм АС как REINFORCE?

Для дальнейшего чтения

Об асинхронной версии алгоритма исполнитель–критик можно прочитать в статье <https://arxiv.org/pdf/1602.01783.pdf>.

Глава 7

Реализация TRPO и PPO

В предыдущей главе мы рассмотрели алгоритмы градиента стратегии. Их отличительная особенность заключается в порядке решения задачи **обучения с подкреплением (ОП)** – алгоритм градиента стратегии делает шаг в направлении наибольшего прироста дохода. Простой вариант алгоритма такого типа (**REINFORCE**) имеет прямолинейную реализацию, но все равно показывает неплохие результаты. Однако он работает медленно и обладает высокой дисперсией. Поэтому мы включили функцию ценности, у которой двойная цель – критиковать исполнителя и служить базовым уровнем. Несмотря на свой большой потенциал, алгоритмы исполнитель–критик могут испытывать нежелательные резкие колебания распределения действий. Это вызывает значительное изменение набора посещенных состояний, сопровождаемое сильным падением качества, от которого алгоритм уже не может оправиться.

В данной главе мы покажем, как сгладить эту проблему с помощью доверительной области или обрезанной целевой функции. Мы опишем два применяемых на практике алгоритма: TRPO и PPO. Была продемонстрирована их способность управлять имитацией ходьбы, управлять прыгающими и плавающими роботами и играть в игры Atari. Будет рассмотрен новый набор окружающих сред для непрерывного управления и показано, как адаптировать алгоритмы градиента стратегии к работе в непрерывном пространстве действий. Применяв алгоритмы TRPO и PPO к этим средам, мы сможем обучить агента ходить, бегать и прыгать.

В этой главе рассматриваются следующие вопросы:

- Roboschool;
- метод естественного градиента стратегии;
- оптимизация стратегии в доверительной области;
- проксимальная оптимизация стратегии.

ROBOSCHOOL

До сих пор мы работали с дискретными задачами управления, в т. ч. играми Atari в главе 5 и игрой LunarLander в главе 6. В них было всего несколько дискретных действий – от двух до пяти. В главе 6 мы узнали, что алгоритмы градиента стратегии легко обобщаются на непрерывное множество действий. Чтобы продемонстрировать эту возможность, мы опишем несколько дополнительных алгоритмов ГС, взаимодействующих с окружающими средами из семейства

Roboschool, а нашей целью будет управление роботом в различных ситуациях. Семейство сред Roboschool было разработано компанией OpenAI и поддерживает тот же самый интерфейс Gym, которым мы пользовались в предыдущих главах. Эти среды основаны на движке Bullet Physics Engine (который моделирует динамику мягкого и твердого тел), аналогичном хорошо известному физическому движку Mujoco. Мы выбрали семейство Roboschool, поскольку его исходный код открыт (для Mujoco необходимо приобрести лицензию) и включает ряд интересных окружающих сред.

Семейство Roboschool включает 12 сред, от простой среды Hopper (RoboschoolHopper), показанной в левой части рис. 7.1 и управляемой тремя непрерывными действиями, до более сложного андроида (RoboschoolHumanoidFlagrun), умеющего совершать 17 непрерывных действий (в правой части рис. 7.1).

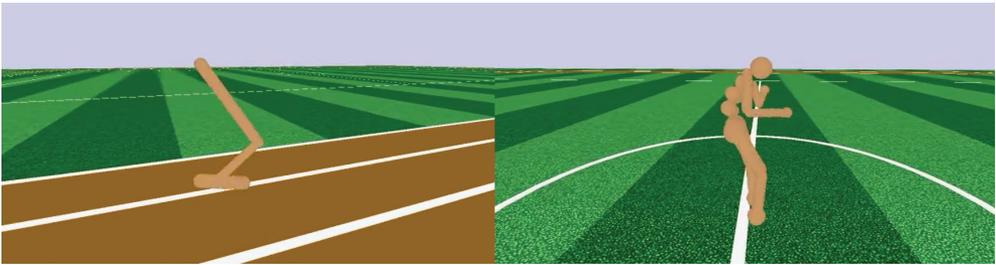


Рис. 7.1 ❖ Слева среда RoboschoolHopper-v1, справа среда RoboschoolHumanoidFlagrun-v1

В одних средах цель – научиться ходить, бегать или прыгать и как можно быстрее достичь финишной черты в 100 м, двигаясь в одном направлении, в других – двигаться по трехмерному полю, принимая во внимание внешние факторы, например летящие предметы. Включена также многопользовательская окружающая среда Pong и интерактивная среда, в которой трехмерный андроид может перемещаться в любом направлении и должен, двигаясь непрерывно, добраться до флажка. Кроме того, существует похожая среда, в которой в робота кидают кубики, чтобы он потерял равновесие, а агент должен повысить надежность управления, чтобы равновесие сохранить.

Все окружающие среды полностью наблюдаемы, т. е. агент имеет полную информацию о своем состоянии. Как мы уже отмечали, пространство состояний непрерывно и представлено классом Box переменного размера, зависящего от среды (от 10 до 40).

Управление непрерывной системой

Алгоритмы градиента стратегии, в т. ч. REINFORCE и AC, а также PPO и TRPO, которые будут реализованы в этой главе, могут работать как с дискретным, так и с непрерывным пространством действий. Переход от одного вида действий к другому довольно прост. Вместо того чтобы вычислять вероятность каждого действия при непрерывном управлении, мы можем описать действия с помощью параметров распределения вероятностей. Самый распространенный

подход – обучить параметры нормального (гауссова) распределения. Это очень важный класс распределений, характеризуемых средним μ и стандартным отклонением σ . На рис. 7.2 приведены примеры нормальных распределений с разными параметрами.

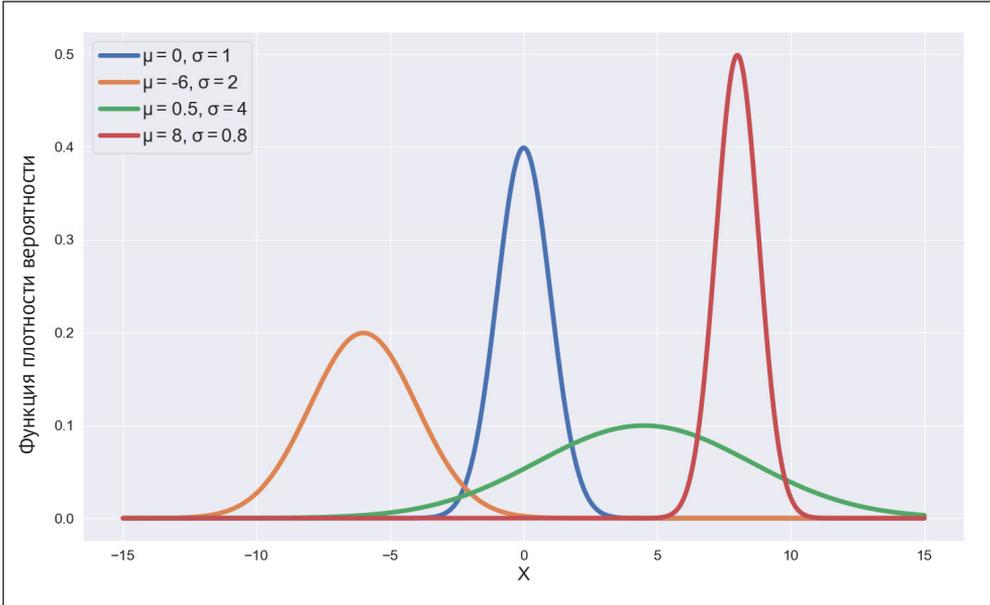


Рис. 7.2 ❖ Кривые трех нормальных распределений с разными средними и стандартными отклонениями

Например, стратегия, представленная параметрической аппроксимацией функции (скажем, нейронной сетью), может предсказать среднее и стандартное отклонение нормального распределения действий в некотором состоянии. Среднее может быть аппроксимировано линейной функцией, а стандартное отклонение обычно не зависит от состояния. В таком случае мы представляем параметрическое среднее в виде функции от состояния, обозначаемой $\mu_{\theta}(s)$, а стандартное отклонение – фиксированным значением σ . Кроме того, мы обычно имеем дело не с самим стандартным отклонением, а с его логарифмом.

Таким образом, параметрическую стратегию дискретного управления можно описать следующей строчкой кода:

```
p_logits = mlp(obs_ph, hidden_sizes, act_dim, activation=tf.nn.relu,
last_activation=None)
```

Здесь `mlp` – функция, которая создает многослойный перцептрон (полно-связную нейронную сеть), в котором размеры скрытых слоев определяются параметром `hidden_sizes`, размерность выхода равна `act_dim`, а функции активации заданы аргументами `activation` и `last_activation`. В случае стратегии непрерывного управления нужно внести следующие изменения:

```
p_means = mlp(obs_ph, hidden_sizes, act_dim, activation=tf.tanh,
last_activation=None)
log_std = tf.get_variable(name='log_std', initializer=np.zeros(act_dim,
dtype=np.float32))
```

где p_means – это $\mu_\theta(s)$, а log_std – это $\log(\sigma)$.

Если все действия принимают значения от 0 до 1, то в качестве функции `last_activation` рекомендует брать `tanh`:

```
p_means = mlp(obs_ph, hidden_sizes, act_dim, activation=tf.tanh,
last_activation=tf.tanh)
```

Для выборки из нормального распределения с целью получения действий стандартное отклонение следует умножить на вектор шума, имеющий нормальное распределение со средним 0 и стандартным отклонением 1, и прибавить к предсказанному среднему:

$$a = \mu_\theta(s) + \sigma * z.$$

Здесь z – вектор гауссова шума, $z \sim N(0, 1)$, имеющий такую же форму, как $\mu_\theta(s)$. Для вычисления достаточно одной строчки кода:

```
p_noisy = p_means + tf.random_normal(tf.shape(p_means), 0, 1) * tf.exp(log_std)
```

Поскольку мы внесли шум, уже нельзя быть уверенным, что ценности действий по-прежнему принадлежат допустимому диапазону, поэтому нужно обрезать `p_noisy`, так чтобы ценности попадали в диапазон между минимумом и максимумом. Обрезка реализуется следующим образом:

```
act_smp = tf.clip_by_value(p_noisy, envs.action_space.low, envs.action_space.high)
```

И наконец, вычисляется логарифм вероятности по формуле:

$$\log \pi_\theta(a|s) = -\frac{1}{2} \left(|a| \log 2\pi + \frac{(a - \mu_\theta(s))^2}{\sigma^2} + 2 \log \sigma \right).$$

Для этого вызывается функция `gaussian_log_likelihood`:

```
p_log = gaussian_log_likelihood(act_ph, p_means, log_std)
```

где `gaussian_log_likelihood` определена так:

```
def gaussian_log_likelihood(x, mean, log_std):
    log_p = -0.5 * (np.log(2*np.pi) + (x-mean)**2 / (tf.exp(log_std)**2 +
1e-9) + 2*log_std)
    return tf.reduce_sum(log_p, axis=-1)
```

Вот и все. Теперь можно реализовать любой алгоритм ГС и протестировать его на различных окружающих средах с непрерывным пространством действий. В предыдущей главе мы применили алгоритмы REINFORCE и AC к игре LunarLander. Для той же игры имеется версия с непрерывным управлением – LunarLanderContinuous-v2.

Зная, как обращаться с непрерывным пространством действий, мы теперь можем расширить круг решаемых задач. Но, вообще говоря, они более трудны, а изученные до сих пор алгоритмы ГС слишком слабы и плохо приспособлены

для их решения. Поэтому далее мы рассмотрим более развитые алгоритмы ГС и начнем с метода естественного градиента стратегии.

МЕТОД ЕСТЕСТВЕННОГО ГРАДИЕНТА СТРАТЕГИИ

Алгоритмы REINFORCE и исполнитель–критик интуитивно вполне понятны и хорошо работают на задачах ОП небольшого и среднего размеров. Но в больших и сложных задачах они начинают испытывать проблемы, которые необходимо как-то решать. Вот перечень основных проблем.

- **Трудно выбрать правильный размер шага.** Это связано с присущей ОП нестационарностью, когда распределение данных непрерывно изменяется, так что обучившемуся чему-то агенту приходится затем исследовать другое пространство состояний. Нахождение скорости обучения, при которой алгоритм сохраняет устойчивость, – далеко не тривиальная проблема.
- **Неустойчивость.** Алгоритм не знает, насколько изменится стратегия. Это тоже связано с вышеупомянутой проблемой. Одно неконтролируемое обновление может привести к значительному изменению стратегии, что, в свою очередь, существенно изменит распределение действий и заведет агента в неподходящую область пространства состояний. Кроме того, если новая область пространства состояний сильно отличается от предыдущей, то на восстановление нормальной работы алгоритма может уйти много времени.
- **Низкая выборочная эффективность.** Эта проблема преследует почти все алгоритмы с единой стратегией. Задача в том, чтобы извлечь как можно больше информации из данных единой стратегии, прежде чем отбросить их.

Описанные в данной главе алгоритмы TRPO и PPO призваны решить эти проблемы. Хотя их подходы различны, теоретическая база одна, и вскоре мы объясним, в чем она состоит. Отметим сразу, что TRPO и PPO – алгоритмы градиента стратегии с единой стратегией, принадлежащие семейству безмодельных алгоритмов, как показано на диаграмме классификации (рис. 7.3).

Метод естественного градиента стратегии (Natural Policy Gradient – NPG) – один из первых алгоритмов, предложенных для решения проблемы неустойчивости, свойственной методам градиента стратегии. Для этого вносится изменение в шаг обновления стратегии, цель которого – точнее контролировать направление ее изменения. К сожалению, алгоритм предназначен только для линейных аппроксимаций функций и к глубоким нейронным сетям неприменим. Однако он лежит в основе более эффективных алгоритмов TRPO и PPO.

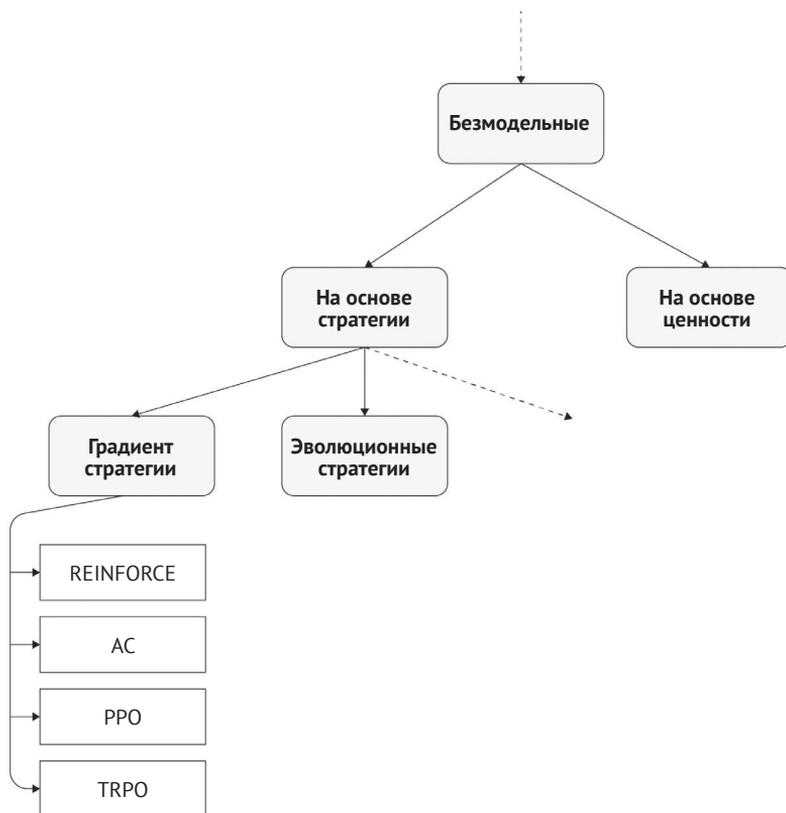


Рис. 7.3 ❖ Место TRPO и PPO на диаграмме классификации алгоритмов ОП

Интуитивное описание NPG

Прежде чем искать потенциальное решение проблемы неустойчивости методов ГС, поймем, из-за чего она появляется. Представьте, что вы поднимаетесь по крутому склону вулкана с кратером на вершине (см. поверхность на рис. 7.4). Предположим также, что вы слепы и можете оценивать окружающий мир только по углу наклона стопы (градиенту). Будем считать, что длина вашего шага (скорость обучения) фиксирована – к примеру, один метр. Вы делаете первый шаг, ощущаете наклон стопы и дальше перемещаетесь на 1 м в направлении наибольшей крутизны подъема. Повторив эту процедуру много раз, вы добираетесь до точки вблизи вершины, где находится кратер, но по-прежнему не подозреваете о его существовании – ведь вы слепы. В этой точке вы по-прежнему ощущаете, что рельеф поднимается. Но если расстояние до вершины вулкана оказалось меньше длины вашего шага, то, сделав этого шаг, вы свалитесь вниз. В этот момент пространство вокруг вас совершенно изменилось. В случае, изображенном на диаграмме, вы быстро оправитесь, поскольку функция простая, но, вообще говоря, она может быть сколь угодно сложной. Чтобы решить проблему, можно было бы уменьшить длину шага, но тогда вы станете

подниматься гораздо медленнее, а гарантии, что достигнете максимума, все равно не будет. Эта проблема свойственна не только ОП, но в этом случае она более серьезна, потому что данные не стационарны и ущерб может оказаться куда больше, чем в других контекстах, например в обучении с учителем.

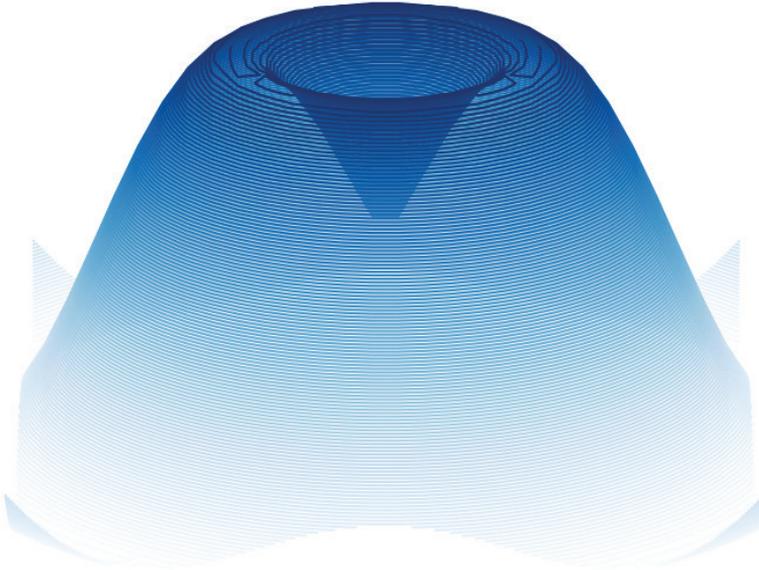


Рис. 7.4 ❖ Пытаясь достичь максимума этой функции, мы можем свалиться в кратер

Можно было бы, как, собственно, и делается в алгоритме NPG, использовать не только градиент, но и кривизну поверхности. Информацию о кривизне несет вторая производная. Она очень полезна, т. к. большая кривизна означает значительное изменение градиента на отрезке между двумя точками, поэтому в качестве меры предосторожности можно уменьшить шаг, что поможет обнаружить возможные обрывы. Таким образом, мы используем вторую производную, чтобы получить больше информации о пространстве действий и гарантировать, что в случае резкого изменения градиента распределение действий изменится не слишком сильно. В следующем разделе мы покажем, как конкретно это делается.

Немного математики

Новизна алгоритма NPG состоит в том, что на шаге обновления параметров учитывается не только первая, но и вторая производная. Чтобы разобраться в алгоритме, нужно объяснить две важные концепции: **информационную матрицу Фишера (ИМФ)** и **расхождение Кульбака–Лейблера (КЛ)**. Но сначала приведем формулу обновления:

$$\theta \leftarrow \theta + \alpha F^{-1} \nabla_{\theta} J(\theta). \quad (7.1)$$

Она отличается от обновления в обычном методе градиента стратегии только наличием члена F^{-1} , который дополняет градиент.

В этой формуле F – информационная матрица Фишера, а $J(\theta)$ – целевая функция.

Как уже было сказано, мы хотели бы, чтобы все шаги в пространстве распределений были одинаковой длины вне зависимости от градиента. Это достигается с помощью обращения ИМФ.

ИМФ и расхождение КЛ

ИМФ определяется как ковариационная матрица целевой функции. Посмотрим, чем это нам поможет. Чтобы ограничить расстояние между распределениями в нашей модели, нужно определить соответствующую метрику. Самый популярный выбор – расхождение КЛ. Оно измеряет степень различия двух распределений и используется в разных местах ОП и машинного обучения. Строго говоря, расхождение КЛ – не метрика, потому что оно не обладает свойством симметрии, но служит неплохим приближением к метрике. Чем сильнее различаются распределения, тем больше расхождение КЛ. Рассмотрим кривые на рис. 7.5. В этом примере вычислено расхождение с зеленой кривой. Поскольку оранжевая кривая похожа на зеленую, расхождение КЛ между ними равно 1.11, что довольно близко к 0. Напротив, зеленая и синяя кривые сильно различаются, и расхождение КЛ, равное 45.8, это подтверждает. Отметим, что расхождение КЛ между функцией и ей самой всегда равно 0.

❗ Для интересующихся скажем, что расхождение КЛ между двумя дискретными распределениями вероятностей вычисляется по формуле

$$D_{KL}(P \parallel Q) = - \sum_{x \in X} P(x) \log \left(\frac{Q(x)}{P(x)} \right).$$

Таким образом, расхождение КЛ позволяет сравнивать два распределения и оценивать степень их схожести. Ну и как этим воспользоваться, чтобы ограничить расстояние между соседними распределениями в стратегии?

Оказывается, что ИМФ определяет локальную кривизну в пространстве распределений, используя расхождение КЛ в качестве метрики. Стало быть, мы можем получить направление и величину шага, при котором расхождение КЛ остается постоянным, соединив кривизну (вторую производную) расхождения КЛ с градиентом (первой производной) целевой функции (как в формуле (7.1)). Поэтому обновление по формуле (7.1) будет более осторожным, поскольку в направлении наискорейшего подъема делаются малые шаги, если ИМФ велика (т. е. расстояние между распределениями действий большое), и большие шаги, если ИМФ мала (т. е. мы находимся на плато, и распределение изменяется слабо).

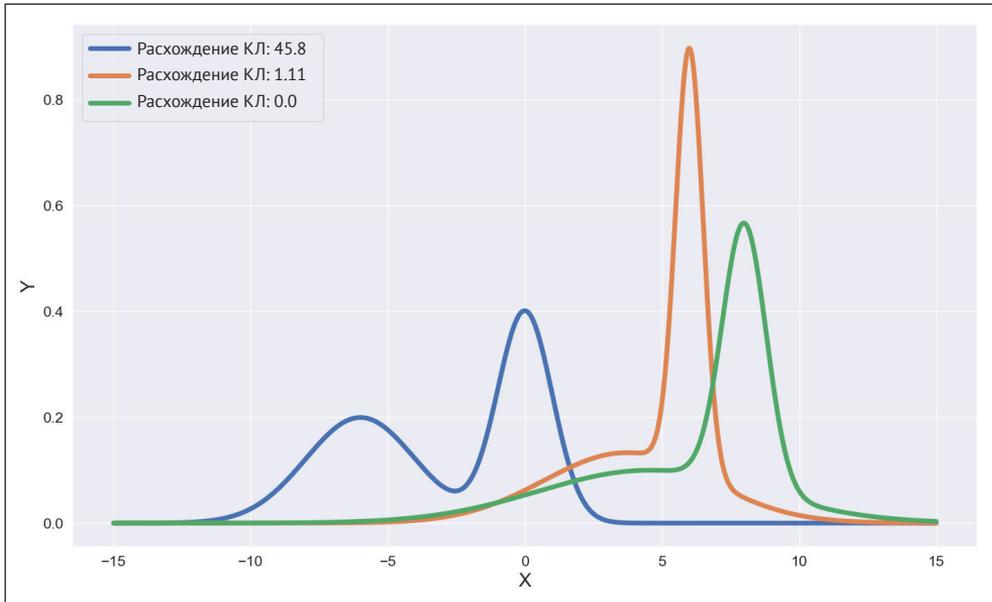


Рис. 7.5 ❖ Расхождение КЛ между всеми кривыми и зеленой кривой. Чем выше расхождение, тем сильнее различаются функции

Осложнения в методе естественного градиента

Как бы полезен ни был естественный градиент в ОП, у него есть серьезный недостаток – вычисление ИМФ обходится дорого. Если сложность вычисления градиента составляет $O(n)$, то для естественного градиента она равна уже $O(n^2)$, где n – число параметров. В посвященной NPG статье, опубликованной в 2003 году, этот алгоритм применялся к очень небольшим задачам с линейными стратегиями. Для современных глубоких нейронных сетей с сотнями тысяч параметров вычисления были бы слишком дорогими. Тем не менее благодаря некоторым аппроксимациям и хитроумным приемам метод естественного градиента можно распространить и на глубокие нейронные сети.

! В обучении с учителем метод естественного градиента не так необходим, как в обучении с подкреплением, потому что современные оптимизаторы, например Adam и RMSProp, и так эмпирически аппроксимируют вторую производную.

ОПТИМИЗАЦИЯ СТРАТЕГИИ В ДОВЕРИТЕЛЬНОЙ ОБЛАСТИ

Оптимизация стратегии в доверительной области (TRPO) – первый успешный алгоритм, в котором используется несколько аппроксимаций для вычисления естественного градиента, имеющих цель повысить управляемость и устойчивость обучения стратегии на основе глубокой нейронной сети. При обсуждении NPG мы пришли к выводу о невозможности вычислить обращение ИМФ для нелинейных функций с большим числом параметров. TRPO, над-

строенный над NPG, преодолевает эти трудности за счет введения суррогатной целевой функции и серии аппроксимаций. В результате с помощью этого алгоритма удастся обучить сложные стратегии ходьбы, прыжков и игр Atari.

TRPO – один из самых сложных безмодельных алгоритмов, и хотя мы уже знакомы с теоретическими основаниями естественного градиента, некоторые трудности все равно остаются. В этой главе мы опишем детали алгоритма только на интуитивном уровне и приведем основные уравнения. Если вас интересуют подробности, обратитесь к оригинальной статье (<https://arxiv.org/abs/1502.05477>), где имеются полные объяснения и доказательства теорем.

Мы также реализуем алгоритм и применим его к семейству окружающих сред Roboschool, но не станем обсуждать все детали. Полный код имеется в репозитории этой книги на GitHub.

Алгоритм TRPO

С некоторой точки зрения, TRPO можно рассматривать как обобщение алгоритма NPG на нелинейные аппроксимации функций. Основное улучшение – использование ограничения на расхождение КЛ между новой и старой стратегиями, в результате которого образуется *доверительная область*. Это позволяет сети выполнять более крупные шаги, оставаясь в пределах доверительной области. Получающаяся задача с ограничениями формулируется следующим образом:

$$\begin{aligned} & \text{maximize}_{\theta} J_{\theta_{old}}(\theta) \\ & \text{при условии } D_{KL}(\theta_{old}, \theta) \leq \delta. \end{aligned} \quad (7.2)$$

Здесь $J_{\theta_{old}}$ – суррогатная целевая функция, с которой мы скоро познакомимся, $D_{KL}(\theta_{old}, \theta)$ – расхождение КЛ между старой стратегией с параметрами θ_{old} и новой стратегией с параметрами θ , а δ – верхняя граница расхождения.

Суррогатная целевая функция спроектирована так, что максимизируется относительно параметров новой стратегии при использовании распределения состояния старой стратегии. Это делается с помощью выборки по значимости, позволяющей оценить распределение новой стратегии (желательное), зная только распределение старой стратегии (известное). Выборка по значимости необходима, потому что выбирается траектория с помощью старой стратегии, а интересует нас распределение новой. Таким образом, суррогатная целевая функция определяется следующим образом:

$$J_{\theta_{old}}(\theta) = E_{s \sim p_{old}, a \sim \pi_{old}} \left[\frac{\pi_{\theta}(a|s)}{\pi_{\theta_{old}}(a|s)} A_{\theta_{old}}(s, a) \right], \quad (7.3)$$

где $A_{\theta_{old}}$ – функция преимущества старой стратегии. Стало быть, задача оптимизации с ограничениями эквивалентна следующей:

$$\begin{aligned} & \text{maximize}_{\theta} E_{s \sim p_{old}, a \sim \pi_{old}} \left[\frac{\pi_{\theta}(a|s)}{\pi_{\theta_{old}}(a|s)} A_{\theta_{old}}(s, a) \right] \\ & \text{при условии } E_{s \sim p_{old}} [D_{KL}(\pi_{\theta_{old}}(\cdot|s) \| \pi_{\theta}(\cdot|s))] \leq \delta. \end{aligned} \quad (7.4)$$

Здесь $\pi(\cdot|s)$ – условное распределение действий при условии состояния s .

Осталось только заменить математическое ожидание эмпирическим средним по выборочному пакету и подставить эмпирическую оценку.

Задачи с ограничениями с трудом поддаются решению, и в случае TRPO задача оптимизации (7.4) решается приближенно с использованием линейной аппроксимации целевой функции и квадратичной аппроксимации ограничения, так что решение похоже на обновление в алгоритме NPG:

$$\theta \leftarrow \theta + \beta F^{-1}g,$$

где $g = \nabla_{\theta} J(\theta)$.

Задачу, получающуюся в результате аппроксимации исходной задачи оптимизации, можно решить методом **сопряженных градиентов (СГ)** – итеративным методом решения систем линейных уравнений. При обсуждении NPG мы отметили, что при большом числе параметров вычисления обходятся очень дорого. Однако метод СГ позволяет приближенно решить линейную задачу, не строя полную матрицу. Таким образом, мы получаем решение

$$s \approx F^{-1}g. \quad (7.5)$$

TRPO также позволяет оценить размер шага:

$$\beta = \sqrt{\frac{2\delta}{s^T F s}}. \quad (7.6)$$

Поэтому обновление принимает вид:

$$\theta \leftarrow \theta + \sqrt{\frac{2\delta}{s^T F s}} s. \quad (7.7)$$

Пока что мы описали частный случай шага алгоритма естественного градиента стратегии, но для полного обновления в TRPO не хватает ключевого элемента. Напомним, что исходную задачу мы аппроксимировали задачей с линейной целевой функцией и квадратичным ограничением. Следовательно, мы находим только локальную аппроксимацию ожидаемого дохода. При таких аппроксимациях нет никакой уверенности, что ограничение на расхождение КЛ по-прежнему удовлетворяется. Чтобы гарантировать выполнение нелинейного ограничения при улучшении нелинейной целевой функции, TRPO производит линейный поиск с целью найти большее значение α , которое удовлетворяет ограничению. Формула обновления с линейным поиском принимает вид:

$$\theta \leftarrow \theta + \alpha \sqrt{\frac{2\delta}{s^T F s}} s. \quad (7.8)$$

Может показаться, что линейный поиск – какая-то малозначительная часть алгоритма, однако, как показано в статье, его роль весьма значительна. Без него алгоритм может вычислять крупные шаги, что приводит к катастрофическому ухудшению качества.

Резюмируем. В алгоритме TRPO с помощью метода сопряженных градиентов вычисляется направление поиска, чтобы найти решение задачи с приближен-

ной целевой функцией и приближенным ограничением. Затем производится линейный поиск максимальной длины шага β , при которой удовлетворяется ограничение на расхождение КЛ, и целевая функция улучшается. Чтобы еще повысить скорость работы алгоритма, в методе сопряженных градиентов применяется эффективный способ вычисления произведения информационной матрицы Фишера на вектор (см. статью <https://arxiv.org/abs/1502.05477>).

Алгоритм TRPO можно включить в архитектуру исполнитель–критик, в которой критик помогает стратегии (исполнителю) обучиться решению задачи. Ниже приведен высокоуровневый псевдокод такого алгоритма (сочетания TRPO с критиком).

Инициализировать π_θ со случайными весами

Инициализировать окружающую среду $s \leftarrow env.reset()$

for эпизод 1..M **do**

Инициализировать пустой буфер

> Сгенерировать несколько траекторий

for step 1..TimeHorizon **do**

> Получать опыт, взаимодействуя со средой

$a \leftarrow \pi_\theta(s)$

$s', r, d \leftarrow env(a)$

$s \leftarrow s'$

if $d == True$:

$s \leftarrow env.reset()$

> Сохранить эпизод в буфере

$D \leftarrow D \cup (s_{1..T}, a_{1..T}, r_{1..T}, d_{1..T})$ # где T – длина эпизода

Вычислить значения функции преимущества A_i и n -шаговое предстоящее вознаграждение G_i

> Оценить градиент целевой функции

$$g = \nabla_\theta \tilde{E} \left[\begin{array}{l} \pi_\theta(a|s) \\ \pi_{\theta_{old}}(a|s) \end{array} A_{\theta_{old}}(s, a) \right] \quad (1)$$

> Вычислить s методом сопряженных градиентов

$$s \approx F^{-1}g \quad (2)$$

> Вычислить длину шага

$$\beta = \sqrt{\frac{2\delta}{s^T F s}} \quad (3)$$

> Обновить стратегию с использованием всего опыта в D

Линейный поиск с возвратом для нахождения максимального α , при котором удовлетворяется ограничение

$$\theta \leftarrow \theta + \alpha \beta s \quad (4)$$

> Обновить критика с использованием всего опыта в D

$$w \leftarrow w + \alpha_w \nabla_w \frac{1}{|D|} \sum_i (V_w(s_i) - G_i)^2$$

Имея такое высокоуровневое описание TRPO, можно приступить к его реализации.

Реализация алгоритма TRPO

В этом разделе мы сосредоточимся на построении графа вычислений и шагах оптимизации стратегии. Реализацию прочих аспектов (цикл для сбора траекторий взаимодействия со средой, метод сопряженных градиентов и алгоритм линейного поиска) мы опускаем, но интересующийся читатель может найти полный код в репозитории этой книги на GitHub. Мы приводим реализацию для непрерывного управления.

Сначала создадим все местозаменители и две глубокие нейронные сети для аппроксимации стратегии (исполнителя) и функции ценности (критика).

```
act_ph = tf.placeholder(shape=(None, act_dim), dtype=tf.float32, name='act')
obs_ph = tf.placeholder(shape=(None, obs_dim[0]), dtype=tf.float32, name='obs')
ret_ph = tf.placeholder(shape=(None,), dtype=tf.float32, name='ret')
adv_ph = tf.placeholder(shape=(None,), dtype=tf.float32, name='adv')
old_p_log_ph = tf.placeholder(shape=(None,), dtype=tf.float32, name='old_p_log')
old_mu_ph = tf.placeholder(shape=(None, act_dim), dtype=tf.float32, name='old_mu')
old_log_std_ph = tf.placeholder(shape=(act_dim), dtype=tf.float32,
name='old_log_std')
p_ph = tf.placeholder(shape=(None,), dtype=tf.float32, name='p_ph')

# результат работы метода сопряженных градиентов
cg_ph = tf.placeholder(shape=(None,), dtype=tf.float32, name='cg')

# нейронная сеть исполнителя
with tf.variable_scope('actor_nn'):
    p_means = mlp(obs_ph, hidden_sizes, act_dim, tf.tanh, last_activation=tf.tanh)
    log_std = tf.get_variable(name='log_std', initializer=np.ones(act_dim,
dtype=np.float32))

# нейронная сеть критика
with tf.variable_scope('critic_nn'):
    s_values = mlp(obs_ph, hidden_sizes, 1, tf.nn.relu, last_activation=None)
    s_values = tf.squeeze(s_values)
```

Здесь следует отметить несколько моментов:

- 1) местозаменители с префиксом `old_` относятся к тензорам старой стратегии;
- 2) исполнитель и критик определяются в разных областях видимости, потому что впоследствии нужно будет выбирать параметры по отдельности;
- 3) пространство действий имеет нормальное распределение с диагональной ковариационной матрицей, не зависящей от состояния. Диагональную матрицу можно затем преобразовать в вектор, содержащий по одному элементу для каждого действия. Мы работаем с логарифмом этого вектора.

Теперь можно прибавить к предсказанному среднему шум, имеющий нормальное распределение с нулевым средним и единичным стандартным отклонением, обрезать действия и вычислить гауссово логарифмическое правдоподобие:

```
p_noisy = p_means + tf.random_normal(tf.shape(p_means), 0, 1) * tf.exp(log_std)
a_sampl = tf.clip_by_value(p_noisy, low_action_space, high_action_space)
p_log = gaussian_log_likelihood(act_ph, p_means, log_std)
```

Далее нужно вычислить целевую функцию $\tilde{E} \left[\frac{\pi_{\theta}(a|s)}{\pi_{\theta_{old}}(a|s)} A_{\theta_{old}}(s, a) \right]$, среднеквадратическую потерю критика и создать оптимизатор критика:

```
# функция потерь TRPO
ratio_new_old = tf.exp(p_log - old_p_log_ph)
p_loss = - tf.reduce_mean(ratio_new_old * adv_ph)

# среднеквадратическая ошибка в качестве потери
v_loss = tf.reduce_mean((ret_ph - s_values)**2)

# оптимизация критика
v_opt = tf.train.AdamOptimizer(cr_lr).minimize(v_loss)
```

Затем создаем граф вычислений по формулам (2), (3) и (4) в приведенном выше псевдокоде. На самом деле вычисления по формулам (2) и (3) в TensorFlow не производятся, поэтому они не являются частями графа вычислений. Но все равно в этом графе нужно позаботиться о некоторых сопутствующих вещах. Ниже перечислены необходимые шаги.

1. Оценить функцию потерь в алгоритме градиента стратегии.
2. Определить процедуру восстановления параметров стратегии. Это необходимо, потому что в алгоритме линейного поиска мы будем оптимизировать стратегию и проверять ограничения, и если новая стратегия им не удовлетворяет, то придется восстановить параметры стратегии и попробовать меньший коэффициент.
3. Вычислить произведение матрицы Фишера на вектор. Существует эффективный способ вычисления Fx без вычисления всей матрицы F .
4. Выполнить один шаг TRPO.
5. Обновить стратегию.

Начнем с шага 1, оценки функции потерь.

```
def variables_in_scope(scope):
    return tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope)

# Собрать и сериализовать параметры исполнителя
p_variables = variables_in_scope('actor_nn')
p_var_flatten = flatten_list(p_variables)

# Градиент функции потерь по параметрам исполнителя
p_grads = tf.gradients(p_loss, p_variables)
p_grads_flatten = flatten_list(p_grads)
```

Поскольку мы работаем с вектором параметров, его необходимо сериализовать с помощью функции `flatten_list`. Функция `variable_in_scope` возвращает обучаемые параметры в области видимости `scope`. Она используется, чтобы получить параметры исполнителя, т. е. градиенты следует вычислить только по ним.

Теперь шаг 2 – параметры стратегии восстанавливаются следующим образом:

```
p_old_variables = tf.placeholder(shape=(None,), dtype=tf.float32,
name='p_old_variables')
```

```
# эта переменная используется как индекс при восстановлении параметров исполнителя
it_v1 = tf.Variable(0, trainable=False)
```

```

restore_params = []
for p_v in p_variables:
    upd_rsh = tf.reshape(p_old_variables[it_v1 :
it_v1+tf.reduce_prod(p_v.shape)], shape=p_v.shape)
    restore_params.append(p_v.assign(upd_rsh))
    it_v1 += tf.reduce_prod(p_v.shape)
restore_params = tf.group(*restore_params)

```

Мы в цикле обходим все параметры и присваиваем каждому старые значения.

Чтобы вычислить произведение матрицы Фишера на вектор на шаге 3, вычисляем вторую производную расхождения КЛ по параметрам стратегии:

```

# гауссово расхождение КЛ между двумя стратегиями
dkl_diverg = gaussian_DKL(old_mu_ph, old_log_std_ph, p_means, log_std)

# якобиан расхождения КЛ (необходим для вычисления произведения матрицы
# Фишера на вектор)
dkl_diverg_grad = tf.gradients(dkl_diverg, p_variables)
dkl_matrix_product = tf.reduce_sum(flatten_list(dkl_diverg_grad) * p_ph)

# произведение матрицы Фишера на вектор
Fx = flatten_list(tf.gradients(dkl_matrix_product, p_variables))

```

На шагах 4 и 5 нужно применить обновления к стратегии. Здесь β_{ph} – значение β , вычисленное по формуле (7.6), а α – масштабный коэффициент, найденный линейным поиском:

```

# обновление NPG
beta_ph = tf.placeholder(shape=(), dtype=tf.float32, name='beta')
npg_update = beta_ph * cg_ph
alpha = tf.Variable(1., trainable=False)

# обновление TRPO
trpo_update = alpha * npg_update

# применить обновления к стратегии
it_v = tf.Variable(0, trainable=False)
p_opt = []
for p_v in p_variables:
    upd_rsh = tf.reshape(trpo_update[it_v :
it_v+tf.reduce_prod(p_v.shape)], shape=p_v.shape)
    p_opt.append(p_v.assign_sub(upd_rsh))
    it_v += tf.reduce_prod(p_v.shape)

p_opt = tf.group(*p_opt)

```

Заметим, что без α мы имеем то же обновление, что в алгоритме NPG.

Обновление применяется к каждому параметру стратегии. Это делает функция $p_v.assign_sub(upd_rsh)$, которая присваивает p_v значения $p_v - upd_rsh$, т. е. вычисляет $\theta \leftarrow \theta - \alpha\beta s$. Вычитание производится, потому что мы преобразовали целевую функцию в функцию потерь.

Теперь посмотрим, как все написанные кусочки соединяются вместе при обновлении стратегии на каждой итерации алгоритма. Показанные ниже фраг-

менты кода нужно поместить после внутреннего цикла, туда, где производится выборка траекторий. Но прежде чем привести код, напомним, что нам предстоит сделать:

- 1) получить выходные данные, логарифм вероятности, стандартное отклонение и параметры стратегии, которую мы использовали для выборки траектории. Это наша старая стратегия;
- 2) применить метод сопряженных градиентов;
- 3) вычислить длину шага β ;
- 4) выполнить линейный поиск с возвратом для нахождения α ;
- 5) произвести обновление стратегии.

Первый пункт реализуется следующими операциями:

```
...
    old_p_log, old_p_means, old_log_std = sess.run([p_log, p_means,
log_std], feed_dict={obs_ph:obs_batch, act_ph:act_batch, adv_ph:adv_batch,
ret_ph:rtg_batch})
    old_actor_params = sess.run(p_var_flatten)
    old_p_loss = sess.run([p_loss], feed_dict={obs_ph:obs_batch,
act_ph:act_batch, adv_ph:adv_batch, ret_ph:rtg_batch,
old_p_log_ph:old_p_log})
```

Для алгоритма сопряженных градиентов нужна входная функция, которая возвращает оценку информационной матрицы Фишера, градиент целевой функции и количество итераций (в TRPO оно находится в диапазоне от 5 до 15).

```
def H_f(p):
    return sess.run(Fx, feed_dict={old_mu_ph:old_p_means,
old_log_std_ph:old_log_std, p_ph:p, obs_ph:obs_batch, act_ph:act_batch,
adv_ph:adv_batch, ret_ph:rtg_batch})

    g_f = sess.run(p_grads_flatten,
feed_dict={old_mu_ph:old_p_means,obs_ph:obs_batch, act_ph:act_batch,
adv_ph:adv_batch, ret_ph:rtg_batch, old_p_log_ph:old_p_log})

    conj_grad = conjugate_gradient(H_f, g_f, iters=conj_iters)
```

Далее можно вычислить длину шага β (`beta_np`) и максимальный коэффициент α (`best_alpha`), при котором удовлетворяется ограничение (методом линейного поиска с возвратом), а затем выполнить оптимизации, подав все значения на вход графа вычислений:

```
beta_np = np.sqrt(2*delta / np.sum(conj_grad * H_f(conj_grad)))

def DKL(alpha_v):
    sess.run(p_opt, feed_dict={beta_ph:beta_np, alpha:alpha_v,
cg_ph:conj_grad, obs_ph:obs_batch, act_ph:act_batch, adv_ph:adv_batch,
old_p_log_ph:old_p_log})
    a_res = sess.run([dkl_diverg, p_loss],
feed_dict={old_mu_ph:old_p_means, old_log_std_ph:old_log_std,
obs_ph:obs_batch, act_ph:act_batch, adv_ph:adv_batch, ret_ph:rtg_batch,
old_p_log_ph:old_p_log})
    sess.run(restore_params, feed_dict={p_old_variables:
old_actor_params})
    return a_res
```

```
best_alpha = backtracking_line_search(DKL, delta, old_p_loss, p=0.8)
sess.run(p_opt, feed_dict={beta_ph:beta_np, alpha:best_alpha,
cg_ph:conj_grad, obs_ph:obs_batch, act_ph:act_batch, adv_ph:adv_batch,
old_p_log_ph:old_p_log})
...
```

Как видим, функция `backtracking_line_search` вызывает функцию `DKL`, которая возвращает расхождение КЛ между старой и новой стратегиями, коэффициент δ (порог ограничения) и потерю старой стратегии. Далее `backtracking_line_search`, начиная с $\alpha = 1$, постепенно уменьшает это значение, пока не будет выполнено следующее условие: расхождение КЛ меньше δ и новая потеря уменьшилась.

Таким образом, специфические гиперпараметры TRPO таковы:

- `delta` (δ) – максимальное расхождение КЛ между старой и новой стратегиями;
- количество итераций в методе сопряженных градиентов, `conj_iters`. Обычно выбирается значение между 5 и 15.

Поздравляем тех, кто добрался до этого места! Это было нелегко.

Применение TRPO

Эффективность и устойчивость алгоритма TRPO позволяют протестировать его на новых, более сложных окружающих средах. Мы применили его к `Roboschool`. Семейство `Roboschool` и его аналог `Mujoco` часто используются в качестве испытательного стенда для алгоритмов управления сложными агентами с непрерывными действиями, в частности TRPO. Конкретно мы тестировали TRPO в среде `RoboschoolWalker2d`, где агент должен научиться ходить как можно быстрее. Эта окружающая среда изображена на рис. 7.6. Взаимодействие со средой завершается, когда агент падает или по прошествии 1000 временных шагов с начала теста. Состояние представлено классом `Vox` размера 22, а агент управляется 6 числами с плавающей точкой в диапазоне $[-1, 1]$.



Рис. 7.6 ❖ Окружающая среда `RoboschoolWalker2d`

В TRPO количество шагов взаимодействия с окружающей средой в каждом эпизоде называется *временным горизонтом*. Это число определяет также размер пакета. Кроме того, имеет смысл запускать параллельно несколько агентов, чтобы собрать более репрезентативные данные о среде. В данном случае размер пакета будет равен временному горизонту, умноженному на количество агентов. Хотя наша модель не рассчитана на параллельное выполнение нескольких агентов, той же цели можно достичь, если сделать временной горизонт длиннее максимально допустимого числа шагов в каждом эпизоде. Например, зная, что в среде RoboschoolWalker2d максимальное число шагов равно 1000, мы можем взять временной горизонт 6000, и тогда наверняка получим не менее шести полных траекторий.

Мы прогоняли TRPO с гиперпараметрами, указанными в следующей таблице. В третьем столбце приведены стандартные диапазоны гиперпараметров.

Гиперпараметр	Для RoboschoolWalker2	Диапазон
Число итераций в методе сопряженных градиентов	10	[7-10]
Дельта (δ)	0.01	[0.005-0.03]
Размер пакета (временной горизонт * количество агентов)		

Ход выполнения TRPO (и PPO, как мы увидим в следующем разделе) можно оценить, глядя на полное вознаграждение в каждой игре и ценности состояний, предсказанные критиком.

Мы провели обучение, выполнив 6 млн шагов, получившиеся результаты показаны на рис. 7.7. После 2 млн шагов агент набрал 1300 баллов (это неплохо) и научился уверенно ходить с умеренной скоростью. На начальном этапе обучения заметен небольшой провал, вероятно, связанный с попаданием в локальный оптимум. Но потом агент исправился, и результат монотонно улучшался до отметки 1250 баллов.

Кроме того, предсказанная ценность состояния – важная метрика для изучения результатов. В целом она ведет себя более стабильно, чем полное вознаграждение, и проще для анализа. График ее изменения показан на рис. 7.8. Действительно, как мы и предполагали, кривая более гладкая, несмотря на несколько пиков в районе 4 и 4.5 млн шагов.

На этом графике также видно, что после 3 млн шагов агент продолжает обучаться, пусть и очень медленно.

Итак, TRPO – довольно сложный алгоритм, содержащий немало настраиваемых компонентов. Однако он доказывает эффективность ограничения стратегии доверительной областью, призванного не допустить слишком большого отклонения от текущего распределения.

А можно ли спроектировать более простой и общий алгоритм, основанный на том же подходе?

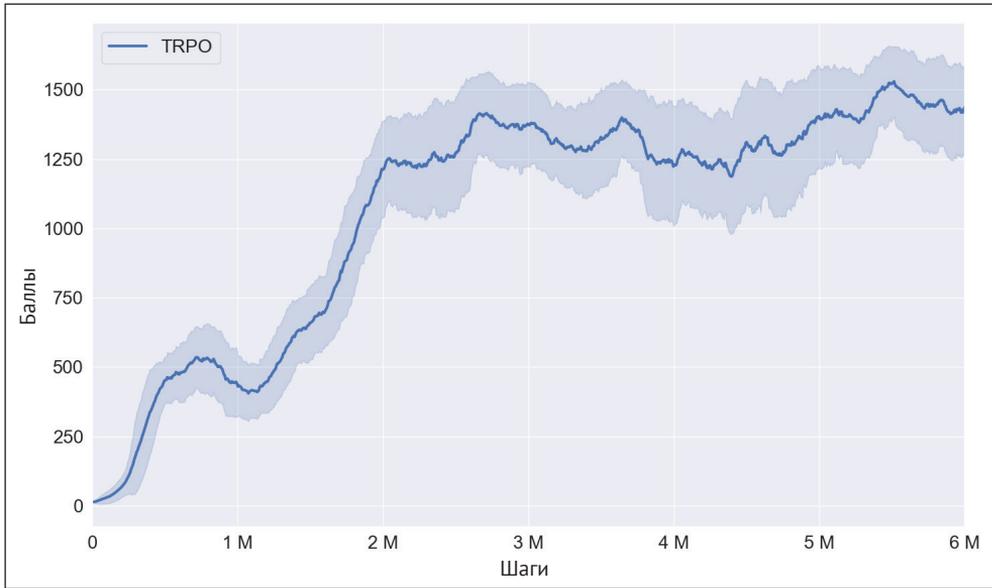


Рис. 7.7 ❖ Кривая обучения TRPO в среде RoboschoolWalker2d

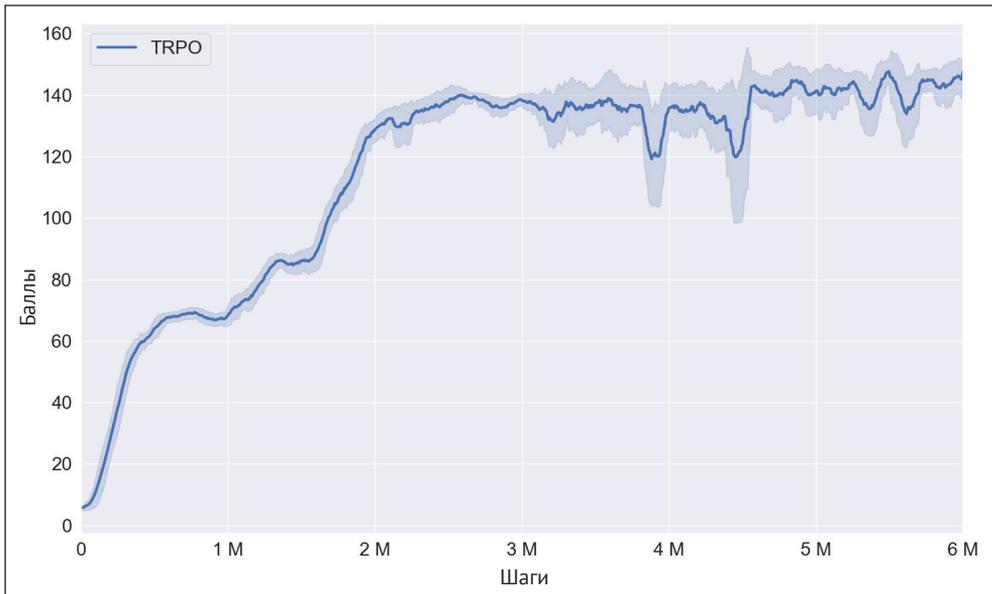


Рис. 7.8 ❖ Ценности состояний, предсказанные критиком в алгоритме TRPO в среде RoboschoolWalker2d

ПРОКСИМАЛЬНАЯ ОПТИМИЗАЦИЯ СТРАТЕГИИ

В работе Шульмана с соавторами (<https://arxiv.org/pdf/1707.06347.pdf>) показано, что это возможно. Они воспользовались той же идеей, что и TRPO, но уменьшили сложность метода. Их алгоритм называется **проксимальной оптимизацией стратегии** (Proximal Policy Optimization – **PPO**), а его главное достоинство – использование только оптимизации первого порядка, но без снижения надежности по сравнению с TRPO. Алгоритм PPO к тому же более общий, обладает лучшей выборочной эффективностью, чем TRPO, и допускает несколько обновлений на мини-пакетах.

Краткое описание

Основная идея PPO – обрезать суррогатную целевую функцию, когда она далеко отклоняется, а не пытаться ограничить ее, как в TRPO. Это не дает стратегии произвести слишком сильные обновления. Главная целевая функция имеет вид:

$$\mathcal{L}^{CLIP}(\theta) = E_{s \sim p_{old}, a \sim \pi_{old}} [\min(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1 - \varepsilon, 1 + \varepsilon)A_t)], \quad (7.9)$$

где $r_t(\theta)$ определено следующим образом:

$$r_t(\theta) = \frac{\pi_{\theta}(a_t | s_t)}{\pi_{\theta_{old}}(a_t | s_t)}. \quad (7.10)$$

Смысл этой целевой функции в том, что если отношение вероятностей $r_t(\theta)$ старой и новой стратегий больше или меньше постоянной ε , то следует брать минимальное значение. Это не дает величине r_t выйти за пределы интервала $[1 - \varepsilon, 1 + \varepsilon]$. Опорное значение равно 1, $r_t(\theta_{old}) = 1$.

Алгоритм PPO

В практическом алгоритме, описанном в работе по PPO, используется усеченный вариант **обобщенной оценки преимущества** (Generalized Advantage Estimation – **GAE**), идея которого впервые была высказана в статье «High-Dimensional Continuous Control using Generalized Advantage Estimation» (<https://arxiv.org/pdf/1506.02438.pdf>). В GAE преимущество вычисляется по формуле:

$$A_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \dots + (\gamma\lambda)^{T-t+1}\delta_{T-1}, \quad (7.11)$$

где $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$,

вместо обычной формулы оценки преимущества:

$$A_t = r_t + \gamma t_{t+1} + \dots + \gamma^{T-t+1} r_{T-1} V(s_T). \quad (7.12)$$

На каждой итерации алгоритма PPO несколько параллельных исполнителей собирают N траекторий с временным горизонтом T , и стратегия обновляется K раз с помощью мини-пакетов. В таблице ниже приведены стандартные зна-

чения гиперпараметров PPO. Хотя гиперпараметры для каждой задачи подбираются отдельно, полезно иметь представление об их диапазонах (третий столбец таблицы).

Гиперпараметр	Обозначение	Диапазон
Скорость обучения стратегии	–	$[1e^{-5}, 1e^{-3}]$
Количество итераций стратегии	K	[3, 15]
Количество траекторий (эквивалентно количеству параллельных исполнителей)	N	[1, 20]
Временной горизонт	T	[64, 5120]
Размер мини-пакета	–	[64, 5120]
Коэффициент обрезания	ϵ	0.1 или 0.2
Дельта (для GAE)	δ	[0.9, 0.97]
Гамма (для GAE)	γ	[0.8, 0.995]

Реализация PPO

Зная о составных частях PPO, мы можем реализовать этот алгоритм на Python с помощью TensorFlow.

Структура и реализация PPO сильно напоминают алгоритмы исполнитель–критик, но вместе с тем есть несколько дополнительных деталей, которые будут рассмотрены ниже.

Одна из них – обобщенная оценка преимущества (7.11), которая укладывается всего в несколько строчек, если воспользоваться уже написанной функцией `discounted_rewards`, вычисляющей выражение (7.12):

```
def GAE(rews, v, v_last, gamma=0.99, lam=0.95):
    vs = np.append(v, v_last)
    delta = np.array(rews) + gamma*vs[1:] - vs[:-1]
    gae_advantage = discounted_rewards(delta, 0, gamma*lam)
    return gae_advantage
```

Функция `GAE` вызывается из метода `store` класса `Buffer` при сохранении траектории:

```
class Buffer():
    def __init__(self, gamma, lam):
        ...

    def store(self, temp_traj, last_sv):
        if len(temp_traj) > 0:
            self.ob.extend(temp_traj[:,0])
            rtg = discounted_rewards(temp_traj[:,1], last_sv, self.gamma)
            self.adv.extend(GAE(temp_traj[:,1], temp_traj[:,3], last_sv,
self.gamma, self.lam))
            self.rtg.extend(rtg)
            self.ac.extend(temp_traj[:,2])

def get_batch(self):
    return np.array(self.ob), np.array(self.ac), np.array(self.adv),
```

```
np.array(self.rtg)
    def __len__(self):
        ...
```

Здесь ... заменяет опущенные строчки.

Теперь можно определить обрезанную суррогатную функцию потерь (7.9):

```
def clipped_surrogate_obj(new_p, old_p, adv, eps):
    rt = tf.exp(new_p - old_p) # i.e. pi / old_pi
    return -tf.reduce_mean(tf.minimum(rt*adv, tf.clip_by_value(rt, 1-eps,
1+eps)*adv))
```

Тут все понятно и не требует пояснений.

Граф вычислений тоже не содержит ничего нового, но все же быстренько пробежимся по его коду.

```
# Местозаменители
act_ph = tf.placeholder(shape=(None,act_dim), dtype=tf.float32, name='act')
obs_ph = tf.placeholder(shape=(None, obs_dim[0]), dtype=tf.float32, name='obs')
ret_ph = tf.placeholder(shape=(None,), dtype=tf.float32, name='ret')
adv_ph = tf.placeholder(shape=(None,), dtype=tf.float32, name='adv')
old_p_log_ph = tf.placeholder(shape=(None,), dtype=tf.float32, name='old_p_log')

# Исполнитель
with tf.variable_scope('actor_nn'):
    p_means = mlp(obs_ph, hidden_sizes, act_dim, tf.tanh, last_activation=tf.tanh)
    log_std = tf.get_variable(name='log_std', initializer=np.ones(act_dim,
dtype=np.float32))
    p_noisy = p_means + tf.random_normal(tf.shape(p_means), 0, 1) * tf.exp(log_std)
    act_smp = tf.clip_by_value(p_noisy, low_action_space, high_action_space)

# Вычислить гауссово логарифмическое правдоподобие
p_log = gaussian_log_likelihood(act_ph, p_means, log_std)

# Критик
with tf.variable_scope('critic_nn'):
    s_values = tf.squeeze(mlp(obs_ph, hidden_sizes, 1, tf.tanh,
last_activation=None))

# Функция потери в PPO
p_loss = clipped_surrogate_obj(p_log, old_p_log_ph, adv_ph, eps)

# Среднеквадратическая ошибка в качестве функции потери
v_loss = tf.reduce_mean((ret_ph - s_values)**2)

# Оптимизаторы
p_opt = tf.train.AdamOptimizer(ac_lr).minimize(p_loss)
v_opt = tf.train.AdamOptimizer(cr_lr).minimize(v_loss)
```

Код взаимодействия с окружающей средой и накопления опыта такой же, как в алгоритмах AC и TRPO. Но в реализации PPO, имеющейся в репозитории на GitHub, приведена простая реализация с несколькими агентами.

После того как данные о $N * T$ переходах собраны (N – количество траекторий, T – временной горизонт каждой траектории), мы готовы обновить стратегию и критика. В обоих случаях оптимизация прогоняется несколько раз и произ-

водится на мини-пакетах. Но предварительно мы должны выполнить `p_log` на полном пакете, потому что обрезанной целевой функции нужны вероятности старой стратегии из журнала действий.

```

...
obs_batch, act_batch, adv_batch, rtg_batch = buffer.get_batch()
old_p_log = sess.run(p_log, feed_dict={obs_ph:obs_batch,
act_ph:act_batch, adv_ph:adv_batch, ret_ph:rtg_batch})
old_p_batch = np.array(old_p_log)
lb = len(buffer)
lb = len(buffer)
shuffled_batch = np.arange(lb)

# Шаги оптимизации стратегии
for _ in range(actor_iter):
    # перемешать пакет на каждой итерации
    np.random.shuffle(shuffled_batch)
    for idx in range(0,lb, minibatch_size):
        minib = shuffled_batch[idx:min(idx+batch_size,lb)]
        sess.run(p_opt, feed_dict={obs_ph:obs_batch[minib],
act_ph:act_batch[minib], adv_ph:adv_batch[minib],
old_p_log_ph:old_p_batch[minib]})

# Шаги оптимизации функции ценности
for _ in range(critic_iter):
    # перемешать пакет на каждой итерации
    np.random.shuffle(shuffled_batch)
    for idx in range(0,lb, minibatch_size):
        minib = shuffled_batch[idx:min(idx+minibatch_size,lb)]
        sess.run(v_opt, feed_dict={obs_ph:obs_batch[minib],
ret_ph:rtg_batch[minib]})
...

```

На каждой итерации оптимизации мы перемешиваем пакет, чтобы все мини-пакеты были различны.

На этом реализация PPO заканчивается, но следует помнить, что до и после каждой итерации мы также подсчитываем сводную статистику, которая позднее будет использована в TensorBoard для анализа результатов и отладки алгоритма. Этот код здесь не приводится, потому что он такой же, как и раньше, да к тому же довольно длинный, но в репозитории книги он имеется. Если вы хотите по-настоящему овладеть этими алгоритмами ОП, то должны хорошо понимать, что означает каждый график.

Применение PPO

Алгоритмы PPO и TRPO очень похожи, и мы решили сравнить их работу в одной и той же окружающей среде RoboschoolWalker2d. Мы потратили некоторое время на настройку обоих алгоритмов, чтобы сделать сравнение справедливым. Гиперпараметры TRPO были приведены в предыдущем разделе, а гиперпараметры PPO – в таблице ниже.

Гиперпараметр	Значение
Нейронная сеть	64, tanh, 64, tanh
Скорость обучения стратегии	$3e^{-4}$
Количество итераций исполнителя	10
Количество агентов	1
Временной горизонт	5000
Размер мини-пакета	256
Коэффициент обрезания	0.2
Дельта (для GAE)	0.95
Гамма (для GAE)	0.99

Результаты сравнения PPO и TRPO показаны на рис. 7.9. Алгоритму PPO необходимо больше опыта, чтобы «взлететь», но, набрав достаточно опыта, он быстро прогрессирует и обходит TRPO. При данных конкретных параметрах PPO в итоге показывает лучшие результаты, чем TRPO. Имейте в виду, что при других значениях гиперпараметров результаты могли бы отличаться, возможно, в лучшую сторону.

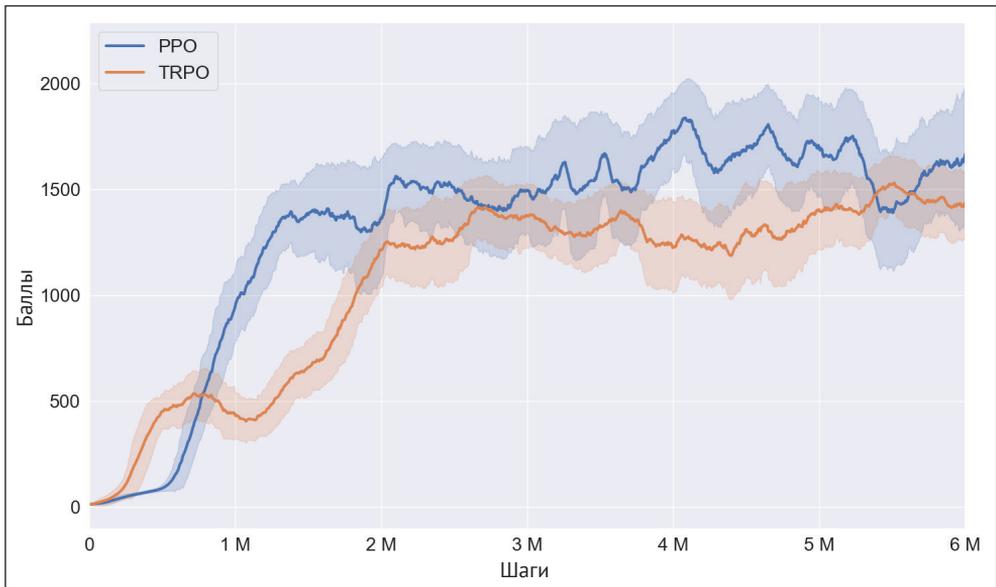


Рис. 7.9 ❖ Сравнение качества алгоритмов PPO и TRPO

❗ Хотим поделиться личными наблюдениями: мы обнаружили, что алгоритм PPO настраивать труднее, чем TRPO. Одна из причин – то, что у него больше гиперпараметров. Кроме того, один из самых важных гиперпараметров – скорость обучения, и его неправильная настройка может оказать заметное влияние на результаты. Сильная сторона TRPO – отсутствие скорости обучения и вообще меньшее число гиперпараметров, которые проще настроить. Но у PPO есть достоинства – он быстрее и может работать с более разнообразными окружающими средами.

РЕЗЮМЕ

В этой главе мы научились адаптировать алгоритмы градиента стратегии к управлению агентами с непрерывными действиями, а затем познакомились с новым семейством окружающих сред, Roboschool.

Мы также обсудили и разработали два более эффективных алгоритма градиента стратегии: оптимизация стратегии в доверительной области (TRPO) и проксимальная оптимизация стратегии (PPO). Они лучше используют данные, полученные от среды, и стремятся ограничить различия в распределении действий в двух соседних стратегиях. Точнее, TRPO строит доверительную область вокруг целевой функции, применяя вторую производную и некоторые ограничения, основанные на расхождении КЛ между старой и новой стратегиями. С другой стороны, PPO оптимизирует целевую функцию, похожую на применяемую в TRPO, но пользуется при этом только первыми производными. PPO запрещает стратегии предпринимать слишком большие шаги, обрезая целевую функцию, если она чрезмерно возрастает.

PPO и TRPO по-прежнему являются алгоритмами с единой стратегией (как и другие алгоритмы градиента стратегии), но их выборочная эффективность выше, чем у AC и REINFORCE. В случае TRPO это объясняется тем, что, применяя вторые производные, этот алгоритм по существу извлекает из данных информацию высшего порядка. А высокая выборочная эффективность PPO связана с выполнением нескольких обновлений стратегии на одних и тех же данных единой стратегии.

Благодаря высокой выборочной эффективности, робастности и надежности TRPO и в особенности PPO применяется во многих очень сложных окружающих средах, например Dota (<https://openai.com/blog/openai-five/>).

PPO и TRPO, равно как AC и REINFORCE, являются стохастическими градиентными алгоритмами.

В следующей главе мы рассмотрим два детерминированных алгоритма градиента стратегии. Они интересны тем, что обладают некоторыми полезными свойствами, у которых нет аналогов в ранее изученных алгоритмах.

Вопросы

1. Как входящая в состав стратегии нейронная сеть может управлять непрерывным агентом?
2. Что такое расхождение Кульбака–Лейблера?
3. В чем основная идея алгоритма TRPO?
4. Как расхождение КЛ используется в TRPO?
5. В чем главное преимущество алгоритма PPO?
6. За счет чего PPO добивается высокой выборочной эффективности?

Для дальнейшего чтения

- С оригинальной статьей об алгоритме NPG можно познакомиться по адресу <https://papers.nips.cc/paper/2073-a-natural-policy-gradient.pdf>.
- Статья «High-Dimensional Continuous Control Using Generalized Advantage Estimation», в которой введена обобщенная функция преимущества, опубликована по адресу <https://arxiv.org/pdf/1506.02438.pdf>.
- Оригинальная статья «Trust Region Policy Optimization» об алгоритме оптимизации стратегии в доверительной области опубликована по адресу <https://arxiv.org/pdf/1502.05477.pdf>.
- Оригинальная статья «Proximal Policy Optimization Algorithms» об алгоритме проксимальной оптимизации стратегии опубликована по адресу <https://arxiv.org/pdf/1707.06347.pdf>.
- Дополнительные сведения об алгоритме проксимальной оптимизации стратегии приведены в блоге <https://openai.com/blog/openai-baselines-ppo/>.
- О применении PPO к окружающей среде Dota 2 см. статью в блоге компании OpenAI по адресу <https://openai.com/blog/openai-five/>.

Глава 8

Применения алгоритмов DDPG и TD3

В предыдущей главе мы завершили рассмотрение основных алгоритмов градиента стратегии. Благодаря способности работать с непрерывными пространствами действий они находят применения в очень сложных и развитых системах управления. В методах градиента стратегии можно применять вторую производную, как в алгоритме TRPO, или иные подходы, чтобы ограничить свободу обновления стратегии и предотвратить неожиданное и нежелательное поведение. Однако главная проблема этих алгоритмов – низкая эффективность с точки зрения объема опыта, необходимого, чтобы научиться качественно решать задачу. Этот недостаток присущ всем алгоритмам с единой стратегией, которые вынуждены накапливать новый опыт после каждого обновления стратегии. В этой главе мы познакомимся с новым типом алгоритмов исполнитель–критик с разделенной стратегией, которые обучаются целевой детерминированной стратегией, но окружающую среду исследуют, применяя стохастическую стратегию. В силу своей основной характеристики они называются детерминированными методами градиента стратегии. Мы покажем, как работают эти алгоритмы, и обсудим их связь с методами Q-обучения. Затем мы представим два детерминированных метода градиента стратегии: **глубокий детерминированный градиент стратегии** (deep deterministic policy gradient – DDPG) и пришедший ему на смену **двойной глубокий детерминированный градиент стратегии с задержкой** (twin delayed deep deterministic policy gradient – TD3). Чтобы лучше прочувствовать их возможности, мы реализуем оба алгоритма и применим их к новой окружающей среде.

В этой главе будут рассмотрены следующие вопросы:

- сочетание оптимизации градиента стратегии с Q-обучением;
- алгоритм DDPG;
- алгоритм TD3.

СОЧЕТАНИЕ ОПТИМИЗАЦИИ ГРАДИЕНТА СТРАТЕГИИ С Q-ОБУЧЕНИЕМ

В этой книге мы рассматриваем два основных типа безмодельных алгоритмов: основанные на градиенте стратегии и на функции ценности. Представителями

первого семейства являются REINFORCE, исполнитель–критик, PPO и TRPO, а второго – Q-обучение, SARSA и DQN. Помимо самого способа обучения стратегии (в алгоритмах градиента стратегии используется метод стохастического градиентного подъема в направлении наискорейшего увеличения оценки дохода, а в алгоритмах на основе ценности сначала обучаются ценности действий для каждой пары состояние–действие, а затем на этой основе строят стратегию), существуют и другие важные различия, побуждающие отдать предпочтение тому или иному семейству. Это использование единой или разделенной стратегиями и способность работать с большими пространствами действий. В предыдущих главах мы уже обсуждали различие между единой и разделенной стратегией, но особенно важно четко понимать это различие сейчас, чтобы по достоинству оценить алгоритмы, рассматриваемые в этой главе.

При обучении с разделенной стратегией есть возможность использовать прошлый опыт для уточнения текущей стратегии, несмотря на то что этот опыт получен на другом распределении вероятностей. В алгоритме DQN для данной цели весь накопленный агентом опыт запоминается в буфере воспроизведения, а при обновлении целевой стратегии из этого буфера выбираются мини-пакеты. На противоположном конце спектра располагается обучение с единой стратегией, в котором опыт накапливается с помощью текущей стратегии. Это означает, что старый опыт использовать нельзя, и при каждом обновлении стратегии старые данные отбрасываются. В итоге, поскольку для обучения с разделенной стратегией данные можно использовать многократно, сокращается количество взаимодействий с окружающей средой, необходимое, чтобы обучиться решению задачи. В тех случаях, когда организовать получение новых примеров дорого или трудно, это различие может играть решающую роль, так что альтернативы алгоритмам с разделенной стратегией может и не оказаться.

Второй фактор – пространство действий. В главе 7 мы видели, что алгоритмы градиента стратегии могут справляться с очень большими и непрерывными пространствами действий. К сожалению, это не так для алгоритмов Q-обучения. Чтобы выбрать действие, они должны выполнить максимизацию по всему пространству действий, а если оно очень велико или непрерывно, то это практически неосуществимо. Таким образом, алгоритмы Q-обучения можно применять к сколь угодно сложным задачам (с очень большим пространством состояний), при условии что пространство действий ограничено.

Напоследок отметим, что ни один из вышеупомянутых алгоритмов не является однозначно лучшим, выбор зависит от конкретной задачи. Однако минусы одних компенсируются плюсами других, и потому возникает естественный вопрос: «Можно ли объединить достоинства, свойственные разным семействам, в одном алгоритме?»

Детерминированный градиент стратегии

Проектирование алгоритма, который обучался бы с разделенной стратегией и мог бы находить устойчивые стратегии в многомерных пространствах действий, – нетривиальная проблема. Алгоритм DQN уже решает задачу обучения устойчивой глубокой нейронной сети в постановке с разделенной стратеги-

ей. Чтобы адаптировать DQN к непрерывному пространству действий, следует дискретизировать последнее. Например, если ценности действий принимают значения от 0 до 1, то можно было бы выделить 11 дискретных значений (0, 0.1, 0.2, ..., 0.9, 1.0) и предсказывать их вероятности с помощью DQN. Но это решение не годится для большого пространства действий, потому что количество возможных дискретных действий растет экспоненциально вместе с количеством степеней свободы агента. Кроме того, эта техника неприменима, когда требуется более точное управление. Необходимо что-то другое.

Ценная идея заключается в построении детерминированного алгоритма исполнитель–критик. Она тесно связана с Q-обучением. Напомним, что в Q-обучении наилучшим действием считается то, которое максимизирует приближенную Q-функцию:

$$\max_a Q_\phi(s, a) = Q_\phi(s, \operatorname{argmax}_a Q_\phi(s, a)).$$

Возникает идея обучить детерминированную стратегию $\mu_\theta(s)$, которая аппроксимирует $\operatorname{argmax}_a Q_\phi(s, a)$. Тем самым обходится проблема вычисления глобального максимума на каждом шаге и открывается возможность обобщения на многомерные и непрерывные пространства действий. В алгоритме **детерминированного градиента стратегии (DPG)** эта идея была успешно применена к некоторым простым задачам, например: машина на горе (Mountain Car), балансировка стержня (Pendulum) и щупальце осьминога (Octopus Arm). Алгоритм DDPG развивает идеи DPG, используя в стратегиях глубокие нейронные сети и применяя некоторые более осмотрительные решения для повышения устойчивости. Последующий алгоритм TD3 решает проблему высокой дисперсии и завышенной оценки, присущую DPG и DDPG. Ниже будут описаны алгоритмы DDPG и TD3. На диаграмме классификации алгоритмов ОП алгоритмы DPG, DDPG и TD3 занимают место между алгоритмами градиента стратегии и Q-обучения (рис. 8.1). Далее мы рассмотрим принципы работы алгоритмов DPG.

В новых алгоритмах DPG сочетаются методы Q-обучения и градиента стратегии. Параметрическая детерминированная стратегия выводит только детерминированные ценности. В непрерывных контекстах это может быть среднее действие. Затем параметры стратегии обновляются, преследуя следующую цель:

$$\theta \leftarrow \operatorname{argmax}_\theta Q_\phi(s, \mu_\theta(s)), \quad (8.1)$$

где Q_ϕ – параметрическая функция ценности действий. Отметим, что детерминированные подходы отличаются от стохастических отсутствием добавленного к действиям шума. В алгоритмах PPO и TRPO мы производили выборку из нормального распределения с заданным средним и стандартным отклонением. Здесь же стратегия имеет только детерминированное среднее. В формуле обновления (8.1) максимизация, как всегда, производится методом стохастического градиентного подъема, при этом стратегия постепенно улучшается в результате мелких шагов. Градиент целевой функции вычисляется по формуле

$$\nabla_\theta J(\mu_\theta) = E_{s \sim p^\mu} \left[\nabla_\theta \mu_\theta(s) \nabla_a Q_\phi(s, a) \Big|_{a=\mu_\theta} \right], \quad (8.2)$$

где p^μ – распределение состояний в соответствии со стратегией μ . Это следует из теоремы о детерминированном градиенте стратегии, которая утверждает, что градиент целевой функции получается применением к Q-функции правила дифференцирования сложной функции по параметрам стратегии θ . С помощью средств автоматизированного дифференцирования, в т. ч. имеющихся в TensorFlow, он вычисляется очень легко. Порядок применения этой теоремы показан на диаграмме ниже (рис. 8.2).

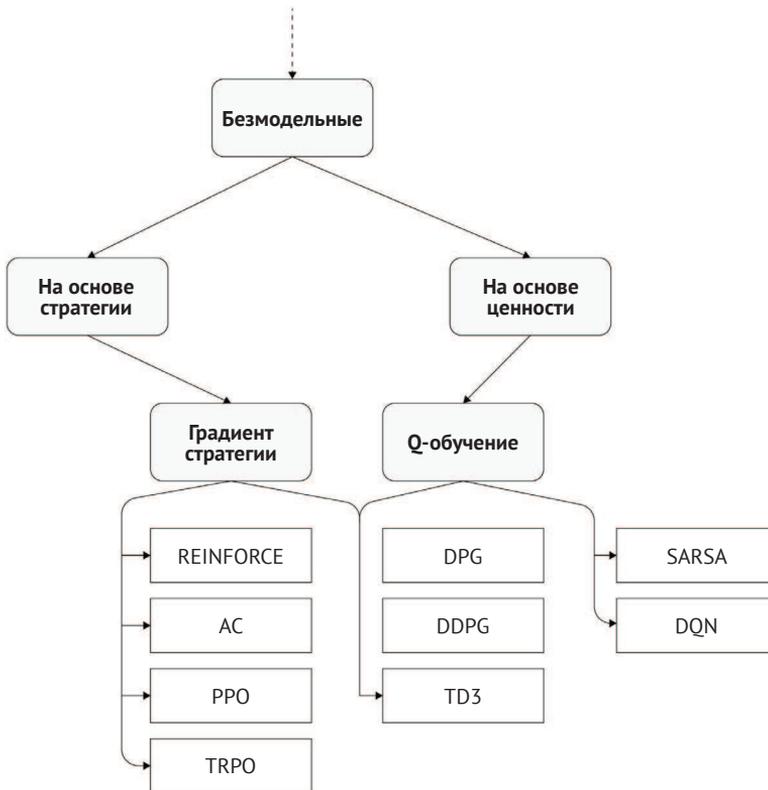


Рис. 8.1 ❖ Классификация уже известных нам безмодельных алгоритмов ОП

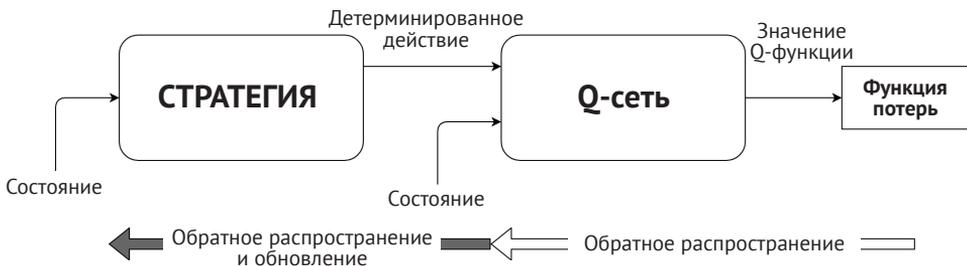


Рис. 8.2 ❖ Иллюстрация теоремы о DPG

❗ Градиент вычисляется начиная со значений Q -функции, но обновляется только стратегия.

Этот результат носит больше теоретический характер. Как мы знаем, детерминированные стратегии не исследуют окружающую среду и потому не находят хорошего решения. Чтобы превратить DPG в алгоритм с разделенной стратегией, нужно сделать еще один шаг и определить градиент целевой функции, так чтобы математическое ожидание вычислялось по распределению стохастической исследовательской стратегии:

$$\nabla_{\theta} J_{\beta}(\mu_{\theta}) \approx E_{s \sim p^{\beta}} \left[\nabla_{\theta} \mu_{\theta}(s) \nabla_a Q_{\phi}(s, a) \Big|_{a=\mu_{\theta}} \right]. \quad (8.3)$$

Здесь β – исследовательская стратегия, называемая также поведенческой. Это уравнение дает *детерминированный градиент стратегии с разделенной стратегией*, оно оценивает градиент относительно детерминированной стратегии (μ), тогда как траектории порождаются, следуя поведенческой стратегией (β). Отметим, что на практике поведенческая стратегия – это просто детерминированная стратегия с добавленным шумом.

Хотя раньше мы говорили о детерминированном алгоритме исполнитель–критик, до сих пор обсуждалось только, как происходит обучение стратегии. Но на самом деле обучается как исполнитель, представленный детерминированной стратегией (μ_{θ}), так и критик, представленный Q -функцией (Q_{ϕ}). Дифференцируемую функцию ценности действий Q_{ϕ} легко обучить при помощи обновления Беллмана, минимизирующего беллмановскую ошибку ($\delta_t = r_t + \gamma Q_{\phi}(s_{t+1}, a_{t+1}) - Q_{\phi}(s_t, a_t)$), как делается в алгоритме Q -обучения.

Алгоритм DDPG

Алгоритм DPG с глубокой нейронной сетью, описанный в предыдущем разделе, очень неустойчив и вряд ли способен чему-то научиться. Мы сталкивались с подобной проблемой, когда включали в Q -обучение глубокие нейронные сети. И чтобы объединить ГНС с Q -обучением в алгоритме DQN, нам пришлось пойти на некоторые уловки, дабы стабилизировать обучение. То же самое относится к алгоритмам DPG. Эти методы, как и Q -обучение, основаны на разделенной стратегии, и, как мы скоро увидим, некоторые элементы, благодаря которым детерминированные стратегии работают совместно с ГНС, аналогичны использованным в DQN.

Алгоритм DDPG (описанный в статье Lillicrap, et al. «Continuous Control with Deep Reinforcement Learning», <https://arxiv.org/pdf/1509.02971.pdf>) стал первым детерминированным алгоритмом типа исполнитель–критик, в котором глубокие нейронные сети используются для обучения как исполнителя, так и критика. Этот безмодельный алгоритм с разделенной стратегией обобщает алгоритмы DQN и DPG в том смысле, что заимствует из DQN некоторые идеи, например буфер воспроизведения и целевую сеть, чтобы заставить DPG работать с глубокими нейронными сетями.

В DDPG есть две ключевые идеи, и обе заимствованы из DQN, но адаптированы для алгоритма типа исполнитель–критик.

- **Буфер воспроизведения.** Информация обо всех переходах, собранных за время существования агента, сохраняется в буфере воспроизведения. Затем из этого буфера выбираются мини-пакеты для обучения исполнителя и критика.
- **Целевая сеть.** Q-обучение неустойчиво, т. к. обновляемая сеть используется также для вычисления целевых значений. Напомним, что в DQN острота этой проблемы сглаживается путем использования целевой сети, которая обновляется через каждые N итераций (параметры онлайн-сети копируются в параметры целевой). В статье, посвященной DDQN, показано, что в этом случае лучше работает мягкое обновление целевой сети. Это означает, что параметры целевой сети θ' на каждом шаге частично обновляются параметрами онлайн-сети θ : $\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$, где $\tau \ll 1$. Да, это может замедлить обучение, поскольку целевая сеть изменяется лишь частично, но зато устойчивость повышается. Этот трюк – использование целевой сети – применяется как для исполнителя, так и для критика, причем параметры целевой сети критика обновляются по правилу $\phi' \leftarrow \tau\phi + (1 - \tau)\phi'$.

Начиная с этого момента мы будем обозначать θ и ϕ параметры онлайн-вого исполнителя и онлайн-вого критика соответственно, а θ' и ϕ' – параметры целевого исполнителя и целевого критика.

От DQN алгоритм DDPG унаследовал возможность обновлять исполнителя и критика на каждом шаге взаимодействия с окружающей средой. Это следует из того, что DDPG – алгоритм с разделенной стратегией, обучающийся на мини-пакетах, выбранных из буфера обновления. DDPG не должен ждать, пока наберется достаточно большая выборка из среды, как то было бы в случае стохастического метода градиента стратегии с единой стратегией.

Ранее мы видели, что DPG действует в соответствии с исследовательской поведенческой стратегией, несмотря на то что по-прежнему обучает детерминированную стратегию. Но как строится эта исследовательская стратегия? В DDPG стратегия строится путем прибавления шума, выбранного из случайного процесса шума (N):

$$\beta_{\theta}(s_t) = \mu_{\theta}(s_t) + N.$$

Процесс N гарантирует достаточную степень исследования среды.

Подведем итоги. DDPG обучается, циклически повторяя следующие три шага до достижения сходимости:

- поведенческая стратегия взаимодействует со средой, накапливая наблюдения и вознаграждения и сохраняя их в буфере;
- на каждом шаге исполнитель и критик обновляются, пользуясь информацией, которая содержится в мини-пакете, выбранном из буфера. Точнее, для обновления критика минимизируется функция потерь, равная среднеквадратической ошибке (СКО) между ценностями, предсказанными онлайн-вым критиком (Q_{ϕ}), и целевыми ценностями, вычисленными с использованием целевой стратегии ($\mu_{\theta'}$) и целевого критика ($Q_{\phi'}$). Исполнитель же обновляется по формуле (8.3);
- к параметрам целевой сети применяется мягкое обновление.

Ниже приведен псевдокод алгоритма:

 Алгоритм DDPG

Инициализировать онлайнные сети Q_ϕ и μ_θ

Инициализировать целевые сети $Q_{\phi'}$ и $\mu_{\theta'}$ с такими же весами, как онлайнные

Инициализировать пустой буфер воспроизведения D

Инициализировать окружающую среду $s \leftarrow env.reset()$

for $episode = 1..M$ **do**

> Выполнить эпизод

while not d :

$a \leftarrow \mu_\beta(s)$

$s', r, d \leftarrow env(a)$

> Сохранить переход в буфере

$D \leftarrow D \cup (s, a, r, s', d)$

$s \leftarrow s'$

> Выбрать мини-пакет

$b \sim D$

> Вычислить целевую ценность для каждого i в b

$y_i \leftarrow r_i + \gamma(1 - d_i)Q_{\phi'}(s'_i, \mu_{\theta'}(s'_i))$ (8.4)

> Обновить критика

$\phi \leftarrow \phi - \alpha_\phi \nabla_\phi \frac{1}{|b|} \sum_i (Q_\phi(s_i, a_i) - y_i)^2$ (8.5)

> Обновить стратегию

$\theta \leftarrow \theta - \alpha_\theta \frac{1}{|b|} \sum_i \nabla_\theta \mu_\theta(s_i) \nabla_a Q_\phi(s_i, a_i) \Big|_{a=\mu_\theta(s_i)}$ (8.6)

> Обновить параметры целевых сетей

$\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$

$\phi' \leftarrow \tau\phi + (1 - \tau)\phi'$

если $d == \text{True}$:

$s \leftarrow env.reset()$

Разобравшись в устройстве алгоритма, мы можем приступить к его реализации.

Реализация DDPG

Псевдокод, приведенный в предыдущем разделе, уже дает достаточно полное представление об алгоритме, но, с точки зрения реализации, некоторые моменты заслуживают большего внимания. Мы акцентируем внимание на интересных особенностях, которые могут встретиться и в других алгоритмах. Полный код имеется в репозитории этой книги на GitHub по адресу <https://github.com/PacktPublishing/Reinforcement-Learning-Algorithmswith-Python>.

Конкретно, мы рассмотрим следующие части:

- как строятся детерминированные исполнитель и критик;
- как выполняется мягкое обновление;

- как оптимизируется функция потерь относительно лишь некоторых параметров;
- как вычисляются целевые ценности.

Мы определяем детерминированного исполнителя и критика в функции `deterministic_actor_critic`. Эта функция вызывается дважды, потому что нужно создать как онлайн-овых, так и целевых исполнителя и критика. Вот ее код.

```
def deterministic_actor_critic(x, a, hidden_sizes, act_dim, max_act):
    with tf.variable_scope('p_mlp'):
        p_means = max_act * mlp(x, hidden_sizes, act_dim, last_activation=tf.tanh)
    with tf.variable_scope('q_mlp'):
        q_d = mlp(tf.concat([x, p_means], axis=-1), hidden_sizes, 1,
last_activation=None)
        with tf.variable_scope('q_mlp', reuse=True): # повторно используем веса
            q_a = mlp(tf.concat([x, a], axis=-1), hidden_sizes, 1, last_activation=None)
        return p_means, tf.squeeze(q_d), tf.squeeze(q_a)
```

Здесь стоит обратить внимание на три вещи. Во-первых, мы различаем типы входных данных одного и того же критика. Один принимает состояние и детерминированное действие `p_means`, возвращенное стратегией; другой – состояние и произвольное действие. Это различие необходимо, потому что один критик будет использоваться для оптимизации исполнителя, а другой – для оптимизации критика. Но хотя эти два критика принимают разные данные на входе, они пользуются одной и той же нейронной сетью, а значит, одними и теми же параметрами. Это достигается путем определения одной и той же области видимости `variable_scope` для обоих экземпляров критика, но для второго задается аргумент `reuse=True`. Тем самым гарантируется, что в обоих определениях параметры одинаковы, т. е. по существу мы создаем только одного критика.

Второй момент заключается в том, что исполнитель создается внутри области видимости `p_mlp`. Это необходимо, потому что впоследствии нам понадобятся только эти параметры, а не параметры критика.

И третий момент – поскольку в последнем слое нейронной сети стратегии используется функция активации `tanh` (чтобы выходные значения не выходили за пределы диапазона от -1 до 1), а исполнителю могут потребоваться значения вне этого диапазона, то необходимо умножить выход на коэффициент `max_act` (при этом предполагается, что минимум и максимум равны по абсолютной величине, но противоположны по значению, т. е. если максимум равен 3 , то минимум равен -3).

Прелестно! Теперь посмотрим на остальные части графа вычислений, где определяются местозаместители, создаются онлайн-овый и целевой исполнители и критики, определяются функции потерь, реализуются оптимизаторы и обновляются целевые сети.

Начнем с создания местозаместителей, которые будут нужны для наблюдений, действий и целевых ценностей:

```
obs_dim = env.observation_space.shape
act_dim = env.action_space.shape

obs_ph = tf.placeholder(shape=(None, obs_dim[0]), dtype=tf.float32, name='obs')
```

```
act_ph = tf.placeholder(shape=(None, act_dim[0]), dtype=tf.float32, name='act')
y_ph = tf.placeholder(shape=(None,), dtype=tf.float32, name='y')
```

Здесь `y_ph` – местозаменитель для целевых значений Q-функции, `obs_ph` – для наблюдений, а `act_ph` – для действий.

Далее мы вызываем уже написанную функцию `deterministic_actor_critic` в областях видимости `online` и `target`, чтобы различить все четыре нейронные сети:

```
with tf.variable_scope('online'):
    p_onl, qd_onl, qa_onl = deterministic_actor_critic(obs_ph, act_ph,
    hidden_sizes, act_dim[0], np.max(env.action_space.high))
with tf.variable_scope('target'):
    _, qd_tar, _ = deterministic_actor_critic(obs_ph, act_ph, hidden_sizes,
    act_dim[0], np.max(env.action_space.high))
```

Потерей критика является среднеквадратическая ошибка между значениями Q-функции онлайн-сети `qa_onl` и целевыми ценностями действий `y_ph`:

```
q_loss = tf.reduce_mean((qa_onl - y_ph)**2)
```

Эта потеря минимизируется с помощью алгоритма Adam:

```
q_opt = tf.train.AdamOptimizer(cr_lr).minimize(q_loss)
```

Что касается функции потерь исполнителя, то она противоположна по знаку целевой функции онлайн-сети. В нашем случае на вход онлайн-сети подаются действия, выбранные онлайн-сетью детерминированным исполнителем (см. формулу (8.6)). Следовательно, значения Q-функции представлены сетью `qd_onl`, и функция потерь стратегии имеет вид:

```
p_loss = -tf.reduce_mean(qd_onl)
```

Знак минус нужен, потому что мы должны преобразовать целевую функцию в функцию потерь, т. к. оптимизаторы, по определению, минимизируют функцию потерь.

Теперь самое главное – хотя вычисляется градиент функции потерь `p_loss`, зависящей и от критика, и от исполнителя, обновлять нужно только исполнителя. Действительно, из описания алгоритма DPG мы знаем, что

$$\nabla_{\theta} J_{\beta}(\mu_{\theta}) \approx E_{s \sim p^{\beta}} \left[\nabla_{\theta} \mu_{\theta}(s) \nabla_a Q_{\phi}(s, a) \Big|_{a=\mu_{\theta}} \right].$$

Таким образом, мы должны передать `p_loss` методу `minimize` оптимизатора вместе с переменными, нуждающимися в обновлении. В данном случае обновить нужно только параметры онлайн-исполнителя, который был определен в области видимости `online/p_mlp`:

```
p_opt = tf.train.AdamOptimizer(ac_lr).minimize(p_loss,
var_list=variables_in_scope('online/p_mlp'))
```

Стало быть, вычисление градиента начинается с `p_loss`, проходит через сеть критика, а затем через сеть исполнителя. В итоге будут оптимизированы только параметры исполнителя.

Теперь нужно определить функцию `variable_in_scope(scope)`, которая возвращает переменные в области видимости `scope`:

```
def variables_in_scope(scope):
    return tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES, scope)
```

Настало время посмотреть, как обновляются целевые сети. С помощью `variable_in_scope` мы можем получить целевые и онлайн-параметры исполнителей и критиков и, воспользовавшись функцией TensorFlow `assign`, обновить целевые параметры по формуле мягкого обновления:

$$\theta' \leftarrow \tau\theta + (1 - \tau)\theta'.$$

Это делается следующим образом:

```
update_target = [target_var.assign(tau*online_var + (1-tau)*target_var) for
target_var, online_var in zip(variables_in_scope('target'),
variables_in_scope('online'))]
update_target_op = tf.group(*update_target)
```

Все! С графом вычислений покончено. Не слишком сложно, правда? Теперь займемся главным циклом, в котором параметры обновляются в направлении оценки градиента на конечном пакете примеров. Взаимодействие стратегии с окружающей средой стандартное – отличие только в том, что теперь возвращенные стратегией действия детерминированные и нужно добавить шум, чтобы не останавливалось исследование среды. Эту часть кода мы здесь не приводим, но ее можно найти на GitHub.

После того как накоплен минимально необходимый опыт и буфер достиг определенного порогового размера, начинается оптимизация стратегии и критика. При этом последовательность шагов такая, как в приведенном выше псевдокоде DDPG, а именно:

- 1) произвести выборку мини-пакета из буфера;
- 2) вычислить ценности целевых действий;
- 3) оптимизировать критика;
- 4) оптимизировать исполнителя;
- 5) обновить целевые сети.

Для выполнения всех этих операций достаточно всего нескольких строчек кода:

```
...
mb_obs, mb_rew, mb_act, mb_obs2, mb_done = buffer.sample_minibatch(batch_size)

q_target_mb = sess.run(qd_tar, feed_dict={obs_ph:mb_obs2})
y_r = np.array(mb_rew) + discount*(1-np.array(mb_done))*q_target_mb

_, q_train_loss = sess.run([q_opt, q_loss], feed_dict={obs_ph:mb_obs,
y_ph:y_r, act_ph: mb_act})

_, p_train_loss = sess.run([p_opt, p_loss], feed_dict={obs_ph:mb_obs})
sess.run(update_target_op)
...
```

В первой строке выбирается мини-пакет размера `batch_size`, а во второй и третьей вычисляются ценности целевых действий по формуле (8.4), для чего целевые сети критика и исполнителя применяются к массиву `mb_obs2`, содержащему следующие состояния. В четвертой строке оптимизируется критик, для чего на вход подается словарь, содержащий только что вычисленные ценности целевых действий, а также наблюдения и действия. В пятой строке оптимизируется исполнитель, а в последней целевые сети обновляются функцией `update_target_op`.

Применение DDPG к среде `BipedalWalker-v2`

Теперь применим алгоритм DDPG к непрерывной задаче `BipedalWalker-v2` – одной из окружающих сред `Gym`, в которой используется двумерный физический движок `Box2D`. На рис. 8.3 показано, как выглядит эта среда. Наша цель – научить агента как можно быстрее ходить по неровному рельефу. Если он дойдет до конца, то получит вознаграждение $+300$, но каждое включение моторов обходится в небольшую сумму. Чем оптимальнее двигается агент, тем меньше будет эта сумма. За падение начисляется штраф -100 . Состояние включает 24 числа с плавающей точкой, которые представляют скорости и положения сочленений и корпуса, а также результаты измерений лазерного дальномера. Для управления агентом предусмотрено четыре непрерывных действия с диапазоном значений $[-1, 1]$.

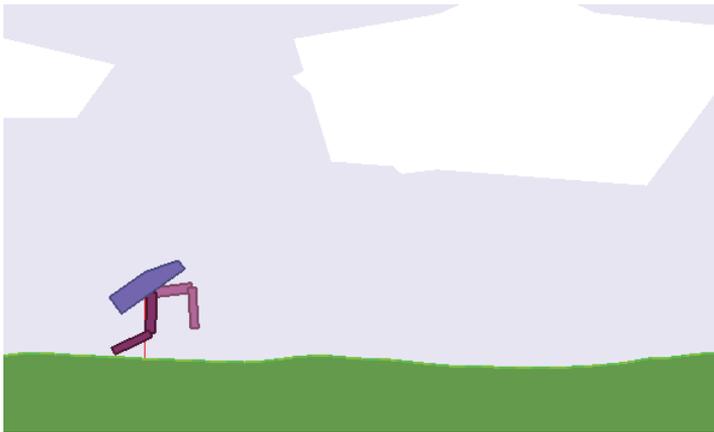


Рис. 8.3 ❖ Окружающая среда `BipedalWalker2d`

Мы прогоняли DDPG с гиперпараметрами, показанными в таблице ниже.

Гиперпараметр	Скорость обучения исполнителя	Скорость обучения критика	Архитектура ГНС	Размер буфера	Размер пакета	Тай
Значение	$3e-4$	$4e-4$	[64,relu,64,relu]	200 000	64	0.003

В процессе обучения мы прибавляли шум к действиям, предсказанным стратегией. Но, чтобы измерить качество алгоритма, мы прогоняли 10 игр на

чистой детерминированной стратегии (без шума) через каждые 10 эпизодов. На рис. 8.4 приведен график зависимости полного вознаграждения, усредненного по 10 играм, от количества временных шагов.

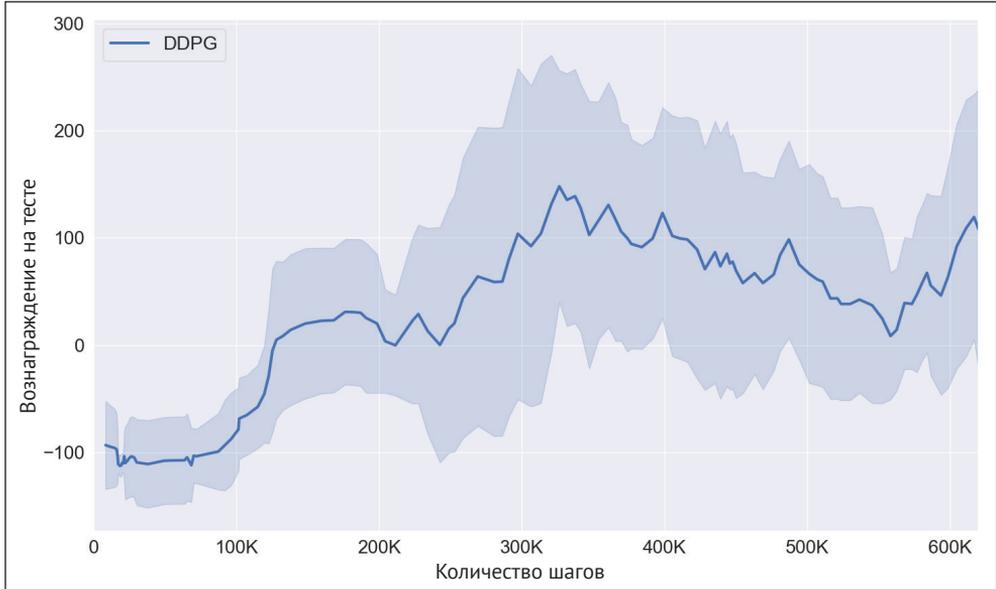


Рис. 8.4 ❖ Качество алгоритма DDPG в среде BipedalWalker2d-v2

Из результатов видно, что алгоритм крайне неустойчив, вознаграждение варьируется от 250 до менее -100 (после всего нескольких тысяч шагов). Известно, что DDPG неустойчив и очень чувствителен к значениям гиперпараметров, но при более тщательной настройке результаты можно улучшить. Так или иначе, мы видим, что на первых 300 000 шагов качество возрастает и достигает среднего значения 100, а в пике доходит до 300.

Добавим, что среда BipedalWalker-v2 действительно трудна для обучения. Успехом считается, когда среднее вознаграждение составляет не менее 300 баллов в 100 последовательных эпизодах. DDPG не позволяет дойти до таких высот, но тем не менее мы получили неплохую стратегию, следуя которой, агент бежит достаточно быстро.

i В нашей реализации коэффициент исследования был постоянным. Используя более сложную функцию, возможно, удалось бы добиться более высокого качества за меньшее число итераций. Например, авторы статьи, где описывается DDPG, применили процесс Орнштейна–Уленбека. Если хотите, можете начать с него.

DDPG – прекрасный пример использования детерминированной стратегии взамен стохастической. Но это первая попытка применить такой подход к решению сложных задач, так что возможно много улучшений. Следующий алгоритм, описанный в этой главе, – шаг вперед по сравнению с DDPG.

Алгоритм TD3

DDPG считается одним из самых выборочно эффективных алгоритмов типа исполнитель–критик, но, как мы показали, он неустойчив и чувствителен к значениям гиперпараметров. В последующих работах были предприняты попытки решить эти проблемы – с помощью новых идей или применения к DDPG приемов, заимствованных у других алгоритмов. В последнее время на замену DDPG пришел другой алгоритм: двойной глубокий детерминированный градиент стратегии с задержкой (TD3). Он описан в статье «Addressing Function Approximation Error in Actor-Critic Methods» (<https://arxiv.org/pdf/1802.09477.pdf>). Слово «замена» здесь означает, что фактически это тот же алгоритм DDPG, но с некоторыми добавлениями, которые повышают его устойчивость и качество.

В фокусе внимания TD3 находятся проблемы, общие для всех алгоритмов с разделенной стратегией. Это завышенная оценка ценности и высокая дисперсия оценок градиента. Для решения первой проблемы использована примерно такая же техника, как в DQN, а для решения второй предложено две новаторские идеи. Сначала рассмотрим проблему завышенной оценки.

Проблема смещения оценки в сторону завышения

Завышение оценки означает, что ценности действий, предсказанные приближенной Q-функцией, выше, чем должны быть. Эта проблема была хорошо изучена для алгоритмов Q-обучения с дискретными действиями, и показано, что она часто приводит к неверным предсказаниям, влияющим на итоговое качество алгоритма. Проблема присутствует и в DDPG, хотя и не так сильно выражена.

Напомним, что существует вариант DQN (двойной DQN, или DDQN), который уменьшает завышение оценки благодаря использованию двух нейронных сетей: одна выбирает действия, другая вычисляет значение Q-функции. При этом целевая сеть, используемая для решения второй задачи, замораживается, т. е. обновляется не так часто, как первая. Это здравая идея, но, как объяснено в статье, посвященной TD3, для методов исполнитель–критик она не эффективна, потому что стратегия изменяется слишком медленно. Поэтому авторы предложили вариацию на эту тему, назвав ее *обрезанным двойным Q-обучением*. Смысл в том, что берется минимум оценок, вычисленных двумя разными критиками (Q_{ϕ_1} , Q_{ϕ_2}). То есть целевая ценность вычисляется следующим образом:

$$y = r + \gamma \min_{i=1,2} Q_{\phi_i}(s', \mu_{\theta'}(s')). \quad (8.7)$$

Это, правда, не предотвращает занижения оценки, но и не так опасно, как завышение. Идею обрезанного двойного Q-обучения можно применить в любом методе исполнитель–критик, нужно только, чтобы у двух критиков были разные смещения.

Реализация TD3

Для реализации этой идеи необходимо создать двух критиков с разными начальными весами, вычислить ценность целевого действия по формуле (8.7) и оптимизировать обоих.

❗ TD3 применяется к реализации DDPG из предыдущего раздела. Ниже приведен только дополнительный код, необходимый для TD3. Полный код имеется в репозитории этой книги на GitHub по адресу <https://github.com/PacktPublishing/Hands-On-Reinforcement-Learning-Algorithms-with-Python>.

Для создания критиков нужно дважды вызвать функцию `deterministic_actor_double_critic`, по разу для целевой и для онлайн-сетей, как в DDPG. Код выглядит так:

```
def deterministic_actor_double_critic(x, a, hidden_sizes, act_dim, max_act):
    with tf.variable_scope('p_mlp'):
        p_means = max_act * mlp(x, hidden_sizes, act_dim, last_activation=tf.tanh)
        # Первый критик
        with tf.variable_scope('q1_mlp'):
            q1_d = mlp(tf.concat([x,p_means], axis=-1), hidden_sizes, 1,
last_activation=None)
            with tf.variable_scope('q1_mlp', reuse=True): # Использовать те же веса, что
в только что определенном mlp
                q1_a = mlp(tf.concat([x,a], axis=-1), hidden_sizes, 1, last_activation=None)

            # Второй критик
            with tf.variable_scope('q2_mlp'):
                q2_d = mlp(tf.concat([x,p_means], axis=-1), hidden_sizes, 1,
last_activation=None)
            with tf.variable_scope('q2_mlp', reuse=True):
                q2_a = mlp(tf.concat([x,a], axis=-1), hidden_sizes, 1, last_activation=None)

        return p_means, tf.squeeze(q1_d), tf.squeeze(q1_a), tf.squeeze(q2_d),
tf.squeeze(q2_a)
```

Для вычисления обрезанной целевой ценности по формуле (8.7) (*90*) мы сначала выполняем целевые сети обоих критиков, `qa1_tar` и `qa2_tar`, затем берем минимум оценок ценностей и, наконец, используем его для вычисления целевых ценностей:

```
...
double_actions = sess.run(p_tar, feed_dict={obs_ph:mb_obs2})

q1_target_mb, q2_target_mb = sess.run([qa1_tar,qa2_tar],
feed_dict={obs_ph:mb_obs2, act_ph:double_actions})
q_target_mb = np.min([q1_target_mb, q2_target_mb], axis=0)
y_r = np.array(mb_rew) + discount*(1-np.array(mb_done))*q_target_mb
...
```

Далее производится обычная оптимизация критиков:

```
...
q1_train_loss, q2_train_loss = sess.run([q1_opt, q2_opt],
feed_dict={obs_ph:mb_obs, y_ph:y_r, act_ph: mb_act})
...
```

Важно, что стратегия оптимизируется только относительно одной приближенной Q -функции, в нашем случае Q_{ϕ_1} . Если вы заглянете в полный код, то увидите, что `p_loss` определена так: `p_loss = -tf.reduce_mean(qd1_onl)`.

Уменьшение дисперсии

Второй и последний вклад TD3 – уменьшение дисперсии. Почему высокая дисперсия составляет проблему? Потому что получается зашумленный градиент, а значит, стратегия обновляется неправильно, что влечет ухудшение качества алгоритма. Проблема высокой дисперсии проявляется при вычислении TD-ошибки, которая оценивает ценности действий по последующим состояниям.

Чтобы сгладить остроту проблемы, в TD3 реализовано отложенное обновление стратегии и техника регуляризации целевой сети. Посмотрим, что это такое и почему дает хорошие результаты.

Отложенное обновление стратегии

Поскольку высокая дисперсия связана с неточностью критика, в TD3 предлагается отложить обновление стратегии до момента, когда ошибка критика станет достаточно малой. Задержка выбирается эмпирически, точнее стратегия обновляется только после фиксированного числа итераций. Таким образом, у критика есть время для обучения и стабилизации, прежде чем начнется оптимизация стратегии. На практике количество итераций невелико – от 1 до 6. Если оно равно 1, то мы получаем в точности DDPG. Вот как реализуется отложенное обновление стратегии:

```
...
q1_train_loss, q2_train_loss = sess.run([q1_opt, q2_opt],
feed_dict={obs_ph:mb_obs, y_ph:y_r, act_ph: mb_act})
if step_count % policy_update_freq == 0:
    sess.run(p_opt, feed_dict={obs_ph:mb_obs})
    sess.run(update_target_op)
...
```

Регуляризация целевой сети

Для критиков, обновляемых на основе детерминированных действий, характерно переобучение в узких пиках. Поэтому дисперсия и увеличивается. TD3 предлагает технику сглаживающей регуляризации – добавление обрезанного шума в небольшой области вокруг целевого действия:

$$y = r + \gamma \min_{i=1,2} Q_{\phi_i}(s', \mu_{\theta'}(s') + \varepsilon),$$

$$\varepsilon \sim \text{clip}(N(0, \sigma), -c, c).$$

Регуляризацию можно реализовать с помощью функции, которая принимает вектор и масштаб:

```
def add_normal_noise(x, noise_scale):
    return x + np.clip(np.random.normal(loc=0.0, scale=noise_scale,
size=x.shape), -0.5, 0.5)
```

Функция `add_normal_noise` вызывается после выполнения целевой стратегии, как показано ниже (изменения относительно реализации DDPG выделены полужирным шрифтом):

```
...
double_actions = sess.run(p_tar, feed_dict={obs_ph:mb_obs2})
```

```

double_noisy_actions = np.clip(add_normal_noise(double_actions,
target_noise), env.action_space.low, env.action_space.high)

q1_target_mb, q2_target_mb = sess.run([qa1_tar, qa2_tar],
feed_dict={obs_ph:mb_obs2, act_ph:double_noisy_actions})
q_target_mb = np.min([q1_target_mb, q2_target_mb], axis=0)
y_r = np.array(mb_rew) + discount*(1-
np.array(mb_done))*q_target_mb
...

```

После прибавления шума мы обрезали действия, чтобы не выходить за пределы диапазона, заданного окружающей средой.

Собирая все вместе, получаем алгоритм, описанный в псевдокоде ниже.

Алгоритм TD3

Инициализировать онлайнные сети Q_{ϕ_1} , Q_{ϕ_2} и μ_{θ}
Инициализировать целевые сети $Q_{\phi'_1}$, $Q_{\phi'_2}$ и $\mu_{\theta'}$ с такими же весами, как онлайнные
Инициализировать пустой буфер воспроизведения D
Инициализировать окружающую среду $s \leftarrow env.reset()$

```

for episode = 1..M do
  > Выполнить эпизод
  while not d:
    a ← μθ(s)
    s', r, d ← env(a)
    > Сохранить переход в буфере
    D ← D ∪ (s, a, r, s', d)
    s ← s'

    > Выбрать мини-пакет
    b ~ D

    > Вычислить целевую ценность для каждого i в b
    y ← r + γ mini=1,2 Qφ'_i(s', μθ(s') + ε),
    ε ~ clip(N(0, σ), -c, c).

    > Обновить критиков
    φ1 ← φ1 - αφ ∇φ1  $\frac{1}{|b|} \sum_i (Q_{\phi_1}(s_i, a_i) - y_i)^2$ 
    φ2 ← φ2 - αφ ∇φ2  $\frac{1}{|b|} \sum_i (Q_{\phi_2}(s_i, a_i) - y_i)^2$ 

    если iter % policy_update_frequency == 0:
      > Обновить стратегию
      θ ← θ - αθ  $\frac{1}{|b|} \sum_i \nabla_{\theta} \mu_{\theta}(s_i) \nabla_a Q_{\phi_i}(s_i, a_i) \Big|_{a=\mu(s_i)}$ 

      > Обновить параметры целевых сетей
      θ' ← τθ + (1 - τ)θ'
      φ'_1 ← τφ1 + (1 - τ)φ'_1
      φ'_2 ← τφ2 + (1 - τ)φ'_2

    если d == True:
      s ← env.reset()

```

Это все, что мы хотели сказать об алгоритме TD3. Теперь вы знаете обо всех детерминированных и недетерминированных методах градиента стратегии. Почти все безмодельные алгоритмы основаны на описанных выше принципах, и если вы хорошо их понимаете, то сможете разобраться в этих алгоритмах и реализовать их.

Применение TD3 к среде VipedalWalker-v2

Чтобы сравнить алгоритмы TD3 и DDPG, мы протестировали TD3 в той же окружающей среде, что DDPG: VipedalWalker-v2.

В таблице ниже приведены наилучшие значения гиперпараметров TD3 для этой среды:

Гипер-параметр	Скорость обучения исполнителя	Скорость обучения критика	Архитектура ГНС	Размер буфера	Размер пакета	Tau	Частота обновления стратегии	Сигма
Значение	$4e-4$	$4e-4$	[64,relu,64,relu]	200 000	64	0.005	2	0.2

Результат показан на рис. 8.5. Кривая более гладкая, хорошие результаты получаются уже после примерно 300 000 шагов, а пик приходится на 450 000 шагов обучения. Он очень близок к заветной цели 300 баллов, но все же не достигает ее.

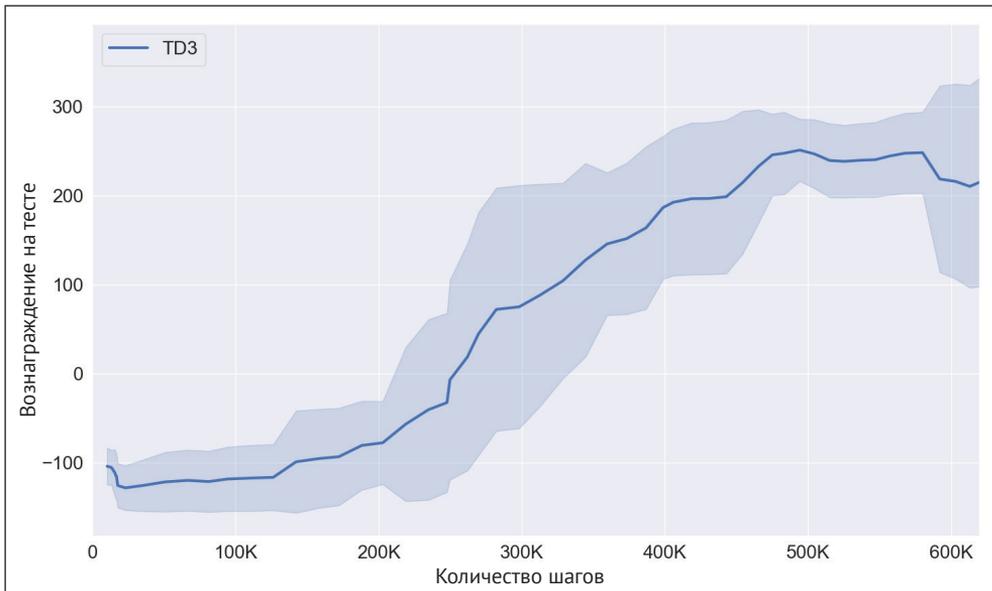


Рис. 8.5 ❖ Качество алгоритма TD3 в среде VipedalWalker-v2

Для подбора хороших значений гиперпараметров TD3 понадобилось меньше времени, чем для DDPG. И хотя мы сравнивали алгоритмы только на одной среде, думается, что полученные результаты дают неплохое представление об

их различиях в плане устойчивости и качества. Ни рис. 8.6 на один график нанесено качество обоих алгоритмов.

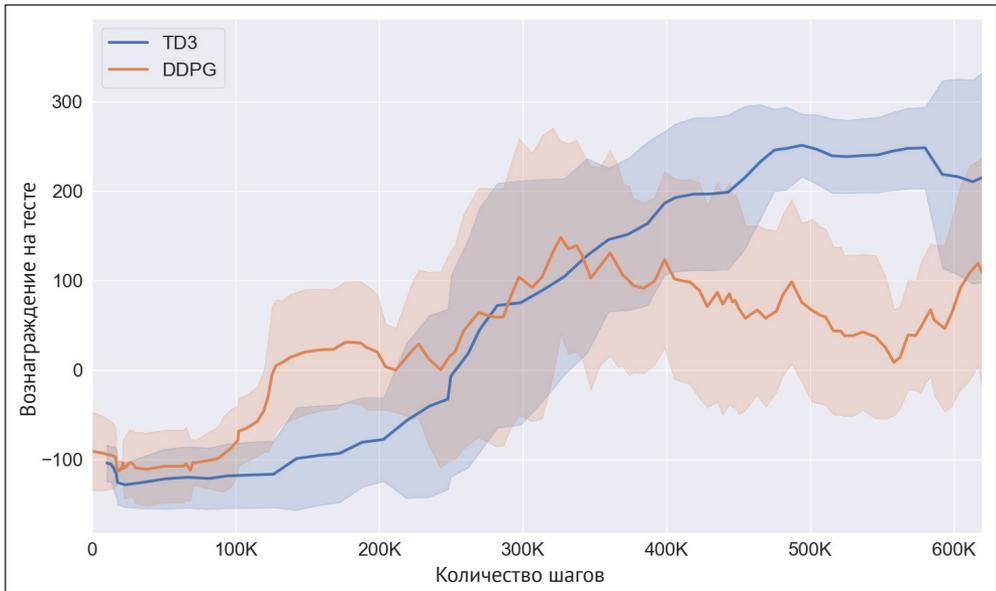


Рис. 8.6 ❖ Сравнение качества DDPG и TD3

i Если вы хотите обучить алгоритм в еще более трудной окружающей среде, попробуйте *BipedalWalkerHardcore-v2*. Она похожа на *BipedalWalker-v2*, но содержит также ступеньки, пни и ямы. Забавно наблюдать, как агент терпит неудачу, пытаясь преодолеть препятствия.

Превосходство TD3 над DDPG очевидно с первого взгляда – с точки зрения конечного результата, скорости обучения и устойчивости.

РЕЗЮМЕ

В этой главе мы рассмотрели два разных подхода к решению задач ОП. Первый заключается в оценке ценностей пар состояние–действие, используемых для выбора наилучшего следующего действия, – так называемые алгоритмы Q-обучения. Второй связан с максимизацией ожидаемого вознаграждения в результате следования стратегии с использованием ее градиента. Соответствующие алгоритмы называются методами градиента стратегии. Мы объяснили преимущества и недостатки обоих подходов и продемонстрировали, что во многих отношениях они дополняют друг друга. Например, алгоритмы Q-обучения обладают высокой выборочной эффективностью, но неприменимы к непрерывным действиям. С другой стороны, алгоритмам градиента стратегии нужно больше данных, зато они способны управлять агентами с непрерывными действиями. Затем мы познакомились с методами DPG, сочетающими Q-обучение с градиентом стратегии. В частности, такие методы позволя-

ют обойти необходимость глобальной оптимизации в алгоритмах Q-обучения путем предсказания детерминированной стратегии. Мы также видели, как теорема о детерминированном градиенте стратегии дает возможность определить обновление стратегии через градиент Q-функции.

Мы изучили и реализовали два алгоритма DPG: DDPG и TD3. Оба являются алгоритмами типа исполнитель–критик с разделенной стратегией, и оба допускают использование в окружающей среде с непрерывным пространством действий. TD3 основан на DDPG, но включает несколько приемов для уменьшения дисперсии и борьбы с завышением оценки, характерным для алгоритмов Q-обучения.

Эта глава завершает обзор безмодельных алгоритмов обучения с подкреплением. Мы рассмотрели все лучшие из известных на данный момент алгоритмов, от SARSA до DQN и от REINFORCE до PPO, а также их сочетания в таких алгоритмах, как DDPG и TD3. Уже эти алгоритмы способны делать поразительные вещи при условии правильной настройки и наличии достаточно большого объема данных (см. OpenAI Five и AlphaStar). Но это еще не все, что следует знать об ОП. В следующей главе мы расстанемся с безмодельными алгоритмами и продемонстрируем алгоритм, основанный на модели, цель которого – уменьшить объем необходимых для обучения данных, а средство достижения – обучиться модели окружающей среды. А далее мы обсудим еще более передовые методы, в т. ч. подражательное обучение, новые полезные алгоритмы ОП типа ESBAS, а также алгоритмы, не связанные с ОП, а именно эволюционные стратегии.

Вопросы

1. В чем состоит главное ограничение алгоритмов Q-обучения?
2. Почему стохастические алгоритмы градиента стратегии выборочно неэффективны?
3. Как в DPG обходится проблема глобальной максимизации?
4. Как в DPG гарантируется достаточный уровень исследования?
5. Как расшифровывается акроним DDPG? И в чем состоит главный вклад этого алгоритма?
6. Какие проблемы стремится решить алгоритм TD3?
7. Какие новые механизмы используются в TD3?

Для дальнейшего чтения

- Статья, содержащая описание алгоритма DPG, находится по адресу <http://proceedings.mlr.press/v32/silver14.pdf>.
- Статья, содержащая описание алгоритма DDPG, находится по адресу <https://arxiv.org/pdf/1509.02971.pdf>.
- Статья, содержащая описание алгоритма TD3, находится по адресу <https://arxiv.org/pdf/1802.09477.pdf>.
- Краткий обзор основных алгоритмов градиента стратегии можно найти в статье Lilian Weng по адресу <https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html>.

Часть III

ЗА ПРЕДЕЛАМИ БЕЗМОДЕЛЬНЫХ АЛГОРИТМОВ

В этой части мы реализуем алгоритмы, основанные на модели, алгоритмы подражательного обучения и эволюционные стратегии, а также расскажем о некоторых идеях, способных раздвинуть рамки обучения с подкреплением.

Глава 9

ОП на основе модели

Алгоритмы обучения с подкреплением делятся на два больших класса: безмодельные и основанные на модели. Различаются они допущением относительно модели окружающей среды. Безмодельные алгоритмы обучаются стратегии просто на взаимодействиях с окружающей средой, о которой ничего не знают, тогда как основанные на модели алгоритмы обладают глубоким пониманием среды и используют эти знания для выбора следующего действия с учетом динамики модели.

В этой главе мы расскажем о преимуществах и недостатках основанных на модели подходах по сравнению с безмодельными, а также о различиях между случаями, когда модель известна и когда ей нужно обучиться. Последнее важно, поскольку влияет на подход к проблеме и выбор инструментария для ее решения. После введения в тему мы поговорим о более сложных случаях, когда алгоритм на основе модели должен иметь дело с пространствами наблюдений очень высокой размерности, например изображениями.

Далее мы рассмотрим класс алгоритмов, сочетающих основанные на модели и безмодельные методы для обучения как модели, так и стратегии в многомерных пространствах. Мы узнаем, как они устроены и зачем вообще использовать такие методы. Чтобы углубить понимание основанных на модели алгоритмов и в особенности алгоритмов, объединяющих оба подхода, мы разработаем современный алгоритм **оптимизации стратегии в доверительной области с применением ансамбля моделей** (model-ensemble trust region policy optimization – **ME-TRPO**) и применим его к непрерывной задаче балансирования стержня (обратного маятника).

В этой главе рассматриваются следующие вопросы:

- методы на основе модели;
- сочетание обучения на основе модели с безмодельным обучением;
- применение алгоритма ME-TRPO к задаче обратного маятника.

Методы на основе модели

Безмодельные алгоритмы способны обучаться очень сложным стратегиям и достигать целей в трудных и неоднородных средах. Как показано в недавних работах компаний OpenAI (<https://openai.com/five/>) и DeepMind (<https://deepmind.com/blog/article/alphastar-mastering-real-timestrategy-game-starcraft-ii>), этим

алгоритмам подвластно долгосрочное планирование, коллективная работа и адаптация к неожиданным ситуациям в таких играх, как StarCraft и Dota 2.

Обученные агенты побеждали лучших профессиональных игроков. Но есть и серьезный недостаток – чтобы достичь уровня мастера, агент должен сыграть очень много игр. На самом деле для достижения таких результатов алгоритмы пришлось подвергнуть массивному масштабированию, чтобы агенты могли сыграть против себя столько игр, что их суммарная продолжительность составляет несколько сотен лет. Так в чем же корень проблемы?

Если вы обучаете агента на взаимодействии с эмулятором, то можно набрать столько опыта, сколько необходимо. Неприятности начинаются, когда агент обучается в среде, которая по своей медленности и сложности сопоставима с миром, в котором мы живем. В самом деле, не можем же мы ждать сотни лет, прежде чем начнут формироваться какие-то интересные способности. А нельзя ли разработать алгоритм, которому нужно будет не так много взаимодействий с реальной окружающей средой? Можно. И как вы помните, мы уже поднимали этот вопрос при обсуждении безмодельных алгоритмов.

Решение дают алгоритмы с разделенной стратегией. Однако выигрыш сравнительно невелик, его недостаточно для многих реальных задач.

Неудивительно, что ответ (или один из возможных ответов) лежит в плоскости алгоритмов обучения на основе моделей. Помните такой? В главе 3 мы использовали модель окружающей среды в сочетании с динамическим программированием, чтобы обучить агента перемещаться по местности с препятствиями. А поскольку в ДП используется модель окружающей среды, оно считается алгоритмом, основанным на модели.

К сожалению, ДП неприменимо в задачах большой или хотя бы умеренной сложности. Поэтому необходимо изучить другие типы алгоритмов на основе модели, которые хорошо масштабируются и могут быть полезны в более трудной среде.

Общая картина обучения на основе модели

Для начала вспомним, что такое модель. Модель состоит из динамики переходов и вознаграждений от среды. Динамика переходов – это отображение состояния s и действия a на следующее состояние s' .

Располагая этой информацией, мы можем полностью заменить окружающую среду ее моделью. Агент, имеющий доступ к модели, может предсказать свое будущее.

В следующих разделах мы увидим, что модель может быть известной или неизвестной. В первом случае модель используется сама по себе для изучения динамики среды, т. е. модель дает представление, используемое вместо среды. Во втором случае неизвестную модель можно обучить путем прямого взаимодействия со средой. Но поскольку в большинстве случаев мы получаем только приближение к окружающей среде, при использовании такой модели нужно учитывать дополнительные факторы.

Поняв, что такое модель, мы можем задуматься о том, как ее использовать и как она может помочь нам уменьшить количество взаимодействий со средой.

Способ использования модели зависит от двух важных факторов: самой модели и способа выбора действий.

Действительно, как мы только что заметили, модель может быть известна или неизвестна, а действия можно планировать или выбирать с помощью обученной стратегии. Алгоритмы при этом существенно различны, поэтому сначала разберемся с подходами в случае, когда модель известна (т. е. мы уже знаем динамику переходов и вознаграждения от среды).

Известная модель

Если модель известна, то ее можно использовать для имитации полных траекторий и вычисления дохода на каждой из них. Затем выбираются действия, приносящие наибольший доход. Этот подход называется **планированием**, без модели в нем никак не обойтись, потому что она дает информацию, необходимую для порождения следующего состояния (если известны текущее состояние и действие) и вознаграждения.

Алгоритмы планирования используются повсеместно, но те, что нас интересуют, зависят от типа пространства действий. Одни работают с дискретными действиями, другие – с непрерывными.

Для дискретных действий обычно применяются алгоритмы поиска, которые строят решающее дерево типа показанного на рис. 9.1.

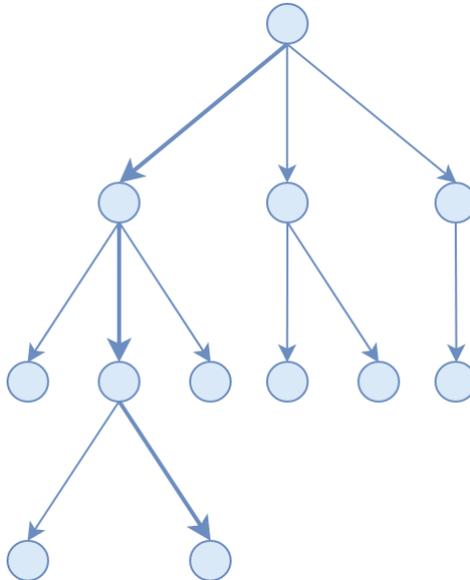


Рис. 9.1

Текущее состояние является корнем, возможные действия представлены стрелками, а прочие узлы – это состояния, достижимые путем следования по стрелкам.

Легко видеть, что, опробовав все возможные последовательности действий, мы рано или поздно найдем оптимальную. К сожалению, в большинстве за-

дач это неосуществимо, поскольку число действий растет экспоненциально. Алгоритмы планирования, применяемые в сложных задачах, подбирают стратегию, опираясь на ограниченное число траекторий.

Один из таких алгоритмов называется поиском по дереву методом Монте-Карло (ПДМК), он, кстати, применяется в программе AlphaGo. ПДМК итеративно строит решающее дерево, генерируя конечную последовательность имитированных игр, которой достаточно для исследования той части дерева, которая еще не посещалась. Когда имитированная траектория доходит до листового узла (т. е. игра заканчивается), результаты распространяются в обратном направлении через посещенные состояния, и при этом обновляется хранящаяся в узлах информация о выигрыше (проигрыше) или о вознаграждении. После этого выбирается действие с наибольшим отношением выигрыш/проигрыш или вознаграждением.

С другой стороны, алгоритмы планирования, работающие с непрерывными действиями, включают методы оптимизации траекторий. Они гораздо сложнее, чем алгоритмы с дискретными действиями, потому что должны решать задачу оптимизации в бесконечномерном пространстве.

Кроме того, для многих из них нужен градиент модели. Примером может служить метод управления с прогнозирующими моделями (Model Predictive Control – MPC), который производит оптимизацию на конечном временном горизонте, но выполняет не всю найденную траекторию, а только первое действие на ней. Это позволяет MPC получать отклик быстрее, чем в других моделях с бесконечным горизонтом планирования.

Неизвестная модель

Что делать, когда модель окружающей среды неизвестна? Обучить ее! Почти все, о чем мы говорили до сих пор, подразумевает обучение. Действительно ли это наилучшее решение? Если вы и вправду хотите воспользоваться подходом на основе модели, то да, и вскоре мы увидим, как это сделать. Однако не всегда это самый предпочтительный путь.

В обучении с подкреплением конечная цель – обучиться оптимальной стратегии решения данной задачи. Ранее в этой главе мы сказали, что подход на основе модели применяется в основном, чтобы уменьшить количество взаимодействий с окружающей средой, но всегда ли это так? Представьте, что ваша цель – приготовить омлет. Точное знание о том, в какой точке лучше всего разбивать яйцо, абсолютно бесполезно; достаточно примерно знать, как его разбить. Поэтому в данной ситуации безмодельный алгоритм, не требующий знания точной структуры яйца, предпочтительнее.

Однако не следует отсюда делать вывод, что алгоритмы на основе модели вообще не нужны. Например, они лучше безмодельных алгоритмов в ситуации, когда обучить модель намного проще, чем стратегию.

Единственный способ обучить модель (к сожалению) – взаимодействия со средой. От этого шага никуда не деться, потому что он позволяет собрать данные о среде. Обычно в процессе обучения участвует учитель, т. е. мы обучаем аппроксиматор функции (например, глубокую нейронную сеть) минимизировать функцию потерь, например среднеквадратическую ошибку между переходами, полученными непосредственно от среды, и предсказанными. Пример

показан на рис. 9.2, где глубокая нейронная сеть обучается моделировать среду, т. е. предсказывать следующее состояние s' и вознаграждение r , зная текущее состояние s и действие a .

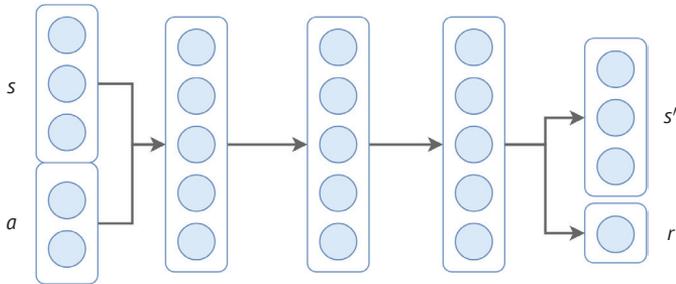


Рис. 9.2

Помимо нейронных сетей, существуют и другие варианты, например гауссовы процессы и гауссовы смесовые модели. Точнее, гауссов процесс учитывает недоверенность модели, считается, что этот подход очень эффективен с точки зрения объема потребных данных. До появления глубоких нейронных сетей это был самый популярный подход.

Но у гауссовых процессов есть серьезный недостаток – они работают медленно на больших наборах данных. Поэтому для обучения в более сложных средах (для которых требуются более крупные наборы данных) глубокие нейронные сети предпочтительнее. К тому же ГНС могут обучать модели окружающих сред, в которых наблюдениями являются изображения.

Существует два основных способа обучить модель окружающей среды. В первом модель обучается один раз и остается неизменной, во втором модель обучается в начале, но переобучается всякий раз, как план или стратегия изменяются. Оба варианта показаны на рис. 9.3.

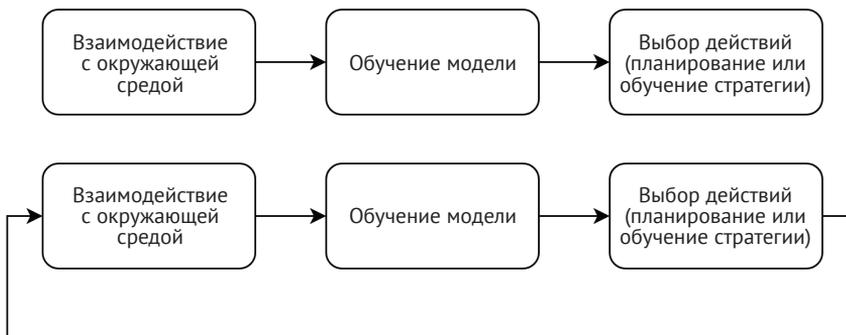


Рис. 9.3

В верхней части рис. 9.3 показан последовательный алгоритм на основе модели, когда агент взаимодействует с окружающей средой только до начала обучения модели. В нижней части мы видим циклический подход к обучению

модели, когда модель уточняется с помощью дополнительных данных от другой стратегии.

Чтобы понять, почему алгоритм может выиграть во втором варианте, необходимо определить одно важное понятие. Чтобы подготовить набор данных для обучения динамике среды, нужна стратегия навигации по ней. Но в самом начале стратегия может быть детерминированной или совершенно случайной. Поэтому при ограниченном количестве взаимодействий исследованная часть пространства будет крайне мала.

Это не дает модели обучиться тем частям среды, которые необходимы для планирования или выработки оптимальных траекторий. Но если модель повторно обучается на новых взаимодействиях, поступающих от новой улучшенной стратегии, то она итеративно адаптируется к стратегии и запоминает все части окружающей среды (с точки зрения стратегии), которые еще не посещались. Это называется агрегированием данных.

На практике модель в большинстве случаев неизвестна и обучается с помощью агрегирования данных, чтобы адаптироваться к вновь выработанной стратегии. Однако обучить модель бывает трудно, на этом пути могут возникнуть следующие проблемы:

- **переобучение модели:** обученная модель слишком точно аппроксимирует локальную область, не улавливая глобальной структуры;
- **неверная модель:** если планировать действия или обучать стратегию на основе неверной модели, то может воспоследовать целый каскад ошибок с потенциально катастрофическими последствиями.

Хорошие алгоритмы на основе модели, которые обучают модель, должны справляться с этими проблемами. Например, можно использовать алгоритмы, оценивающие недостоверность, скажем байесовские нейронные сети. Можно также воспользоваться ансамблем моделей.

Достоинства и недостатки

При разработке алгоритма обучения с подкреплением (любого вида) нужно принимать во внимание три свойства.

- **Асимптотическое качество.** Это наивысшее качество, которого может достичь алгоритм при наличии бесконечных ресурсов (времени и оборудования).
- **Физическое время.** Сколько времени алгоритм должен обучаться, чтобы достичь заданного качества при заданных вычислительных ресурсах.
- **Выборочная эффективность.** Количество взаимодействий с окружающей средой, необходимое для достижения заданного качества.

Мы уже изучали выборочную эффективность безмодельного и основанного на модели ОП и знаем, что последнее в этом смысле гораздо эффективнее. Но как насчет физического времени и качества? Обычно асимптотическое качество алгоритмов на основе модели ниже, а обучаются они медленнее, чем безмодельные алгоритмы. В общем случае повышение выборочной эффективности достигается ценой ухудшения качества и быстродействия.

Одна из причин низкого качества основанного на модели обучения – неточность модели (если она была обучена), из-за чего в стратегии вносятся допол-

нительные ошибки. Большое физическое время может быть связано с медленным алгоритмом планирования или с большим количеством взаимодействий, необходимым для обучения стратегии в неправильно смоделированной среде. Кроме того, у алгоритмов планирования на основе модели велико время логического вывода из-за высокой стоимости вычислений, которые тем не менее нужно производить на каждом шаге.

Итак, следует учитывать дополнительное время, требующееся для обучения алгоритма на основе модели, и смириться с низким асимптотическим качеством подобных подходов. Однако обучение на основе модели очень полезно, когда обучить модель проще, чем саму стратегию, и когда взаимодействие со средой медленное или дорогостоящее.

Оба подхода – безмодельное и основанное на модели обучение – имеют как явные преимущества, так и безусловные недостатки. Нельзя ли взять лучшее из обоих миров?

СОЧЕТАНИЕ БЕЗМОДЕЛЬНОГО И ОСНОВАННОГО НА МОДЕЛИ ОБУЧЕНИЯ

Мы только что видели, что планирование может обойтись дорого с вычислительной точки зрения как на этапе обучения, так и на этапе выполнения и что в сложной окружающей среде алгоритмы планирования не в состоянии достичь хороших результатов. Но мы также намекнули на существование другого подхода – обучить стратегии. В этом случае логический вывод гораздо быстрее, потому что стратегии не нужно планировать на каждом шаге.

Простой, но эффективный способ обучения стратегии – объединить безмодельное и основанное на модели обучение. Учитывая недавние достижения в области безмодельных алгоритмов, эта комбинация завоевала популярность и на сегодня является самым распространенным подходом. В следующем разделе мы разработаем один из таких алгоритмов – ME-TRPO.

Полезная комбинация

Как мы знаем, у безмодельного обучения хорошее асимптотическое качество, но низкая выборочная эффективность. С другой стороны, обучение на основе модели эффективно с точки зрения количества необходимых данных, но с трудом справляется со сложными задачами. Сочетание обоих подходов позволяет повысить выборочную эффективность, сохранив высокое качество безмодельных алгоритмов.

Объединить оба подхода можно многими способами, и соответствующие алгоритмы сильно различаются. Например, если модель задана (как в игре го и в шахматах), то поиск по дереву и алгоритмы на основе ценности могут совместно найти лучшую оценку ценности действия.

Другой пример – объединить обучение среды и стратегии непосредственно в архитектуре глубокой нейронной сети, так чтобы обученная динамика

вносила вклад в планирование стратегии. Еще один часто применяемый подход – воспользоваться обученной моделью окружающей среды для генерации дополнительных примеров с целью оптимизации стратегии.

Иначе говоря, стратегия обучается на имитированных играх с обученной моделью. Сделать это можно по-разному, но основная идея отражена в приведенном ниже псевдокоде.

while not done:

- > получать информацию о переходах $\{(s, a, s', r)_i\}$ от реальной окружающей среды
- > с помощью стратегии π
- > сохранять переходы в буфере D
- > обучить с учителем модель $f(s, a)$, которая минимизирует $\Sigma(f(s, a) - s')^2$,
- > на данных из D
- > (факультативно обучить $g(s, a)$)

повторить K раз:

- > выбрать начальное состояние
- > имитировать переходы $\{(s_s, a, s'_s, r_s)_i\}$ в модели $s'_s = f(s_s, a)$,
- > применяя стратегию π
- > обновить стратегию π , применяя какой-то безмодельный алгоритм ОП

В этом общем сценарии два цикла. Во внешнем цикле собираются данные во взаимодействии с реальной окружающей средой для обучения модели, а во внутреннем модель генерирует имитированные примеры, которые используются для обучения стратегии с помощью безмодельного алгоритма. Обычно модель динамики обучается с учителем, так чтобы минимизировать средне-квадратическую ошибку. Чем точнее предсказания модели, тем правильнее будет стратегия.

Во внутреннем цикле можно имитировать траектории – полные или фиксированной длины. На практике часто применяют второй вариант, чтобы сгладить несовершенство модели. Кроме того, траектории могут начинаться из случайного состояния, выбранного из буфера, который содержит реальные переходы, или из начального состояния. Первый вариант предпочтительнее, когда модель неточна, поскольку не дает траектории слишком далеко отклониться от реальной. Чтобы понять, почему это работает, взгляните на рис. 9.4. Траектории, полученные в реальной окружающей среде, изображены черным цветом, а имитированные – синим.

Легко видеть, что траектории, начинающиеся из начального состояния, длиннее и потому расходятся быстрее, поскольку ошибки неточной модели влияют на все последующие предсказания.



Отметим, что можно было бы выполнить только одну итерацию главного цикла и собрать все данные, необходимые для обучения приемлемой приближенной модели окружающей среды. Однако по вышеуказанным причинам лучше использовать итеративные методы агрегирования данных и циклически переобучать модель на переходах, генерируемых более новой стратегией.

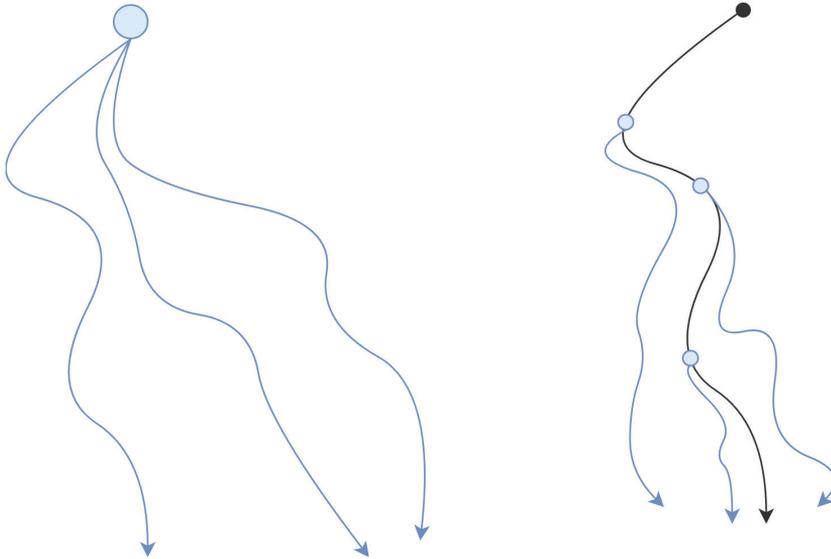


Рис. 9.4

Построение модели из изображений

Рассмотренные выше способы сочетания безмодельного и основанного на модели обучения были разработаны специально для пространств состояний низкой размерности. А как быть с пространствами наблюдений высокой размерности, например изображениями?

Один из способов – производить обучение в скрытом пространстве. Это представление $g(s)$ многомерных входных данных s в пространстве меньшей размерности, его также называют погружением (embedding). Получить его позволяют нейронные сети, например автокодировщики (см. рис. 9.5).

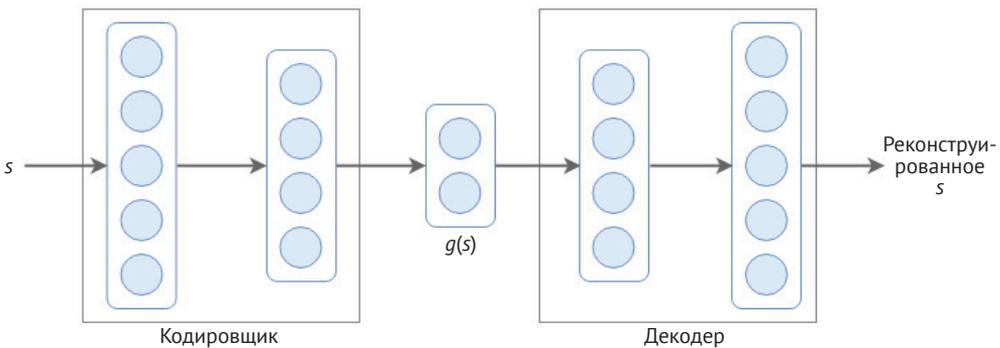


Рис. 9.5

Здесь мы видим кодировщик $g(s)$, который отображает изображение в скрытое пространство низкой размерности, и декодер, который отображает

скрытое пространство в реконструированное изображение. Результат работы автокодировщика должен содержать главные признаки изображения, отобранные так, что два изначально похожих изображения похожи и в скрытом пространстве.

В ОП можно обучить автокодировщик реконструировать вход S или предсказывать следующий кадр S' (при желании вместе с вознаграждением). Затем мы можем использовать скрытое пространство, чтобы обучить динамическую модель и стратегию. Главное преимущество этого подхода – существенный выигрыш в скорости, поскольку представление изображения уменьшилось. Но стратегия, обученная в скрытом пространстве, может оказаться малоприменимой, если автокодировщик не способен реконструировать истинное представление.

Обучение на основе модели в пространствах высокой размерности все еще является областью активных исследований.

i Читателям, интересующимся основанными на модели алгоритмами, которые обучаются на примерах изображений, рекомендуем прочитать статью Kaiser «Model-Based Reinforcement Learning for Atari» (<https://arxiv.org/pdf/1903.00374.pdf>).

Пока что мы рассматривали сочетание основанного на модели и безмодельного обучения скорее в теоретическом и описательном плане. Но чтобы по-настоящему усвоить эти идеи, необходимо перейти в практическую плоскость. Поэтому рассмотрим реализацию нашего первого алгоритма на основе модели.

ПРИМЕНЕНИЕ АЛГОРИТМА ME-TRPO К ЗАДАЧЕ ОБ ОБРАТНОМ МАЯТНИКЕ

Для воплощения в жизнь подхода, описанного в псевдокоде выше, существует много вариантов сочетания основанных на модели и безмодельных алгоритмов. Почти во всех предлагаются разные способы борьбы с несовершенством модели окружающей среды.

Это основная проблема, которую предстоит решить, чтобы достичь такого же качества, как у безмодельных методов. Модели, обученные на взаимодействии со сложной средой, всегда будут в большей или меньшей степени неточны. Так что задача состоит в том, чтобы оценить и контролировать недостоверность модели и тем самым стабилизировать и ускорить процесс обучения.

В алгоритме ME-TRPO предлагается использовать ансамбль моделей для контроля над недостоверностью модели и регуляризации процесса обучения. В роли моделей выступают глубокие нейронные сети с разными начальными весами и обучающими данными. В совокупности они дают более робастную модель окружающей среды, которая менее склонна исследовать области, в которых недостаточно данных.

Затем стратегия обучается на траекториях, имитированных ансамблем. Для ее обучения был выбран алгоритм **оптимизации стратегии в доверительной области (TRPO)**, рассмотренный в главе 7.

Принцип работы ME-TRPO

На первом этапе ME-TRPO обучает ансамбль моделей динамике окружающей среды. Взаимодействие со средой начинается со случайной стратегии π с целью получить набор данных о переходах $(s, a, s', r)_i$. Затем этот набор данных используется для обучения с учителем всех моделей динамики f_{θ_i} . Начальные веса всех моделей выбираются случайным образом, после чего производится обучение на разных мини-пакетах. Чтобы избежать переобучения, по набору данных создается контрольный набор. Кроме того, применяется *ранняя остановка* (техника регуляризации, широко применяемая в машинном обучении), чтобы прервать процесс обучения, когда потеря на контрольном наборе перестает улучшаться.

На втором этапе стратегия обучается с помощью алгоритма TRPO. Точнее, стратегия обучается на данных, получаемых от обученных моделей, которые называют еще *имитированной* (в отличие от реальной) *окружающей средой*. Чтобы стратегия π не использовала те области, в которых какая-то одна обученная модель неточна, обучение производится на данных о переходах, предсказанных целым ансамблем моделей f_{θ_i} . Иначе говоря, стратегия обучается на имитированном наборе данных, содержащем переходы, полученные от моделей f_{θ_i} , случайно выбранных из ансамбля. В процессе обучения за стратегией ведется постоянное наблюдение, и как только качество перестает улучшаться, процесс останавливается.

Цикл, состоящий из двух частей, повторяется до достижения сходимости. Но на каждой итерации с применением вновь обученной стратегии π собираются данные из реальной окружающей среды, и эти данные агрегируются с предыдущим набором данных. Псевдокод алгоритма ME-TRPO приведен ниже:

Случайным образом инициализировать стратегию π и модели $f_{\theta_1}, \dots, f_{\theta_N}$
Инициализировать пустой буфер D

while not done:

- > поместить в буфер D данные о переходах $(s, a, s', r)_i$ в реальной окружающей среде при следовании стратегии π (или случайной)

- > обучить с учителем на данных из D модели $f_{\theta_1}(s, a), \dots, f_{\theta_N}(s, a)$, которые минимизируют $\sum (f_{\theta_i}(s, a) - s')^2$

until достигнута сходимость:

- > выбрать начальное состояние s_0
- > имитировать переходы $(s_s, a, s'_s, r_s)_i$, применяя модели $\{f_{\theta_i}\}_{i=1}^K$
- > и стратегию π
- > применить обновление TRPO для оптимизации стратегии

Важно отметить, что, в отличие от большинства алгоритмов на основе модели, здесь вознаграждение не включено в модель окружающей среды. Поэтому в ME-TRPO предполагается, что функция вознаграждения известна.

Реализация ME-TRPO

Код алгоритма ME-TRPO довольно длинный, и мы не станем приводить его целиком. К тому же многие его части неинтересны, а весь код, относящийся

к TRPO, уже обсуждался в главе 7. Однако для тех, кто хочет поэкспериментировать с алгоритмом, полная версия имеется в репозитории этой книги на GitHub.

Ниже приведены объяснения и реализация следующих частей:

- внутренний цикл, в котором имитируются игры и оптимизируется стратегия;
- функция, которая обучает модели.

Остальной код очень похож на TRPO.

1. **Изменение стратегии.** Единственное изменение в процедуре взаимодействия с реальной окружающей средой – стратегия. Точнее, в первом эпизоде стратегия случайная, но в остальных она выбирает действия из нормального распределения со случайным стандартным отклонением, фиксированного в начале работы алгоритма. Для этого строчка `act, val = sess.run([a_sampl, s_values], feed_dict={obs_ph:[env.n_obs]})` в реализации TRPO заменяется следующими:

```
...
if ep == 0:
    act = env.action_space.sample()
else:
    act = sess.run(a_sampl, feed_dict={obs_ph:[env.n_obs],
log_std:init_log_std})
...
```

2. **Обучение глубоких нейронных сетей f_{θ_i} .** Нейронные сети обучаются модели окружающей среды на наборе данных, собранном на предыдущем шаге. Этот набор данных разбивается на обучающий и контрольный, и при проверке на контрольном наборе применяется техника ранней остановки, чтобы решить, имеет ли смысл продолжать обучение.

```
...
model_buffer.generate_random_dataset()
train_obs, train_act, _, train_nxt_obs, _ =
model_buffer.get_training_batch()
valid_obs, valid_act, _, valid_nxt_obs, _ =
model_buffer.get_valid_batch()
print('Log Std policy:', sess.run(log_std))
for i in range(num_ensemble_models):
    train_model(train_obs, train_act, train_nxt_obs, valid_obs,
valid_act, valid_nxt_obs, step_count, i)
```

`model_buffer` – экземпляр класса `FullBuffer`, который содержит примеры, сгенерированные средой, а метод `generate_random_dataset` разбивает набор данных на обучающий и контрольный, которые затем возвращаются методами `get_training_batch` и `get_valid_batch`.

Далее каждая модель обучается функцией `train_model`, которой передаются оба набора данных, количество шагов и индекс подлежащей обучению модели. Переменная `num_ensemble_models` содержит количество моделей в ансамбле. В статье, посвященной алгоритму ME-TRPO, показано, что достаточно от 5 до 10 моделей. Аргумент `i` говорит о том, какую модель ансамбля оптимизировать.

3. Генерирование фиктивных траекторий в имитированной среде и обучение стратегии:

```
best_sim_test = np.zeros(num_ensemble_models)
for it in range(80):
    obs_batch, act_batch, adv_batch, rtg_batch =
simulate_environment(sim_env, action_op_noise, simulated_steps)

    policy_update(obs_batch, act_batch, adv_batch, rtg_batch)
```

Это повторяется, пока стратегия продолжает улучшаться, но не более 80 раз. Функция `simulate_environment` строит набор данных (содержащий наблюдения, действия, преимущества, ценности и величины дохода), прогоняя стратегию в имитированной среде (представленной обученными моделями). В нашем случае стратегия представлена функцией `action_op_noise`, которая получает состояние и возвращает действие в соответствии с обученной стратегией. `sim_env` – это модель окружающей среды f_{θ} , случайно выбираемая из ансамбля на каждом шаге. Последний аргумент, переданный функции `simulate_environment`, `simulated_steps`, говорит, сколько предпринять шагов в имитированной среде.

И в конце функция `policy_update` выполняет шаг алгоритма TRPO, чтобы обновить стратегию с учетом данных, собранных в имитированной среде.

4. Реализация механизма ранней остановки и оценивание стратегии.

Механизм ранней остановки не дает стратегии переобучиться на моделях окружающей среды. Для этого он следит за качеством стратегии на каждой модели. Если процент моделей, на которых стратегия улучшилась, превышает некоторый порог, то цикл прерывается. Предполагается, что это признак того, что стратегия приближается к переобучению. Отметим, что, в отличие от обучения, когда каждая траектория генерируется всеми обученными моделями среды, в процессе тестирования стратегия проверяется только на одной модели.

```
if (it+1) % 5 == 0:
    sim_rewards = []

    for i in range(num_ensemble_models):
        sim_m_env = NetworkEnv(gym.make(env_name), model_op,
pendulum_reward, pendulum_done, i+1)
        mn_sim_rew, _ = test_agent(sim_m_env, action_op, num_games=5)
        sim_rewards.append(mn_sim_rew)

    sim_rewards = np.array(sim_rewards)
    if (np.sum(best_sim_test >= sim_rewards) >
int(num_ensemble_models*0.7)) \
        or (len(sim_rewards[sim_rewards >= 990]) >
int(num_ensemble_models*0.7)):
        break
    else:
        best_sim_test = sim_rewards
```

Оценивание стратегии производится через каждые пять итераций обучения. Для каждой модели ансамбля создается новый экземпляр

класса `NetworkEnv`. Он обладает такой же функциональностью, как реальная окружающая среда, но возвращает переходы из обученной модели среды. Для этого `NetworkEnv` наследует классу `GymWrapper` и переопределяет функции `reset` и `step`. Первый параметр конструктора – реальная окружающая среда, которая нужна только для того, чтобы получить реальное начальное состояние, а затем функция `model_op` порождает следующее состояние, получив на входе состояние и действие. Наконец, функции `pendulum_reward` и `pendulum_done` возвращают вознаграждение и флаг `done` соответственно. Обе они написаны с учетом особенностей конкретной среды.

5. **Обучение модели динамики.** Функция `train_model` оптимизирует модель для предсказания будущего состояния. В ней нет ничего сложного. Мы использовали ее на шаге 2, когда обучали ансамбль моделей. Аргументы этой функции были описаны выше. На каждой итерации внешнего цикла ME-TRPO мы переобучаем все модели, т. е. обучаем их, инициализируя веса случайным образом, а не используя веса, полученные в результате предыдущей оптимизации. Поэтому при каждом вызове `train_model` мы до начала обучения восстанавливаем случайные начальные веса модели. В следующем фрагменте кода восстанавливаются веса и вычисляется потеря до и после операции:

```
def train_model(tr_obs, tr_act, tr_nxt_obs, v_obs, v_act, v_nxt_obs,
               step_count, model_idx):
    mb_valid_loss1 = run_model_loss(model_idx, v_obs, v_act, v_nxt_obs)
    model_assign(model_idx, initial_variables_models[model_idx])
    mb_valid_loss = run_model_loss(model_idx, v_obs, v_act, v_nxt_obs)
```

Функция `run_model_loss` возвращает потерю текущей модели, а `model_assign` восстанавливает параметры из массива `initial_variables_models[model_idx]`.

Затем мы обучаем модель при условии, что потеря на контрольном наборе улучшилась в последних `model_iter` итерациях. Но поскольку последняя модель необязательно лучшая, мы запоминаем лучшую из найденных и восстанавливаем ее параметры в конце обучения. Мы также случайным образом перемешиваем набор данных и разбиваем его на мини-пакеты. Код приведен ниже.

```
acc_m_losses = []
last_m_losses = []
md_params = sess.run(models_variables[model_idx])
best_mb = {'iter':0, 'loss':mb_valid_loss, 'params':md_params}
it = 0

lb = len(tr_obs)
shuffled_batch = np.arange(lb)
np.random.shuffle(shuffled_batch)

while best_mb['iter'] > it - model_iter:
    # обновить модель на каждом мини-пакете
    last_m_losses = []
```

```

for idx in range(0, lb, model_batch_size):
    minib = shuffled_batch[idx:min(idx+minibatch_size,lb)]
    if len(minib) != minibatch_size:
        _, ml = run_model_opt_loss(model_idx, tr_obs[minib],
tr_act[minib], tr_nxt_obs[minib])
        acc_m_losses.append(ml)
        last_m_losses.append(ml)

    # Проверить, улучшилась ли потеря на контрольном наборе
    mb_valid_loss = run_model_loss(model_idx, v_obs, v_act,
v_nxt_obs)
    if mb_valid_loss < best_mb['loss']:
        best_mb['loss'] = mb_valid_loss
        best_mb['iter'] = it
        best_mb['params'] = sess.run(models_variables[model_idx])

    it += 1

# Восстановить модель с наименьшей потерей на контрольном наборе
model_assign(model_idx, best_mb['params'])

print('Model:{}, iter:{} -- Old Val loss:{:.6f} New Val loss:{:.6f}
-- New Train loss:{:.6f}'.format(model_idx, it, mb_valid_loss1,
best_mb['loss'], np.mean(last_m_losses)))

```

Функция `run_model_opt_loss` запускает оптимизатор модели с индексом `model_idx`.

На этом реализация алгоритма ME-TRPO завершена. В следующем разделе мы посмотрим на его результаты.

Эксперименты в среде RoboSchool

Протестируем алгоритм ME-TRPO в среде `RoboSchoolInvertedPendulum-v1` для непрерывного управления обратным маятником, похожей на хорошо известный дискретный аналог `CartPole`. На рис. 9.6 показано, как она выглядит.

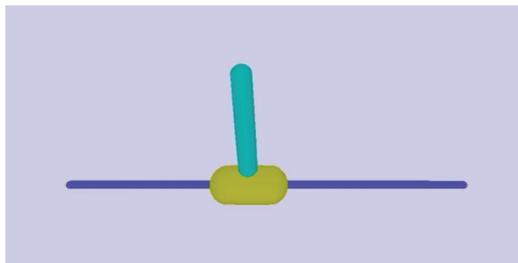


Рис. 9.6

Цель заключается в том, чтобы поддерживать стержень в вертикальном положении, когда тележка движется. За каждый шаг, на котором стержень не упал, начисляется вознаграждение +1.

Для работы ME-TRPO нужна функция вознаграждения и функция done, поэтому определим ту и другую для данной задачи. Функция `pendulum_reward` будет возвращать 1 вне зависимости от наблюдения и действия:

```
def pendulum_reward(ob, ac):
    return 1
```

Функция `pendulum_done` возвращает True, если абсолютная величина угла наклона стержня больше фиксированного порога. Угол можно извлечь непосредственно из состояния, поскольку третий и четвертый элементы состояния – это соответственно косинус и синус угла наклона. Таким образом, `pendulum_done` определена следующим образом:

```
def pendulum_done(ob):
    return np.abs(np.arcsin(np.squeeze(ob[3]))) > .2
```

Обычные гиперпараметры алгоритма TRPO почти не меняются по сравнению с приведенными в главе 7. Для ME-TRPO нужны следующие гиперпараметры:

- скорость обучения оптимизатора моделей динамики, `mb_lr`;
- размер мини-пакета `model_batch_size`, используемого для обучения моделей динамики;
- количество шагов взаимодействия с имитированной средой на каждой итерации, `simulated_steps` (одновременно это размер пакета для обучения стратегии);
- количество моделей в ансамбле, `num_ensemble_models`;
- сколько итераций подождать, прежде чем прерывать обучение модели, если на контрольном наборе качество не ухудшилось (`model_iter`).

Ниже приведены значения гиперпараметров для этой среды.

Гиперпараметр	Значения
Скорость обучения (<code>mb_lr</code>)	1e-5
Размер пакета (<code>model_batch_size</code>)	50
Количество имитированных шагов (<code>simulated_steps</code>)	50 000
Количество моделей (<code>num_ensemble_models</code>)	10
Количество итераций перед ранней остановкой (<code>model_iter</code>)	15

Результаты в среде *RoboSchoolInvertedPendulum*

На рис. 9.7 показан график качества.

Это график зависимости вознаграждения от количества взаимодействий с реальной окружающей средой. После 900 шагов и 15 игр агент достиг наивысшего качества 1000. Стратегия обновлялась 15 раз и обучилась на 750 000 имитированных шагов. Обучение алгоритма заняло около 2 часов на компьютере средней мощности.

Мы отметили очень высокую вариативность результатов – при обучении с разными начальными весами кривые качества сильно различались. Это верно и для безмодельных алгоритмов, но здесь различия выражены более резко. Возможно, что это как-то связано с данными, собираемыми в ходе взаимодействия с реальной средой.

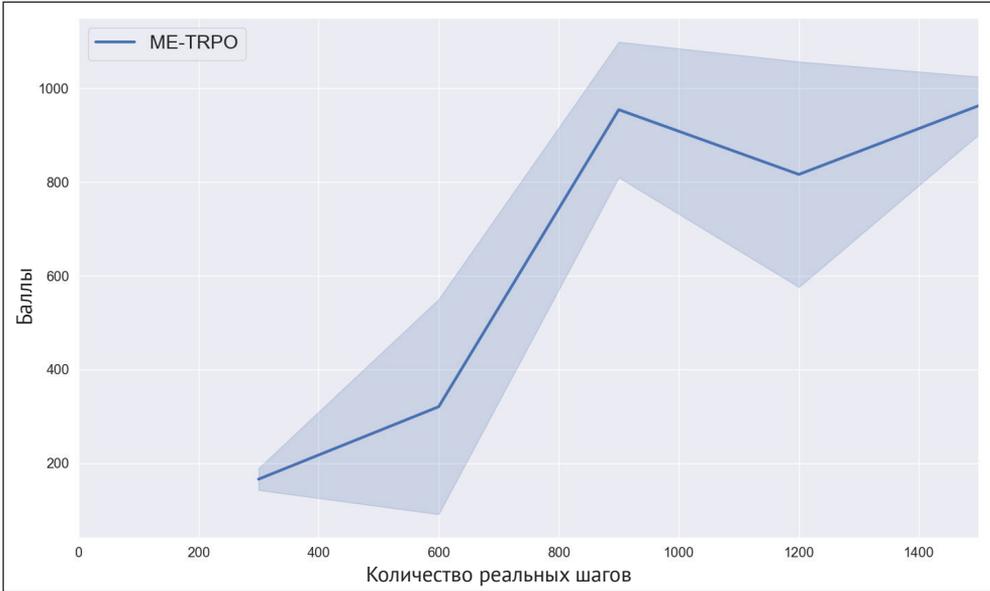


Рис. 9.7

РЕЗЮМЕ

В этой главе мы расстались с безмодельными алгоритмами и начали обсуждать и исследовать алгоритмы, обучающиеся на модели окружающей среды. Мы рассмотрели основные причины смены парадигмы, побудившие разрабатывать алгоритмы такого типа. Затем мы выделили два случая работы с моделью: когда модель уже известна и когда ее предстоит сначала обучить.

Далее мы узнали, что модель можно использовать либо для планирования следующих действий, либо для обучения стратегии. Не существует четкого правила, диктующего, что выбрать, но, вообще говоря, это зависит от сложности пространств наблюдений и действий и от скорости логического вывода. Мы изучили достоинства и недостатки безмодельных алгоритмов и углубили понимание того, как обучить стратегию, сочетая безмодельные алгоритмы с обучением на основе модели. Это открыло новые возможности для применения моделей в пространствах наблюдений очень высокой размерности, например изображениях.

Наконец, чтобы лучше прочувствовать все сказанное о методах на основе модели, мы разработали алгоритм ME-TRPO. Он борется с недостоверностью модели, используя ансамбль моделей, а для обучения стратегии применяет оптимизацию стратегии в доверительной области. Все модели предсказывают следующие состояния, а затем участвуют в создании имитированных траекторий, на которых обучается стратегия. Таким образом, стратегия обучается только на обученной модели окружающей среды.

В этой главе мы завершили рассказ об обучении на основе модели, а в следующей перейдем к новому виду обучения. Мы поговорим об алгоритмах,

обучающихся с помощью подражания, а также разработаем и обучим агента, который, подражая поведению эксперта, научится играть в игру FlappyBird.

Вопросы

1. Если бы в вашем распоряжении было всего 10 партий, на которых нужно обучить агента играть в шашки, то что бы вы выбрали: безмодельный или основанный на модели алгоритм?
2. Назовите недостатки алгоритмов на основе модели.
3. Если модель окружающей среды неизвестна, как ее можно обучить?
4. Для чего используются методы агрегирования данных?
5. Как алгоритм ME-TRPO стабилизирует обучение?
6. Как использование ансамбля моделей улучшает обучение стратегии?

Для дальнейшего чтения

- Чтобы расширить знания об основанных на модели алгоритмах, которые обучаются стратегии на изображениях, прочитайте статью «Model-Based Reinforcement Learning for Atari» (<https://arxiv.org/pdf/1903.00374.pdf>).
- Оригинальная статья об алгоритме ME-TRPO находится по адресу <https://arxiv.org/pdf/1802.10592.pdf>.

Глава 10

Подражательное обучение и алгоритм DAgger

Способность алгоритма обучаться только на вознаграждениях – очень важная характеристика, позволившая нам разработать алгоритмы обучения с подкреплением. Это дает агенту возможность обучаться и улучшать свою стратегию с нуля, без какого-либо дополнительного инструктирования. Однако существуют ситуации, когда в окружающей среде уже имеются другие агенты-эксперты. Алгоритмы **подражательного обучения** (imitation learning – **IL**) используют опыт экспертов, подражая их действиям и перенимая их стратегию.

Эта глава посвящена подражательному обучению. Оно отличается от обучения с подкреплением, но открывает весьма интересные возможности, особенно в средах, где пространство состояний очень велико, а вознаграждение разрежено. Очевидно, что подражательное обучение возможно только тогда, когда существует более опытный агент, поведению которого можно подражать.

Мы поговорим об основных идеях и особенностях методов подражательного обучения. Будет реализован алгоритм DAgger, с помощью которого мы обучим агента играть в игру Flappy Bird. Это поможет овладеть новым семейством алгоритмов и в должной мере оценить их принципы.

В последнем разделе данной главы мы познакомимся с **обратным обучением с подкреплением** (inverse reinforcement learning – **IRL**). Это метод обучения поведению другого агента с точки зрения ценностей и вознаграждений, т. е. IRL обучается функции вознаграждения.

В данной главе будут рассмотрены следующие вопросы:

- подход на основе подражания;
- игра в Flappy Bird;
- алгоритм агрегирования набора данных;
- обратное обучение с подкреплением.

ТЕХНИЧЕСКИЕ ТРЕБОВАНИЯ

После краткого теоретического введения в алгоритмы подражательного обучения мы реализуем настоящий алгоритм IL. Но приведем только главную функцию и наиболее интересные части. Полный код можно найти в репозитории

этой книги на GitHub по адресу <https://github.com/PacktPublishing/Reinforcement-Learning-Algorithms-with-Python>.

Установка Flappy Bird

Ниже мы протестируем наш алгоритм ИЛ на модифицированной версии знаменитой игры Flappy Bird (https://en.wikipedia.org/wiki/Flappy_Bird). А в этом разделе опишем, как ее установить.

Но сначала нужно будет установить некоторые дополнительные библиотеки.

- В Ubuntu выполните следующие команды:

```
$ sudo apt-get install git python3-dev python3-numpy libsdlimage1.2-
dev libsdl-mixer1.2-dev libsdl-ttf2.0-dev libsmpeg-dev
libsdl1.2-dev libportmidi-dev libswscale-dev libavformat-dev
libavcodec-dev libfreetype6-dev
$ sudo pip install pygame
```

- Пользователи Mac могут установить библиотеки, выполнив эти команды:

```
$ brew install sdl sdl_ttf sdl_image sdl_mixer portmidi
$ pip install -c https://conda.binstar.org/quasiben pygame
```

- Далее для пользователей Ubuntu и Mac процедура одинаковая.

1. Сначала нужно клонировать PLE. Для этого выполните команду

```
git clone https://github.com/ntasfi/PyGame-Learning-Environment
```

PLE – это набор окружающих сред, включающий и Flappy Bird.

2. Затем войдите в папку PyGame-Learning-Environment:

```
cd PyGame-Learning-Environment
```

3. И наконец, выполните установку командой

```
sudo pip install -e .
```

Теперь все готово для использования Flappy Bird.

ПОДХОД НА ОСНОВЕ ПОДРАЖАНИЯ

ИЛ – это искусство приобретения новых навыков путем подражания эксперту. Свойство обучения путем подражания не так уж необходимо для обучения стратегиям последовательного принятия решения, но в настоящее время оно приобретает важность в самых разных задачах. Некоторые задачи невозможно решить с помощью простого обучения с подкреплением, и бутстрэппинг стратегии в огромных пространствах состояний, характерных для сложных окружающих сред, становится ключевым фактором. На рис. 10.1 показаны основные компоненты процесса подражательного обучения.

Если в окружающей среде уже имеются опытные агенты (эксперты), то они могут предоставить новому агенту (обучаемому) огромный объем информации о видах поведения, необходимых для решения задачи и навигации в окружающей среде. В такой ситуации новый агент может обучиться гораз-

до быстрее, поскольку не обязан начинать с нуля. Агент-эксперт может также играть роль учителя, который наставляет нового агента и дает отзывы на его действия. Обратите внимание на различие ролей. Эксперт может быть как наставником, которому следует подражать, так и контролером, исправляющим ошибки ученика.

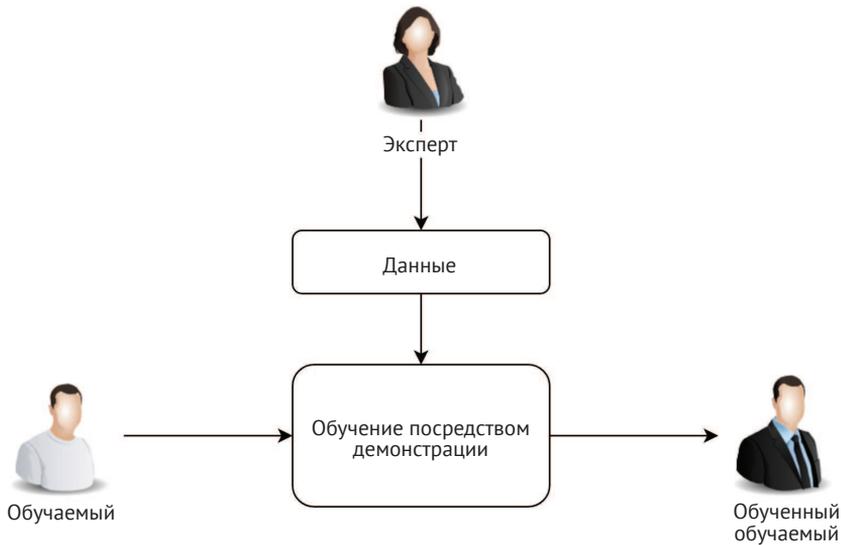


Рис. 10.1

Если имеется модель наставника или контролера, то алгоритм подражательного обучения может ей воспользоваться. Теперь вы понимаете, почему подражательное обучение играет такую важную роль и почему мы никак не могли пройти мимо него в этой книге.

Пример: помощник водителя

Чтобы лучше усвоить эти идеи, рассмотрим пример подростка, который учится водить машину. Предположим, что раньше он никогда не видел автомобиля и ничего не знает о том, как она работает. Существует три подхода к обучению.

1. Ученику выдают ключи, и он должен освоить все сам, вообще без помощи инструктора.
2. Прежде чем получить ключи, ученик проводит на месте пассажира 100 часов, наблюдая, как эксперт водит машину при разных погодных условиях и по разным дорогам.
3. Ученик наблюдает за работой эксперта, но, что гораздо важнее, он участвует в практических занятиях, на которых эксперт делает замечания о его вождении. Например, эксперт может в реальном времени подсказывать, как парковать машину и как оставаться на полосе движения.

Как вы, наверное, догадались, первый случай – это обучение с подкреплением, при котором агент лишь изредка получает вознаграждение за то, что не разбил машину, не вызвал гнев пешеходов и т. д.

Второй случай – это пассивное подражательное обучение, когда опыт приобретается путем простого повторения действий эксперта. В целом это очень близко к обучению с учителем.

Третий случай – активное подражательное обучение. Здесь эксперт на этапе обучения инструктирует обучаемого по каждому действию.

Сравнение подражательного обучения и обучения с подкреплением

Рассмотрим подробнее, чем различаются оба подхода. В подражательном обучении обучаемый не получает никакого вознаграждения. Отсюда вытекают весьма важные следствия.

Вернемся к нашему примеру. Ученик может лишь повторять действия эксперта настолько точно, насколько возможно, – пассивно или активно. Не получая объективного вознаграждения от окружающей среды, он ограничен субъективным мнением эксперта. Значит, при всем желании он не сможет ни понять, ни улучшить резоны учителя.

Поэтому ИЛ следует рассматривать как копирование действий эксперта, не зная его главной цели. В нашем случае юный водитель будет очень хорошо повторять выбранные учителем траектории, но не будет понимать, почему учитель выбрал именно такую траекторию. Не имея понятия о вознаграждении, агент, обученный подражательным алгоритмом, не может максимизировать полное вознаграждение, как в случае ОП.

Отсюда вытекают основные различия между ИЛ и ОП. В первом случае мы не понимаем конечной цели и потому не можем превзойти учителя. Во втором нам не хватает непосредственного сигнала от учителя и в большинстве случаев имеется только разреженное вознаграждение. Эта ситуация наглядно показана на рис. 10.2.

В левой части рис. 10.2 представлен обычный цикл ОП, а в правой – цикл подражательного обучения. Здесь обучаемый не получает никакого вознаграждения, эксперт сообщает только состояние и действие.

Роль эксперта в подражательном обучении

Термины *эксперт*, *учитель* и *инструктор* в контексте подражательного обучения означают одно и то же – персонажа, на примере которого новый агент (обучаемый) может обучаться.

Эксперт может принимать любую форму – от специалиста-человека до экспертной системы. Первый случай более понятный и распространенный. Человек обучает алгоритм решать задачу, которую сам уже решать умеет.Преимущества очевидны, и сама идея применима ко многим задачам.

Второй случай менее распространен. Одна из причин для выбора нового алгоритма, обученного методами ИЛ, связана с тем, что экспертные системы работают медленно, и, в силу технических ограничений, ускорить их нельзя. Например, в роли учителя может выступать точный, но медленный алгоритм поиска по дереву, который не способен производить логический вывод достаточно быстро. Тогда его можно заменить глубокой нейронной сетью. Обучение нейронной сети с использованием алгоритма поиска по дереву в качестве учителя может занять некоторое время, но обученная сеть будет работать намного быстрее.

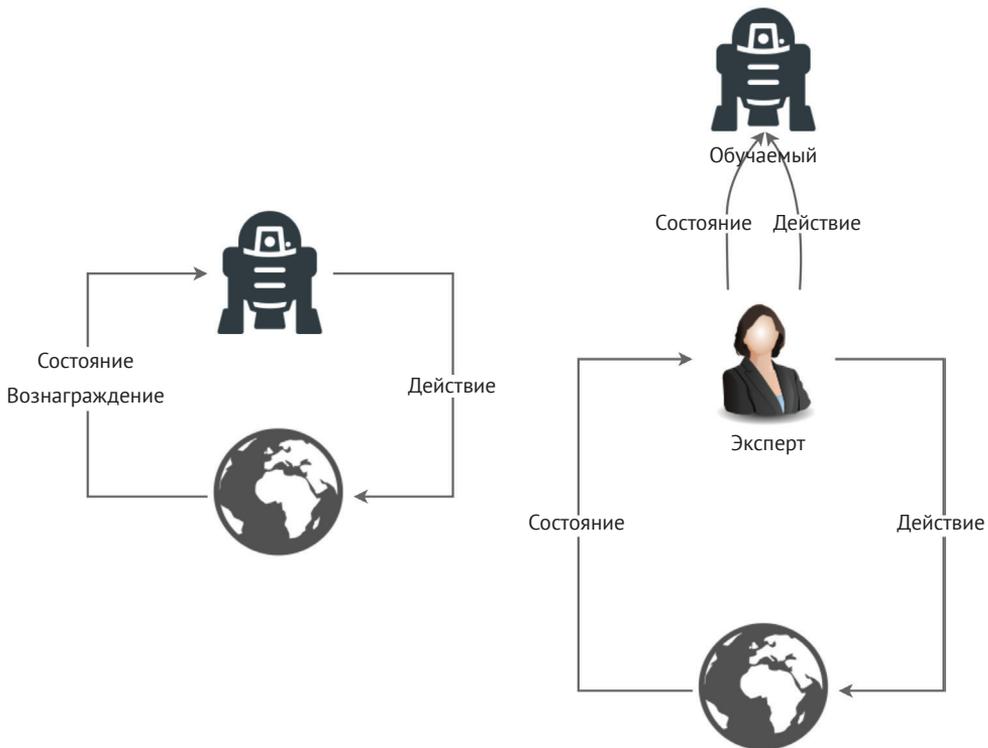


Рис. 10.2

Вы, конечно, уже понимаете, что качество стратегии, построенной обучаемым, сильно зависит от качества информации, предоставляемой экспертом. Качество знаний учителя полагает верхний предел достижениям ученика. Негодный учитель передает плохие данные обучаемому. Поэтому эксперт – ключевой компонент, который задает планку качества конечного агента. Если учитель слабый, то нельзя рассчитывать на получение хороших стратегий.

Структура ИЛ

Описав все ингредиенты подражательного обучения, мы можем перейти к алгоритмам и подходам к их проектированию.

Самый прямолинейный подход к задаче подражательного обучения показан на рис. 10.3.



Рис. 10.3

На этом рисунке можно выделить два основных шага:

- эксперт собирает данные из окружающей среды;
- стратегия обучается с помощью обучения с учителем на собранном наборе данных.

К сожалению, хотя обучение с учителем кажется высшим воплощением подражательного обучения, в большинстве случаев описанный подход не работает.

Чтобы понять, почему это так, напомним основные принципы обучения с учителем. Нас интересуют главным образом два из них: обучающий и тестовый наборы должны выбираться из одного и того же распределения, и данные должны быть независимы и одинаково распределены. Однако стратегия должна быть толерантна к различным траекториям и робастна к возможным сдвигам распределения.

Если агент обучается вождению машины только с помощью обучения с учителем, то любое отклонение от траекторий эксперта будет новым, ранее не встречавшимся состоянием, выбивающимся из обученного распределения. В этом новом состоянии агент не уверен, каким должно быть следующее действие. В обычном обучении с подкреплением это небольшая проблема. Если предсказание оказалось неверным, то на следующее предсказание это никак не повлияет. Но в задаче подражательного обучения алгоритм обучается стратегии, и свойство независимости и одинакового распределения уже не выполняется, поскольку последующие действия сильно коррелированы между собой. То есть предыдущие действия оказывают кумулятивный эффект на последующие.

В примере беспилотного автомобиля, после того как распределение отклонилось от экспертного, восстановить правильную траекторию будет очень трудно, поскольку плохие действия накапливаются, что приводит к катастрофическим последствиям. Чем длиннее траектория, тем хуже проявляется эффект подражательного обучения. Еще раз поясним: задачи обучения с учителем с независимыми и одинаково распределенными данными можно рассматривать как траектории длины 1, последствий для следующих действий нет вообще. Описанную выше парадигму мы ранее назвали *пассивным* обучением.

Для борьбы с катастрофическими последствиями отклонения от распределения для стратегий, обученных в режиме *пассивного* подражания, применяются другие подходы. Одни из них можно считать техническими уловками, другие являются изменениями в алгоритмах. Приведем пример двух таких подходов, доказавших свою эффективность:

- обучение модели, которая хорошо обобщается на новые данные без переобучения;
- использование активного подражания в дополнение к пассивному.

Поскольку первая идея принадлежит более широкому контексту, мы остановимся на второй.

Сравнение активного и пассивного подражания

Рассматривая обучение вождению машине, мы ввели термин *активное подражание*. При этом имелось в виду, что обучаемый получает дополнительную обратную связь от эксперта. В общем случае под активным подражанием понимается обучение на данных единой стратегии с действиями, назначаемыми экспертом.

Если говорить в терминах входа s (состояние или наблюдение) и выхода a (действие), то в пассивном обучении и s , и a могут исходить от эксперта. В активном обучении s выбирается обучаемым, а эксперт сообщает, какое действие a он предпринял бы в этом состоянии. Цель агента – обучиться отображению $\pi(a|s)$.

Активное обучение на данных единой стратегии позволяет обучаемому исправлять небольшие отклонения от экспертной траектории, чего он не смог бы сделать в случае одного лишь пассивного подражания.

ИГРА FLAPPY BIRD

Ниже в этой главе мы разработаем и протестируем алгоритм IL DAgger в новой окружающей среде Flappy Bird, которая имитирует знаменитую одноименную игру. Сейчас мы хотим представить инструменты, необходимые для реализации кода, взаимодействующего с этой средой, и начнем с описания интерфейса.

Среда Flappy Bird входит в состав **PyGame Learning Environment (PLE)** – набора окружающих сред для имитации интерфейса **Arcade Learning Environment (ALE)**. Он похож на интерфейс **Gym**, а различия мы опишем ниже. Так или иначе, использовать его легко.

Цель игры Flappy Bird – провести птицу между вертикальными трубами, не коснувшись их. Она управляется всего одним действием, которое заставляет птицу махать крыльями. Когда птица не летит, она снижается по траектории, диктуемой законом тяготения. Ниже показано, как выглядит экран игры (рис. 10.4).

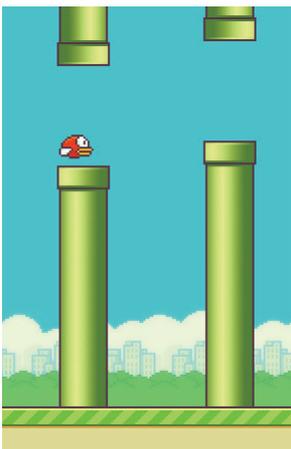


Рис. 10.4

Порядок взаимодействия с окружающей средой

Ниже описано, как используется окружающая среда.

1. Для использования Flappy Bird в Python-скриптах нужно первым делом импортировать PLE и Flappy Bird:

```
from ple.games.flappybird import FlappyBird
from ple import PLE
```

2. Затем создаем объект FlappyBird и передаем его функции PLE вместе с еще несколькими параметрами:

```
game = FlappyBird()
p = PLE(game, fps=30, display_screen=False)
```

Параметр `display_screen` говорит, куда выводить графику.

3. Для инициализации среды служит метод `init()`:

```
p.init()
```

Для взаимодействия со средой и получения ее состояния предназначены четыре основные функции:

- `p.act(act)` – выполнить действие `act` в игре. Вызов `act(act)` возвращает вознаграждение в результате выполненного действия;
- `p.game_over()` – проверить, достигнуто ли заключительное состояние;
- `p.reset_game()` – привести игру в начальное состояние;
- `p.getGameState()` – получить текущее состояние окружающей среды. Для получения RGB-наблюдения (т. е. полного экрана) можно использовать функцию `p.getScreenRGB()`.

4. Ниже показан простой скрипт, который играет пять игр Flappy Bird; в нем демонстрируется все сказанное выше. Заметим, что дополнительно необходимо определить функцию `get_action(state)`, которая возвращает действие, соответствующее переданному состоянию:

```
from ple.games.flappybird import FlappyBird
from ple import PLE

game = FlappyBird()
p = PLE(game, fps=30, display_screen=False)
p.init()

reward = 0

for _ in range(5):
    reward += p.act(get_action(p.getGameState()))

    if p.game_over():
        p.reset_game()
```

Обратим внимание на несколько моментов.

- `getGameState()` возвращает словарь, содержащий положение, скорость и дистанцию игрока, а также положение следующей трубы и следующей за ней. Прежде чем передать состояние творцу стратегии, который здесь представлен функцией `get_action`, словарь преобразуется в массив NumPy и нормируется.

- `act(action)` ожидает на входе `None`, если не нужно совершать никакого действия, или 119, если птица должна махать крыльями, чтобы подняться выше.

АЛГОРИТМ АГРЕГИРОВАНИЯ НАБОРА ДАННЫХ

Один из самых успешных алгоритмов, обучающихся на демонстрациях, называется **Dataset Aggregation (DAgger)**. Это метаалгоритм итеративной стратегии, который хорошо работает с индуцированным распределением состояний. Самое важное свойство DAgger заключается в том, что он справляется с отклонением от распределения, предлагая активный метод, позволяющий эксперту показать обучаемому, как восстановиться после ошибки.

Классический алгоритм ИЛ обучает классификатор, который предсказывает поведение эксперта. Это означает, что модель аппроксимирует набор данных, включающий обучающие примеры, предложенные экспертом. Входами в нем являются наблюдения, а действиями – желательные выходы. Но, как показывает приведенное выше рассуждение, предсказание, сделанное обучаемым, влияет на будущее посещенное состояние или наблюдение, т. е. нарушается условие независимости и одинакового распределения данных.

DAgger решает проблему изменения распределения, циклически выполняя конвейер агрегирования новых данных, полученных от обучаемого, и обучая агрегированный набор данных. Ниже показана простая диаграмма этого алгоритма (рис. 10.5).



Рис. 10.5

Эксперт формирует набор данных, используемый классификатором, но в зависимости от итерации действие, выполняемое в окружающей среде, может исходить как от эксперта, так и от обучаемого.

Алгоритм DAgger

В алгоритме DAgger итеративно повторяется следующая процедура. На первой итерации строится набор данных D , содержащий траектории, основанные на стратегии эксперта, после чего этот набор используется для обучения первой стратегии π_1 , которая оптимально аппроксимирует эти траектории, не допуская переобучения. Затем на i -й итерации, следуя уже обученной стратегии π_i , собираются новые траектории и добавляются в набор данных D . Агрегированный набор данных D , содержащий новые и старые траектории, используется для обучения новой стратегии π_{i+1} .

Согласно статье об алгоритме Dagger (<https://arxiv.org/pdf/1011.0686.pdf>), такое активное обучение с единой стратегией превосходит многие другие алгоритмы подражательного обучения, и с помощью глубоких нейронных сетей удается обучить очень сложные стратегии.

Дополнительно на i -й итерации стратегию можно модифицировать, так чтобы эксперт контролировал количество действий. Это позволяет лучше задействовать знания эксперта и дает возможность обучаемому постепенно принять на себя управление окружающей средой.

Псевдокод пояснит сказанное выше.

Инициализировать $D = \emptyset$

Инициализировать $\pi_0 = \pi'$ (π' - стратегия эксперта)

for $i \in 0..n$:

- > Поместить в набор данных D_i пары $(s, \pi'(s))$. Состояния дает стратегия
- > π_i (иногда эксперт может брать на себя управление ей), а действия -
- > стратегия эксперта π'
- > Обучить классификатор на агрегированном наборе данных

Реализация DAgger

Код состоит из трех основных частей:

- загрузить экспертную функцию логического вывода, которая предсказывает действие по заданному состоянию;
- создать граф вычислений для обучаемого;
- организовать итерации DAgger для построения набора данных и обучения новой стратегии.

Здесь мы объясним наиболее интересные части, оставив все прочее для самостоятельного изучения. Полный код имеется в репозитории этой книги на GitHub.

Загрузка экспертной модели логического вывода

В роли эксперта должна выступать стратегия, которая принимает состояние и возвращает наилучшее действие. Но в остальных отношениях она может быть любой. В частности, для экспериментов мы использовали в качестве агента, обученного алгоритмом проксимальной оптимизации стратегии (PPO). Какого-то особенного смысла в этом нет, но мы приняли такое решение из пе-

дагогических соображений, чтобы облегчить интеграцию с алгоритмами подражательного обучения.

Обученная модель эксперта (ее веса) была сохранена в файле, так что ее можно легко восстановить. Для построения графа и подготовки его к работе нужно выполнить три шага.

1. Импортировать метаграф с помощью метода `tf.train.import_meta_graph`.
2. Восстановить веса. Мы должны загрузить ранее обученные веса в только что импортированный граф вычислений. Для восстановления весов из последней контрольной точки нужно выполнить метод `tf.train.latest_checkpoint(session, checkpoint)`.
3. Получить доступ к выходным тензорам. Для доступа к тензорам восстановленного графа служит метод `graph.get_tensor_by_name(tensor_name)`, где `tensor_name` – имя тензора в графе.

Весь процесс реализован в следующих строчках:

```
def expert():
    graph = tf.get_default_graph()
    sess_expert = tf.Session(graph=graph)

    saver = tf.train.import_meta_graph('expert/model.ckpt.meta')
    saver.restore(sess_expert,tf.train.latest_checkpoint('expert/'))

    p_argmax = graph.get_tensor_by_name('actor_nn/max_act:0')
    obs_ph = graph.get_tensor_by_name('obs:0')
```

Далее, поскольку нас интересует простая функция, возвращающая действие эксперта по заданному состоянию, напомним функцию `expert`, которая будет возвращать такую функцию. То есть внутри `expert()` определяется внутренняя функция `expert_policy(state)`, которая и возвращается в качестве значения.

```
def expert_policy(state):
    act = sess_expert.run(p_argmax, feed_dict={obs_ph:[state]})
    return np.squeeze(act)

return expert_policy
```

Создание графа вычислений обучаемого

Весь приведенный ниже код находится в функции `DAgger`, которая принимает различные гиперпараметры.

Граф вычислений обучаемого простой, поскольку его единственная цель – построить классификатор. В нашем случае нужно предсказывать всего два действия: не делать ничего или махать крыльями. Создадим два местозаполнителя – для входного состояния и для действий эксперта, содержащих *объективную истину*. Действия обозначаются целыми числами: 0 (ничего не делать) и 1 (лететь).

Ниже перечислены шаги построения графа вычислений.

1. Создать глубокую нейронную сеть, точнее полносвязный многослойный перцептрон с функцией активации ReLu в скрытых слоях и линейной функцией активации в выходном слое.
2. Для каждого входного состояния выбирать действие с наибольшей ценностью. Это делает функция `tf.math.argmax(tensor,axis)` с `axis=1`.

3. Преобразовать местозаменители действий в унитарный тензор. Это необходимо, поскольку логиты и метки, используемые в функции потерь, должны иметь размерность `[batch_size, num_classes]`. Однако наши метки, `act_ph`, имеют форму `[batch_size]`. Следовательно, для преобразования их в требуемую форму нужно применить унитарное кодирование. Для этого как раз и предназначена функция TensorFlow `tf.one_hot`.
4. Определить функцию потерь. Мы используем функцию `softmax` для минимизации перекрестной энтропии. Это стандартная функция потерь, применяемая для дискретной классификации с взаимно исключающими классами – как раз наш случай. Значение `softmax` для перекрестной энтропии между логитами и метками вычисляется функцией `softmax_cross_entropy_with_logits_v2(labels, logits)`.
5. Наконец, вычисляется средняя потеря на пакете и минимизируется методом Adam.

Эти пять шагов реализуются следующим образом:

```
obs_ph = tf.placeholder(shape=(None, obs_dim), dtype=tf.float32, name='obs')
act_ph = tf.placeholder(shape=(None,), dtype=tf.int32, name='act')
p_logits = mlp(obs_ph, hidden_sizes, act_dim, tf.nn.relu,
last_activation=None)
act_max = tf.math.argmax(p_logits, axis=1)
act_onehot = tf.one_hot(act_ph, depth=act_dim)
p_loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(
labels=act_onehot, logits=p_logits))
p_opt = tf.train.AdamOptimizer(p_lr).minimize(p_loss)
```

Далее следует инициализировать сеанс, глобальные переменные и определить функцию `learner_policy(state)`, которая получает состояние и возвращает для него действие с наибольшей вероятностью, выбираемое обучаемым (то же самое мы делали для эксперта).

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

def learner_policy(state):
    action = sess.run(act_max, feed_dict={obs_ph:[state]})
    return np.squeeze(action)
```

Создание цикла *Dagger*

Настало время заняться ядром алгоритма *Dagger*. Общая структура уже была описана на псевдокоде выше, теперь приглядимся к деталям.

1. Инициализировать набор данных, состоящий из двух списков, `X` и `y`, в которые мы будем помещать посещенные состояния и целевые действия, подсказанные экспертом. Также инициализируем окружающую среду.

```
X = []
y = []

env = FlappyBird()
env = PLE(env, fps=30, display_screen=False)
env.init()
```

2. Выполнять итерации DAgger. В начале каждой итерации нужно заново инициализировать граф вычислений (поскольку на каждой итерации обучаемый переобучается на новом наборе данных), привести среду в исходное состояние и выполнить сколько-то случайных действий. Эти действия привносят случайность в детерминированную окружающую среду, в результате чего стратегия получается более робастной.

```
for it in range(dagger_iterations):
    sess.run(tf.global_variables_initializer())
    env.reset_game()
    np_op(env)

    game_rew = 0
    rewards = []
```

3. Собирать новые данные во взаимодействии с окружающей средой. Как уже было сказано, на первой итерации эксперт выбирает действия, вызывая функцию `expert_policy`, но на последующих управление постепенно берет в свои руки обучаемый. Обученная стратегия выполняется функцией `learner_policy`. Набор данных строится путем добавления x (входная переменная) к текущему состоянию игры и добавления y (выходная переменная) к действиям, которые эксперт предпринял бы в данном состоянии. По окончании игра возвращается в исходное состояние и переменная `game_rew` устанавливается в 0. Код приведен ниже.

```
for _ in range(step_iterations):
    state = flappy_game_state(env)

    if np.random.rand() < (1 - it/5):
        action = expert_policy(state)
    else:
        action = learner_policy(state)

    action = 119 if action == 1 else None

    rew = env.act(action)
    rew += env.act(action)

    X.append(state)
    y.append(expert_policy(state))
    game_rew += rew

    if env.game_over():
        env.reset_game()
        np_op(env)

        rewards.append(game_rew)
        game_rew = 0
```

Отметим, что действия выполняются дважды. Так делается, чтобы уменьшать количество действий каждые 15 секунд вместо 30, как того требует среда.

4. Обучить новую стратегию на агрегированном наборе данных. Порядок действий стандартный. Набор данных перемешивается и разбивается на мини-пакеты размера `batch_size`. Затем производится оптимизация, для

чего `p_opt` выполняется `train_epochs` раз на каждом мини-пакете. Вот соответствующий код:

```
n_batches = int(np.floor(len(X)/batch_size))
shuffle = np.arange(len(X))
np.random.shuffle(shuffle)
shuffled_X = np.array(X)[shuffle]
shuffled_y = np.array(y)[shuffle]
ep_loss = []

for _ in range(train_epochs):
    for b in range(n_batches):
        p_start = b*batch_size
        tr_loss, _ = sess.run([p_loss, p_opt], feed_dict=
            obs_ph:shuffled_X[p_start:p_start+batch_size],
            act_ph:shuffled_y[p_start:p_start+batch_size])

        ep_loss.append(tr_loss)
print('Ep:', it, np.mean(ep_loss), 'Test:',
    np.mean(test_agent(learner_policy)))
```

Функция `test_agent` тестирует `learner_policy` на нескольких играх, чтобы оценить, насколько хорошо действует обучаемый.

Анализ результатов игры в Flappy Bird

Прежде чем переходить к результатам подражательного обучения, хотим познакомить вас с некоторыми данными, чтобы вы могли сравнить этот подход и обучение с подкреплением. Мы понимаем, что это сравнение не совсем честное (алгоритмы работают в существенно различных условиях), но все же оно показывает, почему подражательное обучение может принести плоды, если, конечно, имеется эксперт.

Эксперт обучался методом проксимальной оптимизации стратегии на протяжении 2 млн шагов и после примерно 400 000 шагов вышел на плато, набрав 138 баллов.

Мы тестировали DAgger в среде Flappy Bird со следующими гиперпараметрами:

Гиперпараметр	Имя переменной	Значение
Число скрытых слов сети обучаемого	<code>hidden_sizes</code>	16,16
Число итераций DAgger	<code>dagger_iterations</code>	8
Скорость обучения	<code>p_lr</code>	1e-4
Число шагов на каждой итерации DAgger	<code>step_iterations</code>	100
Размер мини-пакета	<code>batch_size</code>	50
Число эпох обучения	<code>train_epochs</code>	2000

График на рис. 10.6 показывает, как изменяется качество DAgger с увеличением числа шагов.

Горизонтальная линия показывает среднее качество, достигнутое экспертом. Из результатов видно, что для выхода на плато достаточно нескольких сотен шагов. По сравнению с объемом опыта, необходимого алгоритму PPO

для обучения эксперта, это примерно 100-кратное увеличение выборочной эффективности.

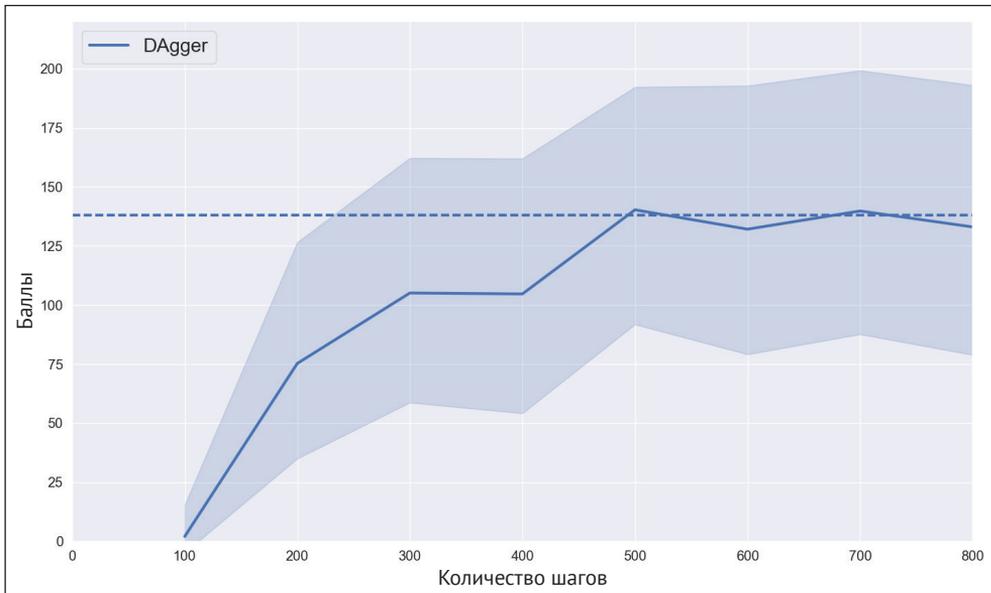


Рис. 10.6

Еще раз подчеркнем, что сравнение проводилось в разных контекстах, но оно наводит на мысль, что при наличии эксперта лучше использовать подражательное обучение (хотя бы для обучения начальной стратегии).

ОБРАТНОЕ ОБУЧЕНИЕ С ПОДКРЕПЛЕНИЕМ

Одно из самых серьезных ограничений подражательного обучения – невозможность обучиться другим траекториям на пути к цели, кроме предложенных экспертом. Подражая эксперту, обучаемый вынужден оставаться в рамках поведения, известного учителю. Он не знает о конечной цели, к которой стремится эксперт. Поэтому такие методы полезны лишь в тех случаях, когда не ставится задача превзойти учителя.

Обратное обучение с подкреплением (IRL) – это алгоритм ОП, в котором, как и в IL, для обучения используется эксперт. Разница в том, что в IRL требуется обучиться функции вознаграждения эксперта. Поэтому вместо копирования демонстраций, как при подражательном обучении, IRL старается узнать цель эксперта. Обучившись функции вознаграждения, агент использует ее для обучения стратегии.

Поскольку демонстрации используются, только чтобы понять цель эксперта, агент не связан действиями учителя и в итоге может обучиться лучшей стратегии. Например, беспилотный автомобиль, обучаемый методами IRL, может

понять, что цель состоит в том, чтобы добраться из пункта А в пункт В за минимальное время, минимизировав при этом ущерб, нанесенный людям и предметам. Затем автомобиль самостоятельно выработает стратегию (например, с помощью какого-нибудь алгоритма ОП), которая максимизирует эту функцию вознаграждения.

Однако у IRL тоже есть ряд недостатков, ограничивающих его применимость. Если проведенная экспертом демонстрация не оптимальна, то обучаемый не сможет раскрыть весь свой потенциал, так и оставшись с неправильной функцией вознаграждения. Другая проблема – как оценить обученную функцию вознаграждения.

РЕЗЮМЕ

В этой главе мы отвлеклись от алгоритмов обучения с подкреплением и изучили другой тип обучения – подражательное. Новизна этой парадигмы связана с тем, как организовано обучение – итоговая стратегия подражает поведению эксперта. Это отличается от обучения с подкреплением отсутствием сигнала вознаграждения и способностью задействовать богатый источник информации в лице эксперта.

Мы видели, что набор данных, на котором обучается агент, можно дополнить новыми парами состояние–действие, чтобы повысить уверенность обучаемого при встрече с неизвестными ситуациями. Этот процесс называется агрегированием данных. При этом новые данные могут поступать от только что обученной стратегии, и в таком случае мы говорим о данных единой стратегии. Такая интеграция состояний единой стратегии и обратной связи с экспертом является очень ценным подходом, который повышает качество обучаемого агента.

Затем мы описали и реализовали один из самых успешных алгоритмов подражательного обучения, DAgger, и применили его к игре Flappy Bird.

Но поскольку алгоритмы подражательного обучения всего лишь копируют поведение эксперта, они никогда не могут стать лучше эксперта. Поэтому мы рассказали об обратном обучении с подкреплением – методе, в котором эта проблема преодолевается путем вывода функции вознаграждения из поведения эксперта. Таким образом, стратегии можно обучаться независимо от учителя.

В следующей главе мы рассмотрим еще один набор алгоритмов для решения последовательных задач – эволюционные алгоритмы. Вы узнаете о механизмах и достоинствах оптимизации методом черного ящика и сможете применить их в трудной окружающей среде. Кроме того, мы подробно рассмотрим один конкретный эволюционный алгоритм – эволюционную стратегию – и реализуем его.

Вопросы

1. Считается ли подражательное обучение одним из методов обучения с подкреплением?

2. Стали бы вы использовать подражательное обучение для создания непобедимого агента для игры го?
3. Как расшифровывается аббревиатура DAgger?
4. В чем основное достоинство DAgger?
5. Когда бы вы стали использовать IRL вместо IL?

Для дальнейшего чтения

- Оригинальная статья о DAgger «A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning» находится по адресу <https://arxiv.org/pdf/1011.0686.pdf>.
- Дополнительные сведения об алгоритмах подражательного обучения см. в статье «Global Overview of Imitation Learning» по адресу <https://arxiv.org/pdf/1801.06503.pdf>.
- Вопросам обратного обучения с подкреплением посвящен обзор «A Survey of Inverse Reinforcement Learning: Challenges, Methods and Progress» (<https://arxiv.org/pdf/1806.06877.pdf>).

Глава 11

Оптимизация методом черного ящика

В предыдущих главах мы рассмотрели алгоритмы обучения с подкреплением – основанные на ценности и на стратегии, безмодельные и на основе модели. В этой главе мы опишем другой способ решения последовательных задач, а именно класс **эволюционных алгоритмов (ЭА)**, в основе которых лежит метод черного ящика. Эволюционные алгоритмы подражают механизмам эволюции и иногда оказываются предпочтительнее обучения с подкреплением (ОП), поскольку не требуют обратного распространения. Зачастую они предлагают и другие преимущества по сравнению с ОП. Мы начнем эту главу с краткого обзора алгоритмов ОП, чтобы вы могли лучше понять место ЭА в такого рода задачах. Затем расскажем об основных элементах ЭА и принципах работы этих алгоритмов. Мы также более подробно рассмотрим один из самых известных ЭА, а именно **эволюционные стратегии (ЭС)**.

Алгоритм, недавно разработанный компанией OpenAI, вызвал всплеск интереса к применению ЭС для решения последовательных задач. Было показано, что алгоритмы ЭС допускают массовое распараллеливание и линейно масштабируются на несколько процессоров, достигая при этом высокой производительности. Объяснив идеи, лежащие в основе эволюционных стратегий, мы детально рассмотрим сам алгоритм и реализуем его с помощью TensorFlow, чтобы продемонстрировать, как он применяется к конкретным задачам.

В этой главе рассматриваются следующие вопросы:

- за рамками ОП;
- основы эволюционных алгоритмов;
- масштабируемые эволюционные стратегии;
- применение масштабируемых ЭС к среде LunarLander.

ЗА РАМКАМИ ОП

Алгоритмы ОП обычно выбирают, сталкиваясь с задачами, в которых требуется последовательное принятие решений. Как правило, другие способы решения таких задач найти трудно. Несмотря на существование сотен методов оптимизации, только ОП показывает хорошие результаты, когда нужно принимать решения последовательно. Но это не значит, что больше вообще ничего нет.

Мы начнем эту главу с того, что вспомним о принципах работы алгоритмов ОП и зададимся вопросом о полезности их компонентов для решения последовательных задач. Это подведет нас к алгоритмам нового типа, обладающим многочисленными преимуществами (и некоторыми недостатками), которые можно использовать как замену ОП.

Краткий обзор ОП

В начале работы стратегия инициализируется случайным образом и используется для взаимодействия с окружающей средой с целью сбора данных на протяжении либо заданного количества шагов, либо полных траекторий. При каждом взаимодействии посещается некоторое состояние, предпринимается действие и запоминается полученное вознаграждение. На основе этой информации можно составить полное описание воздействия агента на окружающую среду. Затем для улучшения стратегии выполняется алгоритм обратного распространения (цель которого – минимизировать функцию потерь и тем улучшить предсказания), который вычисляет градиент по каждому весу сети. Эти градиенты используются в методе стохастического градиентного спуска. Весь процесс (сбор данных от окружающей среды и оптимизация нейронной сети методом СГС) повторяется, пока не будет выполнен критерий сходимости.

Далее нам будут важны два момента.

- **Распределение поощрения во времени.** Поскольку алгоритмы ОП оптимизируют стратегию на каждом шаге, требуется назначить показатель качества каждому действию и состоянию. Для этого каждой паре состояние–действие присваивается ценность. А чтобы уменьшить влияние отдаленных действий и увеличить значимость близких по времени, применяется коэффициент обесценивания. Это помогает решить проблему назначения поощрения действия, но одновременно вносит неточности в систему.
- **Исследование.** Чтобы поддерживать необходимую степень исследования, в стратегию вносится искусственный шум. Как именно это делается, зависит от алгоритма, но обычно действия выбираются из стохастического распределения. Из-за этого агент, дважды оказавшийся в одной и той же ситуации, может выбрать разные действия и пойти по двум разным путям. Такой подход поощряет исследование даже в детерминированных средах. Выбрав другой путь, агент может обнаружить новые, возможно лучшие, решения. Если добавочный шум асимптотически стремится к 0, то агент в итоге сойдется к окончательной детерминированной стратегии.

Но являются ли обратное распространение, распределение поощрений во времени и стохастические действия необходимым условием обучения и построения сложных стратегий?

Альтернатива

На этот вопрос ответ отрицательный.

В главе 10 мы видели, что, сведя обучение стратегии к подражательной задаче с использованием обратного распространения и СГС, мы можем обучить-

ся у эксперта дискриминантной модели, предсказывающей, какое действие выбрать следующим. Но все равно требуется обратное распространение, да к тому же эксперта может и не оказаться.

Существует еще одно подмножество алгоритмов глобальной оптимизации. Это эволюционные алгоритмы (ЭА), они не основаны на обратном распространении и не опираются на два других принципа – распределение поощрения во времени и зашумленные действия. И как было сказано во введении к этой главе, эволюционные алгоритмы очень общие и применимы к широкому кругу задач, включая задачи последовательного принятия решений.

Эволюционные алгоритмы

Как вы, наверное, догадались, ЭА во многих отношениях отличаются от алгоритмов ОП и черпают идеи из биологической эволюции. К классу ЭА можно отнести различные похожие методы: генетические алгоритмы, эволюционные стратегии и генетическое программирование, различающиеся деталями реализации и характером представления. Но все они основаны на четырех базовых механизмах: воспроизводство, мутация, скрещивание и отбор (селекция), – которые повторяются в цикле выдвигания и проверки гипотез. Что это все значит, мы увидим ниже в данной главе.

Эволюционные алгоритмы считаются методами черного ящика. Они оптимизируют функции $f(w)$ относительно w , не делая никаких предположений о f . То есть сама f может быть какой угодно, а нас интересует только выход f . У такого подхода много достоинств, но есть и недостатки. Главное достоинство состоит в том, что не нужно думать о структуре f , и мы вольны брать то, что больше подходит и лучше отвечает конкретной задаче. А главный недостаток в том, что такие методы оптимизации невозможно объяснить, и их механизм не поддается интерпретации. Если интерпретируемость важна, то от этих методов придется отказаться.

Для решения последовательных задач, особенно трудных и умеренно трудных, почти всегда предпочитали обучение с подкреплением. Но в недавней работе компании OpenAI описан эволюционный алгоритм эволюционной стратегии, который может служить альтернативой ОП. В обоснование этого заявления выдвигается главным образом асимптотическое качество алгоритма, а также его невероятная способность масштабироваться на тысячи процессоров.

Прежде чем объяснять, почему этот алгоритм так хорошо масштабируется и при этом обучается хорошим стратегиям на трудных задачах, поговорим подробнее об эволюционных алгоритмах вообще.

ОСНОВЫ ЭВОЛЮЦИОННЫХ АЛГОРИТМОВ

В основе ЭА лежат идеи биологической эволюции, и таковы же механизмы их реализации. Это означает, что ЭА совершает много попыток создать популяцию потенциальных решений. Это те решения, называемые также **особями** (в задачах ОП потенциальным решением является стратегия), которые оказались лучше, чем в предыдущем поколении. Точно так же в природе выживает и получает возможность размножиться сильнейший.

Одно из преимуществ ЭА заключается в том, что они не нуждаются в вычислении производных для нахождения решения. Поэтому они могут одинаково успешно работать с дифференцируемыми и недифференцируемыми функциями, в т. ч. с глубокими нейронными сетями. Эта комбинация схематически изображена на рис. 11.1. Отметим, что каждая особь – это отдельная глубокая нейронная сеть, поэтому в каждый момент времени нейронных сетей столько, сколько особей. На рис. 11.1 популяция состоит из пяти особей.

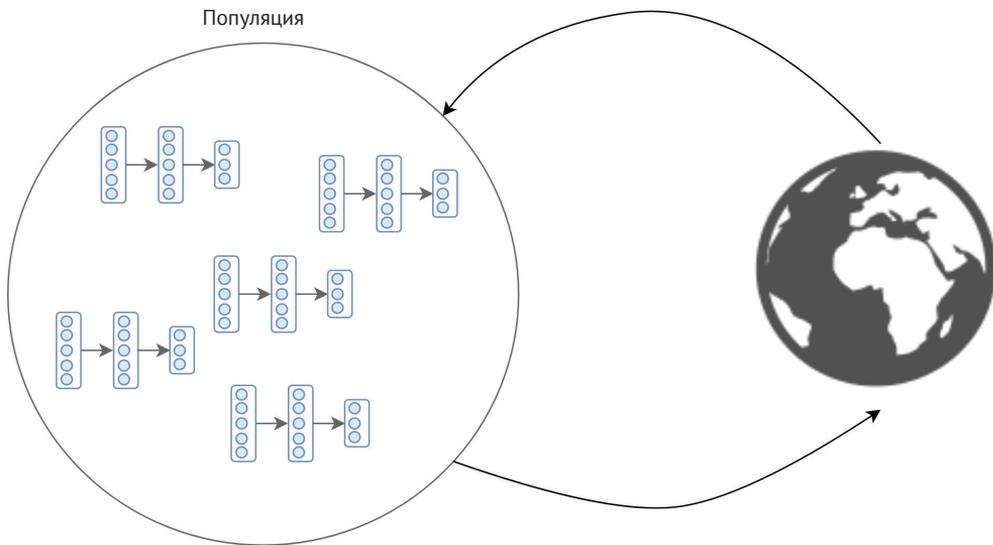


Рис. 11.1 ❖ Оптимизация нейронной сети с помощью эволюционных алгоритмов

У каждого эволюционного алгоритма есть свои особенности, но базовый цикл общий для всех и выглядит следующим образом:

- 1) популяция особей (они же **потенциальные решения**, или **фенотипы**) создается таким образом, чтобы свойства (**хромосомы**, или **генотипы**) всех особей различались. Начальная популяция инициализируется случайным образом;
- 2) каждое потенциальное решение независимо оценивается функцией приспособленности, которая определяет его качество. Обычно функция приспособленности связана с целевой функцией, и если использовать нашу привычную терминологию, то в роли функции приспособленности могло бы выступать полное вознаграждение, полученное агентом (т. е. потенциальным решением) на протяжении всей его жизни;
- 3) затем из популяции выбираются более приспособленные особи, и их генотип модифицируется для порождения нового поколения. В некоторых случаях менее приспособленные потенциальные решения могут использоваться как отрицательные примеры для следующего поколения. Этот шаг сильно зависит от алгоритма. Например, в генетических алгоритмах для выведения новых особей (**потомков**) применяются два процесса,

мутация и скрещивание. А в эволюционных стратегиях новые особи выводятся только путем мутации. Ниже в этой главе мы подробнее объясним, что такое скрещивание и мутация, но, говоря в общих словах, при скрещивании комбинируется генетическая информация обоих родителей, а при мутации изменяются только значения некоторых генов потомка;

- 4) весь процесс (шаги 1–3) повторяется, пока не будет выполнено условие завершения. Популяция, создаваемая на каждой итерации, называется **поколением**.

Этот итеративный процесс, показанный на рис. 11.2, завершается по достижении заданного уровня приспособленности или после порождения максимального количества поколений. В данном случае популяция создается с помощью скрещивания и мутации, но, как мы уже объяснили, процесс зависит от конкретного алгоритма.

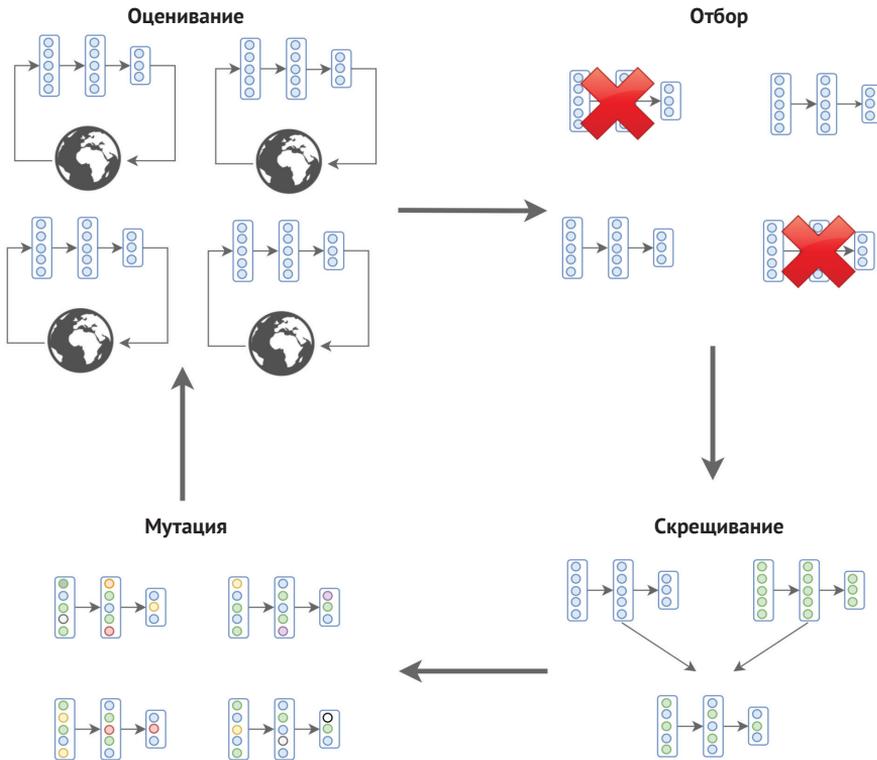


Рис. 11.2 ❖ Главный цикл эволюционных алгоритмов

Тело общего ЭА выглядит очень просто, для его записи достаточно всего нескольких строчек кода. В двух словах: на каждой итерации порождаются и оцениваются новые потенциальные решения (кандидаты) – и так до тех пор, пока не будет создано приспособленное поколение. Кандидаты создаются из максимально приспособленных особей в предыдущем поколении.

```

solver = EvolutionaryAlgorithm()
while best_fitness < required_fitness:
    candidates = solver.generate_candidates() # например, путем мутации и скрещивания
    fitness_values = []
    for candidate in candidates:
        fitness_values.append(evaluate(candidate))
    solver.set_fitness_values(fitness_values)
    best_fitness = solver.evaluate_best_candidate()

```

❗ Детали реализации решателя solver зависят от конкретного алгоритма.

ЭА находят применения в самых разных областях знаний и задачах, от экономики до биологии и от оптимизации компьютерных программ до оптимизации муравьиной колонии.

Поскольку нас больше всего интересует применение эволюционных алгоритмов к решению последовательных задач принятия решений, мы рассмотрим два самых известных ЭА, используемых в этой области: генетические алгоритмы (ГА) и эволюционные стратегии (ЭС). А затем разработаем хорошо масштабируемый вариант ЭС.

Генетические алгоритмы

Идея ГА очень проста – оценить текущее поколение, использовать только самых лучших особей для порождения следующих потенциальных решений и отбросить остальных особей. Это было показано на рис. 11.2. Из выживших образуется новое поколение с помощью скрещивания и мутации. Для скрещивания выбираются два выживших решения, и их параметры комбинируются. Для мутации изменяется несколько случайно выбранных параметров генотипа потомка (рис. 11.3).

Скрещивание и мутацию можно реализовать по-разному. Самый простой способ скрещивания – случайным образом выбрать части двух родителей, а мутации – добавить к решению гауссов шум с фиксированным стандартным отклонением. Если оставлять только лучших особей и наделять их генами вновь порожденных особей, то со временем решения будут улучшаться, пока не окажется выполнено условие завершения. Но в сложных задачах такой подход может привести к застреванию в локальном оптимуме (когда решение оптимально только на небольшом множестве близких потенциальных решений). В таком случае предпочтительнее более развитый генетический алгоритм, например **нейроэволюции нарастающих топологий** (NeuroEvolution of Augmenting Topologies – NEAT). NEAT изменяет не только веса сети, но и ее структуру.

Эволюционные стратегии

Эволюционные стратегии (ЭС) даже проще генетических алгоритмов, поскольку для создания новых популяций применяется только мутация.

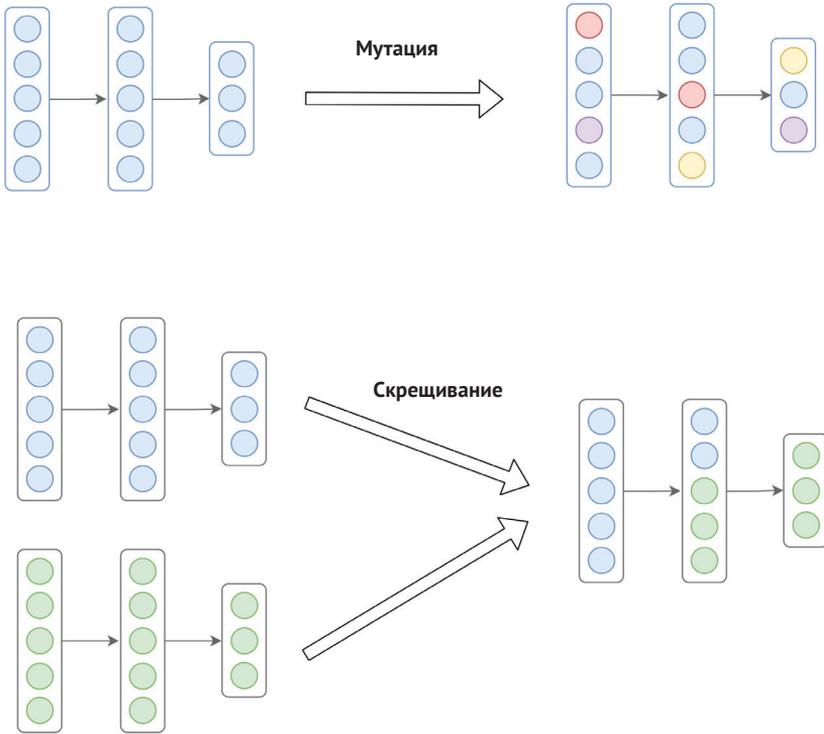


Рис. 11.3 ❖ Иллюстрация мутации и скрещивания

Для мутации к генотипу прибавляются значения, выбранные из нормального распределения. Очень простой вариант ЭС получается, если просто выбрать лучшую особь из всей популяции и в следующее поколение включить особей, выбираемых из нормального распределения с таким же средним, как у лучшей особи, и с фиксированным стандартным отклонением.

Но такой алгоритм можно рекомендовать только для совсем небольших задач, поскольку имеется всего один лидер, а, фиксируя стандартное отклонение, мы препятствуем масштабному исследованию пространства решений. В результате мы, скорее всего, найдем узкий локальный минимум и в нем и останемся. Более перспективная стратегия – генерировать потомка, скомбинировав двух лучших особей с весами, соответствующими их приспособленности. Упорядочение особей при таком подходе называется ранжированием по приспособленности. Эта стратегия предпочтительнее, чем использование самих значений приспособленности, поскольку инвариантна относительно преобразования целевой функции и препятствует слишком сильному движению нового поколения в сторону возможного выброса.

CMA-ES

Эволюционная стратегия с адаптацией ковариационной матрицы (Covariance Matrix Adaptation Evolution Strategy – **CMA-ES**) – один из алгоритмов эволюционной стратегии. В отличие от простейшего варианта, он выбирает

новое потенциальное решение из многомерного нормального распределения. Название СМА связано с тем, что зависимости между параметрами хранятся в ковариационной матрице, которая адаптируется так, чтобы увеличить или уменьшить пространство поиска в следующем поколении.

Проще говоря, алгоритм СМА-ES сжимает пространство поиска, постепенно уменьшая ковариационную матрицу в заданном направлении, если уверен в пространстве вокруг нее. И наоборот, ковариационная матрица увеличивается, расширяя пространство поиска, если алгоритм в нем не уверен.

Сравнение ЭС и ОП

Эволюционные стратегии – интересная альтернатива ОП. Но тем не менее необходимо оценить все плюсы и минусы, чтобы выбрать правильный подход. Перечислим основные преимущества ЭС.

- **Отсутствие производных.** Нет необходимости в обратном распространении. Для оценивания функции приспособленности нужен только прямой проход (или, что эквивалентно, накопленное вознаграждение). Это открывает возможность использовать любые недифференцируемые функции, например механизмы жесткого внимания. К тому же, избавившись от обратного распространения, код становится более быстрым.
- **Большая общность.** Своей общностью ЭС в основном обязан тому, что оптимизация производится методом черного ящика. Поскольку нас не интересует, какие состояния посещает агент и какие действия он выбирает, мы можем от всего этого абстрагироваться и сосредоточиться только на оценивании. Кроме того, ЭС допускает обучение без явного назначения целей и с очень разреженной обратной связью. И еще алгоритм ЭС более общий в том смысле, что способен оптимизировать гораздо более широкое множество функций.
- **Высокая степень параллелизма и робастность.** Как мы вскоре увидим, ЭС гораздо проще распараллелить, чем ОП, так что вычисления можно разнести на тысячи исполнительных устройств. Робастность эволюционных стратегий обусловлена малым количеством гиперпараметров. Например, в отличие от ОП, не нужно задавать длину траекторий, величину λ , коэффициент обесценивания, количество пропускаемых кадров и т. д. К тому же ЭС весьма привлекателен в задачах с очень длинным горизонтом.

С другой стороны, обучение с подкреплением обладает следующими полезными свойствами.

- **Выборочная эффективность.** Алгоритмы ОП лучше используют информацию, полученную от окружающей среды, поэтому им нужно меньше данных и меньше шагов, чтобы обучиться решению задачи.
- **Выдающееся качество.** В целом алгоритмы обучения с подкреплением по качеству результатов превосходят эволюционные стратегии.

МАСШТАБИРУЕМЫЕ ЭВОЛЮЦИОННЫЕ СТРАТЕГИИ

Познакомившись в общих чертах с эволюционными алгоритмами и эволюционными стратегиями в частности, мы готовы применить полученные зна-

ния на практике. Статья OpenAI «Evolution Strategies as a Scalable Alternative to Reinforcement Learning» внесла важный вклад в становление эволюционных стратегий как альтернативы алгоритмам обучения с подкреплением.

Главное достижение этой статьи – описание подхода к масштабированию ЭС на большое число процессоров. Конкретно, в нем используется новаторская стратегия обмена данными между процессорами, в котором участвуют только скаляры, поэтому она масштабируется на тысячи параллельно работающих исполнителей.

В общем случае для ЭС необходим больший объем опыта, поэтому такие алгоритмы менее эффективны, чем ОП. Но благодаря распределению вычислений между таким большим числом исполнителей эту задачу можно решить за меньшее время. В качестве примера авторы статьи решили задачу о трехмерном шагающем андроиде всего за 10 минут на 1440 процессорах, продемонстрировав линейное ускорение с увеличением количества процессоров. Алгоритмам ОП, не способным достичь такого уровня масштабируемости, для решения той же задачи необходимо несколько часов.

Основной принцип

В оригинальной статье ЭС максимизирует среднее, выступающее в роли целевой функции:

$$E_{\theta \sim p_{\mu}} F(\theta).$$

Для этого в популяции p_{μ} с параметром μ производится поиск методом стохастического градиентного подъема. Здесь F – целевая функция (или функция приспособленности), а θ – параметры исполнителя. В наших задачах это просто стохастический доход, полученный агентом в окружающей среде.

Генеральная совокупность (популяция) p_{μ} имеет многомерное нормальное распределение со средним μ и фиксированным стандартным отклонением σ :

$$E_{\theta \sim p_{\mu}} F(\theta) = E_{\varepsilon \sim N(0, I)} F(\theta + \sigma \varepsilon). \quad (11.1)$$

Отсюда можно определить обновление на одном шаге с помощью оценки стохастического градиента:

$$\theta \leftarrow \theta + \alpha \frac{1}{n\sigma} \sum_{i=1}^n F(\theta + \sigma \varepsilon_i) \varepsilon_i. \quad (11.2)$$

При таком обновлении мы можем оценить стохастический градиент (не выполняя обратного распространения), пользуясь результатами эпизодов из популяции. Для обновления параметров можно применить какой-нибудь из широко известных методов, например Adam или RMSProp.

Распараллеливание ЭС

Легко понять, как ЭС масштабируется на несколько процессоров: каждому процессору назначается свое потенциальное решение из популяции. Его оценивание можно провести полностью автономно, и, как описано в статье, оптимизацию можно выполнить параллельно, так что процессорам придется обмениваться лишь несколькими скалярами.

Точнее, единственная информация, разделяемая между исполнителями, – скалярный доход $F(\theta + \sigma\varepsilon_i)$ в эпизоде и случайное начальное значение, использованное для выборки. Объем данных можно еще уменьшить, если передавать только доход, но в этом случае начальные значения всех исполнителей необходимо синхронизировать. Мы решили остановиться на первом варианте, а авторы статьи – на втором. В нашей простой реализации различие пренебрежимо мало, и потребляемая в обоих случаях полоса пропускания крайне незначительна.

Другие приемы

Есть еще два способа улучшить качество алгоритма.

- **Ранжирование по приспособленности.** Мы уже обсуждали эту технику выше. Все очень просто. При вычислении обновления используются не сами доходы, а их ранги. Ранг инвариантен относительно преобразования целевой функции и потому лучше работает в условиях рассредоточенного дохода. Кроме того, он позволяет избавиться от шума в виде выбросов.
- **Отражение шума.** Этот прием уменьшает дисперсию и заключается в оценивании сети с шумом ε и $-\varepsilon$; т. е. для каждой особи мы имеем две мутации: $\theta_+ = \mu + \sigma\varepsilon$ и $\theta_- = \mu - \sigma\varepsilon$.

Псевдокод

Ниже приведен псевдокод распараллеленной эволюционной стратегии, включающий все сказанное.

 Распараллеленная эволюционная стратегия

Инициализировать параметры θ_0 на каждом исполнителе.

Инициализировать начальное значение на каждом исполнителе.

for *iteration* = 1..*M* **do**:

for *worker* = 1..*N* **do**:

 Произвести выборку $\varepsilon \sim N(0, I)$

 Оценить особей $F(\theta_t + \sigma\varepsilon)$ и $F(\theta_t - \sigma\varepsilon)$

 Все исполнители передают друг другу вычисленные доходы

for *worker* = 1..*N* **do**:

 Вычислить нормированный ранг *K* по доходам

 Реконструировать ε_i по случайным начальным значениям других исполнителей

$\theta_{t+1} \leftarrow \theta_t + \alpha \frac{1}{n\sigma} \sum_{i=1}^n K_i \varepsilon_i$ (например, с помощью Adam)

Осталось только реализовать этот алгоритм.

Масштабируемая реализация

Чтобы упростить реализацию и добиться хорошей работы распараллеленной версии ЭС в условиях ограниченного числа исполнителей (и процессоров), мы разработаем структуру, изображенную на рис. 11.4. Главный процесс создает

по одному исполнителю для каждого процессорного ядра и запускает главный цикл. На каждой итерации он ждет, когда исполнители оценят заданное число новых кандидатов. В отличие от реализации, предложенной в оригинальной статье, у нас каждый исполнитель оценивает на каждой итерации более одного агента. Итак, при наличии четырех процессорных ядер будет создано четыре исполнителя. Если мы хотим, чтобы размер пакета был больше количества исполнителей на каждой итерации главного цикла, например 40, то каждый исполнитель будет создавать и оценивать по 10 особей. Доходы и начальные значения возвращаются главному процессу, который ждет результатов обо всех 40 особях и, только получив их, продолжает работу.



Рис. 11.4 ❖ Основные компоненты параллельной версии ЭС

Затем эти результаты одним пакетом передаются всем исполнителям, которые независимо оптимизируют нейронную сеть, применяя обновление (11.2).

В соответствии с описанием код разделен на три части:

- главный процесс, которые создает очереди и исполнителей и управляет ими;
- функция, инкапсулирующая задачу, решаемую исполнителями;
- дополнительные функции для таких простых задач, как ранжирование доходов и оценивание агента.

Сначала рассмотрим код главного процесса, чтобы составить общее представление об алгоритме, прежде чем переходить к подробному описанию исполнителей.

Главная функция

Это функция `ES`, принимающая следующие аргументы: имя окружающей среды `Gym`, размер скрытых слоев нейронной сети, общее количество поколений, количество исполнителей, скорость обучения алгоритма `Adam`, размер пакета и стандартное отклонение шума:

```
def ES(env_name, hidden_sizes=[8,8], number_iter=1000, num_workers=4,
      lr=0.01, batch_size=50, std_noise=0.01):
```

Здесь мы задаем начальное значение генератора случайных чисел, общее для всех исполнителей и используемое для инициализации весов сетей, которые должны быть одинаковы. Далее вычисляется, сколько особей должен создавать и оценивать на каждой итерации каждый исполнитель. Кроме того, создаются две очереди `multiprocessing.Queue`. Через эти очереди происходит обмен данными между исполнителями.

```
initial_seed = np.random.randint(1e7)
indiv_per_worker = int(batch_size / num_workers)
output_queue = mp.Queue(maxsize=num_workers*indiv_per_worker)
params_queue = mp.Queue(maxsize=num_workers)
```

Далее создаются процессы `multiprocessing.Process`. Они асинхронно исполняют функцию `worker`, переданную в первом аргументе конструктору `Process`. Все остальные аргументы, перечисленные в списке `args`, без изменения передаются `worker`. Это те же аргументы, что были переданы функции `ES`, плюс две очереди. Процессы запускаются в момент вызова метода `start()`:

```
processes = []
for widx in range(num_workers):
    p = mp.Process(target=worker, args=(env_name, initial_seed,
    hidden_sizes, lr, std_noise, indiv_per_worker, str(widx), params_queue,
    output_queue))
    p.start()
    processes.append(p)
```

После запуска параллельных исполнителей мы можем пройтись по поколениям и дождаться, когда все особи будут сгенерированы и оценены отдельно каждым исполнителем. Напомним, что общее количество особей в поколении равно количеству исполнителей `num_workers`, умноженному на число особей, сгенерированных каждым исполнителем, `indiv_per_worker`. Эта архитектура выбрана специально для нашей реализации, поскольку у нас всего четыре процессорных ядра, а не тысячи, как в реализации, описанной в статье. В общем случае размер популяции, создаваемой в каждом поколении, обычно находится в пределах от 20 до 1000:

```
for n_iter in range(number_iter):
    batch_seed = []
    batch_return = []

    for _ in range(num_workers*indiv_per_worker):
        p_rews, p_seed = output_queue.get()
        batch_seed.append(p_seed)
        batch_return.extend(p_rews)
```

Здесь вызов `output_queue.get()` получает элемент из очереди `output_queue`, заполняемой исполнителями. В нашей реализации `output_queue.get()` возвращает два элемента. Первый, `p_gews`, – приспособленность (величина дохода) агента, сгенерированного по начальному значению `p_seed`, которое находится во втором элементе.

Когда цикл `for` завершается, мы ранжируем доходы и помещаем пакеты доходов и начальных значений в очередь `params_queue`, которая читается всеми исполнителями для оптимизации агента. Ниже приведен код.

```
batch_return = normalized_rank(batch_return)
for _ in range(num_workers):
    params_queue.put([batch_return, batch_seed])
```

Наконец, по завершении всех итераций обучения исполнителей можно остановить:

```
for p in processes:
    p.terminate()
```

На этом главная функция закончена. Осталось реализовать исполнителей.

Исполнители

Функциональность исполнителей инкапсулирована в функции `worker`, которая была передана в качестве аргумента конструктору `mp.Process`. Мы не станем объяснять весь код, поскольку это заняло бы слишком много места и времени, но основные части опишем. Как обычно, полная реализация имеется в репозитории этой книги на GitHub.

В первых строчках `worker` создается граф вычислений, который прогоняет и оптимизирует стратегию. Точнее, стратегией в данном случае является многослойный перцептрон с нелинейной функцией активации `tanh`. Для применения ожидаемого градиента во втором члене формулы (11.2) используется метод `Adam`.

Затем определяются функции `agent_op(o)` и `evaluation_on_noise(noise)`. Первая выполняет стратегию (потенциальное решение), чтобы получить действие в данном состоянии (наблюдении) `o`, а вторая оценивает новое потенциальное решение, полученное прибавлением возмущающего шума `noise` (который имеет такую же форму, как стратегия) к параметрам текущей стратегии.

Перейдем сразу к самой интересной части. Мы создаем новый сеанс, указывая, что он может рассчитывать не более чем на 4 процессора, и инициализируем глобальные переменные. Не переживайте, если на вашей машине нет четырех процессоров. Поскольку параметр `allow_soft_placement` равен `True`, TensorFlow согласится использовать те устройства, что имеются в наличии.

```
sess = tf.Session(config=tf.ConfigProto(device_count={'CPU': 4},
    allow_soft_placement=True))
sess.run(tf.global_variables_initializer())
```

Каждому исполнителю выделяется только один процессор. В определении графа вычислений указывается, на каком устройстве будут производиться вычисления. Например, если мы хотим, чтобы исполнитель работал только на

процессоре 0, то можем поместить граф внутри предложения `with`, определяющего, какое устройство использовать:

```
with tf.device("/cpu:0"):
    # граф, вычисляемый на процессоре 0
```

Но вернемся к нашей реализации. Мы можем крутиться в цикле бесконечно или, по крайней мере, пока исполнителю есть что делать. Это условие проверяется ниже в цикле `while`.

Отметим важную вещь. Мы выполняем много вычислений с весами нейронной сети, и было бы гораздо легче работать с сериализованными весами. Так, например, вместо того чтобы работать со списком формы `[8,32,32,4]`, проще иметь дело с одномерным массивом длины `8*32*32*4`. В TensorFlow есть функции, выполняющие преобразование одного в другое (если хотите знать, как это делается, загляните в полный код на GitHub).

Кроме того, перед входом в цикл `while` мы получаем форму сериализованного агента:

```
agent_flatten_shape = sess.run(agent_variables_flatten).shape
```

```
while True:
```

В первой части цикла `while` генерируются и оцениваются потенциальные решения. Они строятся путем прибавления нормально распределенного возмущения к весам, $\theta + \sigma \varepsilon$. Для этого мы каждый раз выбираем новое случайное начальное значение, которое определяет уникальную выборку шума σ из нормального распределения. Это ключевая часть алгоритма, потому что впоследствии другие исполнители должны будут реконструировать такой же шум по тому же начальному значению. Затем оцениваются два (поскольку мы применяем отражение шума) новых потомка, и результаты помещаются в очередь `output_queue`:

```
for _ in range(indiv_per_worker):
    seed = np.random.randint(1e7)

    with temp_seed(seed):
        sampled_noise = np.random.normal(size=agent_flatten_shape)

        pos_rew = evaluation_on_noise(sampled_noise)
        neg_rew = evaluation_on_noise(-sampled_noise)

        output_queue.put([[pos_rew, neg_rew], seed])
```

Отметим, что следующая строчка (она встречается в предыдущем фрагменте) – это просто идиома NumPy для локального задания начального значения генератора случайных чисел `seed`:

```
with temp_seed(seed):
    ..
```

Вне предложения `with` начальное значение, используемое для генерации случайных чисел, уже не будет равно `seed`.

Во второй части цикла `while` мы собираем все доходы и начальные значения, реконструируем по этим начальным значениям возмущающий шум, вычисляем оценку стохастического градиента по формуле (11.2) и оптимизируем стратегию. Очередь `params_queue` заполняется главным процессом, который был показан выше. Он помещает в нее нормированные ранги и начальные значения популяции, которые были сгенерированы исполнителями на первом этапе. Код приведен ниже.

```
batch_return, batch_seed = params_queue.get()
batch_noise = []

# реконструкция шума, использованного для генерации особей
for seed in batch_seed:
    with temp_seed(seed):
        sampled_noise = np.random.normal(size=agent_flatten_shape)

        batch_noise.append(sampled_noise)
        batch_noise.append(-sampled_noise)

# Вычисление оценки градиента по формуле (11.2)
vars_grads = np.zeros(agent_flatten_shape)
for n, r in zip(batch_noise, batch_return):
    vars_grads += n * r

vars_grads /= len(batch_noise) * std_noise

sess.run(apply_g, feed_dict={new_weights_ph: -vars_grads})
```

В последних строках этого фрагмента вычисляется оценка градиента, т. е. второй член формулы (11.2):

$$\frac{1}{n\sigma} \sum_{i=1}^n F_i \varepsilon_i. \quad (11.3)$$

Здесь F_i – нормированный ранг i -го кандидата, а ε_i – его возмущение.

Функция `apply_g` применяет вычисляемый по формуле (11.3) градиент `vars_grads` с помощью метода Adam. Отметим, что ей передается `-vars_grads`, потому что мы выполняем градиентный подъем, а не спуск.

С реализацией мы разобрались. Теперь нужно применить ее к среде и посмотреть на результаты.

ПРИМЕНЕНИЕ МАСШТАБИРУЕМОЙ ЭС К СРЕДЕ LUNARLANDER

Насколько хорошо масштабируемая версия эволюционной стратегии будет работать в окружающей среде? Посмотрим!

Напомним, что в главе 6 мы уже тестировали в среде LunarLander алгоритмы A2C и REINFORCE. В этой игре требуется посадить космический корабль на Луну с помощью непрерывных действий. Это среда умеренной трудности, поэтому мы решили ей воспользоваться для сравнения результатов A2C и ЭС.

В этой среде оптимальными оказались гиперпараметры, перечисленные в следующей таблице.

Гиперпараметр	Имя переменной	Значение
Размер нейронной сети	hidden_sizes	[32,32]
Число итераций обучения (поколений)	number_iter	200
Число исполнителей	num_workers	4
Скорость обучения Adam	lr	0.02
Число особей на одного исполнителя	indiv_per_worker	12
Стандартное отклонение	std_noise	0.05

Результаты показаны на графике (рис. 11.5). Сразу бросается в глаза, что кривая очень устойчивая и гладкая. Еще стоит отметить, что она достигает средней оценки 200 за 2,5–3 млн шагов. Сравнив это с результатами A2C (рис. 6.7), мы видим, что эволюционной стратегии потребовалось в 2–3 раза больше шагов, чем A2C и REINFORCE.

Как показано в оригинальной статье, применение массового распараллеливания (не менее сотен процессоров) позволяет выйти на хорошую стратегию всего за несколько минут. К сожалению, мы не располагаем такими вычислительными ресурсами. Но если вам повезло больше, то можете попробовать сами.

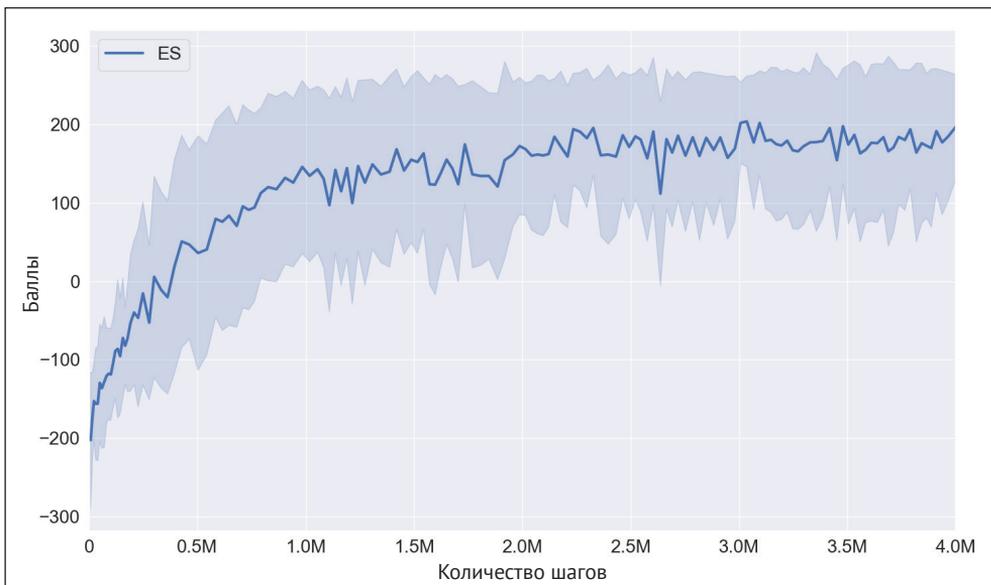


Рис. 11.5 ❖ Качество масштабируемой эволюционной стратегии

В целом результаты выглядят отлично и показывают, что ЭС – достойное решение для задач с очень длинным горизонтом и очень редким вознаграждением.

РЕЗЮМЕ

В этой главе мы рассказали об эволюционных алгоритмах – новом классе методов черного ящика, нашедших опору в идеях биологической эволюции и применимых к задачам ОП. ЭА подходят к проблеме иначе, чем в обучении с подкреплением. Мы видели, что многие свойства, которые следует учитывать при проектировании алгоритмов обучения с подкреплением, не имеют места в эволюционных методах. Различаются как сами методы оптимизации, так и базовые предположения. Например, поскольку ЭА – методы черного ящика, не накладывается никаких ограничений на оптимизируемую функцию, она может быть и недифференцируемой, что в ОП запрещено. Как было показано в этой главе, у ЭА много и других достоинств, но есть и многочисленные недостатки.

Далее мы рассмотрели два класса эволюционных алгоритмов: генетические алгоритмы и эволюционные стратегии. Генетические алгоритмы сложнее, потому что создают потомка двух родителей, применяя скрещивание и мутацию. Метод эволюционной стратегии отбирает лучших особей из популяции, созданной применением мутации к предыдущему поколению. Простота ЭС – один из ключевых элементов, благодаря которому алгоритм допускает массовое распараллеливание на тысячи исполнителей. Эта масштабируемость была продемонстрирована в статье компании OpenAI, доказавшей способность ЭС работать в сложной окружающей среде с качеством, не уступающим алгоритмам ОП.

Чтобы практически познакомиться с эволюционными алгоритмами, мы реализовали масштабируемую эволюционную стратегию, следуя идеям из вышеупомянутой статьи. Кроме того, мы протестировали ее в среде LunarLander и убедились, что ЭС может справиться с этой задачей, продемонстрировав высокое качество. Но для этого ему понадобилось в два-три раза больше шагов, чем алгоритмам AC и REINFORCE. Это и есть главный недостаток метода эволюционной стратегии: ему нужно очень много опыта. Но благодаря способности к линейному масштабированию при наличии достаточных вычислительных ресурсов задачу удастся решить во много раз быстрее, чем с помощью алгоритмов обучения с подкреплением.

В следующей главе мы вернемся к обучению с подкреплением и поговорим о дилемме исследования–использования. Мы узнаем, что это такое и почему так важно в онлайн-алгоритмах. Затем мы разработаем метаалгоритм ESBAS, который выбирает алгоритм, наиболее подходящий для конкретной ситуации.

Вопросы

1. Назовите две альтернативы обучению с подкреплением для решения задач последовательного принятия решения.
2. С помощью каких процессов порождаются новые особи в эволюционных алгоритмах?

3. Что послужило источником вдохновения для эволюционных алгоритмов, в т. ч. для генетических алгоритмов?
4. Как в алгоритме CMA-ES эволюционируют стратегии?
5. Назовите одно достоинство и один недостаток эволюционных стратегий.
6. Какой описанный в статье «Evolution Strategies as a Scalable Alternative to Reinforcement Learning» прием позволяет уменьшить дисперсию?

Для дальнейшего чтения

- Оригинальная статья компании OpenAI, в которой предложена масштабируемая версия ЭС, называется «Evolution Strategies as a Scalable Alternative to Reinforcement Learning» и опубликована по адресу <https://arxiv.org/pdf/1703.03864.pdf>.
- Статья «Evolving Neural Networks through Augmenting Topologies», посвященная алгоритму NEAT, находится по адресу <http://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf>.

Глава 12

Разработка алгоритма ESBAS

Сейчас вы уже знаете, как подходить к задачам ОП систематически и без суеты. Вы можете спроектировать и реализовать алгоритм ОП для решения конкретной задачи, получая максимальную информацию от окружающей среды. А в двух последних главах вы узнали и про алгоритмы, выходящие за рамки ОП, но позволяющие решать задачи такого же типа.

В начале этой главы мы опишем дилемму, с которой уже не раз встречались в предыдущих главах: дилемму исследования–использования. Мы рассказывали о потенциальных подходах к ее решению (например, об ε -жадной стратегии), но теперь хотим представить всесторонний взгляд на эту проблему и кратко описать алгоритмы ее решения. Многие из них, например алгоритм **верхней доверительной границы (ВДГ)** (upper confidence bound – UCB), сложнее и лучше простых эвристик, которыми мы пользовались до сих пор, в т. ч. ε -жадной стратегии. Мы проиллюстрируем эти стратегии на классической задаче о многоруком бандите. Хотя это простая табличная игра, мы воспользуемся ей как отправной точкой, а затем покажем, как подобные стратегии можно применить к более сложным нетабличным задачам.

В этом введении в дилемму исследования–использования мы приведем краткий обзор основных методов, которые применяются во многих современных алгоритмах ОП для обучения агентов в окружающих средах с очень трудным исследованием. Мы также расскажем о применимости этой дилеммы к решению задач других типов. И в качестве вишенки на торте разработаем метаалгоритм ESBAS (epochal stochastic bandit algorithm selection), который решает проблему онлайн-выбора алгоритма в контексте ОП. При этом используются идеи и подходы, выработанные для задачи о многоруком бандите и направленные на выбор наилучшего алгоритма ОП, который максимизирует ожидаемый доход в каждом эпизоде.

В этой главе рассматриваются следующие вопросы:

- исследование и использование;
- подходы к исследованию;
- алгоритм ESBAS.

ИССЛЕДОВАНИЕ И ИСПОЛЬЗОВАНИЕ

Дилемма исследования–использования встречается во многих важных предметных областях, а вовсе не только в контексте ОП. По существу, это поиск ответа на вопрос, что лучше: использовать оптимальное из уже известных решений или попробовать найти что-то новое. Допустим, вы покупаете новую книгу. Можно выбрать какую-нибудь книгу любимого автора или книгу того же жанра из предложенных сайтом Амазон. В первом случае вы точно знаете, что получите, а во втором неизвестно, чего ожидать. Зато во втором случае может повезти, и тогда вы будете наслаждаться чтением книги, которая даже лучше, чем все, что написал ваш любимый автор.

Этот конфликт между использованием того, что вы уже знаете, и несколько рискованным исследованием новых возможностей сплошь и рядом встречается в обучении с подкреплением. Иногда агенту нужно пожертвовать краткосрочным вознаграждением и исследовать новую область, чтобы в будущем получить более высокую награду.

Вряд ли для вас в этом есть что-то новое. Мы столкнулись с данной проблемой, еще когда разрабатывали свой первый алгоритм ОП. Но до сих пор мы применяли простую эвристику, например ε -жадную или стохастическую стратегию, решая, исследовать или использовать. Эмпирически такие стратегии работают очень хорошо, но существуют и другие методы, позволяющие достичь теоретически оптимального качества.

В этой главе мы для начала объясним смысл дилеммы исследование–использование, начиная с азов, а затем познакомимся с алгоритмами исследования, достигающими почти оптимального качества в табличных задачах. Мы также покажем, как те же самые подходы можно адаптировать к более сложным нетабличным задачам.



Рис. 12.1 ❖ Снимок экрана игры Montezuma's Revenge

Для любого алгоритма ОП одной из самых трудных игр Atari является Montezuma's Revenge, показанная на рис. 12.1. Цель игры – набирать очки, собирая алмазы и убивая врагов. Главный персонаж должен получить все ключи, чтобы обойти все комнаты в лабиринте, собирая попутно предметы, необходимые для передвижения и избегая препятствий. Разреженное вознаграждение, длинный горизонт и частичные вознаграждения, не коррелирующие с окончанием игры, – все это делает игру очень сложной для любого алгоритма ОП. Благодаря этим характеристикам Montezuma's Revenge – одна из лучших окружающих сред для тестирования алгоритмов исследования.

Начнем с азов, чтобы составить полное представление об этом вопросе.

Задача о многоруком бандите

Задача о многоруком бандите – классическая проблема ОП, используемая для иллюстрации дилеммы исследования–использования. Существо этой дилеммы в том, что агент должен делать выбор из фиксированного множества ресурсов, стремясь максимизировать ожидаемое вознаграждение. Название «многорукий бандит» происходит от «однорукого бандита» – так называют игровые автоматы, которые выдают выигрыш случайно, причем распределение вероятностей у каждого свое. Игрок должен выработать наилучшую стратегию, чтобы получить максимальное вознаграждение в долгосрочной перспективе.

Эта ситуация показана на рис. 12.2. Здесь игрок (привидение) должен выбрать один из пяти игровых автоматов с разными и неизвестными вероятностями вознаграждения, чтобы в итоге выиграть максимальную сумму.



Рис. 12.2 ❖ Пример задачи о пятируком бандите

Вопрос: как задача о многоруком бандите связана с более интересными задачами типа Montezuma's Revenge? Да просто во всех них возникает одна и та же проблема: будет ли в долгосрочной перспективе вознаграждение больше, если пробовать новое поведение (потянуть другой рычаг) или если продолжать делать то, что давало наилучший результат до сих пор (тянуть самый изученный рычаг)? Но основное различие между многоруким бандитом и Montezuma's Revenge в том, что во втором случае состояние агента каждый раз меняется. В задаче о многоруком бандите есть всего одно состояние и никакой последовательной структуры, т. е. прошлые действия не влияют на будущее.

Ну и как же найти правильный баланс между исследованием и использованием в задаче о многоруком бандите?

Подходы к исследованию

Задачу о многоруком бандите, да и любую задачу исследования можно решить, применяя либо случайную стратегию, либо какую-то более осмысленную. Самый известный алгоритм, относящийся к первой категории, называется ε -жадным, а стратегии оптимистического исследования, например ВДГ, и апостериорного исследования, например выборка Томпсона, относятся ко второй. В этом разделе мы рассмотрим две стратегии: ε -жадную и ВДГ.

Все сводится к выбору баланса между риском и наградой. Но как измерить качество алгоритма исследования? С помощью *сожаления*. Сожаление (regret) определяется как возможность, потерянная на одном шаге, т. е. сожаление L в момент t равно

$$L_t = V^* - Q(a_t),$$

где V^* – оптимальное значение, а $Q(a_t)$ – ценность действия a_t .

Таким образом, для поиска оптимального компромисса между исследованием и использованием требуется минимизировать полное сожаление по всем действиям:

$$L = \sum_i (V^* - Q(a_i)).$$

Отметим, что минимизация полного сожаления равносильна максимизации полного вознаграждения. Мы воспользуемся идеей сожаления, чтобы показать, как работают алгоритмы исследования.

ε -жадная стратегия

Мы уже использовали идеи ε -жадной стратегии при реализации исследования в таких алгоритмах, как Q-обучение и DQN. Это очень простой подход, который тем не менее дает отличные результаты даже в нетривиальных задачах. Именно поэтому он так часто используется во многих алгоритмах глубокого обучения.

Напомним, что ε -жадная стратегия состоит в том, чтобы почти всегда выбирать наилучшее действие, но время от времени брать случайное. Вероятность, что будет выбрано случайное действие, – параметр алгоритма, принимающий значение от 0 до 1. То есть с вероятностью ε алгоритм будет использовать уже известное наилучшее действие, а с вероятностью $1 - \varepsilon$ исследовать альтернативы, применяя случайный поиск.

В задаче о многоруком бандите ценности действий оцениваются, исходя из прошлого опыта, путем усреднения вознаграждения, полученного при выборе этих действий:

$$Q_t(a) = \frac{1}{N_t(a)} \sum_i r_t \mathbb{1}[a_t = a].$$

Здесь $N_t(a)$ – сколько раз было выбрано действие a , а $\mathbb{1}$ – булева величина, показывающая, было ли выбрано действие a в момент t . Игрок действует в со-

ответствии с ε -жадным алгоритмом: исследует, выбирая случайное действие, или использует, выбирая действие с максимальной ценностью.

Недостаток ε -жадной стратегии в том, что ожидаемое сожаление линейно. Но, по закону больших чисел, оптимальное ожидаемое полное сожаление должно логарифмически зависеть от числа временных шагов. Поэтому ε -жадная стратегия не оптимальна.

Простой способ приблизиться к оптимальности – сделать ε убывающим со временем. Тогда вес исследования будет стремиться к нулю, и в конечном итоге будут выбираться только жадные действия. И действительно, в алгоритмах глубокого ОП ε -жадность почти всегда сочетается с линейным или экспоненциальным затуханием ε .

Однако и начальное значение ε , и скорость его затухания трудно подобрать, поэтому разработаны другие стратегии оптимального решения задачи о многооруком бандите.

Алгоритм UCSB

Алгоритм UCSB связан с так называемым принципом оптимизма в условиях неопределенности – статистическим принципом, основанным на законе больших чисел. UCSB строит оптимистическую гипотезу, основанную на выборочном среднем вознаграждении и на оценке верхней доверительной границы вознаграждения. Оптимистическая гипотеза определяет ожидаемую выплату при каждом действии с учетом неопределенности действий. Таким образом, UCSB всегда выбирает действие с более высоким потенциальным вознаграждением, пытаясь найти баланс между риском и наградой. Если оказывается, что у других действий оптимистическая оценка меньше, чем у текущего, то алгоритм переключается на другое действие.

Точнее, UCSB хранит среднее вознаграждение каждого действия $Q_t(a)$ и верхнюю доверительную границу $U_t(a)$ каждого действия. Затем алгоритм выбирает тот рычаг, для которого принимает максимум следующее выражение:

$$a_t = \operatorname{argmax}_{a \in A} Q_t(a) + U_t(a). \quad (12.1)$$

В этой формуле роль U – дополнить среднее вознаграждение членом, учитывающим недовершенство действия.

UCSB1

Алгоритм UCSB1 принадлежит тому же семейству, что UCSB, а его специфика заключается в выборе U .

В UCSB1 запоминается, сколько раз было выбрано действие a (величина $N_t(a)$), а также общее число выбранных действий T . Тогда верхняя доверительная граница $U_t(a)$ вычисляется по формуле

$$U_t(a) = c \sqrt{\frac{\ln T}{N_t(a)}}. \quad (12.2)$$

Таким образом, неопределенность действия зависит от того, сколько раз оно выбиралось. Это имеет смысл, поскольку, по закону больших чисел, при бес-

конечном количестве испытаний математическое ожидание точно известно. С другой стороны, если действие опробовано всего несколько раз, то мы не можем быть уверены в величине ожидаемого вознаграждения, и лишь дополнительный опыт позволит сказать, хорошее это действие или плохое. Следовательно, мы подталкиваем к исследованию действий, которые выбирались всего несколько раз и потому имеют высокую неопределенность. Смысл в том, что если $N_t(a)$ мало, т. е. действие выбиралось лишь изредка, то $U_t(a)$ будет велико, что дает оценку с высокой степенью неопределенности. Но если $N_t(a)$ велико, то $U_t(a)$ мало, и оценка будет более точной. И далее действие a выбирается, только если среднее вознаграждение для него велико.

Основное преимущество UCB перед ε -жадной стратегией обусловлено именно подсчетом действий. Действительно, задачу о многоруком бандите легко решить этим методом, если для каждого действия хранить счетчик и среднее вознаграждение. Эти две величины можно подставить в формулы (12.1) и (12.2) и узнать, какое действие лучше всего предпринять в момент t :

$$a_t = \operatorname{argmax}_{a \in A} Q_t(a) + c \sqrt{\frac{\ln T}{N_t(a)}}. \quad (12.3)$$

UCB – очень мощный метод исследования, он достигает логарифмического ожидаемого полного сожаления в задаче о многоруком бандите, т. е. ведет себя оптимально. Отметим, что ε -жадное исследование тоже может достичь логарифмического сожаления, но для этого его необходимо тщательно спроектировать и настроить экспоненциальное затухание, так что достичь желаемого баланса труднее.



Существуют и другие варианты алгоритма UCB: UCB2, UCB-Tuned и KL-UCB.

Сложность исследования

Мы видели, что UCB и в особенности UCB1 – сравнительно простые алгоритмы, которые тем не менее могут уменьшить полное сожаление и достичь оптимальной сходимости в задаче о многоруком бандите. Однако сама задача несложная и не имеет внутреннего состояния.

А как UCB ведет себя в более сложных задачах? Чтобы ответить на этот вопрос, упростим классификацию, оставив всего три большие категории:

- **Задачи без внутреннего состояния.** Примером может служить задача о многоруком бандите. В таких случаях для исследования можно применить такие сравнительно развитые алгоритмы, как UCB1.
- **Табличные задачи малого и среднего размера.** Как правило, для исследования все еще применимы более-менее продвинутые механизмы, но в некоторых случаях выигрыш мал и дополнительная сложность не оправдана.
- **Крупные нетабличные задачи.** Это уже более сложная окружающая среда. Тут точка зрения еще не устоялась, ученые все еще активно ищут наилучшую стратегию исследования. Причина в том, что по мере увеличения сложности оптимальные методы типа UCB становятся практиче-

ски неосуществимыми. Так, UCS непригоден для задач с непрерывным состоянием. Однако не стоит отказываться сразу от всего, мы можем взять алгоритмы исследования, изученные на примере задачи о много-руком бандите, за отправную точку. Существует немало подходов, которые аппроксимируют оптимальные методы исследования и неплохо работают в непрерывных окружающих средах. В частности, подходы на основе счетчиков, каковым является UCS, адаптированы к задачам с бесконечным числом состояний, так что у похожих состояний счетчики тоже похожи. Такого рода алгоритмы также способны добиться значительно-го улучшения в очень трудных средах, в т. ч. в игре Montezuma's Revenge. И все же в большинстве контекстов ОП дополнительная сложность, прису-щая этим подходам, не оправдана, а более простые рандомизирован-ные стратегии типа ϵ -жадной дают приемлемые результаты.



Отметим также, что, помимо рассмотренного нами подхода к исследованию на основе подсчета (UCB1), есть два других нетривиальных способа организовать исследование, при которых достигается оптимальное сожаление. Первый, называемый апостериорной выборкой (примером может служить выборка Томпсона), основан на апостериорном распределении, а второй – информационный выигрыш – опирается на внутреннюю меру неопределенности, основанную на оценке энтропии.

Алгоритм ESBAS

Стратегии исследования применяются в обучении с подкреплением главным образом для того, чтобы помочь агенту в исследовании окружающей среды. Мы видели это на примере ϵ -жадной стратегии в алгоритме DQN и когда включали дополнительный шум в стратегию в других алгоритмах. Но существуют и иные способы использования стратегий исследования. Чтобы лучше усвоить уже описанную концепцию исследования и познакомиться с альтернативными применениями таких алгоритмов, мы представим и реализуем алгоритм ESBAS. Он впервые был описан в статье «Reinforcement Learning Algorithm Selection».

ESBAS – это метаалгоритм онлайн-выбора алгоритма (algorithm selection – AS) в контексте обучения с подкреплением. В нем используются методы исследования, чтобы решить, какой алгоритм лучше использовать при прохождении траектории, чтобы максимизировать ожидаемый доход.

Сначала мы объясним, что такое выбор алгоритма и как это используется в машинном обучении и в обучении с подкреплением. Затем обратимся непосредственно к ESBAS, детально опишем его внутреннее устройство и приведем псевдокод. Наконец, мы реализуем ESBAS и протестируем его в окружающей среде Acrobot.

Что такое выбор алгоритма

Чтобы лучше понять, что делает ESBAS, разберемся сначала в том, что такое выбор алгоритма (AS). Обычно разрабатывается конкретный фиксированный алгоритм, который обучается решению некоторой задачи. Проблема в том, что

даже если набор данных со временем изменяется, а алгоритм переобучен, или если другой алгоритм при каких-то условиях работает лучше, то изменить мы уже ничего не можем. Выбранный однажды алгоритм останется с нами навсегда. Идея выбора алгоритма позволяет преодолеть это затруднение.

AS – открытая проблема в области машинного обучения. Задача заключается в том, чтобы спроектировать метаалгоритм, который всегда выбирает из пула доступных вариантов, называемого портфелем, наилучший алгоритм для удовлетворения текущих потребностей. Это изображено на рис. 12.3. Идея AS основана на предположении, что одни из имеющихся в портфеле алгоритмов лучше в одной области пространства задачи, а другие – в другой. Поэтому важно, чтобы возможности алгоритмов дополняли друг друга.

Например, на рис. 12.3 метаалгоритм выбирает, какой из присутствующих в портфеле алгоритмов (или агентов), например PPO или TD3, должен взаимодействовать со средой в данный момент. Эти алгоритмы не являются взаимодополнительными, но у каждого есть свои сильные стороны, позволяющие метаалгоритму предпочесть тот или иной в конкретной ситуации.

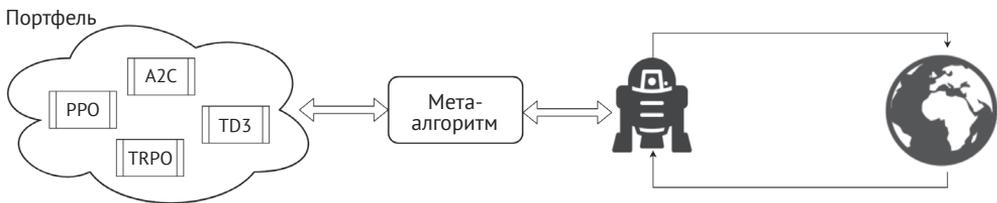


Рис. 12.3 ❖ Метод выбора алгоритма для ОП

Например, если требуется спроектировать беспилотный автомобиль, способный передвигаться по любой местности, то, наверное, будет полезно обучить разные алгоритмы для передвижения по дороге, в пустыне и по льду. Тогда AS мог бы выбрать, какой из них применить в каждой ситуации. Скажем, AS мог бы решить, что в дождливый день лучше покажет себя стратегия, обученная езде по льду.

В ОП стратегия изменяется с очень высокой частотой, а набор данных постоянно растет со временем. Это значит, что между отправной точкой, когда агент находился в зачаточном состоянии, и хорошо обученным агентом существует колоссальная разница в плане оптимального размера нейронной сети и скорости обучения. Например, в начале обучения его скорость может быть высока, а по мере накопления опыта – снижаться. В силу этих особенностей ОП оказывается весьма интересной экспериментальной площадкой для выбора алгоритма, чем мы и воспользуемся.

ESBAS изнутри

В оригинальной статье, посвященной ESBAS, алгоритм тестировался в пакетном и онлайн-режимах. Но мы займемся только первым случаем. Оба варианта очень похожи, и если вас интересует чисто онлайн-вариант, обратитесь к самой статье. В онлайн-режиме ESBAS называется по-другому – **скользя-**

щий стохастический бандит (sliding stochastic bandit – **SSBAS**), поскольку обучается он на скользящем окне, которое содержит последние решения о выборе. Но начнем с основ.

Первое, что нужно сказать о ESBAS, – это то, что он основан на стратегии UCB1 и использует «бандитский» подход к выбору алгоритма с разделенной стратегией из фиксированного портфеля. Конкретно, ESBAS можно разбить на три основные части.

1. Цикл по большому количеству эпох экспоненциально увеличивающегося размера. Внутри каждой эпохи сначала обновляются все алгоритмы с разделенной стратегией, имеющиеся в портфеле. Это делается на данных, собранных к текущему моменту (в первой эпохе набор данных будет пустым). Кроме того, метаалгоритм сбрасывается в начальное состояние.
2. Затем на протяжении каждой эпохи метаалгоритм вычисляет оптимистическую гипотезу по формуле (12.3), чтобы выбрать тот алгоритм с разделенной стратегией (из портфеля), который получит управление на следующей траектории, – это алгоритм, минимизирующий полное сожаление. После этого прогоняется траектория под управлением этого алгоритма. Попутно собирается информация обо всех переходах на траектории и добавляется в набор данных, который впоследствии будет использован алгоритмами с разделенной стратегией для обучения стратегий.
3. По завершении траектории метаалгоритм обновляет средний доход этого конкретного алгоритма с разделенной стратегией, присвоив ему доход, полученный во взаимодействии со средой, и увеличивает счетчик действий. Средний доход и счетчик действий используются для вычисления верхней доверительной границы по формуле (12.2), как в UCB1. Эти величины служат для выбора следующего алгоритма с разделенной стратегией, который будет управлять прохождением следующей траектории.

Для лучшего понимания ниже приведен псевдокод ESBAS.

 ESBAS

Инициализировать стратегию π^a для каждого алгоритма a в портфеле P
 Инициализировать пустой набор данных D

for $\beta=1..M$ **do**

for a in P **do**

 Обучить стратегию π^a на D с помощью алгоритма a

 Инициализировать переменные AS: $n \leftarrow \emptyset$ и для каждого $a \in P$: $n^a \leftarrow \emptyset$, $x^a \leftarrow \emptyset$

for $t=2^\beta..2^{\beta+1}$ **do**

 > Выбрать наилучший алгоритм в смысле UCB1

$$a^{\max} = \operatorname{argmax}_{a \in P} \left(x^a + \sqrt{\frac{\xi \ln(n)}{n^a}} \right)$$

 Пройти траекторию τ , применяя стратегию $\pi^{a^{\max}}$, и добавить переходы в D

 > Обновить средний доход и счетчик a^{\max}

$$\begin{aligned} x^{\max} &\leftarrow \frac{n^{\max} x^{\max} + R(\tau)}{n^{\max} + 1} & (12.4) \\ n^{\max} &\leftarrow n^{\max} + 1 \\ n &\leftarrow n + 1 \end{aligned}$$

Здесь ξ – гиперпараметр, $R(\tau)$ – доход, полученный на траектории τ , n^a – счетчик алгоритма a , x^a – средний доход.

Как объясняется в статье, онлайн-овый AS решает практические проблемы, унаследованные от алгоритмов ОП.

1. **Выборочная эффективность.** Диверсификация стратегий обеспечивает дополнительный источник информации, благодаря которому ESBAS оказывается выборочно эффективным. Кроме того, он сочетает в себе свойства курсового обучения (curriculum learning) и ансамблевого обучения.
2. **Робастность.** Диверсификация портфеля дает защиту от плохих алгоритмов.
3. **Сходимость.** ESBAS гарантирует минимизацию сожаления.
4. **Курсовое обучение.** AS способен обеспечить некое подобие курсовой стратегии, например выбирая более простые, плоские модели в начале и глубокие модели ближе к концу.

Реализация

Реализовать ESBAS легко, поскольку нужно добавить всего несколько компонентов. Самая существенная часть – определение и оптимизация алгоритмов с разделенной стратегией, входящих в портфель. В этом отношении ESBAS никак не ограничивает состав алгоритмов. В оригинальной статье используются Q-обучение и DQN. Мы решили оставить DQN, чтобы иметь в арсенале алгоритм, способный справиться с более сложными задачами, который можно было бы использовать в таких средах, как пространство RGB-состояний. Алгоритм DQN подробно рассмотрен в главе 5, для ESBAS мы взяли ту же самую реализацию.

Последнее, о чем нужно сказать, прежде чем переходить к реализации, – состав портфеля. Мы создали портфель, диверсифицированный с точки зрения архитектуры нейронной сети, но вы можете попробовать и другие комбинации. Например, почему бы не поместить в портфель алгоритмы DQN с разными скоростями обучения?

Код состоит из следующих частей:

- класс `DQN_optimization` строит граф вычислений и оптимизирует стратегию с помощью алгоритма DQN;
- класс `UCB1` содержит реализацию алгоритма UCB1;
- функция `ESBAS` реализует главный конвейер алгоритма ESBAS.

В тексте показана реализация последних двух частей, но в репозитории книги на GitHub по адресу <https://github.com/PacktPublishing/Reinforcement-Learning-Algorithms-with-Python> имеется полный код.

Начнем с функции `ESBAS(...)`. Помимо гиперпараметров DQN, ей передается еще один аргумент x_i , представляющий гиперпараметр ξ . Структура функции

ESBAS повторяет приведенный выше псевдокод, поэтому мы не будем на ней долго задерживаться.

Перечислив все аргументы функции, мы переводим граф TensorFlow в начальное состояние и создаем две окружающие среды Gym (для обучения и для тестирования). Затем строится портфель, для этого мы создаем по одному объекту `DQN_optimization` для каждого размера нейронной сети и помещаем их в список.

```
def ESBAS(env_name, hidden_sizes=[32], lr=1e-2, num_epochs=2000,
         buffer_size=100000, discount=0.99, render_cycle=100,
         update_target_net=1000, batch_size=64, update_freq=4, min_buffer_size=5000,
         test_frequency=20, start_explor=1, end_explor=0.1, explor_steps=100000,
         xi=16000):
    tf.reset_default_graph()

    env = gym.make(env_name)
    env_test = gym.wrappers.Monitor(gym.make(env_name),
    "VIDEOS/TEST_VIDEOS"+env_name+str(current_milli_time()),force=True,
    video_callable=lambda x: x%20==0)
    dqns = []
    for l in hidden_sizes:
        dqns.append(DQN_optimization(env.observation_space.shape,
    env.action_space.n, l, lr, discount))
```

Теперь определим вспомогательную функцию `DQNs_update`, которая обучает стратегии способом, принятым в DQN. Напомним, что в портфеле имеются только алгоритмы DQN, различающиеся лишь размером нейронной сети. Для оптимизации вызываются методы `optimize` и `update_target_network` класса `DQN_optimization`:

```
def DQNs_update(step_counter):
    if len(buffer) > min_buffer_size and (step_counter % update_freq == 0):
        mb_obs, mb_rew, mb_act, mb_obs2, mb_done = buffer.sample_minibatch(batch_size)
        for dqn in dqns:
            dqn.optimize(mb_obs, mb_rew, mb_act, mb_obs2, mb_done)
    if len(buffer) > min_buffer_size and (step_counter % update_target_net == 0):
        for dqn in dqns:
            dqn.update_target_network()
```

Как всегда, необходимо инициализировать некоторые переменные: привести среду в исходное состояние, создать объект `ExperienceBuffer` (этот класс уже использовался в предыдущих главах) и задать коэффициент затухания исследования:

```
step_count = 0
batch_rew = []
episode = 0
beta = 1
buffer = ExperienceBuffer(buffer_size)
obs = env.reset()
eps = start_explor
eps_decay = (start_explor - end_explor) / explor_steps
```

И вот теперь можно войти в цикл по эпохам. Как следует из приведенного выше псевдокода, на каждой эпохе происходит следующее:

1. Стратегии обучаются на буфере воспроизведения.
2. Траектории проходятся в соответствии со стратегией, выбранной UCB1.

Для выполнения первого действия вызывается функция `DQNs_update` для каждого элемента эпохи (длина которой экспоненциально возрастает):

```
for ep in range(num_epochs):
    # обучение стратегий
    for i in range(2**(beta-1), 2**beta):
        DQNs_update(i)
```

Что касается второго действия, то перед прохождением траекторий создается и инициализируется новый объект класса UCB1. Затем в цикле `while` обходятся эпизоды, в каждом из которых объект UCB1 выбирает, какой алгоритм будет управлять прохождением следующей траектории. На протяжении траектории действия выбираются согласно стратегии `dqns[best_dqn]`:

```
ucb1 = UCB1(dqns, xi)
list_bests = []
beta += 1
ep_rew = []

while step_count < 2**beta:
    best_dqn = ucb1.choose_algorithm()
    list_bests.append(best_dqn)

    g_rew = 0
    done = False

    while not done:
        # Затухание эpsilon
        if eps > end_explor:
            eps -= eps_decay

        act = eps_greedy(np.squeeze(dqns[best_dqn].act(obs)), eps=eps)
        obs2, rew, done, _ = env.step(act)
        buffer.add(obs, rew, act, obs2, done)

        obs = obs2
        g_rew += rew
        step_count += 1
```

По завершении каждой траектории `ucb1` обновляется с учетом полученного на ней дохода. Затем окружающая среда сбрасывается, и доход на завершенной траектории добавляется в список, где хранятся все доходы.

```
ucb1.update(best_dqn, g_rew)
obs = env.reset()
ep_rew.append(g_rew)
g_rew = 0
episode += 1
```

На этом функция ESBAS заканчивается.

Класс UCB1 включает конструктор, который инициализирует атрибуты, необходимые для вычислений по формуле (12.3), метод `choose_algorithm()`, который

возвращает наилучший на текущий момент алгоритм из числа имеющихся в портфеле в соответствии с формулой (12.3), и метод `update(idx_algo, traj_return)`, который обновляет средний доход алгоритма `idx_algo` с учетом последнего полученного дохода, как в формуле (12.4).

```
class UCB1:
    def __init__(self, algos, epsilon):
        self.n = 0
        self.epsilon = epsilon
        self.algos = algos
        self.nk = np.zeros(len(algos))
        self.xk = np.zeros(len(algos))

    def choose_algorithm(self):
        return np.argmax([self.xk[i] + np.sqrt(self.epsilon *
np.log(self.n) / self.nk[i]) for i in range(len(self.algos))])

    def update(self, idx_algo, traj_return):
        self.xk[idx_algo] = (self.nk[idx_algo] * self.xk[idx_algo] +
traj_return) / (self.nk[idx_algo] + 1)
        self.nk[idx_algo] += 1
        self.n += 1
```

Теперь мы можем протестировать программу в окружающей среде и посмотреть на результаты.

Тестирование в среде Acrobot

Для тестирования ESBAS мы воспользуемся еще одной окружающей средой Gym – `Acrobot-v1`. Как сказано в документации по OpenAI Gym, *система Acrobot включает два звена и два сочленения, и в действие приводится сочленение между звеньями. В начальном состоянии звенья свисают вниз, а цель заключается в том, чтобы поднять нижнее звено на заданную высоту*. На рис. 12.4 показаны положения акробота в течение короткой последовательности временных шагов от начала до конца.

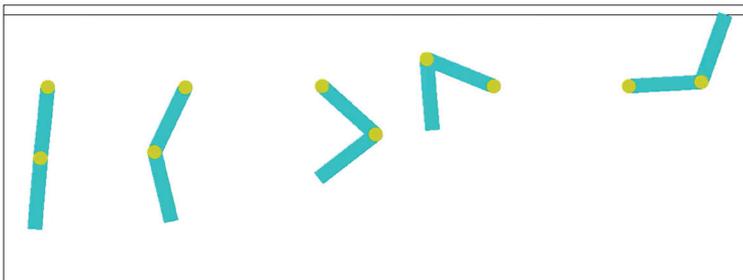


Рис. 12.4 ❖ Последовательность движений акробота

Портфель состоит из трех глубоких нейронных сетей разного размера: небольшой, содержащей всего один скрытый слой с 64 блоками; средней, содержащей два скрытых слоя с 16 блоками; и большой, содержащей два скрытых

слоя с 64 блоками. Гиперпараметр ξ выбран равным 0.25 (такое же значение, как в оригинальной статье).

Результаты

Результаты показаны на рис. 12.5. На графике представлены две кривые обучения ESBAS: с полным портфелем (содержащим все три вышеупомянутые нейронные сети) – синим цветом; и только с самой лучшей нейронной сетью (с двумя скрытыми слоями размера 64) – оранжевым цветом. Мы знаем, что ESBAS с портфелем, содержащим всего один алгоритм, не раскрывает весь потенциал метаалгоритма, но привели эту кривую, чтобы было с чем сравнивать. График говорит сам за себя – синяя кривая всегда расположена выше оранжевой, и это доказывает, что ESBAS действительно выбирает лучший из возможных вариантов. Необычная форма кривых объясняется тем, что алгоритмы DQN обучаются в офлайн-режиме.

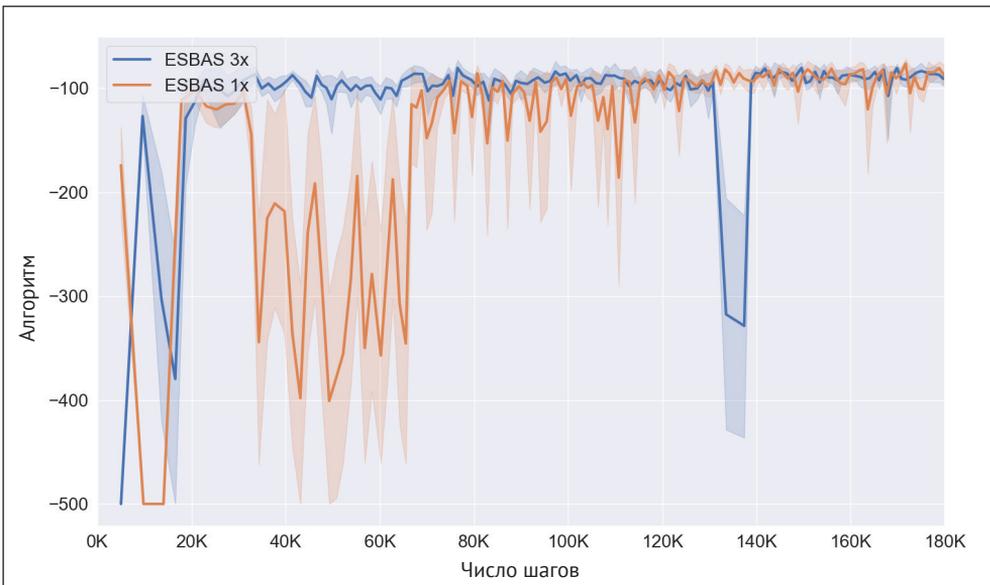


Рис. 12.5 ❖ Результаты ESBAS с портфелем, содержащим три алгоритма (синяя кривая) и всего один алгоритм (оранжевая кривая)

Пики в начале обучения, а затем после 20K, 65K и 131K шагов – это точки, в которых обучаются стратегии и метаалгоритм сбрасывается в исходное состояние.

Можно задаться вопросом, в какой момент ESBAS начинает отдавать предпочтение какому-то алгоритму. Ответ дает график на рис. 12.6. На нем малой нейронной сети соответствует значение 0, средней – значение 1, а большой – значение 2. Точки показывают, какой алгоритм выбирался на каждой траектории. Мы видим, что в самом начале предпочтительной являлась большая сеть, но она сразу же сменилась средней, а затем малой. После 64K шагов метаалгоритм вернулся к большой нейронной сети.

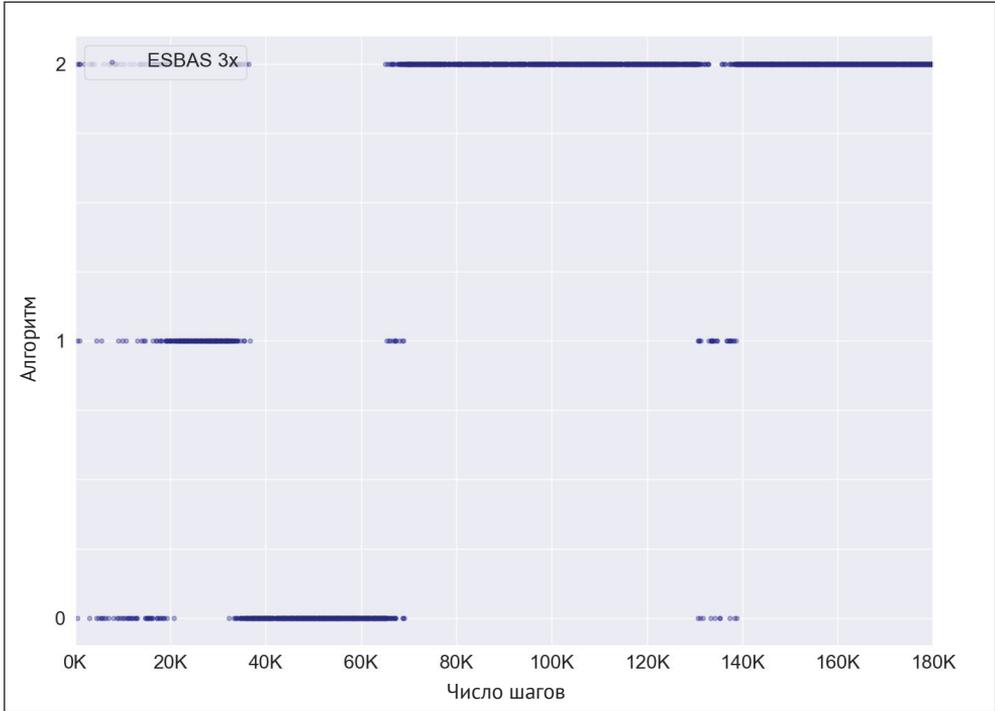


Рис. 12.6 ❖ График показывает предпочтения метаалгоритма

По этому графику видно, что обе версии ESBAS сходятся к одним и тем же значениям, но скорость сильно различается. Версия, раскрывающая весь потенциал выбора алгоритма (когда в портфеле три алгоритма), сходится гораздо быстрее. А к одним и тем же значениям обе сходятся, потому что в долгосрочной перспективе лучшей нейронной сетью оказалась та, что находилась в портфеле второй версии (сеть с двумя скрытыми слоями размера 64).

РЕЗЮМЕ

В этой главе мы рассмотрели дилемму исследования–использования. Мы уже затрагивали ее в предыдущих главах, но лишь поверхностно, ограничиваясь простыми стратегиями. Здесь же мы изучили ее более глубоко, начав с классической задачи о многоруком бандите. Мы видели, что более развитые алгоритмы с подсчетом, в частности UCB, могут достичь оптимального качества – с ожидаемым логарифмическим сожалением.

Затем мы применили алгоритмы исследования к выбору алгоритма (AS). Это интересное применение, поскольку в задачу метаалгоритма входит выбор алгоритма, который дает наилучшие результаты в конкретной задаче. AS находит применение и в обучении с подкреплением. Например, AS можно использовать, чтобы выбрать для прохождения следующей траектории наилучшую стратегию из числа обученных различными алгоритмами в портфеле.

Именно это и делает алгоритм ESBAS. Он решает задачу онлайн-выбора алгоритма ОП с разделенной стратегией, применяя для этой цели метод UCB1. Мы изучили и реализовали ESBAS.

Теперь вы знаете все, что необходимо для разработки высококачественных алгоритмов ОП, способных находить баланс между исследованием и использованием. А в предыдущих главах вы узнали, какой алгоритм использовать в той или иной ситуации. Но мы еще не поговорили о некоторых передовых концепциях ОП. В следующей, последней, главе мы восполним это упущение и расскажем об обучении без учителя, внутренней мотивации, практических проблемах ОП и о том, как повысить робастность алгоритмов. Мы также покажем, как можно использовать перенос обучения для перехода от имитационного моделирования к реальности. И напоследок дадим несколько рекомендаций по обучению и отладке алгоритмов глубокого обучения с подкреплением.

Вопросы

1. В чем состоит дилемма исследования–использования?
2. Какие две стратегии исследования уже встречались в рассмотренных ранее алгоритмах ОП?
3. Что такое ВДГ (UCB)?
4. Какая задача труднее: игра Montezuma’s Revenge или задача о многоруком бандите?
5. Как в метаалгоритме ESBAS решается проблема онлайн-выбора алгоритма ОП?

Для дальнейшего чтения

- Более полный обзор проблематики, связанной с задачей о многоруком бандите, см. в работе «A Survey of Online Experiment Design with Stochastic Multi-Armed Bandit» (<https://arxiv.org/pdf/1510.00757.pdf>).
- В статье «Unifying Count-Based Exploration and Intrinsic Motivation» (<https://arxiv.org/pdf/1606.01868.pdf>) рассматривается вопрос о внутренней мотивации при игре в Montezuma’s Revenge.
- Оригинальная статья, посвященная ESBAS, находится по адресу <https://arxiv.org/pdf/1701.08810.pdf>.

Глава 13

Практические подходы к решению проблем ОП

В этой главе мы подведем итог некоторым идеям, лежащим в основе алгоритмов **глубокого обучения с подкреплением**, которые были рассмотрены в предыдущих главах, представим общую картину их применения и дадим рекомендации по выбору алгоритма, лучше всего подходящего для решения конкретной задачи. Мы также приведем некоторые советы по поводу разработки собственного алгоритма глубокого ОП. Мы расскажем, о чем нужно позаботиться в самом начале разработки, чтобы было легко экспериментировать, не тратя много времени на отладку. Также мы перечислим наиболее важные настраиваемые гиперпараметры и объясним, как организовать нормировку.

Затем обсудим основные проблемы, возникающие в этой области: устойчивость, эффективность и обобщаемость. Они знаменуют переход к более продвинутым приемам обучения с подкреплением, в т. ч. ОП без учителя и перенос обучения. Оба метода чрезвычайно важны для развертывания решений особо требовательных к ресурсам задач ОП, поскольку направлены на решение всех трех вышеупомянутых проблем.

Мы также посмотрим, как ОП применяется к реальным задачам и как алгоритмы ОП используются для преодоления разрыва между имитационными моделями и реальностью.

И в заключение этой главы и книги в целом мы обсудим будущее обучения с подкреплением с технической и социальной точек зрения.

В этой главе рассматриваются следующие вопросы:

- рекомендуемые практики глубокого ОП;
- проблемы глубокого ОП;
- передовые методы;
- ОП в реальном мире;
- будущее ОП и его влияние на общество.

РЕКОМЕНДУЕМЫЕ ПРАКТИКИ ГЛУБОКОГО ОП

В этой книге мы рассмотрели много разных алгоритмов обучения с подкреплением, одни из которых являются всего лишь модернизациями (например, TD3, A2C и т. п.), тогда как другие принципиально отличаются (например, TRPO

и DPG) и предлагают новый путь к достижению той же цели. Мы остановились также на алгоритмах оптимизации, не связанных с ОП, в частности на подражателем обучения и эволюционных стратегиях, применяемых для решения последовательных задач принятия решений. Из-за всего этого разнообразия можно впасть в ступор, не понимая, какой алгоритм лучше подходит для конкретной задачи. Если вам знакомо это чувство, не переживайте – мы сформулируем некоторые правила, на которые можно ориентироваться при выборе алгоритма.

Кроме того, если вы пытались реализовать некоторые из рассмотренных в книге алгоритмов, то, наверное, осознали, как трудно бывает сложить все кусочки воедино, чтобы алгоритм заработал правильно. Алгоритмы глубокого ОП снискали печальную известность из-за трудности обучения и отладки, а время их обучения очень велико. В результате процесс обучения оказывается медленным и изнурительным. По счастью, существует несколько стратегий, которые позволяют избежать мучений при разработке алгоритмов глубокого ОП. Но прежде чем описывать эти стратегии, рассмотрим вопрос о выборе подходящего алгоритма.

Выбор подходящего алгоритма

Основное, чем различаются различные типы алгоритмов ОП, – выборочная эффективность и время обучения.

❗ Под выборочной эффективностью мы понимаем количество взаимодействий с окружающей средой, которое должен совершить агент, чтобы обучиться решению задачи. Этот индикатор позволяет сравнивать эффективность разных алгоритмов в типичных окружающих средах.

Очевидно, что на выбор влияют и другие параметры, но обычно их влияние слабее. Просто для полноты картины перечислим их: доступность процессоров, в т. ч. графических, тип функции вознаграждения, масштабируемость, сложность алгоритма и окружающей среды.

При сравнении мы будем рассматривать алгоритмы черного ящика, не нуждающиеся в вычислении градиента, например эволюционные стратегии, алгоритмы ОП на основе модели типа DAgger и безмодельные алгоритмы. Среди последних будем различать алгоритмы градиента стратегии, например DDPG и TRPO, и алгоритмы на основе ценности, например DQN.

На рис. 13.1 показана выборочная эффективность всех четырех категорий алгоритмов (по возрастанию слева направо). Как видим, больше всего данных от окружающей среды требуют безградиентные методы, за ними идут методы градиента стратегии, методы на основе ценности и, наконец, основанные на модели методы, самые выборочно эффективные.

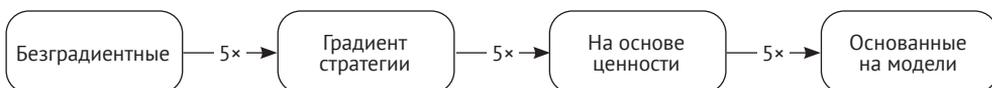


Рис. 13.1 ❖ Сравнение алгоритмов по выборочной эффективности (по возрастанию слева направо)

С другой стороны, для времени обучения этих алгоритмов характерна прямо противоположная тенденция, как следует по рис. 13.2 (слева медленно обучающиеся алгоритмы, справа – быстро обучающиеся). Видно, что основанные на модели алгоритмы обучаются медленнее алгоритмов на основе ценности чуть ли не в пять раз, и такое же соотношение наблюдается дальше по цепочке.

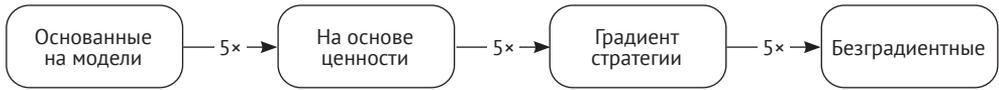


Рис. 13.2 ❖ Сравнение алгоритмов по времени обучения (по возрастанию слева направо)

Мы видим, что чем выше выборочная эффективность алгоритма, тем дольше он обучается, и наоборот. Поэтому необходимо искать компромисс между тем и другим. На самом деле основная цель основанных на модели и более эффективных безмодельных алгоритмов – уменьшить количество шагов взаимодействия со средой, чтобы их было проще обучить и развернуть в реальной окружающей среде, взаимодействие с которой медленнее, чем с имитированной.

От простого к сложному

Определив, какой алгоритм лучше всего отвечает нашим потребностям, будь то один из хорошо известных или какой-то новый, мы должны его реализовать. Как мы видели в этой книге, алгоритмы обучения с подкреплением имеют мало общего с алгоритмами обучения с учителем. Поэтому, говоря об отладке, экспериментировании и настройке тех и других алгоритмов, следует акцентировать внимание на разных аспектах.

- **Начинайте с простых задач.** Начать эксперименты с работоспособной версией алгоритма желательно как можно раньше. Но рекомендуется постепенно увеличивать сложность окружающей среды. Это заметно сократит общее время обучения и отладки. Рассмотрим пример. Если требуется дискретная среда, то можно начать с CartPole-v1, а если непрерывная, то с RoboschoolInvertedPendulum-v1. Затем можно перейти к среде умеренной сложности, например RoboschoolHopper-v1, LunarLander-v2 или еще какой-то, основанной на RGB-изображениях. К этому моменту у вас уже должен быть не содержащий ошибок код, который можно окончательно обучить и настроить на конечную задачу. Кроме того, вы должны как можно лучше понять, как работает алгоритм на простых задачах, чтобы знать, куда смотреть, если что-то пойдет не по плану.
- **Обучение происходит медленно.** Обучение алгоритма ОП занимает время, а форма кривой обучения может быть любой. В предыдущих главах мы видели, что кривые обучения (зависимость полного вознаграждения, усредненного по всем траекториям, от количества шагов) могут быть похожи на логарифмическую функцию, на гиперболический тангенс (см. рис. 13.3) или иметь более сложную форму. Форма зависит от функции и разреженности вознаграждения, а также от сложности окружающей среды. Работая с новой средой, от которой неизвестно чего ожи-

дать, проявите терпение и дайте алгоритму поучиться, пока не будете уверены, что улучшение прекратилось. И не придавайте слишком большого значения графикам обучения.

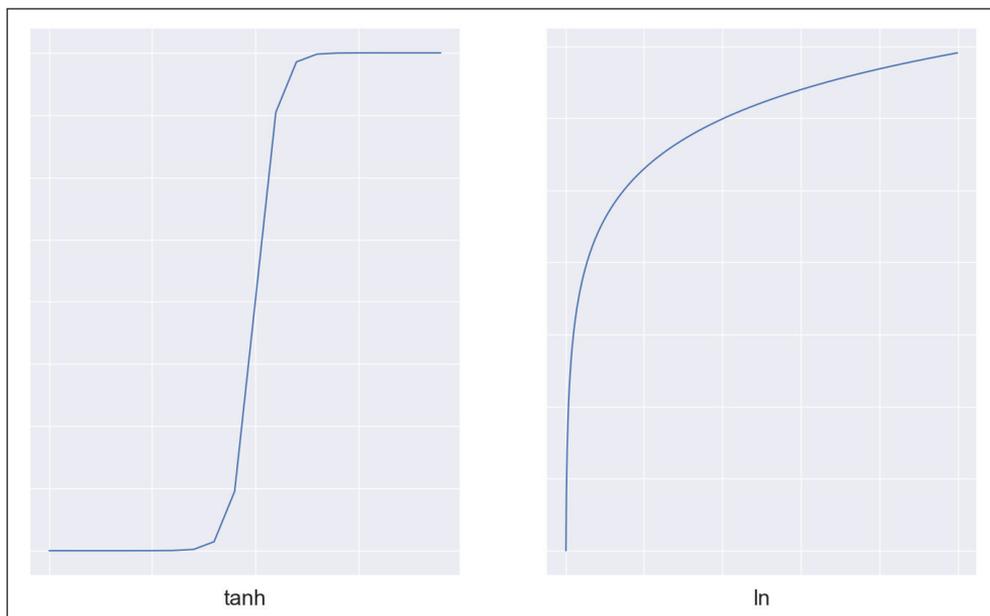


Рис. 13.3 ❖ Форма графиков гиперболического тангенса и логарифма

- **Установите ориентиры.** Для новых задач рекомендуется установить по крайней мере два ориентира, с которыми можно сравнивать алгоритм. Один может быть случайным агентом, а второй – алгоритмом типа REINFORCE или A2C. Затем эти ориентиры можно использовать в качестве нижней границы качества и эффективности.
- **Графики и гистограммы.** Для наблюдения за прогрессированием алгоритма и во время отладки полезно рисовать графики и строить гистограммы ключевых параметров, как то: функция потерь, накопительное вознаграждение, действия (по возможности), длины траекторий, расхождение КЛ, энтропия и функция ценности. Помимо графиков среднего, можно еще построить графики минимума, максимума и стандартного отклонения. В этой книге мы использовали для визуализации TensorBoard, но ничто не мешает взять любой другой инструмент.
- **Используйте несколько начальных значений.** Неотъемлемой частью глубокого обучения является стохастичность как нейронных сетей, так и окружающей среды, поэтому результаты разных прогонов часто довольно сильно различаются. Чтобы обеспечить согласованность и устойчивость, лучше использовать несколько начальных значений генератора случайных чисел.
- **Нормировка.** В зависимости от устройства окружающей среды может быть полезно нормировать вознаграждения, преимущество и наблюдение.

ния. Значения преимущества (например, в алгоритмах TRPO и PPO) можно нормировать, так чтобы среднее по пакету было равно 0, а стандартное отклонение – 1. Для нормировки наблюдений можно использовать набор начальных случайных шагов. А для нормировки вознаграждений использовать скользящее среднее и стандартное отклонение вознаграждения с обесцениванием или без него.

- **Настройка гиперпараметров.** Гиперпараметры зависят от класса и типа алгоритма. Например, у методов на основе ценности много гиперпараметров по сравнению с методами градиента стратегии, но у некоторых представителей этого класса, например TRPO и PPO, гиперпараметров тоже немало. Так или иначе, для каждого из описанных в этой книге алгоритмов мы перечислили все гиперпараметры и отметили, какие из них самые важные. По крайней мере, два гиперпараметра есть у всех алгоритмов ПО: скорость обучения и коэффициент обесценивания. Первый не так важен, как в обучении с учителем, но все равно остается одним из первых кандидатов для настройки. Коэффициент обесценивания – уникальная особенность алгоритмов ОП. Его использование может стать причиной смещения, поскольку модифицируется целевая функция. Впрочем, на практике стратегия получается лучше. Причем тем лучше, чем короче горизонт, поскольку при этом снижается неустойчивость.

Эти рекомендации помогут вам в разработке, обучении и развертывании алгоритмов. И при этом алгоритмы получатся более устойчивыми и робастными.

Критическое отношение к глубокому обучению с подкреплением и понимание его недостатков – ключевой фактор, когда речь заходит о том, чтобы раздвинуть границы возможностей современных алгоритмов. В следующем разделе мы сжато рассмотрим основные проблемы глубокого ОП.

ПРОБЛЕМЫ ГЛУБОКОГО ОП

В последние годы резко возросла интенсивность исследований в области алгоритмов обучения с подкреплением. Особенно впечатляют результаты, достигнутые, когда для аппроксимации функций стали использовать глубокие нейронные сети. Но некоторые важные вопросы все еще остаются нерешенными. Это ограничивает применимость ОП к более сложным и интересным задачам. Мы говорим об устойчивости, воспроизводимости результатов, эффективности и обобщаемости, хотя в этот список можно было бы включить также масштабируемость и проблему исследования.

Устойчивость и воспроизводимость результатов

Устойчивость и воспроизводимость результатов связаны друг с другом, поскольку в обоих случаях цель состоит в том, чтобы спроектировать алгоритм, демонстрирующий стабильную работу в разных прогонах и не слишком сильно зависящий от небольших возмущений, в частности изменения гиперпараметров.

Основной фактор, из-за которого поведение алгоритмов ОП трудно повторить, внутренне присущ глубоким нейронным сетям. Все дело в случайной инициализации весов сети и стохастической оптимизации. Ситуация усугубляется в ОП, поскольку стохастическими являются также окружающие среды. В сочетании эти факторы еще и затрудняют интерпретацию результатов.

Устойчивость также подвергается суровым испытаниям, как мы видели на примере алгоритмов Q-обучения и REINFORCE. Например, для алгоритмов на основе ценности нет никаких гарантий сходимости, и им свойственны высокое смещение и неустойчивость. В алгоритме DQN предпринято немало усилий для стабилизации процесса обучения, например буфер воспроизведения опыта и откладывание обновления целевой сети. Эти приемы в какой-то мере сглаживают проблемы неустойчивости, но не устраняют их полностью.

Для преодоления внутренних проблем, связанных с устойчивостью и воспроизводимостью, необходимо вмешательство извне. С этой целью было разработано много разных эталонных тестов и несколько эвристических правил, которые обеспечивают приемлемый уровень повторяемости и согласованности результатов.

- Если возможно, тестируйте алгоритмы в нескольких похожих средах. Например, протестируйте их в средах Roboschool или Atari Gym, где задачи сравнимы в терминах пространств состояний и действий, но их цели различны.
- Выполняйте много испытаний с разными начальными значениями генератора случайных чисел. Результат может сильно зависеть от начального значения. В качестве примера на рис. 13.4 приведены результаты двух прогонов одного и того же алгоритма с одинаковыми гиперпараметрами, но разными начальными значениями. Как видите, различие велико. Поэтому рекомендуется использовать несколько начальных значений – от трех до пяти в зависимости от поставленной цели. Так, в научных работах считается хорошей практикой усреднять результаты по пяти прогонам, принимая во внимание стандартное отклонение.
- Если результаты нестабильны, попробуйте взять более устойчивый алгоритм или применить дополнительные приемы стабилизации. И всегда помните, что эффект от изменения гиперпараметров может существенно зависеть от алгоритма и окружающей среды.

Эффективность

В разделе «Выбор подходящего алгоритма» мы видели, что алгоритмы существенно различаются по выборочной эффективности. А из предыдущих глав мы знаем, что даже сравнительно эффективные методы, например обучение на основе ценности, все равно нуждаются в большом числе взаимодействий с окружающей средой. Пожалуй, только ОП на основе модели менее прожорливо. Но у таких алгоритмов свои недостатки, в частности более низкая планка качества.

Поэтому были предложены гибридные подходы, сочетающие достоинства основанных на модели и безмодельных методов. Однако их трудно правильно реализовать, и для больших задач они не подходят. Таким образом, проблемы

эффективности решить крайне сложно, но необходимо, если мы собираемся внедрить методы ОП на практике.

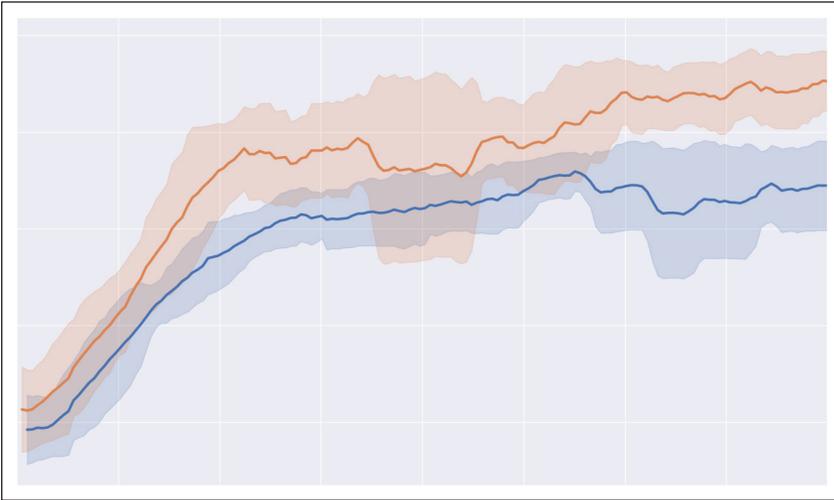


Рис. 13.4 ❖ Результаты двух прогонов одного алгоритма с разными начальными значениями генератора случайных чисел

Есть два способа приспособления к очень медленным окружающим средам, характерным для физического мира. Один – сначала использовать приближенный эмулятор, а затем выполнить точную настройку агента в конечной среде. Другой – обучать агента прямо в конечной среде, но привнести некоторые априорные знания, чтобы не начинать обучение с нуля. Это можно сравнить с обучением вождению, когда ваши органы чувств уже обучены. В обоих случаях речь идет о переносе знания из одной окружающей среды в другую, поэтому такая методология называется переносом обучения. Мы скоро вернемся к ней в разделе «Передовые методы».

Обобщаемость

Под обобщением понимаются два разных, но связанных между собой понятия. В обучении с подкреплением речь идет о способности алгоритма демонстрировать хорошие результаты в сходной окружающей среде. Например, если агент был обучен ходьбе по грунтовым дорогам, то можно ожидать, что он будет вести себя не хуже и на асфальтированных. Чем выше способность к обобщению, тем лучше агент работает в разных средах. Второе, реже употребляемое значение термина «обобщение» относится к свойству алгоритма достигать хороших результатов в окружающей среде, о которой можно собрать ограниченный объем данных.

В ОП агент может самостоятельно выбирать, какие состояния посетить, и делать это так долго, как захочет. Это чревато переобучением. Но если необходима хорошая обобщаемость, то придется искать компромисс. Это вер-

но лишь отчасти, если агенту разрешено собирать потенциально бесконечно много данных о среде, поскольку тогда вступает в дело своего рода механизм саморегуляризации.

Так или иначе, для обобщения на другие окружающие среды агент должен проявить способность к абстрактным рассуждениям, чтобы не просто отображать состояния на действия, а интерпретировать задачу, учитывая различные факторы. Примеры абстрактного рассуждения можно найти в обучении с подкреплением на основе модели, в переносе обучения и в использовании вспомогательных задач. Последнюю тему мы рассмотрим ниже, но в двух словах идея заключается в том, чтобы улучшить обобщаемость и выборочную эффективность, дополнив обучение вспомогательными задачами, которые агент ОП обучается решать вместе с основной.

ПЕРЕДОВЫЕ МЕТОДЫ

У перечисленных выше проблем нет простых решений. Но были предприняты усилия по поиску новаторских подходов, улучшающих эффективность, обобщаемость и устойчивость. Два широко распространенных и многообещающих метода, сосредоточенных на эффективности и обобщаемости, – обучение с подкреплением без учителя и перенос обучения. Чаще всего они работают в симбиозе с алгоритмами глубокого ОП, которые были разработаны в предыдущих главах.

ОП без учителя

ОП без учителя можно сравнить с обычным обучением без учителя, потому что в обоих методах не предполагается наличие наставника. Если в обучении без учителя данные не размечены, то в ОП без учителя не начисляется никакое вознаграждение. То есть в ответ на действие окружающая среда возвращает только следующее состояние. Нет ни вознаграждения, ни флага завершения *done*.

ОП без учителя часто оказывается полезным, например когда невозможно вручную аннотировать среду информацией о вознаграждениях или когда среда может обслуживать разные задачи. Во втором случае обучение без учителя, возможно, позволит обучиться динамике среды. Методы, способные обучаться без учителя, можно использовать также в качестве дополнительного источника информации в средах с сильно разреженным вознаграждением.

Как спроектировать алгоритм, который может что-то узнать об окружающей среде без наставника? Может, просто воспользоваться обучением на основе модели? Но для ОП на основе модели все равно нужен сигнал вознаграждения, чтобы запланировать или логически вывести следующие действия. Так что нужно искать другое решение.

Внутреннее вознаграждение

Одна из возможных альтернатив – разработать внутреннюю функцию вознаграждения агента; это означает, что она контролируется исключительно субъ-

ективными оценками агента – его верой. Этот метод близок тому, как обучаются младенцы. Они применяют чисто исследовательский подход для освоения окружающего мира, не извлекая непосредственной выгоды. Но приобретенные знания могут оказаться полезными позже.

Внутреннее вознаграждение – это своего рода премия за исследование, основанная на оценке новизны состояния. Чем менее знакомо состояние, тем выше внутреннее вознаграждение. Следовательно, у агента имеется стимул к исследованию новых областей окружающей среды. Сейчас вы уже, наверное, понимаете, что внутреннее вознаграждение можно считать альтернативной формой стратегии исследования. На самом деле во многих алгоритмах оно используется наряду с внешним вознаграждением (получаемым от среды), чтобы поощрить исследование в средах с сильно разреженным вознаграждением, каковой является игра *Montezuma's revenge*. Но хотя методы оценивания внутреннего вознаграждения очень похожи на изучавшиеся в главе 12 стратегии, призванные побудить к исследованию среды (эти стратегии все же были связаны с внешним вознаграждением), здесь нас будут интересовать только методы исследования вообще без учителя.

Есть две главные движимые любопытством стратегии, которые вознаграждают за посещение незнакомых состояний и эффективное исследование окружающей среды: на основе подсчета и на основе динамики.

- Стратегии на основе подсчета (или стратегии с **подсчетом посещений**) подсчитывают или оценивают, сколько раз посещалось каждое состояние, и поощряют исследование состояний с низкой частотой посещения, назначая им высокое вознаграждение.
- Стратегии на основе динамики обучают модель динамики среды наряду со стратегией агента и вычисляют внутреннее вознаграждение, ориентируясь на ошибку предсказания, на неопределенность предсказания или на улучшение предсказания. Идея в том, что если подгонять модель к посещенным состояниям, то у новых незнакомых состояний будет более высокая неопределенность или ошибка оценки. Затем эти значения используются для вычисления внутреннего вознаграждения с целью стимулировать исследование неизвестных состояний.

Что, если применять только движимое любопытством исследование к обычным окружающим средам? Этот вопрос изучался в статье «Large-scale study of curiosity-driven learning», и было установлено, что в играх Atari движимые только любопытством агенты могут обучиться решению задач на высоком уровне без какого-либо внешнего вознаграждения. Также отмечено, что в семействе сред Roboschool робот обучился ходить, применяя чистые алгоритмы без учителя, основанные на внутреннем вознаграждении. Авторы статьи предположили, что эти результаты объясняются способом конструирования сред. Действительно, в средах, сконструированных человеком (в частности, в играх), внешнее вознаграждение часто увязывается с целью поиска новизны. Но и в неигровых средах стратегии на основе одного лишь любопытства позволяют исследовать среду и обучаться самостоятельно, без какой-либо помощи наставника. Да и алгоритмы ОП могут значительно выиграть в качестве исследования, а значит, и конечных результатов, если соединят внутреннее вознаграждение с внешним.

Перенос обучения

Перенос знаний из одной окружающей среды в другую, особенно если эти среды похожи, – непростая задача. Стратегии переноса обучения предлагают заполнить разрыв в знаниях, чтобы переход из начальной среды в новую был как можно более простым и безболезненным. Точнее говоря, перенос обучения подразумевает эффективный перенос знаний из исходной окружающей среды (или нескольких сред) в конечную. Чем больше опыта было приобретено на исходных задачах и перенесено на новую, тем быстрее агент обучится решению новой задачи и тем лучше будут результаты этого обучения.

Вообще, говоря о еще не обученном агенте, следует представлять себе систему, в которой нет никакой информации. Мы же, играя в игру, пользуемся большим объемом априорной информации. Например, мы можем догадаться о назначении врагов по их цвету и форме или по динамике движения. Мы можем опознать врага по тому, что он стреляет в нас, как в игре Space Invaders на рис. 13.5. Можно также догадаться об общей динамике игры. Но агент ОП в начале обучения не знает ничего. Это сравнение полезно, потому что позволяет в полной мере оценить важность переноса знаний между разными окружающими средами. Агент, способный использовать опыт, приобретенный при решении исходной задачи, может значительно быстрее обучиться в конечной среде. Например, если исходной была среда Pong, а конечной – Breakout, то можно было бы использовать многие визуальные компоненты, что сэкономило бы немало времени на вычислениях. Чтобы лучше оценить важность этой идеи, только представьте, как удалось бы выиграть в эффективности в существенно более сложных средах.

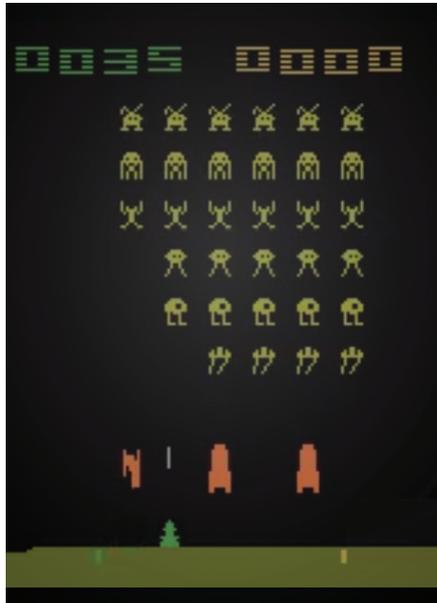


Рис. 13.5 ❖ Игра Space Invaders.
Можете ли вы догадаться о роли разных спрайтов?

В контексте переноса обучения можно услышать термины «обучение с нулевого раза», «обучение с первого раза» и т. д. Имеется в виду количество попыток в конечной предметной области. Например, обучение с нулевого раза означает, что стратегия, обученная в исходной предметной области, применима к конечной непосредственно, без дальнейшего обучения. В этом случае агент должен развить отличные способности к обобщению, чтобы сразу приспособиться к новой задаче.

Типы переноса обучения

Типов переноса обучения много, какой из них использовать, зависит от конкретной ситуации и потребностей. Одно из различий связано с количеством исходных окружающих сред. Очевидно, что чем в большем числе сред обучался агент, тем разнообразнее и обширнее опыт, который он сможет применить к конечной среде. Перенос обучения из нескольких предметных областей называется многозадачным обучением.

1-ЗАДАЧНОЕ ОБУЧЕНИЕ Под 1-задачным обучением, или просто переносом обучения, понимается обучение стратегии в одной предметной области и ее перенос в другую область. Для этого есть три основных способа.

- **Точная настройка.** Имеется в виду уточнение обученной модели на целевой задаче. Если вы занимались машинным обучением, а особенно компьютерным зрением или обработкой естественного языка, то, наверное, знаете, что это такое. К сожалению, в обучении с подкреплением выполнить точную настройку не так просто, как в вышеупомянутых дисциплинах, поскольку требуется более тщательно прорабатывать инженерные детали, а выигрыш в общем случае не так велик. Причина в том, что пропасть между двумя задачами ОП обычно шире, чем между разными видами изображений. Например, классификация кошек и собак различается не так сильно, как игры Pong и Breakout. Тем не менее точную настройку использовать в ОП можно, и настройка лишь нескольких последних слоев (или их замена, если пространство действий полностью отличается) может улучшить свойства обобщаемости.
- **Рандомизация предметной области.** Этот способ основан на идее о том, что диверсификация динамики исходной предметной области увеличивает робастность стратегии в новой окружающей среде. Для реализации производятся различные манипуляции с исходной предметной областью, например изменяется физика эмулятора, так чтобы стратегия, обученная на нескольких случайно модифицированных исходных средах, оказалась достаточно робастной для работы в конечной среде. Этот подход более эффективен для обучения агентов, которые будут эксплуатироваться в реальном мире. В данной ситуации стратегия получается более робастной, и для достижения требуемого качества имитационная модель необязательно должна в точности повторять физический мир.
- **Адаптация предметной области.** Этот процесс чаще всего применяется, чтобы отобразить стратегию, обученную на исходной имитационной модели, на конечный физический мир. Делается это путем изменения распределения данных в исходной предметной области, так чтобы оно

было таким же, как в конечной. Обычно этот подход используется в задачах, основанных на изображениях, а для преобразования синтетических изображений в реалистичные применяются **порождающие состязательные сети (ПСС)** (англ. generative adversarial networks – GAN).

МНОГОЗАДАЧНОЕ ОБУЧЕНИЕ При многозадачном обучении чем больше количество окружающих сред, в которых обучался агент, тем разнообразнее его опыт и тем лучшие результаты он покажет в конечной окружающей среде. На нескольких исходных задачах могут обучаться как один, так и несколько агентов. Если обучался только один агент, то развернуть его в конечной среде легко. Если же несколько агентов обучались решению разных задач, то можно либо объединить результирующие стратегии в ансамбль и усреднять предсказания, данные ими для целевой задачи, либо выполнить промежуточный шаг *дистилляции* для объединения стратегий в одну. В последнем случае знания ансамбля моделей сжимаются в единую стратегию, которую проще развернуть и которая быстрее производит логический вывод.

ОП В РЕАЛЬНОМ МИРЕ

До сих пор в этой главе мы рассматривали передовые практики разработки алгоритмов глубокого ОП и стоящие перед ОП вызовы. Мы также видели, как ОП без учителя и метаобучение могут в какой-то мере снять остроту проблем низкой эффективности и плохой обобщаемости. Теперь мы хотим показать, какие задачи приходится решать агенту ОП в реальном мире и как можно преодолеть разрыв между реальной и имитированной окружающей средой.

Проектирование агента, способного действовать в реальном мире, – далеко не тривиальная задача. Но большинство приложений ОП для этого и разрабатываются. Поэтому необходимо понять, с какими проблемами придется столкнуться, и познакомиться с некоторыми полезными приемами.

Лицом к лицу с реальным миром

Помимо таких крупных проблем, как выборочная эффективность и обобщаемость, в реальном мире мы сталкиваемся и с другими, в т. ч. ограничениями безопасности и предметной области. На самом деле агент часто не свободен в своих взаимодействиях с окружающим миром из-за ограничений по безопасности и стоимости. Решение могут дать алгоритмы с ограничениями, например TRPO и PPO, которые встраиваются в системные механизмы, чтобы ограничить величину изменения действий в процессе обучения. Это может предотвратить слишком сильное изменение в поведении агента. К сожалению, в предметных областях, особенно подверженных риску, этого недостаточно. Например, в нынешних условиях мы не можем начать обучать беспилотный автомобиль прямо на дороге. Стратегии могут потребоваться сотни или тысячи циклов обучения, чтобы понять, что падение с обрыва ведет к печальным последствиям, и научиться избегать его. Обучение стратегии сначала на имитационной модели – разумный и практически осуществимый подход. Как бы

то ни было, при использовании в больших городах следует принимать более осторожные решения.

Как мы только что сказали, начальное обучение на имитационной модели практически осуществимо и в зависимости от сложности задачи может давать хорошие результаты. Но эмулятор должен как можно точнее имитировать реальную окружающую среду. Например, эмулятор, изображенный слева на рис. 13.6, не годится, если мир выглядит так, как показано справа. Это несоответствие между реальным и имитированным мирами называется *разрывом с реальностью* (reality gap):



Рис. 13.6 ❖ Сравнение искусственного и реального миров

С другой стороны, создать очень точную и реалистичную окружающую среду не всегда возможно. Проблема упирается в недостаточность вычислительных ресурсов эмулятора. Частично ее можно преодолеть, начав с более быстрого и менее точного эмулятора и постепенно увеличивая сходство с реальностью, чтобы сократить разрыв. Конечно, скорость при этом уменьшится, но в этот момент агент, по идее, уже должен быть обучен, так что для донастройки потребуется всего несколько итераций. Но разработать высококачественные эмуляторы, точно имитирующие физический мир, очень трудно. Поэтому на практике разрыв с реальностью остается, и для разрешения этой ситуации нужны методы, повышающие обобщаемость.

Преодоление разрыва между имитационной моделью и реальным миром

Для органичного перехода от имитационной модели к реальному миру и тем самым преодоления разрыва с реальностью можно использовать некоторые из представленных выше методов повышения обобщаемости, например адаптацию и рандомизацию предметной области. Так, авторы статьи «Learning Dexterous In-Hand Manipulation» обучили человекоподобного робота невероятно ловко манипулировать предметами, применив рандомизацию предметной области. Стратегия была обучена на большом числе различных параллельных имитаций, спроектированных так, чтобы агент приобрел разнообразный опыт

обращения с предметами, имеющими разные физические и визуальные характеристики. Ключом был механизм, отдающий предпочтение обобщаемости перед реалистичностью; в результате развернутая система продемонстрировала обширный набор стратегий манипуляции – «ловкости рук», многие из которых свойственны и человеку.

Создание собственной окружающей среды

В этой книге мы использовали в основном небольшие, но быстро решаемые задачи, отвечающие нашим педагогическим целям. Но существует немало эмуляторов для тренировки двигательной активности (например, Gazebo, Roboschool и Mujoco), проектирования механических систем, транспортных средств, беспилотных автомобилей, систем безопасности и многого другого. Однако при всем многообразии всегда найдется приложение, для которого окружающей среды еще нет. И тогда придется создать свою собственную.

Спроектировать функцию вознаграждения трудно, но это главная часть ОП. Если функция вознаграждения неправильная, то агент может обучиться неправильному поведению в окружающей среде. В главе 1 мы привели примеры игры с водными катерами, в которой катер максимизировал вознаграждение, двигаясь по кругу и круша возрождающиеся мишени, вместо того чтобы как можно скорее прийти к финишу. Такого рода поведения следует избегать при проектировании функции вознаграждения.

Общая рекомендация по проектированию функции вознаграждения (применимая к любой окружающей среде) – использовать положительные вознаграждения для поощрения исследования и избегания заключительных состояний или отрицательные вознаграждения, если цель состоит в том, чтобы достичь заключительного состояния как можно быстрее. Форма функции вознаграждения тоже имеет большое значение. В этой книге мы не раз предостерегали против разреженного вознаграждения. Оптимальная функция должна быть гладкой и плотной.

Если по какой-то причине функцию вознаграждения очень трудно описать формулами, то есть еще два способа передать сигнал вознаграждения:

- провести демонстрацию решения задачи, применив подражательное обучение или обратное обучение с подкреплением;
- использовать мнение человека для обратной связи с поведением агента.

i Последний подход пока довольно новый. Если интересно, можете почитать о нем в статье «Deep Reinforcement Learning from Policy-Dependent Human Feedback» (<https://arxiv.org/abs/1902.04257>).

Будущее ОП и его влияние на общество

Основы искусственного интеллекта были заложены более 50 лет назад, но только в последние несколько лет инновации, принесенные ИИ, распространились по миру и стали общепринятой технологией. Эта волна инноваций связана прежде всего с развитием глубоких нейронных сетей в системах обучения с учителем. Но совсем уж недавние прорывы в области ИИ опираются на обуче-

ние с подкреплением и, главным образом, на глубокое обучение с подкреплением. Результаты, полученные в играх го и Dota, и другие из того же ряда подтверждают впечатляющее качество алгоритмов ОП, продемонстрировавших способность к долгосрочному планированию, работе в команде и обнаружению новых стратегий, которые трудно понять человеку.

Достойные восхищения результаты, полученные в имитированных средах, породили новую волну приложений обучения с подкреплением в реальном мире. Мы еще только в начале пути, но многие отрасли уже ощутили на себе влияние этой технологии и подверглись значительной трансформации. Агенты ОП, внедряемые в повседневную жизнь, могут повысить качество жизни путем автоматизации рутинной работы, ответа на всемирные вызовы, открытия новых лекарств – и это лишь малая толика открывающихся возможностей. Однако системы, которые войдут в наш мир и в нашу жизнь, должны быть безопасными и надежными. Мы еще не достигли этой цели, но находимся на правильном пути.

Этичное использование ИИ стало предметом общественного внимания, как, например, в случае применения автономного оружия. При таком быстром техническом прогрессе политикам и населению трудно инициировать открытые дискуссии по этим вопросам. Многие влиятельные и уважаемые люди считают, что ИИ представляет потенциальную угрозу человечеству. Но предсказать будущее невозможно, а технологии предстоит пройти еще долгий путь, прежде чем будут разработаны агенты, наделенные способностями, сравнимыми с человеческими. Наша способность к творчеству, наши эмоции, наше умение адаптироваться – все это пока недоступно ОП.

При должном внимании краткосрочные преимущества, которые несет ОП, намного перевешивают любой негатив. Но, размещая развитых агентов ОП в физической окружающей среде, мы не должны забывать о вышеперечисленных проблемах и вызовах. Все они разрешимы, а когда это случится, обучение с подкреплением сможет уменьшить социальное неравенство, повысить качество нашей жизни и условий жизни на нашей планете.

РЕЗЮМЕ

В этой книге мы изучили и реализовали много алгоритмов обучения с подкреплением, но это разнообразие только смущает, когда дело доходит до выбора конкретного алгоритма. Поэтому в последней главе мы привели эвристическое правило, которым можно руководствоваться, выбирая класс алгоритмов ОП, наиболее подходящий для решения поставленной задачи. Оно учитывает главным образом время вычислений и выборочную эффективность. Мы также дали несколько полезных советов по обучению и отладке алгоритмов глубокого обучения с подкреплением, следуя которым, можно улучшить результат и упростить процесс.

Мы обсудили скрытые проблемы обучения с подкреплением: устойчивость, воспроизводимость результатов, эффективность и обобщаемость. Это основные препятствия, которые предстоит преодолеть на пути к использованию агентов ОП в реальном мире. Мы подробно остановились на обучении с под-

крепленим без учителя и переносе обучения – двух подходах, позволяющих значительно улучшить обобщаемость и выборочную эффективность.

Кроме того, мы описали наиболее важные открытые проблемы, а также культурные и технологические последствия обучения с подкреплением для нашей жизни.

Мы надеемся, что эта книга позволила вам разобраться в обучении с подкреплением и подстегнула интерес к этой завораживающей области знаний.

Вопросы

1. Расположите алгоритмы DQN, A2C и ЭС в порядке возрастания выборочной эффективности.
2. А как бы их расположили по возрастанию времени обучения при наличии 100 процессоров?
3. В какой окружающей среде вы начали бы отладку алгоритма ОП: CartPole или Montezuma's Revenge?
4. Почему рекомендуется использовать несколько начальных значений генератора случайных чисел при сравнении нескольких алгоритмов глубокого ОП?
5. Помогает ли внутреннее вознаграждение в исследовании окружающей среды?
6. Что такое перенос обучения?

Для дальнейшего чтения

- Подход к играм Atari на основе одного лишь любопытства описан в статье «Large-scale study of curiosity-driven learning» (<https://arxiv.org/pdf/1808.04355.pdf>).
- О практическом использовании рандомизации предметной области в задаче о манипуляции предметами в руках см. статью «Learning Dexterous In-Hand Manipulation» (<https://arxiv.org/pdf/1808.00177.pdf>).
- Некоторые соображения о том, как обратную связь со стороны человека можно применить в качестве альтернативы функции вознаграждения, см. в статье «Deep Reinforcement Learning from Policy-Dependent Human Feedback» (<https://arxiv.org/pdf/1902.04257.pdf>).

Ответы на вопросы

Глава 3

- Что такое стохастическая стратегия?
Это стратегия, определенная в терминах распределения вероятностей.
- Как можно определить функцию дохода в терминах дохода на следующем временном шаге?
 $G_t = r_t + \lambda G_{t+1}$.
- Почему так важно уравнение Беллмана?
Потому что оно дает общую формулу для вычисления ценности состояния при условии известности текущего вознаграждения и ценности следующего состояния.
- Какие факторы ограничивают применимость алгоритмов ДП?
Из-за экспоненциального роста сложности обучения количество состояний необходимо ограничивать. Другое ограничение – должна быть известна динамика системы.
- Что такое оценивание стратегии?
Это итеративный метод вычисления функции ценности для данной стратегии с помощью уравнения Беллмана.
- В чем различие между итерацией по стратегиям и итерацией по ценности?
В случае итерации по стратегиям шаги оценивания и улучшения стратегии чередуются, а в случае итерации по ценности оба шага объединяются в одно обновление с помощью функции \max .

Глава 4

- Какое основное свойство метода Монте-Карло используется в ОП?
Оценка функции ценности как среднего дохода при старте из данного состояния.
- Почему методы Монте-Карло называются офлайнowymi?
Потому что они обновляют ценность состояния, только когда известна вся траектория. Поэтому они должны дожидаться завершения эпизода.
- Назовите две основные идеи TD-обучения.
Выборка и бутстрэппинг.
- В чем различие между методами Монте-Карло и TD-методами?
Методы Монте-Карло обучаются на полной траектории, а TD-методы – на каждом шаге, приобретая знания даже на неполных траекториях.
- Почему для TD-обучения так важно исследование?
Потому что TD-обновление производится только для посещенных пар состояние–действие, так что если какая-то пара не была обнаружена, то в отсутствие исследования она никогда не будет посещена. Это значит, что хорошая стратегия может так и остаться найденной.

- Почему Q-обучение является алгоритмом с разделенной стратегией?
Потому что обновление в Q-обучении производится независимо от поведенческой стратегии. Применяется жадная стратегия с операцией \max .

Глава 5

- Когда возникает смертельная триада?
Когда обучение с разделенной стратегией сочетается с аппроксимацией функции и бутстрэппингом.
- Как DQN борется с неустойчивостью?
Используется буфер воспроизведения и отдельные онлайн и целевая сеть.
- В чем суть проблемы движущейся мишени?
Эта проблема возникает, когда целевые значения не фиксированы, а изменяются в процессе оптимизации сети.
- Как DQN удается смягчить проблему движущейся мишени?
Путем введения целевой сети, которая обновляется реже, чем онлайн.
- Опишите процедуру оптимизации, используемую в DQN.
Функция потерь, равная среднеквадратической ошибке, оптимизируется методом стохастического градиентного спуска по пакету.
- Как определяется функция преимущества пары состояние–действие?
 $A(s, a) = Q(s, a) - V(s)$.

Глава 6

- Как алгоритмы градиента стратегии максимизируют целевую функцию?
Делают шаг в направлении, противоположном градиенту целевой функции. Длина шага пропорциональна доходу.
- В чем основная идея алгоритмов градиента стратегии?
Поощрять хорошие действия и отговаривать агента от плохих.
- Почему после включения базы в алгоритм REINFORCE он остается несмещенным?
Потому что математическое ожидание $E[\nabla_{\theta} \log \pi_{\theta}(\tau) b] = 0$.
- К какому более широкому классу алгоритмов принадлежит REINFORCE?
Это метод Монте-Карло, потому что ему нужны полные траектории.
- Чем критик в методах AC отличается от функции ценности, которая используется в качестве базы в алгоритме REINFORCE?
Хотя обучаемая функция одинакова, приближенная функция ценности используется критиком для бутстрэппинга ценности пары состояние–действие, тогда как в REINFORCE (но также и в AC) она используется в качестве базы для уменьшения дисперсии.
- Если бы вам нужно было разработать алгоритм агента, который обучается двигаться, то что бы предпочли: REINFORCE или AC?
Сначала следовало бы попробовать алгоритм исполнитель–критик, поскольку агенту предстоит обучиться решению непрерывной задачи.

- Можно ли использовать n -шаговый алгоритм AC как REINFORCE?
Да, при условии что n больше максимально возможного числа шагов взаимодействия с окружающей средой.

Глава 7

- Как входящая в состав стратегии нейронная сеть может управлять непрерывным агентом?
Один из способов – предсказывать среднее и стандартное отклонение нормального распределения. Стандартное отклонение может быть либо обусловлено состоянием (входом нейронной сети), либо быть отдельным параметром.
- Что такое расхождение Кульбака–Лейблера?
Это мера близости двух распределений вероятности.
- В чем основная идея алгоритма TRPO?
Оптимизировать новую целевую функцию в области, близкой к старому распределению вероятностей.
- Как расхождение КЛ используется в TRPO?
Как жесткое ограничение, лимитирующее расхождение между старой и новой стратегиями.
- В чем главное преимущество алгоритма PPO?
В том, что используется только первая производная. Это упрощает алгоритм и повышает выборочную эффективность и качество.
- За счет чего PPO добивается высокой выборочной эффективности?
За счет того, что выполняет обновления на мини-пакете несколько раз, добиваясь лучшего использования данных.

Глава 8

- В чем состоит главное ограничение алгоритмов Q-обучения?
В том, что пространство состояний должно быть дискретным и небольшим, чтобы можно было вычислить глобальный максимум.
- Почему стохастические алгоритмы градиента стратегии выборочно неэффективны?
Потому что это алгоритмы с единой стратегией, и им нужны новые данные при каждом изменении стратегии.
- Как в DPG обходится проблема глобальной максимизации?
DPG моделирует стратегию как детерминированную функцию, предсказывающую только детерминированное действие, а теорема о градиенте детерминированной стратегии подсказывает, как можно вычислить градиент, необходимый для обновления стратегии.
- Как в DPG гарантируется достаточный уровень исследования?
Путем прибавления шума к детерминированной стратегии или путем обучения с помощью другой поведенческой стратегии.

- Как расшифровывается акроним DDPG? И в чем состоит главный вклад этого алгоритма?
DDRG означает Deep Deterministic Policy Gradient. Этот алгоритм адаптирует градиент детерминированной стратегии к работе с глубокими нейронными сетями. В нем используются новые стратегии для повышения устойчивости и ускорения обучения.
- Какие проблемы стремится решить алгоритм TD3?
Завышение оценки, присущее Q-обучению, и высокая дисперсия.
- Какие новые механизмы используются в TD3?
Для борьбы с завышением оценки применяется обрезанное двойное Q-обучение, а для решения проблемы высокой дисперсии – отложенное обновление стратегии и гладкая регуляризация.

Глава 9

- Если бы в вашем распоряжении было всего 10 партий, на которых нужно обучить агента играть в шашки, то что бы вы выбрали: безмодельный или основанный на модели алгоритм?
Алгоритм, основанный на модели. Модель игры в шашки известна, и планирование практически осуществимо.
- Назовите недостатки алгоритмов на основе модели.
Им требуется больше вычислительных ресурсов, а асимптотическое качество хуже, чем у безмодельных алгоритмов.
- Если модель окружающей среды неизвестна, как ее можно обучить?
После того как в процессе взаимодействия с окружающей средой сформирован набор данных, динамику модели можно обучить, пользуясь стандартными методами обучения с учителем.
- Почему используются методы агрегирования данных?
Потому что обычно для первых взаимодействий со средой применяется наивная стратегия, которая не исследует среду. В процессе последующих взаимодействий нужна более акцентированная стратегия, которая точнее отражает модель окружающей среды.
- Как алгоритм ME-TRPO стабилизирует обучение?
Благодаря двум основным особенностям: ансамблю моделей и технике ранней остановки.
- Как использование ансамбля моделей улучшает обучение стратегии?
Предсказания, сделанные ансамблем моделей, учитывают недостоверность одиночной модели.

Глава 10

- Считается ли подражательное обучение одним из методов обучения с подкреплением?
Нет, потому что базовые предпосылки различаются. Целью подражательного обучения не является максимизация вознаграждения, как в случае ОП.

- Стали бы вы использовать подражательное обучение для создания непобедимого агента для игры го?
Скорее всего, нет, потому что необходим эксперт, у которого можно учиться. А если мы хотим, чтобы агент стал лучшим в мире игроком, то равного ему эксперту не найти.
- Как расшифровывается аббревиатура DAgger?
Dataset aggregation – агрегирование набора данных.
- В чем основное достоинство DAgger?
Он обходит проблему несовпадения распределений, благодаря тому что эксперт активно обучает ученика восстанавливаться после ошибок.
- Когда бы вы стали использовать IRL вместо IL?
В задачах, где функцию вознаграждения проще обучить и необходимо обучиться стратегии, лучшей, чем стратегия эксперта.

Глава 11

- Назовите две альтернативы обучению с подкреплением для решения задач последовательного принятия решения.
Эволюционные стратегии и генетические алгоритмы.
- С помощью каких процессов порождаются новые особи в эволюционных алгоритмах?
Мутация – изменение гена родителя и скрещивание – объединение генетической информации двух родителей.
- Что послужило источником вдохновения для эволюционных алгоритмов, в т. ч. для генетических алгоритмов?
Прежде всего биологическая эволюция.
- Как в алгоритме CMA-ES эволюционируют стратегии?
CMA-ES выбирает нового кандидата из многомерного нормального распределения, ковариационная матрица которого адаптирована к популяции.
- Назовите одно достоинство и один недостаток эволюционных стратегий.
Достоинство – эти методы не нуждаются в вычислении производной. Недостаток – низкая выборочная эффективность.
- Какой описанный в статье «Evolution Strategies as a Scalable Alternative to Reinforcement Learning» прием позволяет уменьшить дисперсию?
Авторы предлагают зеркальное отражение шума – генерируют дополнительную мутацию, применяя возмущение с противоположным знаком.

Глава 12

- В чем состоит дилемма исследования–использования?
В том, чтобы решить, что лучше: исследовать пространство состояний в надежде найти лучшие решения или использовать наилучшее из уже известных решений.
- Какие две стратегии исследования уже встречались в рассмотренных ранее алгоритмах ОП?
 ϵ -жадная стратегия и внесение дополнительного шума в стратегию.

- Что такое ВДГ (UCB)?
Верхняя доверительная граница – оптимистический алгоритм исследования, в котором оценивается верхняя доверительная граница для каждой ценности и выбирается действие, которое ее максимизирует – по формуле (12.3).
- Какая задача труднее: игра Montezuma’s Revenge или задача о многоруком бандите?
Игра Montezuma’s Revenge гораздо труднее задачи о многоруком бандите хотя бы потому, что в последней вообще нет внутреннего состояния, тогда как в первой таких состояний астрономическое число. Кроме того, правила игры сложнее.
- Как в метаалгоритме ESBAS решается проблема онлайн-выбора алгоритма ОП?
Применением метаалгоритма, который выбирает, какой алгоритм из фиксированного портфеля лучше работает в данных обстоятельствах.

Глава 13

- Расположите алгоритмы DQN, A2C и ЭС в порядке возрастания выборочной эффективности.
Самый выборочно эффективный – DQN, за ним следуют A2C и ЭС.
- А как бы их расположили по возрастанию времени обучения при наличии 100 процессоров?
Пожалуй, быстрее всего обучится ЭС, за ним A2C и DQN.
- В какой окружающей среде вы начали бы отладку алгоритма ОП: CartPole или Montezuma’s Revenge?
CartPole. Начинать отладку алгоритма следует с легкой задачи.
- Почему рекомендуется использовать несколько начальных значений генератора случайных чисел при сравнении нескольких алгоритмов глубокого ОП?
Результаты одного испытания могут быть очень нестабильны из-за стохастической природы нейронной сети и окружающей среды. Усреднение результатов с несколькими начальными значениями даст более стабильную картину.
- Помогает ли внутреннее вознаграждение в исследовании окружающей среды?
Да, поскольку внутреннее вознаграждение – своего рода премия за исследование, стимулирующая любопытство агента и побуждающая его посещать новые состояния.
- Что такое перенос обучения?
Это эффективный перенос знаний об одной окружающей среде на другую.

Предметный указатель

Символы

- 1-задачное обучение, способы
 - адаптация предметной области, 269
 - рандомизация предметной области, 269
 - точная настройка, 269
- ϵ -жадная стратегия, 246

A

- Advantage Actor-Critic (A2C), 142
- Arcade Learning Environment (ALE), среда разработки, 102, 214
- Asynchronous Advantage Actor-Critic (A3C), 142
- Atari, игры, 102

B

- BipedalWalker-v2
 - применение DDPG, 180
 - применение TD3, 186

C

- CMA-ES, эволюционная стратегия с адаптацией ковариационной матрицы, 231
- CoinRun, среда, 53

D

- Dataset Aggregation (Dagger), алгоритм, 216
 - загрузка экспертной модели логического вывода, 217
 - реализация, 217
 - создание графа вычислений обучаемого, 218
 - создание цикла Dagger, 219
- DDPG, алгоритм, 174
 - буфер воспроизведения, 175
 - целевая сеть, 175
- DDPG, глубокий детерминированный градиент стратегии, 170, 174
 - применение к среде BipedalWalker-v2, 180
 - реализация, 176

DDQN

- реализация, 115
- результаты, 115
- DeepMind Lab, среда, 53
- DeepMind PySC2, среда, 54
- done, флаг, 39
- Double DQN, 114
- DPG, детерминированный градиент стратегии, 172
- DQN, алгоритм, 97
 - архитектура, 101
 - буфер воспроизведения, 97, 106
 - глубокие нейронные сети, 106
 - граф вычислений, 107
 - применение к игре Pong, 102
 - псевдокод, 100
 - реализация, 105
 - функция потерь, 99
 - целевая сеть, 98
 - цикл обучения, 107
- DQN, вариации, 113
 - Double DQN, 114
 - Dueling DQN, 117
 - n -шаговый DQN, 118
- Duckietown, среда, 53
- Dueling DQN, 117
 - реализация, 117
 - результаты, 118

E

- ESBAS (epochal stochastic bandit algorithm selection), 243, 250
 - выбор алгоритма, 249
 - практические проблемы, 252
 - реализация, 252
 - тестирование в среде Acrobot, 255

F

- FIFO-очередь, 106
- Flappy Bird, игра, 214
 - анализ результатов, 221
 - взаимодействие с окружающей средой, 215

установка, 209
 FrozenLake, игра
 применение итерации по стратегиям, 67
 применение итерации по ценности, 70

G

GAE, обобщенная оценка
 преимущества, 163
 Gym Atari, среда, 52
 Gym Classic control, среда, 52
 Gym MuJoCo, среда, 52

L

LunarLander, применение
 масштабируемой ЭС, 239

M

MalmoEnv, среда, 53
 ME-TRPO, оптимизация стратегии
 в доверительной области с применением
 ансамбля моделей, 190
 применение к задаче об обратном
 маятнике, 199
 реализация, 200

N

n -шаговая модель AC, 138
 n -шаговый DQN, 118
 реализация, 119
 NEAT, нейроэволюция нарастающих
 топологий, 230

O

OpenAI Gym
 и цикл ОП, 38
 установка, 36

P

PLE, среда, 53
 Pommerman, среда, 53
 Pong, применение алгоритма DQN, 102
 PPO, проксимальная оптимизация
 стратегии, 163
 краткое описание, 163
 реализация, 164
 сравнение с TRPO, 166
 PyGame Learning Environment (PLE), 214

Q

Q-обучение, 86, 94
 алгоритм, 87
 применение к Taxi-v2, 87

с нейронными сетями, 95
 сочетание с градиентом стратегии, 170
 сравнение с SARSA, 89
 теория, 86
 Q-функция, 60

R

REINFORCE алгоритм, 127
 применение к задаче о посадке
 космического корабля, 132
 реализация, 129
 реализация варианта с базой, 136
 с базой, 134
 Roboschool, 144
 ссылка, 53
 установка, 37
 эксперименты, 204

S

SARSA
 алгоритм, 80
 общее описание, 80
 применение к Taxi-v2, 81
 сравнение с Q-обучением, 89
 SSBAS, скользящий стохастический
 бандит, 251

T

Taxi-v2
 применение Q-обучения, 87
 применение SARSA, 81
 TD3, двойной глубокий
 детерминированный градиент стратегии
 с задержкой, 170, 182
 отложенное обновление стратегии, 184
 применение к среде
 BipedalWalker-v2, 186
 проблема завышения оценки, 182
 реализация, 182
 регуляризация целевой сети, 184
 уменьшение дисперсии, 184
 TD-обучение, 23, 75, 78
 обновление, 79
 улучшение стратегии, 79
 TD-цель, 79
 TensorBoard, 49
 документация, 51
 TensorFlow
 пример линейной регрессии, 46
 разработка моделей MO, 42
 создание графа, 45
 тензор, 43
 TRPO, оптимизация стратегии
 в доверительной области, 152

алгоритм, 153
 применение, 160
 сравнение с PPO, 167

U

UCB (верхняя доверительная граница), алгоритм, 243, 247
 UCB1, 247
 Unity ML-Agents среда, 53

V

V-функция, 60

A

Агент, 20
 Адаптация предметной области, 269
 Активное подражание, 214
 Алгоритмы глубокого ОП
 выбор, 260
 инструменты создания, 36
 недостатки, 263
 Аппроксимация функций, 94

Б

Безмодельное обучение, 76
 оценивание стратегии, 77
 порядок действий, 76
 проблема исследования, 77
 Безмодельные алгоритмы, 62
 алгоритмы градиента стратегии, 62
 алгоритмы на основе ценности, 62
 гибридные алгоритмы, 63
 Беллмана уранение, 60

В

Вознаграждение, 29, 39
 Выбор алгоритма (AS), 249

Г

Генетические алгоритмы, 230
 Генотип, 228
 Гибридные алгоритмы, 63
 Глубокая Q-сеть (DQN), 24
 Глубокие нейронные сети (ГНС), 93
 Глубокое обучение, проблемы, 263
 обобщаемость, 265
 устойчивость и воспроизводимость результатов, 263
 эффективность, 264
 Глубокое обучение с подкреплением, 24
 открытые вопросы, 26
 рекомендуемые практики, 259

Глубокое Q-обучение, неустойчивость, 96
 Градиент, вычисление, 125
 Градиент стратегии (ГС), 123
 алгоритмы, 62
 с единой стратегией, 127
 сочетание с Q-обучением, 170

Д

Действие, 20
 Детерминированная стратегия, 58
 Динамическое программирование (ДП), 23, 64, 75
 итерация по стратегиям, 66
 итерация по ценности, 70
 оценивание стратегии, 65
 улучшение стратегии, 66
 Доверительная область, 153

Е

Единая стратегия, 27
 Естественный градиент стратегии, 148
 интуитивное описание, 149
 информационная матрица Фишера, 151
 осложнения, 152
 проблемы, 148
 расхождение Кульбака–Лейблера (KL), 151

З

Задача о многоруком бандите, 245
 Затухание эpsilon, 78

И

Игра с собой, 31
 Известная модель, 192
 Изображения
 построение модели из, 198
 Информационная матрица Фишера (ИМФ), 150
 Исполнитель–критик (AC), алгоритм, 63, 137
 дополнительные улучшения, 142
 посадка космического корабля, 141
 реализация, 139
 Исследование
 и использование, 244
 подходы, 246
 проблема, 77
 сложность, 248
 Итерация по стратегиям, 66
 применение к игре FrozenLake, 67
 Итерация по ценности, применение к игре FrozenLake, 70

К

Коэффициент обесценивания, 59
 Кульбака–Лейблера (КЛ) расхождение, 150

Л

Линейная регрессия, пример, 46

М

Марковский процесс принятия решений (МППР), 23, 56
 доход, 58
 стратегия, 58
 уравнение Беллмана, 60
 функции ценности, 59
 Марковское свойство, 57
 Масштабируемые эволюционные стратегии, 232
 гиперпараметры, 239
 главная функция, 236
 исполнители, 237
 применение к LunarLander, 239
 реализация, 234
 Метод ценности действий, 28
 Методы, основанные на модели, 30
 Модель
 построение из изображений, 198
 переобучение, 195
 Монте-Карло методы, 77
 Мутация, 229

Н

Наблюдение, 39, 58, 94
 Неверная модель, 195
 Независимость и одинаковое распределение, 97
 Неизвестная модель, 193
 Непрерывная система, управление, 145

О

Обработки естественного языка (ОЕЯ), 32
 Обратное ОП (IRL), 29
 Обучение без учителя, 23
 Обучение на основе модели, 191
 достоинства и недостатки, 195
 сочетание с безмодельным обучением, 196
 Обучение с подкреплением (ОП), 19, 35
 алгоритмы
 безмодельные, 62
 классификация, 61
 на основе модели, 63
 разнообразия, 64

альтернатива, 226
 будущее и влияние на общество, 272
 в реальном мире, 270
 и обучение с учителем, 22
 и подражательное обучение, 211
 история, 23
 и эволюционные стратегии, 232
 окружающие среды
 с открытым исходным кодом, 52
 типы, 51
 характеристики, 52
 полезные свойства, 232
 применения, 30
 примеры, 21
 распределение поощрения
 во времени, 226
 характеристика, 21
 цикл, 38
 элементы, 26
 Обучение с учителем и обучение с подкреплением, 22
 ОП без учителя, 23, 266
 внутреннее вознаграждение, 266
 Оптимальная стратегия, 58
 итерация по стратегиям, 66
 итерация по ценности, 66
 Особи, 227
 Отражение шума, 234

П

Пассивная целевая стратегия, 27
 Перенос обучения, 268
 1-задачное обучение, 269
 многозадачное обучение, 270
 Планирование, 192
 Плотное вознаграждение, 29
 Поведенческая стратегия, 27
 Подражание, 209
 пример помощника водителя, 210
 Подражательное обучение, 29, 208
 активное и пассивное подражание, 214
 и обучение с подкреплением, 211
 структура, 212
 эксперт, 211
 Подсчетом посещений, стратегия, 267
 Поколение, 229
 Полная наблюдаемость, 57
 Полносвязная нейронная сеть (ПНС), 25
 Порождающие состязательные сети (ПСС), 270
 Потенциальные решения, 228
 Потомок, 228
 Признаки, 25

Применения ОП

- здоровоохранение, 32
- игры, 30
- интеллектуальные транспортные системы, 33
- машинное обучение, 32
- оптимизация энергопотребления, 33
- робототехника и индустрия 4.0, 31
- умные сети электроснабжения, 33
- экономика и финансы, 32

Проектирование нейронной архитектуры, 32**Проклятие размерности, 23****Р**

- Разделенная стратегия, 27
- Разреженное вознаграждение, 29
- Рандомизация предметной области, 269
- Ранжирование по приспособленности, 234
- Рекуррентная нейронная сеть (РНС), 25

С

- Сверточная нейронная сеть (СНС), 25
- Скрещивание, 229
- Сопряженных градиентов метод, 154
- Среднеквадратическая ошибка (СКО), 47
- Стохастическая стратегия, 58
- Стохастический градиентный спуск (СГС), 226
- Стратегия, 26, 126

Т

- Табличные методы, 24
- Тензоры, 43
- Теорема о градиенте стратегии, 124
- Траектория, 58

Ф

- Фенотип, 228
- Функция
 - вознаграждения, 272
 - ценности, 26, 28
 - действий, 28, 60
 - состояний, 28

Х

- Ход 37, 24
- Хромосома, 228

Ц

- Целевая стратегия, 27

Ч

- Частично наблюдаемая система, 58

Э

- Эволюционные алгоритмы (ЭА), 225, 227
 - генетические алгоритмы, 230
 - основы, 227
 - эволюционные стратегии (ЭС), 230
- СМА-ES (эволюционная стратегия с адаптацией ковариационной матрицы), 231
- Эволюционные стратегии
 - и обучение с подкреплением, 232
 - преимущества, 225, 230, 232
 - распараллеливание, 233
- Элементы ОП
 - вознаграждение, 29
 - модель, 30
 - стратегия, 26
 - функция ценности, 28
- Эпизодическая задача, 59

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@aliants-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.

Андреа Лонца

Алгоритмы обучения с подкреплением на Python

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. 23,24. Тираж 200 экз.

Отпечатано в ПАО «Т8 Издательские Технологии»
109316, Москва, Волгоградский проспект, д. 42, корпус 5

Веб-сайт издательства: www.dmkpress.com