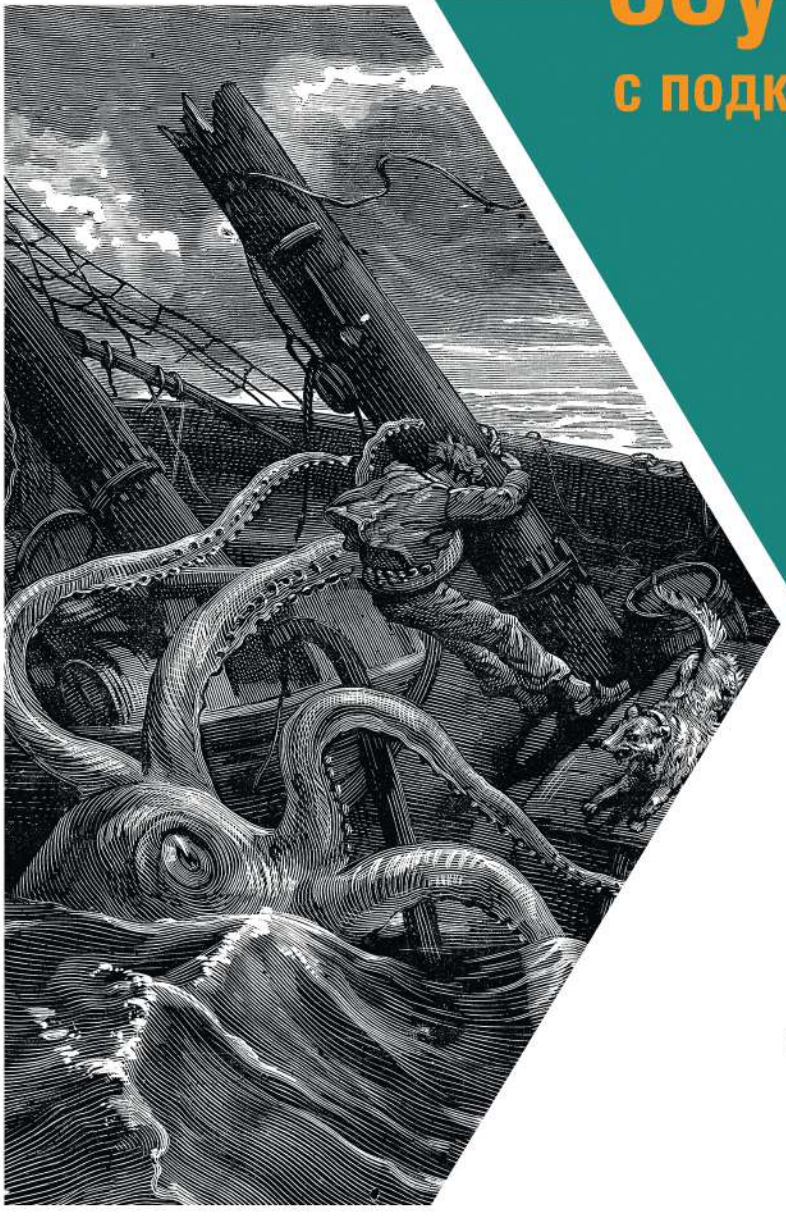
Лаура Грессер

Ван Лун Кенг

Глубокое обучение с подкреплением



Теория и практика на языке Python

Foundations of Deep Reinforcement Learning

Theory and Practice in Python

Laura Graesser
Wah Loon Keng

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Лаура Грессер, Ван Лун Кенг

Глубокое обучение
с подкреплением

Теория и практика
на языке Python

ПИТЕР®

Санкт-Петербург • Москва • Минск

2022

ББК 32.813 + 32.973.23-018
УДК 004.89
Г91

Грессер Лаура, Кенг Ван Лун

Г91 Глубокое обучение с подкреплением: теория и практика на языке Python. — СПб.: Питер, 2022. — 416 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1699-7

Глубокое обучение с подкреплением (глубокое RL) сочетает в себе два подхода к машинному обучению. В ходе такого обучения виртуальные агенты учатся решать последовательные задачи о принятии решений. За последнее десятилетие было много неординарных достижений в этой области — от однопользовательских и многопользовательских игр, таких как го и видеоигры Atari и Dota 2, до робототехники.

Эта книга — введение в глубокое обучение с подкреплением, уникально комбинирующее теорию и практику. Авторы начинают повествование с базовых сведений, затем подробно объясняют теорию алгоритмов глубокого RL, демонстрируют их реализации на примере программной библиотеки SLM Lab и напоследок описывают практические аспекты использования глубокого RL.

Руководство идеально подойдет как для студентов, изучающих компьютерные науки, так и для разработчиков программного обеспечения, которые знакомы с основными принципами машинного обучения и знают Python.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.813 + 32.973.23-018
УДК 004.89

Права на издание получены по соглашению с Pearson Education Inc. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-0135172384 англ.
ISBN 978-5-4461-1699-7

© 2020 Pearson Education, Inc.
© Перевод на русский язык ООО Издательство «Питер», 2022
© Издание на русском языке, оформление ООО Издательство «Питер», 2022
© Серия «Библиотека программиста», 2022

Краткое содержание

Часть III. Практика

Часть IV. Проектирование сред

Приложения

Оглавление

Часть I. Алгоритмы, основанные на стратегиях и полезностях

Часть II. Комбинированные методы

Часть III. Практика

Часть IV. Проектирование сред

Приложения

Тем, кто дал мне понять, что возможно все.

Лаура

Моей жене Даниэле.

Кенг

Предисловие

В апреле 2019 года боты, созданные Open AI Five, сыграли в турнире по Dota 2 против команды OG — чемпионов мира 2018 года. Dota 2 — это сложная многопользовательская игра. Игроки в ней могут выбирать разных персонажей. Для победы важны стратегия, работа в команде и быстрое принятие решений. При таком количестве переменных и кажущемся бесконечном просторе для оптимизации, создание конкурентоспособной системы искусственного интеллекта кажется непосильной задачей. И все же боты OpenAI одержали уверенную победу, а немного погодя стали побеждать в 99 % матчей против официальных игроков. В основе этого достижения лежит глубокое обучение с подкреплением.

Несмотря на то что это недавнее событие, исследования в области как обучения с подкреплением, так и глубокого обучения идут не одно десятилетие. Однако значительная часть последних изысканий вкупе с ростом мощности графических процессоров способствовали развитию возможностей современных алгоритмов. В этой книге дано введение в глубокое обучение с подкреплением и сведены в целостную систему результаты работ за последние шесть лет.

Хотя создание системы обучения компьютерной программы для победы в видеоигре, возможно, не самое важное занятие, это только начало. Обучение с подкреплением — это область машинного обучения, занимающаяся задачами последовательного принятия решений, то есть теми, решение которых занимает определенное время. Оно применимо практически в любой ситуации: в видеоиграх, на прогулке по улице или при вождении автомобиля.

Лаура Грессер и Ван Лун Кенг предложили доходчивое введение в сложную тему, играющую ведущую роль в современном машинном обучении. Мало того, что они использовали свои многочисленные публикации об исследованиях на данную тему, они создали библиотеку с открытым исходным кодом SLM Lab, призванную помочь новичкам быстро освоить глубокое машинное обучение. SLM Lab написана на Python с помощью фреймворка PyTorch, но читателям достаточно знать толь-

ко Python. Эта книга будет полезна и тем, кто собирается применять в качестве фреймворка глубокого обучения другие библиотеки, например TensorFlow. В ней они познакомятся с концепциями и формулировкой задач глубокого обучения с подкреплением.

В издании сведены воедино новейшие исследования в сфере глубокого обучения с подкреплением и даны рабочие примеры и код. Их библиотека также совместима с OpenAI Gym, Roboschool и инструментарием Unity ML-Agents, что делает книгу хорошим стартом для читателей, нацеленных на работу с этими инструментами.

Пол Дикс, редактор серии

Введение

С глубоким обучением с подкреплением (reinforcement learning, RL) мы впервые познакомились, когда DeepMind достиг беспрецедентной производительности в аркадных играх Atari. Используя лишь изображения и не располагая первоначальными знаниями о системе, агенты впервые достигли поведения уровня человека.

Идея искусственного агента, обучающегося методом проб и ошибок, самостоятельно, без учителя, поражала воображение. Это было новым впечатляющим подходом к машинному обучению и несколько отличалось от более привычного обучения с учителем.

Мы решили работать вместе над изучением этой темы. Мы читали книги и статьи, проходили онлайн-курсы, штудировали код и пытались реализовать основные алгоритмы. К нам пришло понимание того, что глубокое обучение с подкреплением сложно не только в концептуальном отношении — реализация любого алгоритма требует таких же усилий, как и большой инженерный проект.

По мере продвижения мы все больше узнавали о характерных чертах глубокого RL — взаимосвязях и различиях между алгоритмами. Формирование целостной картины модели шло с трудом, поскольку глубокое RL — новая область исследований и теоретические знания еще не были оформлены в виде книги. Нам пришлось учиться по исследовательским статьям и онлайн-лекциям.

Другой трудностью был большой разрыв между теорией и реализацией. Зачастую из-за большого количества компонентов и настраиваемых гиперпараметров алгоритмы глубокого RL капризны и ненадежны. Для успеха необходимы корректная совместная работа всех компонентов и подходящие гиперпараметры. Из теории далеко не сразу становятся понятными детали правильной реализации, но они очень важны. Ресурс, объединяющий теорию и практику, был бы неоценим во время нашего обучения.

Нам казалось, что можно найти более простой путь от теории к реализации, который облегчил бы изучение глубокого RL. Данная книга воплощает нашу попытку сделать это. В ней введение в глубокое RL проходит через все стадии: сначала интуитивное понимание, затем объяснение теории и алгоритмов, а в конце реализация и практические рекомендации. В связи с этим к книге прилагается фреймворк SLM Lab, содержащий реализации всех рассматриваемых алгоритмов. Если кратко, это книга, от которой мы не отказались бы в начале своего обучения.

Глубокое RL относится к области обучения с подкреплением. В основе обучения с подкреплением зачастую лежит аппроксимация функции, в глубоком RL аппроксиматоры обучаются посредством глубоких нейронных сетей. Обучение с подкреплением, обучение с учителем и обучение без учителя — три основные методики машинного обучения, каждая из которых отличается формулировкой задач и обучением алгоритмов по данным.

В этой книге мы фокусируемся исключительно на глубоком RL, так как проблемы, с которыми нам приходилось сталкиваться, относятся к данному разделу обучения с подкреплением. Это накладывает на книгу два ограничения. Во-первых, исключаются все прочие методики для обучения аппроксиматоров в обучении с подкреплением. Во-вторых, выделяются исследования между 2013 и 2019 годами, хотя обучение с подкреплением существует с 1950-х годов. Многие современные достижения основаны на более ранних исследованиях, поэтому нам показалось важным проследить развитие основных идей. Однако мы не собираемся описывать всю историю области.

Книга предназначена для студентов, изучающих компьютерные науки, и разработчиков программного обеспечения. Она задумывалась как введение в глубокое RL и не требует предварительных знаний о предмете. Однако предполагается, что читатель имеет базовое представление о машинном обучении и глубоком обучении и программирует на Python на среднем уровне. Опыт работы с PyTorch также полезен, но не обязателен.

Книга организована следующим образом. В главе 1 дается введение в различные аспекты проблем глубокого RL и обзор его алгоритмов.

Часть I книги посвящена алгоритмам, основанным на полезностях и стратегии. В главе 2 рассматривается первый метод градиента стратегии, известный как REINFORCE. В главе 3 описывается первый основанный на полезностях метод под названием SARSA. В главе 4 обсуждаются алгоритмы глубоких Q-сетей (Deep Q-Networks, DQN), а глава 5 сфокусирована на улучшающих их методиках — контрольных сетях (target networks), алгоритмах двойной DQN и приоритизированной памяти прецедентов (Prioritized Experience Replay).

В части II основное внимание уделяется алгоритмам, объединяющим методы, основанные на полезностях и стратегии. В главе 6 представлен алгоритм

актора-критика, расширяющий REINFORCE. В главе 7 приведена оптимизация ближайшей стратегии (Proximal Policy Optimization, PPO), которой может быть расширен метод актора-критика. В главе 8 обсуждаются приемы синхронной и асинхронной параллелизации, применимые ко всем алгоритмам в этой книге. В заключение в главе 9 подводятся итоги по всем алгоритмам.

Все главы об алгоритмах построены по одной схеме. Сначала приводятся основные концепции и прорабатываются соответствующие математические формулировки. Затем идет описание алгоритма и обсуждаются реализации на Python. В конце приводится полный алгоритм с подобранными гиперпараметрами, который может быть запущен в SLM Lab, а основные характеристики алгоритма иллюстрируются графиками.

В части III основной упор сделан на практических деталях реализации алгоритмов глубокого RL. В главе 10 описываются практики проектирования и отладки. В нее включены также справочник гиперпараметров и результаты сравнения производительности алгоритмов. В главе 11 представлены основные команды и функции библиотеки SLM Lab, прилагаемой к книге. В главе 12 рассматривается проектирование нейронных сетей, а в главе 13 обсуждается аппаратное оборудование.

В части IV книги речь идет о проектировании среды. В нее входят главы 14, 15, 16 и 17, в которых рассматриваются способы задания состояний, действий, вознаграждений и функций переходов соответственно.

Читать по порядку следует главы с 1-й по 10-ю. В них дано введение во все алгоритмы книги и представлены практические рекомендации, как их запустить. В следующих трех главах, с 11-й по 13-ю, внимание уделяется более специфическим темам, и их можно читать в любом порядке. Для тех, кто не хочет вдаваться в такие подробности, достаточно прочитать главы 1, 2, 3, 4, 6 и 10, в которых последовательно изложены несколько алгоритмов. В конце, в части IV, помещены отдельные главы, предназначенные для читателей, особо заинтересованных в более глубоком понимании использованных сред или построении собственных сред.

Прилагаемая к книге библиотека программного обеспечения SLM Lab — это модульный фреймворк глубокого RL, созданный с помощью PyTorch. SLM — это аббревиатура для Strange Loop Machine (машина на странных петлях), названия, данного в честь знаменитой книги Ховштадтера «Гёдель, Эшер, Бах: эта бесконечная гирлянда»[1]. Во включенных в книгу примерах из SLM Lab используются синтаксис PyTorch и функции для обучения нейронных сетей. Однако основные принципы реализации алгоритмов глубокого RL применимы и к другим фреймворкам глубокого обучения, таким как TensorFlow.

Для того чтобы новичкам было легче освоить глубокое RL, компоненты SLM Lab разбиты на концептуально понятные части. Кроме того, чтобы упростить переход

[1] Gödel, Escher, Bach: An Eternal Golden Braid.

от теории к коду, они упорядочены в соответствии с изложением глубокого RL в академической литературе.

Другой важный аспект изучения глубокого RL — эксперименты. С этой целью в SLM Lab представлен фреймворк для экспериментов, призванный помочь начинающим в создании и тестировании собственных гипотез.

Библиотека SLM Lab — это проект с открытым исходным кодом на Github. Мы рекомендуем установить ее (на машины с Linux или MacOS) и запустить первое демо, следуя инструкциям, приведенным на сайте репозитория https://github.com/kengz/SLM-Lab. В Git была создана специальная ветка book с версией кода, совместимой с этой книгой. В листинге 0.1 приведена краткая инструкция по установке, скопированная с сайта репозитория.

Листинг 0.1. Установка SLM-Lab с ветки book

```
# клонируйте репозиторий
git clone https://github.com/kengz/SLM-Lab.git
cd SLM-Lab
# переключитесь на специальную ветку для этой книги
git checkout book
# установите зависимости
./bin/setup
# далее следуйте инструкциям, приводимым на сайте репозитория
```

Рекомендуется сначала задать эти настройки, чтобы можно было обучать агентов по алгоритмам в соответствии с их появлением в книге. Помимо установки и запуска первого примера, необходимости в знакомстве с SLM Lab до прочтения глав об алгоритмах (части I и II) не возникнет: все команды для обучения агентов даны там, где нужно. Кроме того, SLM Lab обсуждается более подробно в главе 11, после перехода от алгоритмов к практическим аспектам глубокого обучения с подкреплением.

Благодарности

Завершить этот проект нам помогло немало людей. Спасибо Милану Цвитковичу, Алексу Лидсу, Навдипу Джайтли, Джону Крону, Кате Василаки и Кейтлин Глисон за поддержку и вдохновение. Выражаем благодарность OpenAI, PyTorch, Илье Кострикову и Джамромиру Дженишу за предоставление высококачественной реализации с открытым исходным кодом различных компонентов алгоритмов глубокого RL. Благодарим также Артура Джулиани за предварительные обсуждения структуры сред. Эти ресурсы и обсуждения были бесценны при создании SLM Lab.

Хотелось бы поблагодарить Александра Саблайроллеса, Ананта Гупта, Брендона Стрикланда, Чонга Ли, Джона Крона, Джорди Франка, Кэтрин Джаясурия, Мэтью Ратца, Пидонга Вонга, Раймонда Чуа, Регину Р. Монако, Рико Джоншковски, Софи Табак и Утку Эвси за обстоятельные замечания о ранних набросках этой книги.

Мы очень признательны производственной команде Pearson — Алине Кирсановой, Крису Зану, Дмитрию Кирсанову и Джулии Нахил. Благодаря вашему профессионализму, старанию и вниманию к деталям текст стал гораздо лучше.

Наконец, эта книга не появилась бы на свет без нашего редактора Деборы Вильямс Коли. Благодарим за вашу помощь и терпение.

Об авторах

Лаура Грессер работает разработчиком исследовательского программного обеспечения для робототехнических систем в Google. Получила степень магистра компьютерных наук в Нью-Йоркском университете со специализацией в машинном обучении.

Ван Лун Кенг — разработчик систем искусственного интеллекта в Machine Zone, использует глубокое обучение с подкреплением для решения проблем промышленного производства. Специалист в области теоретической физики и компьютерных наук.

Они вместе разработали две библиотеки программного обеспечения для глубокого обучения с подкреплением и выпустили ряд лекций и учебников по этой теме.

От издательства

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

Мы будем рады узнать ваше мнение!

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

О научном редакторе русскоязычного издания

Александр Игоревич Панов, кандидат физико-математических наук, доцент, заведующий отделом ФИЦ ИУ РАН, руководитель Центра когнитивного моделирования МФТИ, ведущий научный сотрудник Института искусственного интеллекта (AIRI), член Научного совета Российской ассоциации искусственного интеллекта. Специалист в области когнитивной робототехники, обучения с подкреплением, общего искусственного интеллекта.

1 Введение в обучение с подкреплением

В этой главе вводятся основные концепции обучения с подкреплением. Сначала рассматриваются простые примеры, которые позволят вам интуитивно понимать основные компоненты, используемые в обучении с подкреплением, — агента и среду.

В частности, мы рассмотрим процесс оптимизации целевой функции посредством взаимодействия агента со средой. Далее дано более формальное определение, а обучение с подкреплением объясняется с помощью понятия марковского процесса принятия решений. Это теоретические основы обучения с подкреплением.

Затем представлены три основные функции, которые должен выучить агент: *стратегия*, *функции полезности* и *модель среды*. Далее показано, как разные варианты обучения этих функций порождают различные семейства алгоритмов глубокого обучения с подкреплением.

В конце дан краткий обзор основного метода глубокого обучения — метода аппроксимации функций, который используется на протяжении всей книги. Также здесь обсуждаются основные различия между обучением с подкреплением и обучением с учителем.

1.1. Обучение с подкреплением

Обучение с подкреплением (reinforcement learning, RL) занимается задачами последовательного принятия решений. Многие реальные проблемы, возникающие в компьютерных играх, спорте, вождении автомобиля, оптимизации товарных запасов, роботизированном управлении, то есть везде, где действуют люди и машины, могут быть представлены в подобном виде.

Решая каждую из этих задач, мы преследуем какую-то цель: победить в игре, безопасно доехать до пункта назначения или минимизировать стоимость строительных материалов. Мы предпринимаем действия и получаем из окружающего мира ответ о том, насколько близки к цели: текущий счет, расстояние до пункта назначения или цену одного изделия. Достижение цели, как правило, подразумевает

выполнение ряда действий, каждое из которых изменяет окружающий мир. Мы наблюдаем эти изменения мира, а также получаем обратную информацию, опираясь на которую принимаем решение о следующем шаге.

Представьте, что вы на вечеринке, а ваш друг принес флагшток и предлагает на спор как можно дольше балансировать им, поставив на ладонь. Если вы никогда до сих пор не делали этого, то первоначальные попытки будут не слишком удачными. Вероятно, первые несколько минут вы потратите на то, чтобы методом проб и ошибок почувствовать флагшток, ведь он все время падает.

Эти ошибки позволят накопить полезную информацию и приобрести интуитивное понимание того, как удерживать флагшток в равновесии. Вы узнаете, где находится его центр масс, с какой скоростью он наклоняется, при каком угле наклона падает, как быстро вы может подстроиться и т. д. Вы используете эту информацию, чтобы внести коррективы при следующих попытках, совершенствуетесь и снова корректируете свое поведение. Вы даже не заметите, как начнете удерживать равновесие по 5, 10, 30 с, 1 мин и т. д.

Этот процесс наглядно демонстрирует, как работает обучение с подкреплением. В обучении с подкреплением вас можно назвать агентом, а флагшток и ваше окружение — средой. Фактически первая среда, задачу которой мы научимся решать с помощью обучения с подкреплением, — это игровая версия данного сценария под названием CartPole (рис. 1.1). Агент управляет скользящей вдоль оси тележкой так, чтобы удерживать стержень в вертикальном положении в течение заданного времени. Реальные способности людей гораздо шире, ведь мы можем интуитивно понимать физическую сторону происходящего. А можем и применить навыки выполнения схожих заданий, таких как балансирование подносом, уставленным напитками. Но, по сути, формулировка задачи остается той же самой.

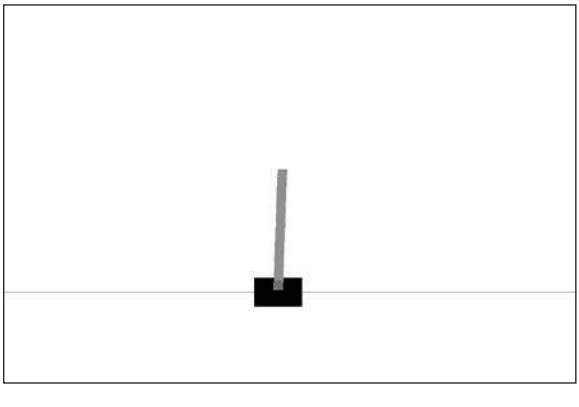


Рис. 1.1. CartPole-v0 — простая игровая среда. Цель — удержание в равновесии стержня на протяжении 200 шагов посредством управления перемещениями тележки вправо и влево

В обучении с подкреплением изучаются подобного рода задачи, а также методы, с помощью которых искусственные агенты учатся их решать. Это область искусственного интеллекта, которая восходит к теории оптимального управления и использует понятие марковского процесса принятия решений (МППР). RL появился в 1950-х годах в контексте динамического программирования и квазилинейных уравнений благодаря Ричарду Беллману. Его имя будет еще не раз упомянуто при изучении получившего известность в обучении с подкреплением уравнения Беллмана.

Задачи RL могут быть представлены как система, состоящая из агента и среды. Среда предоставляет информацию, описывающую состояние системы. Агент взаимодействует со средой, наблюдая состояние и используя данную информацию при выборе *действия*. Среда принимает действие и переходит в следующее состояние, а затем возвращает агенту следующее состояние и *вознаграждение*. Когда цикл *«состояние → действие → вознаграждение»* завершен, предполагается, что сделан один шаг. Цикл повторяется, пока среда не завершится, например, когда задача решена. Полностью этот процесс показан на диаграмме цикла управления (рис. 1.2).

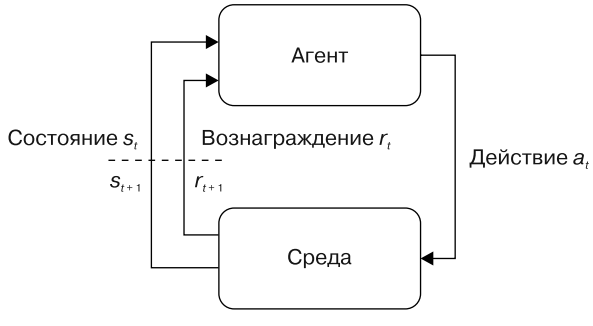


Рис. 1.2. Цикл управления агентом в обучении с подкреплением

Функция, в соответствии с которой агент выбирает действия, называется *стратегией*. Формально стратегия — это функция, отображающая множество состояний в множество действий. Действие изменяет среду и влияет на то, что агент наблюдает и делает дальше. Обмен информацией между агентом и средой разворачивается во времени, однако его можно рассматривать как процесс последовательного принятия решений.

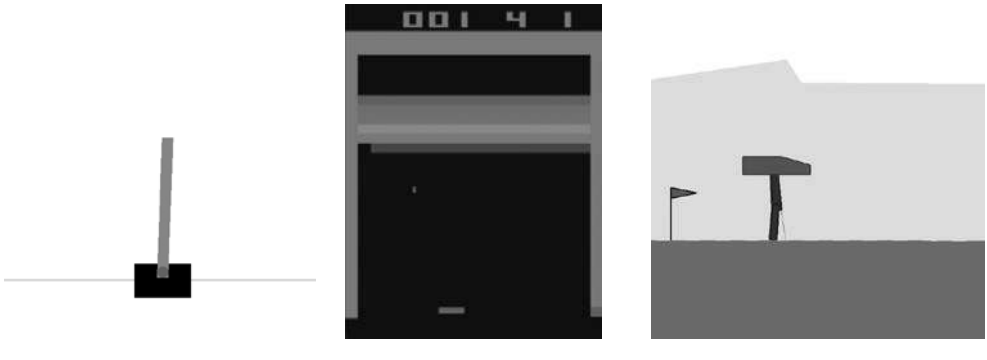
В задачах RL есть *целевая функция*, которая является суммой полученных агентом вознаграждений. Задача агента — максимизировать целевую функцию, выбирая наилучшие действия. Он *учится* этому, взаимодействуя со средой методом проб и ошибок, и использует поощряющие сигналы для *подкрепления* лучших действий.

Агент и среда определены как взаимоисключающие сущности, чтобы мы могли четко отделить друг от друга состояния, действия и вознаграждения. Среду можно

рассматривать как все, что не является агентом. Например, для езды на велосипеде возможны несколько различных, но равнозначных определений агента и среды. Если рассматривать все человеческое тело как агента, который наблюдает свое окружение, а производимые им напряжения мышц — как действия, то среда — это дорога и велосипед. Если считать агентом мыслительный процесс, то средой будут физическое тело, велосипед и дорога, действиями — нервные импульсы, пересылаемые от головного мозга к мускулам, а состояниями — поступающие в мозг сигналы от органов чувств.

По сути, система обучения с подкреплением реализует цикл управления с обратной связью, где агент и среда взаимодействуют и обмениваются сигналами, причем агент пытается максимизировать целевую функцию. Сигналы — это тройка (s_t, a_t, r_t), что соответствует состоянию, действию и вознаграждению, а индекс t указывает на номер шага (момент времени), на котором возник сигнал. Кортеж (s_t, a_t, r_t) называется *прецедентом* или частью получаемого агентом опыта. Цикл управления может повторяться до бесконечности[1] или закончиться по достижении либо конечного состояния, либо максимального значения шага $t = T$. Временной горизонт от $t = 0$ до момента завершения среды носит название *эпизода*. *Траектория* — это последовательность прецедентов, или часть опыта, накопленного в течение эпизода, $\tau = (s_0, a_0, r_0), (s_1, a_1, r_1)$... Обычно агенту для обучения хорошей стратегии требуется от сотен до миллионов эпизодов в зависимости от сложности задачи.

Рассмотрим для обучения с подкреплением три примера сред (рис. 1.3) и определения состояний, действий и вознаграждений. Все эти среды можно получить в OpenAI Gym — библиотеке с открытым исходным кодом, которая предоставляет стандартизованный набор сред.



а) CartPole б) Atari Breakout в) BipedalWalker

Рис. 1.3. Три примера сред с разными состояниями, действиями и вознаграждениями

[1] Бесконечные циклы управления существуют в теории, но не на практике. Как правило, для среды устанавливается максимальное количество шагов T.

CartPole (см. рис. 1.3, *а*) — одна из простейших сред для обучения с подкреплением, была впервые описана Барто, Саттоном и Андерсоном в 1983 году. В этой среде стержень закреплен на тележке, которая может двигаться по дорожке без трения. Основные особенности среды приведены далее.

1. **Цель** — удерживать стержень в вертикальном положении в течение 200 шагов.
2. **Состояние** — массив из четырех элементов: [позиция тележки, скорость тележки; угол наклона стержня; угловая скорость стержня], например [–0,034; 0,032; –0,031; 0,036].
3. **Действие** — целое число: 0 при перемещении тележки на фиксированное расстояние влево и 1 — при перемещении на фиксированное расстояние вправо.
4. **Вознаграждение** равно +1 на каждом шаге, на котором стержень остается в вертикальном положении.
5. Среда **завершается** либо при падении стержня (отклонение от вертикали на угол больше 12°), либо при выходе тележки за пределы экрана, либо при достижении максимального числа шагов, равного 200.

Atari Breakout (см. рис. 1.3, *б*) — это старая консольная аркадная игра, в которой есть мячик, расположенная внизу экрана платформа и блоки. Цель — попадать в блоки и разрушать их, отбивая мячик платформой. В начале игры у игрока пять жизней, одна из которых теряется, когда мячик попадает мимо платформы.

1. **Цель** — набрать максимальный счет в игре.
2. **Состояние** — цифровое изображение в формате RGB с разрешением 160×210 пикселов, то есть то, что мы видим на экране игры.
3. **Действие** — целое число из набора {0, 1, 2, 3}, которое сопоставляется игровым действиям {бездействие, запустить мячик, перемещение вправо, перемещение влево}.
4. **Вознаграждение** — разность в счете между двумя идущими друг за другом состояниями.
5. Среда **завершается**, когда все жизни потеряны.

BipedalWalker (см. рис. 1.3, *в*) — задача непрерывного управления, где агент-робот сканирует окрестности с помощью датчика-лидара и старается удержаться от падения при движении вправо.

1. **Цель** — идти вправо не падая.
2. **Состояние** — массив из 24 элементов: [угол наклона корпуса; угловая скорость корпуса; скорость по оси *X*; скорость по оси *Y*; угол поворота шарнира бедра 1; скорость шарнира бедра 1; угол поворота шарнира колена 1; скорость шарнира колена 1; есть ли контакт с землей ноги 1; угол поворота шарнира бедра 2; скорость шарнира бедра 2; угол поворота шарнира колена 2; скорость

шарнира колена 2; есть ли контакт с землей ноги 2... 10 показаний лидара], например [2,745e–03; 1,180e–05; –1,539e–03; –1,600e–02... 7,091e–01; 8,859e–01; 1,000e+00; 1,000e+00].

3. **Действие** — вектор из четырех чисел с плавающей запятой со значениями в интервале [–1,0; 1,0], имеющий вид [крутящий момент и скорость бедра 1; крутящий момент и скорость колена 1; крутящий момент и скорость бедра 2; крутящий момент и скорость колена 2], например [0,097; 0,430; 0,205; 0,089].
4. **Вознаграждение** — за движение вправо максимально до +300 и –100 за падение робота. Кроме того, за каждый шаг дается небольшое отрицательное вознаграждение (плата за движение), пропорциональное приложенному крутящему моменту.
5. Среда **завершается**, когда тело робота касается земли, либо при достижении цели, расположенной справа, либо после максимального количества шагов, равного 1600.

Эти среды демонстрируют различные варианты того, в каком виде могут быть представлены состояния и действия. В CartPole и BipedalWalker состояния — это векторы, кодирующие свойства, такие как позиции и скорости. В Atari Breakout состояние — это изображение экрана игры. В CartPole и Atari Breakout действия являются одиночными дискретными целыми числами, тогда как в BipedalWalker действие — это вектор из четырех непрерывных вещественных значений. Вознаграждение всегда представлено скалярной величиной, но пределы его значений варьируются в зависимости от задания.

Просмотрев несколько примеров, можно ввести формальные обозначения для состояний, действий и вознаграждений.

Состояние:

$$s_t \in S, \tag{1.1}$$

где S — пространство состояний.

Действие:

$$a_t \in A, \tag{1.2}$$

где A — пространство действий.

Вознаграждение:

$$r_t = R(s_t, a_t, s_{t+1}), \tag{1.3}$$

где R — функция вознаграждения.

Пространство состояний S — это набор из всех возможных в среде состояний. В зависимости от среды оно может быть определено как набор целых или вещественных чисел, векторов, матриц, структурированных или неструктурированных данных. Аналогично пространство действий A — это набор всех возможных действий, определяемых средой. Оно также может иметь разный вид, но чаще всего действие определяется как скалярная величина или вектор. Функция вознаграждения $R(s_t, a_t, s_{t+1})$ присваивает положительное, отрицательное или равное нулю скалярное значение каждому переходу (s_t, a_t, s_{t+1}). Пространство состояний, пространство действий и функция вознаграждения задаются средой. Вместе они составляют кортежи (s, a, r), являющиеся основными информационными единицами при описании систем обучения с подкреплением.

1.2. Обучение с подкреплением как МППР

Теперь рассмотрим, как среда переходит из одного состояния в другое с помощью так называемой *функции переходов*. В обучении с подкреплением функция переходов — это основной компонент марковского процесса принятия решений (МППР), который является математической основой моделирования последовательного принятия решений.

Чтобы понять, как функции переходов связаны с МППР, рассмотрим общую постановку задачи, приведенную в следующем выражении:

$$s_{t+1} \sim P(s_{t+1} \mid (s_0, a_0), (s_1, a_1) \ldots (s_t, a_t)). \tag{1.4}$$

В выражении (1.4) утверждается, что на временном шаге t следующее состояние s_{t+1} берется из распределения вероятностей P, обусловленного всей историей. Вероятность перехода среды из состояния s_t в состояние s_{t+1} зависит от всех предыдущих состояний s и действий a, которые имели место в данном эпизоде до этого момента. Записать функцию переходов в подобной форме сложно, особенно если эпизоды растягиваются на большое количество шагов. При проектировании таких функций переходов нужно учитывать огромное количество комбинаций факторов, возникавших в течение всех предыдущих шагов. Кроме того, в такой формулировке становится очень сложной функция выбора действий агентом — его стратегия. Поскольку для понимания того, как действие может изменить следующее состояние среды, важна вся история состояний и действий, то агенту придется принимать во внимание всю эту информацию, принимая решение о выборе действия.

Чтобы функция переходов среды стала лучше реализуемой на практике, преобразуем ее в МППР. Сделаем такое предположение: переход в следующее состояние s_{t+1} зависит только от предыдущих значений состояния s_t и действия a_t. Данное

предположение известно как *марковское свойство*. С учетом этого функция переходов примет следующий вид:

$$s_{t+1} \sim P(s_{t+1} \mid s_t, a_t). \tag{1.5}$$

Выражение (1.5) гласит, что следующее состояние s_{t+1} берется из распределения вероятностей $P(s_{t+1} \mid s_t, a_t)$. Это упрощенная форма первоначальной функции переходов. Марковское свойство подразумевает, что текущие состояние и действие на шаге t содержат достаточно информации, чтобы в полной мере определить вероятность перехода в следующее состояние на шаге $t + 1$.

Несмотря на простоту данной формулировки, она довольно эффективна. В подобной форме могут быть выражены многие процессы, такие как игры, управление робототехническими системами и планирование. Это связано с тем, что определение состояния может включать любую информацию, позволяющую сделать функцию переходов марковской.

Рассмотрим пример с последовательностью чисел Фибоначчи, описываемой формулой $s_{t+1} = s_t + s_{t-1}$, где каждый член s_t рассматривается как состояние. Чтобы эта функция стала марковской, переопределим состояние как $\boldsymbol{s}'_t = [s_t, s_{t-1}]$. Теперь состояние содержит достаточно информации для расчета следующего элемента последовательности. В более общей форме эта стратегия может быть применена к любой системе с конечным набором k последовательных состояний, содержащих достаточно сведений для перехода в следующее состояние. В примечании 1.1 более подробно описано определение состояний в МППР и в его обобщенной форме — частично наблюдаемом МППР. Обратите внимание: на протяжении всей книги встречаются примечания, где вопросы рассматриваются углубленно. При первом прочтении их можно пропустить — это не помешает понять основную тему.

Примечание 1.1. МППР и частично наблюдаемые МППР

До сих пор понятие состояния применялось в двух случаях. С одной стороны, состояние — это то, что порождается средой и что наблюдает агент. Назовем его *наблюдаемым состоянием* s_t. С другой стороны, состояние — это то, что используется функцией переходов. Назовем его *внутренним состоянием* $\boldsymbol{s}_t^{\text{inn}}$ среды.

В МППР $s_t = \boldsymbol{s}_t^{\text{inn}}$, то есть наблюдаемое состояние идентично внутреннему состоянию среды. Агенту доступна информация, которая используется для перехода среды в следующее состояние.

Это не всегда верно. Наблюдаемое состояние может отличаться от внутреннего состояния среды, $s_t \neq \boldsymbol{s}_t^{\text{inn}}$. В этом случае среда описывается как *частично наблюдаемый МППР*, так как предоставляемое агенту состояние *st* содержит лишь часть информации о состоянии среды.

В этой книге в большинстве случаев данное различие не учитывается и предполагается, что $s_t = \boldsymbol{s}_t^{\text{inn}}$. Однако знать о частично наблюдаемых МППР важно по двум

причинам. Во-первых, некоторые рассматриваемые примеры среды не являются идеальными МППР. Например, в среде Atari наблюдаемое состояние s_t — это одно изображение в формате RGB, в котором передается информация о позиции объекта, количестве жизней агента и т. д., но нет скоростей объекта. Скорости будут включены во внутреннее состояние среды, поскольку они требуются для определения следующего состояния, заданного действием. Тогда для достижения высокой производительности нужно будет преобразовать s_t так, чтобы оно содержало больше сведений. Это обсуждается в главе 5.

Во-вторых, многие интересные реальные проблемы по своей сути являются частично наблюдаемыми МППР, в их число входят случаи проявления ограниченности датчиков или данных, ошибок моделирования и зашумленности среды. Детальное рассмотрение частично наблюдаемых МППР лежит за пределами этой книги, но они будут вкратце затронуты при обсуждении архитектуры нейронных сетей в главе 12.

Наконец, при рассмотрении структуры состояний в главе 14 различие между s_t и s_t^{inn} будет иметь большое значение, поскольку агент обучается по s_t. Информация, которая содержится в s_t, и степень его отличия от s_t^{inn} влияют на то, насколько сложным или простым будет решение задачи.

Теперь можно представить формулировку задачи обучения с подкреплением в виде МППР. МППР определяется кортежем из четырех элементов — S, A, $P(.)$, $\mathcal{R}(.)$, где:

- S — набор состояний;
- A — набор действий;
- $P(s_{t+1} \mid s_t, a_t)$ — функция перехода состояний среды;
- $\mathcal{R}(s_t, a_t, s_{t+1})$ — функция вознаграждения среды.

Рассматриваемые в этой книге задачи обучения с подкреплением используют одно важное предположение: функция переходов $P(s_{t+1} \mid s_t, a_t)$ и функция вознаграждений $\mathcal{R}(s_t, a_t, s_{t+1})$ недоступны для агентов. Они могут получить информацию об этих функциях только через состояния, действия и вознаграждения, воздействию которых подвергаются на данный момент в среде, то есть через кортежи (s_t, a_t, r_t).

Чтобы формулировка задачи была полной, нужно формализовать понятие максимизируемой агентом целевой функции. Во-первых, определим *отдачу*[1] (return) $R(\tau)$, используя траекторию из эпизода $\tau = (s_0, a_0, r_0) \ldots (s_T, a_T, r_T)$:

$$R(\tau) = r_{t1} + \gamma r_1 + \gamma^2 r_2 + \ldots + \gamma^T r_T = \sum_{t=0}^{T} \gamma^t r_t. \tag{1.6}$$

В уравнении (1.6) отдача определена как дисконтированная сумма вознаграждений на траектории, где γ — коэффициент дисконтирования, $\gamma \in [0, 1]$.

[1] Отдача обозначается R, а $\mathcal{R}$ оставлено для функции вознаграждения.

Тогда *целевая функция* $J(\tau)$ становится просто математическим ожиданием отдачи по нескольким траекториям:

$$J(\tau) = \mathbb{E}_{\tau \sim \pi}\left[R(\tau)\right] = \mathbb{E}_{\tau}\left[\sum_{t=0}^{T} \gamma^t r_t\right]. \tag{1.7}$$

Отдача $R(\tau)$ — это сумма дисконтированных вознаграждений $\gamma^t r_t$ за все временные шаги $t = 0... T$. Целевая функция $J(\tau)$ — это отдача, усредненная по нескольким эпизодам. Математическое ожидание предполагает случайный характер выбора действий и поведения среды, то есть при повторных запусках отдача не всегда будет одинаковой. Максимизация целевой функции — то же самое, что и максимизация отдачи.

Коэффициент дисконтирования $\gamma \in [0, 1]$ — важная переменная, влияющая на то, как оцениваются будущие вознаграждения. Чем меньше γ, тем меньший вес имеют будущие вознаграждения, что ведет к «близорукости» агента. В крайнем случае, когда $\gamma = 0$, целевая функция рассматривает только начальное вознаграждение r_0, как показано в уравнении (1.8):

$$R(\tau)_{\gamma=0} = \sum_{t=0}^{T} \gamma^t r_t = r_0. \tag{1.8}$$

Чем больше значение γ, тем больший вес придается вознаграждениям на будущих временных шагах: агент становится «дальнозорким». Если $\gamma = 1$, вознаграждения на всех шагах имеют одинаковый вес, как показано в уравнении (1.9):

$$R(\tau)_{\gamma=1} = \sum_{t=0}^{T} \gamma^t r_t = \sum_{t=0}^{T} r_t. \tag{1.9}$$

Для задач с *бесконечным* временным горизонтом нужно устанавливать $\gamma < 1$, чтобы целевая функция не стала бесконечной. Для задач с *конечным* временным горизонтом γ — важный параметр, так как в зависимости от используемого значения коэффициента дисконтирования решение может стать проще или сложнее. Пример этого будет рассмотрен в главе 2.

Определив целевую функцию и представив задачу обучения с подкреплением как МППР, можно выразить цикл управления агентом (рис. 1.2) как цикл управления в МППР (алгоритм 1.1).

Алгоритм 1.1. Цикл управления МППР

```
Считаем, что агент и среда env уже созданы
for episode = 0,...,MAX_EPISODE do
    state = env.reset()
    agent.reset()
    for t = 0,...,T do
        action = agent.act(state)
        state, reward = env.step(action)
        agent.update(action, state, reward)
```

```
 9:         if env.done() then
10:             break
11:         end if
12:     end for
13: end for
```

Алгоритм 1.1 показывает взаимодействие между агентом и средой на протяжении множества эпизодов и некоторого количества шагов. В начале каждого эпизода среда и агент возвращаются к исходным позициям (строки 3 и 4), при этом среда порождает начальное состояние. Затем начинается их взаимодействие: агент выполняет действие, исходя из данного состояния (строка 6), затем, основываясь на этом действии, среда производит следующее состояние и вознаграждение (строка 7), переходя на следующий шаг. Цикл `agent.act-env.step` длится, пока не будет достигнуто максимальное количество шагов T либо среда не завершится. Здесь появляется новый компонент — `agent.update` (строка 8), который включает в себя алгоритм обучения агента. На протяжении большого количества шагов и эпизодов этот метод накапливает данные и на внутреннем уровне обучается максимизации целевой функции.

Этот алгоритм общий для всех задач обучения с подкреплением, поскольку он определяет непротиворечивый интерфейс взаимодействия между агентом и средой. Данный интерфейс служит основой для реализации многих алгоритмов обучения с подкреплением, включенных в унифицированный фреймворк, как будет видно в прилагаемой к книге библиотеке SLM Lab.

1.3. Обучаемые функции в обучении с подкреплением

Когда обучение с подкреплением сформулировано как МППР, возникает естественный вопрос: чему должен учиться агент?

Мы видели, что агент может сформировать функцию выбора действий, известную как *стратегия*. Однако у среды есть и другие свойства, которые могут быть полезны для агента. В частности, существует *три основных функции*, которые изучаются в обучении с подкреплением.

1. Стратегия π, которая сопоставляет состоянию действие: $a \sim \pi(s)$.
2. Функция полезности $V^{\pi}(s)$ или $Q^{\pi}(s, a)$ для вычисления ожидаемой отдачи $\mathbb{E}_{\tau}\left[R(\tau)\right]$.
3. Модель среды[1] $P(s' \mid s, a)$.

[1] Чтобы сделать нотацию более краткой, принято записывать пару следующих друг за другом кортежей (s_t, a_t, r_t), $(s_{t+1}, a_{t+1}, r_{t+1})$ как (s, a, r), (s', a', r'), где штрих означает следующий шаг. Это обозначение будет встречаться в книге повсеместно.

Стратегия π — это то, каким образом агент производит действия в среде, чтобы максимизировать целевую функцию. Согласно циклу управления агент должен производить действия на каждом шаге после наблюдения состояния *s*. Стратегия имеет фундаментальное значение для цикла управления, поскольку генерирует действия, которые заставляют его продолжаться.

Стратегия может быть стохастической. Это значит, что она может с определенной вероятностью давать на выходе разные действия для одного состояния. Это может быть записано как $\pi(a \mid s)$ и означает вероятность действия a для данного состояния *s*. Действие, выбранное по стратегии, записывается как $a \sim \pi(s)$.

Функции полезностей представляют информацию о цели. Они помогают агенту понять, насколько хороши состояния и доступные действия с точки зрения ожидаемой отдачи. Они записываются в двух формах:

$$V^{\pi}(s) = \mathbb{E}_{s_0 = s,\, \tau \sim \pi}\left[\sum_{t=0}^{T} \gamma^t r_t\right]; \tag{1.10}$$

$$Q^{\pi}(s,a) = \mathbb{E}_{s_1 = s,\, a_0 = a,\, \tau \sim \pi}\left[\sum_{t=0}^{T} \gamma^t r_t\right]. \tag{1.11}$$

Функция полезности V^{π}, приведенная в уравнении (1.10) оценивает, насколько хорошим или плохим является состояние. V^{π} дает оценку ожидаемой отдачи от пребывания в положении *s*, предполагая, что агент продолжает действовать в соответствии со своей текущей стратегией π. Отдача $R(\tau) = \sum_{t=0}^{T} \gamma^t r_t$ подсчитывается, начиная с текущего состояния *s* и до конца эпизода. Это прогнозная оценка, поскольку не учитываются все вознаграждения, полученные до состояния *s*.

Рассмотрим простой пример, который позволит получить представление о функции полезности V^{π}. На рис. 1.4 изображена дискретная среда с конечным числом состояний, в которой агент может перемещаться из клетки в клетку по вертикали и горизонтали. Каждая клетка — это состояние, с которым связано вознаграждение, как показано на рис. 1.4, *слева*. Среда завершается, когда агент попадает в целевое состояние с вознаграждением $r = +1$.

Справа показаны полезности $V^{\pi}(s)$, рассчитанные для каждого состояния по вознаграждениям с помощью уравнения (1.10) при $\gamma = 0{,}9$. Функция полезности V^{π} всегда зависит от конкретной стратегии π. В этом примере взята стратегия π, при которой всегда выбирается кратчайший путь к целевому состоянию. Если стратегия будет другой — к примеру, перемещение только вправо, — то значения полезностей будут иными.

Это показывает, что функция полезности является прогнозной и помогает агенту различать состояния с одинаковым вознаграждением. Чем ближе агент к целевому состоянию, тем выше полезность рассматриваемого состояния.

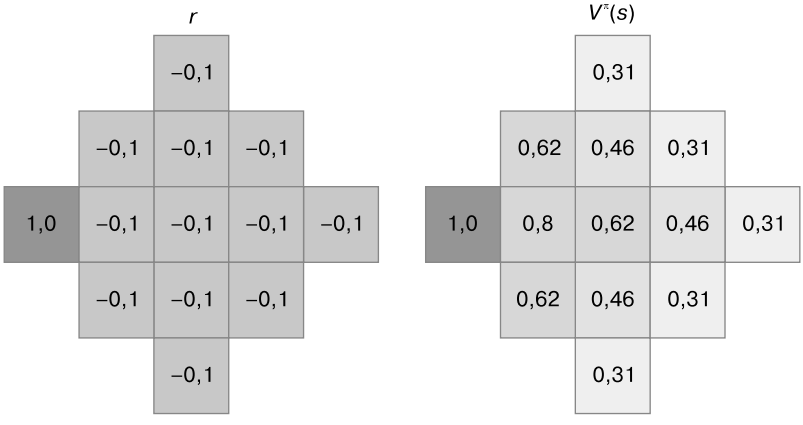


Рис. 1.4. Вознаграждения r и полезности $V^{\pi}(s)$ для каждого состояния s в простой клеточной среде. Полезность состояния рассчитывается по вознаграждениям с помощью уравнения (1.10) при $\gamma = 0{,}9$. Здесь применяется стратегия π, при которой всегда выбирается кратчайший путь к целевому состоянию с $r = +1$

Функция полезности Q^{π} в уравнении (1.11) оценивает, насколько хороша или плоха пара «состояние — действие». Q^{π} дает оценку ожидаемой отдачи от выбора действия a в состоянии s, предполагая, что агент продолжает действовать в соответствии со своей текущей стратегией π. По аналогии с V^{π} отдача подсчитывается, начиная с текущего состояния s и до конца эпизода. Это тоже прогнозная оценка, поскольку не учитываются все вознаграждения, полученные до состояния s.

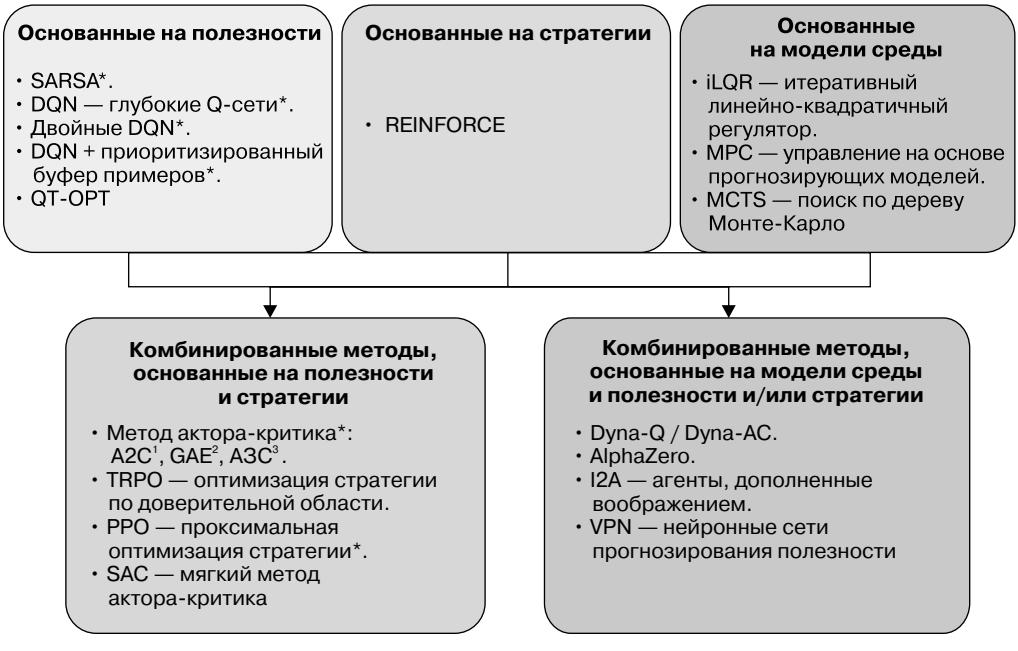
Функции V^{π} и Q^{π} более подробно рассматриваются в главе 3. А сейчас достаточно знать, что они существуют и могут быть использованы агентом для решения задач обучения с подкреплением.

Функция переходов $P(s' \mid s, a)$ предоставляет информацию о среде. Сформировав эту функцию, агент обретает способность предсказывать следующее состояние s', в которое перейдет среда после выбора действия a в состоянии s. Применяя полученную функцию переходов, агент может вообразить последствия действий, не вступая в действительное взаимодействие со средой. В дальнейшем он может использовать эту информацию для планирования оптимальных действий.

1.4. Алгоритмы глубокого обучения с подкреплением

В RL агент настраивает функции, которые помогают ему действовать и максимизировать целевую функцию. Эта книга посвящена глубокому обучению *с подкреплением* (глубокому RL), что означает, что в качестве аппроксимирующего семейства функций будут использоваться глубокие нейронные сети.

В разделе 1.3 были показаны три основные обучаемые функции в обучении с подкреплением. Им соответствуют три больших семейства алгоритмов глубокого обучения с подкреплением — *методы, основанные на стратегии*, *методы, основанные на полезности*, и *методы, основанные на модели среды*. Существуют также комбинированные методы, в которых агенты настраивают несколько этих функций, например функции стратегии и полезности или функцию полезности и модели среды. На рис. 1.5 приведен обзор главных алгоритмов глубокого обучения с подкреплением в каждом из семейств и показаны взаимосвязи между ними.



* — рассматриваются в этой книге.
1. A2C — метод актора-критика с оценкой преимущества за n шагов.
2. A2C — асинхронный метод актора-критика.
3. Метод актора-критика с обобщенной оценкой преимущества

Рис. 1.5. Семейства алгоритмов глубокого обучения с подкреплением

1.4.1. Алгоритмы, основанные на стратегии

Алгоритмы из этого семейства настраивают стратегию π. Хорошие стратегии должны порождать действия, обеспечивающие траектории τ, которые максимизируют целевую функцию агента $J(\tau) = \mathbb{E}_{\tau \sim \pi}\left[\sum_{t=0}^{T} \gamma^t r_t\right]$. Это довольно интуитивный подход: если агенту нужно действовать в среде, то настройка стратегии кажется вполне разумным подходом. Какое действие будет оптимальным в данный момент, зависит

от состояния. То есть функция стратегии π принимает на входе состояние s и выдает действие $a \sim \pi(s)$. Это означает, что агент может принимать хорошие решения в разных ситуациях. Обсуждаемый в главе 2 метод REINFORCE — это основанный на стратегии алгоритм, который приобрел наибольшую известность и стал основой многих других алгоритмов.

Главное преимущество основанных на стратегии алгоритмов как класса методов оптимизации — то, что это очень разнообразный класс. Они применимы к задачам с любым типом действий — дискретным, непрерывным или комбинированным. Кроме того, они оптимизируют непосредственно то, что интересует агента больше всего, — целевую функцию $J(\tau)$. К тому же Саттоном и другими в теореме о градиенте стратегии было доказано, что этот класс методов гарантированно сходится к локально[1] оптимальной политике. Недостатки этих методов — высокая дисперсия и низкая эффективность выборок.

1.4.2. Алгоритмы, основанные на полезности

Агент настраивает либо $V^{\pi}(s)$, либо $Q^{\pi}(s, a)$. Настроенная функция полезностей используется им для оценки пар (s, a) и порождения стратегии. Например, стратегия агента может быть такой, что он всегда выбирает действие a в состоянии s с максимальным оценочным значением $Q^{\pi}(s, a)$. В подходе, основанном только на полезностях, намного чаще применяется настройка $Q^{\pi}(s, a)$, чем $V^{\pi}(s)$, поскольку первую проще преобразовать в стратегию. Это связано с тем, что $Q^{\pi}(s, a)$ содержит информацию о парах состояний и действий, тогда как $V^{\pi}(s)$ — лишь сведения о состояниях.

Обсуждаемый в главе 3 метод SARSA — один из старейших алгоритмов обучения с подкреплением. Несмотря на свою простоту, SARSA объединяет многие ключевые идеи методов, основанных на стратегиях, так что с него хорошо начинать изучение этого семейства алгоритмов. Однако он пока не получил широкого распространения из-за высокой дисперсии и низкой эффективности выборок во время обучения. Глубокая Q-сеть (DQN) и ее последователи, такие как двойные DQN и DQN с приоритизированной памятью прецедентов, намного эффективнее и куда популярнее. Это темы глав 4 и 5.

Как правило, основанные на полезности алгоритмы имеют большую эффективность выборок, чем алгоритмы, основанные на стратегии. Это обусловлено тем, что у них меньшая дисперсия и они продуктивнее используют собранные в среде данные. Однако нет гарантии, что эти алгоритмы сойдутся к оптимальным значениям. К тому же в своей стандартной формулировке их можно применять только

[1] Гарантия глобальной сходимости — все еще открытый вопрос. Не так давно она была доказана для подкласса задач линеаризованного управления. См. статью: *Fazel et al.* Global Convergence of Policy Gradient Methods for Linearized Control Problems (2018).

к средам с дискретным пространством действий. Исторически это было главным ограничением, но последние улучшения, такие как QT-OPT, позволили эффективно применять их к средам с непрерывным пространством действий.

1.4.3. Алгоритмы, основанные на модели среды

Алгоритмы из этого семейства либо занимаются настройкой модели динамики переходов среды, либо задействуют известные динамические модели. Располагая моделью среды $P(s'|s, a)$, агент способен представить, что может случиться в будущем, прогнозируя траекторию на несколько шагов вперед. Пусть среда находится в состоянии s, тогда агент может оценить, насколько состояние изменится, если он предпримет последовательность действий $a_1, a_2 ... a_n$, повторно применяя $P(s'|s, a)$. При этом он никак не изменяет среду, так что прогнозная траектория появляется у него в результате использования модели. Агент может спрогнозировать несколько разных траекторий с различными последовательностями действий и, оценив их, принять решение о наилучшем выборе действия a.

Исключительно модельный подход широко применяется к играм с известным целевым состоянием, таким как победа либо поражение в шахматах или навигационные задания с целевым состоянием s^*. Это связано с тем, что функции переходов в данном случае не моделируют никаких вознаграждений. И чтобы задействовать этот подход для планирования действий, потребуется закодировать информацию о целях агента в самих состояниях.

Поиск по дереву Монте-Карло (Monte Carlo Tree Search, MCTS) — широко распространенный метод, основанный на модели среды. Он применим к задачам с детерминированным дискретным пространством состояний с известными функциями переходов. Под эту категорию подпадают многие настольные игры, такие как шахматы и го, и до недавнего времени многие компьютерные программы для игры в го были основаны на MCTS. В нем не применяется машинное обучение. Для изучения игровых состояний и расчета их полезностей производятся случайные выборки последовательностей действий — случайные симуляции. Этот алгоритм претерпел несколько улучшений, но основная идея осталась неизменной.

Другие методы, такие как итеративный линейно-квадратичный регулятор (iterative Linear Quadratic Regulators, iLQR) или управление на основе прогнозирующих моделей (Model Predictive Control, MPC), включают изучение динамики переходов зачастую с сильно ограничивающими допущениями[1]. Для изучения динамики агенту нужно действовать в среде, чтобы собрать примеры действительных переходов (s, a, r, s').

[1] Например, в iLQR предполагается, что динамика переходов представляет собой линейную функцию от состояний и действий, а функция вознаграждения квадратичная.

Основанные на модели среды алгоритмы весьма интересны тем, что точная модель наделяет агента возможностью предвидения. Он может проигрывать сценарии и понимать последствия своих действий, не производя реальных действий в среде. Это может быть существенным преимуществом, когда накопление опыта из среды очень затратно с точки зрения ресурсов и времени, например, в робототехнике. Кроме того, по сравнению с методами, основанными на стратегии или полезности, этому алгоритму требуется намного меньше примеров данных для обучения оптимальной стратегии. Это обусловлено тем, что наличие модели дает агенту возможность дополнять его реальный опыт воображаемым.

Тем не менее для большинства задач модели среды получить трудно. Многие среды являются стохастическими, и их динамика переходов неизвестна. В таких случаях необходимо настраивать модель. Этот подход все еще находится на ранней стадии разработки, и его реализация сопряжена с рядом проблем. Во-первых, моделирование среды с обширными пространствами состояний и действий может быть очень трудным. Эта задача может даже оказаться неразрешимой, особенно если переходы чрезвычайно сложные. Во-вторых, от моделей есть польза только тогда, когда они могут точно прогнозировать переходы в среде на много шагов вперед. В зависимости от точности модели ошибки прогнозирования могут суммироваться на каждом шаге и быстро расти, что делает модель ненадежной.

На данный момент нехватка хороших моделей — главное ограничение для применения основанного на модели подхода. Однако основанные на модели методы могут быть очень эффективными. Если они работают, то эффективность выборок для них на 1–2 порядка выше, чем у безмодельных методов.

Различие между *методами, основанными на модели среды*, и *безмодельными методами* применяется и в классификации алгоритмов обучения с подкреплением. Алгоритм, основанный на модели, — это любой алгоритм, в котором используется динамика переходов среды, как настраиваемая, так и известная заранее. Безмодельные алгоритмы не задействуют динамику переходов среды в явном виде.

1.4.4. Комбинированные методы

Данные алгоритмы настраивают одну или несколько основных функций обучения с подкреплением. Представляется естественным объединить три рассмотренных метода, чтобы воспользоваться преимуществами каждого из них. Широкое распространение приобрела группа алгоритмов, настраивающих стратегию и функцию полезности. Они получили меткое название методов актора-критика, поскольку в них стратегия генерирует *действие*, а функция полезности *критикует* действия. Эти алгоритмы рассматриваются в главе 6. Основная их суть в том, что во время обучения настроенная функция полезности может передавать актору обратный сигнал, содержащий больше информации, чем последовательность вознаграждений,

доступная из среды. Стратегия настраивается для использования информации, предоставляемой функцией полезности. Затем, как и в методах, основанных на стратегии, стратегия применяется для генерации действий.

Алгоритмы актора-критика — область активных исследований, в которой в последнее время было сделано много интересных усовершенствований. Вот лишь некоторые из них: оптимизация стратегии в доверительной области (Trust Region Policy Optimization, TRPO), оптимизация ближайшей стратегии (Proximal Policy Optimization, PPO), градиенты глубокой детерминированной стратегии (Deep Deterministic Policy Gradients, DDPG), логистический актор-критик (Soft Actor-Critic, SAC). На текущий момент самое широкое распространение получил PPO, он рассматривается в главе 7.

Алгоритмы могут также использовать модель динамики переходов в среде в сочетании с функцией полезности и/или стратегией. В 2016 году исследователи из DeepMind разработали AlphaZero — алгоритм, предназначенный для обучения игре в го. В нем MCTS объединен с настройкой V^{π} и стратегии π. Dyna-Q — еще один известный алгоритм, в котором сначала происходит итерационный процесс настройки модели на полученных из среды реальных данных. Затем обученная модель генерирует воображаемые данные и использует их для настройки Q-функции.

Приведенные в этом разделе примеры — лишь малая часть алгоритмов глубокого обучения с подкреплением. Это далеко не исчерпывающий список. Напротив, данный раздел задумывался как обзор главных идей в глубоком обучении с подкреплением и способов применения стратегии, функции полезностей и модели среды как по отдельности, так и в сочетании друг с другом. В области глубокого обучения с подкреплением ведутся активные исследования, и каждые несколько месяцев появляются новые перспективные разработки.

1.4.5. Алгоритмы, которые обсуждаются в этой книге

В этой книге упор сделан на методы, основанные на стратегии и полезности и их комбинациях. Приводится описание методов REINFORCE (см. главу 2), SARSA (см. главу 3), DQN (см. главу 4) и их расширений (см. главу 5), актора-критика (см. главу 6) и PPO (см. главу 7). Глава 8 посвящена методам параллелизации, которые могут применяться ко всем перечисленным алгоритмам.

Эта книга призвана быть практическим руководством, поэтому в ней не затрагиваются алгоритмы, основанные на модели среды. Безмодельные методы значительно лучше исследованы и ввиду своей общности приложимы к более широкому классу задач. Один и тот же алгоритм (например, PPO) с минимальными изменениями может быть применен и к видеоиграм, таким как Dota 2, и к управлению робото-

техническим манипулятором. По сути, основанным на стратегии или полезности алгоритмам не нужна какая-то дополнительная информация. Их можно просто поместить в среду и позволить обучаться.

Основанным на модели среды методам, напротив, для работы обычно требуются дополнительные знания о среде, то есть модель динамики переходов. Для таких задач, как шахматы или го, модель — это просто правила игры, которые легко запрограммировать. И даже тогда заставить модель работать в связке с алгоритмами обучения с подкреплением — отнюдь не тривиальная задача, как видно по AlphaZero от DeepMind. Зачастую модель среды неизвестна и должна быть настроена, а это само по себе сложная задача.

Алгоритмы, основанные на стратегии и полезности, также не охвачены в полной мере. В книгу вошли алгоритмы, которые получили широкую известность и распространение и при этом иллюстрируют ключевые понятия глубокого обучения с подкреплением. Цель книги — помочь читателям заложить прочные основы знаний в этой области. Надеемся, что приведенное в книге описание алгоритмов подготовит читателей к знакомству с современными исследованиями и их применению в глубоком обучении с подкреплением.

1.4.6. Алгоритмы по актуальному и отложенному опыту

Последнее важное различие между алгоритмами глубокого обучения с подкреплением состоит в том, что они могут работать *по актуальному опыту* (on-policy) и *по отложенному опыту* (off-policy). Это влияет на то, как данные используются в цикле обучения.

Алгоритм обучения *по актуальному опыту* учится по текущей стратегии, то есть для тренировки могут быть задействованы только данные, сгенерированные текущей стратегией π. Это означает, что при обучении в цикле перебираются версии стратегии π_1, π_2, π_3 и т. д., причем на каждой итерации для порождения новых данных используется только текущая стратегия. В результате по окончании обучения все данные должны быть отброшены, поскольку они становятся бесполезными. Это делает методы обучения по актуальному опыту неэффективными с точки зрения выборки прецедентов — им требуется больше обучающих данных. В этой книге обсуждаются следующие методы обучения по актуальному опыту: REINFORCE (см. главу 2), SARSA (см. главу 3), комбинированные методы актора-критика (см. главу 6) и PPO (см. главу 7).

В то же время у алгоритмов обучения *по отложенному опыту* нет таких требований. Любые накопленные данные могут быть повторно использованы при обучении. Как следствие, у методов обучения по отложенному опыту эффективность выборки

прецедентов выше, но хранимые данные могут занимать намного больше памяти. Из методов обучения по отложенному опыту будут рассмотрены DQN (см. главу 4) и его расширения (см. главу 5).

1.4.7. Краткий обзор методов

Мы познакомились с основными семействами алгоритмов глубокого обучения с подкреплением и рассмотрели несколько способов их классификации. В данных подходах упор делается на разные характеристики, и ни один из них не является лучшим. В соответствии с этими различиями все методы могут быть сведены к следующим категориям.

- Основанные на стратегии, полезности, модели среды или комбинированные в зависимости от того, какую из трех основных функций обучения с подкреплением настраивает алгоритм.
- Основанные на модели и безмодельные в зависимости от того, использует алгоритм модель динамики переходов среды или нет.
- Ведущие обучение по актуальному или отложенному опыту в зависимости от того, учится агент на данных, полученных с помощью только текущей стратегии или нет.

1.5. Глубокое обучение для обучения с подкреплением

В этом разделе будут очень кратко рассмотрены глубокое обучение и процесс настройки параметров новой нейронной сети.

Глубокие нейронные сети добились блестящих результатов в аппроксимации сложных нелинейных функций. Их выразительность обусловлена структурой, состоящей из перемежающихся слоев параметров и нелинейных функций активации. В своей нынешней форме нейронные сети существуют с 1980-х годов, когда Ле Кун с коллегами успешно обучили сверточную нейронную сеть распознавать написанные вручную почтовые индексы. Начиная с 2012 года глубокое обучение успешно применялось ко множеству задач. Благодаря ему были достигнуты передовые результаты в целом ряде областей, включая компьютерное зрение, машинный перевод, понимание естественного языка и синтез речи. На момент написания этих строк глубокое обучение — наиболее эффективная из доступных методика аппроксимации функций.

В 1991 году Джеральд Тезауро впервые (и весьма успешно) применил обучение с подкреплением, чтобы научить нейронную сеть играть в нарды на мастерском уровне. Тем не менее лишь в 2015 году DeepMind достигла производительности на

уровне человека для многих игр Atari, которые получили широкое распространение в качестве основного метода аппроксимации функций в данной области. С тех пор все крупные успехи в обучении с подкреплением связаны с применением нейронных сетей для аппроксимации функций. Именно поэтому в книге мы сосредоточились исключительно на глубоком обучении с подкреплением.

Нейронные сети настраивают функции, которые являются простым отображением входных данных в выходные. Они выполняют последовательные вычисления на входных данных для получения выходных данных, этот процесс известен как *прямой проход*. Функция представляется как конкретный набор значений параметров θ сети, говорится, что функция параметризирована θ. Различные наборы параметров соответствуют разным функциям.

Для настройки функции требуются метод, который будет принимать или генерировать достаточно репрезентативный набор входных данных, и способ оценки выходных данных, выдаваемых сетью. Оценить выходные данные можно одним из двух способов. Первый заключается в порождении «корректных» выходных данных, или целевого значения, для всех входных данных и определении функции потерь, которая измеряет ошибку между целевыми и выданными сетью прогнозными выходными значениями. Эти потери должны быть минимизированы. Второй способ — непосредственное предоставление в ответ на все входные данные одного скалярного значения, такого как вознаграждение или отдачи. Эта скалярная величина показывает, насколько хороши или плохи выходные данные сети, и она должна быть максимизирована. Отрицательное значение этой величины может рассматриваться как функция потерь, которую следует минимизировать.

Если задаться функцией потерь, которая оценивает выходные данные сети, то, меняя значения параметров сети, можно минимизировать потери и повысить производительность. Этот процесс известен как градиентный спуск, поскольку параметры изменяются в сторону скорейшего спуска по поверхности функции потерь в поисках глобального минимума.

Процесс изменения параметров сети с целью минимизации потерь известен также как *процесс обучения* нейронной сети. В качестве примера предположим, что настроенная функция $f(x)$ параметризирована весами сети θ как $f(x; \theta)$. Пусть $L(f(x; \theta), y)$ — заранее заданная функция потерь, где x и y — это входные и выходные данные соответственно. Процесс обучения может быть описан следующим образом.

1. Из всего набора данных случайным образом выбирается набор (x, y), размер которого значительно меньше, чем размер всего набора данных.
2. При прямом проходе по сети на основе входных значений x вычисляются прогнозные выходные значения $\hat{y} = f(x; \theta)$.
3. По выданным сетью прогнозным значениям $\hat{y}$ и значениям y из отобранного набора данных рассчитывается функция потерь $L(\hat{y}, y)$.

4. В соответствии с параметрами сети подсчитываются градиенты (частные производные) функции потерь $\nabla_\theta L$. Современные программные библиотеки для работы с нейронными сетями, такие как PyTorch или TensorFlow, выполняют этот процесс автоматически с применением алгоритма обратного распространения ошибки (что известно также как autograd).
5. Оптимизатор используется для обновления параметров сети с помощью градиентов. Например, оптимизатор на основе *стохастического градиентного спуска* (stochastic gradient descent, SGD) производит следующее обновление: $\theta \leftarrow \theta - \alpha\nabla_\theta L$, где α — скалярное значение скорости обучения. Библиотеки для работы с нейронными сетями предоставляют и много других методик оптимизации.

Эти шаги обучения повторяются, пока выходные данные сети не прекратят изменяться или функция потерь не будет минимизирована или стабилизирована, то есть пока сеть не сойдется.

В обучении с подкреплением ни входные данные сети x, ни корректные выходные данные y изначально не даны. Наоборот, эти значения получаются при взаимодействии агента со средой — из наблюдаемых им состояний и вознаграждений. В обучении с подкреплением это представляет особую трудность для обучения нейронных сетей и многократно обсуждается на протяжении всей книги.

Сложность генерации и оценки данных вызвана тем, что настраиваемые функции тесно связаны с циклом МППР. Агент и среда обмениваются интерактивными данными, и в силу своей природы этот процесс ограничен временем, необходимым на то, чтобы агент произвел действие, а среда выполнила переход. Не существует быстрого способа порождения данных для обучения — агент должен приобретать опыт на каждом шаге. Накопление данных и цикл обучения повторяются, на каждом шаге обучения приходится дожидаться, пока новые данные не будут собраны.

Более того, текущее состояние среды и предпринятые агентом действия влияют на последующие состояния, наблюдаемые им. Из-за этого состояния и вознаграждения в любой момент зависят от состояний и вознаграждений на предыдущих шагах. Это нарушает лежащее в основе градиентного спуска предположение о том, что данные независимы и одинаково распределены (i.i.d). Это может, в свою очередь, отрицательно сказаться на скорости, с которой сходится сеть, и качестве конечного результата. Значительные усилия были потрачены на исследования по минимизации данного эффекта, некоторые из методик обсуждаются далее в этой книге.

Невзирая на эти трудности, глубокое обучение — эффективная методика аппроксимации функций. Для преодоления сложности его применения к обучению с подкреплением придется потрудиться, но усилия не пропадут даром, ведь отдача значительно превышает издержки.

1.6. Обучение с подкреплением и обучение с учителем

В основе глубокого обучения с подкреплением лежит аппроксимация функций. Это то, что объединяет его с обучением с учителем (supervised learning, SL[1]). Однако между обучением с подкреплением и обучением с учителем есть несколько различий. Три основных перечислены далее:

- отсутствие оракула[2];
- разреженность обратной связи;
- генерация данных во время обучения.

1.6.1. Отсутствие оракула

Главное различие между обучением с подкреплением и с учителем в том, что в задачах обучения с подкреплением не для всех входных данных модели имеются корректные ответы, тогда как в обучении с учителем для любого примера существует правильный или оптимальный ответ. В обучении с подкреплением аналогом корректного ответа будет доступ к оракулу, сообщающему, выбор какого действия на каждом шаге будет наилучшим для оптимизации целевой функции.

В правильном ответе может быть передано большое количество информации о текущем примере из данных. К примеру, корректный ответ для задач классификации содержит многие биты информации. В нем не только сообщается о правильном классе для каждого примера, но и подразумевается, что этот пример не относится ни к одному другому классу. Если в отдельно взятой задаче классификации есть 1000 классов (как в наборе данных ImageNet), ответ содержит по 1000 бит информации на каждый пример (1 положительное и 999 отрицательных значений). Более того, правильный ответ не обязательно является категорией или вещественным числом. Это может быть как ограничивающий прямоугольник, так и семантическое разбиение — и то и другое содержит большое количество битов информации о рассматриваемом примере.

В обучении с подкреплением агенту после того, как он предпринял действие a в состоянии s, доступно только полученное им вознаграждение. Агенту не сообщается, какое действие оптимальное. Напротив, посредством вознаграждения ему лишь указывается, насколько хорошим или плохим было a. Помимо того что это указание

[1] В сообществе ИИ любят сокращения. Их будет много — почти у всех алгоритмов и названий компонентов есть аббревиатуры.

[2] В компьютерных науках оракул — это гипотетический черный ящик, дающий корректные ответы на заданные вопросы.

содержит меньше информации, чем правильный ответ, агент учится на вознаграждениях, полученных только в тех состояниях, в которых он побывал. Чтобы получить знания о (s, a, r), ему нужно совершить переход (s, a, r, s'). У агента может отсутствовать информация о важных частях пространств состояний и действий по той простой причине, что он с ними не сталкивался.

Один из способов решения этой проблемы состоит в инициализации эпизодов таким образом, чтобы они начинались с состояний, которые агенту нужно изучить. Однако это не всегда возможно по двум причинам. Во-первых, может отсутствовать полный контроль над средой. Во-вторых, состояния может быть просто описать, но трудно идентифицировать. Рассмотрим ситуацию, когда человекоподобный робот учится делать обратное сальто. Чтобы помочь агенту узнать о вознаграждении за успешное приземление, можно инициализировать среду так, чтобы она запускалась сразу после того, как ноги робота войдут в контакт с полом после удачного сальто. Знание функции вознаграждения в этой части пространства состояний критично, поскольку здесь робот может как удержать равновесие и успешно выполнить сальто, так и упасть. Однако для инициализации робота в данной позиции требуется определить точные угловые координаты и скорости каждого из его суставов или приложенную силу, а это непросто. На практике агенту для достижения данного состояния нужно выполнить очень конкретную длинную последовательность действий, чтобы сделать переворот и почти приземлиться. И нет гарантии, что агент научится делать это, так что данная часть пространства состояний может оказаться так и не исследованной.

1.6.2. Разреженность обратной связи

В обучении с подкреплением функция вознаграждений может быть разреженной, так что скалярная величина вознаграждения часто равна нулю. Это означает, что агент большую часть времени не получает информации о том, как изменить параметры сети, чтобы оптимизировать ее производительность. Снова рассмотрим робота, делающего обратное сальто, и предположим, что агент получает ненулевое вознаграждение +1 только после каждого успешного выполнения прыжка. Почти все предпринимаемые им действия приведут к одному и тому же нулевому вознаграждению от среды. При таких обстоятельствах обучение чрезвычайно затруднено, поскольку агент не получает никаких указаний о том, способствуют ли его промежуточные действия достижению цели. В обучении с учителем нет такой проблемы — всем входным примерам соответствуют желаемые выходные значения, в которых передается информация о том, как сеть должна работать.

Разреженность обратной связи вдобавок к отсутствию оракула означает, что в обучении с подкреплением на каждом шаге из среды будет получено намного меньше информации, чем в тренировочных примерах при обучении с учителем. В результате алгоритмы обучения с подкреплением имеют тенденцию к получению гораздо меньшей эффективности выборки прецедентов из среды.

1.6.3. Генерация данных

В обучении с учителем данные обычно генерируются независимо от обучения алгоритма. Действительно, первым шагом применения обучения с учителем к задаче будет поиск или построение хорошего набора данных. В обучении с подкреплением данные должны порождаться при взаимодействии агента со средой. Эти данные в большинстве случаев генерируются в ходе обучения итеративно, с чередованием периодов накопления данных и обучения. Между получаемыми данными и работой алгоритма существует взаимосвязь. Качество алгоритма влияет на данные, на которых он обучается, а они, в свою очередь, влияют на производительность алгоритма. В обучении с учителем нет этой цикличности и не требуется повторная генерация выборок (bootstrapping).

Кроме того, RL интерактивно: выполненные агентом действия фактически трансформируют среду, которая затем изменяет решения агента, которые преобразуют данные, наблюдаемые агентом, и т. д. Этот цикл обратной связи — отличительная особенность обучения с подкреплением. В задачах обучения с учителем этого цикла нет, как нет и понятия, эквивалентного агенту, способному изменять данные, на которых обучается алгоритм.

Последнее, менее значимое различие между обучением с подкреплением и с учителем состоит в том, что в первом нейронные сети не всегда учатся с применением функции потерь, используемой в распознавании. Получаемые из среды вознаграждения применяются не для того, чтобы минимизировать ошибку, обусловленную разницей между выходными данными среды и желаемыми целевыми значениями, а для построения целевой функции, максимизировать которую затем учится сеть. Тем, кто изучал обучение с учителем, поначалу это может показаться немного странным, но механизм оптимизации, по сути, тот же самый. В обоих случаях параметры сети настраиваются с целью максимизации или минимизации некоторой функции.

1.7. Резюме

В этой главе дано описание задач обучения с подкреплением как работы систем, состоящих из агента и среды, которые взаимодействуют и обмениваются информацией в виде состояний, действий и вознаграждений. Агенты учатся действовать в среде с помощью *стратегии* так, чтобы максимизировать ожидаемую сумму вознаграждений, которая является их целевой функцией. С применением этих понятий было показано, как обучение с подкреплением может быть сформулировано в виде МППР, в предположении, что функция переходов среды имеет марковское свойство.

Опыт, полученный агентом в среде, может быть использован для настройки функций, которые помогают агенту максимизировать целевую функцию. В частности,

были рассмотрены три основные настраиваемые функции в обучении с подкреплением: *стратегии* $\pi(s)$, *функции полезности* $V^{\pi}(S)$ и $Q^{\pi}(s, a)$, а также *модели* $P(s' \mid s, a)$. В зависимости от того, какую из этих функций настраивает агент, алгоритмы глубокого обучения с подкреплением могут быть разделены на основанные на стратегии, полезности или модели среды либо комбинированные. Их можно также разделить на категории в соответствии с тем, как порождаются тренировочные данные. В алгоритмах обучения по актуальному опыту используются только данные, сгенерированные текущей стратегией, в алгоритмах обучения по отложенному опыту могут применяться данные, порожденные любой стратегией.

Вкратце были рассмотрены процесс обучения в глубоком обучении с подкреплением и некоторые из различий между обучением с подкреплением и обучением с учителем.

Часть I

Алгоритмы, основанные на стратегиях и полезностях

2 REINFORCE

В этой главе представлен первый из описываемых в книге алгоритмов, REINFORCE.

Алгоритм REINFORCE был предложен Рональдом Дж. Вильямсом в 1992 году и описан им в статье *Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning*[1]. Алгоритм строит параметризированную стратегию, которая получает вероятности действий по состояниям среды. Агенты непосредственно используют эту стратегию, чтобы действовать в среде.

Основной смысл заключается в том, что во время обучения действия, которые приводят к хорошим результатам, должны иметь большую вероятность — они положительно *подкрепляются*. В противовес этому действия, приводящие к плохим результатам, должны иметь меньшую вероятность. Если обучение успешно, то за несколько итераций распределение полученных стратегией вероятностей действий станет таким, которое приводит к повышению производительности в среде. Вероятности действий изменяются в соответствии с градиентом функции стратегии, в связи с чем REINFORCE известен как *алгоритм градиента стратегии*.

Для этого алгоритма необходимо наличие трех составляющих:

- параметризированной стратегии;
- максимизируемой целевой функции;
- метода обновления параметров стратегии.

В разделе 2.1 представлены параметризированные стратегии. Далее, в разделе 2.2, обсуждаются функции, определяющие оценку результатов действий. В разделе 2.3 приведена основа алгоритма REINFORCE — градиент стратегии. Он предоставляет способ вычисления градиента целевой функции по параметрам стратегии. Это важнейший этап, поскольку с помощью градиента стратегии параметры последней преобразуются для максимизации целевой функции.

После знакомства с алгоритмом в разделе 2.5 обсуждаются его ограничения и вводятся новые способы повышения производительности (см. подраздел 2.5.1).

В конце главы приведены две реализации этого алгоритма: первая короткая и самодостаточная и вторая, показывающая, как он выполнен в SLM Lab.

---

[1] «Простые градиентные алгоритмы для нейросетевого обучения с подкреплением».

2.1. Стратегия

Стратегия π — это функция, отображающая состояния вероятности действий, которые используются для выбора действия $a \sim \pi(s)$. В REINFORCE агент настраивает стратегию и с ее помощью действует в среде.

Оптимальна та стратегия, которая максимизирует суммарное дисконтированное вознаграждение. Основной смысл алгоритма — настроить хорошую стратегию, а это подразумевает аппроксимацию функции. Нейронные сети — эффективный и гибкий метод аппроксимации функций, так что стратегия может быть представлена как глубокая нейронная сеть, определяемая настраиваемыми параметрами θ. Такие сети часто называют сетями стратегии π_θ. Говорится, что стратегия параметризирована θ.

Каждый конкретный набор значений параметров сети стратегии представляет отдельную стратегию. Чтобы увидеть, почему так происходит, рассмотрим случай $\theta_1 \neq \theta_2$. Для любого данного состояния s разные сети стратегии могут вернуть разные наборы вероятностей действий, то есть $\pi_{\theta_1}(s) \neq \pi_{\theta_2}(s)$. Отображения состояний в вероятности действий различаются, поэтому говорится, что π_{θ_1} и π_{θ_2} — разные стратегии. Следовательно, одной нейронной сетью могут быть представлены несколько разных стратегий.

В такой формулировке процесс настройки хорошей стратегии соответствует поиску набора оптимальных значений для θ. Поэтому важно, чтобы сеть стратегии была дифференцируемой. В разделе 2.3 будет показано, как стратегия может быть улучшена посредством градиентного подъема в пространстве параметров.

2.2. Целевая функция

В этом разделе дано определение целевой функции, которая максимизируется агентом в алгоритме REINFORCE. Целевую функцию можно понимать как цель агента — победа в игре или достижение наибольшего возможного счета. Сначала вводится понятие отдачи, которая подсчитывается по траектории. Затем отдача используется для того, чтобы определить цель.

Как вы помните из главы 1, действующий в среде агент порождает траекторию, состоящую из последовательности вознаграждений вместе с состояниями и действиями. Траектория обозначается $\tau = s_0, a_0, r_0 \ldots s_T, a_T, r_T$.

Отдача по траектории $R_t(\tau)$ определяется как дисконтированная сумма вознаграждений, начиная с шага t и до конца траектории:

$$R_t(\tau) = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}. \tag{2.1}$$

Заметьте, что в уравнении (2.1) вычисление суммы начинается с шага t, но показатель степени при коэффициенте γ при подсчете отдачи начинается с нуля. Нужно учесть это в показателе степени на шаге t с помощью $t' - t$.

Когда $t = 0$, отдача просто рассчитывается для полной траектории. Это кратко записывается как $R_0(\tau) = R(\tau)$. *Целевая функция* — это ожидаемая отдача по всем полным траекториям, порожденным агентом. Она определяется уравнением:

$$J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[R(\tau)\right] = \mathbb{E}_{\tau \sim \pi_\theta}\left[\sum_{t=0}^{T} \gamma^t r_t\right]. \tag{2.2}$$

Уравнение (2.2) гласит, что ожидание вычисляется по множеству траекторий, выбранных по стратегии, то есть $\tau \sim \pi_\theta$. Математическое ожидание тем ближе к истинному значению, чем больше примеров было накоплено, и оно связано с конкретной используемой стратегией π_θ.

2.3. Градиент стратегии

Мы определили стратегию π_θ и целевую функцию $J(\pi_\theta)$ — ключевые компоненты алгоритма стратегии. Стратегия предоставляет агенту способ действий, а целевая функция — цель, которую нужно максимизировать.

Последний компонент этого алгоритма — градиент стратегии. Формально говорится, что алгоритм градиента стратегии решает следующую задачу:

$$\max_\theta J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[R(\tau)\right]. \tag{2.3}$$

Чтобы максимизировать целевую функцию, выполним градиентный подъем по параметрам стратегии θ. Как вы помните из курса высшей математики, градиент указывает направление скорейшего возрастания функции. Чтобы улучшить значение целевой функции[1], вычислим градиент и применим его для обновления параметров[2]:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\pi_\theta). \tag{2.4}$$

---

[1] Целевую функцию $J(\pi_\theta)$ можно рассматривать как абстрактную гиперповерхность с θ в качестве переменных, на которой мы пытаемся найти точку максимума. Гиперповерхность — это обобщение обычной двумерной поверхности, принадлежащей трехмерному пространству в высших измерениях. Гиперповерхность из N-мерного пространства — это объект с $N - 1$ измерениями.

[2] Обратите внимание на то, что этот параметр может быть получен с помощью любого подходящего оптимизатора, который принимает на входе $\nabla_\theta J(\pi_\theta)$.

В уравнении (2.4) α — скалярная величина, которая известна как скорость обучения и которая контролирует степень обновления параметров. Член $\nabla_\theta J(\pi_\theta)$ называется *градиентом стратегии*. Он определяется уравнением:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[\sum_{t=0}^{T} R_t(\tau)\nabla_\theta \log \pi_\theta(a_t|s_t)\right]. \tag{2.5}$$

Здесь член $\pi_\theta(a_t|s_t)$ — это вероятность действия, предпринятого агентом на шаге t. Действие выбрано по стратегии, $a_t \sim \pi_\theta(s_t)$. Правая часть уравнения утверждает, что градиент по θ логарифма вероятности действия умножается на отдачу $R_t(\tau)$.

Согласно уравнению (2.5) градиент целевой функции эквивалентен ожидаемой сумме градиентов логарифмов вероятностей действий a_t, умноженных на соответствующие отдачи $R_t(\tau)$. Полный вывод этого уравнения приведен в подразделе 2.3.1.

Градиент стратегии — это механизм, с помощью которого изменяются вероятности действий, полученных по стратегии. Если отдача $R_t(\tau) > 0$, то вероятность действия $\pi_\theta(a_t|s_t)$ растет, и, наоборот, если $R_t(\tau) < 0$, то уменьшается. В ходе многих обновлений (см. уравнение (2.4)) стратегия настраивается производить действия, которые ведут к большему значению $R_t(\tau)$.

На уравнении (2.5) основаны методы градиента стратегиям. REINFORCE был первым алгоритмом, в котором использовалась простейшая форма этого уравнения. Позже на его основе были построены алгоритмы (главы 6 и 7) с более высокой производительностью, достигаемой за счет применения преобразования целевой функции. Тем не менее остается последний вопрос: как реализовать и решить идеальное уравнение для вычисления градиента стратегии.

2.3.1. Вывод формулы для градиента по стратегиям

Здесь мы выводим градиент стратегии (см. уравнение (2.5)) из градиента целевой функции (обратите внимание: при первом прочтении этот раздел можно пропустить):

$$\nabla_\theta J(\pi_\theta) = \nabla_\theta \mathbb{E}_{\tau \sim \pi_\theta}\left[R(\tau)\right]. \tag{2.6}$$

Поскольку $R(\tau) = \sum_{t=0}^{T} \gamma^t r_t$ невозможно продифференцировать по θ, решение уравнения (2.6) представляется проблематичным[1]. Вознаграждения r_t генерируются неизвестной функцией вознаграждения $\mathcal{R}(s_t, a_t, s_{t+1})$, которую нельзя продифференцировать. Параметры стратегии θ влияют на $R(\tau)$ лишь путем изменения распределений состояний и действий, которые, в свою очередь, меняют вознаграждения, получаемые агентом.

---

[1] Тем не менее этот градиент можно оценить, прибегнув к оптимизации методами черного ящика, такими как расширенный метод случайного поиска (Augmented Random Search), но это выходит за рамки книги.

Таким образом, нужно привести уравнение (2.6) к форме, позволяющей взять градиент по θ. Воспользуемся для этого некоторыми удобными тождествами.

Если заданы функция $f(x)$, параметризированное распределение вероятностей $p(x \mid \theta)$ и его математическое ожидание $\mathbb{E}_{x \sim p(x|\theta)}[f(x)]$, то градиент математического ожидания может быть записан следующим образом:

$$\nabla_\theta \mathbb{E}_{x \sim p(x|\theta)}\left[f(x)\right] =$$

$$= \nabla_\theta \int dx\, f(x) p(x|\theta) = \quad \text{(определение математического ожидания)} \quad (2.7)$$

$$= \int dx\, \nabla_\theta \left(p(x|\theta) f(x)\right) = \quad \text{(внесение } \nabla_\theta \text{ под знак интеграла)} \quad (2.8)$$

$$= \int dx \left(f(x) \nabla_\theta p(x|\theta) + p(x|\theta) \nabla_\theta f(x)\right) = \quad \text{(дифференцирование сложной функции)} \quad (2.9)$$

$$= \int dx\, f(x) \nabla_\theta p(x|\theta) = \quad (\nabla_\theta f(x) = 0) \quad (2.10)$$

$$= \int dx\, f(x)\, p(x|\theta) \frac{\nabla_\theta p(x|\theta)}{p(x|\theta)} = \quad \left(\text{умножение на } \frac{p(x|\theta)}{p(x|\theta)}\right) \quad (2.11)$$

$$= \int dx\, f(x)\, p(x|\theta) \nabla_\theta \log p(x|\theta) = \quad \text{(подстановка уравнения (2.14))} \quad (2.12)$$

$$= \mathrm{E}_x\left[f(x) \nabla_\theta \log p(x|\theta)\right]. \quad \text{(определение математического ожидания)} \quad (2.13)$$

Данное тождество гласит, что градиент математического ожидания равен математическому ожиданию градиента логарифма вероятности, умноженному на исходную функцию. В первой строке просто дано определение математического ожидания. Для общности использована интегральная форма, поскольку предполагается, что $f(x)$ — непрерывная функция. Но это тождество в равной мере применимо и к вычислению суммы, когда функция дискретная. В следующих трех строках производятся преобразования в соответствии с правилами интегрирования.

Заметьте, что в уравнении (2.10) была решена начальная задача, так как можно взять градиент $p(x \mid \theta)$, но $f(x)$ — это функция черного ящика, которую нельзя проинтегрировать. Для разрешения этой проблемы нужно преобразовать данное уравнение в математическое ожидание, чтобы вычислить его посредством оценки по выборке. Сначала в уравнении (2.11) выполним умножение на $\frac{p(x|\theta)}{p(x|\theta)}$. Полученная дробь $\frac{\nabla_\theta p(x|\theta)}{p(x|\theta)}$ может быть переписана в виде *произведения логарифма*:

$$\nabla_\theta \log p(x|\theta) = \frac{\nabla_\theta p(x|\theta)}{p(x|\theta)}. \quad (2.14)$$

Подстановка равенства (2.14) в уравнение (2.11) дает уравнение (2.12). И наконец, в уравнении (2.13) последнее выражение просто записывается как математическое ожидание.

Теперь очевидно, что данное тождество можно применить к целевой функции. С помощью подстановки $x = \tau$, $f(x) = R(\tau)$, $p(x \mid \theta) = p(\tau \mid \theta)$ перепишем уравнение (2.6):

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[R(\tau)\nabla_\theta \log p(\tau \mid \theta)\right]. \tag{2.15}$$

Однако в уравнении (2.15) нужно связать член $p(\tau \mid \theta)$ со стратегией π_θ, которой мы можем управлять. Следовательно, его нужно развернуть.

Стоит отметить, что траектория τ — это просто последовательность чередующихся событий. a_t и s_{t+1} взяты с использованием вероятности действий агента $\pi_\theta(a_t \mid s_t)$ и вероятности переходов среды $p(s_{t+1} \mid s_t, a_t)$ соответственно. Поскольку эти вероятности относительно независимы, вероятность всей траектории является произведением отдельных вероятностей, как показано в следующем уравнении:

$$p(\tau \mid \theta) = \prod_{t \geq 0} p(s_{t+1} \mid s_t, a_t)\pi_\theta(a_t \mid s_t). \tag{2.16}$$

Прологарифмируем[1] обе части уравнения (2.16), чтобы привести его в соответствие с уравнением (2.15):

$$\log p(\tau \mid \theta) = \log \prod_{t \geq 0} p(s_{t+1} \mid s_t, a_t)\pi_\theta(a_t \mid s_t); \tag{2.17}$$

$$\log p(\tau \mid \theta) = \sum_{t \geq 0}\left(\log p(s_{t+1} \mid s_t, a_t) + \log \pi_\theta(a_t \mid s_t)\right); \tag{2.18}$$

$$\nabla_\theta \log p(\tau \mid \theta) = \nabla_\theta \sum_{t \geq 0}\left(\log p(s_{t+1} \mid s_t, a_t) + \log \pi_\theta(a_t \mid s_t)\right); \tag{2.19}$$

$$\nabla_\theta \log p(\tau \mid \theta) = \nabla_\theta \sum_{t \geq 0} \log \pi_\theta(a_t \mid s_t). \tag{2.20}$$

Уравнение (2.18) вытекает из того, что логарифм произведения равен сумме составляющих его логарифмов. Начиная отсюда к обеим сторонам этого уравнения можно применить градиент ∇_θ, что даст уравнение (2.19). Градиент можно внести под знак суммы и применить к слагаемым. Член $\log p(s_{t+1} \mid s_t, a_t)$ можно опустить, так как он не зависит от θ и его градиент равен нулю. Таким образом получено уравнение (2.20), в котором вероятность $p(\tau \mid \theta)$ выражена через $\pi_\theta(a_t \mid s_t)$. Также следует отметить, что траектория τ слева сопоставляется сумме ее отдельных временных шагов t справа.

Наконец, с учетом этого можно записать $\nabla_\theta J(\pi_\theta)$ из уравнения (2.6) в дифференцируемой форме. Подстановкой уравнения (2.20) в равенство (2.15) и введением множителя $R(\tau)$ получим:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[\sum_{t=0}^{T} R_t(\tau)\nabla_\theta \log \pi_\theta(a_t \mid s_t)\right]. \tag{2.21}$$

---

[1] Одно из преимуществ применения логарифмических вероятностей состоит в том, что они отличаются повышенной численной устойчивостью по сравнению с простыми вероятностями. В качестве небольшого упражнения попытайтесь понять, чем обусловлено это преимущество.

Проблемой было то, что уравнение (2.6) содержало недифференцируемую функцию. После ряда преобразований было получено уравнение (2.20). Найти его решение довольно просто, применив сеть стратегии π_θ и вычислив градиенты с помощью функции автоматического дифференцирования из библиотек нейронных сетей.

2.4. Выборка методом Монте-Карло

Алгоритм REINFORCE рассчитывает градиент стратегии с помощью выборки методом Монте-Карло.

Под *выборкой методом Монте-Карло* подразумевается любой метод, в котором данные, используемые для аппроксимации функции, генерируются путем случайной выборки. По сути, это просто аппроксимация путем случайной выборки. Эта методика стала востребованной благодаря Станиславу Уламу — математику, работавшему в 1940-х годах в Лос-Аламосской исследовательской лаборатории.

Рассмотрим работу метода Монте-Карло на примере вычисления числа π (математической константы) — отношения длины окружности к ее диаметру[1]. Метод Монте-Карло подходит к решению этой задачи следующим образом: берется окружность радиусом $r = 1$ с центром в начале координат и вписывается в квадрат. Площади фигур равны πr^2 и $(2r)^2$ соответственно. Следовательно, отношением этих площадей будет просто:

$$\frac{\text{площадь круга}}{\text{площадь квадрата}} = \frac{\pi r^2}{(2r)^2} = \frac{\pi}{4}. \tag{2.22}$$

Численное значение площади квадрата равно 4, но так как π еще неизвестно, то неизвестна и площадь круга. Рассмотрим одну четверть фигур. Чтобы получить оценку π, произведем выборку множества точек внутри квадрата с помощью равномерного распределения. Расстояние от точки (x, y), лежащей внутри окружности, до начала координат меньше 1, то есть $\sqrt{(x-0)^2 + (y-0)^2} \leq 1$ (рис. 2.1). Если подсчитать с помощью этого условия количество точек внутри окружности, то его отношение к количеству всех выбранных точек будет приблизительно равно отношению, полученному в уравнении (2.22). Последовательная выборка большего количества точек и уточнение данного отношения позволяют приблизиться к точному значению. Умножение этого отношения на 4 дает расчетное значение $\pi \approx 3{,}14159$.

[1] Мы не используем π для обозначения стратегии только в этом примере, но затем вернемся к исходному смыслу.

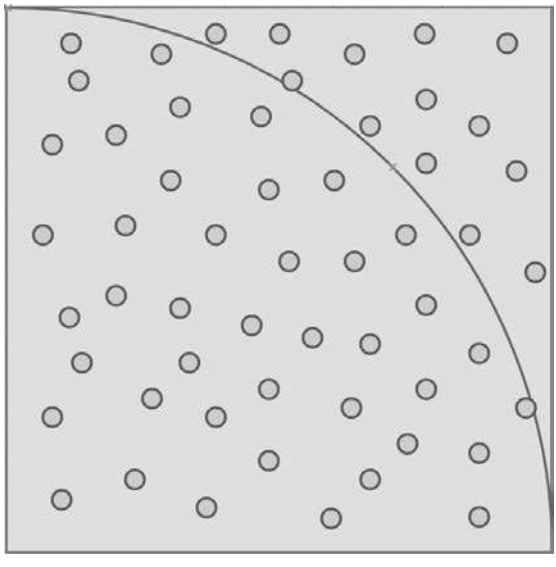


Рис. 2.1. Расчет π с помощью выборки методом Монте-Карло

Теперь вернемся к глубокому RL и рассмотрим применение метода Монте-Карло к численному вычислению градиента стратегии в уравнении (2.5). Это очень просто. Математическое ожидание $\mathbb{E}_{\tau \sim \pi_\theta}$ подразумевает, что чем больше траекторий τ сгенерировано с помощью стратегии π_θ и усреднено, тем оно ближе к реальному градиенту стратегии $\nabla_\theta J(\pi_\theta)$. Для конкретной стратегии вместо большого количества траекторий можно взять лишь одну, как показано в уравнении (2.23):

$$\nabla_\theta J(\pi_\theta) \approx \sum_{t=0}^{T} R_t(\tau) \nabla_\theta \log \pi_\theta(a_t \mid s_t). \tag{2.23}$$

Так реализован градиент стратегии — как оценка методом Монте-Карло с выборкой по сгенерированным траекториям.

Теперь, когда получены все составляющие REINFORCE, рассмотрим сам алгоритм.

2.5. Алгоритм REINFORCE

В этом разделе обсуждается алгоритм REINFORCE и вводится понятие алгоритмов онлайн-обучения. Затем будут рассмотрены некоторые его ограничения и метод базового значения для повышения производительности.

Сам алгоритм 2.1 очень простой. Сначала задается начальное значение скорости обучения α и строится сеть стратегии π_θ со случайно инициализированными весами. Затем в ходе итераций по множеству эпизодов для каждого из них сеть стратегии π_θ

генерирует траектории $\tau = s_0, a_0, r_0 \ldots s_T, a_T, r_T$. Далее для каждого шага t в траектории вычисляется отдача $R_t(\tau)$, которая используется для подсчета градиента стратегии. Сумма градиентов стратегии для всех шагов применяется для обновления параметров сети стратегии θ.

Алгоритм 2.1. REINFORCE

```
 1: Инициализировать скорость обучения α
 2: Инициализировать веса сети стратегии π_θ
 3: for episode = 0, ..., MAX_EPISODE do
 4:     Выбрать траекторию τ = s_0, a_0, r_0, ..., s_T, a_T, r_T
 5:     Установить ∇_θJ(π_θ) = 0
 6:     for t = 0, ..., T do
 7:         R_τ(τ) = Σ_{t'=τ}^{T} γ^{t'-t} r'_t
 8:         ∇_θJ(π_θ) = ∇_θJ(π_θ) + R_t(τ)∇_θlog π_θ(a_t|s_t)
 9:     end for
10:     θ = θ + α∇_θJ(π_θ)
11: end for
```

Важно помнить, что траектория отбрасывается после каждого обновления параметров — она не может использоваться повторно. Это вызвано тем, что REINFORCE — алгоритм *обучения по актуальному опыту*. Как вы помните из раздела 1.4, в алгоритме применяется обучение по актуальному опыту, если уравнение для обновления параметров зависит от текущей стратегии. Это очевидно из строки 8, поскольку градиент стратегии напрямую обусловлен вероятностями действий $\pi_\theta(a_t \mid s_t)$, порожденными текущей стратегией π_θ, а не какой-то прошлой стратегией π_θ'. Соответственно, и отдача $R_t(\tau)$, где $\tau \sim \pi_\theta$, также должна генерироваться стратегией π_θ, иначе вероятности действий будут регулироваться на основе отдач, которые не были порождены стратегией.

2.5.1. Усовершенствование метода REINFORCE

В приведенной ранее формулировке алгоритма REINFORCE градиент стратегии вычисляется с помощью выборки по методу Монте-Карло по одной траектории. Это несмещенная оценка градиента стратегии, но у такого подхода есть недостаток — высокая дисперсия. В этом разделе вводится метод базового значения для снижения дисперсии оценки. Кроме того, далее обсуждается нормализация вознаграждений с целью решения проблемы их масштабирования.

При использовании выборки по методу Монте-Карло оценка градиента стратегии может иметь высокую дисперсию, так как отдачи могут значительно варьироваться от траектории к траектории. Это обусловлено тремя факторами. Во-первых, действиям присуща определенная случайность, поскольку они выбраны из распределения вероятностей. Во-вторых, начальное состояние может быть разным в различных эпизодах. В-третьих, функция переходов среды может быть стохастической.

Один из способов снижения дисперсии оценки — преобразовать отдачи, выделив соответствующее базовое значение, не зависящее от действий, как показано в уравнении (2.24):

$$\nabla_{\theta_0} J(\pi_{\theta_0}) \approx \sum_{t=0}^{T} \left(R_t(\tau) - b(s_t) \right) \nabla_0 \log \pi_{\theta_0}(a_t \mid s_t). \tag{2.24}$$

Один из примеров базового значения — функция полезности V^{π}. На таком выборе базового значения основан алгоритм актора-критика. Он обсуждается в главе 6.

Альтернативный вариант — использование средних отдач на траектории. Пусть $b = \frac{1}{T} \sum_{t=0}^{T} R_t(\tau)$. Обратите внимание: это базовое значение для траектории постоянно и не изменяется с состоянием s_t. В результате для всех траекторий значения отдачи будут центрированы относительно нуля. Для каждой траектории в среднем 50 % действий будут поощряться, а остальные — нет.

Чтобы понять, чем это может быть полезно, рассмотрим случай, когда все вознаграждения в среде отрицательные. Без базового значения агент не поощряется даже за очень хорошие действия, так как отдача всегда отрицательная. Со временем это все-таки может привести к хорошей стратегии, ведь худшие действия поощряются и того меньше, что косвенно повышает вероятность лучших действий. Тем не менее это может замедлить обучение, поскольку уточнение вероятностей может быть выполнено лишь в одном направлении. В средах, где все вознаграждения положительные, происходит обратное. Обучение более эффективно, когда можно как повышать, так и снижать вероятность действий. Для этого необходимо иметь как положительные, так и отрицательные вознаграждения.

2.6. Реализация REINFORCE

В этом разделе представлены две реализации, первая из которых является минимальной работоспособной. Во второй показано, как алгоритм REINFORCE реализован в SLM Lab, это немного более продвинутая версия, которая интегрируется с компонентами библиотеки.

2.6.1. Минимальная реализация REINFORCE

Минимальная реализация полезна тем, что она в большей степени соответствует алгоритму 2.1. Это хороший опыт превращения теории в код. REINFORCE — хороший кандидат на эту роль, поскольку является простейшим алгоритмом RL, который можно реализовать в нескольких строках кода (листинг 2.1).

Другие алгоритмы RL сложнее, в связи с чем для упрощения их реализации применяются распространенные фреймворки и повторно используемые модульные

компоненты. Поэтому, чтобы привести REINFORCE в соответствие с остальными алгоритмами в книге, здесь дается его реализация в SLM Lab. Это служит введением в API алгоритмов библиотеки.

Листинг 2.1. Независимая рабочая реализация REINFORCE для решения CartPole-v0

```
from torch.distributions import Categorical
import gym
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim

gamma = 0.99

class Pi(nn.Module):
    def __init__(self, in_dim, out_dim):
        super(Pi, self).__init__()
        layers = [
            nn.Linear(in_dim, 64),
            nn.ReLU(),
            nn.Linear(64, out_dim),
        ]
        self.model = nn.Sequential(*layers)
        self.onpolicy_reset()
        self.train() # установить режим обучения

    def onpolicy_reset(self):
        self.log_probs = []
        self.rewards = []

    def forward(self, x):
        pdparam = self.model(x)
        return pdparam

    def act(self, state):
        x = torch.from_numpy(state.astype(np.float32)) # преобразование
        ➥ в тензор
        pdparam = self.forward(x) # прямой проход
        pd = Categorical(logits=pdparam) # вероятностное распределение
        action = pd.sample() # pi(a|s) выбор действия по распределению pd
        log_prob = pd.log_prob(action) # логарифм вероятности pi(a|s)
        self.log_probs.append(log_prob) # сохраняем для обучения
        return action.item()

def train(pi, optimizer):
    # Внутренний цикл градиентного восхождения в алгоритме REINFORCE
    T = len(pi.rewards)
    rets = np.empty(T, dtype=np.float32) # отдачи
    future_ret = 0.0
    # эффективное вычисление отдачи
```

```
45    for t in reversed(range(T)):
46        future_ret = pi.rewards[t] + gamma * future_ret
47        rets[t] = future_ret
48    rets = torch.tensor(rets)
49    log_probs = torch.stack(pi.log_probs)
50    loss = - log_probs * rets # член градиента; знак минуса для максимизации
51    loss = torch.sum(loss)
52    optimizer.zero_grad()
53    loss.backward() # обратное распространение, вычисление градиентов
54    optimizer.step() # градиентное восхождение, обновление весов
55    return loss
56
57 def main():
58    env = gym.make('CartPole-v0')
59    in_dim = env.observation_space.shape[0] # 4
60    out_dim = env.action_space.n # 2
61    pi = Pi(in_dim, out_dim) # стратегия pi_theta для REINFORCE
62    optimizer = optim.Adam(pi.parameters(), lr=0.01)
63    for epi in range(300):
64        state = env.reset()
65        for t in range(200): # 200 — максимальное количество шагов в cartpole
66            action = pi.act(state)
67            state, reward, done, _ = env.step(action)
68            pi.rewards.append(reward)
69            env.render()
70            if done:
71                break
72        loss = train(pi, optimizer) # обучение в эпизоде
73        total_reward = sum(pi.rewards)
74        solved = total_reward > 195.0
75        pi.onpolicy_reset() # обучение по актуальному опыту:
          ➥ очистить память после обучения
76        print(f'Episode {epi}, loss: {loss}, \
77        total_reward: {total_reward}, solved: {solved}')
78
79 if __name__ == '__main__':
80 main()
```

Пройдемся по минимальной реализации.

1. `Pi` строит сеть стратегии — простой многослойный перцептрон с одним слоем из 64 скрытых элементов (строки 10–20).
2. `act` определяет метод, с помощью которого выполняются действия (строки 30–37).
3. `train` реализует шаги обновления в алгоритме 2.1. Заметьте, что функция потерь выражена как сумма отрицательных логарифмов вероятностей, умноженных на отдачи (строки 50–51). Знак минус необходим, так как по умолчанию оптимизатор PyTorch минимизирует потери, тогда как нам нужно максимизировать целевую функцию. Более того, функция потерь сформулирована таким образом, чтобы воспользоваться функцией автоматического дифференцирования

PyTorch. При вызове `loss.backward()` вычисляется градиент функции потерь (строка 53), который равен градиенту стратегии. Наконец, посредством вызова `optimizer.step()` (строка 54) обновляются параметры стратегии.

4. `main` — это основной цикл, в котором создаются среда CartPole, сеть стратегии `Pi` и оптимизатор. Затем запускается цикл обучения по 300 эпизодам. В ходе обучения полное вознаграждение за эпизод должно расти до значения 200. Задача в среде решена, когда полное вознаграждение выше 195.

2.6.2. Построение стратегий с помощью PyTorch

Реализация стратегии πθ заслуживает более пристального рассмотрения. В этом разделе будет показано, как выходные данные нейронной сети преобразуются в распределение вероятностей действий, используемое для выбора действий, $a \sim \pi_\theta(s)$.

Основная идея состоит в том, что распределение вероятностей может быть параметризировано двумя путями. Первый заключается в вычислении полных вероятностей для дискретного распределения. Второй — в указании среднего значения и среднего квадратичного отклонения[1] для непрерывного распределения, такого как нормальное распределение. Эти параметры распределения вероятностей могут быть настроены нейронной сетью и получены на ее выходе.

Чтобы получить на выходе действие, сначала применяется сеть стратегии для вычисления параметров распределения вероятностей по состояниям. Затем эти параметры используются для построения распределения вероятностей действий. Наконец, исходя из распределения вероятностей, выбирается действие и вычисляется его логарифм вероятности.

Рассмотрим, как это работает в псевдокоде. В алгоритме 2.2 строится распределение вероятностей для дискретных действий, которое используется для вычисления логарифма вероятности. Здесь строится распределение по категориям (полиномиальное), но могут быть задействованы и другие виды дискретных распределений. Поскольку выходные данные сети не обязательно нормализованы, параметры распределения обрабатываются как логиты, а не вероятности. Алгоритм 2.2 приближен к реализации стратегии в листинге 2.1 (строки 31–35).

Алгоритм 2.2. Построение дискретной стратегии

```
Заданы сеть стратегии net, класс распределения
   ➥ по категориям Categorical и состояние state
Вычислить выходное значение pdparams = net(state)
Построить экземпляр распределения вероятностей
   ➥ одного действия pd = Categorical(logits=pdparams)
```

[1] Для отдельных распределений вероятностей могут потребоваться специальные параметры, но эта же идея применима и к ним.

```
4: Использовать pd для выбора действия, action = pd.sample()
5: С помощью pd и action вычислить логарифм вероятности
   ➥ действия, log_prob = pd.log_prob(action)
```

В листинге 2.2 приведена простая реализация алгоритма 2.2 с помощью PyTorch, которая в значительной степени соответствует его шагам. Для простоты используются фиктивные выходные данные сети стратегии. PyTorch преобразует указанные логиты в вероятности внутри конструктора класса `Categorial` при построении распределения по категориям.

Листинг 2.2. Реализация дискретной стратегии с помощью распределения по категориям

```
 1 from torch.distributions import Categorical
 2 import torch
 3
 4 # считаем, что для двух действий (движений влево и вправо в CartPole)
 5 # из сети стратегии были получены логиты вероятностей
 6 policy_net_output = torch.tensor([-1.6094, -0.2231])
 7 # pdparams — это логиты, соответствующие probs = [0.2, 0.8]
 8 pdparams = policy_net_output
 9 pd = Categorical(logits=pdparams)
10
11 # выбор действия
12 action = pd.sample()
13 # => tensor(1), или движение вправо
14
15 # вычисление логарифма вероятности действия
16 pd.log_prob(action)
17 # => tensor(-0.2231), логарифм вероятности движения вправо
```

Стратегия для непрерывных действий строится аналогичным образом. Алгоритм 2.3 демонстрирует пример применения нормального распределения. Как и многие другие распределения, оно параметризируется средним значением и средним квадратичным отклонением. В коде многих библиотек для научных вычислений этим параметрам соответствуют `loc` и `scale`, под которыми подразумевается регулирование сдвига и масштаба распределения. После построения распределения вероятности действий остальные шаги по вычислению логарифма вероятностей действий такие же, как и в алгоритме 2.2.

Алгоритм 2.3. Построение непрерывной стратегии

```
1: Заданы сеть стратегии net, класс нормального
   ➥ распределения Normal и состояние state
2: Вычислить выходное значение pdparams = net(state)
3: Построить распределение вероятностей одного
   ➥ действия pd = Normal(loc=pdparams[0], scale=pdparams[1])
4: Использовать pd для выбора действия, action = pd.sample()
5: С помощью pd и action вычислить логарифм вероятности
   ➥ действия, log_prob = pd.log_prob(action)
```

Для полноты изложения приводится также реализация алгоритма 2.3 с помощью PyTorch (листинг 2.3).

Листинг 2.3. Реализация непрерывной стратегии с помощью нормального распределения

```
from torch.distributions import Normal
import torch

# считаем, что для одного действия (крутящего момента маятника)
# сеть стратегии выдала среднее значение mean
  ➥ и среднее квадратичное отклонение std
policy_net_output = torch.tensor([1.0, 0.2])
# pdparams — это (mean, std) или (loc, scale)
pdparams = policy_net_output
pd = Normal(loc=pdparams[0], scale=pdparams[1])

# выбор действия
action = pd.sample()
# => tensor(1.0295), величина крутящего момента

# вычисление логарифма вероятности действия
pd.log_prob(action)
# => tensor(-0.2231), логарифм вероятности этого крутящего момента
```

Универсальность процесса построения стратегии позволяет легко применить его к средам как с дискретными, так и с непрерывными действиями. Его простота — еще одна сильная сторона алгоритмов, основанных на стратегии.

Алгоритмы 2.2 и 2.3 реализованы в SLM Lab в обобщенной для всех основанных на стратегии алгоритмов форме как модули, которые можно использовать повторно. Кроме того, эти реализации оптимизированы для повышения производительности. Например, для ускорения вычислений все логарифмы вероятности действий рассчитываются одновременно и только во время обучения.

Рассмотрев очень простую реализацию REINFORCE, теперь обсудим, как он реализован в SLM Lab и как интегрирован с ее компонентами. Это послужит введением в реализацию алгоритмов с помощью фреймворка SLM Lab. Далее в книге код REINFORCE используется в качестве родительского класса, путем расширения которого реализованы более сложные алгоритмы. Поскольку все последующие эксперименты будут запускаться с помощью SLM Lab, пришло время познакомиться с этим фреймворком.

В следующих разделах обсуждаются лишь методы, специфические для рассматриваемых алгоритмов. Их интеграция с компонентами фреймворка библиотеки оставлена в качестве практического задания по изучению кода. Полный код со всеми компонентами, необходимыми для запуска обучения в RL, можно найти в прилагаемом репозитории кода SLM Lab.

2.6.3. Выборка действий

Разберем подробно методы выборки действий класса `Reinforce`, приведенные в листинге 2.4. Обратите внимание на то, что на протяжении всей книги для ясности изложения некоторые строки кода, несущественные для пояснения функциональности методов, опущены или заменены на «...». Кроме того, методы с тегом `@lab_api` — это стандартные методы API алгоритмов в SLM Lab.

`calc_pdparam` рассчитывает параметры распределения действий и вызывается методом `self.action_policy` (строка 15), который производит действие (это подробнее описано в подразделе 2.6.2). `act` выполняет выборку действия по стратегии действий, которая может быть представлена непрерывным или дискретным распределением, например `Categorical` или `Normal`.

Листинг 2.4. Реализация REINFORCE, выборка действий по сети стратегии

```
 1 # slm_lab/agent/algorithm/reinforce.py
 2
 3 class Reinforce(Algorithm):
 4     ...
 5
 6    @lab_api
 7    def calc_pdparam(self, x, net=None):
 8        net = self.net if net is None else net
 9        pdparam = net(x)
10        return pdparam
11
12    @lab_api
13    def act(self, state):
14        body = self.body
15        action = self.action_policy(state, self, body)
16        return action.cpu().squeeze().numpy()
17        # применяем squeeze для преобразования скалярной величины
```

2.6.4. Расчет потерь, обусловленных стратегией

В листинге 2.5 приведен метод, который в два этапа вычисляет функцию потерь для стратегии. Сначала нужно рассчитать подкрепляющий сигнал, что делается с помощью метода `calc_ret_advs`, подсчитывающего значения отдачи для каждого элемента в пакете (batch). При необходимости можно выделить базовое значение из отдачи. В данном примере рассчитываются только средние значения отдачи в пакете.

Имея значения отдачи, можно найти функцию потерь для стратегии. В `calc_policy_loss` мы сначала получаем распределение вероятностей действий (строка 15), необходимое для подсчета логарифма вероятностей действий (строка 18).

Затем объединяем значения отдачи (обозначены `advs`) с логарифмами вероятностей, чтобы образовать значение потерь, обусловленных стратегией (строка 19). Это выполняется в той же форме, что и в листинге 2.1 (строки 50–51), поэтому можно воспользоваться автоматической функцией дифференцирования PyTorch.

При желании дополнительно к функции потерь можно найти параметр энтропии для улучшения исследования среды (строки 20–23). Более детально это обсуждается в разделе 6.3.

Листинг 2.5. Реализация REINFORCE, расчет функции потерь для стратегии

```
 1 # slm_lab/agent/algorithm/reinforce.py
 2
 3 class Reinforce(Algorithm):
 4    ...
 5
 6    def calc_ret_advs(self, batch):
 7        rets = math_util.calc_returns(batch['rewards'], batch['dones'],
          ➥ self.gamma)
 8        if self.center_return:
 9            rets = math_util.center_mean(rets)
10        advs = rets
11        ...
12        return advs
13
14    def calc_policy_loss(self, batch, pdparams, advs):
15        action_pd = policy_util.init_action_pd(self.body.ActionPD, pdparams)
16        actions = batch['actions']
17        ...
18        log_probs = action_pd.log_prob(actions)
19        policy_loss = - self.policy_loss_coef * (log_probs * advs).mean()
20        if self.entropy_coef_spec:
21            entropy = action_pd.entropy().mean()
22            self.body.mean_entropy = entropy # обновление логируемой переменной
23            policy_loss += (-self.body.entropy_coef * entropy)
24        return policy_loss
```

2.6.5. Цикл обучения в REINFORCE

В листинге 2.6 показаны цикл обучения и связанный с ним метод выборки из памяти. `train` обновляет один параметр сети стратегии с помощью пакета накопленных траекторий. Обучение начнется, как только будет собрано достаточно данных. Получение траекторий из памяти агента происходит посредством вызова `sample` (строка 17).

После выборки пакета данных вычисляются параметры распределения вероятностей действий `pdparams` и значения отдачи `advs`, которые используются для расчета функции потерь для стратегии (строки 19–21). Затем с помощью этих потерь обновляются параметры сети стратегии (строка 22).

Листинг 2.6. Реализация REINFORCE, метод обучения

```
 1 # slm_lab/agent/algorithm/reinforce.py
 2
 3 class Reinforce(Algorithm):
 4     ...
 5
 6     @lab_api
 7     def sample(self):
 8         batch = self.body.memory.sample()
 9         batch = util.to_torch_batch(batch, self.net.device,
            ➥ self.body.memory.is_episodic)
10         return batch
11
12     @lab_api
13     def train(self):
14         ...
15         clock = self.body.env.clock
16         if self.to_train == 1:
17             batch = self.sample()
18             ...
19             pdparams = self.calc_pdparam_batch(batch)
20             advs = self.calc_ret_advs(batch)
21             loss = self.calc_policy_loss(batch, pdparams, advs)
22             self.net.train_step(loss, self.optim, self.lr_scheduler,
               ➥ clock=clock, global_net=self.global_net)
23             # сброс параметров
24             self.to_train = 0
25             return loss.item()
26         else:
27             return np.nan
```

2.6.6. Класс Memory для хранения примеров при обучении по актуальному опыту

В этом разделе рассматривается класс памяти, в котором реализована генерация выборок при обучении по актуальному опыту. В строке 17 листинга 2.6 можно видеть, что объект памяти вызывается, чтобы получить траектории для обучения. Поскольку REINFORCE — это алгоритм обучения *по актуальному опыту*, выбранные им траектории должны храниться в классе `Memory` для обучения и отбрасываться после каждого шага обновления весов.

В данном разделе приводятся сведения о том, какая информация хранится в памяти и как она используется в цикле обучения. Читатели, которым не нужны детали классов памяти, могут пропустить его без ущерба для понимания алгоритма REINFORCE.

Рассмотрим класс `OnPolicyReplay`, в котором реализована эта логика. Он содержит следующие методы API:

- `reset` — очищает память и устанавливает начальные значения для переменных класса памяти;
- `update` — добавляет пример опыта в память;
- `sample` — производит выборку пакета данных для обучения.

Инициализация, очистка и установление начальных значений переменных памяти. `__init__` инициализирует переменные класса, включая ключи хранилища в строке 15. Затем он вызывает `reset`, чтобы создать пустую структуру данных.

В листинге 2.7 `reset` используется для очистки памяти после каждого шага цикла обучения. Это присуще только обучению по актуальному опыту, поскольку траектории не могут повторно применяться для последующих циклов обучения.

Класс памяти может хранить траектории из большого количества эпизодов в атрибутах, инициализируемых в строках 21 и 22. Отдельные эпизоды строятся посредством хранения опыта в словаре данных текущего эпизода `self.cur_epi_data`, который очищается в строке 23.

Листинг 2.7. OnPolicyReplay, очистка и установка начальных значений переменных

```
 1 # slm_lab/agent/memory/onpolicy.py
 2
 3 class OnPolicyReplay(Memory):
 4     ...
 5
 6     def __init__(self, memory_spec, body):
 7         super().__init__(memory_spec, body)
 8         # ВНИМАНИЕ: для примеров в обучении по актуальному опыту
           ➥ frequency = episode, для других приведенных далее
           ➥ классов frequency = frames
 9         util.set_attr(self, self.body.agent.agent_spec['algorithm'],
           ➥ ['training_frequency'])
10         # Не нужно очищать все примеры опыта, когда память эпизодическая
11         self.is_episodic = True
12         self.size = 0 # все примеры опыта сохранены
13         self.seen_size = 0 # все примеры опыта просмотрены накопительно
14         # объявляется, какие ключи нужно хранить
```

```
15        self.data_keys = ['states', 'actions', 'rewards', 'next_states',
          ➥ 'dones']
16        self.reset()
17
18    @lab_api
19    def reset(self):
20        '''Очищает память, используется для инициализации переменных памяти'''
21        for k in self.data_keys:
22            setattr(self, k, [])
23        self.cur_epi_data = {k: [] for k in self.data_keys}
24        self.most_recent = (None,) * len(self.data_keys)
25        self.size = 0
```

Обновление памяти. Функция `update` играет роль метода API для класса памяти. Добавление опыта в память, по большому счету, происходит просто, единственная сложная часть — это отслеживание границ эпизода. Код в листинге 2.8 можно разделить на следующие шаги.

1. Добавление прецедента в текущий эпизод (строки 14 и 15).
2. Проверка того, завершился ли эпизод (строка 17). Для этого служит переменная `done`, которая принимает значение 1, если эпизод завершен, и 0 — в противном случае.
3. Если эпизод окончен, добавление всего набора прецедентов для эпизода в основные контейнеры класса памяти (строки 18 и 19).
4. Если эпизод завершен, очистка словаря текущего эпизода (строка 20), чтобы подготовить класс памяти для хранения следующего эпизода.
5. Если было накоплено желаемое количество эпизодов, установка в агенте для флага обучения значения 1 (строки 23 и 24). Это сигнализирует агенту о том, что он должен обучиться данному временному шагу.

Листинг 2.8. OnPolicyReplay, добавление прецедентов

```
 1 # slm_lab/agent/memory/onpolicy.py
 2
 3 class OnPolicyReplay(Memory):
 4    ...
 5
 6    @lab_api
 7    def update(self, state, action, reward, next_state, done):
 8        '''Метод интерфейса для обновления памяти'''
 9        self.add_experience(state, action, reward, next_state, done)
10
11    def add_experience(self, state, action, reward, next_state, done):
12        '''Вспомогательный метод интерфейса для update()
          ➥ для добавления опыта в память'''
```

```
        self.most_recent = (state, action, reward, next_state, done)
        for idx, k in enumerate(self.data_keys):
            self.cur_epi_data[k].append(self.most_recent[idx])
        # Если эпизод окончен, добавить его в память и очистить cur_epi_data
        if util.epi_done(done):
            for k in self.data_keys:
                getattr(self, k).append(self.cur_epi_data[k])
            self.cur_epi_data = {k: [] for k in self.data_keys}
            # Если агент накопил желаемое количество
            ➥ эпизодов, то он готов к обучению
            # длина — это количество эпизодов с учетом вложенной структуры
            if len(self.states) == self.body.agent.algorithm.training_
            ➥ frequency:
                    self.body.agent.algorithm.to_train = 1
        # Отслеживание размера памяти и количества примеров опыта
        self.size += 1
        self.seen_size += 1
```

Выборка из памяти. В листинге 2.9 `sample` просто возвращает все полные эпизоды, упакованные в словарь `batch` (строка 6). Затем он очищает память и устанавливает начальные значения переменных (строка 7), поскольку после завершения шага обучения агентом сохраненные прецеденты больше не действительны.

Листинг 2.9. OnPolicyReplay, выборка

```
# slm_lab/agent/memory/onpolicy.py

class OnPolicyReplay(Memory):

    def sample(self):
        batch = {k: getattr(self, k) for k in self.data_keys}
        self.reset()
        return batch
```

2.7. Обучение агента в REINFORCE

У алгоритмов глубокого обучения с подкреплением зачастую много гиперпараметров. Например, должны быть указаны тип сети, архитектура, функции активации, оптимизатор и скорость обучения. Более продвинутая функциональность нейронной сети может включать ограничение градиентов и снижение скорости обучения. И это только «глубокая» часть! Алгоритмы обучения с подкреплением подразумевают также выбор гиперпараметров, таких как коэффициент дисконтирования γ и частота обучения агента. Для упрощения управления все выбранные значения гиперпараметров указаны в SLM Lab в файле формата JSON под названием `spec` (то есть спецификация). Более подробные сведения о файле `spec` вы найдете в главе 11.

Листинг 2.10 содержит пример файла `spec` для REINFORCE. Этот файл имеется также в SLM Lab в `slm_lab/spec/benchmark/reinforce/reinforce_cartpole.json`.

Листинг 2.10. Простой файл spec для игры в cartpole с помощью метода REINFORCE

```
# slm_lab/spec/benchmark/reinforce/reinforce_cartpole.json

{
   "reinforce_cartpole": {
       "agent": [{
           "name": "Reinforce",
           "algorithm": {
               "name": "Reinforce",
               "action_pdtype": "default",
               "action_policy": "default",
               "center_return": true,
               "explore_var_spec": null,
               "gamma": 0.99,
               "entropy_coef_spec": {
                   "name": "linear_decay",
                   "start_val": 0.01,
                   "end_val": 0.001,
                   "start_step": 0,
                   "end_step": 20000,
               },
               "training_frequency": 1
           },
           "memory": {
               "name": "OnPolicyReplay"
           },
           "net": {
               "type": "MLPNet",
               "hid_layers": [64],
               "hid_layers_activation": "selu",
               "clip_grad_val": null,
               "loss_spec": {
                   "name": "MSELoss"
               },
               "optim_spec": {
                   "name": "Adam",
                   "lr": 0.002
               },
               "lr_scheduler_spec": null
           }
       }],
       "env": [{
           "name": "CartPole-v0",
           "max_t": null,
           "max_frame": 100000,
       }],
       "body": {
```

```
47            "product": "outer",
48            "num": 1
49        },
50        "meta": {
51            "distributed": false,
52            "eval_frequency": 2000,
53            "max_session": 4,
54            "max_trial": 1,
55        },
56        ...
57    }
58 }
```

Может показаться, что здесь слишком много настроек, поэтому стоит пройтись по основным компонентам.

- **Алгоритм** — это REINFORCE (строка 8). Значение γ установлено в строке 13. Поскольку в строке 11 отключено `center_return`, в качестве подкрепляющего сигнала используется базовое значение. Чтобы стимулировать изучение, к функции потерь добавляется энтропия с линейной скоростью уменьшения значения (строки 14–20).
- **Архитектура сети** — многослойный перцептрон с одним скрытым слоем, включающим 64 элемента и функцию активации SeLU (строки 27–29).
- **Оптимизатор** — это Adam [68] со скоростью обучения 0,002 (строки 34–37). Скорость обучения всегда постоянная, поскольку схеме скорости обучения присвоено значение `null` (строка 38).
- **Частота обучения** — обучение по эпизодам, так как выбрана память `OnPolicyReplay` (строка 24) и агент обучается в конце каждого эпизода. Это контролируется параметром `training_frequency` (строка 21), значение 1 которого подразумевает, что сеть будет обучаться на каждом эпизоде.
- **Среда** — CartPole из OpenAI Gym [18] (строка 42).
- **Длительность обучения** — 100 000 временных шагов (строка 44).
- **Оценка** — агент оценивается каждые 2000 временных шагов (строка 52). Во время оценки запускаются четыре эпизода, затем подсчитываются средние полные вознаграждения и выдается отчет о результатах.

Чтобы обучить агента REINFORCE с помощью SLM Lab, запустите в терминале команды из листинга 2.11.

Листинг 2.11. Обучение агента REINFORCE

```
1 conda activate lab
2 python run_lab.py slm_lab/spec/benchmark/reinforce/reinforce_cartpole.json
  ➥ reinforce_cartpole train
```

Здесь с помощью указанного файла `spec` будет запущено пробное обучение `Trial`[1], состоящее из четырех повторяющихся сеансов `Session`. Средний результат затем будет отображен на графике для пробных испытаний с интервалом значений погрешностей. Дополнительно график результатов пробных испытаний выводится также со скользящим средним по оценкам в 100 контрольных точках, чтобы получить сглаженную кривую. Оба графика показаны на рис. 2.2[2].

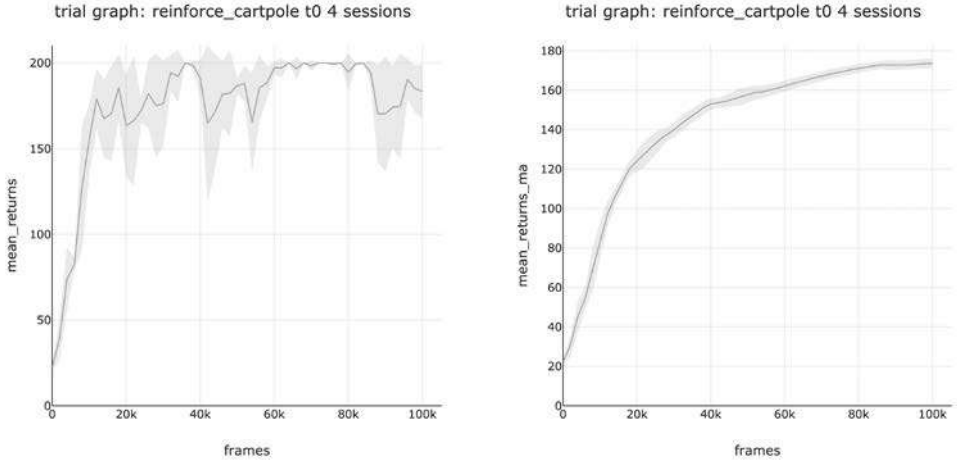


Рис. 2.2. Графики результатов пробных испытаний для метода REINFORCE из SLM Lab, построенные по средним значениям для четырех повторных сессий. По вертикали отложены полные вознаграждения (обозначены mean_returns), усредненные по восьми эпизодам на протяжении контрольных точек. Горизонтальная ось соответствует всем кадрам обучения. Справа приведена сглаженная версия графика, отображающая скользящее среднее по оценкам в 100 контрольных точках. Следует отметить, что максимальное полное вознаграждение в CartPole составляет 200

2.8. Результаты экспериментов

В этом разделе мы воспользуемся функцией экспериментирования SLM Lab для изучения влияния отдельных компонентов алгоритма на его работу. В первом эксперименте сравниваются результаты применения разных значений коэффициента дисконтирования γ. Во втором — демонстрируется улучшение благодаря использованию базового значения в подкрепляющем сигнале.

---

[1] Trial должен запускаться с большим количеством Session, причем с одним и тем же spec, но разными случайными начальными числами. В SLM Lab это делается автоматически.

[2] Полные вознаграждения обозначены на графиках как mean_returns. Для оценки mean_returns вычисляются без дисконтирования.

2.8.1. Эксперимент по оценке влияния коэффициента дисконтирования γ

Коэффициент дисконтирования γ регулирует вес будущих вознаграждений при вычислении отдачи $R(\tau)$, которая используется как подкрепляющий сигнал. Чем больше значение коэффициента дисконтирования, тем больший вес имеют будущие вознаграждения. Оптимальное значение γ обусловлено задачей. Для задач, где агенту нужно учитывать влияние своих действий на будущие вознаграждения при длительном временно́м горизонте, значение γ должно быть выше.

Чтобы изучить влияние коэффициента дисконтирования, мы можем запустить алгоритм REINFORCE и сравнить результаты с разными значениями γ. Для этого измените файл `spec` из листинга 2.10, добавив спецификацию `search` для γ. Это показано в листинге 2.12, где в строке 15 определяется сеточный поиск по гиперпараметру `gamma`. В SLM Lab полный файл `spec` содержится в `slm_lab/spec/benchmark/reinforce/reinforce_cartpole.json`.

Листинг 2.12. Файл spec для REINFORCE со спецификацией search для разных значений gamma

```
 1 # slm_lab/spec/benchmark/reinforce/reinforce_cartpole.json
 2
 3 {
 4    "reinforce_cartpole": {
 5        ...
 6        "meta": {
 7            "distributed": false,
 8            "eval_frequency": 2000,
 9            "max_session": 4,
10            "max_trial": 1,
11        },
12        "search": {
13            "agent": [{
14                "algorithm": {
15                    "gamma__grid_search": [0.1, 0.5, 0.7, 0.8, 0.90,
                      ➥ 0.99, 0.999]
16                }
17            }]
18        }
19    }
20 }
```

Чтобы запустить эксперимент в SLM Lab, воспользуйтесь командами, приведенными в листинге 2.13. Эксперимент незначительно отличается от обучения аген-

та — мы применяем один и тот же файл `spec`, изменив режим обучения `train` на режим поиска `search`.

Листинг 2.13. Запуск эксперимента по проверке разных значений gamma, заданных в файле spec

```
1 conda activate lab
2 python run_lab.py slm_lab/spec/benchmark/reinforce/reinforce_cartpole.json
  ➥ reinforce_cartpole search
```

Здесь будет запущен эксперимент `Experiment`, который породит множество пробных испытаний `Trial`. Для каждого из них используется свое значение `gamma`, которое подставляется в изначальный файл `spec` для REINFORCE. В каждом `Trial` запускаются четыре повторяющиеся сессии `Session`, чтобы получить среднее значение с доверительным интервалом. Оно применяется для построения графика результатов множества пробных испытаний. Для облегчения сравнения также приводится сглаженный график со скользящим средним по оценкам в 100 контрольных точках. Они изображены на рис. 2.3.

На рис. 2.3 показано, что для $\gamma > 0{,}9$ производительность выше. Наилучший результат достигается в шестом пробном испытании при $\gamma = 0{,}999$. Когда значение γ слишком низкое, алгоритм не может настроить стратегию, позволяющую решить задачу, и кривая обучения остается пологой.

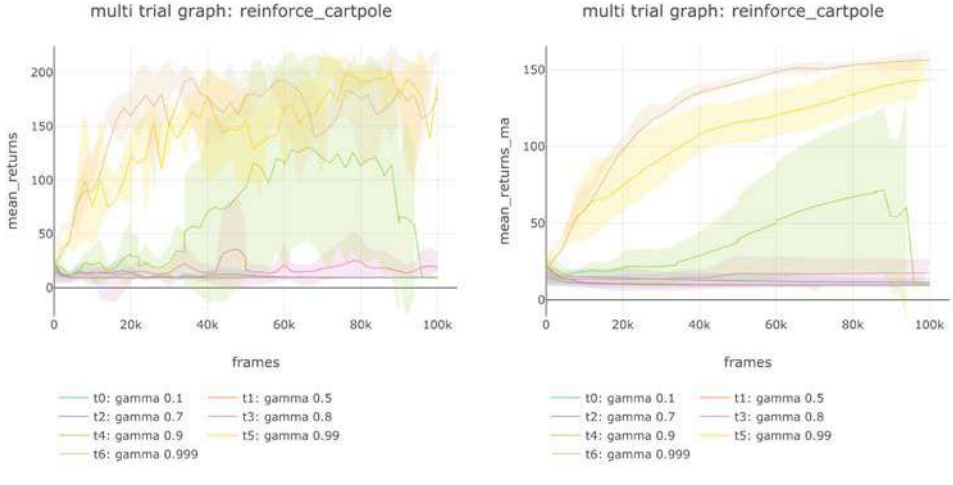


Рис. 2.3. Влияние различных значений коэффициента дисконтирования γ. Для CartPole низкие значения γ приводят к плохой производительности. При $\gamma = 0{,}999$ в шестом пробном испытании достигается наилучший результат. Как правило, для разных задач оптимальные значения γ могут различаться

2.8.2. Эксперимент по оценке влияния базового значения

В этой главе мы узнали, что применение базового значения может помочь уменьшить дисперсию при оценке градиентов стратегии методом Монте-Карло. Давайте проведем эксперимент, чтобы сравнить производительность REINFORCE с использованием и без использования базового значения. Оно может быть включено или отключено с помощью гиперпараметра `center_return` в файле `spec` алгоритма.

Для проведения эксперимента скопируем изначальный файл `spec` для REINFORCE и переименуем его в `reinforce_baseline_cartpole`. Частично этот файл `spec` приведен в листинге 2.14. Затем добавим спецификацию `search` для выполнения поиска по сетке посредством включения и отключения `center_return` (строка 15). Файл `spec` доступен в SLM Lab по адресу `slm_lab/spec/benchmark/reinforce/reinforce_cartpole.json`.

Листинг 2.14. Файл spec для REINFORCE со спецификацией search с включением и отключением базового значения (center_return)

```
 1 # slm_lab/spec/benchmark/reinforce/reinforce_cartpole.json
 2
 3 {
 4    "reinforce_baseline_cartpole": {
 5        ...
 6        "meta": {
 7            "distributed": false,
 8            "eval_frequency": 2000,
 9            "max_session": 4,
10            "max_trial": 1,
11        },
12        "search": {
13            "agent": [{
14                "algorithm": {
15                    "center_return__grid_search": [true, false]
16                }
17            }]
18        }
19    }
20 }
```

Воспользуйтесь командами из листинга 2.15 для запуска эксперимента в SLM Lab.

Листинг 2.15. Запуск эксперимента по сравнению производительности с базовым значением и без него в соответствии с файлом spec

```
1 conda activate lab
2 python run_lab.py slm_lab/spec/benchmark/reinforce/reinforce_cartpole.json
  ➥ reinforce_baseline_cartpole search
```

Таким образом будет запущен эксперимент `Experiment` с двумя пробными испытаниями `Trial`, в каждом из которых будет по четыре сессии `Session`. На рис. 2.4 приведены график результатов множества пробных испытаний и его скользящее среднее по оценкам в 100 контрольных точках.

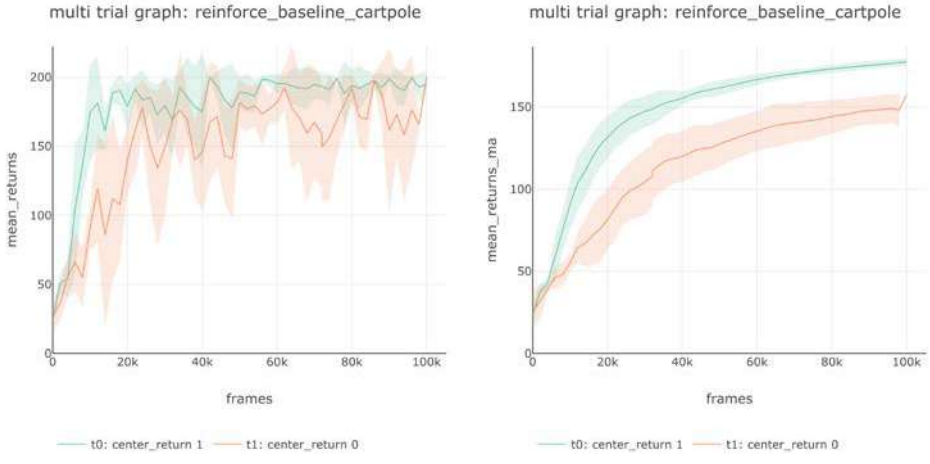


Рис. 2.4. На этих графиках показано, что при использовании базового значения для уменьшения дисперсии производительность возрастает

Как и ожидалось, производительность REINFORCE с применением базового значения в сравнении с версией без базового значения получилась выше за счет уменьшения дисперсии оценок градиентов стратегии. На рис. 2.4 показано, что версия с базовым значением из пробного испытания с номером 0 обучается быстрее и получает бóльшие полные вознаграждения.

Этот раздел был минимально возможным введением в возможности функции экспериментирования SLM Lab по запуску отдельных экспериментов. Более детально проектирование экспериментов с помощью SLM Lab рассматривается в главе 11. Эта функция понадобится нам лишь в сложных случаях и не является необходимой для понимания алгоритмов, обсуждаемых в этой книге.

2.9. Резюме

В этой главе дано введение в REINFORCE — алгоритм градиента стратегии. Его основная идея — настройка параметров сети стратегии для максимизации целевой функции, которой является ожидаемая отдача $J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[R(\tau)\right]$ для агента.

Мы дали определение градиента, продифференцировали его и получили элегантное выражение. Оно позволяет настраивать вероятности действий для стратегии так, чтобы поощрять хорошие действия и не поощрять плохие. В REINFORCE градиент

стратегии вычисляется с помощью выборки методом Монте-Карло. Эта оценка может иметь высокую дисперсию, и общепринятым методом ее уменьшения является введение базового значения.

Реализовать этот алгоритм довольно просто, как показано в листинге 2.1. Была также рассмотрена его реализация в SLM Lab в качестве введения в API алгоритма, который используется на протяжении всей книги.

2.10. Рекомендуемая литература

- *Levine S.* Sep 6: Policy Gradients Introduction, Lecture 4. 2017 [74].
- *Fei-Fei L., Johnson J., Yeung S.* Lecture 14: Reinforcement Learning. 2017 [39].
- *Metropolis N.* The Beginning of the Monte Carlo Method. 1987 [85].

2.11. Историческая справка

Метод Монте-Карло был популяризован Станиславом Уламом, работавшим в исследовательской лаборатории Лос-Аламос в 1940-х годах. В названии «Монте-Карло» нет какого-то особого смысла, это просто запоминающаяся альтернатива термину «выборочная оценка». Но его происхождение все же занятное. Оно было предложено Николасом Метрополисом, физиком и конструктором вычислительных машин, ответственным за ироничное название компьютера MANIAC [5]. Метрополис услышал о дяде Улама, который брал деньги в долг у родственников, потому что «ему просто нужно поехать в Монте-Карло». После этого название казалось совершенно очевидным [85].

В то же время в Пенсильванском университете в Филадельфии был усовершенствован компьютер ENIAC. Огромная машина, состоящая из более чем 18 000 электронных ламп, была одним из наиболее ранних электронных компьютеров общего назначения. Вычислительная мощь и гибкость ENIAC произвели впечатление на Улама. Он посчитал его подходящим инструментом для многочисленных регулярных вычислений, необходимых для применения статистических методов в оценке функций. Его коллега по Лос-Аламосу выдающийся ученый-универсал Джон фон Нейман сразу же увидел полезность подхода Монте-Карло. В 1947 году он схематично описал этот метод в записке, поданной главе теоретического отделения Лос-Аламоса [85]. Более подробные сведения о возникновении методов Монте-Карло можно почерпнуть из статьи Николаса Метрополиса *The Beginning of the Monte Carlo Method*, опубликованной в специальном издании *Los Alamos Science* в 1987 году.

3 SARSA

В этой главе рассматривается SARSA — наш первый алгоритм, основанный на полезности. Он был предложен Руммери и Ниранжаном в 1994 году и описан в статье *On-Line Q-Learning Using Connectionist Systems* [118]. Он получил такое название, так как для его работы «нужно знать последовательность значений "состояние — действие — вознаграждение — состояние — действие" (State — Action — Reward — State — Action) перед выполнением обновления»[1].

Основанные на полезности алгоритмы оценивают пары «состояние — действие» (s, a) посредством настройки одной из функций полезности — $V^{\pi}(s)$ или $Q^{\pi}(s, a)$ — и используют полученные оценки для выбора действий. Настройка функций полезности противоположна непосредственному обучению агента стратегии, сопоставляющей состояниям действия, применяемому в REINFORCE (см. главу 2). Алгоритм SARSA настраивает функцию $Q^{\pi}(s, a)$, тогда как другие алгоритмы, такие как алгоритм актора-критика, — $V^{\pi}(s)$. В разделе 3.1 объясняется, почему в данном случае функция $Q^{\pi}(s, a)$ — хороший выбор.

В основе алгоритма SARSA лежат две идеи. Первая — это метод настройки Q-функций, известный как метод временных различий (temporal difference), или TD-обучение. Он является альтернативой выборке методом Монте-Карло при оценке состояний или полезности пар «состояние — действие» на основе опыта, полученного агентом в среде. TD-обучение — тема раздела 3.2.

Вторая идея — это метод порождения действий с помощью Q-функций. И здесь возникает вопрос: как агенты выбирают хорошие действия? Одной из фундаментальных проблем RL является достижение оптимального баланса между использованием знаний агента и исследованием среды для обучения новым навыкам. Проблема выбора между *использованием стратегии* и *исследованием среды* обсуждается в разделе 3.3. Здесь также рассказывается о простом подходе к ее решению — ε-жадной стратегии.

---

[1] В действительности SARSA не был назван так Руммери и Ниранжаном в их статье 1994 года. Авторы предпочли термин «модифицированное Q-обучение нейронных сетей» (Modified Connectionist Q-Learning). Ричард Саттон предложил альтернативное название, которое и прижилось.

После знакомства с идеями, лежащими в основе метода SARSA, в разделе 3.4 приводится сам алгоритм, а в разделе 3.5 — его реализация. В конце главы даны инструкции по обучению агента с помощью SARSA.

3.1. Q-функция и V-функция

В этом разделе рассказывается, почему SARSA настраивает *Q*-функцию, а не *V*-функцию. Даны определения этих функций и интуитивно понятное объяснение того, что измеряет каждая из них, далее описываются преимущества *Q*-функции.

В разделе 1.3 были введены две функции полезности — $V^{\pi}(s)$ и $Q^{\pi}(s, a)$. *Q*-функция измеряет полезность пары «состояние — действие» (s, a) при определенной стратегии π (3.1). Полезность (s, a) — это ожидаемое суммарное дисконтированное вознаграждение за выбор действия a в состоянии s и последующие действия в соответствии со стратегией π:

$$Q^{\pi}(s, a) = \mathbb{E}_{s_{t+1}=s,\, a_t=a,\, \tau\sim\pi}\left[\sum_{t=0}^{T}\gamma^t r_t\right]. \tag{3.1}$$

Функции полезности всегда определяются с учетом конкретной стратегии π, в связи с чем в их обозначении присутствует верхний индекс π. Чтобы пояснить это, предположим, что мы оцениваем (s, a). $Q^{\pi}(s, a)$ зависит от последовательности вознаграждений, которые агент может получить после выбора действия a в состоянии s. Эти вознаграждения обусловлены будущими цепочками действий, а те, в свою очередь, — стратегией. Разные стратегии могут порождать различные будущие последовательности действий для данной пары (s, a), в результате чего вознаграждения будут различаться.

$V^{\pi}(s)$ измеряет полезность состояния s при определенной стратегии π (3.2). Полезность состояния $V^{\pi}(s)$ — это ожидаемое суммарное дисконтированное вознаграждение, начиная с состояния s и далее в соответствии с определенной стратегией π:

$$V^{\pi}(s) = \mathbb{E}_{s_{t+1}=s,\, \tau\sim\pi}\left[\sum_{t=0}^{T}\gamma^t r_t\right]. \tag{3.2}$$

$Q^{\pi}(s, a)$ тесно связана с $V^{\pi}(s)$, которая является математическим ожиданием *Q*-значений для всех действий a, доступных в отдельно взятом состоянии s при стратегии π:

$$V^{\pi}(s) = \mathbb{E}_{a\sim\pi(s)}\left[Q^{\pi}(s, a)\right]. \tag{3.3}$$

Какую же функцию лучше настраивать агенту — $Q^{\pi}(s, a)$ или $V^{\pi}(s)$?

Рассмотрим эту ситуацию в контексте игры в шахматы с точки зрения одного игрока. Этот игрок может быть представлен стратегией π. Конкретная расстановка

фигур на доске — это состояние s. Опытный шахматист интуитивно понимает, хорошая эта диспозиция или плохая. $V^{\pi}(s)$ и есть интуиция, выраженная количественно, например, как число от 0 до 1. Сначала значение $V^{\pi}(s)$ равно 0,5, поскольку обе стороны начинают игру в равных условиях. В ходе игры наш шахматист добивается успеха или терпит поражения, и значение $V^{\pi}(s)$ растет или уменьшается с каждым ходом. Если игрок очень успешен, то значение функции полезности $V^{\pi}(s)$ близко к 1. В данном конкретном случае $V^{\pi}(s)$ тождественна вероятности победы нашего шахматиста, вычисляемой для каждой диспозиции в игре. Тем не менее интервал значений $V^{\pi}(s)$ зависит от того, как определяется сигнал вознаграждения r, поэтому она не обязательно равна вероятности победы. Более того, в играх для одного игрока отсутствует само понятие победы над противником.

Помимо оценки диспозиции, шахматист рассматривает несколько альтернативных ходов и их возможные последствия. $Q^{\pi}(s, a)$ представляет количественную оценку каждого хода. На основе этого значения может быть принято решение о лучшем ходе (действии) при конкретной диспозиции (состоянии). Существует и другой способ оценки хода — с помощью лишь $V^{\pi}(s)$. Нужно рассмотреть следующие состояния s' для каждого действительного хода a, рассчитать $V^{\pi}(s')$ для каждого из этих состояний и выбрать действие, руководствуясь лучшим s'. Однако это требует времени и знания функции переходов, которая в шахматах доступна, но во многих других средах — нет.

Преимущество $Q^{\pi}(s, a)$ заключается в том, что она предоставляет агенту метод для непосредственного осуществления действий. Агенты могут вычислить $Q^{\pi}(s, a)$ для каждого действия $a \in A_s$, доступного в конкретном состоянии s, и выбрать действие по максимальному значению. В идеальном случае $Q^{\pi}(s, a)$ представляет оптимальную ожидаемую полезность выбора действия a в состоянии s, обозначаемую $Q^{*}(s, a)$. Это лучшее, что можно сделать, если действовать оптимально при всех последовательностях действий. Следовательно, знание $Q^{*}(s, a)$ дает оптимальную стратегию.

Недостаток настройки $Q^{\pi}(s, a)$ состоит в том, что аппроксимация функции сложнее с вычислительной точки зрения и требует большего количества данных по сравнению с $V^{\pi}(s)$. Чтобы научиться давать хорошую оценку в случае с $V^{\pi}(s)$, требуется, чтобы данные в достаточной мере охватывали пространство состояний. В то же время для $Q^{\pi}(s, a)$ нужны все пары (s, a), а не только все состояния s [76, 132]. Комбинированное пространство состояний и действий зачастую значительно больше, чем пространство состояний, поэтому для обучения хорошей оценке Q-функции необходимо больше данных. Рассмотрим конкретный пример: предположим, что есть состояние s с 100 возможными действиями и агент, попробовавший каждое действие по одному разу. Если мы настраиваем $V^{\pi}(s)$, то все эти 100 единиц данных могут быть использованы для настройки полезности s, тогда как при настройке $Q^{\pi}(s, a)$ для каждой пары (s, a) есть лишь по одной единице данных. Как правило,

чем больше количество данных, тем лучше аппроксимация функции. При том же количестве данных оценка V-функции будет, скорее всего, лучше, чем оценка Q-функции.

Хотя $V^{\pi}(s)$ может оказаться проще аппроксимировать, у нее есть один существенный недостаток. Чтобы использовать $V^{\pi}(s)$ для выбора действий, агент должен иметь возможность выбрать каждое из действий $a \in A_s$, доступных в состоянии s, и наблюдать следующее состояние s', в которое перейдет среда[1]. Кроме того, ему нужно знать последующее вознаграждение r. Тогда агент может действовать оптимально, выбирая действия, которые ведут к большим ожидаемым суммарным вознаграждениям $\mathbb{E}\left[r + V^{\pi}(s')\right]$. Однако если переходы в среде случайны, то выбор действия a в состоянии s может привести к разным следующим состояниям s'. Тогда агенту для получения хорошей оценки ожидаемой полезности выбора конкретного действия может понадобиться повторить этот процесс многократно. И это потенциально затратно с вычислительной точки зрения.

Рассмотрим следующий пример. Предположим, что выбор действия a в состоянии s в одной половине случаев приводит к состоянию s'_1 с вознаграждением $r'_1 = 1$, а в другой — к состоянию s'_2 с вознаграждением $r'_2 = 2$. Кроме того, считаем, что известны полезности состояний s'_1 и s'_2, а $V^{\pi}(s'_1) = 10$ и $V^{\pi}(s'_2) = -10$. Получаем:

$$r'_1 + V^{\pi}(s'_1) = 1 + 10 = 11; \tag{3.4}$$

$$r'_2 + V^{\pi}(s'_2) = 2 - 10 = -8. \tag{3.5}$$

Тогда из уравнения (3.6) ожидаемая полезность $\mathbb{E}\left[r + V^{\pi}(s')\right] = 1{,}5$:

$$\mathbb{E}\left[r + V^{\pi}(s')\right] = \frac{1}{2}\left(r'_1 + V^{\pi}(s'_1) + r'_2 + V^{\pi}(s'_2)\right) = \frac{1}{2}(11 - 8) = 1{,}5. \tag{3.6}$$

Однако если пример один, то оценка $\mathbb{E}\left[r + V^{\pi}(s')\right]$ будет равна 11 или –8, что далеко от ожидаемого значения 1,5.

Требование *знания следующего шага* для $V^{\pi}(s)$ часто является проблемой. Например, может оказаться, что агента можно перезапустить не во всех состояниях, которые нужны для изучения всех доступных действий, или перезапуск агента может быть затратным. $Q^{\pi}(s, a)$ позволяет избежать этой проблемы, так как напрямую настраивает полезности (s, a). Ее можно рассматривать как хранение следующего шага для каждого действия a в каждом состоянии s [132]. Поэтому алгоритмы RL с выбором действий с помощью настроенной функции полезности чаще аппроксимируют $Q^{\pi}(s, a)$.

---

[1] Здесь для простоты опущены индексы шагов и принята стандартная нотация (s, a, r, s', a', r'). На текущем шаге t используется обозначение кортежа (s, a, r), а на следующем шаге $t + 1$ — (s', a', r').

3.2. Метод временных различий

В этом разделе обсуждается настройка Q-функции с помощью метода временных различий (TD-обучения). Основная суть — использование нейронной сети, которая на входе оценивает Q-значения заданных пар (s, a). Это известно как *сеть полезности* (value network).

Настройка параметров сети полезности происходит следующим образом. Генерируются траектории τ и даются прогнозные $\hat{Q}$-значения для каждой пары (s, a). Затем с помощью траекторий генерируются целевые Q-значения Q_{tar}. Наконец, минимизируются расстояния между $\hat{Q}$ и Q_{tar} с помощью стандартной регрессионной функции потерь, такой как средняя квадратическая ошибка. Этот процесс повторяется многократно. Он похож на процесс обучения с учителем, в котором прогнозные значения ассоциируются с целевыми значениями. Однако в данном случае нужен способ порождения целевых значений для каждой траектории, тогда как в обучении с учителем они известны заранее.

В SARSA целевые значения Q_{tar} порождаются с помощью TD-обучения. В качестве пояснения полезно рассмотреть, как это происходит при выборке методом Монте-Карло, которая обсуждалась в разделе 2.4.

Пусть даны N траекторий τ_i, $i \in \{1... N\}$, начиная с состояния s, в котором агент выбрал действие a. Тогда оценка по методу Монте-Карло $Q^{\pi}_{\text{tar:MC}}(s, a)$ — это среднее значений отдачи всех траекторий (уравнение (3.7)). Следует отметить, что уравнение (3.7) применимо к любому набору траекторий с любой начальной парой (s, a):

$$Q^{\pi}_{\text{tar:MC}}(s, a) = \frac{1}{N}\sum_{i=1}^{N} R(\tau_i). \tag{3.7}$$

Если доступна полная траектория τ_i, то можно рассчитать действительное Q-значение, которое агент получает на этой траектории, в данном конкретном случае — для каждой пары (s, a) в τ_i. Это следует из определения Q-функции (уравнение (3.1)), так как ожидание будущих суммарных дисконтированных вознаграждений для одного-единственного примера — это просто суммарное дисконтированное вознаграждение, начиная с текущего временного шага и до конца эпизода для этого примера. В уравнении (3.7) показано, что это также оценка по методу Монте-Карло для $Q^{\pi}(s, a)$, где количество использованных траекторий $N = 1$.

Теперь каждая из пар (s, a) в наборе данных ассоциируется с целевым Q-значением. Можно сказать, что набор данных помечен ярлыком, поскольку у каждой единицы данных (s, a) есть соответствующее целевое значение $Q^{\pi}_{\text{tar}}(s, a)$.

Один из минусов выборки методом Монте-Карло — то, что агенту приходится ждать конца эпизода, прежде чем можно будет воспользоваться для обучения

какими-нибудь данными из него. Это очевидно из уравнения (3.7). Для расчета $Q^{\pi}_{\text{tar:MC}}(s, a)$ нужны вознаграждения для остальной части траектории, начиная с (s, a). Эпизоды могут состоять из большого количества временных шагов, ограниченного T, что задерживает обучение. Этим обусловлен другой подход к настройке Q-значений — TD-обучение.

Ключевым моментом в TD-обучении является то, что Q-значения для текущего временного шага могут быть определены через Q-значения для следующего временного шага. То есть $Q^{\pi}(s, a)$ определяются рекурсивно:

$$Q^{\pi}(s, a) = \mathbb{E}_{s' \sim p(s'|s, a),\, r \sim \mathcal{R}(s, a, s')}\left[r + \gamma \mathbb{E}_{a' \sim \pi(s')}\left[Q^{\pi}(s', a')\right]\right]. \tag{3.8}$$

Уравнение (3.8) известно как уравнение Беллмана. Если Q-функция корректна для стратегии π, то это уравнение дает точный результат. Оно также предлагает способ настройки Q-значений. Мы только что видели, как выборка методом Монте-Карло может быть задействована для настройки $Q^{\pi}(s, a)$, когда дано множество траекторий опыта. Этот же прием может быть применен и здесь, если воспользоваться TD-обучением для вывода целевых значений $Q^{\pi}_{\text{tar}}(s, a)$ для каждой пары (s, a).

Предположим, что есть нейронная сеть Q_{θ}, представляющая Q-функцию. В TD-обучении $Q^{\pi}_{\text{tar}}(s_t, a_t)$ получается посредством оценки правой части уравнения (3.8) с помощью Q_{θ}. На каждой итерации обучения значение $\hat{Q}^{\pi}_{\text{tar}}(s_t, a_t)$ обновляется, чтобы приблизиться к $Q^{\pi}_{\text{tar}}(s_t, a_t)$.

Если $Q^{\pi}_{\text{tar}}(s_t, a_t)$ получено с помощью той же нейронной сети, которая производит и $\hat{Q}^{\pi}_{\text{tar}}(s_t, a_t)$, то почему это работает? Почему это может дать хорошую аппроксимацию Q-функции после большого количества шагов? Более подробно это обсуждается в подразделе 3.2.1, но если кратко, то $Q^{\pi}_{\text{tar}}(s_t, a_t)$ использует информацию на шаг вперед по сравнению с $\hat{Q}^{\pi}_{\text{tar}}(s_t, a_t)$ и таким образом получает доступ к вознаграждению r из следующего состояния s'. Следовательно, $Q^{\pi}_{\text{tar}}(s_t, a_t)$ чуть более информативна в отношении того, какое направление примет траектория. Эта формулировка основана на предположении о том, что данная информация о целевой функции — максимизированном суммарном вознаграждении — выявляется по мере прохождения траектории и недоступна в начале эпизода. Это основной признак задач RL.

Однако использование уравнения (3.8) для построения $Q^{\pi}_{\text{tar}}(s_t, a_t)$ сопряжено с двумя проблемами из-за применения двух математических ожиданий. Первое из них — это внешнее математическое ожидание $\mathbb{E}_{s' \sim p(s'|s, a),\, r \sim \mathcal{R}(s, a, s')}[\ldots]$ по следующим состояниям и вознаграждениям. Проиллюстрируем это, предположив, что существует набор траекторий $\{\tau_1, \tau_2 \ldots \tau_M\}$, содержащих кортежи (s, a, r, s'). Для каждого такого кортежа доступен только один пример следующего состояния s' для данного действия.

Если среда детерминирована, то при подсчете математического ожидания для следующих состояний правильно рассматривать только действительное следующее состояние. Но если среда стохастическая, то это не работает. Выбор действия a в состоянии s может привести к переходу среды в некое количество разных следующих состояний, но в реальности на каждом шаге наблюдалось только одно из них. Эту проблему можно решить, если рассматривать только один пример — тот, который действительно имел место. Это означает, что у оценки Q-значения в случае стохастической среды может быть высокая дисперсия, но это же позволяет сделать данную оценку более гибкой. Использование только одного примера для оценки распределения по следующим состояниям s' и вознаграждениям r дает возможность отбросить внешнее математическое ожидание и переписать уравнение Беллмана в виде уравнения (3.9):

$$Q^{\pi}(s,a) = r + \gamma \mathbb{E}_{a' \sim \pi(s')}\left[Q^{\pi}(s',a')\right]. \tag{3.9}$$

Вторая проблема — внутреннее математическое ожидание $\mathbb{E}_{a' \sim \pi(s')}[\ldots]$ по действиям в уравнении (3.8). Поскольку для расчета Q-значений используется текущая оценка Q-функции, то эти значения доступны для всех действий в следующем состоянии s'. Проблема в том, что неизвестно распределение вероятностей по действиям, которое необходимо для подсчета математического ожидания. Это можно решить с помощью двух алгоритмов — SARSA и DQN (см. главу 4).

Решение методом SARSA заключается в использовании действия a', которое было действительно выбрано в следующем состоянии:

$$Q^{\pi}(s,a) \approx r + \gamma Q^{\pi}(s',a') = Q^{\pi}_{\text{On:SARSA}}(s,a). \tag{3.10}$$

После реализации большого количества примеров для заданных состояний доля выбранных действий должна приблизительно соответствовать распределению вероятностей по действиям.

Решение с помощью DQN — это применение максимального Q-значения:

$$Q^{\pi}(s,a) \approx r + \gamma \max_{a'_j} Q^{\pi}(s',a'_j) = Q^{\pi}_{\text{Off:DQN}}(s,a), \tag{3.11}$$

что соответствует неявной стратегии выбора действий с максимальным Q-значением с вероятностью, равной 1. Как будет показано в главе 4, в Q-обучении неявная стратегия является жадной относительно Q-значений, и такое преобразование данного уравнения справедливо[1].

---

[1] Третий альтернативный вариант — это вывод распределения вероятностей по действиям из стратегии и вычисление математического ожидания. Этот алгоритм известен как Expected SARSA. За дополнительной информацией обращайтесь к главе 6 второго издания книги Саттона и Барто «Обучение с подкреплением» [132].

Теперь можно подсчитать $Q^{\pi}_{\text{tar}}(s, a)$ для каждого кортежа (s, a, r, s', a'), воспользовавшись правой частью уравнения (3.10)[1]. Как и при выборке методом Монте-Карло, каждая пара (s, a) в наборе данных связана с Q-значением. Поэтому для настройки аппроксимации функции $Q^{\pi}(s, a)$ нейронной сетью могут быть применены те же методы, что и в обучении с учителем. Заметьте, что для вычисления целевых Q-значений нужна информация только для следующего шага, а не для всей траектории эпизода. Это дает возможность при TD-обучении обновлять Q-функцию после получения набора из нескольких примеров либо итеративно после каждого примера.

TD-обучение — это метод обучения с бутстрэппингом, так как при расчете $Q^{\pi}_{\text{tar}}(s, a)$ используются существующие оценки Q-значений для пары (s, a). Его преимуществом является снижение дисперсии оценки по сравнению с выборкой методом Монте-Карло. В TD-обучении задействуется лишь действительное вознаграждение r из следующего состояния s', которое объединяется с оценками Q-значений для аппроксимации остальной части полезности. Q-функция представляет собой математическое ожидание по разным траекториям и, как правило, имеет меньшую дисперсию, чем полная траектория, применяемая в выборке методом Монте-Карло. Однако она тоже вносит в данный подход к обучению систематическую погрешность, ведь аппроксимация Q-функции неидеальна.

3.2.1. Принцип метода временных различий

Рассмотрим на примере, как проходит TD-обучение. Предположим, что агент учится играть в среде, представленной на рис. 3.1. По сути, это коридор, и агент должен научиться доходить до его конца, где находится хорошее конечное состояние s_{T2}, отмеченное звездой. Всего здесь пять состояний: s_1, s_2, s_3 и два конечных — s_{T1} и s_{T2}. Возможны только два действия — a_{up} и a_{down}. Выбор a_{up} переводит агента на одну клетку вверх по коридору, а a_{down} — на одну клетку вниз. Агент всегда начинает игру из состояния s_1, обозначенного $\boldsymbol{S}$. Игра заканчивается, если агент попадает в одно из конечных состояний. s_{T2} — целевое состояние, при его достижении агент получает вознаграждение 1. Во всех остальных состояниях вознаграждение нулевое. Коэффициент дисконтирования агента $\gamma = 0{,}9$. Таким образом, оптимальное решение игры — стратегия, при которой s_{T2} достигается за наименьшее количество шагов. Причина в том, что вознаграждениям, полученным раньше по времени, агент дает более высокую оценку, чем полученным позже. В данном случае наименьшее

[1] В уравнении Беллмана для SARSA (3.10) необходимо знать действие a', которое действительно было выбрано агентом в следующем состоянии s'. Как следствие, в SARSA каждый кортеж опыта (s, a, r, s', a') содержит дополнительный элемент a'. В DQN, напротив, нужно знать только следующее состояние s', поэтому кортеж опыта имеет вид (s, a, r, s').

количество шагов, которое может предпринять агент, чтобы найти оптимальное решение задачи среды, составляет 3.

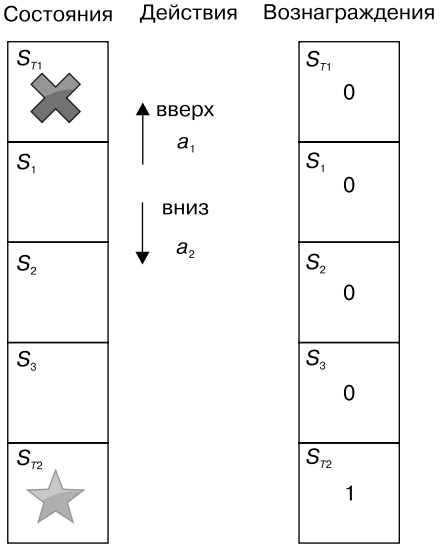


Рис. 3.1. Простая среда с пятью состояниями и двумя действиями в каждом из них

Для такой простой среды Q-функция может быть представлена таблицей, в которой на каждую пару (s, a) приходится по одной клетке. Это представление известно как табличная Q-функция. Пар всего шесть, так как агент не может действовать после достижения конечного состояния. Оптимальная Q-функция определяется как ожидаемая сумма дисконтированных вознаграждений за выбор действия a в состоянии s и последующее следование оптимальной стратегии. В данной среде оптимальная стратегия — это выбор действия a_{down} во всех состояниях, поскольку это кратчайший путь к состоянию s_{T2} из любого другого состояния. Оптимальная для данной среды Q-функция в табличной форме представлена на рис. 3.2.

$Q^*(s, a)$	вверх	вниз
S_1	0	0,81
S_2	0,73	0,9
S_3	0,81	1,0

Рис. 3.2. Оптимальные Q-значения для простой среды (см. рис. 3.1), $\gamma = 0{,}9$

Оптимальные Q-значения получены из определения Q-функции. Рассмотрим несколько прецедентов.

- (s_0, a_{up}) — агент выходит из коридора, получает нулевое вознаграждение, и эпизод заканчивается, то есть $Q^*(s_0, a_{\text{up}}) = 0$.

- (s_3, a_{down}) — агент достигает конечного состояния, получает вознаграждение, равное 1, и эпизод завершается, то есть $Q^*(s_3, a_{\text{down}}) = 1$.
- (s_2, a_{down}) — агент получает нулевое вознаграждение и переходит в s_3, то есть $Q^*(s_2, a_{\text{down}}) = r_2 + 0{,}9Q^*(s_3, a_{\text{down}}) = 0 + 0{,}9 \cdot 1 = 0{,}9$.
- (s_3, a_{up}) — агент получает вознаграждение, равное 0, и переходит в s_2, то есть $Q^*(s_3, a_{\text{up}}) = r_3 + 0{,}9Q^*(s_2, a_{\text{down}}) = 0 + 0{,}9 \cdot (0 + 0{,}9 \cdot 1) = 0{,}81$.

Как с помощью TD-обучения настроить оптимальную Q-функцию? Предположим, что каждая клетка таблицы Q-значений инициализирована 0. Процесс настройки Q-функции включает в себя получение случайной выборки траекторий и обновление таблицы Q-значений с помощью уравнения Беллмана каждый раз, когда агент получает кортеж (s, a, r, s'):

$$Q^*(s, a) = r + \gamma Q^*(s', a'); \tag{3.12}$$

$$Q^*(s, a) = r + 0{,}9Q^*(s', a_{\text{down}}). \tag{3.13}$$

На рис. 3.3 в табличной форме показан процесс настройки оптимальной Q-функции на наборе из пяти траекторий из среды. Диаграмма разбита на пять блоков, идущих сверху вниз. Каждый блок соответствует примерам из среды для отдельно взятого эпизода. Первый блок соответствует первому эпизоду, второй — второму и т. д. В каждом блоке столбцы пронумерованы, далее приведены пояснения к ним в порядке расположения слева направо.

- **Q-функция, начало эпизода** — значения Q-функции в начале эпизода. В начале первого эпизода все значения инициализированы нулями, так как пока еще нет информации о функции.
- **Эпизод** — это номер эпизода.
- **Временной шаг.** В каждом блоке свое количество прецедентов. Например, во втором эпизоде прецедентов три, а в четвертом — семь. Временной индекс каждого прецедента внутри одного блока соответствует номеру шага.
- **Действие** — действие, которое выбирает агент на каждом шаге.
- (s, a, r, s') — прецеденты для агента на каждом шаге. Каждый прецедент состоит из текущего состояния s, выбранного агентом действия a, полученного вознаграждения r и следующего состояния s', в которое перешла среда.
- $r + \gamma Q^*(s', a)$ — целевое значение (то есть правая часть уравнения), применяемое в обновлении функции Беллмана: $Q^*(s, a) = r + \gamma Q^*(s', a')$.
- **Q-функция, конец эпизода** — значения Q-функции в конце эпизода. Обновление функции Беллмана было применено ко всем прецедентам в порядке следования шагов. Это означает, что оно применялось сначала к прецеденту,

соответствующему первому шагу, затем — второму и т. д. В таблице приведен конечный результат, после того как к эпизоду были применены все обновления функции Беллмана.

Q-функция, начало эпизода			Эпизод	Временной шаг	Действие	$(s_1, a, 0, s')$	$r + \gamma Q^*(s', a)$	Q-функция, конец эпизода		
	вверх	вниз							вверх	вниз
S_1	0	0	1	1	↓	$(S_1, D, 0, S_2)$	$0 + 0{,}9 \times 0 = 0$	S_1	0	0
			1	2	↑	$(S_2, U, 0, S_1)$	$0 + 0{,}9 \times 0 = 0$			
S_2	0	0	1	3	↓	$(S_1, D, 0, S_2)$	$0 + 0{,}9 \times 0 = 0$	S_2	0	0
			1	4	↓	$(S_2, D, 0, S_3)$	$0 + 0{,}9 \times 0 = 0$			
S_3	0	0	1	5	↓	$(S_3, D, 1, S_{T2})$	1	S_3	0	1
	вверх	вниз							вверх	вниз
S_1	0	0	2	1	↓	$(S_1, D, 0, S_2)$	$0 + 0{,}9 \times 0 = 0$	S_1	0	0
S_2	0	0	2	2	↓	$(S_2, D, 0, S_3)$	$0 + 0{,}9 \times 1 = 0{,}9$	S_2	0	0,9
S_3	0	1	2	3	↓	$(S_3, D, 1, S_{T2})$	1	S_3	0	1
	вверх	вниз							вверх	вниз
S_1	0	0	3	1	↓	$(S_1, D, 0, S_2)$	$0 + 0{,}9 \times 0{,}9 = 0{,}81$	S_1	0	0,81
S_2	0	0,9	3	2	↓	$(S_2, U, 0, S_1)$	$0 + 0{,}9 \times 1 = 0{,}9$	S_2	0	0,9
S_3	0	1	3	3	↑	$(S_1, U, 0, S_2)$	$0 + 0{,}9 \times 0{,}9 = 0{,}81$	S_3	0,81	1
			3	4	↓	$(S_2, D, 0, S_3)$	$0 + 0{,}9 \times 1 = 0{,}9$			
			3	5	↓	$(S_3, D, 1, S_{T2})$	1			
	вверх	вниз							вверх	вниз
S_1	0	0,81	4	1	↓	$(S_1, D, 0, S_2)$	$0 + 0{,}9 \times 0{,}9 = 0{,}81$	S_1	0	0,81
S_2	0	0,9	4	2	↑	$(S_2, U, 0, S_1)$	$0 + 0{,}9 \times 0{,}81 = 0{,}73$	S_2	0,73	0,9
S_3	0,81	1	4	3	↓	$(S_1, D, 0, S_2)$	$0 + 0{,}9 \times 0{,}9 = 0{,}81$	S_3	0,81	1
			4	4	↑	$(S_2, U, 0, S_1)$	$0 + 0{,}9 \times 0{,}81 = 0{,}73$			
			4	5	↓	$(S_1, D, 0, S_2)$	$0 + 0{,}9 \times 0{,}9 = 0{,}81$			
			4	6	↓	$(S_2, D, 0, S_3)$	$0 + 0{,}9 \times 1 = 0{,}9$			
			4	7	↓	$(S_2, D, 1, S_{T2})$	1			
	вверх	вниз							вверх	вниз
S_1	0	0,81						S_1	0	0,81
S_2	0,73	0,9	5	1	↑	$(S_3, U, 0, S_{T2})$	0	S_2	0,73	0,9
S_3	0,81	1						S_3	0,81	1

Рис. 3.3. Настройка $Q^*(s, a)$ для простой среды (см. рис. 3.1)

После пяти траекторий, состоящих в общей сложности из 21 шага, табличная Q-функция принимает те же значения, что и оптимальная Q-функция (см. рис. 3.2). То есть она *сошлась* к оптимальной Q-функции.

В TD-обучении при обновлении используется действительное вознаграждение, полученное в следующем состоянии. Это дает эффект частичного возврата сигналов вознаграждения за будущие временные шаги на более ранние шаги по одному шагу за каждое обновление. В TD-обучении при каждом возврате данных Q-функция включает в себя информацию еще об одном будущем шаге. Чем больше возвращено данных, тем больше сведений о будущем включено в оценку Q-функции в момент времени t. Благодаря этому механизму Q-функция может содержать информацию о длительных периодах времени.

Количество временных шагов, которое требуется для того, чтобы Q-функция сошлась, зависит от среды и выбранных агентом действий, поскольку и то и другое влияет на прецеденты, используемые для ее настройки. Например, если агент не выберет действие перехода вверх до шестого эпизода, то Q-функция будет сходиться дольше (в масштабе эпизодов), потому что до наступления этого эпизода не будет информации о левой части Q-таблицы.

Значение коэффициента дисконтирования γ влияет на оптимальную Q-функцию. На рис. 3.4 показаны оптимальные Q-значения для экстремальных значений γ 0 и 1. Если $\gamma = 0$, агент становится близоруким и заботится лишь о вознаграждении, получаемом в текущем состоянии. В данном случае единственная пара (s, a) с ненулевым Q-значением — это (s_3, DOWN). Это не слишком-то полезно, так как в Q-значениях больше не содержится информации о вознаграждениях на будущих шагах. Как следствие, агент не имеет ни малейшего понятия о том, как действовать в s_1 и s_2. Если $\gamma = 1$, то для всех пар (s, a), за исключением (s_1, UP), которая приводит к завершению игры с нулевым вознаграждением, Q-значение равно 1, потому что агент не беспокоится о получении вознаграждения за скорейшее достижение s_{T2}.

Взглянем на γ с другой стороны — рассмотрим его влияние на скорость настройки Q-функции. Малые значения γ соответствуют агенту с коротким временным горизонтом, который заглядывает в будущее лишь на несколько временных шагов, поэтому и Q-значения нужно возвращать обратно лишь на несколько шагов. Возможно, при таких обстоятельствах Q-функция будет настроена быстрее. Однако она может не содержать важной информации о более поздних вознаграждениях, что плохо скажется на производительности агента. Если значение γ велико, то на настройку Q-функции может уйти намного больше времени, ведь тогда нужно возвращать обратно информацию о большом количестве шагов. Но полученная в результате Q-функция будет значительно информативнее для агента. Следовательно, оптимальное значение γ зависит от среды. На протяжении книги будут рассмотрены примеры конкретных значений γ. О роли γ более подробно рассказано в примечании 3.1.

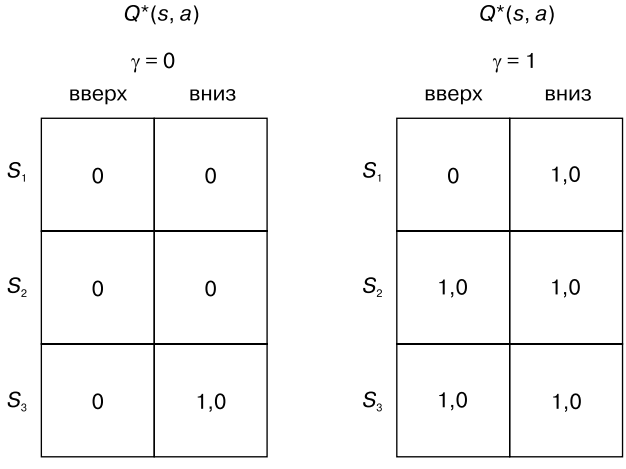


Рис. 3.4. Оптимальные Q-значения для простой среды (см. рис. 3.1): *слева* — $\gamma = 0$, *справа* — $\gamma = 1$

Примечание 3.1. Горизонт задачи

Коэффициент дисконтирования γ задает горизонт задачи для агента. Влияние этого параметра на все алгоритмы столь критично, что стоит продемонстрировать временные горизонты, соответствующие разным значениям γ.

Коэффициент γ определяет то, насколько весомыми станут для агента будущие вознаграждения в сравнении с полученными на текущем шаге. Отсюда следует, что изменение γ влечет за собой преобразование сути задачи RL в представлении агента. Если γ мало, то агент будет учитывать вознаграждения за текущий шаг и лишь за несколько шагов в будущем. Это эффективно ограничивает задачу RL до нескольких временных шагов и может значительно упростить проблему распределения вознаграждений. При малом γ обучение может происходить быстрее, ведь тогда требуется возвращать обратно Q-значения за горизонт агента на меньшее количество шагов или уменьшается длина траектории, необходимой для обновления градиента стратегии. Обеспечит ли это хорошую производительность, зависит от природы задачи, поскольку агент будет учиться максимизировать вознаграждения, полученные в ближайшем будущем[1].

В табл. 3.1 для разных значений γ приведены результаты применения дисконтирования к вознаграждениям, полученным через k шагов в будущем. Это дает представление о горизонте задачи для агента при различных значениях γ. Например, если

[1] Дисконтирование вознаграждений используется только агентом, его производительность обычно оценивается с помощью накопленных недисконтированных вознаграждений.

$\gamma = 0{,}8$, то эффективный горизонт задачи составляет примерно 10 шагов, тогда как при $\gamma = 0{,}99$ он составит 200–500 шагов.

Таблица 3.1. Применение дисконтирования к вознаграждениям после *k* временных шагов

γ	**Временной шаг**						
	10	**50**	**100**	**200**	**500**	**1000**	**2000**
0,8	0,110	0	0	0	0	0	0
0,9	0,350	0,010	0	0	0	0	0
0,95	0,600	0,080	0,010	0	0	0	0
0,99	0,900	0,610	0,370	0,130	0,070	0	0
0,995	0,950	0,780	0,610	0,370	0,080	0,010	0
0,999	0,990	0,950	0,900	0,820	0,610	0,370	0,140
0,9997	0,997	0,985	0,970	0,942	0,861	0,741	0,549

Хорошее значения для γ определяется тем, насколько обычно отложены вознаграждения в среде. Это могут быть десятки, сотни или тысячи шагов. К счастью, обычно γ не нуждается в столь длительной настройке, для широкого ряда проблем 0,99 — хорошее значение по умолчанию. Например, в этой книге значение 0,99 использовалось в обучающих примерах для игр Atari.

Полезным может быть и такой простой показатель, как длительность эпизода. В некоторых средах есть максимальное количество шагов. Например, в CartPole по умолчанию максимальная длительность — 200 шагов. При таких коротких эпизодах нет смысла в использовании большого значения γ, которое подразумевает учет вознаграждений на сотни шагов в будущем. Напротив, в случае обучения игре в Dota 2 с 80 000 шагов в OpenAI посчитали удобным постепенное увеличение коэффициента γ с 0,9980 до 0,9997 [104].

Многие среды в глубоком RL (и соответствующие им оптимальные Q-функции) намного сложнее, чем простая среда, обсуждавшаяся в этом разделе. В более сложных случаях Q-функцию невозможно представить в табличной форме. Вместо этого будут использоваться нейронные сети. Одно из следствий — обратное распространение информации будет медленнее. $\hat{Q}^{\pi}(s, a)$ не обновляются полностью до $r + \gamma Q^{\pi}(s', a)$ для каждого прецедента, так как нейронные сети обучаются постепенно путем градиентного спуска по стратегии. При каждом обновлении $\hat{Q}^{\pi}(s, a)$ будет лишь частично приближаться к $r + \gamma Q^{\pi}(s', a)$. Однако, несмотря на простоту рассмотренного примера, в нем была наглядно продемонстрирована фундаментальная концепция TD-обучения как механизма обратного распространения информации о функции вознаграждений среды с более поздних на более ранние временные шаги.

3.3. Выбор действий в SARSA

Теперь вернемся к открытому вопросу из предыдущего раздела: как метод для обучения хорошей аппроксимации Q-функции может быть преобразован в алгоритм для обучения хорошей стратегии? TD-обучение предоставляет метод для обучения оценке действий. Теперь нужна стратегия — механизм для выбора действий.

Предположим, что оптимальная Q-функция уже настроена. Тогда полезность каждой пары «состояние — действие» будет представлять наибольшую возможную ожидаемую полезность выбора данного действия. Это дает возможность составить оптимальный способ действий в обратном порядке. Если агент во всех состояниях всегда выбирает действия, которым соответствуют максимальные Q-значения, то есть агент действует жадно по отношению к Q-значениям, то он действует оптимально. Это предполагает также ограниченность применимости алгоритма SARSA дискретными пространствами действий, что поясняется в примечании 3.2.

Примечание 3.2. Применение SARSA ограничено дискретными пространствами действий

Для определения максимального Q-значения в состоянии s нужно вычислить Q-значения для всех возможных действий в этом состоянии. Когда действия дискретны, сделать это легко, потому что можно перечислить все возможные действия и рассчитать их Q-значения. Однако если действия непрерывны, то невозможно перечислить их все, и это проблема. По этой причине применимость SARSA и других основанных на полезности методов, таких как DQN (см. главы 4 и 5), как правило, ограничена дискретными пространствами действий. Тем не менее существуют методы типа *QT-Opt* [64], которые аппроксимируют максимальное Q-значение посредством выборки из непрерывного пространства действий, но они выходят за рамки данной книги.

К несчастью, оптимальная Q-функция обычно неизвестна. Однако связь между оптимальной Q-функцией и оптимальными действиями говорит о том, что если есть хорошая Q-функция, то можно найти хорошую стратегию. Это предполагает также итеративный подход к улучшению Q-значений.

Сначала инициализируем случайным образом нейронную сеть с параметрами θ, которая представляет Q-функцию и обозначена $Q^{\pi}(s, a; \theta)$. Затем повторяем приведенные далее шаги до тех пор, пока агент не перестанет улучшаться. Улучшение оценивается как сумма вознаграждений, полученных за эпизод.

1. Используем $Q^{\pi}(s, a; \theta)$ для действий в среде, действуя жадно по отношению к Q-значениям. Сохраняем все прецеденты (s, a, r, s').
2. Используем сохраненные прецеденты для обновления $Q^{\pi}(s, a; \theta)$ с помощью уравнения Беллмана в методе SARSA (уравнение (3.10)). Это улучшает оценку Q-функции, что, в свою очередь, совершенствует стратегию, так как оценки Q-значений теперь лучше.

3.3.1. Исследование и использование

При жадных действиях по отношению к Q-значениям проблема заключается в том, что стратегия детерминирована. Это означает, что пространство состояний и действий может быть исследовано агентом не полностью. Если агент всегда выбирает одно и то же действие a в состоянии s, то многие пары (s, a) могут быть никогда им не испытаны. В результате оценки Q-функции для некоторых пар (s, a) могут оказаться неточными, поскольку эти (s, a) были инициализированы случайными величинами и нет опыта их реального наблюдения. У сети не было возможности изучить эту часть пространства состояний и действий, поэтому агент может предпринимать условно оптимальные действия и застрять в локальном минимуме.

Для устранения этого затруднения для агента вместо чисто жадной стратегии обычно используется ε-жадная[1] стратегия. При такой стратегии агент выбирает жадное действие с вероятностью $1 - \varepsilon$ и действует случайным образом с вероятностью ε. Это вероятность исследования, так как выполнение случайных действий в $\varepsilon \cdot 100\,\%$ случаев помогает агенту исследовать пространство состояний и действий. Но за исследования приходится платить: подобная стратегия может быть менее эффективна, чем жадная. Причина в том, что агент вместо действий, максимизирующих Q-значения, предпринимает случайные действия с отличной от нуля вероятностью.

Противоречие между потенциальной ценностью исследования во время обучения и выбором наилучших действий на основе информации, доступной агенту (Q-значений), известно как проблема выбора между использованием стратегии и исследованием среды. Должен ли агент использовать свои текущие знания, чтобы действовать настолько хорошо, насколько это возможно, или исследовать, действуя случайно? Иногда исследование может снизить производительность, но оно также дает шанс обнаружить лучшие состояния и способы действия. Иначе это можно выразить так: если агенту доступна оптимальная Q-функция, то он должен действовать жадно, но во время настройки Q-функции жадные действия могут не дать ему стать лучше.

Чаще всего эту проблему решают следующим образом: начинают с высокого значения ε, равного или близкого к 1,0. Тогда агент может совершать случайные действия и быстро исследовать пространство состояний и действий. Поскольку в начале обучения агент еще ничего изучил, то использовать нечего. С течением времени ε постепенно уменьшается, так что после многих шагов можно надеяться, что стратегия приблизится к оптимальной. Раз агент учится лучшим Q-функциям, а значит, улучшается и стратегия, то больше нет смысла в исследовании и агенту нужно действовать более жадно.

В идеальном случае агент по прошествии некоторого времени найдет оптимальную Q-функцию, и тогда значение ε можно снизить до 0. На практике настроенная

[1] Произносится «эпсилон-жадная».

Q-функция не полностью сходится к оптимальной стратегии. Это может быть обусловлено временными ограничениями, непрерывностью пространств состояний или высокой размерностью дискретных пространств состояний, а также нелинейной аппроксимацией функций. Тем не менее ε обычно снижают до фиксированного малого значения, например 0,100–0,001, что позволяет производить небольшое количество непрерывных этапов исследования.

Была доказана сходимость TD-обучения к оптимальной Q-функции для линейной аппроксимации функций [131, 137]. Однако в последнее время значительный прогресс в обучении с подкреплением был вызван введением сложной нелинейной аппроксимации функций (нейронной сетью), позволяющей представлять гораздо более сложные Q-функции.

К сожалению, сходимость TD-обучения при переходе к нелинейной аппроксимации функции не гарантируется. Полное разъяснение выходит за рамки этой книги. Заинтересованному читателю следует посмотреть превосходную лекцию Сергея Левина по теории настройки функций полезности, которая является частью курса CS294 в Калифорнийском университете в Беркли[1]. К счастью, на практике было показано, что даже без этой гарантии могут быть достигнуты хорошие результаты [88, 135].

3.4. Алгоритм SARSA

Теперь приведем описание алгоритма SARSA и поясним, почему это метод обучения по актуальному опыту.

Псевдокод SARSA с ε-жадной стратегией приведен в алгоритме 3.1. Оценка Q-функции $\hat{Q}^{\pi}(s,a)$ параметризирована сетью с параметрами θ, обозначенной как Q^{π_θ}, так что $\hat{Q}^{\pi}(s,a) = Q^{\pi_\theta}(s,a)$. Обратите внимание на то, что оптимальные начальное и минимальное значения для ε, а также скорость его уменьшения зависят от среды.

Алгоритм 3.1. SARSA

```
Инициализация скорости обучения α
Инициализация ε
Инициализация параметров сети θ случайными значениями
for m = 1...MAX_STEPS do
    Накопить N прецедентов (s_i, a_i, r_i, s'_i, a'_i), используя
    ➥ текущую ε-жадную стратегию
    for i = 1...N do
        # Рассчитать целевые Q-значения для каждого прецедента
```

[1] Видео доступно на https://youtu.be/k1vNh4rNYec [76].

```
 8:          y_i = r_i + δ_{s'_i} γQ^{π_θ}(s'_i, a'_i), где δ_{s'_i} = 0, если s'_i – конечное
  :          ➥ состояние, иначе 1
 9:      end for
10:      # Расчет функции потерь, например, с помощью среднеквадратичной ошибки
11:      L(θ) = (1/N) Σ_i (y_i − Q^{π_θ}(s_i, a_i))^2
12:      # Обновление параметров сети
13:      θ = θ − α∇_θ J(θ)
14:      Уменьшение ε
15:  end for
```

3.4.1. Алгоритмы обучения по актуальному опыту

Важная особенность SARSA состоит в том, что это *алгоритм обучения по актуальному опыту*. Как вы помните из главы 2, в алгоритмах обучения по актуальному опыту используемая для улучшения текущей стратегии информация зависит от стратегии, которая применялась для сбора данных. Обучение по актуальному опыту может проходить двумя способами.

Во-первых, используемое для настройки Q-функции целевое значение может зависеть от стратегии, которая применялась для порождения прецедентов. Примером может служить SARSA, как показано в алгоритме 3.1 (строка 7). Целевое значение y_i зависит от действия a', которое действительно было выбрано в следующем состоянии s'. Это действие определяется стратегией, порождающей прецеденты, то есть текущей ε-жадной стратегией.

Во-вторых, алгоритм может быть алгоритмом обучения по актуальному опыту при непосредственном обучении стратегии. Это подразумевает такое изменение стратегии, что вероятность хороших действий повышается, а плохих — снижается. Чтобы выполнить данное улучшение, необходим доступ к вероятностям, которые текущая стратегия присваивает выбранным действиям. REINFORCE (см. главу 2) — пример алгоритма обучения по актуальному опыту с непосредственным изучением стратегии.

То, что SARSA является алгоритмом обучения по актуальному опыту, повлияло на тип прецедентов, которые могут быть использованы для настройки аппроксимации Q-функции в течение времени. На каждой итерации обучения могут применяться только прецеденты, накопленные с помощью текущей стратегии. После каждого обновления параметров аппроксимации функции все прецеденты должны быть отброшены и процесс накопления опыта должен начаться заново. Так происходит, даже если обновление параметров было незначительным. Это типично для обучения нейронных сетей, когда обновление одного параметра приводит к изменению стратегии. В алгоритме 3.1 это выражено явно, так как на каждой итерации обучения происходит сбор новых прецедентов.

Почему так важно, чтобы для обновления Q-функции на каждой итерации использовались именно прецеденты, накопленные с помощью текущей стратегии? Рассмотрим еще раз TD-обновление в SARSA (уравнение (3.14)).

$$Q^{\pi_1}(s, a) \approx r + \gamma Q^{\pi_1}(s', a'_1). \tag{3.14}$$

Второй член уравнения $\gamma Q^{\pi_1}(s', a'_1)$ предполагает, что a' было выбрано с помощью стратегии π_1. Это следует из определения Q-функции, которая является ожидаемой будущей полезностью (s, a), в предположении, что агент действует в соответствии со стратегией π_1. Здесь неуместны прецеденты, сгенерированные с помощью другой стратегии.

Предположим, существует прецедент (s, a, r, s', a'_2), порожденный стратегией π_2. Например, он может быть получен на более ранней итерации во время обучения с другими значениями параметров θ аппроксимации Q-функции. a'_2 не обязательно будет таким же, как a'_1. Действие, выбранное в s' стратегией π_2, может быть другим, не таким, какое предпримет текущая стратегия π_1. Если это так, то $Q^{\pi_1}(s, a)$ не будет отражать ожидаемое суммарное будущее дисконтированное вознаграждение за выбор действия a в состоянии s в соответствии с π_1:

$$Q^{\pi_1}_{\text{fair}}(s, a) = r + \gamma Q^{\pi_1}(s', a'_1) \neq r + \gamma Q^{\pi_1}(s', a'_2). \tag{3.15}$$

Уравнение (3.15) показывает: если вместо (s, a, r, s', a'_2) использовать (s, a, r, s', a'_1), то $Q^{\pi_1}(s, a)$ будет обновлена некорректно. Это вызвано тем, что она применяет оценки Q-значений, полученные с помощью последовательности действий, отличной от той, которая соответствует текущей стратегии.

3.5. Реализация SARSA

В этом разделе вы познакомитесь с реализацией алгоритма SARSA. Сначала рассматривается код `epsilon_greedy`, определяющий действия агента SARSA в среде. Затем идет обзор метода расчета целевого Q-значения. В конце раздела приводится цикл обучения сети и обсуждаются классы `Memory` для SARSA.

3.5.1. ε-жадная функция выбора действий

Функция выбора действий возвращает действие a, которое агент должен предпринять в состоянии s. Любая функция выбора действий принимает на входе состояние `state`, алгоритм `algorithm` для доступа к аппроксимации функции нейронной сетью, а также тело агента `body`, в котором хранится информация о переменной исследования (то есть ε) и пространстве действий. Функции выбора действий возвращают действие, определенное агентом, и распределение вероятностей, из которого оно было взято. В случае SARSA распределение вырожденное — вся вероятностная мера присвоена действию, максимизирующему Q-значение.

Код `epsilon_greedy` (листинг 3.1) состоит из трех частей.

- Агент решает, как действовать — случайно или жадно (строка 5). Это обусловлено текущим значением ε (строка 4).
- Если действия случайны, вызывается функция выбора действий `random` (строка 6). Она возвращает действие из пространства действий посредством выборки из случайного равномерного распределения вероятностей по действиям.
- Если действия жадные, то оцениваются Q-значения для всех действий в `state`. Выбирается действие, которое соответствует максимальному Q-значению. Это выполняется посредством вызова функции выбора действий по умолчанию `default` (строки 11–14).

Важная особенность реализации аппроксимации Q-функции — то, что она выдает одновременно все Q-значения для определенного состояния. Такая структура нейронной сети для аппроксимации Q-функции эффективнее, чем вариант, когда на входе принимаются состояние и действие, а на выходе выдается одно значение. Для нее требуется лишь один дополнительный проход через сеть вместо `action_dim` прямых проходов. Более подробную информацию о построении и инициализации разных типов сетей смотрите в главе 12.

Листинг 3.1. Реализация SARSA: ε-жадная функция выбора действий

```
 1 # slm_lab/agent/algorithm/policy_util.py
 2
 3 def epsilon_greedy(state, algorithm, body):
 4     epsilon = body.explore_var
 5     if epsilon > np.random.rand():
 6         return random(state, algorithm, body)
 7     else:
 8         # Возвращает действие с максимальным Q-значением
 9         return default(state, algorithm, body)
10
11 def default(state, algorithm, body):
12     pdparam = calc_pdparam(state, algorithm, body)
13     action = sample_action(body.ActionPD, pdparam)
14     return action
```

3.5.2. Расчет Q-функции потерь

Следующий шаг в алгоритме SARSA — это расчет Q-функции потерь, которая используется для обновления параметров аппроксимации Q-функции.

В листинге 3.2 сначала для каждого прецедента в наборе рассчитывается $\hat{Q}^{\pi}(s, a)$ действия a, выбранного агентом в текущем состоянии s. Это подразумевает прямой проход через сеть полезности для получения оценок Q-значений для всех пар (s, a)

(строка 11). Как вы помните, в варианте уравнения Беллмана для SARSA для вычисления $Q^{\pi}_{\text{tar}}(s,a)$ используется действие, которое на самом деле было выбрано агентом в следующем состоянии:

$$\hat{Q}^{\pi}(s,a) = r + \gamma \hat{Q}^{\pi}(s',a') = Q^{\pi}_{\text{tar}}(s,a). \tag{3.16}$$

Таким образом, происходит повторение одного и того же шага, но для следующих состояний, а не для текущих (строки 12 и 13). Заметьте, что целевые *Q*-значения фиксированы, в связи с чем на этом шаге градиент не рассчитывается. Поэтому, чтобы предотвратить расчет градиентов PyTorch, прямой проход вызывается в контексте `torch.no_grad()`.

Затем выделяются оценки *Q*-значений для действий, которые на самом деле были предприняты агентом в текущем и следующем состояниях (строки 15 и 16). Располагая этими данными, можно оценить $Q^{\pi}_{\text{tar}}(s,a)$ с помощью `act_next_q_preds` (строка 17). Конечные состояния, для которых $Q^{\pi}_{\text{tar}}(s,a)$ — это просто полученное вознаграждение, обрабатываются особым образом. Это достигается умножением $\hat{Q}^{\pi}(s',a')$ (`act_next_q_preds`) на вектор, элементы которого `1 - batch['dones']` имеют значение 0 для конечных состояний и 1 — в других случаях. Получив оценки текущего и целевого *Q*-значений (`q_preds` и `act_q_targets` соответственно), подсчитать функцию потерь просто (строка 18).

Необходимо знать об одном важном моменте: агент получает информацию лишь о действии, которое на самом деле было выбрано. Как следствие, он может получить информацию только об этом действии, а не обо всех доступных. Поэтому функция потерь рассчитывается с учетом лишь тех полезностей, которые соответствуют действиям, выбранным в текущем и следующем состояниях.

Листинг 3.2. Реализация SARSA, расчет целевых *Q*-значений и соответствующей функции потерь

```
 1 # slm_lab/agent/algorithms/sarsa.py
 2
 3 class SARSA(Algorithm):
 4     ...
 5
 6     def calc_q_loss(self, batch):
 7         '''Расчет функции потерь, связанных с Q-значениями,
           ➥ с помощью прогнозного и целевого Q-значений, полученных
           ➥ из соответствующих сетей'''
 8         states = batch['states']
 9         next_states = batch['next_states']
10         ...
11         q_preds = self.net(states)
12         with torch.no_grad():
13             next_q_preds = self.net(next_states)
```

```
14          ...
15          act_q_preds = q_preds.gather(-1,
            ➥ batch['actions'].long().unsqueeze(-1)).squeeze(-1)
16          act_next_q_preds = next_q_preds.gather(-1,
            ➥ batch['next_actions'].long().unsqueeze(-1)).squeeze(-1)
17          act_q_targets = batch['rewards'] + self.gamma *
            ➥ (1 - batch['dones']) * act_next_q_preds
18          q_loss = self.net.loss_fn(act_q_preds, act_q_targets)
19          return q_loss
```

3.5.3. Цикл обучения в SARSA

Цикл обучения (листинг 3.3) выглядит следующим образом.

1. На каждом шаге обучения вызывается `train` и агент проверяет, готов ли он к обучению (строка 10). Флаг `self.to_train` устанавливается в классе памяти, который обсуждается в следующем разделе.
2. Если пришло время обучения, то агент производит выборку данных из памяти путем вызова `self.sample()` (строка 11). В случае алгоритма SARSA пакет `batch` включает все прецеденты, накопленные с последнего этапа обучения агента.
3. Для `batch` рассчитываются Q-функция потерь (строка 13).
4. Q-функция потерь используется для одного обновления параметров сети полезности (строка 14). Когда функция потерь определена, PyTorch обновляет параметры с помощью автоматического дифференцирования, что очень удобно.
5. Для `self.to_train` устанавливается начальное значение 0, которое гарантирует, что агент не начнет обучаться, пока снова не будет готов (строка 16), то есть не накопит достаточного количества новых прецедентов.
6. Обновляется ε (`explore_var`) (строка 23) в соответствии с выбранным расписанием, например с линейной скоростью уменьшения значения.

Листинг 3.3. Реализация SARSA: цикл обучения

```
 1 # slm_lab/agent/algorithms/sarsa.py
 2
 3 class SARSA(Algorithm):
 4     ...
 5
 6     @lab_api
 7     def train(self):
 8         ...
 9         clock = self.body.env.clock
10        if self.to_train == 1:
11            batch = self.sample()
```

```
            ...
            loss = self.calc_q_loss(batch)
            self.net.train_step(loss, self.optim, self.lr_scheduler,
            ➥ clock=clock, global_net=self.global_net)
            # обнуление
            self.to_train = 0
            return loss.item()
        else:
            return np.nan

    @lab_api
    def update(self):
        self.body.explore_var = self.explore_var_scheduler.update(self,
        ➥ self.body.env.clock)
        return self.body.explore_var
```

Резюме

На высоком уровне реализация SARSA состоит из трех основных компонентов.

1. `epsilon_greedy` определяет метод для действий в среде согласно текущей стратегии.
2. `calc_q_loss` использует накопленные прецеденты для расчета целевых Q-значений и соответствующих значений функции потерь.
3. `train` производит одно обновление параметров сети полезности с помощью Q-функции потерь. Происходит обновление всех значимых переменных, таких как `explore_var` (ε).

3.5.4. Память для хранения пакетов прецедентов при обучении по актуальному опыту

Код из предыдущего раздела предполагает, что на момент начала обучения агенту доступен набор актуальных прецедентов.

Это видно по `self.sample()` в функции `train()` класса алгоритма SARSA. В этом разделе рассказывается о хранении прецедентов и предоставлении их агенту, когда они нужны ему для обучения.

В классе `Memory` (см. главу 2) механизм хранения и получения опыта — кортежей (s, a, r, s', a') — абстрактный. Типы прецедентов, которые могут быть использованы для аппроксимации функции обучения, подразделяются в зависимости от алгоритмов обучения по актуальному опыту. Это прецеденты, накопленные с помощью текущей стратегии. Как вы помните, все классы памяти должны реализовывать

следующие функции: `update`, которая добавляет прецеденты агента в память, `sample`, возвращающую набор данных для обучения, и `reset` для очистки памяти.

Из-за того что SARSA — это алгоритм TD-обучения, нужно лишь дождаться следующего временного шага, чтобы обновить параметры алгоритма с помощью кортежа (s, a, r, s', a'). Благодаря этому в алгоритме SARSA возможно обучение *по актуальному опыту*. Тем не менее использование только одного прецедента при обновлении может привести к сильной зашумленности. В связи с этим обычно практикуются ожидание в течение нескольких шагов и обновление параметров алгоритма с помощью среднего значения функции потерь, вычисленных по множеству прецедентов. Это известно как пакетное обучение (batch training). Альтернативная стратегия — накопить примеры опыта из не менее чем одного полного эпизода и с помощью всех этих данных за один раз обновить параметры алгоритма. Это называется обучением по эпизодам (episodic training).

Эти варианты отражают компромисс между скоростью и значением дисперсии. Данные из разных эпизодов, как правило, обладают меньшей дисперсией, но обучение алгоритма может происходить очень медленно, так как параметры обновляются нечасто. При обновлении с помощью одного прецедента дисперсия наивысшая, но обучение может идти быстро. Выбор количества прецедентов, которые будут использоваться для каждого обновления, обусловлен средой и алгоритмом. Некоторые среды могут выдавать данные с чрезвычайно высокой дисперсией, и для ее существенного снижения до уровня, при котором агент обучается, может понадобиться много прецедентов. В других средах дисперсия данных низкая, и количество прецедентов на обновление может быть довольно малым, что повышает предполагаемую скорость обучения.

Спектр функциональности: обучение по актуальному опыту, пакетное обучение и эпизодическое обучение — предоставляется двумя классами `Memory`. Класс `OnPolicyReplay` реализует эпизодическое обучение, а его потомок `OnPolicyBatchReplay` — пакетное обучение. Последний класс также поддерживает обучение по актуальному опыту при установке единичного размера пакета. Обратите внимание на то, что в SARSA для обучения по актуальному опыту нужны как минимум два шага в среде, чтобы стали известны следующее состояние и действие, что соответствует размеру пакета 2.

Поскольку с `OnPolicyReplay` мы познакомились в главе 2, здесь приводится только `OnPolicyBatchReplay`.

Обновление пакетной памяти. Для преобразования эпизодической памяти в пакетную можно воспользоваться наследованием классов и перегрузкой функции `add_experience`, как показано в листинге 3.4.

Для пакетной памяти не нужна вложенная структура, как у эпизодической памяти, так что в листинге 3.4 текущий прецедент добавляется в контейнеры основной

памяти напрямую (строки 8 и 9). Если накоплено достаточно прецедентов, класс памяти устанавливает для агента флаг обучения (строки 14 и 15).

Листинг 3.4. Добавление опыта в OnPolicyBatchReplay

```
 1 # slm_lab/agent/memory/onpolicy.py
 2
 3 class OnPolicyBatchReplay(OnPolicyReplay):
 4     ...
 5
 6     def add_experience(self, state, action, reward, next_state, done):
 7         self.most_recent = [state, action, reward, next_state, done]
 8         for idx, k in enumerate(self.data_keys):
 9             getattr(self, k).append(self.most_recent[idx])
10            # Отслеживание размера памяти и количества прецедентов
11            self.size += 1
12            self.seen_size += 1
13            # Принятие решения о том, должен ли агент обучаться
14            if len(self.states) ==
              ➥ self.body.agent.algorithm.training_frequency:
15                self.body.agent.algorithm.to_train = 1
```

3.6. Обучение агента SARSA

Настройка агента SARSA производится с помощью файла `spec`, содержимое которого приведено в листинге 3.5. Этот файл имеется в SLM Lab в `slm_lab/spec/benchmark/sarsa/sarsa_cartpole.json`.

Листинг 3.5. Простой файл spec для игры в CartPole с помощью SARSA

```
 1 # slm_lab/spec/benchmark/sarsa/sarsa_cartpole.json
 2
 3 {
 4     "sarsa_epsilon_greedy_cartpole": {
 5         "agent": [{
 6             "name": "SARSA",
 7             "algorithm": {
 8                 "name": "SARSA",
 9                 "action_pdtype": "Argmax",
10                 "action_policy": "epsilon_greedy",
11                 "explore_var_spec": {
12                     "name": "linear_decay",
13                     "start_val": 1.0,
14                     "end_val": 0.05,
15                     "start_step": 0,
16                     "end_step": 10000
17                 },
18                 "gamma": 0.99,
19                 "training_frequency": 32
```

```
            },
            "memory": {
                "name": "OnPolicyBatchReplay"
            },
            "net": {
                "type": "MLPNet",
                "hid_layers": [64],
                "hid_layers_activation": "selu",
                "clip_grad_val": 0.5,
                "loss_spec": {
                    "name": "MSELoss"
                },
                "optim_spec": {
                    "name": "RMSprop",
                    "lr": 0.01
                },
                "lr_scheduler_spec": null
            }
        }],
        "env": [{
            "name": "CartPole-v0",
            "max_t": null,
            "max_frame": 100000
        }],
        "body": {
            "product": "outer",
            "num": 1
        },
        "meta": {
            "distributed": false,
            "eval_frequency": 2000,
            "max_trial": 1,
            "max_session": 4
        },
        ...
    }
}
```

Рассмотрим основные компоненты.

- **Алгоритм** — это SARSA (строка 6). Стратегия выбора действий ε-жадная (строка 10) с линейной скоростью уменьшения (строки 11–17) переменной исследования ε. Значение γ устанавливается в строке 18.
- **Архитектура сети** — многослойный перцептрон с одним скрытым слоем из 64 элементов и функцией активации SeLU (строки 25–27).
- **Оптимизатор** — это RMSprop [50] со скоростью обучения 0,01 (строки 32–35).
- **Частота обучения** — по пакетам, так как была выбрана память `OnPolicyBatchReplay` (строка 22). Размер пакета 32 (строка 19), контролируемый параметром

`training_frequency` (строка 19), означает, что сеть будет тренироваться по прошествии каждых 32 шагов.

- **Среда** — это CartPole [18] из OpenAI Gym (строка 40).
- **Длительность обучения** — 100 000 шагов (строка 42).
- **Оценка агента** производится каждые 2000 шагов (строка 50). Во время оценки ε присваивается его итоговое значение (строка 14). Запускаются четыре эпизода, затем подсчитываются средние полные вознаграждения и выдается отчет о результатах.

Чтобы обучить агента SARSA с помощью SLM Lab, запустите в терминале команды из листинга 3.6.

Листинг 3.6. Обучение агента SARSA

```
1 conda activate lab
2 python run_lab.py slm_lab/spec/benchmark/sarsa/sarsa_cartpole.json
  ➥ sarsa_epsilon_greedy_cartpole train
```

Файл `spec` используется для запуска пробного обучения `Trial` с четырьмя сессиями `Session`, чтобы получить усредненные результаты, которые затем выводятся на графике с доверительным интервалом. Также генерируется график со скользящим средним по 100 оценкам. Оба графика приведены на рис. 3.5.

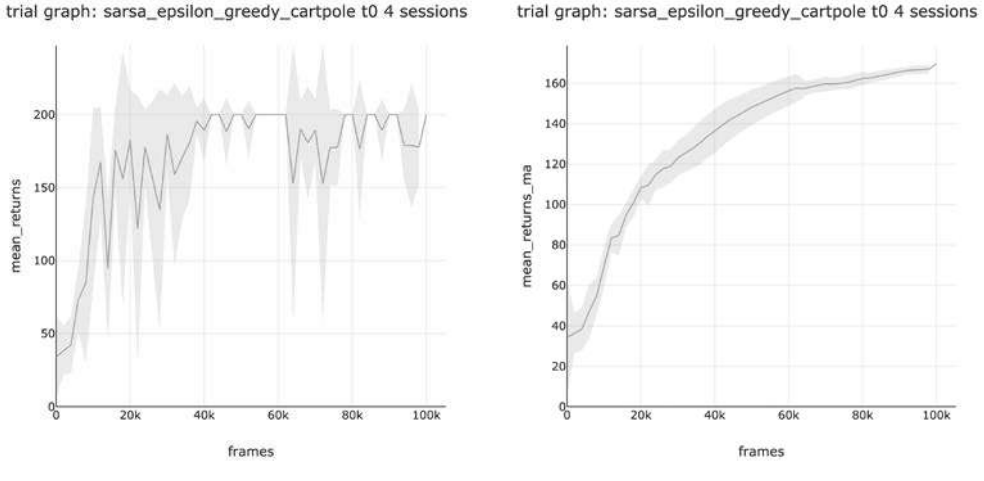


Рис. 3.5. Графики пробного обучения с помощью SARSA из SLM Lab на основе данных, усредненных по четырем сессиям. По вертикальной оси отложены полные вознаграждения (оценка mean_return вычисляется без дисконтирования), усредненные по восьми эпизодам в контрольных точках. По горизонтальной оси — полные кадры обучения. График справа — это скользящее среднее по оценкам в 100 контрольных точках

3.7. Результаты экспериментов

В этом разделе рассматривается влияние скорости обучения на производительность SARSA в среде CartPole. Функция экспериментирования SLM Lab используется для поиска по сетке по параметру «скорость обучения». Затем для них выводятся графики и производится сравнение.

3.7.1. Эксперимент по определению влияния скорости обучения

Скорость обучения определяет, часто ли обновляются параметры сети. Более высокая скорость может сделать обучение быстрым, но также вызвать переполнение пространства параметров из-за интенсивных обновлений. И наоборот, вероятность переполнения ниже, если скорость обучения низкая, но сходимость будет достигаться очень долго. Для поиска оптимальной скорости обучения часто применяется настройка гиперпараметров.

В данном эксперименте будет проверено влияние разных скоростей обучения на SARSA с помощью поиска по сетке по большому количеству скоростей обучения. Чтобы добавить спецификацию для поиска, воспользуйтесь файлом `spec` из листинга 3.5, как это показано в листинге 3.7. В строке 10 задается поиск по сетке по списку значений скоростей обучения `lr`. Полный файл `spec` находится в SLM Lab по адресу `slm_lab/spec/benchmark/sarsa/sarsa_cartpole.json`.

Листинг 3.7. Файл spec для SARSA со спецификацией для поиска по разным значениям скорости обучения

```
 1 # slm_lab/spec/benchmark/sarsa/sarsa_cartpole.json
 2
 3 {
 4     "sarsa_epsilon_greedy_cartpole": {
 5         ...
 6         "search": {
 7             "agent": [{
 8                 "net": {
 9                     "optim_spec": {
10                         "lr__grid_search": [0.0005, 0.001, 0.001, 0.005,
                           ➥ 0.01, 0.05, 0.1]
11                     }
12                 }
13             }]
14         }
15     }
16 }
```

Для запуска эксперимента в SLM Lab используйте команды, приведенные в листинге 3.8.

Листинг 3.8. Запуск эксперимента с поиском по разным скоростям обучения в соответствии с файлом spec

```
1 conda activate lab
2 python run_lab.py slm_lab/spec/benchmark/sarsa/sarsa_cartpole.json
  ➥ sarsa_epsilon_greedy_cartpole search
```

Будет запущен эксперимент `Experiment`, который породит множество пробных испытаний `Trial` с подстановкой разных значений `lr` в изначальный файл `spec` для SARSA. В каждом пробном испытании запускается по четыре сессии `Session` для получения среднего значения. График для множества пробных испытаний и его версия со скользящим средним по оценкам в 100 контрольных точках приведены на рис. 3.6.

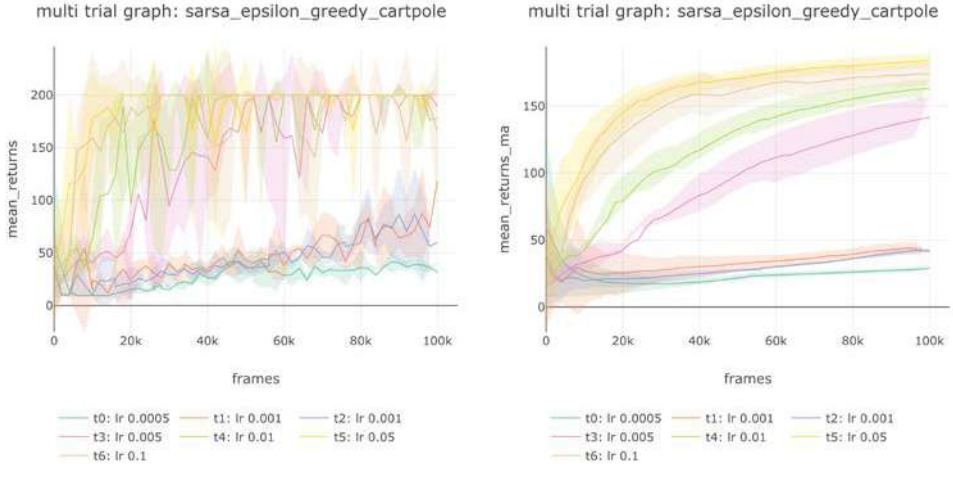


Рис. 3.6. Влияние высоких скоростей обучения на производительность SARSA. Как и ожидалось, агент обучается быстрее при высоких скоростях обучения

На рис. 3.6 показано явное влияние увеличения скорости обучения на кривую обучения SARSA, когда скорость не слишком велика. В пятом и шестом пробных испытаниях SARSA быстро получает максимальное вознаграждение 200 в CartPole. Напротив, когда скорость обучения низкая, агент обучается слишком медленно, как продемонстрировали нулевое, первое и второе пробные испытания.

3.8. Резюме

Два основных элемента алгоритма SARSA — это настройка Q-функции с помощью TD-обучения и метод для выбора действий с использованием оценок Q-значений.

В начале главы обсуждалось, почему для обучения в SARSA в качестве функции полезности следует выбирать Q-функцию. Она аппроксимируется с помощью

TD-обучения, которое, исходя из уравнения Беллмана, стремится минимизировать разницу между двумя выражениями *Q*-функции. В основе TD-обучения лежит мысль о том, что в задачах RL вознаграждения определяются с течением времени. Это дает возможность при оценке *Q*-функции возвращать информацию с будущих шагов на более ранние.

После настройки *Q*-функции становится очевидным, что хорошая стратегия может быть получена путем ε-жадных действий относительно *Q*-значений. Это говорит о том, что агент действует случайно с вероятностью ε, в других случаях он выбирает действие, соответствующее максимальной оценке *Q*-значения. Для поиска лучших решений агентам нужно найти баланс между использованием своих знаний и *исследованием среды*. ε-жадная стратегия — простой способ решения этой проблемы.

При реализации алгоритма SARSA наиболее важными компонентами являются цикл обучения и функция выбора действий, по которой рассчитываются *Q*-значения и связанная с ними функция потерь. Они реализованы в методах `epsilon_greedy`, `calc_q_loss` и `train`. Накопленные агентом SARSA прецеденты хранятся в классе памяти агента `Memory`. Эти прецеденты предоставляются агенту классом `Memory` во время каждого этапа обучения. SARSA — алгоритм обучения по актуальному опыту, так как применяемое для обучения сети полезности $Q_{\text{tar}}^{\pi}(s, a)$ целевое значение обусловлено стратегией сбора прецедентов. Использование прецедентов, которые не были накоплены с помощью текущей стратегии, может привести к неверным *Q*-значениям. В связи с этим `Memory` очищается после каждого этапа обучения агента.

Выборку методом Монте-Карло и TD-обучение можно трактовать как два противоположных полюса спектра решений проблемы поиска компромисса между смещением и дисперсией. Для введения в алгоритм SARSA этой проблемы его можно расширить, задав параметр количества шагов с действительными вознаграждениями, используемыми агентом, до оценки остальной полезности с помощью сети полезности. Это противоположно одному шагу с действительными вознаграждениями, применяемому в стандартном алгоритме SARSA. Во «Введении в обучение с подкреплением» [132] можно почерпнуть более подробные сведения по этому вопросу, а также увидеть связанный с ним более эффективный с точки зрения вычислений подход с использованием контрольных следов (eligibility traces).

3.9. Рекомендуемая литература

- *Sutton R. S., Barto A. G.* Reinforcement Learning: An Introduction. Second ed. 2018. Главы 4 и 5 в [132].
- *Rummery G. A., Niranjan M.* On-Line Q-Learning Using Connectionist Systems. 1994 [118].
- *Tesauro G.* Temporal Difference Learning and TD-Gammon. 1995. P. 58–68 [135].

3.10. Историческая справка

Метод временных различий (TD-обучение) впервые был применен Артуром Сэмюелем в алгоритме для игры в шахматы [119, 120] более 60 лет назад. Тридцать лет спустя Ричардом Саттоном были представлены первые официальные результаты [131], подтверждающие сходимость подмножества линейных алгоритмов TD-обучения. В 1997 году Цициклис и Ван Рой [137] доказали сходимость общей формы алгоритма TD-обучения с линейной аппроксимацией функций. Кроме того, на TD-обучении основан самый ранний алгоритм глубокого обучения — TD-Gammon [135]. Джеральд Тесауро, объединив в своем алгоритме многослойный перцептрон с TD-обучением и обучением агента на играх самого с собой, добился в 1991 году игры на уровне мастера. Испытание проводилось Биллом Роберти, мастером игры в нарды и бывшим мировым чемпионом [135]. Более подробный рассказ можно найти в статье Джеральда Тесауро *Temporal Difference Learning and TD-Gammon*, впервые опубликованной в журнале Communications of the ACM в марте 1995 года.

Уравнение Беллмана получило свое название в честь Ричарда Беллмана, прославленного математика, который изобрел динамическое программирование [34]. Уравнение Беллмана содержит в себе принцип оптимальности [16], который гласит: «Оптимальная стратегия обладает тем свойством, что, каковы бы ни были первоначальные состояние и решение, остальные решения должны образовывать оптимальную стратегию относительно состояния, полученного в результате первого решения». В обобщенном виде его можно представить следующим образом. Предположим, что известно, как действовать наилучшим образом во всех будущих состояниях, но не в текущем состоянии. Тогда поиск оптимального действия в текущем состоянии сводится к сравнению результатов всех действий, возможных в нем, включая сведения о полезностях действий во всех последующих состояниях, и выбору наилучшего из них. То есть полезность текущего состояния может быть определена рекурсивно с помощью полезностей будущих состояний. Краткую занимательную историю зарождения динамического программирования, записанную со слов самого Беллмана, можно прочесть в статье Стюарта Дрейфуса *Richard Bellman on the Birth of Dynamic Programming* [34].

4 Глубокие Q-сети

В этой главе рассматривается алгоритм глубоких *Q*-сетей (Deep Q-Networks, DQN), предложенный Мнихом и его коллегами в 2013 году. Как и SARSA, алгоритм DQN — это основанный на полезностях метод временных различий, который аппроксимирует *Q*-функцию. Настроенная функция используется агентом для выбора действий. DQN применим лишь к средам с дискретными пространствами действий. Однако DQN, в отличие от SARSA, настраивает не *Q*-функцию для текущей стратегии, а *оптимальную Q*-функцию. Это небольшое, но важное изменение позволяет повысить устойчивость и скорость обучения.

В разделе 4.1 с помощью уравнения Беллмана для DQN поясняется, почему этот метод настраивает оптимальную *Q*-функцию. Важным следствием является то, что благодаря этому DQN — алгоритм с обучения по отложенному опыту. Данная оптимальная *Q*-функция не зависит от стратегии сбора данных, что означает: DQN может обучаться на прецедентах, накопленных любым агентом.

Теоретически для порождения обучающих данных для DQN можно задействовать любую стратегию. Но практика показывает, что одни стратегии подходят для этого больше, чем другие. В разделе 4.2 рассматриваются характеристики хорошей стратегии сбора данных и вводится стратегия Больцмана в качестве альтернативы ε-жадной стратегии (см. раздел 3.4).

В разделе 4.3 мы обдумаем возможность применения того, что DQN является алгоритмом обучения по отложенному опыту, для повышения эффективности выборок по сравнению с SARSA. Во время обучения все прецеденты, накопленные более ранними стратегиями, хранятся в памяти (буфере) прецедентов [82] и используются повторно.

В разделе 4.4 представлен алгоритм DQN, а в разделе 4.5 рассматривается пример его реализации. В конце главы сравнивается производительность агента DQN в среде CartPole из OpenAI Gym при различных вариантах архитектуры сети.

В этой главе мы увидим, что для преобразования SARSA в DQN потребуется лишь незначительное изменение в обновлении *Q*-функции, но в результате будет получен алгоритм, который снимает многие ограничения SARSA. DQN дает возможность

устранить корреляцию между прецедентами и использовать их повторно путем случайной выборки пакетов из обширной памяти прецедентов. Кроме того, он позволяет выполнять множество обновлений параметров с помощью одного и того же пакета данных. Эти идеи, позволяющие значительно повысить эффективность DQN по сравнению с SARSA, — тема данной главы.

4.1. Настройка Q-функции в DQN

Как и SARSA, DQN настраивает *Q*-функцию с помощью TD-обучения. Разница между этими алгоритмами заключается в том, как строится $Q^{\pi}_{\text{tar}}(s,a)$.

Уравнения (4.1) и (4.2) — это уравнения Беллмана для DQN и SARSA соответственно:

$$Q^{\pi}_{\text{DQN}}(s,a) \approx r + \gamma \max_{a'} Q^{\pi}(s',a'); \tag{4.1}$$

$$Q^{\pi}_{\text{SARSA}}(s,a) \approx r + \gamma Q^{\pi}(s',a'). \tag{4.2}$$

В уравнениях (4.3) и (4.4) показано, как строится $Q^{\pi}_{\text{tar}}(s,a)$ согласно каждому из алгоритмов:

$$Q^{\pi}_{\text{tar:DQN}}(s,a) = r + \gamma \max_{a'} Q^{\pi}(s',a'); \tag{4.3}$$

$$Q^{\pi}_{\text{tar:SARSA}}(s,a) = r + \gamma Q^{\pi}(s',a'). \tag{4.4}$$

DQN для оценки $Q^{\pi}_{\text{tar}}(s,a)$ вместо действия a', на самом деле выбранного в следующем состоянии s', использует максимальное из *Q*-значений для всех возможных действий, доступных в этом состоянии.

В DQN *Q*-значение, используемое в следующем состоянии s', не зависит от стратегии, применявшейся для накопления прецедентов. Поскольку вознаграждение r и следующее состояние s' порождаются средой для данных состояния s и действия a, то никакая из частей оценки $Q^{\pi}_{\text{tar:DQN}}(s,a)$ не зависит от стратегии сбора данных. Это делает DQN алгоритмом обучения *по отложенному опыту*, поскольку настроенная функция независима от того, в соответствии с какой стратегией производятся действия в среде и накапливаются прецеденты [132]. Напротив, SARSA — это метод обучения по актуальному опыту, так как он использует действие a', выбранное по текущей стратегии в состоянии s', для расчета *Q*-значения для следующего состояния. Он напрямую зависит от стратегии, применявшейся для накопления прецедентов.

Предположим, что есть две стратегии π^1 и π^2, а $\hat{Q}^{\pi^1}(s,a)$ и $\hat{Q}^{\pi^2}(s,a)$ — две *Q*-функции, соответствующие этим стратегиям. Пусть заданы последовательности данных (s,a,r,s',a'_1) и (s,a,r,s',a'_2), порожденные π^1 и π^2 соответственно. В подразделе 3.4.1

показано, почему при текущей стратегии π^1 и более ранней стратегии с ранних этапов обучения π^2 применение (s, a, r, s', a'_1) для обновления параметров $Q^{\pi^1}(s, a)$ некорректно.

Однако, если применяется DQN, то при расчете $Q^{\pi}_{\text{tar:DQN}}(s, a)$ не используется ни a'_1, ни a'_2. Поскольку в обоих случаях следующее состояние s' одно и то же, для этих примеров опыта значения $Q^{\pi}_{\text{tar:DQN}}(s, a) = r + \gamma \max_{a'} Q^{\pi}(s', a')$ одинаковы, а значит, оба этих примера действительны. Даже если для накопления каждой из последовательностей данных применялись разные стратегии, $Q^{\pi}_{\text{tar:DQN}}(s, a)$ будет тем же самым.

Если функции переходов и вознаграждений стохастические, то при каждом выборе действия a в состоянии s могут быть получены различные r и s'. Это вызвано тем, что Q^{π} определяется как ожидаемая будущая *отдача* от выбора действия a в состоянии s, так что по-прежнему справедливо использование (s, a, r, s', a'_1) для обновления $\hat{Q}^{\pi^1}(s, a)$. В противном случае это можно истолковать как то, что среда несет ответственность за вознаграждение r и переход в следующее состояние s'. Это внешние условия по отношению к агенту и его стратегии, так что $Q^{\pi}_{\text{tar:DQN}}(s, a)$ все так же не зависит от стратегии, применявшейся для накопления прецедентов.

Если $Q^{\pi}_{\text{tar:DQN}}(s, a)$ не зависит от стратегии, использованной для накопления прецедентов, то какая стратегия ей соответствует? Это оптимальная стратегия по определению Саттона и Барто [132]: «Стратегия π' определена как лучшая по сравнению со стратегией π или равноценная ей, если во всех состояниях ожидаемое вознаграждение для нее больше, чем ожидаемое вознаграждение для π, или равно ему. Другими словами, $\pi' \geq \pi$ тогда и только тогда, когда $V^{\pi'}(s) \geq V^{\pi}(s)$ для всех $s \in S$. Всегда существует хотя бы одна стратегия, которая лучше, чем все другие стратегии, или равноценна им. Это оптимальная стратегия π^*».

Оптимальная Q-функция определяется как выбор действия a в состоянии s и дальнейшее следование оптимальной стратегии π^*:

$$Q^*(s, a) = \max_{\pi} Q^{\pi}(s, a) = Q^{\pi^*}(s, a). \tag{4.5}$$

Рассмотрим $Q^{\pi}_{\text{tar:DQN}}(s, a)$ в свете уравнения (4.5). Если оценки для Q^{π} верны, то оптимальным будет выбор действия, максимизирующего $Q^{\pi}(s', a')$. Это лучшее, что может сделать агент. Это означает, что стратегия, которой соответствует $Q^{\pi}_{\text{tar:DQN}}(s, a)$, — оптимальная стратегия π^*.

Следует отметить: хотя цель DQN — настройка оптимальной Q-функции, это еще не значит, что все будет получаться. Причин тому может быть много. Например, в гипотетическом пространстве, представленном нейронной сетью, в действительности может не содержаться оптимальной Q-функции. Методы невыпуклой оптимизации несовершенны и могут не находить глобального минимума. Кроме того, временные и вычислительные ограничения устанавливают предел длительности обучения агента. Тем не менее можно сказать, что верхняя граница производитель-

ности для DQN оптимальна по сравнению с потенциально неоптимальной верхней границей для SARSA, полученной при изучении *Q*-функции в соответствии с ε-жадной стратегией.

4.2. Выбор действий в DQN

Несмотря на то что DQN — алгоритм обучения по отложенному опыту, попрежнему имеет значение, каким образом агент накапливает прецеденты. В связи с этим нужно рассмотреть два важных фактора.

Во-первых, перед агентом все еще стоит проблема выбора между использованием стратегии и исследованием среды, обсуждавшаяся в главе 3. В начале обучения агенту нужно быстро исследовать пространство состояний и действий, чтобы повысить свои шансы на обнаружение хорошей последовательности действий в среде. В ходе обучения агент приобретает все больше сведений, и доля времени на исследование должна постепенно уменьшаться, чтобы можно было тратить больше усилий на использование изученного. Это повышает эффективность обучения, поскольку агент фокусируется на лучших действиях.

Во-вторых, если пространство состояний и действий очень велико из-за того, что содержит непрерывные значения или является дискретным, но высокоразмерным[1], то даже однократное испытание всех пар (*s*, *a*) может быть весьма затруднительным. В таком случае *Q*-значения для непосещенных пар (*s*, *a*) могут быть не лучше, чем случайные.

К счастью, аппроксимация функций с помощью нейронных сетей устраняет эту проблему, так как они способны делать обобщения на основе посещенных пар (*s*, *a*) и применять их к схожим[2] состояниям и действиям, как показано в примечании 4.1. Однако это не решает данную проблему полностью. В пространстве состояний и действий могут остаться пары, которые очень удалены и сильно отличаются от состояний и действий, испытанных агентом. Вряд ли в этом случае нейронная сеть сможет сделать хорошее обобщение, и оценочные *Q*-значения могут оказаться неточными.

---

[1] Например, пространство действий может описываться посредством изображений. Значения для пикселов, как правило, дискретны, например, от 0 до 255 для черно-белых изображений, но даже одна картинка может содержать тысячи и миллионы таких пикселов.

[2] Сходство между состояниями или действиями может быть установлено многими способами, это отдельная большая тема. Простой мерой сходства может служить расстояние L2. Чем меньше расстояние, тем больше сходство между двумя состояниями или действиями. Однако для высокоразмерных состояний, таких как изображения, расстояние L2 в пространстве пикселов может не охватывать важные случаи сходства. Например, схожие состояния в игре визуально могут быть разными. В других подходах эту проблему пытаются решить, например настраивая сначала низкоразмерные представления состояний, а затем измеряя расстояние L2 между представлениями состояний.

Примечание 4.1. Обобщения в нейронных сетях

Важным свойством методов аппроксимации функций является то, насколько хорошо они делают обобщение для не виденных ранее входных данных. Например, насколько хороши оценки Q-функции для непосещенных пар (s, a)? Как линейные (то есть линейные регрессии), так и нелинейные (то есть нейронные сети) методы аппроксимации функций способны делать некоторые обобщения, тогда как табличные методы — нет.

Рассмотрим табличное представление для $\hat{Q}^{\pi}(s, a)$. Предположим, что пространство состояний и действий очень велико и содержит миллионы пар (s, a) и в начале обучения все ячейки, представляющие отдельные значения $\hat{Q}^{\pi}(s, a)$, инициализированы нулями. Во время обучения агент посещает пары (s, a), и таблица обновляется, но для непосещенных пар (s, a) по-прежнему $\hat{Q}^{\pi}(s, a) = 0$. Поскольку пространство состояний и действий велико, многие пары (s, a) не будут посещены и их оценки Q-значений останутся равными нулю, даже если (s, a) является желаемым с $\hat{Q}^{\pi}(s, a) \gg 0$. Основная проблема в том, что табличное представление функции не определяет связи между разными состояниями и действиями.

Нейронные сети, напротив, могут экстраполировать Q-значения для известных (s, a) на неизвестные (s', a'), так как они определяют взаимосвязь между разными состояниями и действиями. Это полезно, когда пространство (s, a) очень велико или содержит бесконечно большое количество элементов, так как означает, что агенту не нужно посещать все (s, a), чтобы научиться хорошо оценивать Q-функцию. Агенту нужно посетить лишь (s, a) из репрезентативной для этого пространства состояний и действий выборки.

Способность сети делать обобщения ограничена, и существуют два общих случая, когда это зачастую невозможно. Во-первых, когда на вход сети подаются данные, значительно отличающиеся от тех, на которых она обучалась, вряд ли на выходе будут получены хорошие результаты. Как правило, обобщение значительно лучше, когда пространство входных данных является малой окрестностью тренировочных данных. Во-вторых, нейронные сети имеют свойство делать плохие обобщения, если аппроксимируемая функция имеет точки разрыва. Это связано с тем, что нейронные сети исходят из предположения, что пространство входных данных локально гладкое. Если входные значения x и x' схожи, то и соответствующие им выходные значения y и y' должны быть схожи.

Рассмотрим пример, приведенный на рис. 4.1 и представляющий одномерную среду, в которой агент может ходить по плоской вершине горы. Состояние — это положение агента относительно оси X, действие — изменение координаты x за шаг δx, а вознаграждения отложены вдоль оси Y. В этой среде есть обрыв, которому соответствует перепад значений вознаграждений при достижении агентом состояния $s = x = 1$. Падение с обрыва путем перемещения из $s = x = 0{,}99$ в $s' = x' = 1{,}0$ приведет к резкому уменьшению полученного вознаграждения, хотя состояния s и s' очень близки друг к другу. Перемещение на то же самое расстояние в других частях пространства состояний не вызовет такого значительного изменения в вознаграждении.

Рис. 4.1. Среда с обрывом с прерывистой функцией вознаграждений

Если пространство состояний и действий среды велико, то маловероятно, что агент сможет научиться давать хорошие оценки *Q*-значений для всех его частей. Но в таких средах все еще можно добиться хорошей производительности, если обучение агента будет сфокусировано на состояниях и действиях, которые при хорошей стратегии посещались бы чаще. Если объединить эту стратегию с нейронной сетью, велика вероятность того, что аппроксимация *Q*-функции будет хороша в локальных областях, окружающих часто посещаемые участки пространства состояний и действий.

Следовательно, при применяемой агентом DQN стратегии должны посещаться состояния и выбираться действия, которые в разумной степени близки к тем, которые посещались бы при жадных действиях по отношению к текущей оценке *Q*-функции агента. Это является текущей оценкой оптимальной стратегии. Говорится, что две стратегии должны порождать схожие распределения данных.

На практике этого можно достичь, прибегнув к ε-жадной стратегии, обсуждавшейся в главе 3, или стратегии Больцмана, которая представлена в следующем разделе. Применение обеих стратегий помогает агенту сосредоточиться на обучении на данных, которые он, вероятно, получил бы, воспользовавшись своей оценкой оптимальной стратегии[1].

[1] Трудности при настройке Q-функции для большого пространства состояний и действий возникают и в SARSA. Однако, поскольку SARSA — метод обучения по актуальному опыту, процесс настройки автоматически фокусируется на области пространства состояний и действий, которая в соответствии со стратегией посещается часто.

4.2.1. Стратегия Больцмана

Выбор вариантов реализации агентов DQN или SARSA не ограничивается лишь жадной и ε-жадной стратегиями. В этом разделе рассматривается третий вариант — стратегия Больцмана, получившая название по распределению вероятностей Больцмана.

Хорошие стратегии должны находить баланс между исследованием пространства состояний и действий и использованием знаний, полученных агентом.

ε-жадные стратегии достигают этого баланса путем уменьшения вероятности ε случайного выбора действий в ходе обучения. При такой стратегии сначала больше времени тратится на исследование среды, но со временем возрастает доля использования полученных знаний. Одна из проблем, связанных с этим, — то, что стратегия исследования наивная. Агенты выполняют исследования случайным образом и не используют полученные ранее знания о среде.

Стратегия Больцмана пытается улучшить случайные исследования путем выбора действий с помощью их относительных *Q*-значений. Максимизирующее *Q*-значение действие *a* в состоянии *s* будет выбираться чаще, но велика будет и вероятность выбора других действий с относительно высокими *Q*-значениями. И наоборот, действия с очень низкими *Q*-значениями вряд ли будут выбраны. Это приведет к тому, что исследование сосредоточится на последовательности с многообещающими действиями, максимизирующими *Q*-значения, вместо выбора всех действий с равной вероятностью.

Чтобы получить стратегию Больцмана, построим распределение вероятностей по *Q*-значениям для всех действий *a* в состоянии *s* с помощью логистической функции (softmax) (уравнение (4.6)). Логистическая функция параметризирована температурой $\tau \in (0, \infty)$, которая определяет, насколько равномерным или концентрированным будет результирующее распределение вероятностей. При высоких значениях τ распределение вероятностей становится более равномерным, при низких — более концентрированным. Затем в соответствии с этим распределением вероятностей выбираются действия, как показано в уравнении (4.7):

$$p_{\text{softmax}}(a \mid s) = \frac{e^{Q^{\pi}(s,a)}}{\sum_{a'} e^{Q^{\pi}(s,a')}}; \tag{4.6}$$

$$p_{\text{B}}(a \mid s) = \frac{e^{Q^{\pi}(s,a)/\tau}}{\sum_{a'} e^{Q^{\pi}(s,a')/\tau}}. \tag{4.7}$$

Роль параметра температуры τ в стратегии Больцмана аналогична роли ε в ε-жадной стратегии. Он стимулирует исследования пространства состояний и действий.

Чтобы понять, почему так происходит, рассмотрим пример, приведенный в уравнении (4.8):

- softmax:

$$x : [1; 2] \rightarrow p(x) : [0{,}27; 0{,}73;];$$

- распределение Больцмана:

$$\begin{aligned} \tau = 5&: x : [1; 2] \rightarrow p(x) : [0{,}45; 0{,}55]; \\ \tau = 2&: x : [1; 2] \rightarrow p(x) : [0{,}38; 0{,}62]; \\ \tau = 0{,}5&: x : [1; 2] \rightarrow p(x) : [0{,}12; 0{,}88]; \\ \tau = 0{,}1&: x : [1; 2] \rightarrow p(x) : [0{,}00; 1{,}00]. \end{aligned} \tag{4.8}$$

Высокие значения τ (то есть $\tau = 5$) смещают распределение вероятностей в сторону равномерного распределения. Это приводит к тому, что действия агента становятся в основном случайными. Низкие значения τ (то есть $\tau = 0{,}1$) повышают вероятность выбора действий с большими Q-значениями, поэтому агент будет действовать более жадно. При $\tau = 1$ распределение вероятностей сводится к распределению, полученному с помощью логистической функции. Настройка значения τ во время обучения позволяет сбалансировать исследование среды и использование знаний. Высокие значения τ в начале обучения будут способствовать исследованию среды, но с течением времени по мере снижения τ стратегия будет приближаться к жадной.

Основное преимущество стратегии Больцмана по сравнению с ε-жадной стратегией — менее случайное исследование среды. Каждый раз, выбирая действие, агент производит выборку a из порожденного стратегией Больцмана распределения вероятностей по действиям. Вместо того чтобы действовать случайным образом с вероятностью ε, агент выбирает действие a с вероятностью $p_{\text{B}}(a \mid s)$. В связи с этим вероятность выбора действий с высокими Q-значениями будет выше. Даже если агент не выбирает действие, максимизирующее Q-значение, он, вероятнее всего, выберет второе, а не третье или четвертое из действий, лучших с точки зрения $\hat{Q}^{\pi}(s, a)$.

Кроме того, стратегия Больцмана приводит к более гладкой функции, описывающей связь между оценками Q-значений и вероятностями действий, чем ε-жадная стратегия. Рассмотрим в качестве примера табл. 4.1, где есть состояние s с двумя доступными в нем действиями a_1 и a_2. В таблице сравниваются вероятности выбора a_1 и a_2 в состоянии s при ε-жадной стратегии и стратегии Больцмана при изменении $Q^{\pi}(s, a_1)$ и $Q^{\pi}(s, a_2)$. В этом примере $\tau = 1$ и $\varepsilon = 0{,}1$.

В двух крайних левых столбцах приведены Q-значения для a_1 и a_2. Когда они сильно различаются, например $Q^{\pi}(s, a_1) = 1$, $Q^{\pi}(s, a_2) = 9$, обе стратегии присваивают a_1 малую вероятность. Но когда эти значения близки, например $Q^{\pi}(s, a_1) = 5{,}05$,

$Q^{\pi}(s, a_2) = 4{,}95$, при ε-жадной стратегии бóльшая часть вероятности приходится на a_2, ведь ему соответствует максимальное Q-значение. Стратегия Больцмана, наоборот, присваивает обоим действиям почти одинаковые вероятности, $p(a_1) = 0{,}53$ и $p(a_2) = 0{,}48$.

Таблица 4.1. Вероятности действий

$Q^{\pi}(s, a_1)$	$Q^{\pi}(s, a_2)$	$p_{\varepsilon}(a_1 \mid s)$	$p_{\varepsilon}(a_2 \mid s)$	$p_{в}(a_1 \mid s)$	$p_{в}(a_2 \mid s)$
1,00	9,00	0,05	0,95	0,00	1,00
4,00	6,00	0,05	0,95	0,12	0,88
4,90	5,10	0,05	0,95	0,45	0,55
5,05	4,95	0,95	0,05	0,53	0,48
7,00	3,00	0,95	0,05	0,98	0,02
8,00	2,00	0,95	0,05	1,00	0,00

Интуитивно понятно, что это лучше, чем ε-жадная стратегия. Выбор в состоянии s действия a_1 или a_2 практически равноценен: их Q-значения очень близки, значит, они должны выбираться с приблизительно равной вероятностью. Это определяет стратегия Больцмана, тогда как ε-жадные стратегии ведут к крайним случаям поведения. Если одно из Q-значений немного больше другого, то ε-жадная стратегия присвоит этому действию всю вероятность неслучайных действий $(1 - \varepsilon)$. Если на следующей итерации другое Q-значение станет чуть больше, ε-жадная стратегия немедленно присвоит всю вероятность неслучайных действий уже этому действию. ε-жадная стратегия может быть менее стабильной, чем стратегия Больцмана, и агенту может оказаться труднее ее настраивать.

При стратегии Больцмана агент может застрять в локальном минимуме, если для некоторых частей пространства состояний оценки Q-функции неточные. Рассмотрим еще раз действия a_1 и a_2 в состоянии s. Пусть $Q^{*}(s, a_1) = 2$ и $Q^{*}(s, a_2) = 10$, а текущие оценки $\hat{Q}^{\pi}(s, a_1) = 2{,}5$ и $\hat{Q}^{\pi}(s, a_2) = -3$. Тогда a_2 — оптимальное действие, но при стратегии Больцмана вероятность выбора a_2 чрезвычайно мала (менее 0,5 % при $\tau = 1$). Из-за этого шансы агента на то, что он попробует a_2 и обнаружит, что оно лучше, очень малы. Скорее всего, он будет продолжать выбирать a_1 в s. При ε-жадной стратегии, напротив, a_2 будет выбираться с вероятностью $p = \varepsilon/2$ вне зависимости от оценок Q-значений. Поэтому велика вероятность того, что с течением времени оценка Q-значения будет скорректирована.

Один из путей решения этой проблемы с помощью стратегии Больцмана — использование большого значения τ в начале обучения, чтобы сделать распределение вероятностей действий более однородным. По мере обучения агент приобретет больше знаний и τ можно будет уменьшить, что позволит агенту применять то, чему он научился. Однако следует позаботиться о том, чтобы τ не уменьшалось слишком быстро, так как стратегия может застрять в локальном минимуме.

4.3. Хранение прецедентов в памяти

В этом разделе рассказывается, как DQN повышает эффективность выборок в SARSA с помощью *хранения прецедентов* в памяти [82]. Сначала рассмотрим две причины неэффективности выборок в алгоритмах обучения по актуальному опыту.

Во-первых, в алгоритмах обучения по актуальному опыту для обновления параметров стратегии могут применяться лишь данные, собранные с помощью текущей стратегии. Каждый прецедент используется однократно. Это может быть проблемой, если одновременно задействуются методы аппроксимации функций, которые настраиваются с помощью градиентного спуска, такие как нейронные сети. Все обновления параметров должны быть небольшими, потому что в градиентах передается лишь значимая информация о направлении спуска в малой окрестности текущих значений параметров. В то же время для некоторых прецедентов оптимальным было бы значительное обновление параметров, например, если прогнозируемые сетью Q-значения $\hat{Q}^{\pi}(s, a_1)$ и $Q^{\pi}(s, a_1)$ сильно различаются. В таких случаях может потребоваться многократное обновление параметров с помощью прецедентов, чтобы воспользоваться всей содержащейся в них информацией. Алгоритмы обучения по актуальному опыту на это не способны.

Во-вторых, между применяемыми для настройки алгоритмов обучения по актуальному опыту прецедентами существует сильная корреляция. Это вызвано тем, что данные для одного обновления параметров часто взяты из одного эпизода, поскольку будущие состояния и вознаграждения зависят от предыдущих состояний и действий[1]. Это может привести к высокой дисперсии при обновлении параметров.

Алгоритмам обучения по отложенному опыту, таким как DQN, не нужно отбрасывать прецеденты после их однократного использования. Лонг-Джи Лин [82] впервые сделал это наблюдение в 1992 году и предложил дополнение к Q-обучению под названием *«память прецедентов»* (experience replay). Он заметил, что TD-обучение может быть медленным из-за применяемого для сбора данных механизма последовательных приближений, свойственного RL, и необходимости обратного распространения информации во времени. Повышение скорости TD-обучения равносильно либо ускорению процесса присвоения коэффициентов доверия, либо сокращению этапа проб и ошибок [82]. В памяти прецедентов сделан упор на второе за счет повторного использования прецедентов.

В памяти хранятся k последних прецедентов, накопленных агентом. Если память заполнена, то, чтобы освободить место для новых прецедентов, самые старые примеры отбрасываются. Выборка одного или нескольких пакетов данных из памяти прецедентов производится случайно и равномерно на каждом этапе обучения

[1] Можно накопить данные из большого количества эпизодов. Однако из-за этого обучение будет отложено, а между прецедентами внутри каждого эпизода все равно сохранится сильная корреляция.

агента. Каждый из этих пакетов используется для обновления параметров сети, представляющей Q-функцию. Значение k, как правило, довольно большое — от 10 000 до 1 000 000, тогда как количество элементов в пакете намного меньше — обычно между 32 и 2048.

Размер памяти должен быть достаточным для хранения прецедентов из множества эпизодов. В каждом пакете обычно содержатся прецеденты, взятые из разных эпизодов и полученные с помощью разных стратегий. Это позволяет устранить корреляцию между прецедентами, используемыми для обучения агента, что, в свою очередь, снижает дисперсию при обновлении параметров, способствуя повышению стабильности обучения. В то же время память должна быть достаточно мала для того, чтобы каждый из прецедентов был выбран более одного раза, прежде чем будет отброшен, что делает обучение более эффективным.

Также важно отбрасывать самые старые из прецедентов. По мере обучения агента изменяется распределение пар (s, a), которые наблюдал агент. Более старые примеры становятся бесполезными, поскольку маловероятно, что агент их посетит. Если время и вычислительные ресурсы ограничены, предпочтительно, чтобы агент обучался на прецедентах, собранных недавно, поскольку они будут более значимыми. Этот принцип реализован в форме хранения в памяти лишь k наиболее свежих прецедентов.

4.4. Алгоритм DQN

Познакомившись со всеми компонентами DQN, перейдем к описанию самого алгоритма. Псевдокод для DQN со стратегией Больцмана приведен в алгоритме 4.1. Оценка Q-функции $\hat{Q}^{\pi}(s, a)$ параметризирована сетью с параметрами θ, обозначенной Q^{π_θ}, таким образом, имеем $\hat{Q}^{\pi}(s, a) = Q^{\pi_\theta}(s, a)$.

Алгоритм 4.1. DQN

```
Инициализация скорости обучения α
Инициализация τ
Инициализация количества прецедентов на шаг обучения, B
Инициализация количества обновлений на пакет, U
Инициализация размера пакета данных, N
Инициализация памяти прецедентов с максимальным размером K
Инициализация параметров сети θ случайными значениями
for m = 1...MAX_STEPS do
    Накопить и сохранить h прецедентов (s_i, a_i, r_i, s'_i),
    ➥ используя текущую стратегию
    for b = 1...B do
        Выбрать b-й пакет прецедентов из памяти
        for u = 1...U do
            for i = 1...N do
                # Рассчитать целевые Q-значения для каждого примера
```

```
15:                 y_i = r_i + δ_{s_i} γ max_{a'_i} Q^{π_k}(s'_i, a'_i), где δ_{s'_i} = 0, если s'_i –
                    ➥ конечное состояние, иначе δ_{s'_i} = 1
16:             end for
17:             # Рассчитать функцию потерь, например,
                ➥ с помощью средней квадратической ошибки
18:             L(θ) = (1/N) Σ_i (y_i – Q^{π_θ}(s_i, a_i))^2
19:             # Обновление параметров сети
20:             θ = θ – α∇_θ L(θ)
21:         end for
22:     end for
23:     Уменьшение τ
24: end for
```

Мы начинаем со сбора и сохранения данных (строка 9), полученных в соответствии со стратегией Больцмана, порождающей Q-значения с помощью текущей Q^{π_k}. Нужно отметить, что можно было воспользоваться и ε-жадной стратегией. Для обучения агента производится выборка B пакетов прецедентов из памяти (строки 10 и 11). Для каждого набора данных происходит U обновлений параметров. Сначала рассчитываются целевые Q-значения для всех элементов в пакете (строка 15). Заметьте, что эти расчеты включают выбор максимизирующего Q-значение действия в следующем состоянии s'. Это одна из причин того, что DQN применим лишь к средам с дискретными пространствами действий. В этом случае вычисления производятся очень просто — нужно лишь рассчитать Q-значения для всех действий и выбрать максимальное. Однако если пространство действий непрерывное, то количество возможных действий бесконечное и рассчитать Q-значения для них всех невозможно. Другая причина та же, что и в случае SARSA: для ε-жадной стратегии требуются действия, максимизирующие Q-значения.

Затем рассчитываются функции потерь (строка 18). И наконец, вычисляется градиент функции потерь и обновляются параметры сети θ (строка 20). После выполнения всего этапа обучения (строки 10–22) обновляется τ (строка 23).

В алгоритме 4.1 продемонстрированы два практических следствия из того, что DQN является алгоритмом обучения по отложенному опыту. Первое: на каждой итерации обучения для обновления оценки Q-функции может быть использовано более одного пакета прецедентов. Второе: агент не ограничен одним обновлением параметров на пакет данных. По желанию можно настраивать аппроксимацию Q-функции до сходимости на каждом пакете данных. Эти шаги увеличивают вычислительную нагрузку на каждую итерацию обучения по сравнению с SARSA, но они могут и значительно ускорить обучение. Тут возникает проблема выбора между дальнейшим обучением на прецедентах, которые были накоплены агентом на текущий момент, и использованием улучшенной Q-функции для сбора лучших прецедентов для обучения. Ее отражают количество обновлений параметров на пакет U и количество пакетов на шаг обучения B. Хорошие значения для этих параметров обусловлены задачей и доступными вычислительными ресурсами. Обычно применяется от одного до пяти пакетов и обновлений параметров на пакет.

4.5. Реализация DQN

DQN можно рассматривать как преобразование алгоритма SARSA. В этом разделе приводится реализация DQN как расширения класса `SARSA` в SLM Lab.

`VanillaDQN` расширяет `SARSA` и повторно использует большую часть его методов. Поскольку оценка *Q*-функции в DQN происходит иначе, нужно переопределить `calc_q_loss`. Следует также внести изменения в `train`, чтобы отразить дополнительный выбор количества пакетов и обновлений параметров на пакет. И конечно же, нужно реализовать хранение прецедентов — новый класс памяти `Replay`.

4.5.1. Расчет Q-функции потерь

Метод `calc_q_loss` для DQN приведен в листинге 4.1, он очень похож на аналогичный метод в SARSA.

Сначала для всех действий в каждом состоянии s в пакете рассчитываются $\hat{Q}^{\pi}(s,a)$ (строка 9). Этот же шаг, но без учета градиента, повторяется для следующего состояния s' (строки 10 и 11).

Затем для каждого прецедента в пакете выбирается оценка *Q*-значения для действия a, предпринятого агентом в текущем состоянии s (строка 12). В строке 13 для каждого следующего состояния выбирается максимальная оценка *Q*-значения, которая используется для вычисления $Q^{\pi}_{\text{tar|DQN}}(s,a)$ (строка 14). В конце рассчитывается функция потерь с помощью $\hat{Q}^{\pi}(s,a)$ и $Q^{\pi}_{\text{tar|DQN}}(s,a)$ (`act_q_preds` и `max_q_targets` соответственно).

Листинг 4.1. Реализация DQN, расчет целевых Q-значений и соответствующей функции потерь

```
 1 # slm_lab/agent/algorithms/dqn.py
 2
 3 class VanillaDQN(SARSA):
 4     ...
 5
 6     def calc_q_loss(self, batch):
 7         states = batch['states']
 8         next_states = batch['next_states']
 9         q_preds = self.net(states)
10         with torch.no_grad():
11             next_q_preds = self.net(next_states)
12         act_q_preds = q_preds.gather(-1,
           ➥ batch['actions'].long().unsqueeze(-1)).squeeze(-1)
13         max_next_q_preds, _ = next_q_preds.max(dim=-1, keepdim=False)
14         max_q_targets = batch['rewards'] + self.gamma *
           ➥ (1 - batch['dones']) * max_next_q_preds
15         q_loss = self.net.loss_fn(act_q_preds, max_q_targets)
16         ...
17         return q_loss
```

4.5.2. Цикл обучения DQN

Цикл обучения в листинге 4.2 протекает следующим образом.

1. На каждом шаге выполняется вызов `train`, и агент проверяет свою готовность обучаться (строка 10). Флаг `self.to_train` устанавливается классом памяти.
2. Если пришло время обучения, то агент делает выборку `self.training_iter` пакетов данных из памяти (строка 12), вызывая `self.sample()` (строка 13).
3. Каждый из пакетов используется для выполнения `self.training_batch_iter` обновлений параметров (строки 15–18). Все обновления параметров происходят в соответствии с приведенным далее описанием.
4. Рассчитывается *Q*-функция потерь для пакета (строка 16).
5. *Q*-функция потерь применяется для однократного обновления параметров сети (строка 17). Когда функция потерь определена, PyTorch обновляет параметры с помощью очень удобной функции автоматического дифференцирования.
6. Суммируются значения функции потерь (строка 18).
7. После выполнения всех обновлений параметров среднее значение функции потерь рассчитывается и записывается в журнал (строка 19).
8. `self.to_train` устанавливается равным нулю, чтобы гарантировать, что агент не начнет обучаться до того, как снова будет готов (строка 20).

Листинг 4.2. Реализация DQN, метод обучения

```
 1 # slm_lab/agent/algorithms/dqn.py
 2
 3 class VanillaDQN(SARSA):
 4     ...
 5
 6     @lab_api
 7     def train(self):
 8         ...
 9         clock = self.body.env.clock
10         if self.to_train == 1:
11             total_loss = torch.tensor(0.0)
12             for _ in range(self.training_iter):
13                 batch = self.sample()
14                 ...
15                 for _ in range(self.training_batch_iter):
16                     loss = self.calc_q_loss(batch)
17                     self.net.train_step(loss, self.optim,
                       ➥ self.lr_scheduler, clock=clock,
                       ➥ global_net=self.global_net)
18                     total_loss += loss
19             loss = total_loss / (self.training_iter *
               ➥ self.training_batch_iter)
```

```
20              self.to_train = 0
21              return loss.item()
22          else:
23              return np.nan
```

4.5.3. Память прецедентов

В этом разделе рассматривается класс памяти `Replay`, в котором реализована память прецедентов. При первом прочтении его можно пропустить — это не помешает понять алгоритм DQN. Он важен для понимания обсуждавшихся в разделе 4.3 идей, на которых основано хранение опыта.

Класс памяти `Replay` соответствует Memory API в SLM Lab и содержит три основных метода для очистки памяти, добавления прецедента и выборки пакета.

Инициализация и создание структур данных в Replay. В листинге 4.3 `__init__` инициализирует переменные класса, включая ключи хранилища в строке 20. В памяти `Replay` нужно отслеживать текущий размер памяти (`self.size`) и расположение самого последнего прецедента (`self.head`). Этот метод затем вызывает `reset` для создания пустых структур данных.

В классе памяти `Replay` все данные прецедентов хранятся в списках. В отличие от `OnPolicyReplay` из подраздела 2.6.6, в данном случае `reset` вызывается однократно для создания структур данных при инициализации памяти. Данные в памяти хранятся на протяжении всех шагов, что делает возможным повторное использование прецедентов.

Листинг 4.3. Replay, инициализация и создание структур данных

```
 1 # slm_lab/agent/memory/replay.py
 2
 3 class Replay(Memory):
 4     ...
 5
 6     def __init__(self, memory_spec, body):
 7         super().__init__(memory_spec, body)
 8         util.set_attr(self, self.memory_spec, [
 9             'batch_size',
10             'max_size',
11             'use_cer',
12         ])
13         self.is_episodic = False
14         self.batch_idxs = None
15         self.size = 0      # количество сохраненных прецедентов
16         self.seen_size = 0 # общее количество просмотренных прецедентов
17         self.head = -1     # индекс самого последнего прецедента
18         self.ns_idx_offset = self.body.env.num_envs if body.env.is_venv else 1
```

```
19          self.ns_buffer = deque(maxlen=self.ns_idx_offset)
20          self.data_keys = ['states', 'actions', 'rewards', 'next_states',
            ➥ 'dones']
21          self.reset()
22
23      def reset(self):
24          for k in self.data_keys:
25              if k != 'next_states': # повторно использовать self.states
26                  setattr(self, k, [None] * self.max_size)
27          self.size = 0
28          self.head = -1
29          self.ns_buffer.clear()
```

Обновление памяти Replay. `add_experience` в классе памяти `Replay` (листинг 4.4) действует иначе, чем в классах памяти при обучении по актуальному опыту.

Для добавления прецедента увеличивается `self.head` — индекс самого последнего примера опыта. При выходе индекса за пределы массива происходит возврат его значения к 0 (строка 13). Если память не заполнена до конца, то `self.head` будет указывать на пустую область в памяти, если заполнена — на самый старый прецедент, который будет замещен. Таким образом, в `Replay` хранится `self.max_size` самых последних прецедентов. Затем прецедент добавляется в память (строки 14–18). При этом значениям элементов списков в хранилище с индексами `self.head` присваиваются значения элементов текущего прецедента. В строках 20–22 отслеживаются фактический размер памяти и количество всех прецедентов, собранных агентом, а в строках 23 и 24 устанавливается флаг обучения `algorithm.to_train`.

Листинг 4.4. Replay, добавление прецедентов

```
 1 # slm_lab/agent/memory/replay.py
 2
 3 class Replay(Memory):
 4     ...
 5
 6     @lab_api
 7     def update(self, state, action, reward, next_state, done):
 8         ...
 9         self.add_experience(state, action, reward, next_state, done)
10
11     def add_experience(self, state, action, reward, next_state, done):
12         # Смещение указателя в голову списка,
           ➥ возврат к нулевому элементу при необходимости
13         self.head = (self.head + 1) % self.max_size
14         self.states[self.head] = state.astype(np.float16)
15         self.actions[self.head] = action
16         self.rewards[self.head] = reward
17         self.ns_buffer.append(next_state.astype(np.float16))
18         self.dones[self.head] = done
```

```
19          # Фактический размер занятой памяти
20          if self.size < self.max_size:
21              self.size += 1
22          self.seen_size += 1
23          algorithm = self.body.agent.algorithm
24          algorithm.to_train = algorithm.to_train or (self.seen_size >
            ➥ algorithm.training_start_step and self.head %
            ➥ algorithm.training_frequency == 0)
```

Выборка из памяти Replay. Выборка пакета прецедентов включает в себя выборку множества действительных индексов и выделение на их основе значимых примеров из каждого списка в хранилище памяти (листинг 4.5). Сначала посредством вызова функции `sample_idxs` происходит выбор `batch_size` индексов (строка 8). Индексы выбираются случайно и равномерно и замещают элементы списка индексов {0... `self.size`} (строка 18). Если `self.size == self.max_size`, то память заполнена до конца, что соответствует выборке индексов из всего хранилища памяти. Затем на основе этих индексов формируется пакет прецедентов (строки 9–14).

Касательно выборки следует упомянуть еще одну деталь. Чтобы уменьшить объем занимаемой памяти (RAM), следующие состояния не хранятся явным образом. Они уже присутствуют в буфере состояний, за исключением самых последних из них. Как следствие, построение пакета следующих состояний немного сложнее и выполнятся функцией `sample_next_states`, вызываемой в строке 12. Заинтересованный читатель может посмотреть, как это реализовано в SLM Lab, в `slm_lab/agent/memory/replay.py`.

Листинг 4.5. Replay, выборка

```
 1 # slm_lab/agent/memory/replay.py
 2
 3 class Replay(Memory):
 4     ...
 5
 6     @lab_api
 7     def sample(self):
 8         self.batch_idxs = self.sample_idxs(self.batch_size)
 9         batch = {}
10         for k in self.data_keys:
11             if k == 'next_states':
12                 batch[k] = sample_next_states(self.head, self.max_size,
                   ➥ self.ns_idx_offset, self.batch_idxs, self.states,
                   ➥ self.ns_buffer)
13             else:
14                 batch[k] = util.batch_get(getattr(self, k), self.batch_idxs)
15         return batch
16
17     def sample_idxs(self, batch_size):
18         batch_idxs = np.random.randint(self.size, size=batch_size)
19         ...
20         return batch_idxs
```

4.6. Обучение агента DQN

В листинге 4.6 приведен пример файла `spec` для обучения агента DQN. Файл имеется в SLM Lab в `slm_lab/spec/benchmark/dqn/dqn_cartpole.json`.

Листинг 4.6. Файл spec для DQN

```
# slm_lab/spec/benchmark/dqn/dqn_cartpole.json

{
    "vanilla_dqn_boltzmann_cartpole": {
        "agent": [{
            "name": "VanillaDQN",
            "algorithm": {
                "name": "VanillaDQN",
                "action_pdtype": "Categorical",
                "action_policy": "boltzmann",
                "explore_var_spec": {
                    "name": "linear_decay",
                    "start_val": 5.0,
                    "end_val": 0.5,
                    "start_step": 0,
                    "end_step": 4000,
                },
                "gamma": 0.99,
                "training_batch_iter": 8,
                "training_iter": 4,
                "training_frequency": 4,
                "training_start_step": 32
            },
            "memory": {
                "name": "Replay",
                "batch_size": 32,
                "max_size": 10000,
                "use_cer": false
            },
            "net": {
                "type": "MLPNet",
                "hid_layers": [64],
                "hid_layers_activation": "selu",
                "clip_grad_val": 0.5,
                "loss_spec": {
                    "name": "MSELoss"
                },
                "optim_spec": {
                    "name": "Adam",
                    "lr": 0.01
                },
                "lr_scheduler_spec": {
                    "name": "LinearToZero",
                    "frame": 10000
                },
```

```
46                 "gpu": false
47             }
48         }],
49         "env": [{
50             "name": "CartPole-v0",
51             "max_t": null,
52             "max_frame": 10000
53         }],
54         "body": {
55             "product": "outer",
56             "num": 1
57         },
58         "meta": {
59             "distributed": false,
60             "eval_frequency": 500,
61             "max_session": 4,
62             "max_trial": 1
63         },
64         ...
65     }
66 }
```

Как и в случае с SARSA, файл `spec` состоит из нескольких элементов (все номера строк относятся к листингу 4.6).

- **Алгоритм** — это `"VanillaDQN"` (строка 8), такое название дано для того, чтобы отличать его от модификации с прогнозной сетью из главы 5. Стратегия выбора действий — это стратегия Больцмана (строки 9 и 10) с линейной скоростью уменьшения (строки 11–17) параметра температуры τ, известного как переменная исследования `explore_var` (строка 11). Значение γ устанавливается в строке 18.
- **Архитектура сети** — многослойный перцептрон с одним скрытым слоем из 64 элементов (строка 32) и функцией активации SeLU (строка 33).
- **Оптимизатор** — Adam [68] со скоростью обучения 0,01 (строки 38–41).
- **Частота обучения** — обучение начинается после того, как агент сделает 32 шага в среде (строка 22), и в дальнейшем происходит каждые четыре шага (строка 21). На каждом шаге обучения производится выборка четырех пакетов прецедентов из памяти `Replay` (строка 20), и с их помощью выполняются восемь обновлений параметров (строка 19). В наборах содержится по 32 элемента (строка 26).
- **Память** — `Replay` с сохранением до 10 000 самых последних прецедентов (строка 27).
- **Среда** — CartPole из OpenAI Gym [18] (строка 50).
- **Длительность обучения** — 10 000 шагов (строка 52).
- **Контрольные точки** — агент оценивается каждые 500 шагов (строка 60).

Когда обучение выполняется с помощью стратегии Больцмана, важно установить для `action_pdtype` значение `Categorical`, чтобы агент выбирал действия из катего-

риального распределения вероятностей, порожденного стратегией выбора действий Больцмана. В то же время при ε-жадных стратегиях используется вырожденное распределение вероятностей (argmax). То есть агент будет выбирать действие с максимальным *Q*-значением с вероятностью 1,0.

Обратите внимание также на то, что в стратегии Больцмана максимальное значение для `τ` (строка 14) не ограничено 1, как в ε-жадных стратегиях. В этом случае переменная исследования `τ` представляет не вероятность, а температуру, поэтому она может принимать любое положительное значение. Однако она никогда не должна быть равной нулю, поскольку это приведет к возникновению ошибок, вызванных делением на ноль.

Чтобы обучить агента DQN с помощью SLM Lab, запустите в терминале команды, приведенные в листинге 4.7.

Листинг 4.7. Обучение агента DQN игре в CartPole

```
1 conda activate lab
2 python run_lab.py slm_lab/spec/benchmark/dqn/dqn_cartpole.json
  ➥ vanilla_dqn_boltzmann_cartpole train
```

Этот код использует файл `spec` для запуска обучения `Trial` с четырьмя сессиями `Session` для получения среднего результата и выводит графики для пробных испытаний, показанные на рис. 4.2.

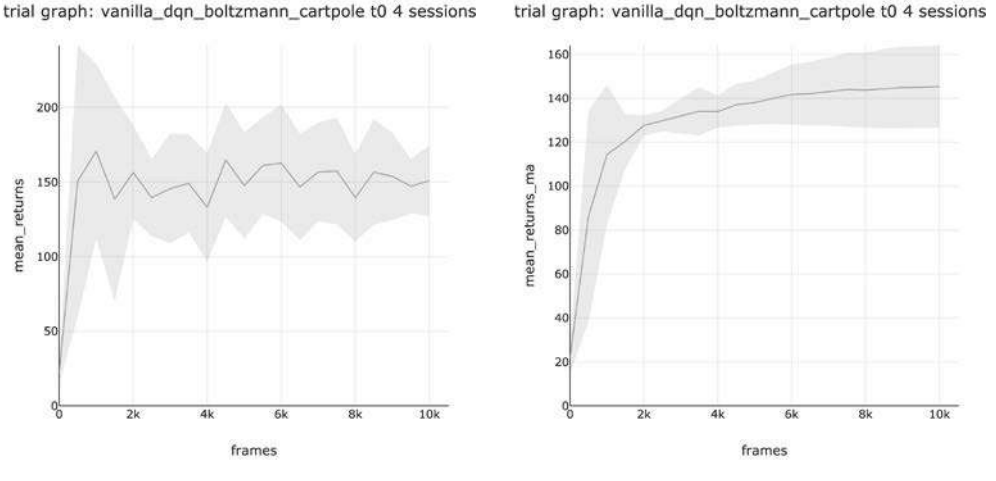


Рис. 4.2. Графики пробных испытаний для DQN из SLM Lab, усредненные по результатам четырех сессий. По вертикальной оси отложены полные вознаграждения (mean_return для оценки рассчитывается без дисконтирования), усредненные по данным из восьми эпизодов на протяжении контрольных точек. По горизонтальной оси отложены кадры всего обучения. Справа приведен график со скользящим средним по оценкам в 100 контрольных точках

4.7. Результаты экспериментов

В этом разделе рассматривается, как изменение архитектуры нейронной сети влияет на производительность DQN в CartPole. Для поиска по сетке параметров архитектуры сети используется функция экспериментирования из SLM Lab.

4.7.1 Эксперимент по определению влияния архитектуры сети

Архитектура нейронных сетей влияет на их способность аппроксимировать функции. В этом эксперименте выполняется простой поиск по сетке на нескольких вариантах настройки скрытого слоя, затем выводятся графики и сравнивается производительность. В листинге 4.8 показан файл `spec` для эксперимента, представляющий собой расширенную версию из листинга 4.6. В строках 8–15 задаются параметры поиска по сетке архитектур скрытых слоев `net.hid_layers` *Q*-сети. Полностью файл `spec` имеется в SLM Lab в `slm_lab/spec/benchmark/dqn/dqn_cartpole.json`.

Листинг 4.8. Файл spec для DQN со спецификацией поиска по разным вариантам архитектуры сети

```
 1 # slm_lab/spec/benchmark/dqn/dqn_cartpole.json
 2
 3 {
 4     "vanilla_dqn_boltzmann_cartpole": {
 5         ...
 6         "search": {
 7             "agent": [{
 8                 "net": {
 9                     "hid_layers__grid_search": [
10                         [32],
11                         [64],
12                         [32, 16],
13                         [64, 32]
14                     ]
15                 }
16             }]
17         }
18     }
19 }
```

В листинге 4.9 приведены команды для запуска эксперимента. Заметьте, что здесь используется тот же файл `spec`, что и при обучении, но режим обучения `train` изменен на режим поиска `search`.

Листинг 4.9. Запуск эксперимента с поиском по разным вариантам архитектуры сети в соответствии с файлом spec

```
1 conda activate lab
2 python run_lab.py slm_lab/spec/benchmark/dqn/dqn_cartpole.json
  ➥ vanilla_dqn_boltzmann_cartpole search
```

Таким образом, будет запущен эксперимент `Experiment`, который породит четыре пробных испытания `Trial`. Их получают, подставляя в исходный файл `spec` для DQN с использованием стратегии Больцмана разные значения `net.hid_layers`. В каждом `Trial` запускаются четыре повторяющиеся сессии `Session` для получения усредненных результатов. На рис. 4.3 приведены графики для множества пробных испытаний из этого эксперимента.

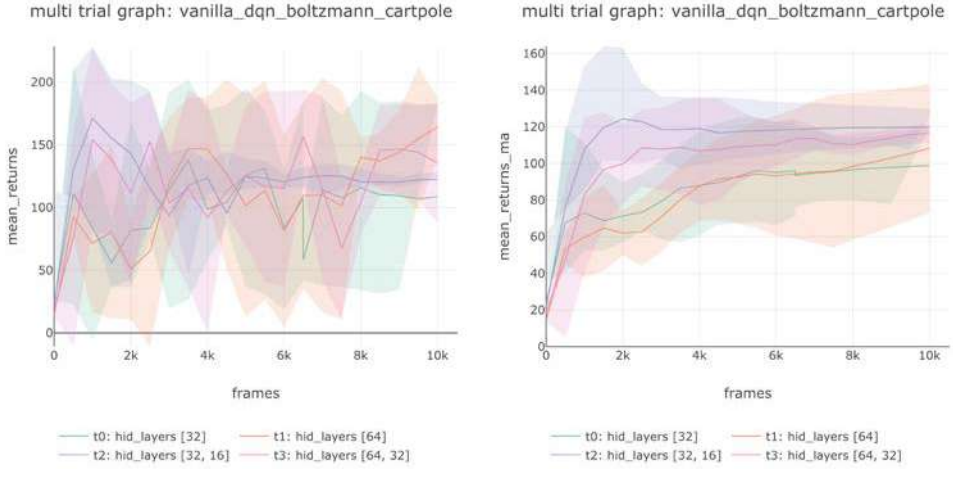


Рис. 4.3. Влияние разных вариантов архитектуры сети. Производительность сетей с двумя скрытыми слоями, обладающими более высокой способностью к обобщению, немного выше, чем сетей с одним скрытым слоем

Из рис. 4.3 видно, что производительность сетей с двумя скрытыми слоями (испытания *2* и *3*) выше, чем сетей с одним слоем (испытания *0* и *1*), благодаря их повышенной способности к обобщению. Когда количество слоев одинаковое, сети с меньшим количеством элементов (испытания *0* и *2*) при условии, что их способность к обобщению довольно высока, учатся немного быстрее. Это связано с тем, что в них нужно обновлять меньше параметров. Разработка архитектуры нейронной сети — тема главы 12.

Из эксперимента и пробных испытаний видно, что простой алгоритм DQN может быть неустойчивым и не очень производительным. Это показывают широкие

диапазоны ошибок в кривых обучения и полные вознаграждения в CartPole, меньшие 200. В следующей главе будут рассмотрены методы улучшения DQN, позволяющие сделать его достаточно мощным для решения задач Atari.

4.8. Резюме

В этой главе дано введение в алгоритм DQN. Он во многом схож с SARSA, но имеет одно важное отличие. В DQN для расчета целевых Q-значений используется максимальное Q-значение в следующем состоянии. В результате DQN настраивает оптимальную Q-функцию вместо Q-функции, соответствующей текущей стратегии. Благодаря этому DQN является алгоритмом обучения по отложенному опыту, в котором настроенная Q-функция не зависит от стратегии сбора прецедентов.

Следовательно, агент DQN может повторно применять накопленные прежде прецеденты, которые хранятся в памяти. На каждой итерации обучения для настойки Q-функции случайным образом выбирается один или несколько пакетов прецедентов. Это помогает устранить корреляцию между прецедентами, используемыми для обучения. Сочетание повторного использования данных с устранением корреляции значительно повышает эффективность выборки в DQN по сравнению с SARSA.

Было показано, что применяемая в DQN стратегия сбора прецедентов должна обладать двумя характеристиками: облегчать исследование пространства состояний и с течением времени приближаться к оптимальной стратегии. Одна из таких стратегий — ε-жадная, но в этой главе представлен и альтернативный вариант, известный как стратегия Больцмана. Одно из основных преимуществ этой стратегии заключается в том, что она исследует пространство состояний и действий менее случайным образом, чем ε-жадная стратегия. Второе ее преимущество состоит в том, что действия выбираются из распределения вероятностей, которое плавно изменяется с изменением Q-значений.

4.9. Рекомендуемая литература

- *Mnih V., Kavukcuoglu K., Silver D., Graves A., Antonoglou I., Wierstra D., Riedmiller M. A.* Playing Atari with Deep Reinforcement Learning. 2013 [88].
- *Lin L.-J.* Self-Improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching. 1992 [82].
- *Sutton R. S., Barto A. G.* Reinforcement Learning: An Introduction. Second ed. 2018. Главы 6 и 7 в [132].
- *Levine S.* Sep 13: Value Functions Introduction, Lecture 6. 2017 [76].
- *Levine S.* Sep 18: Advanced Q-Learning Algorithms, Lecture 7. 2017 [77].

4.10. Историческая справка

Компонент *Q*-обучения в DQN был предложен в 1989 году Кристофером Уоткинсом и описан в его кандидатской диссертации *Learning from Delayed Rewards* [145]. Вскоре — в 1992 году — Лонг-Джи Лином была выдвинута идея памяти прецедентов [82]. Это сыграло важную роль в повышении эффективности *Q*-обучения. Однако в последующие годы в области глубокого *Q*-обучения выдающихся успехов достигнуто не было. Это и неудивительно, ведь вычислительные мощности в 1990-х и начале 2000-х годов были ограничены, архитектура сетей глубокого обучения требовала большого количества данных, а сигналы обратной связи в RL были зашумленными и отложенными. Развитие возобновилось после появления графических процессоров общего назначения и выпуска CUDA в 2006 году [96]. Кроме того, ему способствовал повторный всплеск интереса к глубокому обучению в сообществе машинного обучения, который произошел в середине 2000-х годов и резко возрос после 2012-го.

В 2013 году важным открытием стала опубликованная DeepMind статья *Playing Atari with Deep Reinforcement Learning* [88]. Авторы ввели термин «глубокие *Q*-сети» (Deep *Q*-Networks, DQN) и впервые продемонстрировали пример обучения с помощью RL стратегии управления непосредственно на высокоразмерных данных в виде изображений. Улучшения не заставили себя ждать: в 2015 году были разработаны двойная *Q*-сеть (Double DQN) [141] и приоритизированная память прецедентов (Prioritized Experience Replay) [121]. Однако основным достижением стало объединение алгоритма, представленного в этой главе, с простой сверточной нейронной сетью, обработкой состояний и обучением на графическом процессоре. В главе 5 более подробно обсуждается статья *Playing Atari with Deep Reinforcement Learning* [88].

5 Улучшение DQN

В этой главе рассматриваются три модификации алгоритма DQN: прогнозные сети, двойная DQN [141] и приоритизированная память прецедентов [121]. Каждая из них решает отдельную проблему в DQN, поэтому их объединение дает значительное повышение производительности.

В разделе 5.1 обсуждаются прогнозные сети, которые являются отложенной копией $\hat{Q}^{\pi}(s, a)$. Прогнозная сеть применяется для порождения максимального Q-значения в следующем состоянии s' при расчете $Q^{\pi}_{1:n}$. Это отличает ее от алгоритма DQN (см. главу 4), в котором для расчета $Q^{\pi}_{1:n}$ используется $\hat{Q}^{\pi}(s, a)$. Применение прогнозной сети позволяет сделать обучение более устойчивым благодаря снижению скорости изменения $Q^{\pi}_{1:n}$.

Далее в разделе 5.2 рассматривается алгоритм двойной DQN, вычисляющий $Q^{\pi}_{1:n}$ с помощью двух Q-сетей. Первая сеть выбирает действие, которое соответствует ее оценке максимального Q-значения. Вторая сеть порождает для действия, выбранного первой сетью, Q-значение, которое применяется в расчете $Q^{\pi}_{1:n}$. В данной модификации учтено систематическое завышение DQN оценочных Q-значений по сравнению с истинными, которое может привести к замедлению обучения и неэффективным стратегиям.

Дальнейшее улучшение DQN может быть произведено за счет изменения процесса выборки пакетов прецедентов из памяти. Вместо того чтобы выбирать прецеденты случайно и равномерно, их можно приоритизировать в соответствии с тем, насколько они информативны для агента на текущий момент. Данный метод известен как *приоритизированная память прецедентов* (Prioritized Experience Replay). В нем прецеденты, из которых агент может приобрести больше навыков, выбираются чаще, чем малоинформативные. Это обсуждается в разделе 5.3.

После знакомства с идеями, лежащими в основе всех модификаций, в разделе 5.4 приводится реализация, в которую включены все эти изменения.

В конце главы объединяются разные элементы, рассмотренные в главах 4 и 5, и агент DQN обучается играм Pong и Breakout из Atari. Реализации, архитектура

сети и гиперпараметры основаны на алгоритмах, описанных в трех статьях: *Human-Level Control through Deep Reinforcement Learning* [89], *Deep Reinforcement Learning with Double Q-Learning* [141] и *Prioritized Experience Replay* [121].

5.1. Прогнозные сети

Первое усовершенствование алгоритма DQN — это применение прогнозной сети для расчета $Q^{\pi}_{цель}$. Было предложено Мнихом и его коллегами в статье *Human-Level Control through Deep Reinforcement Learning* [89] и позволяет повысить устойчивость обучения. Это обусловлено тем, что в первоначальном алгоритме DQN $Q^{\pi}_{цель}$ постоянно изменяется, так как зависит от $\hat{Q}^{\pi}(s, a)$. Во время обучения параметры θ *Q*-сети настраиваются так, чтобы минимизировать разницу между $\hat{Q}^{\pi}(s, a) = Q^{\pi_\theta}(s, a)$ и $Q^{\pi}_{цель}$, что трудно сделать, когда $Q^{\pi}_{цель}$ меняется на каждом шаге обучения.

Для уменьшения изменений $Q^{\pi}_{цель}$ между шагами обучения применяется прогнозная сеть. Это вторая сеть с параметрами ϕ, которая является отложенной копией *Q*-сети $Q^{\pi_\phi}(s, a)$. Прогнозная сеть $Q^{\pi_\phi}(s, a)$ используется для расчета $Q^{\pi}_{цель}$, как показано в измененном обновлении функции Беллмана (уравнение (5.2)). Для упрощения сравнения приведено исходное обновление DQN (уравнение (5.1)):

$$Q^{\pi_\theta}_{цель}(s, a) = r + \gamma \max_{a'} Q^{\pi_\theta}(s', a'); \tag{5.1}$$

$$Q^{\pi_\phi}_{цель}(s, a) = r + \gamma \max_{a'} Q^{\pi_\phi}(s', a'). \tag{5.2}$$

Периодически происходит обновление ϕ до текущих значений θ, что известно как *обновление замещением* (replacement update). Частота обновлений ϕ определяется видом задачи. Например, в играх Atari ϕ обновляется каждые 1000–10 000 шагов в среде. Для менее сложных задач может не понадобиться ждать так долго — достаточно будет обновлять ϕ каждые 100–1000 шагов.

Почему это позволяет повысить устойчивость обучения? При каждом вычислении $Q^{\pi_\theta}_{цель}(s, a)$ *Q*-функция, представленная параметрами θ, будет немного другой. Поэтому $Q^{\pi_\theta}_{цель}(s, a)$ может принимать разные значения для одной той же пары (s, a). Возможно даже, что $Q^{\pi_\theta}_{цель}(s, a)$ на следующем шаге обучения будет значительно отличаться от предыдущей оценки. Эта «движущаяся цель» может снизить устойчивость обучения, так как неясно, к каким значениям должна приближаться сеть. Введение прогнозной сети в буквальном смысле останавливает «движущуюся цель». Между обновлениями ϕ до θ эти ϕ фиксированы и представленная ими *Q*-функция не изменяется. Это сводит данную задачу к стандартной регрессии в обучении с учителем [77]. Не вызывая фундаментальных изменений в исходной задаче оптимизации, прогнозная сеть помогает повысить устойчивость обучения и снижает вероятность возникновения в стратегии высокой дисперсии или колебаний [77, 89, 141].

Различия между алгоритмом DQN с прогнозной сетью (алгоритм 5.1) и исходным алгоритмом DQN (см. алгоритм 4.1) приведены далее.

- Частота обновлений прогнозной сети F — дополнительный гиперпараметр, который нужно выбирать (строка 7).
- В строке 8 происходит инициализация дополнительной сети как прогнозной, и ее параметрам ϕ присваиваются значения θ.
- В строке 17 y_i рассчитывается с помощью прогнозной сети $Q^{\pi_\phi}_{1:\infty}$.
- Прогнозная сеть периодически обновляется (строки 26–29).

Следует отметить, что именно параметры сети θ используются при расчете $Q^{\pi_\theta}_{1:\infty}(s, a)$ (строка 20) и обновляются во время обучения (строка 22). Это происходит так же, как в исходном алгоритме DQN.

Алгоритм 5.1. DQN с прогнозной сетью

```
 1: Инициализация скорости обучения α
 2: Инициализация τ
 3: Инициализация количества пакетов данных на шаг обучения B
 4: Инициализация количества обновлений на пакет U
 5: Инициализация размера пакета данных N
 6: Инициализация памяти прецедентов с максимальным размером K
 7: Инициализация частоты обновления прогнозной сети F
 8: Инициализация параметров сети θ случайными значениями
 9: Инициализация параметров прогнозной сети ϕ = θ
10: for m = 1...MAX_STEPS do
11:     Накопить и сохранить h прецедентов (s_i, a_i, r_i, s'_i),
        ➥ используя текущую стратегию
12:     for b = 1...B do
13:         Выбрать пакет b прецедентов из памяти
14:         for u = 1...U do
15:             for i = 1...N do
16:                 # Рассчитать целевые Q-значения для каждого прецедента
17:                 y_i = r_i + δ_{s'_i} γ max_{a'_i} Q^{π_ϕ}(s'_i, a'_i), где δ_{s'_i} = 0, если s'_i –
                    ➥ конечное состояние, иначе δ_{s'_i} = 1.
18:             end for
19:             # Расчет функции потерь, например, с помощью средней
                ➥ квадратической ошибки
20:             L(θ) = (1/N) Σ_i (y_i − Q^{π_θ}(s_i, a_i))^2
21:             # Обновление параметров сети
22:             θ = θ − α∇_θ L(θ)
23:         end for
24:     end for
25:     Уменьшение τ
26:     if (m mod F) == 0 then
27:         # Обновить параметры прогнозной сети
28:         ϕ = θ
29:     end if
30: end for
```

Периодическое замещение параметров ϕ прогнозной сети копией параметров сети θ — общепринятый способ выполнения обновлений. В качестве альтернативного варианта на каждом временном шаге ϕ может присваиваться среднее значение между ϕ и θ, как показано в уравнении (5.4). Это известно как *обновление Поляка* и может трактоваться как мягкое обновление: на каждом шаге параметры ϕ и θ смешиваются, чтобы породить новую прогнозную сеть. В отличие от *обновления замещением* (уравнение (5.3)), в этом случае ϕ изменяется на каждом шаге, но медленнее, чем обучающаяся сеть θ:

$$\phi \leftarrow \theta; \tag{5.3}$$

$$\phi \leftarrow \beta\phi + (1 - \beta)\theta. \tag{5.4}$$

Гиперпараметр β управляет скоростью изменения ϕ, определяя, в какой мере старая прогнозная сеть ϕ сохраняется при каждом обновлении. Чем выше β, тем медленнее изменяется ϕ.

У каждого из подходов есть свои преимущества, и ни один из них не лучше другого. Основное преимущество обновления замещением в том, что благодаря неизменности ϕ на некотором количестве шагов временно устраняется «движущаяся цель». И наоборот, при обновлении Поляка ϕ по-прежнему изменяется на каждой итерации обучения, но более постепенно, чем θ. Однако при обновлении замещением наблюдается динамическое отставание между ϕ и θ, которое зависит от числа шагов, прошедших после последнего обновления ϕ. У обновления Поляка нет этой особенности, так как значение, полученное смешением ϕ и θ, остается постоянным.

Один из недостатков прогнозных сетей — то, что они могут замедлять обучение, поскольку $Q^{\pi}_{\text{tar}}(s, a)$ генерируется старой прогнозной сетью. Если ϕ и θ — очень близкие значения, обучение может быть неустойчивым, но если ϕ меняется слишком медленно, то процесс обучения может быть более длительным, чем нужно. Для поиска разумного соотношения между устойчивостью и скоростью обучения нужно настраивать гиперпараметр (частоту обновления или β), управляющий скоростью изменения ϕ. В разделе 5.6 показано влияние изменения частоты обновления на агента DQN, обучающегося игре Pong из Atari.

5.2. Двойная DQN

Второе усовершенствование алгоритма DQN — это применение *двойной оценки* для вычисления $Q^{\pi}_{\text{tar}}(s, a)$. Известная под названием *двойной DQN*, данная версия алгоритма решает проблему завышения оценок Q-значений. Сначала разберемся, почему в исходном алгоритме DQN происходит завышение и почему это является проблемой, затем опишем ее решение с помощью двойной DQN.

В DQN $Q^{\pi}_{\text{tar}}(s, a)$ строится путем выбора максимальной оценки Q-значения в состоянии s' (уравнение (5.5)):

$$Q^{\pi_0}_{\text{tar}}(s, a) = r + \gamma \max_{a'} Q^{\pi_0}(s', a') =$$

$$= r + \max\left(Q^{\pi_0}(s', a'_1), Q^{\pi_0}(s', a'_2), \ldots, Q^{\pi_0}(s', a'_n)\right). \quad (5.5)$$

Q-значение — это ожидаемая будущая отдача от выбора действия a в состоянии s, поэтому расчет $\max_{a'} Q^{\pi_0}(s', a')$ подразумевает выбор максимального значения из набора ожидаемых полезностей $\left(Q^{\pi_0}(s', a'_1), Q^{\pi_0}(s', a'_2), \ldots, Q^{\pi_0}(s', a'_n)\right)$.

В статье *Deep Reinforcement Learning with Double Q-Learning* [141] Ван Хасселт и другие показали, что если $Q^{\pi_0}(s', a')$ содержит какие-нибудь ошибки, то $\max_{a'} Q^{\pi_0}(s', a')$ будет иметь положительное смещение и полученные в результате оценки Q-значений окажутся завышенными. К сожалению, существует много причин неточности $Q^{\pi_0}(s', a')$. Аппроксимация функций с помощью нейронных сетей неидеальна, агент может не исследовать среду полностью, да и сама среда может быть зашумленной. Таким образом, следует ожидать, что $Q^{\pi_0}(s', a')$ будет содержать ошибки и оценки Q-значений окажутся завышенными. Более того, чем шире выбор действий в состоянии s', тем больше вероятность завышения оценки. Конкретный пример приведен в примечании 5.1.

Совершенно не очевидно, до какой степени завышение оценки Q-значения является проблемой. Например, если все оценки Q-значений были завышены в равной мере, то агент по-прежнему будет выбирать правильное действие a в состоянии s и можно ожидать, что производительность не ухудшится. Более того, из-за неопределенности завышение оценок может даже оказаться полезным [61]. Например, в начале обучения завышенные оценки $Q^{\pi}(s, a)$ для непосещенных или редко посещаемых пар (s, a) могут повысить вероятность их посещения, что позволит агенту узнать, насколько они хороши или плохи.

Однако DQN дает завышенные оценки $Q^{\pi}(s, a)$ для пар (s, a), которые посещаются часто. Это становится проблемой для агента, если исследование им (s, a) происходит неравномерно. Тогда завышение оценок $Q^{\pi}(s, a)$ также будет неравномерным, что может привести к неверному изменению ранжирования действий относительно $Q^{\pi}(s, a)$. При таких обстоятельствах a, которое агент считает лучшим для выбора в s, на самом деле таковым не является. Когда завышение оценки $Q^{\pi}(s, a)$ происходит вкупе с обучением с бутстрэппингом (как в DQN), взаимосвязанные неверные Q-значения будут обратно распространены в более ранние пары (s, a) и внесут смещение и в их оценки. В связи с этим целесообразно уменьшение завышения оценок Q-значений.

Алгоритм двойной DQN уменьшает завышение оценок Q-значений путем настройки двух оценок Q-функции, полученных на разных прецедентах. Максимизирующее Q-значение действия a' выбирается с помощью первой оценки.

Q-значение, с помощью которого рассчитывается $Q^{\pi}_{tar}(s, a)$, порождается второй оценкой с помощью действия a, выбранного с использованием первой оценки. Применение второй Q-функции, настроенной на других прецедентах, устраняет положительное смещение из итоговой оценки.

Примечание 5.1. Оценка максимального ожидаемого значения

Завышение оценок Q-значений — это частный случай более общей хорошо известной проблемы. Ожидаемое максимальное значение в наборе оценочных значений имеет положительное смещение, когда оценки значений зашумлены [128]. В DQN максимальное ожидаемое значение — это $\mathbb{E}\left[\max_a Q^{\pi}(s, a)\right]$, а зашумленные оценки порождаются Q^{π_u}.

Чтобы понять происхождение этой проблемы, рассмотрим состояние s, в котором $Q^{\pi}(s, a) = 0$ для всех доступных действий a. Предположим, что оценки Q-значений зашумлены, но не имеют положительного смещения. Это может быть смоделировано как выборка Q-значений из стандартного нормального распределения со средним значением 0 и стандартным отклонением 1.

Предельное значение, на которое оценка $\max_{a'} Q^{\pi}(s', a')$ будет завышена, можно найти, взяв k значений из стандартного нормального распределения и выбрав максимальное. В этом случае k представляет количество действий a' в состоянии s'. После многократного повторения этого процесса рассчитываем среднее значение для всех максимальных значений, чтобы оценить ожидаемое максимальное значение для каждого k.

В табл. 5.1 приведены ожидаемые максимальные значения при k = 1... 10 и выборке 10 000 значений для каждого k. Правильное значение $\max_{a'} Q^{\pi}(s', a') = 0$, и при k = 1 оценка ожидаемого значения верна. Однако по мере увеличения k в оценке появляется все большее положительное смещение. Например, при k = 2 $\mathbb{E}\left[\max_a Q^{\pi}(s, a)\right] = 0{,}56$, а при k = 10 $\mathbb{E}\left[\max_a Q^{\pi}(s, a)\right] = 1{,}53$.

Таблица 5.1. Ожидаемые значения $\max_a Q^{\pi}(s, a)$, когда оценки для $Q^{\pi}(s, a)$ зашумлены, но не имеют смещения, а $Q^{\pi}(s, a) = 0$ для всех a. Это воспроизведение эксперимента Смита и Уинклера из The Optimizer's Curse [128]

Количество действий	$\mathbb{E}\left[\max_a Q^{\pi}(s, a)\right]$	**Количество действий**	$\mathbb{E}\left[\max_a Q^{\pi}(s, a)\right]$
1	0	6	1,27
2	0,56	7	1,34
3	0,86	8	1,43
4	1,03	9	1,48
5	1,16	10	1,53

Преобразование DQN в двойную DQN происходит просто. Какие изменения нужно внести, станет понятнее, если переписать $Q^{\pi}_{\text{tar}}(s,a)$ в виде уравнения (5.6):

$$Q^{\pi}_{\text{tar:DQN}}(s,a) = r + \gamma \max_{a'} Q^{\pi_\theta}(s',a') = r + \gamma Q^{\pi_\theta}\left(s', \max_{a'} Q^{\pi_\theta}(s',a')\right). \tag{5.6}$$

В алгоритме DQN для выбора действия a' и получения оценки Q-функции для него используется одна и та же сеть θ. В двойной DQN применяются две разные сети, θ и ϕ: первая — для выбора a', вторая — для расчета Q-значения для (s', a'):

$$Q^{\pi}_{\text{tar:double-DQN}}(s,a) r + \gamma Q^{\pi_\psi}\left(s', \max_{a'} Q^{\pi_\theta}(s',a')\right). \tag{5.7}$$

После введения прогнозной сети в разделе 5.1 у нас уже есть две сети: обучаемая θ и прогнозная ϕ. Обе они обучались на пересекающихся прецедентах, но если количество шагов между установкой ϕ = θ довольно велико, то на практике они работают по-разному — как две разные сети в двойной DQN.

Обучаемая сеть θ используется для выбора действий. Важно обеспечить, чтобы после преобразования в двойную DQN она по-прежнему настраивала оптимальную стратегию [141]. Прогнозная сеть ϕ применяется для оценки этого действия. Следует отметить, что если между ϕ и θ нет расхождения, то есть если ϕ = θ, то уравнение (5.7) сводится к исходному уравнению для DQN.

Двойная DQN с прогнозной сетью описана в алгоритме 5.2, где в строке 17 y_i рассчитывается с помощью обеих сетей, θ и ϕ. Это его единственное отличие от алгоритма 5.1.

Алгоритм 5.2. Двойная DQN с целевой сетью

```
 1: Инициализация скорости обучения α
 2: Инициализация τ
 3: Инициализация количества пакетов данных на шаг обучения B
 4: Инициализация количества обновлений на пакет U
 5: Инициализация размера пакета данных N
 6: Инициализация памяти прецедентов с максимальным размером K
 7: Инициализация частоты обновления прогнозной сети F
 8: Инициализация параметров сети θ случайными значениями
 9: Инициализация параметров прогнозной сети ϕ = θ
10: for m = 1...MAX_STEPS do
11:     Накопить и сохранить h прецедентов (s_i, a_i, r_i, s'_i),
        ➥ используя текущую стратегию
12:     for b = 1...B do
13:         Выбрать пакет b прецедентов из памяти
14:         for u = 1...U do
15:             for i = 1...N do
16:                 # Рассчитать целевые Q-значения для каждого прецедента
17:                 y_i = r_i + δ_{s'_i} γQ^{π_ϕ}(s'_i, max_{a'_i} Q^{π_θ}(s'_i, a'_i)), где δ_{s'_i} = 0, если
                    ➥s'_i — конечное состояние, иначе δ_{s'_i} = 1.
```

```
            end for
            # Расчет функции потерь, например, с помощью средней
            ➥ квадратической ошибки
            L(θ) = 1/N Σ_i (y_i − Q^π_θ(s_i, a_i))^2
            # Обновление параметров сети
            θ = θ − α∇_θ L(θ)
        end for
    end for
    Уменьшение τ
    if (m mod F) == 0 then
        # Обновить параметры прогнозной сети
        ϕ = θ
    end if
end for
```

5.3. Приоритизированная память прецедентов

Последнее усовершенствование DQN — это использование *приоритизированной памяти прецедентов* (Prioritized Experience Replay, PER), введенное Шоулом и другими в 2015 году [121]. Его основная суть в том, что некоторые прецеденты, хранящиеся в памяти, более информативны, чем другие. Если можно обучать агента, выбирая более информативные прецеденты чаще, чем менее информативные, то агент будет учиться быстрее.

Можно представить, что при решении нового задания некоторые прецеденты будут содержать больше сведений, чем другие. Рассмотрим пример: агент-андроид пытается научиться вставать. В начале каждого эпизода он инициализируется сидящим на полу. Сначала большинство действий будет приводить к падению агента на пол, и лишь некоторые прецеденты будут нести значимую информацию о том, какая комбинация движений суставов позволит достичь равновесия и встать. Для того чтобы научить агента вставать, эти прецеденты важнее, чем те, в которых он остается на полу. Они позволяют агенту обучиться тому, как делать правильно, а не тому, что он может сделать неправильно. В то же время можно рассматривать прецеденты, в которых $Q^{\pi_\theta}(s,a)$ сильнее всего отклоняется от $Q^{\pi}_{\text{tar}}(s,a)$. Это наиболее неожиданные для агента прецеденты, и можно считать, что он должен как можно больше обучаться на них. Агент может учиться быстрее, если он тренируется на этих прецедентах чаще, чем на прецедентах, для которых он может точно предсказать $Q^{\pi}_{\text{tar}}(s,a)$. В свете этих размышлений можем ли мы предпринять что-то лучшее, чем равномерная выборка прецедентов из памяти? Может быть, вместо этого применить *приоритизированное* обучение агента?

Приоритизированная память прецедентов основана на этой простой и интуитивно понятной идее, но при реализации такого обучения возникает две проблемы.

Первая связана с автоматическим присвоением приоритетов всем прецедентам. Вторая — с эффективной выборкой прецедентов из памяти с помощью этих приоритетов.

Естественное решение первой проблемы — вывод приоритета из абсолютной разности между $Q^{\pi_w}(s,a)$ и $Q^{\pi}_{tar}(s,a)$, что известно как TD-ошибка. Чем больше разница между этими двумя значениями, тем больше несоответствие между ожиданиями агента и тем, что происходит на следующем шаге, и тем сильнее агент должен корректировать $Q^{\pi_w}(s,a)$. Кроме того, после несложных вычислений и небольших изменений в реализации TD-ошибка будет доступна для всех прецедентов как часть алгоритмов DQN или двойной DQN. Остается лишь вопрос о том, какие приоритеты присваивать прецедентам вначале, когда TD-ошибка еще недоступна. Как правило, это решается установлением для приоритета большого постоянного значения, чтобы каждый из прецедентов был выбран хотя бы однажды.

Шоул с коллегами предложили два варианта выборки на основе баллов: ранжированную и пропорциональную приоритизацию. Оба подхода основаны на интерполяции между жадной приоритизацией (всегда выбирать прецеденты с самым высоким количеством баллов) и равномерной случайной выборкой. Это гарантирует, что прецеденты с большим количеством баллов выбираются чаще, но вероятность выбора для всех прецедентов отлична от нуля. Здесь рассматривается только метод пропорциональной приоритизации. С деталями ранжированной приоритизации можно ознакомиться в статье *Prioritized Experienced Replay* [121]. Если ω_i — TD-ошибка для i-го прецедента, ε — произвольно заданная малая положительная величина[1] и $\eta \in [0, \infty)$, то приоритет прецедента может быть определен из следующего уравнения:

$$P(i) = \frac{\left(|\omega_i| + \varepsilon\right)^{\eta}}{\sum_j \left(|\omega_j| + \varepsilon\right)^{\eta}}. \tag{5.8}$$

Константа ε нужна, для того чтобы все эпизоды были выбраны хотя бы раз, когда $\omega_i = 0$. η определяет степень приоритизации. $\eta = 0$ соответствует равномерной выборке, так как все прецеденты будут иметь приоритет 1. Как показывает уравнение (5.9), чем больше η, тем выше степень приоритизации:

$$\begin{aligned}
&\eta = 0{,}0\text{: } (\omega_1 = 2{,}0, \omega_2 = 3{,}5) \rightarrow P(1) = 0{,}50, P(2) = 0{,}50);\\
&\eta = 0{,}5\text{: } (\omega_1 = 2{,}0, \omega_2 = 3{,}5) \rightarrow P(1) = 0{,}43, P(2) = 0{,}57);\\
&\eta = 1{,}0\text{: } (\omega_1 = 2{,}0, \omega_2 = 3{,}5) \rightarrow P(1) = 0{,}36, P(2) = 0{,}64);\\
&\eta = 1{,}5\text{: } (\omega_1 = 2{,}0, \omega_2 = 3{,}5) \rightarrow P(1) = 0{,}30, P(2) = 0{,}70);\\
&\eta = 2{,}0\text{: } (\omega_1 = 2{,}0, \omega_2 = 3{,}5) \rightarrow P(1) = 0{,}25, P(2) = 0{,}75).
\end{aligned} \tag{5.9}$$

[1] Эта ε не имеет отношения к ε-жадной стратегии — это другая константа.

5.3.1. Выборка по значимости

Приоритизация определенных прецедентов меняет математическое ожидание всего распределения данных, что вносит систематическую ошибку в процесс обучения. Это может быть исправлено умножением TD-ошибки для всех прецедентов на набор весов, что известно как *выборка по значимости*. Когда значение систематической ошибки невелико, нельзя с уверенностью сказать, насколько эффективна выборка по значимости, ведь присутствуют и другие факторы, такие как зашумленность действий или сильно нестационарное распределение данных. Они могут влиять сильнее, чем небольшая систематическая ошибка, особенно на ранних этапах обучения. Шоул с коллегами [121] предположили, что исправление систематической ошибки имеет значение лишь в конце обучения, и показали неоднозначность эффекта от ее исправления. В одних случаях введение выборки по значимости привело к повышению производительности, в других — разница была невелика либо производительность ухудшалась. Для простоты в реализации, обсуждаемой в этой главе, выборка по значимости была опущена. За более подробной информацией обращайтесь к статье *Prioritized Experience Replay* [121] и лекции 4 (https://youtu.be/tWNpiNzWuO8) для класса *Deep Reinforcement Learning* Сергея Левина [74].

Очевидным решением для добавления приоритизированной памяти прецедентов является расширение двойной DQN с алгоритмом прогнозной сети. Для этого необходимо выполнить четыре изменения.

1. В памяти прецедентов для каждого прецедента нужно хранить дополнительный элемент — его приоритет (строка 14 алгоритма 5.3).
2. Выборка пакетов из памяти должна происходить пропорционально их приоритетам (строка 16).
3. Нужно рассчитывать и хранить TD-ошибки для прецедентов (строка 22).
4. Наконец, следует использовать TD-ошибки для обновления приоритетов соответствующих прецедентов в памяти (строки 29 и 30).

Алгоритм 5.3. Двойная DQN с целевой сетью и приоритизированным буфером повторений

```
 1: Инициализация скорости обучения α
 2: Инициализация τ
 3: Инициализация количества пакетов данных на шаг обучения B
 4: Инициализация количества обновлений на пакет U
 5: Инициализация размера пакета данных N
 6: Инициализация памяти прецедентов с максимальным размером K
 7: Инициализация частоты обновления прогнозной сети F
 8: Инициализация максимального приоритета P
 9: Инициализация ε
10: Инициализация параметра приоритизации η
11: Инициализация параметров сети θ случайными значениями
12: Инициализация параметров прогнозной сети ϕ = θ
13: for m = 1...MAX_STEPS do
```

```
    Накопить и сохранить h прецедентов (s_i, a_i, r_i, s'_i, p_i), где p_i = P
    for b = 1...B do
        Выбрать приоритизированный пакет b прецедентов из памяти
        for u = 1...U do
            for i = 1...N do
                # Рассчитать целевые Q-значения для каждого прецедента
                y_i = r_i + δ_{s'_i} γ Q^{π_ϕ}(s'_i, max_{a'_i} Q^{π_θ}(s'_i, a'_i)), где δ_{s'_i} = 0, если
                 ➥ s'_i — конечное состояние, иначе δ_{s'_i} = 1.
                # Рассчитать абсолютную TD-ошибку для каждого прецедента
                ω_i = |y_i − Q^{π_θ}(s_i, a_i)|
            end for
            # Расчет функции потерь, например, с помощью средней
            ➥ квадратической ошибки
            L(θ) = (1/N) Σ_i (y_i − Q^{π_θ}(s_i, a_i))^2
            # Обновление параметров сети
            θ = θ − α∇_θ L(θ)
            # Определение приоритетов для всех прецедентов
            p_i = (|ω_i| + ε)^η / Σ_j (|ω_j| + ε)^η
            Обновление значений приоритетов в памяти прецедентов
        end for
    end for
    Уменьшение τ
    if (m mod F) == 0 then
        # Обновить прогнозную сеть
        ϕ = θ
    end if
end for
```

5.4. Реализация улучшенной DQN

В этом разделе представлена реализация DQN, в которую включены прогнозная сеть и алгоритм двойной DQN. Приоритизированная память прецедентов реализована как новый класс памяти `PrioritizedReplay` добавлением всего нескольких строк кода в `calc_q_loss`.

Изменения в алгоритме DQN реализованы в классе `DQNBase` — потомке `VanillaDQN`. Большая часть кода может быть использована повторно, но нужно изменить `init_nets`, `calc_q_loss` и `update`.

Чтобы сделать различные варианты DQN и связь между ними как можно более понятными, каждой версии придается в соответствие отдельный класс, даже если в код не нужно вносить никаких дополнений. Поэтому DQN с прогнозными сетями реализована в классе `DQN`, расширяющем `DQNBase`. А двойная DQN реализована в `DoubleDQN`, который, в свою очередь, расширяет `DQN`.

5.4.1. Инициализация сети

Имеет смысл вкратце рассмотреть метод `init_nets` из `DQNBase` (листинг 5.1), так как в нем показано, как происходит работа с двумя сетями.

В строках 12 и 13 сети инициализируются. `self.net` — это обучаемая сеть с параметрами θ, а `self.target_net` — прогнозная сеть с параметрами ϕ. Кроме того, есть два других атрибута класса: `self.online_net` и `self.eval_net` (строки 16 и 17). В зависимости от запущенного на выполнение варианта DQN `self.online_net` и `self.eval_net` указывают либо на `self.net`, либо на `self.target_net`. Они могут указывать и на разные сети: один — на `self.net`, а другой — на `self.target_net`. Благодаря такому подходу можно легко переключаться между DQN и двойной DQN с прогнозной сетью, меняя сеть, которая присвоена `self.online_net` и `self.eval_net`. В `DQNBase` они оба указывают на прогнозную сеть.

Листинг 5.1. Реализация улучшенной DQN, инициализация сетей

```
 1 # slm_lab/agent/algorithm/dqn.py
 2
 3 class DQNBase(VanillaDQN):
 4     ...
 5
 6     @lab_api
 7     def init_nets(self, global_nets=None):
 8         ...
 9         in_dim = self.body.state_dim
10         out_dim = net_util.get_out_dim(self.body)
11         NetClass = getattr(net, self.net_spec['type'])
12         self.net = NetClass(self.net_spec, in_dim, out_dim)
13         self.target_net = NetClass(self.net_spec, in_dim, out_dim)
14         self.net_names = ['net', 'target_net']
15         ...
16         self.online_net = self.target_net
17         self.eval_net = self.target_net
```

5.4.2. Расчет Q-функции потерь

В `calc_q_loss` (листинг 5.2) есть два отличия от исходной реализации DQN: расчет $Q^{\pi}(s', a')$ и обновление приоритетов прецедентов в пакете.

Сначала рассчитываем $\hat{Q}^{\pi}(s, a)$ для всех действий a во всех состояниях в пакете (строка 9) и выбираем действие, максимизирующее Q-значение для каждого из состояний s.

Затем рассчитываем $\hat{Q}^{\pi}(s', a')$ для всех действий a' во всех состояниях s' в пакете. Это выполняется дважды с помощью `self.online_net` и `self.eval_net` (строки 11–14). Далее выбираем действие, максимизирующее Q-значение, для каждого

состояния s' с помощью `self.online_net` и сохраняем в `online_actions` (строка 16). Потом применяем `self.eval_net` для выбора Q-значений, соответствующих этим действиям (строка 17). `max_next_q_preds` — это оценка для $\hat{Q}^{\pi}(s', a')$.

Расчет $Q^{\pi}_{\text{tar}}(s, a)$ (строка 18) и `q_loss` (строка 20) такой же, как в исходной реализации DQN.

Для приоритизированной памяти прецедентов дополнительно рассчитываем TD-ошибки для всех прецедентов в пакете (строка 23) и используем эти значения для обновления приоритетов (строка 24).

Листинг 5.2. Реализация улучшенной DQN, расчет Q-функции потерь

```
 1 # slm_lab/agent/algorithm/dqn.py
 2
 3 class DQNBase(VanillaDQN):
 4     ...
 5
 6     def calc_q_loss(self, batch):
 7         states = batch['states']
 8         next_states = batch['next_states']
 9         q_preds = self.net(states)
10         with torch.no_grad():
11             # online_net используется для выбора действий
               ➥ в следующем состоянии
12             online_next_q_preds = self.online_net(next_states)
13             # eval_net применяется для расчета next_q_preds
               ➥ для действий, выбранных online_net
14             next_q_preds = self.eval_net(next_states)
15         act_q_preds = q_preds.gather(-1,
           ➥ batch['actions'].long().unsqueeze(-1)).squeeze(-1)
16         online_actions = online_next_q_preds.argmax(dim=-1, keepdim=True)
17         max_next_q_preds = next_q_preds.gather(-1,
           ➥ online_actions).squeeze(-1)
18         max_q_targets = batch['rewards'] + self.gamma *
           ➥ (1 - batch['dones']) * max_next_q_preds
19         ...
20         q_loss = self.net.loss_fn(act_q_preds, max_q_targets)
21
22         if 'Prioritized' in util.get_class_name(self.body.memory): # PER
23             errors = (max_q_targets -
               ➥ act_q_preds.detach()).abs().cpu().numpy()
24             self.body.memory.update_priorities(errors)
25         return q_loss
```

5.4.3. Обновление прогнозной сети

После каждого шага обучения вызывается метод `update` алгоритма, в который добавлено обновление прогнозной сети, приведенное в листинге 5.3.

Реализация простая. Если применяется обновление замещением, то `self.net` напрямую копируется в `self.target_net` (строки 13 и 14, 21 и 22). Если используется обновление Поляка, то рассчитываем средневзвешенное значение для каждого из параметров `self.net` и текущей `self.target_net`. Затем используем полученные результаты для обновления `self.target_net` (строки 15 и 16, 24–26).

Листинг 5.3. Реализация улучшенной DQN, обновление параметров прогнозной сети

```
 1 # slm_lab/agent/algorithm/dqn.py
 2
 3 class DQNBase(VanillaDQN):
 4     ...
 5
 6     @lab_api
 7     def update(self):
 8         self.update_nets()
 9         return super().update()
10
11     def update_nets(self):
12         if util.frame_mod(self.body.env.clock.frame,
           ➥ self.net.update_frequency, self.body.env.num_envs):
13             if self.net.update_type == 'replace':
14                 net_util.copy(self.net, self.target_net)
15             elif self.net.update_type == 'polyak':
16                 net_util.polyak_update(self.net, self.target_net,
                   ➥ self.net.polyak_coef)
17             ...
18
19 # slm_lab/agent/net/net_util.py
20
21     def copy(src_net, tar_net):
22         tar_net.load_state_dict(src_net.state_dict())
23
24     def polyak_update(src_net, tar_net, old_ratio=0.5):
25         for src_param, tar_param in
           ➥ zip(src_net.parameters(),tar_net.parameters()):
26             tar_param.data.copy_(old_ratio * src_param.data +
               ➥ (1.0 - old_ratio) * tar_param.data)
```

5.4.4. DQN с прогнозными сетями

Для реализации DQN с прогнозными сетями не требуется никакого дополнительного кода, как показано в листинге 5.4.

DQN является потомком DQNBase, а из листинга 5.1 видно, что в нем производится инициализация `self.target_net`, присваиваемой `self.online_net` и `self.eval_net`. Следовательно, в `calc_q_loss` для расчета $Q_{tar}^{\pi}(s,a)$ используется одна сеть `self.target_net`.

Листинг 5.4. Класс DQN

```
1 # slm_lab/agent/algorithm/dqn.py
2
3 class DQN(DQNBase):
4
5     @lab_api
6     def init_nets(self, global_nets=None):
7         super().init_nets(global_nets)
```

5.4.5. Двойная DQN

Из листинга 5.5 видно, что DoubleDQN наследует DQN. При инициализации сетей в `init_nets` переменной `self.online_net` присваивается `self.net`, а переменной `self.eval_net` — `self.target_net` (строки 18 и 19).

Теперь при расчете $Q_{tar}^{\pi}(s, a)$ для выбора действия a' применяется `self.net`, так как `self.online_net` указывает на нее. `self.target_net` оценивает Q-значения для (s', a'), поскольку на нее указывает `self.eval_net`.

Листинг 5.5. Класс DoubleDQN

```
1 # slm_lab/agent/algorithm/dqn.py
2
3 class DoubleDQN(DQN):
4
5     @lab_api
6     def init_nets(self, global_nets=None):
7         super().init_nets(global_nets)
8         self.online_net = self.net
9         self.eval_net = self.target_net
```

5.4.6. Приоритизированная память прецедентов

Приоритизированная память прецедентов изменяет порядок выбора прецедентов из памяти, что повышает эффективность выборки в DQN. В этом разделе поясняются идеи, лежащие в основе приоритизированной памяти прецедентов, обсуждавшейся в разделе 5.3. Как и рассмотрение других классов памяти, его можно пропустить при первом прочтении — это не помешает понять улучшения в DQN.

Для реализации PER нужен новый класс памяти `PrioritizedReplay`. Большая часть его кода — та же, что у класса памяти `Replay`, который он расширяет. Тем не менее требуется добавить три функции: хранение приоритетов, обновление приоритетов и пропорциональную выборку.

- **Хранение приоритетов.** Прецедент должен содержать свой приоритет, и для того, чтобы его отслеживать, нужно создать дополнительный буфер в памяти

прецедентов. Это можно сделать, переопределив функции `__init__`, `reset` и `add_experience` в классе памяти `Replay`.

- **Обновление приоритетов.** На каждом этапе обучения агента приоритеты всех прецедентов могут меняться. Введем новую функцию `update_priorities` для изменения приоритетов прецедентов в памяти и отслеживания индексов в пакетах, которые были выбраны самыми последними, чтобы знать, какой прецедент нужно обновлять.
- **Пропорциональная выборка.** Это самая сложная часть реализации PER. Прецеденты должны выбираться в соответствии с их приоритетами, но, даже если память очень большая, такая выборка все равно должна выполняться быстро, чтобы не замедлять обучение. Следовательно, нужно переопределить `sample_idxs` и создать для хранения приоритетов новую структуру данных `SumTree`, чтобы выборка оставалась эффективной при увеличении размера памяти. Следует отметить, что в памяти `Replay` такой проблемы не возникало, так как все индексы выбирались случайно.

Далее мы сначала обсудим новую структуру данных `SumTree` и поговорим о том, в чем ее польза для реализации PER. Затем еще раз рассмотрим класс памяти `PrioritizedReplay` и то, как он реализует функции, необходимые для PER.

В алгоритме 5.3 было продемонстрировано, как просто посчитать абсолютную TD-ошибку между $Q^{\pi_w}(s,a)$ и $Q^{\pi}_{tar}(s,a)$ для всех прецедентов в пакете. В уравнении (5.8) было показано, как преобразовать эту величину ошибки (обозначена $|\omega|$) в приоритет для каждого из элементов.

Для расчета приоритета одного прецедента требуется следить за изменениями суммы ненормализованных приоритетов $\sum_j \left(|\omega_j| + \varepsilon\right)^{\eta}$ (знаменатель в уравнении (5.8)), которая вычисляется по всем прецедентам в памяти. Получив все приоритеты $P(i)$, можно производить выборку, руководствуясь ими.

Рассмотрим следующий подход к выборке пакета прецедентов.

1. Преобразуем алгоритм 5.3, чтобы следить за изменениями $\sum_j \left(|\omega_j| + \varepsilon\right)^{\eta}$. При обновлении приоритетов сначала рассчитываем изменение $\sum_j \left(|\omega_j| + \varepsilon\right)^{\eta}$, чтобы получить правильное значение.
2. Случайно равномерно выбираем число x из промежутка от 0 до $\sum_j \left(|\omega_j| + \varepsilon\right)^{\eta}$.
3. Выполняем обход всех прецедентов в памяти, вычисляя сумму всех $|\omega_i| + \varepsilon$, полученных до этого момента.
4. Если $\sum_i^k \left(|\omega_i| + \varepsilon\right) \geq x$, прекращаем обход. Индекс k текущего $|\omega_k| + \varepsilon$ соответствует прецеденту, который должен быть выбран из памяти.
5. Повторяем шаги 3 и 4 до тех пор, пока не будет получен полный пакет индексов.
6. Строим пакет прецедентов с помощью индексов, определенных в пункте 5.

В этом методе прецеденты выбираются из памяти согласно их приоритету, поскольку значения x распределены равномерно между 0 и $\sum_j \left(|\omega_j| + \varepsilon\right)^\eta$. Чем больше $|\omega_j| + \varepsilon$, тем больше значений из множества x будет принимать j-й прецедент и тем больше вероятность того, что это будет прецедент, который меняет $\sum_j \left(|\omega_i| + \varepsilon\right) \le x$ на $\sum_j \left(|\omega_i| + \varepsilon\right) \ge x$.

Проблема с этим подходом в том, что он медленный, поскольку для определения индекса k прецедента, который нужно выбрать, требуется последовательный обход всех данных в памяти. Его вычислительная сложность $O(n)$, где n — текущий размер памяти. При выборке пакета прецедентов процесс должен быть повторен N (размер пакета) раз.

Память может содержать миллионы элементов, а во время обучения агента выборка пакетов происходит очень часто. Метод выборки, при котором выполняется перебор всех прецедентов, значительно замедлит обучение: если раньше оно длилось часы или дни, то теперь будут уходить недели.

В качестве альтернативного варианта для хранения приоритетов можно воспользоваться бинарным деревом поиска (sum tree), значения в родительских узлах которого равны суммам значений их дочерних узлов. Эта структура данных позволяет производить выборку прецедентов с вычислительной сложностью $O(\log_2 n)$ вместо $O(n)$, что обсуждается в примечании 5.2. Это значительное улучшение, если объем памяти большой. Например, если в памяти содержится 1 000 000 прецедентов, то для выборки индекса потребуется лишь 20 шагов, так как $\log_2(1\,000\,000) \approx 20$, вместо 1 000 000 при худшем сценарии, когда происходит итерация по всей памяти.

Примечание 5.2. Бинарное дерево, в котором все узлы, кроме листовых, содержат суммы значений своих дочерних узлов

Это бинарное дерево поиска, в листьях которого хранятся приоритеты прецедентов, а все внутренние узлы содержат суммы элементов их дочерних узлов. Следовательно, в корне будет находиться сумма значений $\left(|\omega_j| + \varepsilon\right)^\eta$, а это и есть знаменатель в уравнении (5.8).

Благодаря такой структуре дерева поиска выборка прецедентов согласно их приоритетам упрощается. Сначала случайно равномерно выбираем число x между 0 и $\sum_j \left(|\omega_j| + \varepsilon\right)^\eta$, последнее значение можно получить за постоянное время, запросив элемент в корне дерева. Затем спускаемся по дереву до листа. Решение о том, какой дочерний узел выбрать, правый или левый, принимается следующим образом. Если `x <= node.left_child`, выбираем левый дочерний узел и устанавливаем его в качестве текущего узла. В противном случае присваиваем `x = x - node.left_child` и определяем правый дочерний узел как текущий. Повторяем эту процедуру, пока не достигнем листового элемента, и возвращаем его индекс.

Чтобы понять, почему это дает пропорциональную выборку, рассмотрим пример для нескольких элементов. На рис. 5.1 приведен пример для шести элементов, соответ-

ствующих шести прецедентам. Значения $\left(\left|\omega_j\right|+\varepsilon\right)^\eta$ для всех элементов показаны в листовых узлах внизу дерева и равны (5, 25, 10, 0, 35, 24). $\sum_j\left(\left|\omega_j\right|-\varepsilon\right)^\eta = 99$ находится в корне, вверху дерева.

Если x выбирается случайно равномерно между 0 и 99, то значения x, для которых будут выбраны все элементы, показаны ниже каждого листа вместе с соответствующим количеством выборок каждого элемента, выраженным в процентах. Чем больше $\left(\left|\omega_j\right|+\varepsilon\right)^\eta$, тем чаще выбирается элемент i. Например, пятый элемент с $\left(\left|\omega_j\right|+\varepsilon\right)^\eta = 35$ будет выбираться в 35 % случаев, тогда как третий с $\left(\left|\omega_j\right|+\varepsilon\right)^\eta = 10$ — лишь в 10 % случаев.

На рис. 5.1 показан также спуск по дереву при $x = 37$. Из корня переходим в левый дочерний узел, так как его значение 40 больше, чем x ($x \le 40$). Далее $x > 30$, поэтому переходим в правый дочерний узел со значением 10 и устанавливаем $x - 30 = 7$. Наконец, сравниваем 7 и 10 и, поскольку $7 \le 10$, выбираем левый дочерний элемент и достигаем листа. Индекс этого листа 3, его мы в итоге и выбираем.

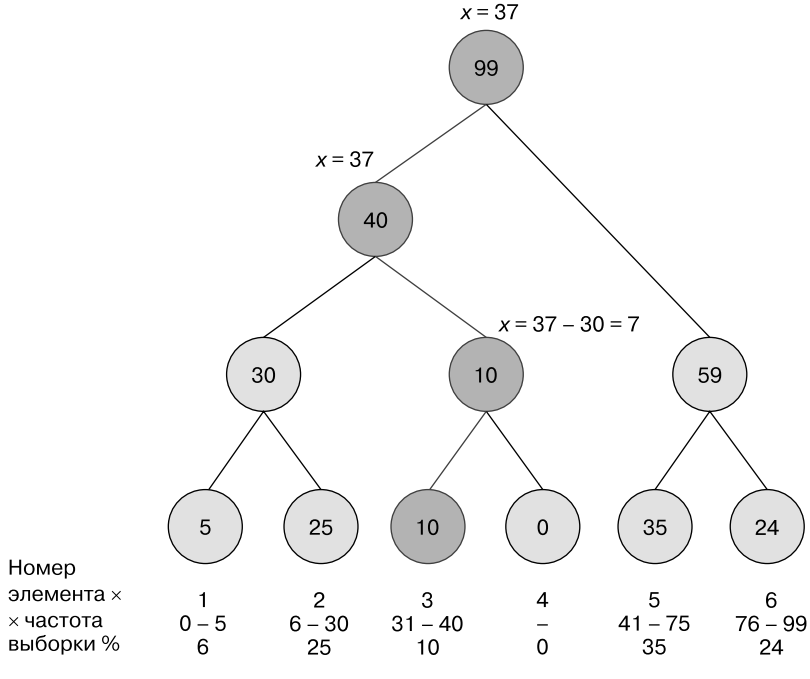


Рис. 5.1. Бинарное дерево из шести элементов, в котором все узлы, кроме листовых, содержат суммы значений своих дочерних узлов

Теперь количество шагов при выборе индекса прецедента, включаемого в пакет, равно высоте бинарного дерева, то есть $\log_2 n$. Например, если в памяти находится 100 000 прецедентов, то выборка индекса произойдет за 17 шагов, так как $\log_2 (100\,000) \approx 17$.

Инициализация памяти и установка начальных значений. В листинге 5.6 `__init__` вызывает соответствующий метод из базового класса (строка 13), который инициализирует переменные класса, включая ключи хранилища. В памяти `PrioritizedReplay` также нужно хранить приоритеты и инициализировать бинарное дерево. Это производится путем переопределения ключей хранилища `self.data_keys` с добавлением еще одного элемента для приоритетов (строка 18) и последующего повторного вызова `self.reset` (строка 19).

`reset` вызывает аналогичный метод базового класса и дополнительно инициализирует дерево поиска (строка 23).

Листинг 5.6. Приоритизированная память прецедентов, инициализация и установка первоначальных значений

```
 1 # slm_lab/agent/memory/prioritized.py
 2
 3 class PrioritizedReplay(Replay):
 4
 5     def __init__(self, memory_spec, body):
 6         util.set_attr(self, memory_spec, [
 7             'alpha',
 8             'epsilon',
 9             batch_size',
10             'max_size',
11             'use_cer',
12         ])
13         super().__init__(memory_spec, body)
14
15         self.epsilon = np.full((1,), self.epsilon)
16         self.alpha = np.full((1,), self.alpha)
17         # добавление скалярной величины
           ➥ для приоритетов 'priorities' в ключи data_keys
           ➥ и повторный вызов reset
18         self.data_keys = ['states', 'actions', 'rewards', 'next_states',
           ➥ 'dones', 'priorities']
19         self.reset()
20
21     def reset(self):
221        super().reset()
23         self.tree = SumTree(self.max_size)
```

Хранение приоритетов. В листинге 5.7 `add_experience` сначала вызывает метод базового класса для добавления в память кортежей (`state`, `action`, `reward`, `next_state`, `done` (состояние, действие, вознаграждение, следующее состояние, флаг выполнения)) (строка 7).

Затем нужно получить приоритеты прецедентов для заданных абсолютных TD-ошибок `error` (строка 8). Здесь используется одна эвристика: новые прецеденты, скорее всего, будут более информативными для агента, поэтому вероятность их

выборки должна быть высокой. Это реализовано как присвоение им большого значения ошибки 100 000.

В конце добавляем приоритеты как в память, так и в бинарное дерево (строки 9 и 10).

Листинг 5.7. Приоритизированная память прецедентов, хранение приоритетов

```
 1 # slm_lab/agent/memory/prioritized.py
 2
 3 class PrioritizedReplay(Replay):
 4     ...
 5
 6     def add_experience(self, state, action, reward, next_state, done,
       ➥ error=100000):
 7         super().add_experience(state, action, reward, next_state, done)
 8         priority = self.get_priority(error)
 9         self.priorities[self.head] = priority
10         self.tree.add(priority, self.head)
11
12     def get_priority(self, error):
13         return np.power(error + self.epsilon, self.alpha).squeeze()
```

Обновление приоритетов. В листинге 5.8 приведено обновление приоритетов. Благодаря реализации `SumTree` это происходит просто. Сначала абсолютные TD-ошибки преобразуются в приоритеты (строки 7). Затем с помощью индексов прецедентов в пакете обновляются приоритеты в основной структуре памяти (строки 9 и 10). Обратите внимание: в `self.batch_idxs` всегда хранятся индексы из последних выбранных пакетов. Наконец, обновляются приоритеты в `SumTree` (строки 11 и 12).

Листинг 5.8. Приоритизированная память прецедентов, обновление приоритетов

```
 1 # slm_lab/agent/memory/prioritized.py
 2
 3 class PrioritizedReplay(Replay):
 4     ...
 5
 6     def update_priorities(self, errors):
 7         priorities = self.get_priority(errors)
 8         assert len(priorities) == self.batch_idxs.size
 9         for idx, p in zip(self.batch_idxs, priorities):
10             self.priorities[idx] = p
11         for p, i in zip(priorities, self.tree_idxs):
12             self.tree.update(i, p)
```

Пропорциональная выборка. `sample_idxs` (листинг 5.9) отвечает за определение индексов прецедентов, из которых должен состоять пакет. Для выбора индекса сначала берем число между 0 и $\sum_j \left(\left|\omega_j\right| + \varepsilon\right)^{\eta}$ (строка 11). Это число используется для выбора элемента из дерева с помощью процедуры, описанной в примечании 5.2

(строка 12), а этому элементу соответствует индекс в памяти `PrioritizedReplay`. После выбора всех индексов они сохраняются в `self.batch_idxs` для применения при построении набора тренировочных данных (строка 13) и в `self.tree_idxs` — для использования при обновлении приоритетов (строка 14).

Следует отметить, что данная реализация пропорциональной выборки была позаимствована из блога Джерома Дженишa (https://jaromiru.com/2016/09/27/lets-make-a-dqn-theory/). Реализацию дерева поиска ищите в `slm_lab/agent/memory/prioritized.py`.

Листинг 5.9. Приоритизированная память прецедентов, пропорциональная выборка

```
 1 # slm_lab/agent/memory/prioritized.py
 2
 3 class PrioritizedReplay(Replay):
 4     ...
 5
 6     def sample_idxs(self, batch_size):
 7         batch_idxs = np.zeros(batch_size)
 8         tree_idxs = np.zeros(batch_size, dtype=np.int)
 9
10         for i in range(batch_size):
11             s = random.uniform(0, self.tree.total())
12             (tree_idx, p, idx) = self.tree.get(s)
13             batch_idxs[i] = idx
14             tree_idxs[i] = tree_idx
15
16         batch_idxs = np.asarray(batch_idxs).astype(int)
17         self.tree_idxs = tree_idxs
18         ...
19         return batch_idxs
```

5.5. Обучение агента DQN играм Atari

На данный момент мы располагаем всеми необходимыми элементами для обучения агентов DQN играм Atari по состояниям в виде изображений. Однако для повышения производительности необходимо преобразовать состояния и вознаграждения среды. Эти преобразования были впервые введены в известной статье *Human-Level Control through Deep Reinforcement Learning* [89] и с тех пор стали распространенной практикой для сред Atari.

В этом разделе сначала приводятся краткие сведения об играх Atari, а также преобразования для их состояний и вознаграждений, затем — конфигурация файла `spec` для агента двойной DQN с PER для игры в Pong из Atari.

Atari 2600 — бывшая ранее популярной игровая консоль (выпущена в 1977 году), на которую наряду с оригинальными играми было перенесено большое количество ставших теперь классическими аркадных игр. Доступные на Atari 2600 игры часто были

сложными, и в них было трудно играть. Тем не менее их вычислительные требования были низкими, что позволило легко эмулировать их выполнение на современных компьютерах. Емкость ОЗУ консоли — 128 байт, а игровой экран составляет лишь 160 пикселов в ширину и 210 пикселов в высоту. Беллемар с коллегами в 2012 году поняли, что благодаря этим условиям игры Atari являются идеальной тестовой платформой для алгоритмов обучения с подкреплением. Они создали среду обучения Arcade Learning Environment (ALE), в которой эмулированы более 50 игр [14].

В SLM Lab используются игры Atari, предоставляемые через OpenAI Gym [18]. Во всех играх Atari состояние — это изображение игрового экрана в формате RGB с низким разрешением, закодированное как трехмерный массив размером (210, 160, 3) (рис. 5.2). Пространство состояний дискретно и имеет несколько измерений. В зависимости от игры агент на каждом временном шаге может предпринять от 4 до 18 различных действий. Например, в OpenAI Gym действия в Pong — это 0 (бездействие), 1 (подача), 2 (вверх) и 3 (вниз).

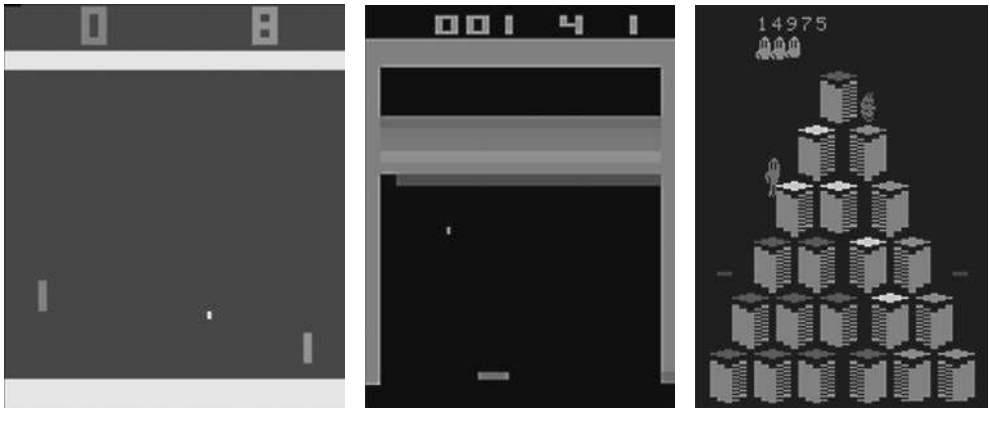


Рис. 5.2. Примеры состояний из трех игр Atari, предоставляемых через OpenAI Gym [18]

Размерность пространства состояний в Atari значительно выше, чем в любой из виденных нами до сих пор игр. В каждом состоянии $210 \cdot 160 \cdot 3 = 100\,800$ измерений по сравнению с четырьмя измерениями в CartPole. Кроме того, игры Atari намного сложнее, чем CartPole. Длительность эпизодов составляет тысячи шагов, и для достижения хорошей производительности требуются сложные последовательности действий. В совокупности эти два фактора значительно усложняют задачу обучения агента. Чтобы помочь ему обучаться в таких условиях, авторы [88, 141] произвели следующие изменения в алгоритмах стандартной DQN и двойной DQN.

- **Специальная структура сети для обработки изображений.** Для аппроксимации Q-функции используется сверточная нейронная сеть с тремя скрытыми сверточными слоями и одним скрытым полносвязным слоем.

- **Предварительная обработка состояний.** Включает уменьшение размера изображения, преобразование цветов в градации серого, объединение кадров и пропуск кадров с максимальным значением пикселов.
- **Предварительная обработка вознаграждений.** На каждом временном шаге вознаграждение преобразуется в значение –1, 0, +1 в зависимости от знака первоначального вознаграждения.
- **Установка начальных значений среды.** В зависимости от игры при потере жизни происходит задание переменным среды исходных значений, начальному состоянию — случайного значения, и для перезагрузки может быть выбрано FIRE.

Структура сети реализована с помощью класса `ConvNet` из SLM Lab. Преобразования состояний, вознаграждений и среды более подробно обсуждаются в разделе 10.3. Эти изменения обрабатываются с помощью обертки вокруг среды OpenAI Gym. Она не меняет интерфейс среды и дает возможность произвести все желаемые преобразования незаметно. Это позволяет обучать агента DQN или двойной DQN без каких-либо изменений в коде рассмотренных ранее реализаций.

В листинге 5.10 приведена конфигурация для обучения игре Pong из Atari агента двойной DQN с PER. Файл имеется в SLM Lab в `slm_lab/spec/benchmark/dqn/ddqn_per_pong_spec.json`.

Листинг 5.10. Файл spec для двойной DQN с PER с настройками для игры Pong из Atari

```
# slm_lab/spec/benchmark/dqn/ddqn_per_pong.json

{
    "ddqn_per_pong": {
        "agent": [{
            "name": "DoubleDQN",
            "algorithm": {
                "name": "DoubleDQN",
                "action_pdtype": "Argmax",
                "action_policy": "epsilon_greedy",
                "explore_var_spec": {
                    "name": "linear_decay",
                    "start_val": 1.0,
                    "end_val": 0.01,
                    "start_step": 10000,
                    "end_step": 1000000
                },
                "gamma": 0.99,
                "training_batch_iter": 1,
                "training_iter": 4,
                "training_frequency": 4,
                "training_start_step": 10000
            },
            "memory": {
                "name": "PrioritizedReplay",
                "alpha": 0.6,
```

```
27                 "epsilon": 0.0001,
28                 "batch_size": 32,
29                 "max_size": 200000,
30                 "use_cer": false,
31             },
32             "net": {
33                 "type": "ConvNet",
34                 "conv_hid_layers": [
35                     [32, 8, 4, 0, 1],
36                     [64, 4, 2, 0, 1],
37                     [64, 3, 1, 0, 1]
38                 ],
39                 "fc_hid_layers": [256],
40                 "hid_layers_activation": "relu",
41                 "init_fn": null,
42                 "batch_norm": false,
43                 "clip_grad_val": 10.0,
44                 "loss_spec": {
45                     "name": "SmoothL1Loss"
46                 },
47                 "optim_spec": {
48                     "name": "Adam",
49                     "lr": 2.5e-5,
50                 },
51                 "lr_scheduler_spec": null,
52                 "update_type": "replace",
53                 "update_frequency": 1000,
54                 "gpu": true
55             }
56         }],
57         "env": [{
58             "name": "PongNoFrameskip-v4",
59             "frame_op": "concat",
60             "frame_op_len": 4,
61             "reward_scale": "sign",
62             "num_envs": 16,
63             "max_t": null,
64             "max_frame": 4e6
65         }],
66         "body": {
67             "product": "outer",
68             "num": 1
69         },
70         "meta": {
71             "distributed": false,
72             "eval_frequency": 10000,
73             "log_frequency": 10000,
74             "max_session": 4,
75             "max_trial": 1
76         }
77     }
78 }
```

Сделаем обзор основных компонентов.

- **Алгоритм** — это двойная DQN (строка 8). Чтобы использовать DQN, замените в строке 8 `"name": "DoubleDQN"` на `"name": "DQN"`. Политика выбора действий ε-жадная (строка 10) с линейной скоростью уменьшения: между 10 000 и 1 000 000 шагов значение ε снижается с начального 1,00 до 0,01 (строки 12–16).
- **Архитектура сети** — сверточная нейронная сеть (строка 33) с функциями активации ReLU (строка 40). В сети есть три сверточных слоя (строки 34–38), за которыми следует один полносвязный слой (строка 39). Обучение производится на графическом процессоре (строка 54).
- **Оптимизатор** — это Adam [68] (строки 47–50), а функция потерь — функция потерь Хьюбера, носящая в PyTorch название `SmoothL1Loss` (строки 44–46). Она является квадратичной для абсолютных значений меньше 1 и линейной в остальных случаях, что делает ее менее чувствительной к резко выделяющимся значениям.
- **Частота обучения** — обучение начинается после того, как агент сделает 10 000 шагов в среде (строка 22), и в дальнейшем происходит каждые четыре шага (строка 21). На каждом шаге обучения производится выборка четырех пакетов прецедентов из памяти (строка 20). Каждый пакет содержит 32 элемента (строка 28) и используется для одного обновления параметров сети (строка 19).
- **Память** — `PrioritzedReplay` (строка 25), которая может содержать максимум 200 000 прецедентов (строка 29). Параметр приоритизации η устанавливается с помощью переменной `alpha` (строка 26), а небольшое постоянное значение ε — с помощью `epsilon` (строка 27).
- **Среда** — игра Pong из Atari (строка 58). Для получения одного состояния объединяются четыре кадра (строки 59 и 60), в процессе обучения вознаграждение на каждом временном шаге преобразуется согласно его знаку в –1, 0 или +1 (строка 61). Для ускорения обучения параллельно используются 16 сред (строка 62), этот простой метод обсуждается в главе 8.
- **Длительность обучения** — 4 000 000 шагов (строка 64).
- **Контрольные точки** — агент оценивается каждые 10 000 шагов (строка 72). Параметры сети также проверяются в контрольных точках после каждой оценки.

Чтобы обучить агента DQN с помощью SLM Lab, запустите в терминале команды, приведенные в листинге 5.11.

Листинг 5.11. Обучение агента двойной DQN с PER игре Pong из Atari

```
1 conda activate lab
2 python run_lab.py slm_lab/spec/benchmark/dqn/ddqn_per_pong.json
  ➥ ddqn_per_pong train
```

Файл `spec` используется для запуска обучения `Trial` с четырьмя сессиями `Session`, чтобы получить усредненные результаты, которые затем выводятся на графике с доверительным интервалом. Также генерируется график со скользящим средним по 100 оценкам. Оба графика приведены на рис. 5.3. В начале обучения агент будет иметь средний счет –21. Производительность стабильно повышается в течение 2 млн кадров, после чего агент в среднем достигает близкого к максимальному счета 21.

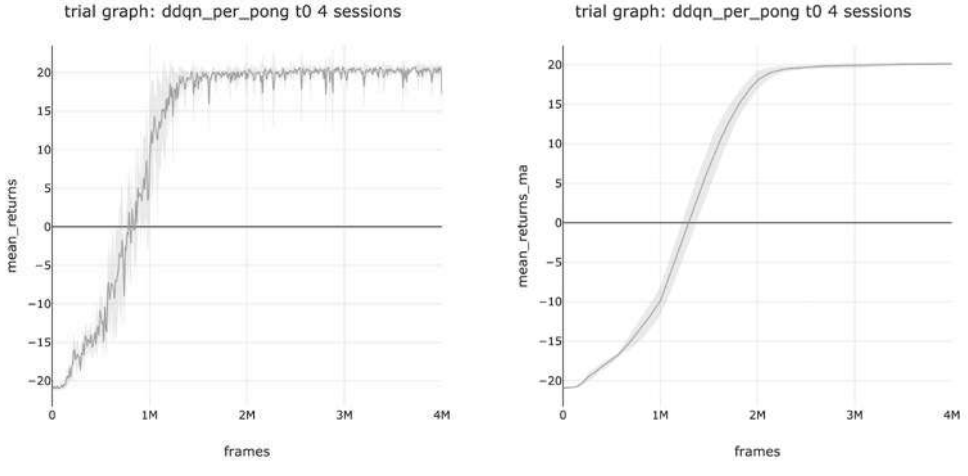


Рис. 5.3. Графики усредненных по четырем сессиям результатов испытания двойной DQN с PER из SLM Lab. По вертикальной оси отложены полные вознаграждения (при оценке mean_return вычисляется без дисконтирования) в контрольных точках, усредненные по восьми эпизодам, по горизонтальной оси — все кадры обучения. График справа — это скользящее среднее по оценкам в 100 контрольных точках

Нужно отметить, что для обучения этого агента требуется больше вычислительных ресурсов, чем для описанных ранее в книге агентов. Пробное обучение должно продолжаться около одного дня при запуске на графическом процессоре.

5.6. Результаты экспериментов

В этом разделе будет рассмотрен комплексный эффект от улучшений DQN, а именно прогнозной сети, двойной DQN и PER. Начиная с DQN с прогнозной сетью (которую будем называть просто DQN), мы будем запускать ряд испытаний, чтобы проверить влияние данных улучшений. В испытания входят четыре сочетания: DQN, DQN + PER, двойная DQN и двойная DQN с PER. Запуск испытаний производится в среде Pong из Atari. Для сравнения их результаты выводятся на общем графике.

5.6.1. Эксперимент по оценке применения двойной DQN с PER

Во-первых, есть файл `spec` для DQN, приведенный в листинге 5.12. Он аналогичен коду из листинга 5.10, за исключением DQN (строки 4–8), с отключением PER путем использования простого класса памяти прецедентов (строки 11–16) и с увеличением скорости обучения (строка 21). Этот файл есть в SLM Lab в `slm_lab/spec/benchmark/dqn/dqn_pong_spec.json`.

Листинг 5.12. Файл spec для DQN с настройками для игры Pong из Atari

```
 1 # slm_lab/spec/benchmark/dqn/dqn_pong.json
 2
 3 {
 4     "dqn_pong": {
 5         "agent": [{
 6         "name": "DQN",
 7         "algorithm": {
 8             "name": "DQN",
 9             ...
10         },
11         "memory": {
12             "name": "Replay",
13             "batch_size": 32,
14             "max_size": 200000,
15             "use_cer": false
16         },
17         "net": {
18             ...
19             "optim_spec": {
20             "name": "Adam",
21             "lr": 1e-4,
22         },
23         ...
24     }
25 }
```

Во-вторых, есть файл `spec` для DQN с PER, приведенный в листинге 5.13, также аналогичный листингу 5.10, но с изменениями для применения DQN (строки 4–8). Этот файл имеется в SLM Lab в `slm_lab/spec/benchmark/dqn/dqn_per_pong_spec.json`.

Листинг 5.13. Файл spec для двойной DQN с PER с настройками для игры Pong из Atari

```
 1 # slm_lab/spec/benchmark/dqn/dqn_per_pong.json
 2
 3 {
 4     "dqn_per_pong": {
 5         "agent": [{
 6             "name": "DQN",
```

```
 7             "algorithm": {
 8                 "name": "DQN",
 9                 ...
10     }
11 }
```

В-третьих, есть файл `spec` для двойной DQN, приведенный в листинге 5.14, с внесением изменений в код из листинга 5.10 для отключения PER (строки 11–16) и повышения скорости обучения (строка 21). Этот файл находится в SLM Lab в `slm_lab/spec/benchmark/dqn/ddqn_pong_spec.json`.

Листинг 5.14. Файл spec для двойной DQN с настройками для игры Pong из Atari

```
 1 # slm_lab/spec/benchmark/dqn/ddqn_pong.json
 2
 3 {
 4 "ddqn_pong": {
 5     "agent": [{
 6         "name": "DoubleDQN",
 7         "algorithm": {
 8             "name": "DoubleDQN",
 9             ...
10         },
11         "memory": {
12             "name": "Replay",
13             "batch_size": 32,
14             "max_size": 200000,
15             "use_cer": false,
16         },
17         "net": {
18             ...
19             "optim_spec": {
20                 "name": "Adam",
21                 "lr": 1e-4,
22             },
23     ...
24     }
25 }
```

Наконец, есть файл `spec` для двойной DQN с PER, показанный ранее в листинге 5.10, также находящийся в SLM Lab в `slm_lab/spec/benchmark/dqn/ddqn_per_pong_spec.json`. Обратите внимание на то, что при использовании PER скорость обучения, как правило, меньше. Это обусловлено тем, что PER чаще выбирает переходы с большим значением ошибки, что в среднем дает большие градиенты. Шоул с коллегами установили, что для компенсации большого значения градиентов целесообразно снизить скорость обучения в четыре раза.

Четыре команды для обучения этих четырех версий DQN с помощью SLM Lab показаны в листинге 5.15. Для завершения всех испытаний потребуется почти целый день при запуске на графическом процессоре. Впрочем, если вычислительных ресурсов достаточно, они могут быть запущены параллельно.

Листинг 5.15. Обучение игре Pong из Atari четырех вариантов алгоритмов: DQN, DQN с PER, двойной DQN и двойной DQN с PER

```
# запуск DQN
conda activate lab
python run_lab.py slm_lab/spec/benchmark/dqn/dqn_pong.json dqn_pong train

# запуск DQN с PER
conda activate lab
python run_lab.py slm_lab/spec/benchmark/dqn/dqn_per_pong.json
  ➥ dqn_per_pong train

# запуск двойной DQN
conda activate lab
python run_lab.py slm_lab/spec/benchmark/dqn/ddqn_pong.json ddqn_pong train

# запуск двойной DQN с PER
conda activate lab
python run_lab.py slm_lab/spec/benchmark/dqn/ddqn_per_pong.json
  ➥ ddqn_per_pong train
```

Каждое из четырех испытаний `Trial` породит собственный график, построенный на основе усредненных данных для четырех сессий `Session`. Для сравнения они выводятся вместе с помощью вспомогательного метода `viz.plot_multi_trial` из SLM Lab как график по результатам множества испытаний (рис. 5.4).

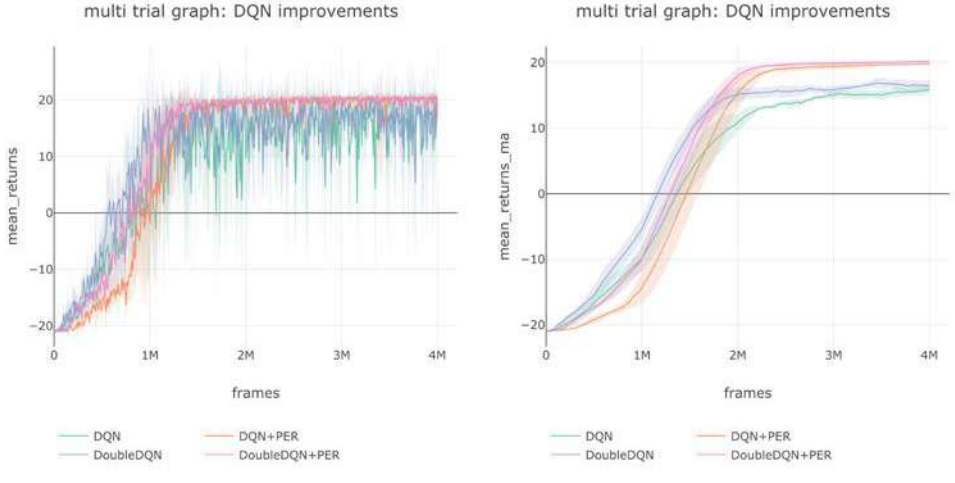


Рис. 5.4. На графиках сравнивается производительность четырех вариантов улучшения DQN на игре Pong из Atari. Ожидаемо наилучшей производительности достигла двойная DQN с PER, следом идут DQN с PER, двойная DQN и DQN

На рис. 5.4 сравнивается производительность четырех вариантов улучшения DQN на игре Pong из Atari. Как и ожидалось, при применении всех усовершенствова-

ний двойная DQN с PER достигает наилучшей производительности. К ней близка DQN с PER, за ними следуют двойная DQN и DQN. В целом, использование PER дает существенное улучшение и повышает устойчивость обучения, что видно по более высоким и гладким кривым для вознаграждений в процессе обучения. Для сравнения: применение двойной DQN приводит к менее значимому улучшению.

5.7. Резюме

В этой главе обсуждалось три метода усовершенствования алгоритма DQN, каждый из которых предназначен для преодоления одного из его недостатков.

- **Прогнозная сеть**, использующая отложенную копию изначальной сети для расчета $Q^{\pi}_{\text{tar}}(s, a)$, упрощает задачу оптимизации и позволяет сделать обучение более устойчивым.
- **Двойная DQN** с применением двух разных сетей для оценки *Q*-значения следующего состояния при расчете $Q^{\pi}_{\text{tar}}(s, a)$ уменьшает тенденцию DQN к завышению оценок *Q*-значений. На практике в качестве двух этих сетей используются обучаемая сеть θ и прогнозная сеть ϕ.
- **Приоритизированная память прецедентов** основана на том, что не все прецеденты одинаково информативны для агента. Задание более высокого приоритета прецедентам, с помощью которых агент может получить больше опыта, что измеряется абсолютной TD-ошибкой между $\hat{Q}^{\pi}(s, a)$ и $Q^{\pi}_{\text{tar}}(s, a)$, повышает эффективность выборки в DQN.

Также была рассмотрена особая реализация DQN с прогнозной сетью и PER, которая спроектирована для игр Atari. Эти игры являются наиболее сложными средами из виденных нами до сих пор. Для достижения высокой производительности нужно было преобразовать состояния и вознаграждения. Преобразования включали в себя понижение размерности пространства состояний посредством уменьшения размеров изображений, их обрезки и преобразования цветов в градации серого. Также применялось совмещение четырех последних состояний, позволяющее агенту видеть недавнее прошлое при выборе действия. Кроме того, увеличивалось время между последующими кадрами путем показа агенту каждого четвертого кадра. И наконец, для *Q*-функции была спроектирована сеть, специально предназначенная для обработки изображений.

5.8. Рекомендуемая литература

- *Mnih V., Kavukcuoglu K., Silver D., Rusu A. A., Veness J., Bellemare M. G., Graves A. et al.* Human-Level Control through Deep Reinforcement Learning. 2015 [89].
- *Hasselt H. van* Double Q-Learning. 2010 [140].

- *Hasselt H. van, Guez A., Silver D.* Deep Reinforcement Learning with Double Q-Learning. 2015 [141].
- *Schaul T., Quan J., Antonoglou I., Silver D.* Prioritized Experience Replay. 2015 [121].
- *Wang Z., Schaul T., Hessel M., Hasselt H. van, Lanctot M., Freitas N. D.* Dueling Network Architectures for Deep Reinforcement Learning. 2015 [144].
- *Bellemare M. G., Naddaf Y., Veness J., Bowling M.* The Arcade Learning Environment: An Evaluation Platform for General Agents. June 2013 [14].

Часть II

Комбинированные методы

6 Метод актора-критика с преимуществом (A2C)

В данной главе будут рассмотрены алгоритмы актора-критика (actor-critic), элегантно объединяющие в себе идеи, с которыми мы познакомились в этой книге, а именно градиент стратегии и настроенную функцию полезностей. В этих алгоритмах стратегия подкрепляется с помощью *настроенного подкрепляющего сигнала*, порожденного с помощью настроенной функции полезностей. Это противоположно REINFORCE, который для подкрепления стратегии использует высокодисперсную оценку отдачи по методу Монте-Карло.

Во всех алгоритмах актора-критика присутствует два совместно обучаемых компонента: *актор* (actor), который настраивает параметризованную стратегию, и *критик* (critic), настраивающий функцию полезности для оценки пар «состояние — действие». Критик предоставляет актору подкрепляющий сигнал.

В основе этих алгоритмов лежит соображение о том, что настроенный подкрепляющий сигнал может быть более информативным для стратегии, чем вознаграждения, доступные из среды. Например, разреженное вознаграждение, при котором агент получает +1 при успехе, может быть преобразовано в плотный подкрепляющий сигнал. Более того, настроенные функции полезности, как правило, имеют меньшую дисперсию, чем оценки отдачи по методу Монте-Карло. Это снижает неопределенность при настройке стратегии [11], упрощая процесс обучения. Однако он становится сложнее. Теперь обучение стратегии зависит от качества оценки функции полезности, которая настраивается параллельно. Пока функция полезности не начнет порождать адекватные сигналы для стратегии, учиться выбирать хорошие действия будет сложно.

В этих методах общепринятой является настройка функции *преимущества* $A^{\pi}(s, a) = Q^{\pi}(s, a) - V^{\pi}(s)$ для генерации подкрепляющих сигналов. Основной смысл — лучше выбрать действие, исходя из его эффективности по отношению к другим действиям, доступным в отдельно взятом состоянии, чем использовать абсолютное значение полезности этого действия, измеряемое Q-функцией. Преимущество количественно определяет, насколько какое-либо действие лучше или хуже, чем среднее из доступных действий. Алгоритмы актора-критика, настраивающие функцию преимущества, известны как алгоритмы актора-критика с преимуществом (Advantage Actor-Critic, A2C).

Сначала в разделе 6.1 вкратце обсуждается актор, который аналогичен REINFORCE. Затем в разделе 6.2 вводятся критик и два разных метода оценки функции преимущества: выгода за n шагов и обобщенная оценка преимущества (Generalized Advantage Estimation) [123].

В разделе 6.3 приведен алгоритм актора-критика, а в разделе 6.4 — пример его реализации. В конце главы даны инструкции по обучению агента методом актора-критика.

6.1. Актор

Актор настраивает параметризированные стратегии π_θ с помощью градиента стратегии, как показано в уравнении (6.1). Это очень похоже на REINFORCE (см. главу 2), за исключением того, что теперь в качестве подкрепляющего сигнала вместо оценки отдачи $R_t(\tau)$ по методу Монте-Карло (см. уравнение (2.1)) применяется преимущество A_t^π:

- актор-критик:

$$\nabla_{\theta_\pi} J(\pi_\theta) = \mathbb{E}_t\left[A_t^\pi \nabla_{\theta_\pi} \log \pi_{\theta_\pi}(a_t \mid s_t)\right]; \tag{6.1}$$

- REINFORCE:

$$\nabla_{\theta_\pi} J(\pi_\theta) = \mathbb{E}_t\left[R_t(\tau) \nabla_{\theta_\pi} \log \pi_{\theta_\pi}(a_t \mid s_t)\right]. \tag{6.2}$$

Следующим шагом будет настройка функции преимущества.

6.2. Критик

Критики отвечают за обучение тому, как оценивать пары (s, a), и за применение этих навыков для порождения A^π.

Сначала приводится описание функции преимущества и рассказывается, почему она является хорошим выбором для подкрепляющего сигнала. Затем представлены два метода для оценки функции преимущества: выгода за n шагов и обобщенная оценка преимущества [123]. В завершение обсуждается их настройка на практике.

6.2.1. Функция преимущества

Интуитивно понятно, что функция преимущества $A^\pi(s_t, a_t)$ является мерой того, насколько какое-то действие лучше или хуже, чем среднее действие по стратегии в конкретном состоянии. Преимущество определяется из уравнения (6.3):

$$A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t). \tag{6.3}$$

У этого уравнения есть несколько замечательных свойств. В первую очередь это $\mathbb{E}_{a\in A}\left[A^{\pi}(s_t, a)\right] = 0$. Если все действия по сути равноценны, то для них A^{π} имеет нулевое значение и вероятность их выбора не изменяется при настройке стратегии с помощью A^{π}. Сравните это с подкрепляющим сигналом, основанным на абсолютной полезности состояния или пары «состояние — действие». В аналогичной ситуации этот сигнал имел бы постоянное значение, но оно могло бы быть отличным от нуля. Следовательно, выбранное действие будет активно поощряться (если значение положительное) или не поощряться (при отрицательном значении). Поскольку все действия равноценны, то на практике могут возникнуть проблемы, кроме того, это непонятно на интуитивном уровне.

Можно привести более сложный пример, когда выбранное действие хуже среднего, но ожидаемая отдача все еще имеет положительное значение. То есть $Q^{\pi}(s_t, a_t) > 0$, но $A^{\pi}(s_t, a_t) < 0$. В идеале вероятность выбранного действия должна снизиться, поскольку доступны лучшие варианты. В этом случае применение A^{π} дает более подходящее в нашем понимании поведение, так как выбранное действие не будет поощряться. А применение Q^{π} или даже Q^{π} с базовым значением может стимулировать это действие.

Преимущество также является относительной величиной. Для определенного состояния s и конкретного действия a рассматривается полезность $Q^{\pi}(s, a)$ пары «состояние — действие» и относительно $V^{\pi}(s)$ оценивается, в лучшую или худшую сторону изменится стратегия при выборе действия a. Преимущество избегает выбраковки действия за то, что стратегия на текущий момент привела к крайне плохому состоянию. И наоборот, действие не поощряется за пребывание в хорошем состоянии. Это предпочтительно по той причине, что a может влиять лишь на будущую траекторию, но не на то, как стратегия достигла текущего состояния. Действие следует оценивать, исходя из того, как оно изменяет полезность в будущем.

Рассмотрим пример. В уравнении (6.4) стратегия привела к хорошему состоянию $V^{\pi}(s) = 100$, тогда как в уравнении (6.5) состояние плохое с $V^{\pi}(s) = -100$. В обоих случаях действие дает относительное улучшение на 10, что отражено в одинаковом значении преимущества. Однако это может быть неочевидно, если смотреть только на $Q^{\pi}(s, a)$:

$$Q^{\pi}(s, a) = 110,\ V^{\pi}(s) = 100,\ A^{\pi}(s, a) = 10; \tag{6.4}$$

$$Q^{\pi}(s, a) = -90,\ V^{\pi}(s) = -100,\ A^{\pi}(s, a) = 10. \tag{6.5}$$

Подобная трактовка функции преимущества позволяет уловить долгосрочное влияние действия, поскольку она рассматривает все будущие шаги[1], не принимая во внимание эффект от всех действий, произведенных до текущего момента. Шульман

[1] В рамках временного горизонта, явно заданного γ.

с коллегами представили аналогичную интерпретацию в своей статье *Generalized Advantage Estimation* [123].

Узнав, почему функция преимущества $A^{\pi}(s, a)$ — хороший выбор для подкрепляющего сигнала в алгоритме актора-критика, рассмотрим теперь способы ее оценки.

Оценка преимущества по отдаче за *n* шагов

Для расчета A^{π} нужны оценки для Q^{π} и V^{π}. Один из подходов — можно настраивать Q^{π} и V^{π} по отдельности с помощью разных нейронных сетей. Но у него есть два недостатка. Во-первых, нужно позаботиться о том, чтобы обе эти оценки были согласованы. Во-вторых, это менее эффективно с точки зрения обучения. Вместо этого мы обычно настраиваем только V^{π} и для оценки Q^{π} объединяем ее с вознаграждениями из траектории.

Настраивать V^{π} предпочтительнее, чем Q^{π}, по двум причинам. Во-первых, Q^{π} — более сложная функция и для настройки хорошей оценки может потребоваться больше примеров. Это может быть крайне проблематичным, когда актор и критик тренируются совместно. Во-вторых, получение оценки V^{π} из Q^{π} может быть более затратным в вычислительном отношении. Выведение оценки $V^{\pi}(s)$ из $Q^{\pi}(s, a)$ требует вычисления полезностей для всех возможных действий в состоянии *s*, а затем получения средневзвешенного значения вероятностей действий для $V^{\pi}(s)$. Кроме того, это затруднительно в случае сред с непрерывными действиями, поскольку для оценки V^{π} потребовалась бы репрезентативная выборка действий из непрерывного пространства.

Рассмотрим получение оценки Q^{π} из V^{π}.

Предположим, что у нас есть идеальная оценка $V^{\pi}(s)$, тогда *Q*-функция может быть записана как множество ожидаемых вознаграждений для *n* шагов, за которым следует $V^{\pi}(s_{n+1})$ (уравнение (6.6)). Чтобы упростить оценку, будем использовать одну траекторию вознаграждений ($r_1 \ldots r_n$) вместо математического ожидания и подставим настроенную критиком $\hat{V}^{\pi}(s)$. Выражение из уравнения (6.7) известно как будущая отдача за *n* шагов:

$$Q^{\pi}(s_t, a_t) = \mathbb{E}_{\tau \sim \pi}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots + \gamma^n r_{t+n}\right] + \gamma^{n+1} V^{\pi}(s_{t+n+1}) \approx \tag{6.6}$$

$$\approx r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots + \gamma^n r_{t+n} + \gamma^{n+1} \hat{V}^{\pi}(s_{t+n+1}). \tag{6.7}$$

В уравнении (6.7) в явном виде выражено соотношение между смещением и дисперсией оценки. На *n* шагах с действительными вознаграждениями смещение отсутствует, но дисперсия высокая, так как все они взяты из одной траектории. У $\hat{V}^{\pi}(s)$ дисперсия ниже, поскольку она отражает математические ожидания по всем виденным до сих пор траекториям, но есть смещение в связи с тем, что оно вычисляется

с помощью аппроксимации функции. Из сочетания этих двух типов оценки вытекает вывод, что дисперсия действительных вознаграждений тем больше, чем больше шагов прошло с момента t. Ближе к t выгода от использования несмещенной оценки может перевесить внесенную дисперсию. При увеличении n дисперсия в оценках, вероятно, начнет вызывать проблемы, и лучше перейти к оценке со смещением, но с низкой дисперсией. Число шагов с действительными вознаграждениями n регулирует соотношение между смещением и дисперсией оценки.

Объединив оценки за n шагов для Q^π с $\hat{V}^\pi(s_t)$, получим формулу для оценки функции преимущества (уравнение (6.8)):

$$A^\pi_{\text{nstep}}(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t) \approx$$

$$\approx r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots + \gamma^n r_{t-n} + \gamma^{n+1}\hat{V}^\pi(s_{t-n+1}) - \hat{V}^\pi(s_t). \tag{6.8}$$

Число шагов n с действительными вознаграждениями количественно регулирует дисперсию оценки преимущества и является гиперпараметром, который нужно настраивать. Малое значение n дает оценку с низкой дисперсией, но большим смещением, большое значение — оценку с высокой дисперсией, но меньшим смещением.

Обобщенная оценка преимущества

Обобщенная оценка преимущества (Generalized Advantage Estimation, GAE) [123] была предложена Шульманом и другими в качестве усовершенствования оценки по отдаче за n шагов для функции преимущества. Она решает проблему, вызванную необходимостью явного задания n — количества шагов с получением отдачи. Основная идея здесь в том, что вместо выбора одного значения n используется массив из большого количества значений. То есть преимущество рассчитывается с помощью средневзвешенного значения отдельных преимуществ, вычисленных при $n = 1, 2, 3... k$. Цель GAE — существенное снижение дисперсии оценки при как можно меньшем смещении.

GAE определяется как экспоненциальное средневзвешенное значение всех оценок преимуществ для последующей отдачи за n шагов. Это определение приведено в уравнении (6.9), а полный вывод GAE дан в примечании 6.1:

$$A^\pi_{\text{GAE}}(s_t, a_t) = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}, \tag{6.9}$$

где $\delta_t = r_t + \gamma V^\pi(s_{t+1}) - V^\pi(s_t)$.

Очевидно, GAE определяет средневзвешенное значение нескольких оценок преимущества с разными смещениями и дисперсиями. GAE придает наибольший вес преимуществу при одном шаге с большим смещением и низкой дисперсией, а также учитывает вклад оценок с меньшим смещением и более высокой дисперсией

при 2, 3... n шагах. Их вклад уменьшается экспоненциально по мере увеличения количества шагов. Скорость уменьшения регулируется коэффициентом λ. Следовательно, чем больше λ, тем выше дисперсия.

Примечание 6.1. Вывод обобщенной оценки преимущества

Вывод GAE не самый простой, но над ним стоит потрудиться, чтобы понимать, как она оценивает функцию преимущества. К счастью, в конечном итоге для GAE будет получено простое выражение, которое довольно легко реализовать.

$A_t^\pi(n)$ используется для обозначения оценки преимущества, вычисленной по последующей отдаче за n шагов. Например, в уравнениях (6.10) приведены $A_t^\pi(1)$, $A_t^\pi(2)$ и $A_t^\pi(3)$:

$$\begin{aligned} A_t^\pi(1) &= r_t + \gamma V^\pi(s_{t+1}) - V^\pi(s_t); \\ A_t^\pi(2) &= r_t + \gamma r_{t+1} + \gamma^2 V^\pi(s_{t+2}) - V^\pi(s_t); \\ A_t^\pi(3) &= r_t + \gamma r_{t-1} + \gamma^2 r_{t+2} + \gamma^3 V^\pi(s_{t+3}) - V^\pi(s_t). \end{aligned} \tag{6.10}$$

GAE определяется как экспоненциальное средневзвешенное значение всех оценок преимуществ по последующей отдаче за n шагов:

$$A_{t,\mathrm{GAE}}^\pi(s_t, a_t) = (1-\lambda)\left(A_t^\pi(1) + \lambda A_t^\pi(2) + \lambda^2 A_t^\pi(3) + \ldots\right), \tag{6.11}$$

где $\lambda \in [0, 1]$.

С уравнением (6.11) не слишком просто работать, к счастью, для упрощения GAE Шульман с коллегами [123] ввели переменную δ_t:

$$\delta_t = r_t + \gamma V^\pi(s_{t+1}) - V^\pi(s_t). \tag{6.12}$$

Заметьте, что $r_t + \gamma V^\pi(s_{t+1})$ — это оценка для $Q^\pi(s_t, a_t)$ при одном шаге, поэтому δ_t представляет функцию преимущества для временного шага t, рассчитанную для последующей отдачи, полученной за один шаг. Рассмотрим представление с помощью δ функции преимущества при n шагах. Во-первых, уравнение (6.10) может быть расширено и переписано в виде уравнений (6.13). Обратите внимание на то, что из уравнений (6.13) исключены все промежуточные члены от $V^\pi(s_{t+1})$ до $V^\pi(s_{t+n-1})$, остались только $V^\pi(s_t)$ и член, содержащий $V^\pi(s_{t+n})$ для последнего шага, как в уравнении (6.10):

$$\begin{aligned} A_t^\pi(1) &= r_t + \gamma V^\pi(s_{t+1}) - V^\pi(s_t); \\ A_t^\pi(2) &= r_t + \gamma V^\pi(s_{t+1}) - V^\pi(s_t) + \gamma\left(r_{t-1} + \gamma V^\pi(s_{t-2}) - V^\pi(s_{t+1})\right); \\ A_t^\pi(3) &= r_t + \gamma V^\pi(s_{t+1}) - V^\pi(s_t) + \gamma\left(r_{t+1} + \gamma V^\pi(s_{t+2}) - V^\pi(s_{t+1})\right) + \\ &+ \gamma^2\left(r_{t-2} + \gamma V^\pi(s_{t+3}) - V^\pi(s_{t+2})\right). \end{aligned} \tag{6.13}$$

Такая форма записи функции преимущества полезна, так как показывает, что она состоит из множества преимуществ для одного шага, вес которых экспоненциально изменяется с ростом степени γ при увеличении шага. Упростим с помощью δ члены уравнений (6.13) и получим уравнения (6.14). В них показано, что преимущество

на n шагах $A^{\pi}(n)$ — это сумма экспоненциально взвешенных значений δ, то есть одношаговых преимуществ:

$$\begin{aligned} A_t^{\pi}(1) &= \delta_t; \\ A_t^{\pi}(2) &= \delta_t + \gamma\delta_{t+1}; \\ A_t^{\pi}(3) &= \delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2}. \end{aligned} \tag{6.14}$$

Получив выражение $A^{\pi}(i)$ через δ, можно подставить его в уравнение (6.11) и получить простое выражение для GAE (уравнение (6.15)):

$$A_{t,\mathrm{GAE}}^{\pi}(s_t, a_t) = \sum_{i=0}^{\infty}(\gamma\lambda)^i\delta_{t+i}. \tag{6.15}$$

Как GAE, так и оценки функции преимущества по отдаче за n шагов включают коэффициент дисконтирования γ, который определяет, насколько алгоритм учитывает будущие вознаграждения по сравнению с текущим. Кроме того, в обоих случаях присутствует параметр, регулирующий соотношение смещения и дисперсии: n для функции преимущества и λ для GAE. Так чего же мы добились, применив GAE?

Несмотря на то что как n, так и λ регулируют соотношение смещения и дисперсии, они делают это разными способами. Значение n — жестко заданное, так как оно точно определяет точку, в которой при оценке V-функции включаются высокодисперсные вознаграждения. В противоположность этому λ — гибкий параметр: меньшие значения λ придадут больший вес оценке V-функции, тогда как бóльшие значения сделают более весомыми действительные вознаграждения. Тем не менее, если только λ не равна нулю[1] или единице[2], ее применение по-прежнему позволяет учитывать оценки с более высокой и более низкой дисперсией ввиду гибкости такого подхода.

6.2.2. Настройка функции преимущества

Мы рассмотрели два способа оценки функции преимущества. Оба они предполагают, что у нас есть оценка V^{π}, как показано далее:

$$A_{\mathrm{nstep}}^{\pi}(s_t, a_t) \approx r_t + \gamma r_{t+1} + \ldots + \gamma^n r_{t+n} + \gamma^{n+1}\hat{V}^{\pi}(s_{t+n+1}) - \hat{V}^{\pi}(s_t); \tag{6.16}$$

$$A_{\mathrm{GAE}}^{\pi}(s_t, a_t) \approx \sum_{i=0}^{\infty}(\gamma\lambda)^i\delta_{t+i}, \tag{6.17}$$

где $\delta_t = r_t + \gamma\hat{V}^{\pi}(s_{t+1}) - \hat{V}^{\pi}(s_t)$.

---

[1] То, что сводит GAE к одношаговой оценке функции преимущества, — это оценка значений отдачи по методу временных разностей.

[2] В этом случае для получения оценки используются только действительные вознаграждения — это оценка вознаграждений по методу Монте-Карло.

Мы настраиваем V^{π} с помощью TD-обучения так же, как при настройке Q^{π} для DQN. Вкратце процесс обучения происходит следующим образом. Параметризируем V^{π} с помощью θ, порождаем V_{tar}^{π} для каждого из накопленных агентом прецедентов, минимизируем разницу между $\hat{V}^{\pi}(s;\theta)$ и V_{tar}^{π} с помощью регрессионной функции потерь, такой как средняя квадратическая ошибка. Повторяем этот процесс много раз.

V_{tar}^{π} могут быть порождены с помощью любой подходящей оценки. Простейший способ — установить $V_{\text{tar}}^{\pi}(s) = r + \hat{V}^{\pi}(s';\theta)$. Это естественным образом может быть обобщено в оценку для n шагов):

$$V_{\text{tar}}^{\pi}(s_t) = r_t + \gamma r_{t+1} + \ldots + \gamma^n r_{t+n} + \gamma^{n+1}\hat{V}^{\pi}(s_{t+n+1}). \tag{6.18}$$

В качестве альтернативы можно использовать оценку V_{tar}^{π} по методу Монте-Карло, приведенную в уравнении (6.19):

$$V_{\text{tar}}^{\pi}(s_t) = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}. \tag{6.19}$$

Или задать:

$$V_{\text{tar}}^{\pi}(s_t) = A_{\text{GAE}}^{\pi}(s_t, a_t) + \hat{V}^{\pi}(s_t). \tag{6.20}$$

На практике во избежание дополнительных вычислений выбор V_{tar}^{π} часто связывают с методом, применяемым для оценки преимущества. Например, можно воспользоваться уравнением (6.18) при оценке преимущества с помощью отдачи, полученной за n шагов, или уравнением (6.20) — для оценки преимущества с помощью GAE.

При настройке $\hat{V}^{\pi}$ возможно применение более продвинутых процедур оптимизации. Например, в статье о GAE [123] $\hat{V}^{\pi}$ настраивается с помощью метода доверительной области.

6.3. Алгоритм A2C

Здесь актор и критик объединяются для получения алгоритма актора-критика с преимуществом (A2C) (алгоритм 6.1).

Алгоритм 6.1. A2C

```
Установка β ≥ 0 # вес, регулирующий энтропию
Установка α_a ≥ 0 # скорость обучения актора
Установка α_c ≥ 0 # скорость обучения критика
Инициализация параметров актора θ_a и критика θ_c[1] случайными значениями
```

[1] Следует отметить, что актор и критик могут быть как отдельными сетями, так и общей сетью. За более подробными сведениями обращайтесь к разделу 6.5.

```
 5: for episode = 0...MAX_EPISODE do
 6:     Накопить и сохранить данные (s_t, a_t, r_t, s'_t),
        ➥ действуя в среде с помощью текущей стратегии
 7:     for t = 0...T do
 8:         Рассчитать прогнозные V-значения V̂^π(s_t) с помощью сети критика θ_c
 9:         Вычислить преимущество Âπ(s_t, a_t), воспользовавшись сетью критика θ_c
10:         Рассчитать V^π_tar(s_t), применив сеть критика θ_c
            ➥ и/или данные по траектории
11:         При необходимости найти энтропию H_t распределения для стратегии
            с помощью сети актора θ_A. Иначе установить β = 0
12:     end for
13:     Расчет функции потерь для полезностей, например, с помощью
        ➥ средней квадратической ошибки:
14:     L_val(θ_c) = (1/T) Σ_{t=0}^{T} (V̂^π(s_t) − V^π_tar(s_t))^2
15:     Расчет функции потерь для стратегии:
16:     L_pol(θ_A) = (1/T) Σ_{t=0}^{T} (−Â^π(s_t, a_t) log π_{θ_A}(a_t | s_t) − βH_t)
17:     Обновление параметров критика, например, с помощью SGD[1]:
18:     θ_c = θ_c + α_c ∇_{θ_c} L_val(θ_c)
19:     Обновление параметров актора, например, с помощью SGD:
20:     θ_A = θ_A + α_A ∇_{θ_A} L_pol(θ_A)
21: end for
```

Все алгоритмы, которые мы до сих пор встречали, фокусировались на настройке чего-то одного: как действовать (стратегия) или как оценивать действия (критик). Алгоритмы актора-критика настраивают и то и другое вместе. За исключением этого, все элементы цикла обучения должны быть вам знакомы, так как они были частью приведенных ранее алгоритмов. Пройдем алгоритм 6.1 пошагово.

- Строки 1–3. Установка значений важных гиперпараметров β, α_a, α_c. β определяет, насколько нужно регулировать энтропию (детали смотрите далее). Параметры α_a и α_c — это скорости обучения, применяемые при оптимизации обеих сетей. Их значения могут быть одинаковыми или разными. Они в значительной степени зависят от решаемой задачи, и их нужно находить опытным путем.
- Строка 4. Инициализация параметров обеих сетей случайными значениями.
- Строка 6. Накопление данных с помощью текущей сети стратегии θ_a. В этом алгоритме показано обучение на эпизодах, однако данный подход применим и к обучению на пакетах.
- Строки 8–10. Для всех прецедентов (s_t, a_t, r_t, s'_t) в эпизоде рассчитать $\hat{V}^{\pi}(s_t)$, $V^{\pi}_{tar}(s_t)$ и $\hat{A}^{\pi}(s_t, a_t)$ с помощью сети критика.
- Строка 11. Для всех прецедентов (s_t, a_t, r_t, s'_t) в эпизоде при необходимости вычислить энтропию для текущего распределения действий по стратегии π_{θ_a} с помощью сети критика. Роль энтропии подробнее обсуждается в примечании 6.2.

[1] Стохастический градиентный спуск.

- Строки 13 и 14. Расчет функции потерь для полезности. Как и в случае с алгоритмами DQN, в качестве меры различия $\hat{V}^{\pi}(s_t)$ и $V_{tgt}^{\pi}(s_t)$ мы выбираем среднее квадратическое отклонение. Тем не менее могут быть использованы любые другие функции потерь, например функция потерь Хьюберта.
- Строки 15 и 16. Определение функции потерь для стратегии. Производится в той же форме, что и в алгоритме REINFORCE, с прибавлением при необходимости члена, соответствующего регуляризации энтропии. Обратите внимание: мы минимизируем функцию потерь, но градиент стратегии нужно максимизировать, отсюда и знак минус перед $\hat{A}^{\pi}(s_t, a_t)\log\pi_{\theta_a}(a_t|s_t)$, как в REINFORCE.
- Строки 17 и 18. Обновление параметров критика с помощью градиента функции потери для полезностей.
- Строки 19 и 20. Обновление параметров актора с помощью стратегии.

Примечание 6.2. Регуляризация энтропии

Роль регуляризации энтропии заключается в том, чтобы поощрять исследование разных действий. Эта идея впервые была предложена Вильямсом и Пенгом в 1991 году [149] и с тех пор стала популярным преобразованием алгоритмов обучения с подкреплением, задействующих градиент стратегии.

Чтобы понять, почему в этом случае поощряется исследование, сначала следует отметить, что у более однородного распределения энтропия больше. Чем однороднее распределение действий по стратегии, тем больше различных действий она выдает. Напротив, менее однородные стратегии, производящие схожие действия, имеют меньшую энтропию.

Рассмотрим, почему добавление энтропии в градиент стратегии, приведенное в уравнении (6.21), поощряет более однородную стратегию:

$$L_{pol}(\theta_A) = \frac{1}{T}\sum_{t=0}^{T}\left(-\hat{A}^{\pi}(s_t, a_t)\log\pi_{\theta_A}(a_t|s_t) - \beta H_t\right). \tag{6.21}$$

Энтропия H и параметр β всегда неотрицательные, тогда значение $-\beta H$ всегда отрицательное. Когда стратегия далека от однородной, энтропия снижается, $-\beta H$ растет и сильнее влияет на функцию потерь, тем самым поощряется переход стратегии к более однородной. Напротив, когда стратегия более однородная, энтропия растет, $-\beta H$ снижается и вносит меньший вклад в функцию потерь, а у стратегии мало стимулов для изменения через добавку, соответствующую энтропии.

В таком способе преобразования функции потерь выражена предпочтительность большего разнообразия действий при условии, что член $\hat{A}^{\pi}(s_t, a_t)\log\pi_{\theta_a}(a_t|s_t)$ не уменьшается слишком сильно. Соотношение между двумя этими членами целевой функции регулируется параметром β.

6.4. Реализация A2C

Теперь мы располагаем всеми необходимыми элементами для реализации алгоритма актора-критика. Основные компоненты:

- оценка преимущества, например, по отдаче за n шагов или GAE;
- функции потерь для полезности и стратегии;
- цикл обучения.

Далее мы обсудим реализацию каждого из этих компонентов, закончив циклом обучения, в котором все они объединяются. Поскольку метод актора-критика является концептуальным расширением REINFORCE, его реализация унаследована от класса `Reinforce`.

Метод актора-критика — это основанный на стратегии алгоритм, поскольку компонент актора настраивает стратегию с помощью градиента стратегии. Как следствие, обучение алгоритмов актора-критика происходит с помощью классов памяти для обучения по актуальному опыту, таких как `OnPolicyReplay` с обучением на эпизодах или `OnPolicyBatchReplay` с обучением на пакетах данных. Приведенный далее код применим к обоим подходам.

6.4.1. Оценка преимущества

Оценка преимущества по отдаче за n шагов

Основная хитрость при реализации оценки Q^{π} за n шагов в том, что у нас есть доступ к вознаграждениям, полученным за эпизод, в виде последовательности. Можно параллельно рассчитать дисконтированные суммы n вознаграждений для всех элементов в пакете, воспользовавшись векторной арифметикой, как показано в листинге 6.1. Подход заключается в следующем.

1. Инициализация вектора `rets` оценками Q-значений, то есть значениями отдачи за n шагов (строка 4).
2. По соображениям эффективности все вычисления выполняются, начиная с последнего члена до первого. `future_ret` — временная переменная для суммирования вознаграждений по мере нашего продвижения в обратном направлении (строка 5). Она инициализируется значением `next_v_pred`, так как последний член Q-оценки за n шагов — это $\hat{V}^{\pi}(s_{t-n+1})$.
3. `note_dones` — это бинарная переменная для обработки границ эпизодов и для предотвращения продолжения суммирования за рамками эпизода.
4. Обратите внимание на то, что Q-оценка за n шагов определяется рекурсивно, то есть $Q_{\text{current}} = r_{\text{current}} + \gamma Q_{\text{next}}$. Это в точности отражено в строке 8.

Листинг 6.1. Реализация актора-критика, расчет преимуществ по отдаче за n шагов

```
# slm_lab/lib/math_util.py

def calc_nstep_returns(rewards, dones, next_v_pred, gamma, n):
    rets = torch.zeros_like(rewards)
    future_ret = next_v_pred
    not_dones = 1 - dones
    for t in reversed(range(n)):
        rets[t] = future_ret = rewards[t] + gamma * future_ret *
        ➥ not_dones[t]
    return rets
```

Теперь классу `ActorCritic` нужен метод для расчета оценок преимущества и целевых значений V для вычисления функции потерь для полезности и стратегии соответственно. Это относительно просто, как показано в листинге 6.2.

Следует упомянуть одну важную деталь об `advs` и `v_targets`. У них нет градиентов, как можно видеть по операциям `torch.no_grad()` и `.detach()` в строках 9–11. В функции потерь для стратегии (уравнение (6.1)) преимущество выступает лишь в роли скалярного множителя градиента логарифма вероятности стратегии. Что касается функции потерь для полезности из алгоритма 6.1 (строки 13 и 14), то мы предполагаем, что целевое V-значение фиксировано, а цель — это обучение критика прогнозированию V-значений, близких к нему.

Листинг 6.2. Реализация актора-критика, расчет преимуществ за n шагов и целевых V-значений

```
 1 # slm_lab/agent/algorithm/actor_critic.py
 2
 3 class ActorCritic(Reinforce):
 4     ...
 5
 6     def calc_nstep_advs_v_targets(self, batch, v_preds):
 7         next_states = batch['next_states'][-1]
 8         ...
 9         with torch.no_grad():
10             next_v_pred = self.calc_v(next_states, use_cache=False)
11         v_preds = v_preds.detach()
12         ...
13         nstep_rets = math_util.calc_nstep_returns(batch['rewards'],
           ➥ batch['dones'], next_v_pred, self.gamma,
           ➥ self.num_step_returns)
14         advs = nstep_rets - v_preds
15         v_targets = nstep_rets
16         ...
17         return advs, v_targets
```

Оценка преимущества с помощью GAE

Реализация GAE, приведенная в листинге 6.3, во многом аналогична реализации для оценки преимущества по отдаче за n шагов. В ней используется то же самое вычисление в обратном порядке, за исключением того, что на каждом шаге нужно дополнительно рассчитывать член δ (строка 11).

Листинг 6.3. Реализация актора-критика, расчет GAE

```
 1 # slm_lab/lib/math_util.py
 2
 3 def calc_gaes(rewards, dones, v_preds, gamma, lam):
 4     T = len(rewards)
 5     assert T + 1 == len(v_preds) # включение в v_preds состояний
       ➥ и одного последнего состояния next_state
 6     gaes = torch.zeros_like(rewards)
 7     future_gae = torch.tensor(0.0, dtype=rewards.dtype)
 8     # умножение на not_dones для учета границы
       ➥ эпизода (у последнего эпизода нет V(s'))
 9     not_dones = 1 - dones
10     for t in reversed(range(T)):
11         delta = rewards[t] + gamma * v_preds[t + 1] * not_dones[t] -
           ➥ v_preds[t]
12         gaes[t] = future_gae = delta + gamma * lam * not_dones[t] * future_gae
13    return gaes
```

В листинге 6.4 метод класса актора-критика для расчета преимуществ и целевых V-значений почти аналогичен соответствующему методу для отдачи за n шагов, но имеет два важных отличия. Во-первых, `calc_gaes` (строка 14) возвращает полные оценки преимуществ, тогда как `calc_nstep_returns` в случае отдачи за n шагов — оценки Q-значений. Поэтому для восстановления целевых V-значений нужно прибавить прогнозные V-значения (строка 15). Во-вторых, хорошей практикой является унификация оценок преимуществ, полученных с помощью GAE (строка 16).

Листинг 6.4. Реализация актора-критика, расчет преимуществ с помощью GAE и целевых V-значений

```
 1 # slm_lab/agent/algorithm/actor_critic.py
 2
 3 class ActorCritic(Reinforce):
 4     ...
 5
 6     def calc_gae_advs_v_targets(self, batch, v_preds):
 7         next_states = batch['next_states'][-1]
 8         ...
 9         with torch.no_grad():
10             next_v_pred = self.calc_v(next_states, use_cache=False)
11         v_preds = v_preds.detach() # adv не суммирует градиенты
12         ...
13         v_preds_all = torch.cat((v_preds, next_v_pred), dim=0)
```

```
14          advs = math_util.calc_gaes(batch['rewards'], batch['dones'],
            ➥ v_preds_all, self.gamma, self.lam)
15          v_targets = advs + v_preds
16          advs = math_util.standardize(advs) # унификация только
            ➥ для advs, а не v_targets
17          ...
18          return advs, v_targets
```

6.4.2. Расчет функции потерь для полезности и стратегии

В листинге 6.5 функция потерь для стратегии, вычисляется таким же образом, как и в реализации REINFORCE. Единственное отличие — использование преимущества вместо значений отдачи в качестве подкрепляющего сигнала, так что метод из REINFORCE может быть унаследован и применен повторно (строка 7).

Функция потерь для полезности — это просто мера различия $\hat{V}^{\pi}$ (`v_preds`) и V^{π}_{tar} (`v_targets`). Мы вольны выбрать любую подходящую меру, такую как средняя квадратическая ошибка, установив параметр `net.loss_spec` в файле `spec`. Это инициализирует функцию потерь `self.net_loss_fn`, применяемую в строке 11.

Листинг 6.5. Реализация актора-критика, две функции потерь

```
 1 # slm_lab/agent/algorithm/actor_critic.py
 2
 3 class ActorCritic(Reinforce):
 4     ...
 5
 6     def calc_policy_loss(self, batch, pdparams, advs):
 7         return super().calc_policy_loss(batch, pdparams, advs)
 8
 9     def calc_val_loss(self, v_preds, v_targets):
10         assert v_preds.shape == v_targets.shape, f'{v_preds.shape}
           ➥ != {v_targets.shape}'
11         val_loss = self.val_loss_coef * self.net.loss_fn(v_preds, v_targets)
12         return val_loss
```

6.4.3. Цикл обучения актора-критика

Актор и критик могут быть реализованы либо как отдельные сети, либо как одна общая сеть. Это отражено в методе `train` из листинга 6.6. В строках 10–15 для обучения рассчитываются функции потерь для полезности и стратегии. Если в реализации задействуется общая сеть (строка 16), то два значения функции потерь объединяются и используются для ее обучения (строки 17 и 18). Если актор и критик — раздельные сети, то два значения функции потерь применяются по отдельности для обучения соответствующих сетей (строки 20 и 21). В разделе 6.5 архитектура сети рассматривается более детально.

Листинг 6.6. Реализация актора-критика, метод обучения

```
# slm_lab/agent/algorithm/actor_critic.py

class ActorCritic(Reinforce):
    ...

    def train(self):
        ...
        clock = self.body.env.clock
        if self.to_train == 1:
            batch = self.sample()
            clock.set_batch_size(len(batch))
            pdparams, v_preds = self.calc_pdparam_v(batch)
            advs, v_targets = self.calc_advs_v_targets(batch, v_preds)
            policy_loss = self.calc_policy_loss(batch, pdparams, advs)
            ➥ # из актора
            val_loss = self.calc_val_loss(v_preds, v_targets) # из критика
            if self.shared: # общая сеть
                loss = policy_loss + val_loss
                self.net.train_step(loss, self.optim, self.lr_scheduler,
                ➥ clock=clock, global_net=self.global_net)
            else:
                self.net.train_step(policy_loss, self.optim,
                ➥ self.lr_scheduler, clock=clock, global_net=self.global_net)
                self.critic_net.train_step(val_loss, self.critic_optim,
                ➥ self.critic_lr_scheduler, clock=clock,
                ➥ global_net=self.global_critic_net)
                loss = policy_loss + val_loss
            # возврат к начальным значениям переменных
            self.to_train = 0
            return loss.item()
        else:
            return np.nan
```

6.5. Архитектура сети

В алгоритмах актора-критика настраиваются две параметризированные функции: π (актор) и $V^{\pi}(s)$ (критик). Это отличает их от всех рассмотренных нами до сих пор алгоритмов, которые настраивают одну функцию. Естественно, в этом случае при проектировании нейронных сетей нужно рассматривать больше факторов. Должны ли они иметь общие параметры? И если да, то сколько?

У общих параметров есть плюсы и минусы. Это хорошо с концептуальной точки зрения, так как настройка π и настройка $V^{\pi}(s)$ для одного и того же задания взаимосвязаны. Настройка $V^{\pi}(s)$ связана с эффективностью оценки состояний, а настройка π — с пониманием того, как выбирать оптимальные действия в этих состояниях. У них общие входные данные, и в обоих случаях хорошая аппроксимация

функции включает в себя настройку такого представления пространства состояний, чтобы схожие состояния находились в одном кластере. Поэтому вполне вероятно, что обе аппроксимации извлекут выгоду из совместной настройки пространства состояний на низших уровнях. Кроме того, общие параметры позволяют уменьшить общее количество настраиваемых параметров, и таким образом может быть повышена эффективность выборок для алгоритма. Это особенно важно для алгоритмов актора-критика, поскольку стратегия подкрепляется путем обучения критика. Если сети актора и критика раздельные, то первая может ничему не научиться до тех пор, пока последняя не станет приемлемой, а на это может уйти много шагов обучения. Однако, если у них есть общие параметры, актор может извлечь выгоду из представления состояний, настроенного критиком.

Высокоуровневая архитектура общей сети, приведенная в правой части рис. 6.1, демонстрирует типичный обмен параметрами в алгоритмах актора-критика. Актор и критик объединены нижними слоями сети. У них также есть по одному или по несколько отдельных слоев в верхней части сети. Это обусловлено тем, что пространство выходных данных для каждой из сетей разное: для актора это распределение вероятностей по действиям, а для критика — одно скалярное значение, представляющее $V^{\pi}(s)$. Однако так как у каждой из сетей свое задание, то можно ожидать, что им для хорошей работы потребуется некоторое количество специальных параметров. Это может быть один слой вверху общего тела или несколько слоев. Более того, слои, специфичные для актора и критика, не обязательно должны совпадать по количеству или типу.

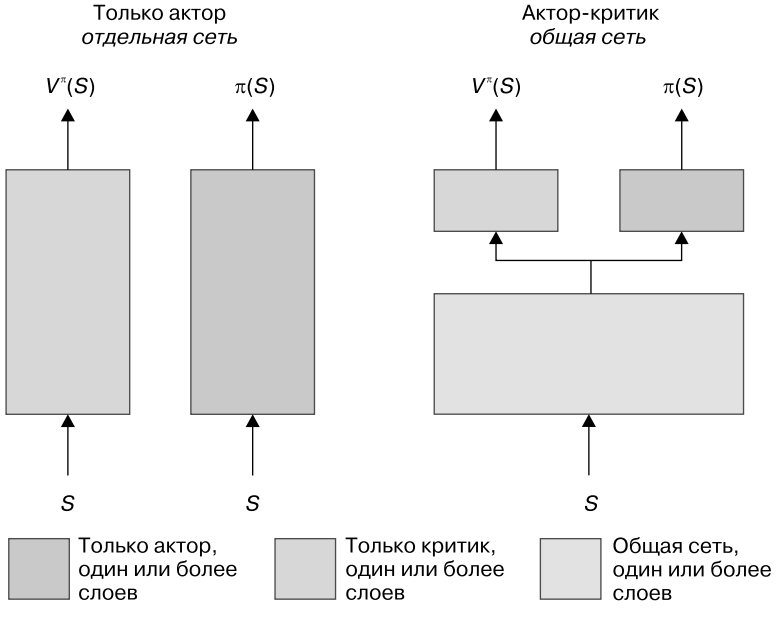


Рис. 6.1. Архитектура сетей актора-критика: сравнение общей сети и отдельных сетей

Один серьезный недостаток общих параметров заключается в том, что они могут сделать обучение менее устойчивым. Это вызвано тем, что теперь через всю сеть совместно обратно распространяются градиенты для двух компонентов: градиент стратегии из актора и градиент функции полезности из критика. Масштабы двух этих градиентов могут различаться, и разница между ними должна быть компенсирована, чтобы агент хорошо обучался [75]. Например, компонент логарифма вероятности из функции потерь для стратегии находится в пределах $(-\infty, 0]$, а абсолютное значение функции преимущества может быть довольно малым, особенно если разрешение[1] действия невысокое. В совокупности два этих фактора могут привести к малым абсолютным значениям функции потерь для стратегии. В то же время масштаб величины функции потерь связан с полезностью пребывания в состоянии, которая может быть очень большой.

Компенсация разности между двумя градиентами, потенциально различающимися масштабами, обычно реализуется как прибавление скалярного веса к одной из функций потерь, чтобы уменьшить или увеличить ее масштаб. Например, функция потерь для полезности в листинге 6.5 масштабируется с помощью `self.val_loss_coef` (строка 11). Тем не менее это еще один гиперпараметр, который нужно настраивать во время обучения.

6.6. Обучение агента A2C

В этом разделе мы покажем, как обучить агента методом актора-критика в игре Pong из Atari с помощью различных оценок преимущества: сначала отдачи за *n* шагов, затем GAE. Потом мы применим A2C с GAE к среде с непрерывным управлением BipedalWalker.

6.6.1. A2C с оценкой преимущества по отдаче за n шагов в Pong

Файл `spec`, который настраивает агента в методе актора-критика с оценкой преимущества по отдаче за *n* шагов, приведен в листинге 6.7. Этот файл имеется в SLM Lab в `slm_lab/spec/benchmark/a2c/a2c_nstep_pong.json`.

Листинг 6.7. Файл spec для A2C по отдаче за n шагов

```
1 # slm_lab/spec/benchmark/a2c/a2c_nstep_pong.json
2
3 {
```

[1] Разрешение можно интерпретировать как модельное время, прошедшее между шагами. Среды с высоким разрешением моделируют время мелкозернистым образом. Это означает, что при переходе из одного состояния в другое изменения невелики, так как проходит немного времени. Среды с низким разрешением моделируют время более крупными отрезками, что приводит к большим изменениям при переходе из одного состояния в другое.

```
    "a2c_nstep_pong": {
        "agent": [{
            "name": "A2C",
            "algorithm": {
                "name": "ActorCritic",
                "action_pdtype": "default",
                "action_policy": "default",
                "explore_var_spec": null,
                "gamma": 0.99,
                "lam": null,
                "num_step_returns": 11,
                "entropy_coef_spec": {
                    "name": "no_decay",
                    "start_val": 0.01,
                    "end_val": 0.01,
                    "start_step": 0,
                    "end_step": 0
                },
                "val_loss_coef": 0.5,
                "training_frequency": 5
            },
            "memory": {
                "name": "OnPolicyBatchReplay"
            },
            "net": {
                "type": "ConvNet",
                "shared": true,
                "conv_hid_layers": [
                    [32, 8, 4, 0, 1],
                    [64, 4, 2, 0, 1],
                    [32, 3, 1, 0, 1]
                ],
                "fc_hid_layers": [512],
                "hid_layers_activation": "relu",
                "init_fn": "orthogonal_",
                "normalize": true,
                "batch_norm": false,
                "clip_grad_val": 0.5,
                "use_same_optim": false,
                "loss_spec": {
                    "name": "MSELoss"
                },
                "actor_optim_spec": {
                    "name": "RMSprop",
                    "lr": 7e-4,
                    "alpha": 0.99,
                    "eps": 1e-5
                },
                "critic_optim_spec": {
                    "name": "RMSprop",
                    "lr": 7e-4,
                    "alpha": 0.99,
                    "eps": 1e-5
```

```
57                 },
58                 "lr_scheduler_spec": null,
59                 "gpu": true
60             }
61         }],
62         "env": [{
63             "name": "PongNoFrameskip-v4",
64             "frame_op": "concat",
65             "frame_op_len": 4,
66             "reward_scale": "sign",
67             "num_envs": 16,
68             "max_t": null,
69             "max_frame": 1e7
70         }],
71         "body": {
72             "product": "outer",
73             "num": 1,
74         },
75         "meta": {
76             "distributed": false,
77             "log_frequency": 10000,
78             "eval_frequency": 10000,
79             "max_session": 4,
80             "max_trial": 1
81         }
82     }
83 }
```

Пройдемся по основным компонентам.

- **Алгоритм** — это актор-критик (строка 8), стратегия выбора действий соответствует принятой по умолчанию стратегии (строка 10) для дискретных пространств действий (категориальное распределение). Значение γ устанавливается в строке 12. Если для λ указано `lam` (не `null`), то для оценки преимуществ применяется GAE. Если вместо этого указано `num_step_returns`, то используется отдача за n шагов (строки 13 и 14). Коэффициент энтропии и скорость его уменьшения во время обучения задаются в строках 15–21. Коэффициент функции потерь для полезности определен в строке 22.
- **Архитектура сети** — сверточная нейронная сеть с тремя сверточными слоями и одним полносвязным слоем с функцией активации ReLU (строки 29–37). У актора и критика общая сеть, что указано в строке 30. Сеть тренируется на графическом процессоре, если он доступен (строка 59).
- **Оптимизатор** — это RMSprop [50] со скоростью обучения 0,0007 (строки 46–51). Если применяются раздельные сети, то возможно задание другого оптимизатора для сети критика (строки 52–57) путем установки значения `false` для `use_same_optim` (строка 42). Поскольку в данном случае сеть общая, он не используется. Здесь нет уменьшения скорости обучения (строка 58).

- **Частота обучения** — по пакетам, в связи с тем что была выбрана память `OnPolicyBatchReplay` (строка 26), размер пакета составляет 5 × 16. Это регулируется `training_frequency` (строка 23) и количеством параллельных сред (строка 67). Параллельные среды обсуждаются в главе 8.
- **Среда** — Pong из Atari [14], [18] (строка 63).
- **Длительность обучения** — 10 млн шагов (строка 69).
- **Оценка агента** производится каждые 10 000 шагов (строка 78).

Для обучения агента методом актора-критика с помощью SLM Lab запустите в терминале команды, приведенные в листинге 6.8. Агент должен начинать со счета –21 и в среднем достигать счета, близкого к максимальному значению 21, после 2 млн кадров.

Листинг 6.8. Обучение агента A2C по отдаче за n шагов

```
1 conda activate lab
2 python run_lab.py slm_lab/spec/benchmark/a2c/a2c_nstep_pong.json a2c_nstep_
pong train
```

Таким образом, будет запущено испытание `Trial` с четырьмя сессиями `Session` для получения усредненных результатов. На завершение испытания должно уйти около половины дня при запуске на графическом процессоре. График результатов и график со скользящим средним показаны на рис. 6.2.

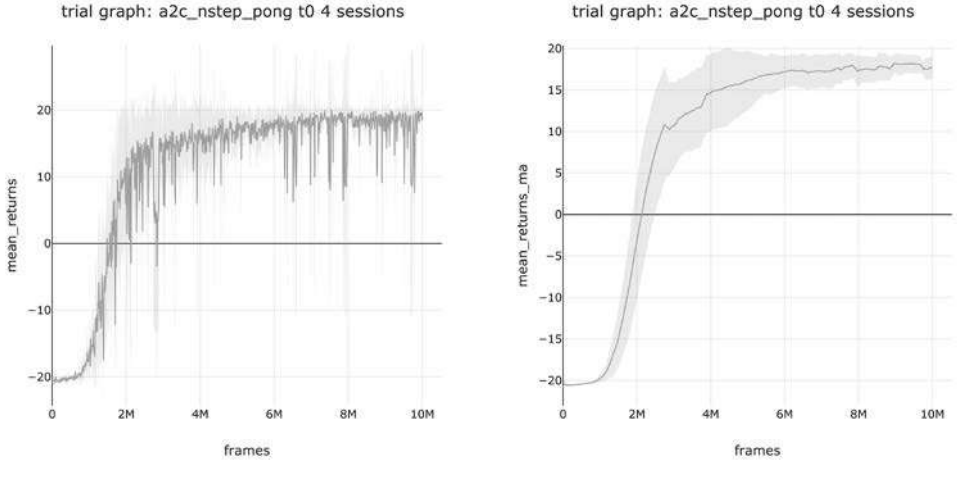


Рис. 6.2. Графики усредненных по четырем сессиям результатов испытания актора-критика (с оценкой преимущества по отдаче за n шагов) из SLM Lab. По вертикальной оси отложены полные вознаграждения в контрольных точках, усредненные по восьми эпизодам. По горизонтальной оси — все кадры обучения. График со скользящим средним по оценкам в 100 контрольных точках показан справа

6.6.2. A2C с GAE в Pong

Чтобы переключиться с оценки преимущества по отдаче за *n* шагов на GAE, просто измените файл `spec` из листинга 6.7, указав значение для `lam` и установив для `num_step_returns` значение `null`, как показано в листинге 6.9. Этот файл также имеется в SLM Lab в `slm_lab/spec/benchmark/a2c/a2c_gae_pong.json`.

Листинг 6.9. Файл spec для A2C с GAE

```
# slm_lab/spec/benchmark/a2c/a2c_nstep_pong.json

{
    "a2c_gae_pong": {
        "agent": [{
            "name": "A2C",
            "algorithm": {
                ...
                "lam": 0.95,
                "num_step_returns": null,
                ...
    }
}
```

Затем для обучения агента запустите в терминале команды, приведенные в листинге 6.10.

Листинг 6.10. Обучение агента A2C с GAE

```
conda activate lab
python run_lab.py slm_lab/spec/benchmark/a2c/a2c_gae_pong.json
  ➥ a2c_gae_pong train
```

Будет запущено испытание `Trial` для получения графиков, приведенных на рис. 6.3.

6.6.3. A2C по отдаче за n шагов в BipedalWalker

До сих пор мы проводили обучение в дискретных средах. Как вы помните, основанные на стратегии методы могут быть напрямую применены к задачам непрерывного управления. Теперь рассмотрим среду BipedalWalker, с которой познакомились в разделе 1.1.

В листинге 6.11 показан файл `spec`, который настраивает агента A2C по отдаче за *n* шагов для среды BipedalWalker. Файл хранится в SLM Lab в `slm_lab/spec/benchmark/a2c/a2c_nstep_cont.json`. Обратите особое внимание на изменения в архитектуре сети (строки 29–31) и среде (строки 54–57).

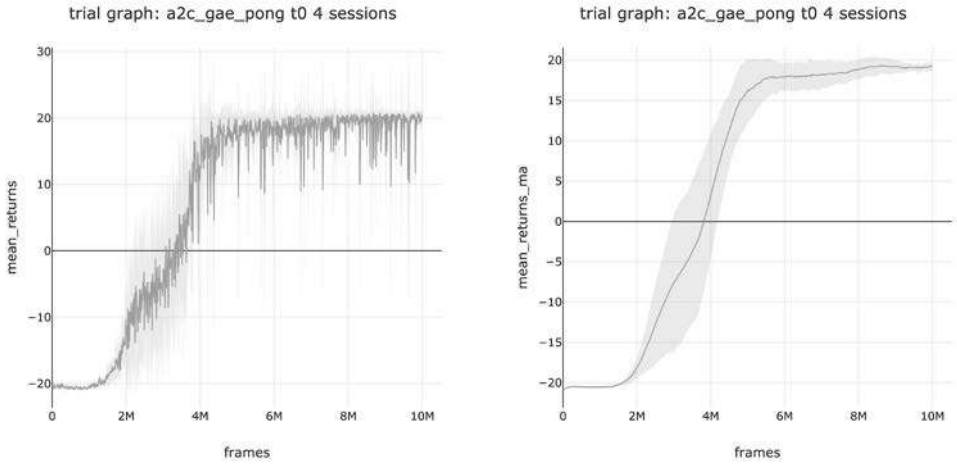


Рис. 6.3. Графики усредненных по четырем сессиям результатов испытания актора-критика (с GAE) из SLM Lab

Листинг 6.11. Файл spec для A2C по отдаче за n шагов в BipedalWalker

```
# slm_lab/spec/benchmark/a2c/a2c_nstep_cont.json

{
    "a2c_nstep_bipedalwalker": {
        "agent": [{
            "name": "A2C",
            "algorithm": {
                "name": "ActorCritic",
                "action_pdtype": "default",
                "action_policy": "default",
                "explore_var_spec": null,
                "gamma": 0.99,
                "lam": null,
                "num_step_returns": 5,
                "entropy_coef_spec": {
                    "name": "no_decay",
                    "start_val": 0.01,
                    "end_val": 0.01,
                    "start_step": 0,
                    "end_step": 0
                },
                "val_loss_coef": 0.5,
                "training_frequency": 256
            },
            "memory": {
                "name": "OnPolicyBatchReplay",
            },
```

```
            "net": {
                "type": "MLPNet",
                "shared": false,
                "hid_layers": [256, 128],
                "hid_layers_activation": "relu",
                "init_fn": "orthogonal_",
                "normalize": true,
                "batch_norm": false,
                "clip_grad_val": 0.5,
                "use_same_optim": false,
                "loss_spec": {
                    "name": "MSELoss"
                },
                "actor_optim_spec": {
                    "name": "Adam",
                    "lr": 3e-4,
                },
                "critic_optim_spec": {
                    "name": "Adam",
                    "lr": 3e-4,
                },
                "lr_scheduler_spec": null,
                "gpu": false
            }
        }],
        "env": [{
            "name": "BipedalWalker-v2",
            "num_envs": 32,
            "max_t": null,
            "max_frame": 4e6
        }],
        "body": {
            "product": "outer",
            "num": 1
        },
        "meta": {
            "distributed": false,
            "log_frequency": 10000,
            "eval_frequency": 10000,
            "max_session": 4,
            "max_trial": 1
        }
    }
}
```

Для обучения агента запустите в терминале команды, приведенные в листинге 6.12.

Листинг 6.12. Обучение агента A2C по отдаче за n шагов в BipedalWalker

```
conda activate lab
python run_lab.py slm_lab/spec/benchmark/a2c/a2c_nstep_cont.json
  ➥ a2c_nstep_bipedalwalker train
```

Будет запущено испытание `Trial`, чтобы получить графики, приведенные на рис. 6.4.

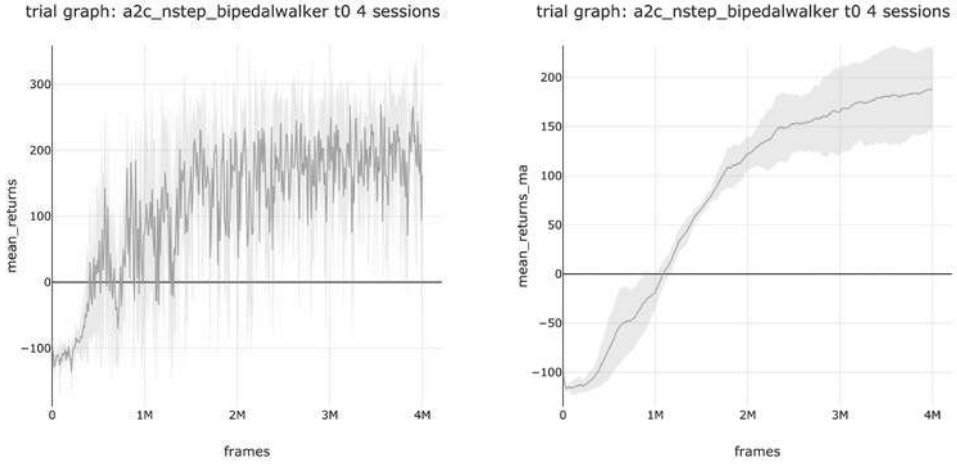


Рис. 6.4. График усредненных по четырем сессиям результатов испытания A2C по отдаче за n шагов в BipedalWalker из SLM Lab

BipedalWalker — сложная непрерывная среда, задача которой считается решенной, когда скользящее среднее для полного вознаграждения выше 300. На рис. 6.4 агент не смог достичь этого за 4 млн кадров. Мы вернемся к этой проблеме в главе 7, где будет представлена более удачная попытка.

6.7. Результаты экспериментов

В этом разделе мы проведем для актора-критика два эксперимента с помощью SLM Lab. В первом изучается влияние количества шагов с помощью оценки преимущества по отдаче за n шагов. Во втором — влияние λ при использовании GAE. В качестве более сложной среды возьмем игру Breakout из Atari.

6.7.1. Эксперимент по определению влияния отдачи за n шагов

Количество шагов n регулирует соотношение между смещением и дисперсией в оценке преимущества по отдаче за n шагов — чем выше n, тем больше дисперсия. Количество шагов n — это настраиваемый гиперпараметр.

В этом эксперименте путем поиска по сетке изучается эффект применения разных значений n в методе актора-критика с оценкой преимущества по отдаче за n шагов.

Файл spec для эксперимента является расширением листинга 6.7 с помощью добавления спецификации поиска для num_step_returns, как показано в листинге 6.13.

Заметьте, что теперь мы используем другую среду, Breakout, которая немного сложнее, чем Pong. В строках 4 и 7 указано изменение среды. В строке 19 определен поиск по списку num_step_returns значений *n*. Полный файл spec имеется в SLM Lab в slm_lab/spec/experimental/a2c/a2c_nstep_n_search.json.

Листинг 6.13. Файл spec для A2C по отдаче за n шагов со спецификацией поиска для разных значений n num_step_returns

```
 1 # slm_lab/spec/experimental/a2c/a2c_nstep_n_search.json
 2
 3 {
 4     "a2c_nstep_breakout": {
 5         ...
 6         "env": [{
 7             "name": "BreakoutNoFrameskip-v4",
 8             "frame_op": "concat",
 9             "frame_op_len": 4,
10             "reward_scale": "sign",
11             "num_envs": 16,
12             "max_t": null,
13             "max_frame": 1e7
14         }],
15         ...
16         "search": {
17             "agent": [{
18                 "algorithm": {
19                     "num_step_returns__grid_search": [1, 3, 5, 7, 9, 11]
20                 }
21             }]
22         }
23     }
24 }
```

Для запуска эксперимента в SLM Lab воспользуйтесь командами из листинга 6.14.

Листинг 6.14. Запуск эксперимента с поиском по разным количествам шагов n для отдачи за n шагов в соответствии с файлом spec

```
1 conda activate lab
2 python run_lab.py slm_lab/spec/experimental/a2c/a2c_nstep_n_search.json
  ➥ a2c_nstep_breakout search
```

Будет запущен эксперимент Experiment, который породит шесть испытаний Trial, все с разными значениями num_step_returns, подставляемыми в первоначальный файл spec для актора-критика. В каждом Trial будут запущены по четыре сессии Session для получения среднего значения. Графики результатов множества испытаний показаны на рис. 6.5.

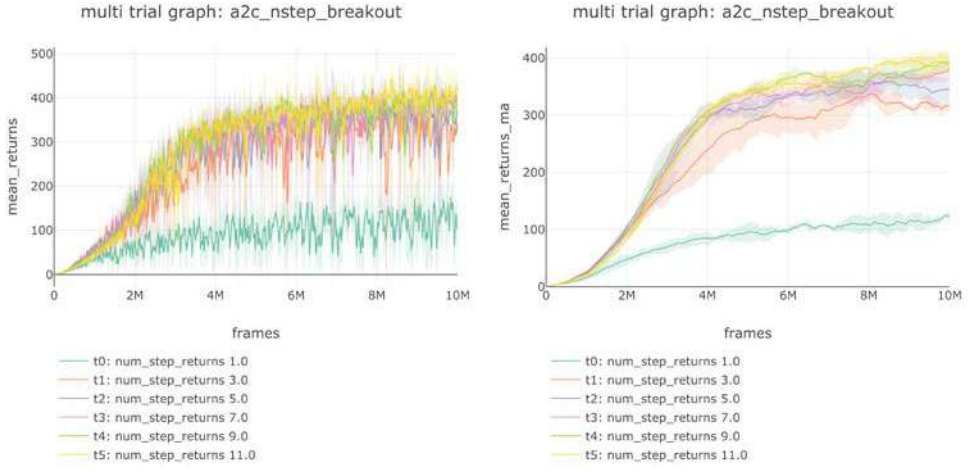


Рис. 6.5. Влияние разного числа шагов n при оценке преимущества по отдаче за n шагов на метод актора-критика в среде Breakout. При большем количестве шагов n производительность лучше

На рис. 6.5 показано влияние разного числа шагов при оценке преимущества по отдаче за n шагов на метод актора-критика в среде Breakout. При большем количестве шагов n производительность лучше. Можно также видеть, что n — не слишком чувствительный гиперпараметр. При $n = 1$ отдача за n шагов сводится к оценке отдачи по методу временных разностей, что дает наихудшую производительность в данном эксперименте.

6.7.2. Эксперимент по выявлению влияния λ в GAE

Как вы помните, GAE — это экспоненциальное средневзвешенное значение всех оценок преимущества по отдаче за n шагов, и чем больше уменьшающий коэффициент λ, тем выше дисперсия оценки. Оптимальное значение λ — это гиперпараметр, который настраивается под конкретную задачу.

В этом эксперименте рассматривается влияние применения разных значений λ на метод актора-критика с GAE путем выполнения поиска по сетке. Файл `spec` для эксперимента является расширением листинга 6.9 с добавлением спецификации поиска для `lam`, как показано в листинге 6.15.

В качестве среды мы также будем использовать Breakout. Изменение среды указано в строках 4 и 7. В строке 19 задается поиск по списку `lam` значений λ. Полный файл `spec` имеется в SLM Lab в `slm_lab/spec/experimental/a2c/a2c_gae_lam_search.json`.

Листинг 6.15. Файл spec для метода актора-критика со спецификацией поиска по разным значениям λ в GAE, lam

```
# slm_lab/spec/experimental/a2c/a2c_gae_lam_search.json

{
    "a2c_gae_breakout": {
        ...
        "env": [{
            "name": "BreakoutNoFrameskip-v4",
            "frame_op": "concat",
            "frame_op_len": 4,
            "reward_scale": "sign",
            "num_envs": 16,
            "max_t": null,
            "max_frame": 1e7
        }],
        ...
        "search": {
            "agent": [{
                "algorithm": {
                    "lam__grid_search": [0.50, 0.70, 0.90, 0.95, 0.97, 0.99]
                }
            }]
        }
    }
}
```

Используйте команды, приведенные в листинге 6.16, для запуска эксперимента.

Листинг 6.16. Запуск эксперимента с поиском по разным значениям λ в GAE в соответствии с файлом spec

```
conda activate lab
python run_lab.py slm_lab/spec/experimental/a2c/a2c_gae_lam_search.json
  ➥ a2c_gae_breakout search
```

Будет запущен эксперимент `Experiment`, который породит шесть испытаний `Trial`. Для каждого из них в исходный файл `spec` для актора-критика будет подставлено свое значение `lam`. В каждом `Trial` будут запущены по четыре сессии `Session` для получения средних результатов. Графики для множества испытаний показаны на рис. 6.6.

Как видно из рис. 6.6, при $\lambda = 0{,}97$ достигается наилучшая производительность со счетом в эпизодах, близким к 400, чуть меньшую производительность дают $\lambda = 0{,}90$ и 0,99. Данный эксперимент продемонстрировал, что λ — не очень чувствительный гиперпараметр. Отклонение значения λ от оптимального слабо влияет на производительность. Например, $\lambda = 0{,}70$ все еще дает хороший результат, но при $\lambda = 0{,}50$ производительность падает.

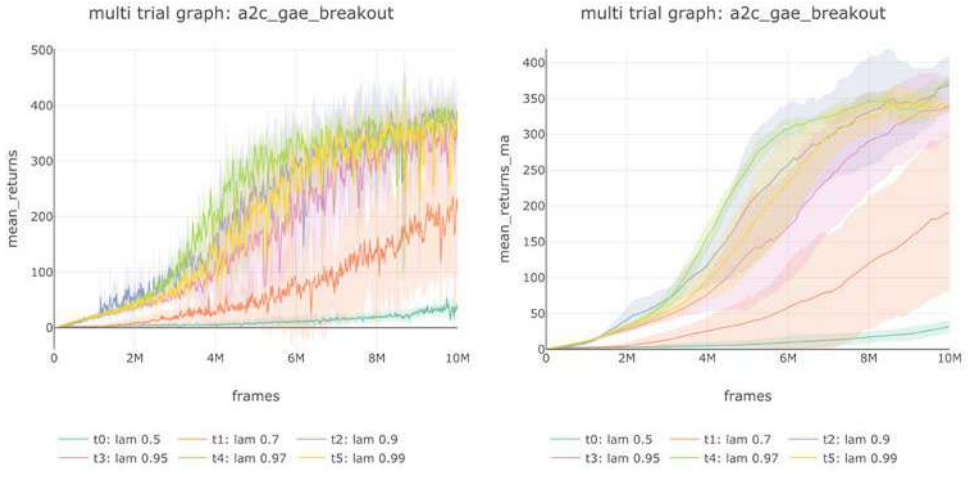


Рис. 6.6. Влияние разных значений λ в GAE на метод актора-критика в среде Breakout. При λ = 0,97 достигается наилучшая производительность, следом идут значения 0,90 и 0,99

6.8. Резюме

В этой главе мы познакомились с алгоритмами актора-критика, которые состоят из двух компонентов — актора и критика. Актор настраивает стратегию π, а критик — функцию полезности V^{π}. Настроенная $\hat{V}^{\pi}$ в сочетании с действительными вознаграждениями используется для порождения подкрепляющего сигнала для стратегии. Таким сигналом зачастую является функция преимущества.

В алгоритмах актора-критика объединены идеи из методов, основанных на стратегиях и полезностях, которые были представлены в предыдущих главах. Оптимизация актора аналогична REINFORCE, но с настроенным подкрепляющим сигналом вместо оценки по методу Монте-Карло, порожденной из текущей траектории вознаграждений. Оптимизация критика схожа с DQN в том, что в ней используется метод временных разностей с бутстрэппингом.

В этой главе были рассмотрены два способа оценки функции преимущества: по отдаче за n шагов и GAE. Оба метода позволяют пользователям контролировать долю смещения и дисперсии в функции преимущества, выбирая, насколько весомой будет действительная траектория вознаграждений по сравнению с оценкой функции полезности $\hat{V}^{\pi}$. Оценка преимущества по отдаче за n шагов жестко ограничивается n, тогда как GAE мягко регулируется параметром λ.

В конце главы мы обсудили два подхода к проектированию архитектуры нейронной сети для алгоритмов актора-критика: с общими параметрами или с полным разделением сетей актора и критика.

6.9. Рекомендуемая литература

- *Barto A. G., Sutton R. S., Anderson C. W.* Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems. 1983 [11].
- *Schulman J., Moritz P., Levine S., Jordan M. I., Abbeel P.* High-Dimensional Continuous Control Using Generalized Advantage Estimation. 2015 [123].
- *Schulman J., Levine S., Moritz P., Jordan M. I., Abbeel P.* Trust Region Policy Optimization. 2015 [122].

6.10. Историческая справка

Большая часть концепций, лежащих в основе современных алгоритмов актора-критика, была выдвинута в 1983 году Саттоном, Барто и Андерсоном в статье *Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems* [11]. В ней озвучивалась идея совместного обучения двух взаимодействующих модулей: компонента стратегии или актора, на который в этой главе ссылались как на элемент ассоциативного поиска (Associative Search Element, ASE), и критика, названного элементом адаптивной критики (Adaptive Critic Element, ACE). Алгоритм был продиктован нейробиологией — областью, которая часто вдохновляет исследователей методов глубокого обучения. Например, идея сверточных нейронных сетей изначально была позаимствована из строения зрительной коры головного мозга человека [43, 71]. Саттон с коллегами аргументировали то, что моделирование такой сложной структуры, как нейрон, на уровне элементов, подобных логическим вентилям, нецелесообразно. Они предложили ASE/ACE в качестве первоначального шага в сторону построения систем, включающих сети из сложных обучаемых подэлементов. Они надеялись на то, что переход от простых базовых элементов к сложным поможет решить значительно более трудные задачи.

Название ACE, вероятно, было навеяно Видроу и др. [146], которые за 10 лет до того, в 1973 году, использовали слова «обучение с критиком», чтобы отличать RL от обучения с учителем (supervised learning, SL) [11][1]. Тем не менее применение обученного критика для улучшения подкрепляющего сигнала, передаваемого в другую часть системы, было нововведением. Саттон с коллегами пояснили предназначение ACE очень ясно: «Он [ACE] адаптивно развивает функцию оценки, которая более

[1] В SL учитель дает вам правильные ответы для каждой точки данных, тогда как в RL агента лишь критикуют с помощью функции вознаграждений. Критика аналогична тому, что вам говорят: то, что вы сделали, было хорошо, плохо или приемлемо. Вам не говорят, что вы должны были выполнить какое-то определенное действие.

информативна, чем доступная напрямую из среды обучающейся системы. Это снижает неопределенность при обучении ASE [11]»[1].

Присваивание коэффициентов доверия и разреженные сигналы вознаграждения — две наиболее важные проблемы в RL. ACE решает задачу присваивания коэффициентов в RL путем применения критика к преобразованию потенциально разреженного вознаграждения в более информативный плотный подкрепляющий сигнал. После выхода этой статьи значительная часть исследований была сфокусирована на развитии лучших критиков.

Функция преимущества была впервые упомянута Бэйрдом в 1983 году [9]. Однако определение преимущества как $A^{\pi}(s, a) = Q^{\pi}(s, a) - V^{\pi}(s)$ дано Саттоном с соавторами в 1999 году [133].

После публикации Мнихом и другими в 2016 году алгоритма асинхронного актора-критика (Asynchronous Advantage Actor-Critic, A3C) наблюдался рост интереса к таким алгоритмам. В этой статье был предложен простой масштабируемый метод параллелизации обучения в алгоритмах обучения с подкреплением с помощью асинхронного градиентного спуска. Мы рассмотрим этот метод в главе 8.

[1] Если вы хотите больше узнать об этом, то вам стоит прочитать оригинальную статью. Она написана понятным языком, и в ней хорошо объясняются задача назначения вознаграждений в RL и разница между обучением с подкреплением и обучением с учителем.

7 Оптимизация ближайшей стратегии

Одна из трудностей, возникающих при обучении агентов с помощью алгоритмов градиента стратегии, заключается в том, что они склонны к *резкому падению производительности*, когда агент неожиданно начинает действовать плохо. Последствия этого трудно устранить, так как агент начинает порождать плохие траектории, которые затем используются для дальнейшей настройки стратегии. Кроме того, как мы видели, основанные на стратегии алгоритмы неэффективны с точки зрения качества выборки, так как в них невозможно повторное использование данных.

Представленная Шульманом и другими оптимизация ближайшей стратегии (Proximal Policy Optimization, PPO) — класс алгоритмов оптимизации, решающих две эти проблемы. В основе PPO лежит идея о суррогатной целевой функции, которая позволяет избежать *падения производительности* за счет гарантированного монотонного роста производительности. Еще одно преимущество такой целевой функции — повторное использование данных, полученных по отложенному опыту.

PPO может быть задействована для расширения REINFORCE или актора-критика путем замещения изначальной целевой функции $J(\pi_\theta)$ преобразованной целевой функцией PPO. Это преобразование обеспечивает более устойчивое и эффективное с точки зрения качества выборки обучение.

В этой главе в подразделе 7.1.1 обсуждаются случаи резкого падения производительности. Затем в подразделе 7.1.2 рассматривается решение этой проблемы с помощью подхода монотонного улучшения. Он выражается в преобразовании целевой функции, выраженной через градиент стратегии, в суррогатную целевую функцию.

После введения в теоретические основы перейдем к обсуждению алгоритма PPO в разделе 7.2. Затем в разделе 7.4 покажем реализацию PPO как расширения алгоритма актора-критика в SLM Lab.

7.1. Суррогатная целевая функция

В этом разделе вводится понятие суррогатной целевой функции для алгоритма PPO. Сначала в качестве обоснования этого обсуждается проблема резкого падения производительности. Затем рассматривается устранение проблемы преобразованием целевой функции, выраженной через градиент стратегии.

7.1.1. Падение производительности

Как мы видели при обсуждении алгоритмов градиента стратегии, стратегия π_θ оптимизируется путем изменения параметров θ с помощью градиента стратегии $\nabla_\theta J(\pi_\theta)$. Это непрямой подход, так как мы ищем оптимальную стратегию в пространстве стратегий, которым не можем управлять напрямую. Чтобы понять, почему так происходит, сначала нужно рассмотреть разницу между пространствами стратегий и параметров.

В ходе оптимизации поиск производится по последовательности стратегий π_1, π_2, $\pi_3 \ldots \pi_n$ из набора всех стратегий. Этот набор известен как *пространство стратегий*[1] Π:

$$\Pi = \{\pi_i\}. \tag{7.1}$$

Параметризировав стратегию π_θ, можно получить удобное выражение для пространства параметров θ. Каждое уникальное θ параметризирует экземпляр стратегии. *Пространство параметров* обозначается Θ:

$$\Theta = \left\{\theta \in \mathbb{R}^m\right\}, \tag{7.2}$$

где m — количество параметров.

Целевая функция $J(\pi_\theta)$ вычисляется с помощью траекторий, созданных стратегией из пространства стратегий $\pi_\theta \in \Pi$, но в действительности поиск оптимальной стратегии происходит в пространстве параметров путем отыскания правильных их значений $\theta \in \Theta$. То есть управление происходит в Θ, а результат получаем в Π. На практике количество шагов при обновлении параметров регулируется с помощью параметра скорости обучения α, как показано в уравнении (7.3):

$$\Delta\theta = \alpha\nabla_\theta J(\pi_\theta) \tag{7.3}$$

К сожалению, пространства стратегий и параметров не всегда конгруэнтны, а расстояния в обоих пространствах не обязательно сопоставимы. Возьмем две пары

[1] Следует отметить, что пространство стратегий может содержать бесконечно большое число стратегий.

параметров, скажем (θ_1, θ_2) и (θ_2, θ_3), и предположим, что в пространстве параметров расстояния между параметрами в одной паре в обоих случаях равны, то есть $d_\theta(\theta_1, \theta_2) = d_\theta(\theta_2, \theta_3)$. Однако расстояния для пар сопоставляемых им стратегий $\left(\pi_{\theta_1}, \pi_{\theta_2}\right)$ и $\left(\pi_{\theta_2}, \pi_{\theta_3}\right)$ не обязательно одинаковы. Таким образом,

$$d_\theta(\theta_1, \theta_2) = d_\theta(\theta_2, \theta_3) \Leftrightarrow d_\pi\left(\pi_{\theta_1}, \pi_{\theta_2}\right) = d_\pi\left(\pi_{\theta_2}, \pi_{\theta_3}\right). \quad (7.4)$$

Потенциально это представляет проблему по той причине, что становится труднее определить идеальный размер шага α для обновления параметров. Заранее не известно, насколько малому или большому размеру шага оно будет сопоставлено в пространстве стратегий Π. Если значение α слишком мало, то потребуется больше итераций и обучение будет длительным. Другой вариант — стратегия может застрять в каком-нибудь локальном максимуме. Если значение α слишком велико, то на соответствующем шаге в пространстве стратегий произойдет выход из окрестности хороших стратегий, что вызовет резкое падение производительности. Когда это случится, новая стратегия, которая намного хуже, будет использоваться для порождения плохих траекторий для следующего обновления и резко ухудшит стратегию на следующих итерациях. Поэтому так трудно исправить ситуацию после однократного падения производительности. На рис. 7.1 представлен пример такого быстрого падения производительности.

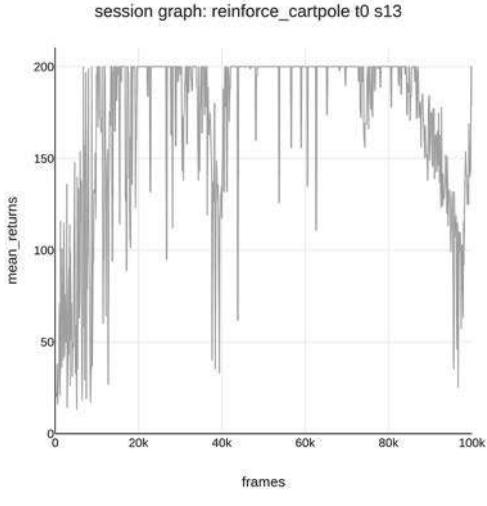


Рис. 7.1. График из SLM Lab, демонстрирующий пример падения производительности REINFORCE в CartPole. Обратите внимание: отдача (недисконтированная, то есть полное вознаграждение за эпизод) достигает максимального значения 200 и затем резко снижается после 80 000 кадров

В принципе, при постоянном размере шага этой проблемы не избежать. В выражении для градиентного восхождения $\theta \leftarrow \theta + \alpha\nabla_\theta J(\pi_\theta)$ скорость обучения α, как правило, фиксирована или уменьшается, а меняется только направление $\nabla_\theta J(\pi_\theta)$.

Но поскольку сопоставление неизометрично, одному и тому же постоянному размеру шага также могут быть сопоставлены разные расстояния в П.

Выбор оптимального значения α будет сильно зависеть от того, в какой части пространства стратегий П находится текущая стратегия π_θ и как происходит отображение параметра θ в окрестность π_θ в П из его собственной окрестности в Θ. В идеальном случае алгоритм должен адаптивно изменять размер шага на основе этих факторов.

Для определения размера шага, адаптивно изменяющегося в зависимости от того, как конкретное обновление в пространстве параметров влияет на пространство стратегий, сначала нужно найти способ измерения разницы между производительностью двух стратегий.

7.1.2. Преобразование целевой функции

В этом разделе вводится определение *соотношения стратегий по относительной эффективности*, по которому измеряется разница между производительностью двух стратегий. Затем будет показано, как это может быть использовано для преобразования целевой функции, выраженной в градиенте стратегии, чтобы обеспечить монотонное улучшение производительности в ходе оптимизации.

Поскольку проблема связана с размером шага обучения, интуитивно понятно, что можно попробовать ввести константу, которая привяжет его к некоему безопасному размеру, чтобы предотвратить падение производительности. Применение правильной константы позволяет вывести то, что известно как *подход монотонного улучшения*[1].

Во-первых, пусть дана стратегия π и пусть ее следующая итерация (после обновления параметров) будет π'. Можно определить *соотношение стратегий по относительной эффективности* как разность между их целевыми функциями (уравнение (7.5)). Отметим, что значение преимущества $A^\pi(s_t, a_t)$ всегда вычисляется по старой стратегии π:

$$J(\pi') - J(\pi) = \mathbb{E}_{\tau \sim \pi'}\left[\sum_{t=0}^{T} \gamma^t A^\pi(s_t, a_t)\right]. \tag{7.5}$$

[1] Вывод, приведенный в этом разделе, является адаптированной и развернутой версией вывода из замечательной лекции Deep Reinforcement Learning, прочитанной Сергеем Левиным в институте в Беркли осенью 2017 года. Некоторые из наиболее сложных доказательств будут опущены, поскольку они выходят за рамки этой книги, но любопытный читатель может найти их в оригинальном конспекте лекции номер 13 Advanced Policy Gradient Methods [78] Джошуа Ачьяма. В этой книге используется почти такая же нотация, как и в конспекте, так что их будет легко читать и сравнивать. Кроме того, запись лекции, в которой приводится больше деталей, чем в конспекте, доступна на YouTube под заголовком CS294-112 10/11/17 (https://youtu.be/ycCtmp4hcUs).

Примечание 7.1. Вывод соотношения стратегий по относительной эффективности

Здесь приводится вывод уравнения (7.5) для соотношения стратегий по относительной эффективности.

Для начала перепишем значение преимущества в подходящем для нашего вывода виде (уравнение (7.6)). Член Q развернем в математическое ожидание первого вознаграждения и ожидаемые V-значения во всех возможных последующих состояниях. Это математическое ожидание необходимо для учета стохастических переходов среды, которые не зависят от конкретной стратегии:

$$A^{\pi}(s_t, a_t) = Q^{\pi}(s_t, a_t) - V^{\pi}(s_t) = \mathbb{E}_{s_{t+1}, r_t \sim p(s_{t+1}, r_t | s_t, a_t)}\left[r_t + \gamma V^{\pi}(s_{t+1})\right] - V^{\pi}(s_t). \quad (7.6)$$

Используем это выражение для преобразования правой части уравнения (7.5) следующим образом:

$$\mathbb{E}_{\tau \sim \pi'}\left[\sum_{t \geq 0} \gamma^t A^{\pi}(s_t, a_t)\right] = \quad (7.7)$$

$$= \mathbb{E}_{\tau \sim \pi'}\left[\sum_{t \geq 0} \gamma^t \left(\mathbb{E}_{s_{t+1}, r_t \sim p(s_{t+1}, r_t | s_t, a_t)}\left[r_t + \gamma V^{\pi}(s_{t+1})\right] - V^{\pi}(s_t)\right)\right] = \quad (7.8)$$

$$= \mathbb{E}_{\tau \sim \pi'}\left[\sum_{t \geq 0} \gamma^t \left(r_t + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t)\right)\right] = \quad (7.9)$$

$$= \mathbb{E}_{\tau \sim \pi'}\left[\sum_{t \geq 0} \gamma^t r_t + \sum_{t \geq 0} \gamma^{t+1} V^{\pi}(s_{t+1}) - \sum_{t \geq 0} \gamma^t V^{\pi}(s_t)\right] = \quad (7.10)$$

$$= \mathbb{E}_{\tau \sim \pi'}\left[\sum_{t \geq 0} \gamma^t r_t\right] + \mathbb{E}_{\tau \sim \pi'}\left[\sum_{t \geq 0} \gamma^{t+1} V^{\pi}(s_{t-1}) - \sum_{t \geq 0} \gamma^t V^{\pi}(s_t)\right] = \quad (7.11)$$

$$= J(\pi') + \mathbb{E}_{\tau \sim \pi'}\left[\sum_{t \geq 1} \gamma^t V^{\pi}(s_t) - \sum_{t \geq 0} \gamma^t V^{\pi}(s_t)\right] = \quad (7.12)$$

$$= J(\pi') - \mathbb{E}_{\tau \sim \pi'}\left[V^{\pi}(s_0)\right] = \quad (7.13)$$

$$= J(\pi') - \mathbb{E}_{\tau \sim \pi'}\left[J(\pi)\right] = \quad (7.14)$$

$$= J(\pi') - J(\pi). \quad (7.15)$$

Сначала переформулируем уравнение (7.5), затем подставим определение значения преимущества из уравнения (7.6) и получим уравнение (7.8). На следующем шаге заметим, что внешнее математическое ожидание $\mathbb{E}_{\tau \sim \pi'}$ фактически содержит ожидаемые действия по стратегии ($\mathbb{E}_{a_t \sim \pi'}$) и ожидаемые состояния и вознаграждения из функции переходов ($\mathbb{E}_{s_{t+1}, r_t \sim p(s_{t+1}, r_t | s_t, a_t)}$). Следовательно, можно включить внутреннее математическое ожидание во внешнее, поскольку у них есть перекрывающиеся составляющие, что даст уравнение (7.9).

Затем в уравнении (7.10) разложим сумму на подсуммы. Ввиду линейности математического ожидания выделим в уравнении (7.11) первый член, который попросту является новой целевой функцией $J(\pi')$. Это даст нам уравнение (7.12). Теперь произведем сдвиг индекса в первой сумме в выражении для математического

ожидания, заменив индекс $t + 1$ на индекс t, который начинается с 1. Теперь обе суммы выглядят идентично, за исключением индексов, поэтому их можно сократить, оставив лишь член при $t = 0$, которым является $V^{\pi}(s_0)$ в уравнении (7.13). Функция полезности в начальном состоянии — это просто целевая функция $J(\pi)$, так как $V^{\pi}(s_0) = \mathbb{E}_{\tau\sim\pi}\left[\sum_{t\geq 0}\gamma^t r_t\right] = J(\pi)$. Так мы получаем уравнение (7.14). Наконец целевая функция стратегии π не зависит от новой стратегии π', и можно опустить математическое ожидание, получив $J(\pi') - J(\pi)$.

Соотношение стратегий по относительной эффективности $J(\pi') - J(\pi)$ служит мерой улучшения производительности. Если значение разности положительное, то более новая стратегия π' лучше, чем π. В процессе итерации по стратегиям следует так выбрать новую стратегию π', чтобы максимизировать эту разность. С учетом сказанного максимизация целевой функции $J(\pi')$ равнозначна максимизации данного соотношения. И то и другое можно выполнить с помощью градиентного восхождения:

$$\max_{\pi'} J(\pi') \Leftrightarrow \max_{\pi'}\left(J(\pi') - J(\pi)\right). \tag{7.16}$$

Такое ограничение целевой функции также подразумевает, что на каждой итерации по стратегиям следует обеспечить неотрицательное (монотонное) улучшение. То есть $J(\pi') - J(\pi) \geq 0$, так как в худшем случае можно просто принять $\pi' = \pi$, что даст отсутствие улучшений. Это позволит избежать падения производительности на протяжении обучения, и именно это нам нужно.

Однако существует ограничение, не позволяющее использовать данное соотношение в качестве целевой функции. Заметьте, что в выражении $\mathbb{E}_{\tau\sim\pi'}\left[\sum_{t\geq 0} A^{\pi}(s_t, a_t)\right]$ для вычисления математического ожидания требуется выбирать траектории по новой стратегии π' для выполнения обновления, но π' недоступна до обновления. Чтобы разрешить этот парадокс, нужно найти формулировку, которая даст возможность применять доступную старую стратегию π.

С этой целью можно предположить, что успешные стратегии π и π' довольно близки (мерой этого может служить малое расстояние Кульбака — Лейблера), значит, схожи и распределения состояний для них. Тогда уравнение (7.5) можно аппроксимировать с помощью траекторий из старой стратегии $\tau \sim \pi$, скорректировав их *весами из выборки по значимости*[1] $\dfrac{\pi'(a_t \mid s_t)}{\pi(a_t \mid s_t)}$. Значения отдачи, порожденные с помощью π, корректируются умножением их на отношение вероятностей действий для двух успешных стратегий, π и π'. Вознаграждения, связанные с действиями, более вероятными при новой стратегии π', будут более весомыми, а вознагражде-

[1] Выборка по значимости — это метод оценки неизвестного распределения по данным, выбранными из известного распределения. Полный вывод уравнения (7.5) выходит за рамки этой книги, но его можно найти в лекции 13 Advanced Policy Gradient Methods [78].

ния, связанные с менее вероятными при π' действиями, — менее весомыми. Данная аппроксимация приведена в уравнении

$$J(\pi') - J(\pi) = \mathbb{E}_{\tau \sim \pi'}\left[\sum_{t \geq 0} A^{\pi}(s_t, a_t)\right] \approx$$

$$\approx \mathbb{E}_{\tau \sim \pi}\left[\sum_{t \geq 0} A^{\pi}(s_t, a_t)\frac{\pi'(a_t \mid s_t)}{\pi(a_t \mid s_t)}\right] = J_{\pi}^{\mathrm{CPI}}(\pi'). \tag{7.17}$$

Новая целевая функция в правой части уравнения (7.17) называется *суррогатной целевой функцией*, так как содержит соотношение между новой и старой стратегиями, π' и π. Верхний индекс CPI расшифровывается как conservative policy iteration — консервативная итерация по стратегиям.

Теперь, чтобы воспользоваться новой целевой функцией в алгоритме градиента стратегии, нужно проверить, продолжает ли выполняться градиентное восхождение по стратегии при *оптимизации с помощью этой целевой функции*. К счастью, можно показать, что градиент суррогатной целевой функции равен градиенту по стратегии:

$$\nabla_{\theta} J_{\theta_{old}}^{\mathrm{CPI}}(\theta)\big|_{\theta_{old}} = \nabla_{\theta} J(\pi_{\theta})\big|_{\theta_{old}}. \tag{7.18}$$

Вывод приведен в примечании 7.2.

Примечание 7.2. Вывод суррогатной целевой функции

Здесь будет показано, что градиент суррогатной целевой функции равен градиенту стратегии, как утверждает уравнение (7.18).

Начнем с вычисления градиента $J_{\pi}^{\mathrm{CPI}}(\pi')$. Для этого запишем переменную θ в явном виде. В частности, обозначим новую стратегию π' как переменную π_{θ}, а старую стратегию π — как постоянную $\pi_{\theta_{old}}$, при которой вычисляется градиент, что обозначено как $|_{\theta_{old}}$. Это необходимо ввиду явной зависимости $J_{\pi}^{\mathrm{CPI}}(\pi')$ от π, которую нельзя опустить:

$$\nabla_{\theta} J_{\theta_{old}}^{\mathrm{CPI}}(\theta)\big|_{\theta_{old}} = \nabla_{\theta}\mathbb{E}_{\tau \sim \pi_{\theta_{old}}}\left[\sum_{t \geq 0} A^{\pi_{\theta_{old}}}(s_t, a_t)\frac{\pi_{\theta}(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)}\right]\Big|_{\theta_{old}} = \tag{7.19}$$

$$= \mathbb{E}_{\tau \sim \pi_{\theta_{old}}}\left[\sum_{t \geq 0} A^{\pi_{\theta_{old}}}(s_t, a_t)\frac{\nabla_{\theta}\pi_{\theta}(a_t \mid s_t)\big|_{\theta_{old}}}{\pi_{\theta_{old}}(a_t \mid s_t)}\right] = \tag{7.20}$$

$$= \mathbb{E}_{\tau \sim \pi_{\theta_{old}}}\left[\sum_{t \geq 0} A^{\pi_{\theta_{old}}}(s_t, a_t)\frac{\nabla_{\theta}\pi_{\theta}(a_t \mid s_t)\big|_{\theta_{old}}}{\pi_{\theta}(a_t \mid s_t)\big|_{\theta_{old}}}\right] = \tag{7.21}$$

$$= \mathbb{E}_{\tau \sim \pi_{\theta_{old}}}\left[\sum_{t \geq 0} A^{\pi_{\theta_{old}}}(s_t, a_t)\nabla_{\theta}\log\pi_{\theta}(a_t \mid s_t)\big|_{\theta_{old}}\right] = \tag{7.22}$$

$$= \nabla_{\theta} J(\pi_{\theta})\big|_{\theta_{old}}. \tag{7.23}$$

Сначала вычислим градиент при старой стратегии суррогатной целевой функции из уравнения (7.17). Из-за того что переменная θ содержится только в верхней части

отношения, туда можно внести знак градиента и получить уравнение (7.20). Теперь заметим, что нижняя часть отношения $\pi_{\theta_{old}}(a_t \mid s_t)$ — это не что иное, как запись π_θ, вычисленной при $\theta = \theta_{old}$, таким образом получаем уравнение (7.21). Наконец, вспомним прием из уравнения (2.14), с помощью которого мы выводили градиент стратегии, $\nabla_\theta \log p(x \mid \theta) = \dfrac{\nabla_\theta p(x \mid \theta)}{p(x \mid \theta)}$. Воспользуемся табличной производной логарифма, чтобы переписать отношение и прийти к уравнению (7.22). Мы получили просто градиент стратегии $\nabla_\theta J(\pi_\theta)$. Отсюда вытекает:

$$\nabla_\theta J^{CPI}_{\theta_{old}}(\theta)\big|_{\theta_{old}} = \nabla_\theta J(\pi_\theta)\big|_{\theta_{old}}. \tag{7.24}$$

Уравнение (7.24) говорит о том, что градиент суррогатной целевой функции равен градиенту стратегии. Это гарантия того, что при оптимизации с суррогатной целевой функцией продолжает выполняться градиентное восхождение по стратегиям. Это полезно еще и потому, что улучшение производительности теперь может быть измерено напрямую и максимизация суррогатной целевой функции означает максимизацию производительности. Кроме того, теперь нам известно, что в уравнении (7.17) $J^{CPI}_\pi(\pi')$ является линейной аппроксимацией $J(\pi') - J(\pi)$, так как их производные первого порядка (градиенты) равны.

Прежде чем принять $J^{CPI}_\pi(\pi')$ в качестве новой целевой функции для алгоритма градиента стратегии, осталось удовлетворить одно последнее требование. Так как $J^{CPI}_\pi(\pi')$ — это лишь приближение для $J(\pi') - J(\pi)$, она может иметь погрешность. Тем не менее нужно гарантировать, что $J(\pi') - J(\pi) \geq 0$, если $J^{CPI}_\pi(\pi') \approx J(\pi') - J(\pi)$. Следовательно, нужно понять, какова погрешность этого приближения.

Если между успешными стратегиями π и π' наблюдается значительное сходство, что измеряется расстоянием Кульбака — Лейблера, то одна из них может быть записана как ограничение *относительной границы производительности стратегии*[1]. Сделаем это, выразив абсолютную погрешность через разность между новой целевой функцией $J(\pi')$ и оценкой ее улучшения $J(\pi) + J^{CPI}_\pi(\pi')$. Тогда можно задать границы погрешности через расстояние Кульбака — Лейблера между π и π':

$$\left|\left(J(\pi') - J(\pi)\right) - J^{CPI}_\pi(\pi')\right| \leq C\sqrt{\mathbb{E}_t\left[\mathrm{KL}\left(\pi'(a_t \mid s_t) \,\|\, \pi(a_t \mid s_t)\right)\right]}, \tag{7.25}$$

где C — константа, которую нужно выбирать, а KL — расстояние Кульбака — Лейблера[2]. Уравнение (7.25) показывает, что если распределения для успешных стра-

---

[1] Это понятие было введено в 2017 году в статье Ачьяма и др. *Constrained Policy Optimization* [2].

[2] Расстояние Кульбака — Лейблера между двумя распределениями $p(x)$, $q(x)$ при случайном $x \in X$ определяется как $\mathrm{KL}(p(x) \,\|\, q(x)) = \sum_{x \in X} p(x) \log \dfrac{p(x)}{q(x)}$ для дискретных распределений или $\mathrm{KL}(p(x) \,\|\, q(x)) = \int_{x \in X} p(x) \log \dfrac{p(x)}{q(x)} dx$ для непрерывных распределений.

тегий π и π' близки и расстояние Кульбака — Лейблера в правой части выражения мало, то член погрешности в левой части мал[1]. При малом значении погрешности $J_\pi^{\text{CPI}}(\pi')$ является хорошим приближением для $J(\pi') - J(\pi)$.

Воспользовавшись этим, можно легко прийти к тому результату, который нам нужен, а именно $J(\pi') - J(\pi) \geq 0$. Это носит название *подхода монотонного улучшения* [78], так как обеспечивает положительное или по крайней мере нулевое улучшение на каждой итерации по стратегиям, никогда не допуская ухудшения. С этой целью сначала раскроем модуль в уравнении (7.25) и рассмотрим нижнюю границу:

$$J(\pi') - J(\pi) \geq J_\pi^{\text{CPI}}(\pi') - C\sqrt{\mathbb{E}_t\left[\text{KL}\left(\pi'(a_t \mid s_t) \| \pi(a_t \mid s_t)\right)\right]}. \tag{7.26}$$

Теперь перейдем к наихудшему случаю на шаге итерации по стратегиям. Рассмотрим все варианты новой стратегии π', в которую также включена старая стратегия π (размер шага обновления параметров равен нулю). Если кандидаты с лучшей производительностью отсутствуют, просто устанавливаем $\pi' = \pi$ и не выполняем обновление на этой итерации. Если это так, то уравнение (7.17) говорит, что $J_\pi^{\text{CPI}}(\pi) = \mathbb{E}_{t\sim\pi}\left[\sum_{t\geq 0} A^\pi(s_t, a_t)\frac{\pi(a_t \mid s_t)}{\pi(a_t \mid s_t)}\right] = 0$, поскольку у стратегии нет ожидаемого преимущества над самой собой. Для тождественных стратегий расстояние Кульбака — Лейблера $\text{KL}(\pi \| \pi) = 0$.

Из уравнения (7.26) ясно, что для принятия изменения в стратегии оценка улучшения стратегии $J_\pi^{\text{CPI}}(\pi')$ должна быть больше, чем максимальная погрешность $C\sqrt{\mathbb{E}_t\left[\text{KL}\left(\pi'(a_t \mid s_t) \| \pi(a_t \mid s_t)\right)\right]}$.

Если в задачу оптимизации в качестве штрафной функции ввести границу погрешности, то можно гарантировать монотонное улучшение стратегии. Тогда задача оптимизации примет следующий вид:

$$\arg\max_{\pi'}\left(J_\pi^{\text{CPI}}(\pi') - C\sqrt{\mathbb{E}_t\left[\text{KL}\left(\pi'(a_t \mid s_t) \| \pi(a_t \mid s_t)\right)\right]}\right) \Rightarrow$$
$$\Rightarrow J(\pi') - J(\pi) \geq 0. \tag{7.27}$$

Полученное выражение удовлетворяет нашему последнему требованию. Это позволяет избежать падения производительности, которое может произойти при использовании первоначальной целевой функции $J(\pi)$. Следует упомянуть одну важную особенность — монотонное улучшение не гарантирует сходимости к оптимальной стратегии π^*. Например, процесс оптимизации стратегии по-прежнему может застрять в локальном максимуме, в котором на каждой итерации по стратегиям не происходит улучшения, то есть $J(\pi') - J(\pi) = 0$. Обеспечение сходимости остается трудным открытым вопросом.

---

[1] Полный вывод можно найти в лекции 13 Advanced Policy Gradient Methods [78].

В качестве последнего шага рассмотрим практическую реализацию задачи оптимизации из уравнения (7.27). Одна из идей — ввести прямое ограничение расстояния Кульбака — Лейблера, как показано в уравнении:

$$\mathbb{E}_t\left[\mathrm{KL}\left(\pi'(a_t \mid s_t) \| \pi(a_t \mid s_t)\right)\right] \le \delta. \tag{7.28}$$

Параметр δ задает границу наибольшего значения расстояния Кульбака — Лейблера, эффективно ограничивая то, насколько новая стратегия π' может расходиться со старой стратегией π. Будут рассматриваться лишь кандидаты из близкой окрестности π в пространстве стратегий. Эта окрестность называется *доверительной областью*, а уравнение (7.28) — *границей доверительной области*. Заметим, что δ — это гиперпараметр, который нужно настраивать.

Граница $\mathbb{E}_t\left[\mathrm{KL}\left(\pi'(a_t \mid s_t) \| \pi(a_t \mid s_t)\right)\right]$ записана как математическое ожидание по одному временному шагу t, поэтому целесообразно переписать целевую функцию $J_\pi^{\mathrm{CPI}}(\pi')$ в такой же форме, как показано в уравнении (7.29). Кроме того, так как мы будем максимизировать целевую функцию относительно θ, выразим стратегии через θ (уравнение (7.30)). Новая стратегия записывается как $\pi' = \pi_\theta$, тогда как более старая стратегия с фиксированными параметрами — как $\pi = \pi_{\theta_{\text{old}}}$. Для значения преимущества, вычисленного по старой стратегии, используется также краткая форма записи $A^\pi(s_t, a_t) = A_t^{\pi_{\theta_{\text{old}}}}$:

$$J_\pi^{\mathrm{CPI}}(\pi') = \mathbb{E}_t\left[\frac{\pi'(a_t \mid s_t)}{\pi(a_t \mid s_t)} A^\pi(s_t, a_t)\right]; \tag{7.29}$$

$$J^{\mathrm{CPI}}(\theta) = \mathbb{E}_t\left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} A_t^{\pi_{\theta_{\text{old}}}}\right]. \tag{7.30}$$

Задача оптимизации стратегии по доверительной области, в которой объединены ограничение и суррогатная целевая функция, представлена в уравнении (7.31):

$$\max_\theta \mathbb{E}_t\left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} A_t^{\pi_{\theta_{\text{old}}}}\right] \text{ при } \mathbb{E}_t\left[\mathrm{KL}\left(\pi_\theta(a_t \mid s_t) \| \pi_{\theta_{\text{old}}}(a_t \mid s_t)\right)\right] \le \delta. \tag{7.31}$$

В конечном итоге $J_\pi^{\mathrm{CPI}}(\pi')$ — это линейное приближение $J(\pi') - J(\pi)$, так как его градиент равен градиенту стратегии. Это гарантирует монотонное улучшение с точностью до ограниченной погрешности. Для гарантированного улучшения с учетом данной потенциальной погрешности установим границу доверительной области, чтобы лимитировать разницу между новой и старой стратегиями. Предусмотрев невыход изменений в стратегии за пределы доверительной области, можно избежать падения производительности.

Для решения данной задачи оптимизации по доверительной области были предложены несколько алгоритмов. Вот некоторые из них: *естественный градиент по стратегиям* (Natural Policy Gradient, NPG) [63, 112, 113], *задача оптимизации по доверительной области* (Trust Region Policy Optimization, TRPO) [122]

и *ограниченная оптимизация стратегии* (Constrained Policy Optimization, CPO) [2]. В основе алгоритмов лежат довольно сложные теории, и их трудно реализовать. Их градиенты могут быть вычислительно затратными, а хорошее значение для δ тяжело подобрать. Эти алгоритмы выходят за рамки данной книги, но их недостатки послужили основанием для появления нашего следующего алгоритма — оптимизации ближайшей стратегии (Proximal Policy Optimization, PPO).

7.2. Оптимизация ближайшей стратегии

Статья *Proximal Policy Optimization Algorithms* [124] была опубликована в 2017 году Шульманом и др. PPO — легко реализуемый вычислительно малозатратный алгоритм без выбора δ. Благодаря этому он стал одним из самых популярных алгоритмов градиента стратегии.

PPO — это семейство алгоритмов, которые решают задачу оптимизации стратегии, ограниченную доверительной областью, с помощью простых и эффективных эвристик. Есть два варианта: первый основан на адаптируемой штрафной функции расстояния Кульбака — Лейблера, а второй — на усеченной целевой функции. Оба они представлены в данном разделе.

В первую очередь упростим суррогатную целевую функцию $J^{\text{CPI}}(\theta)$, записав $r_t(\theta) = \dfrac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)}$, как показано в уравнении (7.32). Для краткости также обозначим преимущество $A_t^{\pi_{\theta_{old}}}$ как A_t, так как известно, что значения преимущества всегда рассчитываются с помощью более старой стратегии $\pi_{\theta_{old}}$:

$$J^{\text{CPI}}(\theta) = \mathbb{E}_t\left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)} A_t^{\pi_{\theta_{old}}}\right] = \mathbb{E}_t\left[r_t(\theta) A_t\right]. \tag{7.32}$$

Первый вариант PPO называется *PPO с адаптивной штрафной функцией* расстояния Кульбака — Лейблера. Это преобразование ограничения расстояния Кульбака — Лейблера $\mathbb{E}_t\left[\mathrm{KL}\left(\pi_\theta(a_t \mid s_t) \| \pi_{\theta_{old}}(a_t \mid s_t)\right)\right] \le \delta$ в адаптивную штрафную функцию расстояния Кульбака — Лейблера, которая вычитается из значения преимущества, скорректированного весовым коэффициентом значимости. Математическое ожидание полученного выражения — новая целевая функция, которую нужно максимизировать:

$$J^{\text{KLpen}}(\theta) = \max_\theta \mathbb{E}_t\left[r_t(\theta) A_t - \beta \mathrm{KL}\left(\pi_\theta(a_t \mid s_t) \| \pi_{\theta_{old}}(a_t \mid s_t)\right)\right]. \tag{7.33}$$

Уравнение (7.33) известно как *суррогатная целевая функция со штрафом по расстоянию Кульбака — Лейблера*. β — адаптивный коэффициент, контролирующий размер штрафа по расстоянию Кульбака — Лейблера. Он выполняет ту же роль, что и уравнение (7.31) в управлении доверительной областью оптимизации.

Чем больше β, тем больше разница между π_θ и $\pi_{\theta_{old}}$. Чем меньше β, тем выше схожесть стратегий.

Одна из трудностей применения постоянного коэффициента в том, что из-за различия между характеристиками разных задач трудно подобрать одно значение, подходящее во всех случаях. Даже для одной задачи при итерации по стратегиям происходит смена ландшафта функции потерь, и значение β, которое раньше было рабочим, может в дальнейшем стать неподходящим, то есть его нужно адаптировать к этим изменениям.

В статье о PPO в качестве решения этой проблемы предлагается основанное на эвристике правило обновления β, которое позволяет адаптировать его с течением времени. Коэффициент β обновляется после каждого обновления стратегии, и на следующей итерации берется его новое значение. Правило обновления для β приведено в алгоритме 7.1. Оно может быть использовано в качестве процедуры в алгоритме PPO, который будет представлен позже в этом разделе.

Алгоритм 7.1. Адаптивный коэффициент штрафной функции расстояния Кульбака — Лейблера

Установить целевое значение δ_{tar} для матожидания
➥ расстояния Кульбака – Лейблера
Инициализировать β случайным значением
Использовать большое количество эпох стохастического градиентного спуска
➥ по мини-пакетам, оптимизировать суррогатную целевую функцию
➥ со штрафом по расстоянию Кульбака – Лейблера:
➥ $J^{KLpen}(\theta) = E_t\left[r_t(\theta)A_t - \beta KL\left(\pi_\theta(a_t \mid s_t) \,||\, \pi_{\theta_{old}}(a_t \mid s_t)\right)\right]$
Вычислить $\delta = E_t\left[KL\left(\pi_\theta(a_t \mid s_t) \,||\, \pi_{\theta_{old}}(a_t \mid s_t)\right)\right]$:
if $\delta < \delta_{tar}/1{,}5$ **then**
    $\beta \leftarrow \beta/2$
else if $\delta > \delta_{tar} \times 1{,}5$ **then**
    $\beta \leftarrow \beta \times 2$
else
    pass
end if

В конце каждой итерации вычисляем δ и сравниваем его с желаемым целевым δ_{tar}. Если δ меньше, чем δ_{tar} с некоторым запасом, то снижаем штраф по расстоянию Кульбака — Лейблера путем уменьшения β. Но если δ больше, чем δ_{tar} с некоторым запасом, увеличиваем штраф, повышая β. Конкретные значения для определения величины запаса и правил обновления для β подбираются опытным путем. Авторы выбрали 1,5 и 2 соответственно, но обнаружили, что алгоритм не очень чувствителен к этим значениям.

Кроме того, было замечено, что расстояние Кульбака — Лейблера периодически сильно отклоняется от целевого значения δ_{tar}, но β быстро подстраивается. Оптимальное значение δ_{tar} тоже нужно искать эмпирически.

Преимущество такого подхода — простота реализации. Тем не менее он не решает проблемы выбора целевого значения δ. Более того, он может быть вычислительно затратным из-за необходимости определения расстояния Кульбака — Лейблера.

PPO с усеченной суррогатной целевой функцией решает это отказом от ограничения расстояния Кульбака — Лейблера и применением более простого преобразования суррогатной целевой функции из уравнения (7.30):

$$J^{\text{clip}}(\theta) = \mathbb{E}_t\left[\min\left(r_t(\theta)A_t, \text{clip}\left(r_t(\theta), 1-\varepsilon, 1+\varepsilon\right)A_t\right)\right]. \tag{7.34}$$

Уравнение (7.34) известно как *усеченная суррогатная целевая функция*. Значение ε ограничивает окрестность, по которой производится усечение, $|r_t(\theta) - 1| \le \varepsilon$. Это настраиваемый гиперпараметр, который может уменьшаться в процессе обучения. Первый член выражения под знаком минимума, min (·), — это просто суррогатная целевая функция J^{CPI}. Второй член, $\text{clip}(r_t(\theta), 1 - \varepsilon, 1 + \varepsilon)A_t$, ограничивает область значений J^{CPI} между $(1 - \varepsilon)A_t$ и $(1 + \varepsilon)A_t$. Когда $r_t(\theta)$ находится в пределах интервала $[1 - \varepsilon, 1 + \varepsilon]$, оба члена под знаком минимума равны.

Такая целевая функция предотвращает обновления параметров, которые могут вызвать большие и рискованные изменения стратегии π_θ. В качестве количественной меры больших изменений стратегии применяется вероятностное отношение $r_t(\theta)$. Когда новая стратегия равноценна старой, $r_t(\theta) = r_t(\theta_{\text{old}}) = \dfrac{\pi_{\theta_{\text{new}}}(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} = 1$. Если новая стратегия отличается от старой, значение $r_t(\theta)$ отклоняется от 1.

Суть здесь в том, чтобы ограничить $r_t(\theta)$ до ε-окрестности $[1 - \varepsilon, 1 + \varepsilon]$. Обычно максимизация суррогатной целевой функции J^{CPI} без ограничений может способствовать большим обновлениям стратегии. Это вызвано тем, что одним из механизмов, посредством которого может быть улучшена производительность по целевой функции, являются большие изменения $r_t(\theta)$. Усечением целевой функции мы устраняем причины, порождающие большие обновления стратегии, которые вызвали бы выход $r_t(\theta)$ из ε-окрестности. Чтобы понять, почему это происходит, рассмотрим случай, когда $r_t(\theta)A_t$ принимает большие положительные значения при $A_t > 0$, $r_t(\theta) > 0$ либо $A_t < 0$, $r_t(\theta) < 0$.

При $A_t > 0$, $r_t(\theta) > 0$, если $r_t(\theta)$ становится намного больше 1, верхняя граница усечения $1 - \varepsilon$ применяется для ограничения сверху $r_t(\theta) \le 1 + \varepsilon$, следовательно, $J^{\text{clip}} \le (1 + \varepsilon)A_t$. С другой стороны, при $A_t < 0$, $r_t(\theta) < 0$, если $r_t(\theta)$ становится намного меньше 1, нижняя граница усечения $1 - \varepsilon$ снова применяется для ограничения сверху $J^{\text{clip}} \le (1 - \varepsilon)A_t$. При взятии минимума в уравнении (7.34) J^{clip} всегда ограничена сверху одним из двух способов. Кроме того, взятие минимума также подразумевает, что J^{clip} — пессимистическая нижняя граница начальной суррогатной целевой функции J^{CPI}. Таким образом, влияние $r_t(\theta)$ игнорируется при попытке улучшить целевую функцию более чем на εA_t, но всегда учитывается, когда из-за него целевая функция ухудшается.

Поскольку J^{clip} ограничена сверху, то нет стимула для больших обновлений, которые приводят к выходу $r_t(\theta)$ из окрестности $[1 - \varepsilon, 1 + \varepsilon]$. Следовательно, обновления стратегии безопасны.

Кроме того, данная целевая функция использует прецеденты, порожденные $\pi_{\theta_{\text{old}}}$, то есть целевая функция не зависит от текущей стратегии π_θ, за исключением коэффициентов значимости. Это дает нам основания для многократного повторного использования выбранных траекторий для обновления параметров. Таким образом повышается эффективность выборки алгоритма. Обратите внимание на то, что целевая функция зависит от $\pi_{\theta_{\text{old}}}$, которая обновляется после каждого шага обучения. Значит, PPO — алгоритм обучения по актуальному опыту, поэтому старые траектории должны отбрасываться после каждого шага обучения.

Усеченная целевая функция J^{clip} вычислительно малозатратна, очень проста для понимания, и для ее реализации требуется лишь несколько тривиальных преобразований изначальной суррогатной целевой функции J^{CPI}.

Наиболее вычислительно затратными являются шаги вычисления вероятностного отношения $r_t(\theta)$ и значения преимущества A_t. Однако это минимальные расчеты, необходимые для любых алгоритмов оптимизации суррогатной целевой функции. Остальные вычисления — это усечение и минимизация, которые выполняются за постоянное время.

В статье о PPO вариант с усеченной суррогатной целевой функцией показал большую производительность, чем вариант с целевой функцией со штрафом по расстоянию Кульбака — Лейблера. Версия PPO с усечением предпочтительнее, так как она и проще, и производительнее.

На этом завершается обсуждение ограничений на градиент стратегии в этой главе. Мы начали с целевой функции градиента стратегии $J(\theta) = \mathbb{E}_t\left[A_t \log \pi_\theta(a_t \mid s_t)\right]$, используемой в REINFORCE, и методе актора-критика. Затем ввели понятие суррогатной целевой функции $J^{\text{CPI}}(\theta)$ с перспективой монотонного улучшения, которое обеспечивается ограничением разницы между успешными стратегиями. PPO предоставляет два способа ограничения суррогатной целевой функции: с помощью штрафной функции расстояния Кульбака — Лейблера и эвристики с усечением, что дает нам J^{KLpen} и J^{clip} соответственно. Сравним все эти целевые функции градиента стратегии.

- Первоначальная целевая функция:

$$J(\theta) = \mathbb{E}_t\left[A_t^{\pi_\theta} \log \pi_\theta(a_t \mid s_t)\right]. \tag{7.35}$$

- Суррогатная целевая функция:

$$J^{\text{CPI}}(\theta) = \mathbb{E}_t\left[r_t(\theta) A_t^{\pi_{\theta_{\text{old}}}}\right]. \tag{7.36}$$

- Ограниченная суррогатная целевая функция:

$$J^{\text{CPI}}(\theta) \text{ при } \mathbb{E}_t\left[\mathrm{KL}\left(\pi_{\theta} \,\|\, \pi_{\theta_{\text{old}}}\right)\right] \leq \delta. \tag{7.37}$$

- PPO со штрафом по расстоянию Кульбака — Лейблера:

$$J^{\text{KLpen}}(\theta) = \mathbb{E}_t\left[r_t(\theta) A_t^{\pi_{\theta_{\text{old}}}} - \beta \mathrm{KL}\left(\pi_{\theta} \,\|\, \pi_{\theta_{\text{old}}}\right)\right]. \tag{7.38}$$

- PPO с усеченной суррогатной целевой функцией:

$$J^{\text{clip}}(\theta) = \mathbb{E}_t\left[\min\left(r_t(\theta) A_t^{\pi_{\theta_{\text{old}}}}, \text{clip}\left(r_t(\theta), 1-\varepsilon, 1+\varepsilon\right) A_t^{\pi_{\theta_{\text{old}}}}\right)\right]. \tag{7.39}$$

Для наглядности выразим равенства через θ, чтобы показать, какие из стратегий меняются, а какие используются для вычисления значений преимущества. Следует отметить, что лишь в изначальной целевой функции текущая стратегия π_θ применяется для расчета преимущества $A_t^{\pi_\theta}$, тогда как в остальных функциях задействуется более старая стратегия для определения преимущества $A_t^{\pi_{\theta_{\text{old}}}}$.

7.3. Алгоритм PPO

Поскольку PPO — метод градиента стратегии с преобразованием целевой функции, он совместим с обоими изученными нами алгоритмами градиента стратегии: REINFORCE и алгоритмом актора-критика. В алгоритме 7.2 показан PPO с реализацией усеченной целевой функции как расширения алгоритма актора-критика из главы 6.

Алгоритм 7.2. PPO с усеченной целевой функцией, расширяющий алгоритм актора-критика

```
Установить β ≥ 0, коэффициент для регуляризации энтропии
Установить ε ≥ 0, переменную усечения
Установить K, число эпох
Установить N, количество акторов
Установить T, временной горизонт
Установить M ≤ NT, размер мини-пакета
Установить α_a ≥ 0, скорость обучения актора
Установить α_c ≥ 0, скорость обучения критика
Инициализировать параметры актора θ_a и критика θ_c случайными значениями
Инициализировать "старую" сеть актора θ_A_old
for i = 1, 2, ..., do
    Установить θ_A_old = θ_a
    for actor = 1, 2, ..., N do
        Запустить стратегию θ_A_old в среде на T шагов и накопить траектории
        Вычислить значения преимущества A_1... A_T с помощью θ_A_old
        Рассчитать V^π_tar, 1, ..., V^π_tar, T с помощью сети
        ➥ критика θ_c и/или по данным траекторий
    end for
```

```
18:     Заполнить пакет размера NT накопленными траекториями, значениями
        ➥ преимущества и целевыми V-значениями
19:     for epoch = 1, 2, ..., K do
20:         for minibatch m in batch do
21:             Приведенные далее вычисления выполняются по всему мини-пакету m
22:             Рассчитать r_m(θ_a)
23:             Вычислить J_m^clip(θ_a) на основе r_m(θ_a) и преимуществ A_m из мини-пакета
24:             Рассчитать значения энтропии H_m, воспользовавшись сетью актора θ_a
25:             Определить функцию потерь для стратегии:
26:             L_pol(θ_a) = J_m^clip(θ_a) − βH_m
27:
28:             Рассчитать прогнозные V-значения V̂^π(s_m),
                ➥ используя сеть критика θ_c
29:             Вычислить значение функции потерь
                ➥ на основе V-значений из мини-пакета:
30:             L_val(θ_c) = MSE(V̂^π(s_m), V_tar^π(s_m))
31:
32:             Обновить параметры актора, например, с помощью SGD:
33:             θ_a = θ_a + α_a∇_θa L_pol(θ_a)
34:             Обновить параметры критика, например, с помощью SGD:
35:             θ_c = θ_c + α_c∇_θc L_val(θ_c)
36:         end for
37:     end for
38: end for
```

Рассмотрим алгоритм пошагово.

- Строки 1–8. Установка значений гиперпараметров.
- Строка 9. Инициализация сетей актора и критика.
- Строка 10. Инициализация старой сети критика, чтобы она выступала в качестве $\pi_{\theta_{old}}$ при расчете вероятностного отношения $r_t(\theta)$.
- Строки 12–18. Обновление старой сети актора до новой. Накопление данных траекторий с помощью старой сети актора и вычисление значений преимущества и целевых *V*-значений. Затем траектории с преимуществами и целевыми *V*-значениями сохраняются в пакет, из которого будут в дальнейшем выбираться. Обратите внимание на то, что в этой версии актора-критика используются параллельные акторы. Метод параллелизации обсуждается в главе 8.
- Строка 19. Проход по всему пакету в цикле из *K* эпох.
- Строки 20 и 21. Выборка мини-пакетов размера *M* из пакета. В этом блоке в расчетах применяются все элементы из мини-пакета.
- Строки 22 и 23. Вычисление усеченной суррогатной целевой функции.
- Строка 24. Определение энтропии стратегии для действий в мини-пакете.
- Строки 25 и 26. Расчет функции потерь для стратегии.

- Строки 28–30. Определение функции потерь для функции полезности. Заметьте, что целевые *V*-значения вычисляются в пакете один раз и используются повторно.
- Строки 32 и 33. Обновление параметров актора с помощью градиента функции потерь для стратегии.
- Строки 34 и 35. Обновление параметров критика с помощью градиента функции потерь для полезностей.

7.4. Реализация PPO

Мы уже видели, как PPO может быть использован для расширения алгоритма актора-критика. В этом разделе рассмотрим реализацию в SLM Lab.

Естественно, PPO может расширять класс `ActorCritic` и повторно применять большую часть его методов. Для преобразования в PPO нужно переопределить только два метода: цикл обучения и целевую функцию. Более того, преобразование целевой функции затрагивает лишь функцию потерь для стратегии, метод определения функции потерь для полезностей остается тем же.

7.4.1. Расчет функции потерь для стратегии в PPO

В листинге 7.1 для определения функции потерь для стратегии с помощью усеченной целевой функции в PPO переопределяется родительский метод из `ActorCritic`.

Логарифм вероятности действий вычисляется для текущей и старой сетей актора (строки 14–18). Они используются для расчета вероятностного отношения $r_t(\theta)$ (строка 20). Обратите внимание: мы возводим в степень их разности для преобразования логарифма вероятностей в отношение вероятностей.

Усеченная суррогатная целевая функция вычисляется как кусочно-заданная с дальнейшим объединением в одну кривую в строках 21–24. Остальная часть расчетов функции потерь для стратегии выполняется так же, как в родительском методе.

Листинг 7.1. Реализация PPO, вычисление функции потерь для стратегии

```
1 # slm_lab/agent/algorithm/ppo.py
2
3 class PPO(ActorCritic):
```

```
4     ...
5
6     def calc_policy_loss(self, batch, pdparams, advs):
7         clip_eps = self.body.clip_eps
8         action_pd = policy_util.init_action_pd(self.body.ActionPD, pdparams)
9         states = batch['states']
10        actions = batch['actions']
11        ...
12
13        # L^CLIP
14        log_probs = action_pd.log_prob(actions)
15        with torch.no_grad():
16            old_pdparams = self.calc_pdparam(states, net=self.old_net)
17            old_action_pd = policy_util.init_action_pd(self.body.ActionPD,
              ➥ old_pdparams)
18            old_log_probs = old_action_pd.log_prob(actions)
19        assert log_probs.shape == old_log_probs.shape
20        ratios = torch.exp(log_probs - old_log_probs)
21        sur_1 = ratios * advs
22        sur_2 = torch.clamp(ratios, 1.0 - clip_eps, 1.0 + clip_eps) * advs
23        # Смена знака, необходимая для максимизации
24        clip_loss = -torch.min(sur_1, sur_2).mean()
25
26        # Регуляризация энтропии H
27        entropy = action_pd.entropy().mean()
28        self.body.mean_entropy = entropy # обновление данных в журнале
29        ent_penalty = -self.body.entropy_coef * entropy
30
31        policy_loss = clip_loss + ent_penalty
32        return policy_loss
```

7.4.2. Цикл обучения PPO

В листинге 7.2 приведен цикл обучения PPO. Сначала старая сеть актора обновляется до основной сети (строка 10). Пакет накопленных траекторий выбирается из памяти (строка 11). В целях повышения эффективности значения преимущества и целевые *V*-значения рассчитываются и сохраняются для использования в мини-пакетах (строки 12–15). Приведенная здесь версия оптимизирует вычисление значений преимуществ и целевых *V*-значений в пакете в противоположность их расчету на каждом шаге в алгоритме 7.2.

Мы многократно производим итерации по всем данным (строка 18), чтобы повторно использовать их для обучения. Для каждой эпохи данные для обучения делятся на мини-пакеты (строки 19 и 20). В остальном логика обучения такая же, как в классе `ActorCritic`. Она включает вычисление функции потерь для стратегии и полезностей и их применение для обучения сетей актора и критика.

Листинг 7.2. Реализация PPO, метод обучения

```
# slm_lab/agent/algorithm/ppo.py

class PPO(ActorCritic):
    ...

    def train(self):
        ...
        clock = self.body.env.clock
        if self.to_train == 1:
            net_util.copy(self.net, self.old_net)
            batch = self.sample()
            clock.set_batch_size(len(batch))
            _pdparams, v_preds = self.calc_pdparam_v(batch)
            advs, v_targets = self.calc_advs_v_targets(batch, v_preds)
            batch['advs'], batch['v_targets'] = advs, v_targets
            ...
            total_loss = torch.tensor(0.0)
            for _ in range(self.training_epoch):
                minibatches = util.split_minibatch(batch,
                ➥ self.minibatch_size)
                for minibatch in minibatches:
                    ...
                    advs, v_targets = minibatch['advs'],
                    ➥ minibatch['v_targets']
                    pdparams, v_preds = self.calc_pdparam_v(minibatch)
                    policy_loss = self.calc_policy_loss(minibatch,
                    ➥ pdparams, advs) # из актора
                    val_loss = self.calc_val_loss(v_preds, v_targets)
                    ➥ # из критика
                    if self.shared: # общая сеть
                        loss = policy_loss + val_loss
                        self.net.train_step(loss, self.optim,
                        ➥ self.lr_scheduler, clock=clock,
                        ➥ global_net=self.global_net)
                    else:
                        self.net.train_step(policy_loss, self.optim,
                        ➥ self.lr_scheduler, clock=clock,
                        ➥ global_net=self.global_net)
                        self.critic_net.train_step(val_loss,
                        ➥ self.critic_optim, self.critic_lr_scheduler,
                        ➥ clock=clock, global_net=self.global_critic_net)
                        loss = policy_loss + val_loss
                    total_loss += loss
            loss = total_loss / self.training_epoch / len(minibatches)
            # сброс в начальное состояние
            self.to_train = 0
            return loss.item()
        else:
            return np.nan
```

7.5. Обучение агента PPO

В этом разделе мы будем обучать PPO игре Pong из Atari и действиям в среде BipedalWalker.

7.5.1. PPO в Pong

В листинге 7.3 приведен файл `spec`, который настраивает агента PPO для игры Pong из Atari. Файл имеется в SLM Lab в `slm_lab/spec/benchmark/ppo/ppo_pong.json`.

Листинг 7.3. Файл spec для PPO

```
# slm_lab/spec/benchmark/ppo/ppo_pong.json

{
    "ppo_pong": {
        "agent": [{
            "name": "PPO",
            "algorithm": {
                "name": "PPO",
                "action_pdtype": "default",
                "action_policy": "default",
                "explore_var_spec": null,
                "gamma": 0.99,
                "lam": 0.70,
                "clip_eps_spec": {
                    "name": "no_decay",
                    "start_val": 0.10,
                    "end_val": 0.10,
                    "start_step": 0,
                    "end_step": 0
                },
                "entropy_coef_spec": {
                    "name": "no_decay",
                    "start_val": 0.01,
                    "end_val": 0.01,
                    "start_step": 0,
                    "end_step": 0
                },
                "val_loss_coef": 0.5,
                "time_horizon": 128,
                "minibatch_size": 256,
                "training_epoch": 4
            },
            "memory": {
                "name": "OnPolicyBatchReplay",
            },
```

```
            "net": {
                "type": "ConvNet",
                "shared": true,
                "conv_hid_layers": [
                    [32, 8, 4, 0, 1],
                    [64, 4, 2, 0, 1],
                    [32, 3, 1, 0, 1]
                ],
                "fc_hid_layers": [512],
                "hid_layers_activation": "relu",
                "init_fn": "orthogonal_",
                "normalize": true,
                "batch_norm": false,
                "clip_grad_val": 0.5,
                "use_same_optim": false,
                "loss_spec": {
                    "name": "MSELoss"
                },
                "actor_optim_spec": {
                    "name": "Adam",
                    "lr": 2.5e-4,
                },
                "critic_optim_spec": {
                    "name": "Adam",
                    "lr": 2.5e-4,
                },
                "lr_scheduler_spec": {
                    "name": "LinearToZero",
                    "frame": 1e7
                },
                "gpu": true
            }
        }],
        "env": [{
            "name": "PongNoFrameskip-v4",
            "frame_op": "concat",
            "frame_op_len": 4,
            "reward_scale": "sign",
            "num_envs": 16,
            "max_t": null,
            "max_frame": 1e7
        }],
        "body": {
            "product": "outer",
            "num": 1
        },
        "meta": {
            "distributed": false,
            "log_frequency": 10000,
```

```
85             "eval_frequency": 10000,
86             "max_session": 4,
87             "max_trial": 1,
88         }
89     }
90 }
```

Рассмотрим основные компоненты.

- **Алгоритм** — это PPO (строка 8), стратегия выбора действий соответствует принятой по умолчанию стратегии (строка 10) для дискретных пространств действий (категориальное распределение). Значение γ устанавливается в строке 12. Для оценки преимуществ применяется GAE с заданием в строке 13 `lam` для λ. Гиперпараметр для усечения ε и скорость его уменьшения указаны в строках 14–20, а коэффициент энтропии — в строках 21–27. Коэффициент функции потерь для полезностей определен в строке 28.
- **Архитектура сети** — сверточная нейронная сеть с тремя сверточными слоями и одним полносвязным слоем с функцией активации ReLU (строки 37–45). У актора и критика общая сеть, как указано в строке 38. Сеть обучается на графическом процессоре, если он доступен (строка 66).
- **Оптимизатор** — Adam [68] со скоростью обучения 0,00025 (строки 54–57). Скорость обучения уменьшается до нулевой за 10 млн кадров (строки 62–65).
- **Частота обучения** — по пакетам в соответствии с требованиями алгоритма, поэтому выбрана память `OnPolicyBatchReplay` (строка 34). Количество эпох равно 4 (строка 31), а размер мини-пакета — 256 (строка 30). Временной горизонт T задан в строке 29, и число акторов указано в строке 74 как `num_envs`.
- **Среда** — Pong из Atari [14], [18] (строка 70).
- **Длительность обучения** — 10 млн шагов (строка 76).
- **Оценка агента** происходит каждые 10 000 шагов (строка 85).

Для тренировки этого агента PPO с помощью SLM Lab запустите в терминале команды, приведенные в листинге 7.4. Агент должен начинать со счета –21 и в среднем достигать счета, близкого к максимальному 21.

Листинг 7.4. Обучение агента PPO для Pong

```
1 conda activate lab
2 python run_lab.py slm_lab/spec/benchmark/ppo/ppo_pong.json ppo_pong train
```

Для получения усредненных результатов будет запущено испытание `Trial` с четырьмя сессиями `Session`. Испытание должно занять около половины дня при запуске на графическом процессоре. График результатов и его скользящее среднее приведены на рис. 7.2.

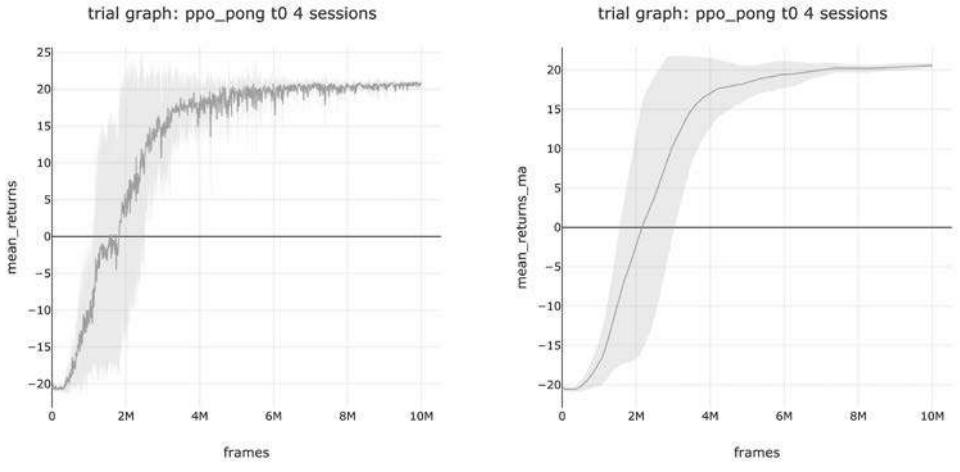


Рис. 7.2. Графики усредненных результатов по четырем сессиям испытания PPO из SLM Lab. По вертикальной оси отложены усредненные по восьми эпизодам полные вознаграждения в контрольных точках. По горизонтальной оси — все кадры обучения. По сравнению с методом актора-критика из главы 6 PPO обучается и достигает наибольшего счета намного быстрее

7.5.2. PPO в BipedalWalker

Как метод, основанный на стратегии, PPO может быть применен и к задачам непрерывного управления. В листинге 7.5 показан файл `spec`, который настраивает агента PPO для среды BipedalWalker. Файл находится в SLM Lab в `slm_lab/spec/benchmark/ppo/ppo_cont.json`. Обратите особое внимание на архитектуру сети (строки 37–39) и среду (строки 62–65).

Листинг 7.5. Файл spec для PPO в BipedalWalker

```
 1 # slm_lab/spec/benchmark/ppo/ppo_cont.json
 2
 3 {
 4     "ppo_bipedalwalker": {
 5         "agent": [{
 6             "name": "PPO",
 7             "algorithm": {
 8                 "name": "PPO",
 9                 "action_pdtype": "default",
10                 "action_policy": "default",
11                 "explore_var_spec": null,
12                 "gamma": 0.99,
13                 "lam": 0.95,
14                 "clip_eps_spec": {
```

```
                    "name": "no_decay",
                    "start_val": 0.20,
                    "end_val": 0.0,
                    "start_step": 10000,
                    "end_step": 1000000
                },
                "entropy_coef_spec": {
                    "name": "no_decay",
                    "start_val": 0.01,
                    "end_val": 0.01,
                    "start_step": 0,
                    "end_step": 0
                },
                "val_loss_coef": 0.5,
                "time_horizon": 512,
                "minibatch_size": 4096,
                "training_epoch": 15
            },
            "memory": {
                "name": "OnPolicyBatchReplay",
            },
            "net": {
                "type": "MLPNet",
                "shared": false,
                "hid_layers": [256, 128],
                "hid_layers_activation": "relu",
                "init_fn": "orthogonal_",
                "normalize": true,
                "batch_norm": false,
                "clip_grad_val": 0.5,
                "use_same_optim": true,
                "loss_spec": {
                    "name": "MSELoss"
                },
                "actor_optim_spec": {
                    "name": "Adam",
                    "lr": 3e-4,
                },
                "critic_optim_spec": {
                    "name": "Adam",
                    "lr": 3e-4,
                },
                "lr_scheduler_spec": null,
                "gpu": false
            }
        }],
        "env": [{
            "name": "BipedalWalker-v2",
            "num_envs": 32,
            "max_t": null,
```

```
            "max_frame": 4e6
        }],
        "body": {
            "product": "outer",
            "num": 1
        },
        "meta": {
            "distributed": false,
            "log_frequency": 10000,
            "eval_frequency": 10000,
            "max_session": 4,
            "max_trial": 1
        }
    }
}
```

Для обучения агента запустите команды из листинга 7.6 в терминале.

Листинг 7.6. Обучение агента PPO в BipedalWalker

```
conda activate lab
python run_lab.py slm_lab/spec/benchmark/ppo/ppo_cont.json ppo_bipedalwalker
  ➥ train
```

Будет запущено испытание `Trial` для получения графиков, приведенных на рис. 7.3.

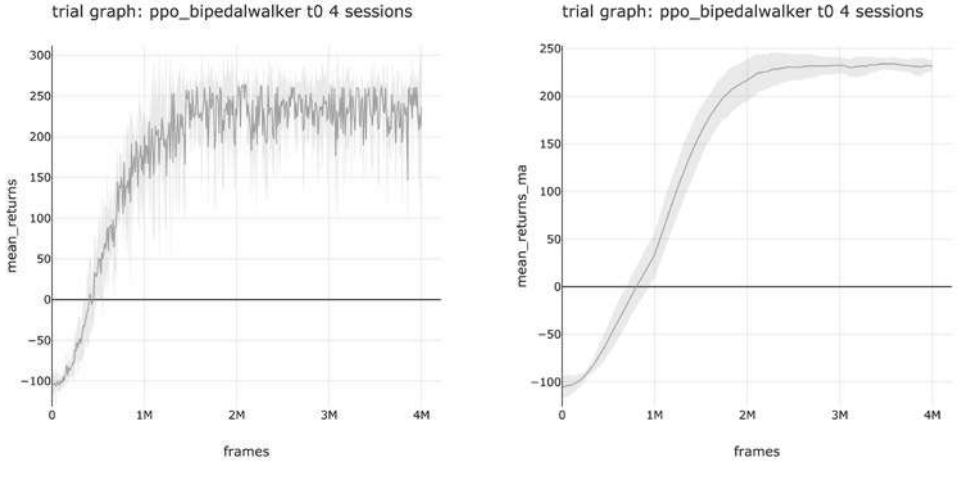


Рис. 7.3. Усредненные по четырем сессиям результаты испытания PPO в BipedalWalker из SLM Lab. Это испытание показало хорошую производительность при приближении решения к счету 300

7.6. Результаты экспериментов

В этом разделе мы с помощью SLM Lab проведем эксперимент по изучению влияния переменной λ в GAE на PPO. В качестве более сложной среды будем использовать игру Breakout из Atari.

7.6.1. Эксперимент по определению влияния λ в GAE

Поскольку PPO реализован как расширение актора-критика с применением GAE, мы проведем такой же эксперимент, как в разделе 6.7.2, для изучения влияния разных значений λ на PPO путем поиска по сетке. Приведенный в листинге 7.7 файл `spec` для эксперимента — версия из листинга 7.3, дополненная спецификацией поиска для `lam`. Строки 4 и 7 указывают изменение среды, а в строке 19 определен поиск по списку `lam` значений λ. Полный файл `spec` находится в SLM Lab в `slm_lab/spec/experimental/ppo/ppo_lam_search.json`.

Листинг 7.7. Файл spec для PPO со спецификацией поиска для разных значений lam коэффициента λ в GAE

```
 1 # slm_lab/spec/experimental/ppo/ppo_lam_search.json
 2
 3 {
 4     "ppo_breakout": {
 5         ...
 6         "env": [{
 7             "name": "BreakoutNoFrameskip-v4",
 8             "frame_op": "concat",
 9             "frame_op_len": 4,
10             "reward_scale": "sign",
11             "num_envs": 16,
12             "max_t": null,
13             "max_frame": 1e7
14         }],
15         ...
16         "search": {
17             "agent": [{
18                 "algorithm": {
19                     "lam__grid_search": [0.50, 0.70, 0.90, 0.95, 0.97, 0.99]
20                 }
21             }]
22           }
23         }
24     }
```

Для запуска эксперимента в SLM Lab воспользуйтесь командами из листинга 7.8.

Листинг 7.8. Запуск эксперимента с поиском по разным значениям λ в GAE в соответствии с файлом spec

```
1 conda activate lab
2 python run_lab.py slm_lab/spec/experimental/ppo/ppo_lam_search.json
  ➥ ppo_breakout search
```

Будет запущен эксперимент `Experiment`, порождающий шесть испытаний `Trial`, каждое с подстановкой разных значений из `lam` в исходный файл `spec` для PPO. В каждом `Trial` запускаются по четыре сессии `Session`. На рис. 7.4 приведены графики результатов множества испытаний.

Здесь показан эффект, оказываемый разными значениями λ в GAE на PPO в среде Breakout. При λ = 0,70 производительность наилучшая со счетом в эпизоде около 400, близкий результат дает λ = 0,50. Бóльшие значения λ, близкие к 0,90, не дают хорошей производительности. По сравнению с таким же экспериментом с решением такой же задачи для актора-критика из подраздела 6.7.2 оптимальное значение λ для PPO (0,70) значительно отличается от значения для актора-критика (0,90). При этом, как и ожидалось, производительность PPO выше, чем у метода актора-критика. Данный эксперимент продемонстрировал, что и в PPO λ — не очень чувствительный гиперпараметр.

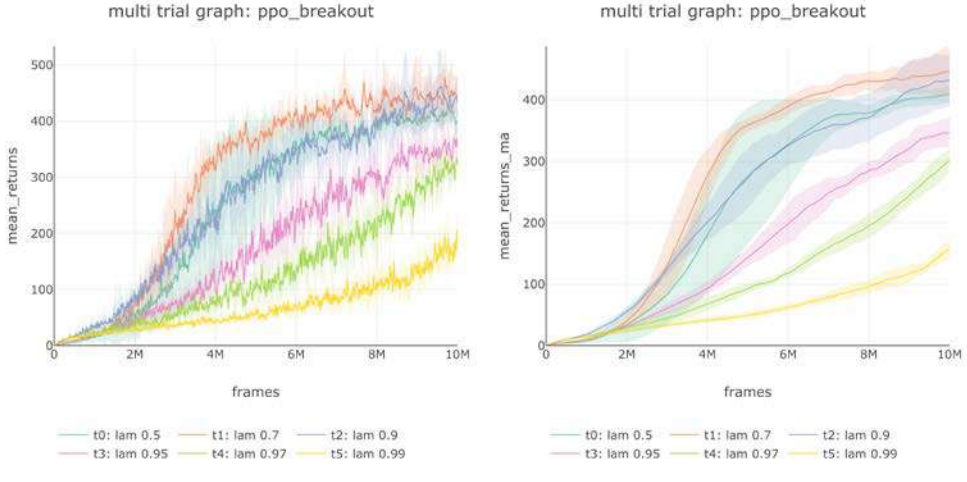


Рис. 7.4. Влияние разных значений λ в GAE на PPO в среде Breakout. При λ = 0,70 производительность наилучшая, тогда как значения λ, близкие к 0,90, показали худшие результаты

7.6.2. Эксперимент по определению влияния переменной ε для усеченной функции потерь

Переменная ε определяет ограничивающую окрестность $|r_t(\theta) - 1| \leq \varepsilon$ для усеченной суррогатной целевой функции в уравнении (7.34).

В этом эксперименте рассматривается влияние значений ε посредством поиска по сетке. Файл `spec` для эксперимента (листинг 7.9) является расширением листинга 7.3 с добавлением спецификации поиска для `clip_eps`. Также используется более сложная среда Qbert из Atari. В строках 4 и 7 указано изменение среды, а в строке 19 определен поиск по списку `clip_eps` значений ε. Полный файл `spec` находится в SLM Lab в `slm_lab/spec/experimental/ppo/ppo_eps_search.json`.

Листинг 7.9. Файл spec для PPO со спецификацией поиска для разных значений clip_eps ε для усеченной функции потерь

```
 1 # slm_lab/spec/experimental/ppo/ppo_eps_search.json
 2
 3 {
 4     "ppo_qbert": {
 5         ...
 6         "env": [{
 7             "name": "QbertNoFrameskip-v4",
 8             "frame_op": "concat",
 9             "frame_op_len": 4,
10             "reward_scale": "sign",
11             "num_envs": 16,
12             "max_t": null,
13             "max_frame": 1e7
14         }],
15         ...
16         "search": {
17             "agent": [{
18                 "clip_eps_spec": {
19                     "start_val__grid_search": [0.1, 0.2, 0.3, 0.4]
20                 }
21             }]
22         }
23     }
24 }
```

Для запуска эксперимента в SLM Lab воспользуйтесь командами из листинга 7.10.

Листинг 7.10. Запуск эксперимента с поиском по разным значениям ε для усеченной функции потерь в соответствии с файлом spec

```
1 conda activate lab
2 python run_lab.py slm_lab/spec/experimental/ppo/ppo_eps_search.json
  ➥ ppo_qbert search
```

Будет запущен эксперимент `Experiment`, состоящий из четырех испытаний `Trial`, в каждом из которых по четыре сессии `Session`. На рис. 7.5 приведены графики результатов множества испытаний.

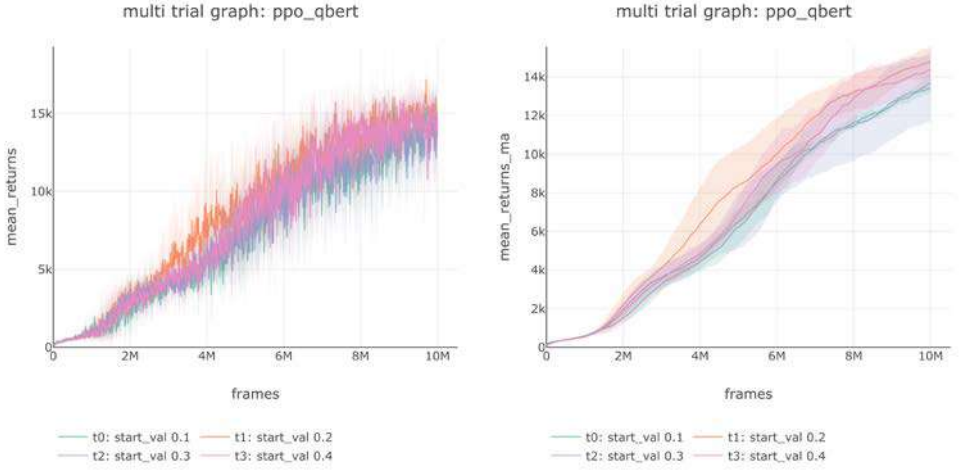


Рис. 7.5. Влияние разных значений ε для усеченной функции потерь на PPO в среде Qbert. В целом алгоритм не очень чувствителен к этому гиперпараметру

На рис. 7.5 показан эффект применения разных значений ε для усеченной функции для PPO в среде Qbert. Наилучшая производительность достигается при $\varepsilon = 0{,}20$. В целом чувствительность алгоритма к этому гиперпараметру низкая, что видно по близким значениям производительности при разных значениях ε.

7.7. Резюме

В этой главе рассматривалась проблема резкого падения производительности в методах градиента стратегии, вызванная тем, что поиск стратегии выполняется путем непрямого управления параметрами в пространстве параметров. Это может приводить к неустойчивым изменениям в стратегии во время обучения.

Для решения этой проблемы мы преобразовали целевую функцию градиента стратегии и получили суррогатную целевую функцию, такую, что изменения в пространстве параметров гарантировали монотонное улучшение стратегии. Это известно как подход монотонного улучшения. На практике суррогатная целевая функция может быть посчитана лишь приблизительно, вследствие чего возникает некоторая погрешность. Для данной погрешности можно задать границы, которые превращаются в ограничения целевой функции, и таким образом можно гарантировать улучшение стратегии как на практике, так и в теории.

Мы видели, что алгоритм PPO — простой и эффективный подход к решению задачи оптимизации с ограничениями. Существует два варианта: PPO со штрафом по расстоянию Кульбака — Лейблера и PPO с усеченной целевой функцией. Последний — более простой, вычислительно менее затратный и более производительный.

Три рассмотренных в этой книге алгоритма градиента стратегии являются естественным расширением друг друга. Начав с наиболее простого алгоритма, REINFORCE, мы расширили его до метода актора-критика, который улучшает подкрепляющий сигнал, передаваемый стратегии. Наконец, было показано, что PPO может служить расширением метода актора-критика, если преобразовать вычисление функции потерь для стратегии, что повышает устойчивость и эффективность выборок в процессе обучения.

7.8. Рекомендуемая литература

- *Levine S., Achiam J.* Oct 11: Advanced Policy Gradients, Lecture. 2017 [78].
- *Kakade S.* A Natural Policy Gradient. 2001 [63].
- *Schulman J., Levine S., Moritz P., Jordan M. I., Abbeel P.* Trust Region Policy Optimization. 2015 [122].
- *Duan Y., Chen X., Houthooft R., Schulman J., Abbeel P.* Benchmarking Deep Reinforcement Learning for Continuous Control. 2016 [35].
- *Achiam J., Held D., Tamar A., Abbeel P.* Constrained Policy Optimization. 2017 [2].
- *Schulman J., Wolski F., Dhariwal P., Radford A., Klimov O.* Proximal Policy Optimization Algorithms. 2017 [124].
- Microsoft Research. Policy Gradient Methods: Tutorial and New Frontiers [86].
- OpenAI Blog. Proximal Policy Optimization. 2017 [106].
- OpenAI Blog. More on Dota 2. 2017 [102].
- OpenAI Blog. OpenAI Five Benchmark: Results. 2018 [105].
- OpenAI Blog. OpenAI Five. 2018 [104].
- OpenAI Blog. Learning Dexterity. 2018 [101].

8 Методы параллелизации

При обсуждении алгоритмов глубокого RL в этой книге возникал вопрос об их низкой эффективности с точки зрения качества выборки. Для нетривиальных задач обычно требуются миллионы прецедентов, чтобы агент научился действовать с хорошей производительностью. Дни или даже недели могут уйти на то, чтобы породить достаточное количество прецедентов, если сбор данных выполняется последовательно одним агентом в рамках одного процесса.

Другая тема, вытекающая из наших рассуждений об алгоритме DQN, — важность разнообразия данных для обучения и отсутствия корреляции между ними для устойчивого и быстрого обучения. Это может быть достигнуто с помощью памяти прецедентов, но для нее нужно, чтобы DQN был реализован как обучение по отложенному опыту. Следовательно, данный подход не подходит для алгоритмов градиента стратегии, по крайней мере не напрямую[1]. Тем не менее эти алгоритмы могут повысить производительность при обучении на разнообразных данных.

В этой главе обсуждаются методы параллелизации, применимые ко всем алгоритмам глубокого RL. Они могут как существенно снизить время обучения, так и позволить порождать более разнообразные данные для алгоритмов обучения по актуальному опыту.

Основная суть этих методов — создание множества идентичных экземпляров агента и среды, которые одновременно независимо друг от друга используются для накопления траекторий. Поскольку агент параметризирован сетью, мы создаем много одинаковых действующих сетей и одну глобальную сеть. Действующие сети непрерывно накапливают траектории, а глобальная сеть пе-

---

[1] Можно обучать алгоритмы на актуальном опыте с данными, полученными при обучении по отложенному опыту, применив выборку по значимости, чтобы скорректировать разницу между вероятностями действий стратегий. Однако весовые коэффициенты значимости могут широко варьироваться как в сторону сильного увеличения, так и в сторону нуля. Поэтому так трудно на практике заставить работать коррекцию с помощью обучения по отложенному опыту.

риодически обновляется с помощью данных из рабочих сетей перед передачей в них изменений.

Есть две категории параллелизации: синхронная и асинхронная. Первую обсудим в разделе 8.1, а вторую — в разделе 8.2.

8.1. Синхронная параллелизация

Когда параллелизация синхронная (блокирующая), глобальная сеть перед обновлением своих параметров ожидает получения пакета обновлений от всех действующих сетей. А действующие сети после отправки обновлений ожидают, пока глобальная сеть обновится и отправит обратно новые параметры, чтобы обновить все действующие сети одновременно. Это показано в алгоритме 8.1.

Алгоритм 8.1. Синхронная параллелизация, глобальное вычисление градиентов

```
Установка скорости обучения α
Инициализация глобальной сети θ_g
Инициализация N действующих сетей θ_W,1, θ_W,2, ..., θ_W,N
for каждая действующая сеть do synchronously
    Взять параметры из глобальной сети и установить θ_W,i ← θ_g
    Накопить траектории
    Передать траектории в глобальную сеть
    Ожидать обновления глобальной сети
end for
    Ожидать получения траекторий из всех действующих сетей
    Вычислить градиент ∇_θg с помощью всех траекторий
    Обновить глобальную сеть θ_g ← θ_g + α∇_θg
```

При таком варианте параллелизма действующие сети используются для накопления траекторий, которые отсылаются в глобальную сеть. А она отвечает за применение этих траекторий для расчета градиентов и обновления параметров. Важно то, что все происходит с задействованием этапа синхронизации — синхронно.

При параллельном выполнении все действующие сети получают опыт в разных экземплярах среды, которые развертываются и создают ответвления для разных сценариев, в связи со стохастичностью в стратегиях и средах. Чтобы обеспечить это, нужно инициализировать экземпляры действующих сетей и сред с помощью разных случайных начальных значений. По мере развертывания траекторий при параллельном исполнении действующие сети могут оказаться в разных фазах среды: одни ближе к начальным шагам, другие — к конечным. Параллелизация вносит разнообразие в обучающие данные для глобальной сети, так как она обучается на многих сценариях и фазах среды. В целом это позволяет стабилизировать стратегию, представленную глобальной сетью, и сделать ее более устойчивой к шумам и изменениям.

Чередование прецедентов из разных действующих сетей помогает устранить корреляцию данных, так как на каждом временном шаге глобальная сеть получает коллекцию прецедентов из разных фаз среды. Например, если среда состоит из многих уровней игры, глобальная сеть, вероятно, будет обучаться на нескольких из них одновременно. Действующие сети и так уже будут исследовать различные сценарии по причине стохастичности, но мы можем дополнительно разрешить им использовать разные стратегии исследования среды, чтобы еще увеличить разнообразие данных. Один из способов сделать это — применить разные скорости уменьшения доли ε-жадной стратегии в *Q*-обучении из главы 4.

В SLM Lab синхронная параллелизация реализована с помощью обертки для векторизации среды из OpenAI [99]. Она создает множество экземпляров среды в разных процессах, которые сообщаются с основным процессом, в котором находится агент. Далее агент работает в этом наборе сред, эффективно функционирующем как множество действующих сетей. Здесь используется тот факт, что при синхронной параллелизации действующие сети гарантированно располагают одной и той же копией глобальной сети. Обертка для векторизации среды — обширная тема, ее детали выходят за рамки этой книги. Исходный код находится в `slm_lab/env/vec_env.py` в SLM Lab.

Чтобы задействовать синхронную параллелизацию в SLM Lab, просто укажите желаемое количество сред `num_envs` в файле `spec`. Это применимо к любому алгоритму, мы пользовались этим приемом при проведении экспериментов и испытаний в этой книге. Пример — строка 74 листинга 7.3 в файле `spec` для PPO.

8.2. Асинхронная параллелизация

Если используется асинхронная (неблокирующая) параллелизация, то глобальная сеть обновляет свои параметры, как только получает данные от какой-нибудь действующей сети. Аналогично каждая действующая сеть периодически обновляет свои параметры до параметров глобальной сети. Это значит, что наборы параметров действующих сетей могут немного различаться, что показано в алгоритме 8.2.

Асинхронная параллелизация была впервые применена к глубокому RL в работе Мниха и др. *Asynchronous Methods for Deep Reinforcement Learning* [87], известной также как статья об алгоритме A3C. Она сыграла решающую роль в популяризации методов актора-критика, позволив им конкурировать с методами, основанными на полезности, такими как DQN. В результате применения асинхронного метода актора-критика (Asynchronous Advantage Actor-Critic, A3C) были достигнуты самые современные показатели производительности для игр Atari. Кроме того, была продемонстрирована возможность ускоренного обучения алгоритмов RL на процессоре, который значительно дешевле, чем графический. A3C — это просто асинхронная версия алгоритмов актора-критика с оценкой преимущества из главы 6.

Алгоритм 8.2. Асинхронная параллелизация, расчет градиентов для действующих сетей

```
 1: Установка скорости обучения α
 2: Инициализация глобальной сети θ_g
 3: Инициализация N действующих сетей θ_W,1, θ_W,2, ..., θ_W,N
 4: for каждая действующая сеть do asynchronously
 5:     Взять параметры из глобальной сети и установить θ_W,i ← θ_g
 6:     Накопить траектории
 7:     Вычислить градиент ∇_θW,i
 8:     Передать ∇_θW,i в глобальную сеть
 9: end for
10: При получении градиента из действующей сети
    ➥ обновить глобальную сеть θ_g ← θ_g + α∇_θW,i
```

В этом примере действующие сети, помимо накопления траекторий, также рассчитывают собственные градиенты на основе собственных данных. По мере того как градиенты становятся доступными, они асинхронно пересылаются в глобальную сеть. В отличие от случая синхронной параллелизации, в этом варианте нет общего этапа синхронизации.

Существует много способов реализации асинхронного обучения. В число основных проектных решений входят следующие.

1. **С запаздыванием и без запаздывания.** У действующих сетей есть необходимость периодически обновлять свои параметры в соответствии с глобальной сетью. Частота обновлений может быть разной. Если обновление немедленное, то параметры действующих сетей всегда соответствуют параметрам глобальной сети. Один из способов реализации этого — общая сеть в памяти компьютера. Однако это значит, что всем действующим сетям нужен доступ к общей памяти, из-за чего все они должны быть запущены на одной и той же машине. Если обновления происходят изредка, у действующих сетей чаще всего будут копии глобальных параметров с запаздыванием. Сеть с запаздыванием позволяет обеспечить сохранность обучения по актуальному опыту для всех действующих сетей, так как гарантирует неизменность стратегии при порождении траекторий и вычислении градиентов.
2. **Локальный и глобальный расчет градиентов.** Градиенты можно вычислять как локально — с помощью действующих сетей, так и глобально — с использованием глобальной сети. Для обучения по актуальному опыту требуется, чтобы градиенты рассчитывались локально, поскольку глобальная сеть постоянно меняется, у алгоритмов обучения по отложенному опыту нет такого ограничения. Кроме того, из практических соображений следует учитывать, насколько затратен обмен данными между действующими сетями и глобальной сетью. Пересылка между процессами данных для траекторий, как правило, более затратна, чем отправка градиентов.
3. **С блокировкой и без блокировки.** Глобальная сеть может быть заблокирована во время обновления параметров, для того чтобы этот процесс протекал

последовательно. Выполняемые одновременно обновления могут переписать части друг друга. Обычно это считается проблемой, поэтому в большинстве случаев глобальная сеть блокируется. Это увеличивает время обучения, так как одновременные обновления нужно разрешать последовательно. Тем не менее в ряде задач в целях экономии времени может быть применена глобальная сеть без блокировки. На этом основан алгоритм Hogwild!, обсуждаемый далее.

8.2.1. Hogwild!

В SLM Lab асинхронная параллелизация реализована с помощью алгоритма Hogwild![1] — *безблокировочного* метода параллелизации стохастического градиентного спуска [93]. В Hogwild! глобальная сеть не блокируется во время обновления параметров. При его реализации применяются также общая глобальная сеть без запаздывания и локальное вычисление градиентов.

Прежде чем перейти к разбору кода, стоит уделить некоторое время рассмотрению принципа работы алгоритма Hogwild!.

С параллельным обновлением параметров при отсутствии блокировки связана проблема перезаписи, которая вызывает разрушительные конфликты. Однако этот эффект можно свести к минимуму, если допустить, что задача оптимизации разреженная. То есть для данной параметризированной нейронной сетью функции при обновлении параметров будет, как правило, преобразован лишь малый их поднабор. В этом случае обновления редко будут вступать в противоречие, и перезапись перестанет происходить постоянно. Если исходить из данного *допущения о разреженности*, то параметры, преобразованные при множестве одновременных обновлений, являются непересекающимися в вероятностном смысле. Следовательно, стратегия безблокировочного параллелизма эффективна. Она жертвует небольшим количеством конфликтов в пользу повышения скорости обучения.

Можно сравнить обе схемы, чтобы увидеть, как разреженность применяется для распараллеливания обновлений параметров, которые иначе были бы последовательными. Пусть θ_i — параметр сети на i-й итерации. Последовательное обновление может быть записано в следующем виде:

$$\theta_1 \xrightarrow{u_{1\to 2}} \theta_2 \xrightarrow{u_{2\to 3}} \theta_3 \dots \xrightarrow{u_{n-1\to n}} \theta_n, \tag{8.1}$$

где $u_i \to_{i+1}$ — обновление, такое что $\theta_{i+1} = \theta_i + u_i \to_{i+1}$. Тогда согласно допущению о разреженности небольшая группа из w обновлений $u_j \to_{j+1} \dots u_{j+w-1} \to_{j+w}$ содержит весьма малое количество пересекающихся элементов. И последовательные обновления могут быть сжаты до параллельного обновления, которое w действующих сетей порождают независимо друг от друга, чтобы получить $u_j \to_{j+1} \dots u_j \to_{j+w}$.

[1] Hogwild (безудержный) — меткое название для данного алгоритма, ведь он подразумевает неуправляемость и отсутствие ограничений.

Для приведенных далее заданных w обновлений, производимых последовательно и параллельно,

$$\theta_j \xrightarrow{\Delta\theta_{j+1}} \theta_{j+1} \xrightarrow{\Delta\theta_{j+2}} \theta_{j+2} \cdots \xrightarrow{\Delta\theta_{j+w-1}} \theta_{j+w} \quad \text{(последовательно);}$$

$$\theta_j \xrightarrow{\Delta\theta_{j+1}} \theta_{\|j+1} \xrightarrow{\Delta\theta_{j+2}} \theta_{\|j+2} \cdots \xrightarrow{\Delta\theta_{j+w}} \theta_{\|j+w} \quad \text{(параллельно),}$$

полученные в обоих случаях параметры на w-х итерациях точно или приблизительно равны $\theta_{j+w} \simeq \theta_{\|j+w}$. Следовательно, параллелизация с помощью w действующих сетей может быть быстрой аппроксимацией, близкой к последовательному выполнению обновлений. Поскольку в обоих случаях мы решаем одну и ту же задачу оптимизации, они должны давать одинаковые или почти одинаковые результаты.

При использовании Hogwild! нужно учитывать, что чем больше действующих сетей, тем выше вероятность конфликтов при обновлении параметров. Разреженность возникает лишь при небольшом количестве параллельных операций. Если конфликты возникают слишком часто, это допущение неприменимо.

Насколько оправданно допущение о разреженности применительно к RL? К несчастью, этот вопрос недостаточно изучен. Как бы то ни было, этот подход был реализован Мнихом и др. в статье об A3C. Успешность применения данного подхода предполагает несколько возможных объяснений. Во-первых, вполне вероятно, что в задачах, при решении которых он использовался, обновления параметров часто были разреженными. Во-вторых, для компенсации шума, внесенного конфликтующими обновлениями, могли быть проведены дополнительные этапы обучения. В-третьих, одновременные обновления параметров могли происходить редко, например, по той причине, что этапы обновления некоторых действующих сетей были разнесены между собой. В-четвертых, согласно исследованиям методов сжатия нейронных сетей [30, 41, 90] предполагается, что во время их обучения множество параметров необязательны либо излишни. Если конфликты возникнут при обновлении этих параметров, то вряд ли это повлияет на производительность в целом. Полученные результаты могут объясняться совокупностью всех факторов. Эффективность Hogwild! в глубоком RL остается интересной открытой проблемой.

В листинге 8.1 приведена минимальная реализация Hogwild!, примененного к типичному процессу обучения нейронной сети. Основная логика связана с созданием сети и размещением ее в общей памяти перед передачей в действующие сети для обновлений. Такая простота реализации возможна благодаря превосходной интеграции PyTorch с многопроцессорной обработкой данных.

Листинг 8.1. Пример минимальной реализации Hogwild!

```
# Пример минимальной реализации Hogwild!
import torch
import torch.multiprocessing as mp
```

```
4
5 # примеры сети, оптимизатора и функции потерь из PyTorch
6 net = Net()
7 optimizer = torch.optim.SGD(net.parameters(), lr=0.001)
8 loss_fn = torch.nn.F.smooth_l1_loss
9
10 def train(net):
11     # создание data_loader, optimizer, loss_fn
12     net.train()
13     for x, y_target in data_loader:
14         optimizer.zero_grad() # удалить все ранее накопленные градиенты
15         # autograd начинает накапливать следующие градиенты
16         y_pred = net(x) # прямой проход
17         loss = loss_fn(y_pred, y_target) # расчет функции потерь
18         loss.backward() # обратное распространение
19         optimizer.step() # обновление весов сети
20
21 def hogwild(net, num_cpus):
22     net.share_memory() # это заставляет все действующие сети
       ➥ использовать общую память
23     workers = []
24     for _rank in range(num_cpus):
25         w = mp.Process(target=train, args=(net, ))
26         w.start()
27         workers.append(w)
28     for w in workers:
29         w.join()
30
31 if __name__ == '__main__':
32     net = Net()
33     hogwild(net, num_cpus=4)
```

8.3. Обучение агента A3C

Если к алгоритму актора-критика с любой функцией преимущества применена параллелизация с помощью асинхронного метода, то он называется A3C [87]. Ко всем реализованным в SLM Lab алгоритмам можно применить параллелизацию путем простого добавления флага в файл `spec`. В листинге 8.2 это продемонстрировано на примере модифицированной спецификации для актора-критика из главы 6. Полный файл есть в SLM Lab в `slm_lab/spec/benchmark/a3c/a3c_nstep_pong.json`.

Листинг 8.2. Файл spec для A3C для игры Pong из Atari

```
1 # slm_lab/spec/benchmark/a3c/a3c_nstep_pong.json
2
3 {
4     "a3c_nstep_pong": {
5         "agent": [{
```

```
        "name": "A3C",
        "algorithm": {
            "name": "ActorCritic",
            "action_pdtype": "default",
            "action_policy": "default",
            "explore_var_spec": null,
            "gamma": 0.99,
            "lam": null,
            "num_step_returns": 5,
            "entropy_coef_spec": {
                "name": "no_decay",
                "start_val": 0.01,
                "end_val": 0.01,
                "start_step": 0,
                "end_step": 0
            },
            "val_loss_coef": 0.5,
            "training_frequency": 5
        },
        "memory": {
            "name": "OnPolicyBatchReplay",
        },
        "net": {
            "type": "ConvNet",
            "shared": true,
            "conv_hid_layers": [
                [32, 8, 4, 0, 1],
                [64, 4, 2, 0, 1],
                [32, 3, 1, 0, 1]
            ],
            "fc_hid_layers": [512],
            "hid_layers_activation": "relu",
            "init_fn": "orthogonal_",
            "normalize": true,
            "batch_norm": false,
            "clip_grad_val": 0.5,
            "use_same_optim": false,
            "loss_spec": {
                "name": "MSELoss"
            },
            "actor_optim_spec": {
                "name": "GlobalAdam",
                "lr": 1e-4
            },
            "critic_optim_spec": {
                "name": "GlobalAdam",
                "lr": 1e-4
            },
            "lr_scheduler_spec": null,
            "gpu": false
        }
```

```
57          }],
58          "env": [{
59              "name": "PongNoFrameskip-v4",
60              "frame_op": "concat",
61              "frame_op_len": 4,
62              "reward_scale": "sign",
63              "num_envs": 8,
64              "max_t": null,
65              "max_frame": 1e7
66          }],
67          "body": {
68              "product": "outer",
69              "num": 1
70          },
71          "meta": {
72              "distributed": "synced",
73              "log_frequency": 10000,
74              "eval_frequency": 10000,
75              "max_session": 16,
76              "max_trial": 1
77          }
78      }
79  }
```

В листинге 8.2 устанавливаются спецификация метаданных `"distributed": "synced"` (строка 72) и количество действующих сетей `max_session`, равное 16 (строка 75). Оптимизатор изменен на версию `GlobalAdam` (строка 47), которая больше подходит для Hogwild!. Также меняем количество сред `num_envs` на 8 (строка 63). Нужно отметить, что если количество сред больше 1, алгоритм станет гибридным с объединением синхронных (векторизация среды) и асинхронных (Hogwild!) методов. Тогда количество действующих сетей составит `num_envs` · `max_session`. С концептуальной точки зрения их можно рассматривать как иерархию в Hogwild! действующих сетей, каждая из которых порождает несколько синхронных действующих сетей.

Для обучения агента A3C с оценкой преимущества по отдаче за *n* шагов с помощью SLM Lab запустите в терминале команды из листинга 8.3.

Листинг 8.3. Обучение агента A3C

```
conda activate lab
python run_lab.py slm_lab/spec/benchmark/a3c/a3c_nstep_pong.json
  ➥ a3c_nstep_pong train
```

Как обычно, будет запущено испытание `Trial` для получения графиков, показанных на рис. 8.1. Однако обратите внимание на то, что теперь сессии исполняют роль асинхронных действующих сетей. При запуске на ЦПУ испытание должно занять лишь несколько часов, хотя для этого потребуется машина по крайней мере с 16 ядрами.

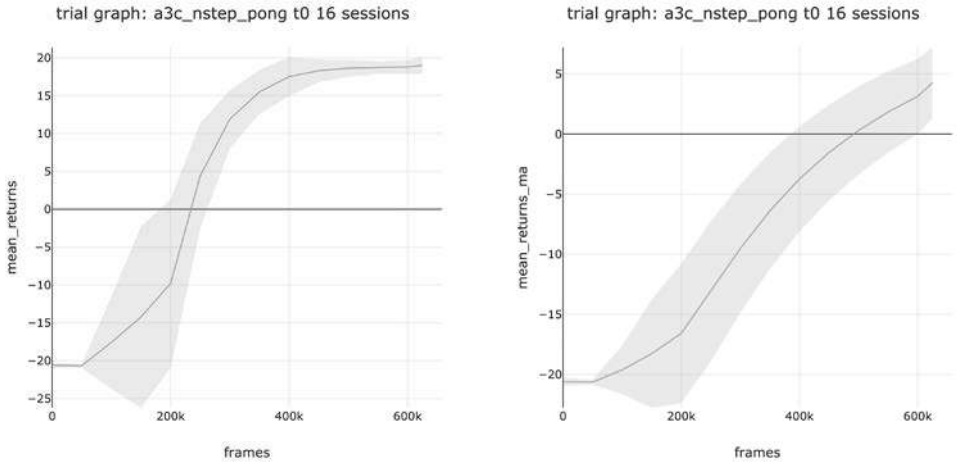


Рис. 8.1. Графики результатов испытаний A3C (оценка преимущества по отдаче за n шагов) с 16 действующими сетями. Поскольку сессии выступают в роли действующих сетей, по горизонтальной оси отложено количество кадров для отдельной сети. Следовательно, полное количество кадров для всех сетей равно сумме количеств кадров для отдельных сетей — 10 млн кадров

8.4. Резюме

В этой главе обсуждались два широко применяемых метода параллелизации: синхронная и асинхронная. Было показано, что они могут быть реализованы с помощью векторизации среды и алгоритма Hogwild! соответственно.

Двумя преимуществами параллелизации являются ускорение обучения и повышение разнообразия данных. Последнее играет основную роль в повышении устойчивости и улучшении обучения алгоритмов градиента стратегии. Фактически от этого зависит успех или неудача обучения.

Чтобы выбрать, какой из методов параллелизации применить, нужно рассмотреть такие их факторы, как простота реализации, вычислительная сложность и масштаб задачи.

Синхронные методы (то есть векторизация среды) зачастую простые, и их легче реализовать, чем асинхронные методы, особенно если параллелизация используется только при сборе данных. Порождение данных, как правило, менее затратно, в связи с чем для одного и того же количества кадров потребуется меньше ресурсов. Поэтому лучше масштабировать до умеренного количества действующих сетей, например менее 100. Однако этап синхронизации становится сдерживающим

фактором при дальнейшем масштабировании. В этом случае асинхронные методы могут оказаться значительно более быстрыми.

Необходимость в параллелизации возникает далеко не всегда. Основное правило: попробуйте понять, достаточно ли проста задача для того, чтобы решить ее без параллелизации, прежде чем тратить на последнюю время и ресурсы. Необходимость параллелизации обусловлена применяемым алгоритмом. Алгоритмы обучения по отложенному опыту, такие как DQN, зачастую достигают наибольшей производительности без параллелизации из-за того, что память прецедентов уже предоставляет разнообразные данные. Даже если на обучение уходит очень много времени, агенты по-прежнему могут эффективно учиться. Но для того, чтобы алгоритмы обучения по актуальному опыту, такие как метод актора-критика, могли обучаться на разнообразных данных, часто необходима параллелизация.

8.5. Рекомендуемая литература

- *Mnih V., Badia A. P., Mirza M., Graves A., Harley T., Lillicrap T. P., Silver D., Kavukcuoglu K.* Asynchronous Methods for Deep Reinforcement Learning. 2016 [87].
- *Niu F., Recht B., Re C., Wright S. J.* HOGWILD! A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. 2011 [93].

9 Сравнительный анализ алгоритмов

В этой книге были введены три основные характеристики алгоритмов. Во-первых, выбираем мы алгоритм обучения по актуальному опыту или по отложенному? Во-вторых, к каким типам пространств параметров его можно применить? И в-третьих, какие функции он настраивает?

REINFORCE, SARSA, A2C и PPO — алгоритмы обучения по актуальному опыту, тогда как DQN и двойная DQN с PER — по отложенному. SARSA, DQN и двойная DQN с PER — основанные на полезности алгоритмы, которые настраивают аппроксимацию функции Q^{π}. Следовательно, они применимы только к средам с дискретными пространствами действий.

REINFORCE — это в чистом виде основанный на стратегии алгоритм, поэтому он настраивает только стратегию π. A2C и PPO — гибридные методы, настраивающие стратегию π и функцию V^{π}. REINFORCE, A2C и PPO применимы к средам как с дискретным, так и с непрерывным пространством действий. Характеристики всех алгоритмов сведены в табл. 9.1.

Таблица 9.1. Характеристики всех алгоритмов из этой книги

Алгоритм	**Обучение по актуальному или отложенному опыту**	**Среда**		**Настраиваемая функция**		
		Дискретная	**Непрерывная**	V^{π}	Q^{π}	**Стратегия** π
REINFORCE	Актуальный опыт	+	+			+
SARSA	Актуальный опыт	+			+	
DQN	Отложенный опыт	+			+	
Двойная DQN с PER	Отложенный опыт	+			+	
A2C	Актуальный опыт	+	+	+		+
PPO	Актуальный опыт	+	+	+		+

Обсуждавшиеся нами алгоритмы образуют два семейства, как показано на рис. 9.1. В каждом семействе есть базовый алгоритм, который расширяют все остальные. Первое семейство — основанные на полезности алгоритмы SARSA, DQN и двойная DQN с PER. В этом семействе SARSA — базовый алгоритм. DQN можно рассматривать как расширение SARSA с более высокой эффективностью выборки, так как в нем

применяется обучение по отложенному опыту. PER и двойная DQN — это расширения DQN, которые повышают эффективность выборки и устойчивость обучения.

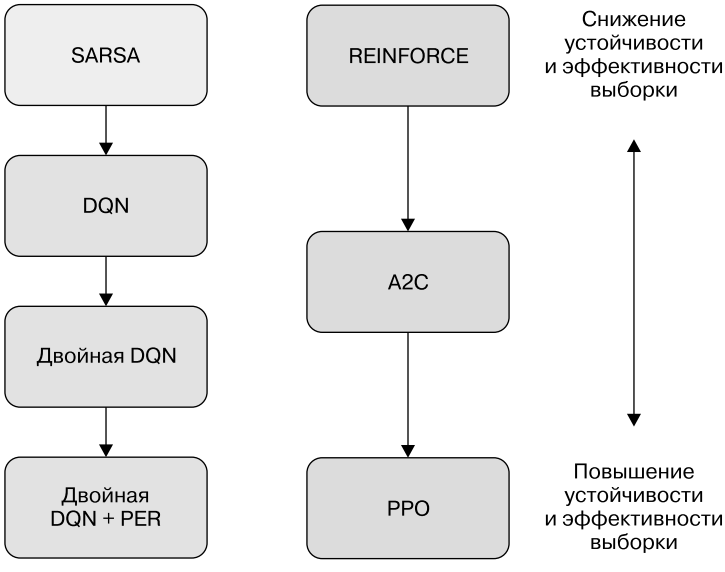


Рис. 9.1. Все алгоритмы в этой книге являются расширением SARSA и REINFORCE

Второе семейство состоит из основанных на стратегии и комбинированных алгоритмов REINFORCE, A2C и PPO. REINFORCE — базовый алгоритм в этом семействе. A2C — расширение REINFORCE с заменой оценки отдачи по методу Монте-Карло на настройку функции полезности. PPO расширяет A2C за счет преобразования целевой функции во избежание резкого падения производительности и для повышения эффективности выборки.

Из всех обсуждавшихся в книге алгоритмов наиболее производительными являются PPO и двойная DQN с PER. В своих семействах эти алгоритмы, как правило, наиболее устойчивые и эффективные с точки зрения качества выборки. Поэтому, приступая к работе над новой задачей, начинать лучше с них.

Принимая решение, что использовать: двойную DQN с PER или PPO, нужно рассмотреть два важных фактора — пространство действий среды и затратность порождения траекторий. Агент PPO может обучаться в средах с любым типом пространства действий, тогда как двойная DQN с PER ограничена дискретными действиями. Тем не менее двойная DQN с PER может обучаться с повторным использованием данных, порожденных при обучении по отложенному опыту с помощью любых других методов. Это становится преимуществом, когда сбор данных затратен с точки зрения ресурсов или времени, например, если нужно накапливать данные из реального мира. В отличие от него PPO — алгоритм обучения по актуальному опыту, поэтому он может обучаться только на данных, порожденных его собственной стратегией.

Часть III
Практика

10 Начало работы с глубоким RL

В глубоком RL система обучения представляет собой агента, взаимодействующего со средой. Агенты, в свою очередь, состоят из таких компонентов, как память, стратегия, нейронные сети и алгоритмические функции. Каждый из этих элементов по отдельности может быть довольно сложным, а для получения работоспособного алгоритма глубокого RL они должны еще и объединяться и действовать совместно. В результате кодовая база, в которой реализовано множество алгоритмов глубокого RL, переходит в класс больших программных систем с существенным количеством кода. Разные компоненты в них зависят друг от друга, что мешает им работать свободно и увеличивает вероятность возникновения ошибок. Из-за этого систему трудно заставить действовать, а вызвать сбой в ее работе — легко.

В этой главе приводится несколько практических советов по отладке реализаций алгоритмов глубокого RL. В разделе 10.1 вы познакомитесь с полезными методами проектирования, которые позволяют упростить код. Затем, в разделе 10.2, мы обсудим несколько методов отладки, а в разделе 10.3 — особые приемы обучения агентов в средах Atari. В конце главы имеется справочник по *глубокому RL*, в котором перечислены оптимальные значения гиперпараметров для основных алгоритмов и сред, обсуждаемых в этой книге. В нем также представлена информация о приблизительном времени выполнения и указаны требования к вычислительным ресурсам.

10.1. Приемы проектирования программ

На самом высоком уровне проектирования необходима точность исполнения с точки зрения теории и практики. При проектировании нового алгоритма RL или нового компонента перед его реализацией нужно теоретически доказать его корректность. Это особенно важно на стадии исследований. Аналогично при попытке решения новой задачи в среде в первую очередь нужно удостовериться в том, что она действительно решается с помощью RL, прежде чем применять какие-либо алгоритмы. Это особенно важно для приложений. Если задача разрешима и алгоритм RL теоретически верен, то его сбой может быть обусловлен ошибками реализации. И дальше нам нужно заниматься отладкой кода.

Отладка включает в себя определение ошибок во фрагменте кода и их исправление. Часто наиболее длительный процесс — это поиск ошибок и попытка понять, почему они здесь произошли. Как только вы разобрались в ошибке, ее исправление может оказаться простым и быстрым.

Существует много разных причин, по которым алгоритмы глубокого RL могут потерпеть неудачу. К тому же циклы разработки RL высокозатратны с точки зрения ресурсов и времени, что затрудняет процесс отладки. Для эффективной отладки кода RL нужно применить к определению ошибок систематический подход, чтобы быстро сузить перечень возможных мест их возникновения. К счастью, эта проблема далеко не нова в области проектирования ПО, и существует много установившихся хороших практик, которые мы можем позаимствовать. В частности, это модульное тестирование, качество кода и работа с Git.

10.1.1. Модульное тестирование

Модульное тестирование — универсальное требование к разработке хорошего ПО по той простой причине, что *любой непротестированный код небезопасен*. Надежные системы ПО должны строиться на серьезных основаниях. Для компонентов, которые выступают в качестве базовых для другого кода, необходимы всестороннее тестирование и верификация корректности их работы. Если основа ненадежная, то неразрешенные ошибки будут распространяться и влиять на все большее количество компонентов по мере разрастания системы. Прогресс замедлится, поскольку нам придется тратить больше времени на отладку и меньше — на разработку.

Модульное тестирование гарантирует, что протестированные компоненты работают, как задумывалось. Как только фрагмент кода должным образом протестирован, мы можем доверять ему и более уверенно использовать его при проектировании. Систематическое модульное тестирование базового кода способствует устранению того, что может работать неправильно, тем самым сужая список кандидатов на ошибку при отладке.

Покрытие тестами — полезный показатель, применяемый разработчиками ПО для измерения выраженной в процентах доли кодовой базы, проверенной модульными тестами. Она рассчитывается с помощью инструмента статического анализа, который определяет количество всех строк кода или логических путей и проверяет, как много строк охвачено при запуске всех модульных тестов. Отношение этих величин преобразуется в процент покрытия тестами. Все современные инструменты модульного тестирования автоматически рассчитывают этот показатель и выдают отчет о нем. В SLM Lab это производится с помощью `PyTest` при каждом запуске модульных тестов.

Каким бы заманчивым ни было стопроцентное покрытие тестами, оно может быть непродуктивным. Возможно, абсолютное покрытие тестами желательно при

запуске промышленного ПО в промышленной среде, но в нем мало смысла при проведении исследований, которые должны идти быстрыми темпами. Написание модульных тестов требует времени и усилий, и если с ним переусердствовать, то оно может начать соперничать за приоритет с исследованием и разработкой. Компромиссное решение — тестировать наиболее важный код. У автора кода должна быть возможность решать, что достаточно важно для тестирования. Как показывает практика, тестировать следует код, который широко используется, предрасположен к ошибкам, является сложным или значимым.

Модульные тесты в большинстве случаев проверяют корректность того, как запускается фрагмент кода. Однако нужно указать, какое поведение должно быть протестировано. Например, в тесте для функции указываются входные данные вместе с ожидаемым результатом, который сравнивается с вычисленным выходным значением. Для функции обычно тестируются несколько показателей: корректность выполнения для ряда входных значений (должны рассматриваться общие и крайние случаи) и правильность типа и структуры данных. Если в функции реализована формула, протестируйте ее эмпирически: рассчитайте вручную несколько численных результатов и сравните их с выходными значениями функции. При исправлении новой ошибки для предотвращения ее повторного появления нужно добавить тест.

В качестве подтверждения в листинге 10.1 приведен пример теста, который прогоняет от начала до конца алгоритм DQN в среде Pong из Atari. Тест инициализирует агента и среду и запускает полный цикл обучения на небольшое количество шагов. Несмотря на свою простоту, он служит общей и полной проверкой, гарантирующей, что все выполняется без нарушений. Этим обеспечивается также выполнение по всему алгоритму важных утверждений, касающихся вычисления функции потерь и обновления сети. Общие базовые тесты, такие как `test_atari`, особенно полезны при разрастании кодовой базы, так как помогают проверить, что новые функции не нарушают старую основную функциональность.

Листинг 10.1. Сквозной тест алгоритма DQN в среде Pong из Atari

```
# slm_lab/test/spec/test_spec.py

from flaky import flaky
from slm_lab.experiment.control import Trial
from slm_lab.spec import spec_util
import pytest

# вспомогательный метод для запуска всех тестов в test_spec
def run_trial_test(spec_file, spec_name=False):
    spec = spec_util.get(spec_file, spec_name)
    spec = spec_util.override_test_spec(spec)
    spec_util.tick(spec, 'trial')
    trial = Trial(spec)
    trial_metrics = trial.run()
```

```
15     assert isinstance(trial_metrics, dict)
16
17 ...
18
19 @flaky
20 @pytest.mark.parametrize('spec_file,spec_name', [
21     ('benchmark/dqn/dqn_pong.json', 'dqn_pong'),
22     ('benchmark/a2c/a2c_gae_pong.json', 'a2c_gae_pong'),
23 ])
24 def test_atari(spec_file, spec_name):
25     run_trial_test(spec_file, spec_name)
```

В листинге 10.2 показаны два эмпирических теста: `test_calc_gaes` тестирует реализацию обобщенной оценки преимущества из главы 6, а `test_linear_decay` — функцию линейного уменьшения какой-либо переменной. Последнее часто используется в SLM Lab, например, для уменьшения переменных исследования ε или τ. Подобные эмпирические тесты имеют особое значение для сложных или труднореализуемых функций — они дают нам уверенность, что функция возвращает ожидаемые выходные значения.

Листинг 10.2. Примеры эмпирических тестов со сравнением вычисленных вручную по нескольким математическим формулам результатов с выходными значениями функции

```
 1 # slm_lab/test/lib/test_math_util.py
 2
 3 from slm_lab.lib import math_util
 4 import numpy as np
 5 import pytest
 6 import torch
 7
 8 def test_calc_gaes():
 9     rewards = torch.tensor([1., 0., 1., 1., 0., 1., 1., 1.])
10     dones = torch.tensor([0., 0., 1., 1., 0., 0., 0., 0.])
11     v_preds = torch.tensor([1.1, 0.1, 1.1, 1.1, 0.1, 1.1, 1.1, 1.1, 1.1])
12     assert len(v_preds) == len(rewards) + 1 # включение
       ➥ последнего состояния
13     gamma = 0.99
14     lam = 0.95
15     gaes = math_util.calc_gaes(rewards, dones, v_preds, gamma, lam)
16     res = torch.tensor([0.84070045, 0.89495, -0.1, -0.1, 3.616724,
       ➥ 2.7939649, 1.9191545, 0.989])
17     # используйте allclose вместо equal для учета atol
18     assert torch.allclose(gaes, res)
19
20 @pytest.mark.parametrize('start_val, end_val, start_step, end_step, step,
   ➥ correct', [
21     (0.1, 0.0, 0, 100, 0, 0.1),
22     (0.1, 0.0, 0, 100, 50, 0.05),
23      (0.1, 0.0, 0, 100, 100, 0.0),
24      (0.1, 0.0, 0, 100, 150, 0.0),
```

```
25     (0.1, 0.0, 100, 200, 50, 0.1),
26     (0.1, 0.0, 100, 200, 100, 0.1),
27     (0.1, 0.0, 100, 200, 150, 0.05),
28     (0.1, 0.0, 100, 200, 200, 0.0),
29     (0.1, 0.0, 100, 200, 250, 0.0),
30 ])
31 def test_linear_decay(start_val, end_val, start_step, end_step, step,
   ➥ correct):
32     assert math_util.linear_decay(start_val, end_val, start_step,
       ➥ end_step, step) == correct
```

Чтобы решить, какие тесты писать, полезно систематизировать их в соответствии со структурой ПО и его компонентами. Это один из поводов потратить время на улучшение архитектуры ПО. Например, при организации всех методов в один неструктурированный сценарий все будет сделано быстро и по-хакерски. Но этого делать не стоит, если предполагается более чем однократное использование кода, не говоря уже о долгосрочном проектировании промышленной системы. Хорошее ПО должно быть рационально разбито на отдельные компоненты, которые можно независимо разрабатывать, использовать и тестировать. На протяжении всей книги мы обсуждали организацию агента в виде компонентов, таких как классы памяти, самого алгоритма и нейронной сети. SLM Lab следует принципу компонентного проектирования, и для каждого компонента написаны тесты. Когда структура тестов соответствует структуре частей ПО, можно легко отследить, что было протестировано и, соответственно, что работает. Таким образом создается ментальная карта не покрытых тестами слабых мест, которые являются потенциальными кандидатами для возникновения ошибки. За счет этого существенно повышается эффективность отладки.

В листинге 10.3 показан пример компонентного теста для сверточной сети, которая проектируется и тестируется как независимый модуль. Он проверяет наличие характерных для сети элементов, таких как архитектура, структура данных и обновление модели, чтобы удостовериться, что эта реализация работает так, как задумывалось.

Листинг 10.3. Примеры компонентного теста для сверточной сети, элементы, присущие сети, разрабатываются и тестируются в независимом модуле

```
 1 # slm_lab/test/net/test_conv.py
 2
 3 from copy import deepcopy
 4 from slm_lab.env.base import Clock
 5 from slm_lab.agent.net import net_util
 6 from slm_lab.agent.net.conv import ConvNet
 7 import torch
 8 mport torch.nn as nn
 9
10 net_spec = {
11     "type": "ConvNet",
```

```
    "shared": True,
    "conv_hid_layers": [
        [32, 8, 4, 0, 1],
        [64, 4, 2, 0, 1],
        [64, 3, 1, 0, 1]
    ],
    "fc_hid_layers": [512],
    "hid_layers_activation": "relu",
    "init_fn": "xavier_uniform_",
    "batch_norm": False,
    "clip_grad_val": 1.0,
    "loss_spec": {
        "name": "SmoothL1Loss"
    },
    "optim_spec": {
        "name": "Adam",
        "lr": 0.02
    },
    "lr_scheduler_spec": {
        "name": "StepLR",
        "step_size": 30,
        "gamma": 0.1
    },
    "gpu": True
}
in_dim = (4, 84, 84)
out_dim = 3
batch_size = 16
net = ConvNet(net_spec, in_dim, out_dim)
# инициализация оптимизатора сети и способа
   ➥ изменения его скорости обучения lr
optim = net_util.get_optim(net, net.optim_spec)
lr_scheduler = net_util.get_lr_scheduler(optim, net.lr_scheduler_spec)
x = torch.rand((batch_size,) + in_dim)

def test_init():
    net = ConvNet(net_spec, in_dim, out_dim)
    assert isinstance(net, nn.Module)
    assert hasattr(net, 'conv_model')
    assert hasattr(net, 'fc_model')
    assert hasattr(net, 'model_tail')
    assert not hasattr(net, 'model_tails')

def test_forward():
    y = net.forward(x)
    assert y.shape == (batch_size, out_dim)

ef test_train_step():
   y = torch.rand((batch_size, out_dim))
   clock = Clock(100, 1)
    loss = net.loss_fn(net.forward(x), y)
```

```
62     net.train_step(loss, optim, lr_scheduler, clock=clock)
63     assert loss != 0.0
64
65 def test_no_fc():
66     no_fc_net_spec = deepcopy(net_spec)
67     no_fc_net_spec['fc_hid_layers'] = []
68     net = ConvNet(no_fc_net_spec, in_dim, out_dim)
69     assert isinstance(net, nn.Module)
70     assert hasattr(net, 'conv_model')
71     assert not hasattr(net, 'fc_model')
72     assert hasattr(net, 'model_tail')
73     assert not hasattr(net, 'model_tails')
74
75     y = net.forward(x)
76     assert y.shape == (batch_size, out_dim)
77
78 def test_multitails():
79     net = ConvNet(net_spec, in_dim, [3, 4])
80     assert isinstance(net, nn.Module)
81     assert hasattr(net, 'conv_model')
82     assert hasattr(net, 'fc_model')
83     assert not hasattr(net, 'model_tail')
84     assert hasattr(net, 'model_tails')
85     assert len(net.model_tails) == 2
86
87     y = net.forward(x)
88     assert len(y) == 2
98     assert y[0].shape == (batch_size, 3)
90     assert y[1].shape == (batch_size, 4)
```

Модульные тесты важны, поэтому их следует писать часто и они не должны быть сложными. Фактически чем они проще, тем лучше. Хорошие тесты должны быть короткими и понятными и при этом охватывать все важные аспекты тестируемых функций. Кроме того, они должны быть быстрыми и устойчивыми, поскольку часто применяются как показатель качества вновь разработанного кода и тем самым влияют на продолжительность цикла разработки. Функции можно доверять на основании тестов, которые могут понятно и быстро показать, как она работает. Если обеспечены достоверность и надежность, то дальнейшие исследование и проектирование будут протекать более эффективно.

10.1.2. Качество кода

Модульные тесты необходимы, но их недостаточно для разработки хорошего ПО — нужно еще и чтобы сам код был хорошего качества. Он не только передает инструкции компьютеру, но и содержит идеи для программистов. Хороший код легко понять и с ним просто работать как его авторам, так и всем, кто с ними сотрудничает. Если мы не в состоянии понять отрывок кода, написанный собственноручно три месяца назад, то это плохой код, который трудно поддерживать.

Стандартная практика обеспечения качества кода при проектировании ПО — внедрение рекомендаций из руководства по стилю оформления кода и анализ последнего. Для языка программирования *руководство по стилю оформления кода* — это набор лучших практик и соглашений по написанию кода. Он содержит рекомендации от общего синтаксиса форматирования и соглашений об именовании до конкретных указаний, что можно и чего нельзя делать, чтобы создавать безопасный и высокопроизводительный код. Руководства по стилю оформления кода обычно создаются большими сообществами программистов в целях распространения общепринятых соглашений. Благодаря этому не только код становится более понятным при совместной работе, но и повышается его качество в целом.

Руководства по стилю оформления кода постоянно изменяют, чтобы поспевать за развитием языков программирования и сменой требований их сообществ. Обычно их размещают на GitHub как документы с открытой лицензией, которые поддерживаются с помощью краудсорсинга. SLM Lab в качестве основного руководства по стилю оформления кода использует *Google Python Style Guide* (https://github.com/google/styleguide), а в качестве дополнительного — *Python Style Guide* (https://github.com/kengz/python) от Ван Лун Кенга.

Чтобы программистам было легче писать хороший код, современные руководства по стилю оформления кода преобразуют в программы, которые называются *линтерами*. Они работают с текстовыми редакторами, обеспечивая визуальные подсказки и автоформатирование, что позволяет соблюдать рекомендации. Эти же программы применяются и для автоматизированного анализа кода, что подводит нас к следующей теме.

С помощью постоянного анализа гарантируется качество кода, добавленного в репозиторий ПО. Обычно это подразумевает, что один или несколько человек проверяют вновь добавленный код на логическую корректность и строгое соблюдение руководства по стилю оформления кода. Крупные платформы хостинга кода, такие как GitHub, поддерживают анализ кода. Перед принятием каждого нового фрагмента кода его сначала пересматривают посредством того, что известно как запрос на *принятие изменений* (pull request). Сейчас этот рабочий процесс анализа кода — стандартная практика разработки ПО.

За последние несколько лет автоматизированный анализ кода широко распространился с помощью компаний, предоставляющих бесплатные и платные облачные сервисы для проверки качества кода, такие как Code Climate [27] и Codacy [26]. Как правило, эти инструменты выполняют статический анализ кода с проверкой нескольких элементов: строгого соблюдения руководства по стилю оформления кода, проблем безопасности, сложности кода и покрытия тестами. Данные инструменты используются как привратники, гарантирующие, что только код, прошедший этап анализа, может быть принят и присоединен к репозиторию. Все, что рассматривается как проблема, относят к тому, что называется *техническим долгом*. Подобно денежному долгу, технический долг рано или поздно должен

быть погашен. Он также растет: по мере расширения проекта ПО неразрешенные проблемы, связанные с плохим кодом, проникают во все большее число мест. Чтобы обеспечить работоспособность проекта в долгосрочной перспективе, важно держать технический долг под контролем.

10.1.3. Рабочий процесс Git

Git — это современный инструмент контроля версий исходного кода. Его основная идея проста: каждое добавление и удаление кода должно производиться как внесение отслеживаемых изменений в кодовую базу. Это означает версионирование с помощью сообщения с описанием того, что было сделано в коммите. Если изменение кода нарушает работу ПО, то можно легко и быстро вернуться к предыдущей версии. Кроме того, различия между коммитами можно использовать при отладке. Просмотреть их можно с помощью команды `git diff` или на странице pull request на GitHub. Если в новом коммите ПО внезапно дает сбой, то это вызвано, скорее всего, последними изменениями в коде. Рабочий процесс Git — это просто поступательное внесение в код изменений с понятными сообщениями, а затем рецензирование коммитов перед их присоединением к основной кодовой базе.

Ввиду сложности ПО для глубокого обучения и того, как легко нарушить его работу, реализация рабочего процесса Git может оказаться чрезвычайно полезной. Он был незаменимым подспорьем при разработке SLM Lab. Он приносит пользу не только в случае функционального кода, но и для определения гиперпараметров алгоритма. На рис. 10.1 продемонстрирован пример представления различий в Git, где файл `spec` для агента двойной DQN настраивается путем изменения нормы для усечения градиентов, оптимизатора нейронной сети и типа обновления сети. Если в результате производительность агента повысилась или понизилась, то известно почему.

Рабочий процесс Git позволяет быть уверенными в воспроизводимости экспериментов в SLM Lab. Любой результат можно воспроизвести с помощью SHA из Git. Он используется для проверки точной версии кода, которая применялась при запуске эксперимента, и возврата к коду именно в том виде, в котором он был написан. Без рабочего процесса Git нам пришлось бы прибегнуть к изменению кода вручную, что быстро могло бы стать затруднительным из-за наличия в глубоком RL множества изменяющихся частей.

Основная часть выполняемых вручную действий при анализе кода предусматривает просмотр в Git различий, аналогичных представленным на рис. 10.1. В pull request накапливается большое количество действительных коммитов для новых функций или исправления ошибок. Затем подводятся итоги изменений, которые выводятся на странице для рецензентов и авторов для проверки и обсуждения. Так происходит поступательная разработка ПО. Рабочий процесс Git в совокупно-

сти с модульными тестами и руководствами по стилю оформления кода формирует основы современного проектирования ПО.

Рис. 10.1. Снимок экрана с представлением различий в Git для запроса на принятие изменений в SLM Lab и показом в параллельном режиме кода до и после изменений

Модульные тесты, руководства по стилю оформления кода и рецензирование кода помогают управлять реализацией сложных проектов ПО. Они облегчают построение большого и сложного ПО и помогают сохранить корректность выполнения при его разрастании. Из-за сложности алгоритмов глубокого RL целесообразно внедрять эти практики, как было сделано в SLM Lab. Чтобы узнать о специфике их применения к глубокому RL, рассмотрим несколько примеров pull request из репозитория для SLM Lab на GitHub. На рис. 10.2 приведен пример, на котором pull request автоматически запускает модульные тесты и проверку качества кода. До тех пор пока pull request не пройдет эту проверку, изменения в коде не будут одобрены.

Для того чтобы процесс построения сложного ПО был управляемым, необходимо качественно проектировать ПО. В этом разделе рассмотрены лишь несколько советов для тех, кто начал интересоваться этой обширной и глубокой темой. К сожалению, этому не учат на курсах по информатике. Хорошему проектированию можно научиться лишь в ходе непосредственной работы и накопления опыта. К счастью, благодаря изобилию сообществ и проектов с открытым исходным кодом его можно освоить, непосредственно участвуя в этих проектах. Если вы будете внимательны к коду, то получить такой навык будет легко. Исходя из этого, дадим несколько рекомендаций, как начать работать с глубоким RL.

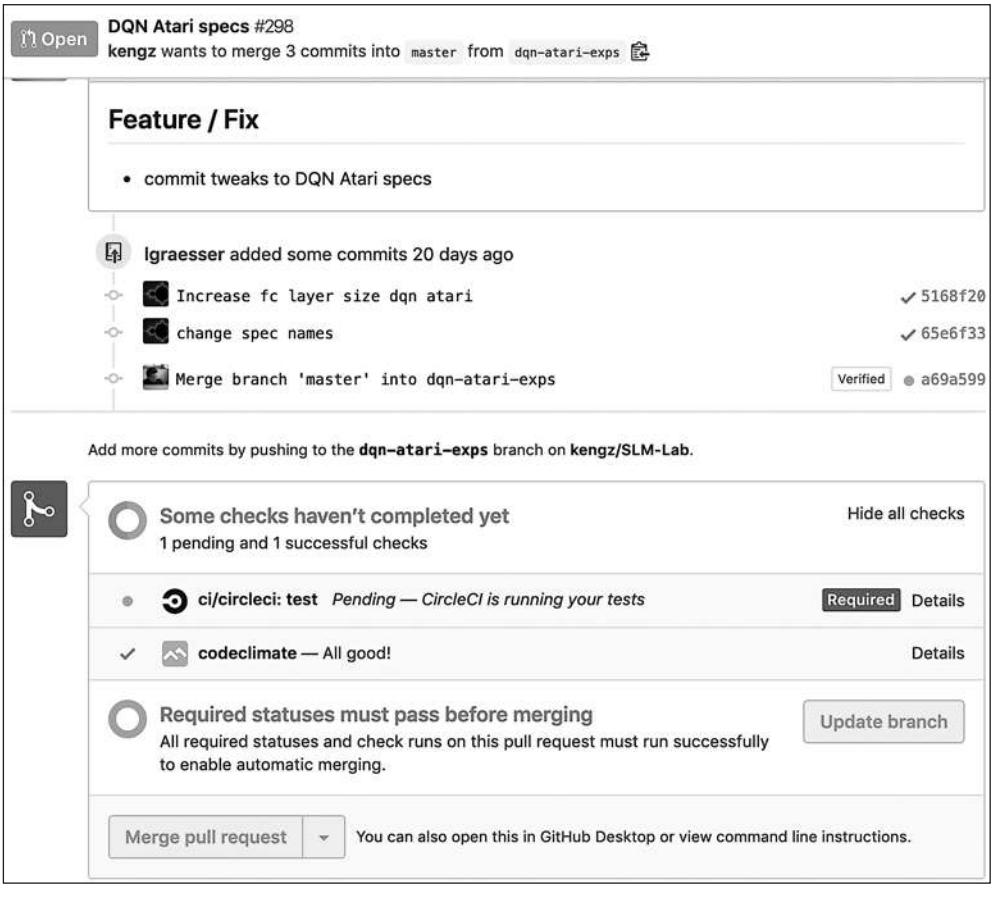


Рис. 10.2. Снимок экрана с pull request в SLM Lab, в котором производится запуск модульных тестов на удаленном сервере с автоматической проверкой качества кода с помощью Code Climate

10.2. Рекомендации по отладке

В этом разделе будут рассмотрены несколько рекомендаций по отладке, которые могут помочь сделать алгоритмы RL работоспособными. Из-за своих сложности и новизны отладка в глубоком RL остается скорее искусством, чем наукой. Для этого необходим существенный опыт практической работы с учетом особенностей ПО глубокого обучения, численных вычислений и аппаратного обеспечения. И самое главное: как и в любом сложном проекте, чтобы все заработало, потребуется проявить немалую *настойчивость*.

Главная цель отладки — определение первопричины сбоя. Этот процесс заключается в фактической изоляции проблемы, когда ошибку ищут, систематически проверяя разные подозрительные места. Важно, чтобы это происходило последовательно. Следует составить список предположений и подозрительных мест в порядке их приоритетности, затем протестировать их по одному, по возможности изолированно. После каждого неудачного теста очередной кандидат выбывает, а следующее предположение уточняется. Если повезет, то это быстро покажет первопричину сбоя.

Далее с помощью проекта SLM Lab в общих чертах обозначается основной план действий при отладке в RL.

10.2.1. Признаки жизни

Чтобы узнать, работает ли алгоритм глубокого RL, нужно проверить несколько основных «признаков жизни». Простейшим индикатором являются *полное вознаграждение* и его скользящее среднее. В рабочем алгоритме вознаграждения должны быть выше, чем у случайно действующего агента. Если они приблизительно равны или ниже, значит, алгоритм не работает. В задачах, в которых успех соотносится с количеством шагов, дополнительно проверьте длину эпизода. Если задание основано на быстром прохождении, то лучше короткие эпизоды, если оно включает множество этапов, то длительные.

Кроме того, если *скорость обучения* или *переменная исследования* уменьшаются со временем, нужно еще и прочитать вывод системы отладки, чтобы проверить, происходило ли это уменьшение правильно. Например, если в ε-жадной стратегии переменная ε не уменьшается из-за какой-нибудь ошибки, то алгоритм всегда будет действовать случайным образом. При запуске сессии в SLM Lab выполняется периодическое журналирование этих переменных, как показано в отрывке журнала в листинге 10.4.

Листинг 10.4. Пример журналирования диагностических переменных при запуске сессии в SLM Lab

```
1 [2019-07-07 20:42:55,791 PID:103674 INFO __init__.py log_summary] Trial 0
  ➥ session 0 dqn_pong_t0_s0 [eval_df] epi: 0  t: 0  wall_t: 48059  opt_step:
  ➥ 4.775e+06  frame: 3.83e+06  fps: 79.6937  total_reward: 9.25
  ➥ total_reward_ma: 14.795  loss: 0.00120682  lr: 0.0001  explore_var: 0.01
  ➥ entropy_coef: nan  entropy: nan  grad_norm: nan
2 [2019-07-07 20:44:51,651 PID:103674 INFO __init__.py log_summary] Trial 0
  ➥ session 0 dqn_pong_t0_s0 [train_df] epi: 0  t: 3.84e+06  wall_t: 48178
  ➥ opt_step: 4.7875e+06  frame: 3.84e+06  fps: 79.7044  total_reward: 18.125
  ➥ total_reward_ma: 18.3331  loss: 0.000601919  lr: 0.0001  explore_var: 0.01
  ➥ entropy_coef: nan  entropy: nan  grad_norm: nan
```

10.2.2. Диагностирование градиента стратегии

Для методов градиента стратегии также существуют характерные для стратегии переменные, которые можно контролировать. Это количественные величины, такие как *энтропия*, которые можно вычислить по распределению вероятностей. Чаще всего в начале обучения стратегия носит случайный характер, поэтому энтропия распределения вероятностей действий должна быть близка к теоретическому максимуму. Конкретные значения будут разниться в зависимости от размера и типа пространства действий. Если энтропия не убывает, значит, стратегия остается случайной и обучения не происходит. Когда она уменьшается слишком быстро, обучение агента также может быть неправильным. Это показывает, что агент прекратил исследование пространства действий и выбирает действия с очень высокой вероятностью. Опыт показывает, что если обучение агента длится 1 млн шагов, но энтропия вероятности выбора действий быстро снижается после первых 1000 шагов, то весьма маловероятно, что агент обучится оптимальной стратегии.

Вероятность действия или логарифм вероятности тесно связаны с энтропией распределения вероятностей действий. Они показывают вероятность выбора действия. В начале обучения вероятности действий должны приближаться к случайным однородным. По мере обучения агента лучшей стратегии выбор действия должен происходить более обдуманно, следовательно, вероятность этого действия будет в среднем выше.

Другой полезный показатель — *расстояние Кульбака — Лейблера*, которым измеряется «размер» обновления стратегии. Если KL мало, значит, стратегия между обновлениями также изменилась незначительно, то есть прогресс в обучении медленный или несущественный. Но если KL внезапно становится очень большим, это говорит нам: только что произошло значительное изменение стратегии, которое может вызвать резкое падение производительности, как обсуждалось в главе 7. В SLM Lab некоторые из этих переменных также заносятся в журнал, как показано в листинге 10.4.

10.2.3. Диагностирование данных

Крайне важно обеспечить корректность данных, особенно если информация, которой обмениваются среда и агент, претерпевает многочисленные преобразования. Каждый шаг — потенциальный источник ошибок. Часто очень полезно проводить *отладку данных* вручную. На разных шагах преобразования можно отследить и проверить состояние, действие и вознаграждение. Для состояний и вознаграждений поступающие из среды исходные данные предварительно обрабатываются, сохраняются агентом и затем используются в обучении. Для действий порожденные нейронной сетью данные объединяются и, возможно, предварительно обраба-

тываются, а затем передаются обратно в среду. Значения переменных на каждом шаге можно распечатать и проверить вручную. После того как вы много раз просмотрите их, у вас появится интуитивное понимание исходных данных. Это поможет при отборе полезной отладочной информации и определении того, что выглядит неправильно.

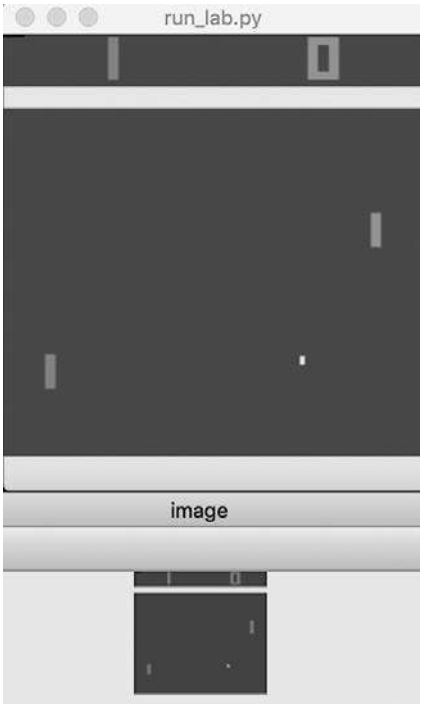


Рис. 10.3. Пример отладки изображения для Atari Pong в SLM Lab. Вверху показано необработанное цветное изображение, а внизу для сравнения — предварительно обработанное (уменьшенное и в оттенках серого). Можно проходить по кадрам, нажимая любую кнопку

Для состояния в виде изображения следует не только вывести на печать численную информацию, но и взглянуть на его *визуализацию*. Полезно визуализировать как исходное изображение из среды, так и предварительно обработанную картинку, подаваемую на вход алгоритма. Если расположить их рядом и сравнить, то на стадии предварительной обработки будет проще определить малозаметные ошибки, которые способны привести к потере или ошибочности информации, представляемой алгоритму. Пошаговое прохождение по изображениям или видео при запуске среды также помогает увидеть, как агент выполняет задание. Таким образом можно обнаружить гораздо больше информации, чем если просматривать только количество пикселов или журналы. В листинге 10.5 приведен метод из SLM Lab для визуализации предварительно обработанного изображения. Он применяется для порождения уменьшенного изображения в оттенках серого, показанного для сравнения на рис. 10.3 ниже первоначального изображения в формате RGB из среды. В этом режиме процесс приостанавливается, чтобы пользователь просмотрел изображения, и возобновляется после нажатия любой кнопки.

Листинг 10.5. Метод визуализации предварительно обработанного изображения для отладки в SLM Lab. При его вызове на предварительно обработанном состоянии в виде изображения то, что видит агент, будет визуализироваться и сравниваться с тем, что порождено средой

```
# slm_lab/lib/util.py

import cv2

def debug_image(im):
    '''
```

```
    Воспользуйтесь этим методом для визуализации того, что видит агент,
    перед продолжением выполнения ожидается нажатие любой кнопки
    '''
    cv2.imshow('image', im.transpose())
    cv2.waitKey(0)
```

10.2.4. Предварительная обработка

Состояния и вознаграждения из среды перед их сохранением в памяти агента или применением для обучения обычно предварительно обрабатываются. Помимо просмотра данных вручную нужно проверить функции, порождающие данные. Если предварительно обработанные данные выглядят неправильно, значит, *препроцессор* некорректно реализует некоторые преобразования. Возможно, неправильны форма или ориентация выходных данных или в результате ошибочного приведения типов все данные были сведены к 0. Одна небольшая особенность обработки данных изображений заключается в различии в соглашениях о *порядке следования каналов изображения*. По сложившейся традиции проектирования в большинстве библиотек компьютерного зрения каналы изображения находятся в конце, поэтому оно имеет структуру (ширину, высоту, канал). В соглашении PyTorch из соображений эффективности используются обратный порядок и структура изображения (канал, высота, ширина). В изображениях, порожденных средой в OpenAI Gym, применяется старое соглашение, поэтому их нужно правильно обрабатывать перед передачей в сверточную сеть PyTorch.

10.2.5. Память

Память агента работает как хранилище данных. Сначала проверяются корректность структуры и тип данных. Нужно также гарантировать правильный порядок данных. Это особо критично при порождении последовательностей данных для рекуррентной сети. В обучении по актуальному опыту жизненно важно обеспечить исправную очистку памяти после каждого шага обучения. При выборке данных для обучения проверяйте также корректность методов, применяемых для генерации случайных индексов и получения данных. У более продвинутых структур памяти, таких как приоритизированная память прецедентов, есть свои алгоритмы выборки и структуры данных, которые нужно тестировать отдельно.

10.2.6. Алгоритмические функции

Если порождение данных и их сохранение прошли корректно, то нужно отладить начальные алгоритмические функции, в которые они передаются. Функции RL, такие как уравнение Беллмана или обобщенная оценка преимущества (GAE), могут быть довольно сложными, а значит, возможны ошибки при их реализации.

Эмпирические модульные тесты, аналогичные примерам, приведенным в листинге 10.2, должны выполняться путем сравнения рассчитанных вручную значений с выходными данными, полученными для реализации. Порой ошибка может быть обусловлена проблемами с индексами массива, численными ошибками, неучтенными частными случаями или просто неверным приведением типов. Следует также распечатывать и *проверять вручную* входные и выходные значения функции преимущества, функций *V*- или *Q*-значений и вычисленные значения функции потерь. Проверьте порядок этих значений, их изменение в течение времени и посмотрите, не выглядят ли они странно. Зачастую эти вычисления тесно связаны с нейронными сетями.

10.2.7. Нейронные сети

Когда алгоритм RL не работает, стоит взглянуть на нейронную сеть, поскольку именно в ней происходит обучение. Проще всего проверить *архитектуру* сети. Размерности входных и выходных значений должны быть корректными. Скрытые слои должны быть правильного типа и нужного размера. Функции активации также должны применяться должным образом. Любые сжимающие функции, такие как гиперболический тангенс или сигмоид, станут ограничивать диапазон значений, из-за этого у выходных слоев будут отсутствовать функции активации. Единственное исключение — сети стратегии, выдающие на выходе вероятности действий. В них функции активации могут применяться к выходному слою. Кроме того, алгоритмы глубокого RL известны своей чувствительностью к параметрам инициализации сети по причине влияния последних на начальные условия обучения.

Диапазоны значений входных и выходных данных тоже нужно проверять. Нейронные сети довольно чувствительны к разным масштабам входных значений, поэтому общепринятой практикой является нормализация входных значений сети по измерениям. Во время обучения желательно избегать *быстро растущих функций потерь*. Если на выходе сети вдруг появляется значение `NaN`, то очень вероятно, что некоторые из параметров сети стали бесконечными из-за серьезных обновлений, связанных с чрезмерно большими значениями функции потерь. Тогда нужно исправить вычисление потерь. К тому же есть хорошее практическое правило — всегда использовать *усечение градиентов* с нормой от 0,5 до 1,0, так как это позволяет избежать значительных обновлений параметров.

Затем проследите, чтобы на шагах обучения действительно происходило обновление *параметров сети*. Если параметры остаются неизменными после множества этапов обучения с ненулевыми значениями функции потерь, то проверьте *граф вычислений*. Обычно причина заключается в отсоединении графа — в каком-то месте цепочки вычислений от входных до выходных значений вдруг прекратилось распространение градиентов, поэтому обратное распространение не может быть

в полной мере применено ко всем соответствующим операциям. В современных фреймворках глубокого обучения есть встроенные проверки, сигнализирующие об отсоединении графа вычислений и показывающие, правильно ли вычисляются градиенты. В SLM Lab реализован метод-декоратор для шагов обучения, который в режиме разработки автоматически проверяет наличие обновлений параметров сети. Это показано в листинге 10.6.

Листинг 10.6. В SLM Lab реализован метод-декоратор, который может быть использован совместно с методом определения шага обучения. В режиме разработки он автоматически проверяет наличие обновлений параметров сети и выдает ошибку, если проверка завершилась неудачей

```
# slm_lab/agent/net/net_util.py

from functools import partial, wraps
from slm_lab.lib import logger, optimizer, util
import os
import pydash as ps
import torch
import torch.nn as nn

logger = logger.get_logger(__name__)

def to_check_train_step():
    '''Условие для запуска assert_trained'''
    return os.environ.get('PY_ENV') == 'test' or util.get_lab_mode() == 'dev'

def dev_check_train_step(fn):
    '''
    Декоратор, проверяющий, действительно ли на net.train_step происходит
    ➥ правильное обновление весов сети
    Срабатывает, только если to_check_train_step равно True
    ➥ (режим dev/test)
    @example

    @net_util.dev_check_train_step
    def train_step(self,...):
        ...
    '''
    @wraps(fn)
    def check_fn(*args, **kwargs):
        if not to_check_train_step():
            return fn(*args, **kwargs)

        net = args[0] # сначала сеть задает свои параметры
        # получить параметры до обновления для  сравнения
        pre_params = [param.clone() for param in net.parameters()]

        # запустить train_step, получить значения функции потерь
        loss = fn(*args, **kwargs)
```

```
        assert not torch.isnan(loss).any(), loss

        # получить параметры после обновления для сравнения
        post_params = [param.clone() for param in net.parameters()]
        if loss == 0.0:
            # если значения функции потерь нулевые,
            ➥ то обновление не должно происходить
            for p_name, param in net.named_parameters():
                assert param.grad.norm() == 0
        else:
            # проверить обновления параметров
            try:
                assert not all(torch.equal(w1, w2) for w1, w2 in
                ➥ zip(pre_params, post_params)),f'Параметр модели
                ➥ не был обновлен на train_step(), проверьте,
                ➥ не отключился ли тензор от графа. Потери: {loss:g}'
                logger.info(f'Параметр модели обновлен на train_step().
                ➥ Потери: {loss: g}')
            except Exception as e:
                logger.error(e)
                if os.environ.get('PY_ENV') == 'test':
                    # если выполняется модульный тест, сгенерировать
                    ➥ исключение
                    raise(e)

            # проверить нормы градиентов
            min_norm, max_norm = 0.0, 1e5
            for p_name, param in net.named_parameters():
            try:
                grad_norm = param.grad.norm()
                assert min_norm < grad_norm < max_norm, f'Норма градиента
                ➥ для {p_name}, равная {grad_norm:g}, выходит
                ➥ за предельные значения {min_norm} < grad_norm
                ➥ < {max_norm}. Функция потерь: {loss:g}.
                ➥ Проверьте вашу сеть и вычисление функции потерь.'
            except Exception as e:
                logger.warning(e)
        logger.info(f'Значения норм градиентов успешно прошли проверку.')
    logger.debug('Проверка обновлений параметров прошла успешно.')
    # сохранить нормы градиентов для отладки
    net.store_grad_norms()
    return loss
return check_fn
```

Если функция потерь рассчитывается как распределение, полученное из множества отдельных значений потерь, то каждую из них можно проверить, чтобы гарантировать, что все они порождают корректное поведение при обучении и обновлении параметров. Чтобы проверить отдельно взятые *функции потерь*, просто отключите другие функции потерь и запустите несколько шагов обучения. Например, в алгоритме актора-критика с общей сетью функция потерь — это сумма функций потерь

для стратегии и полезности. Чтобы проверить первые, просто устраните потери, связанные с полезностями, и запустите обучение, чтобы удостовериться, что сеть обновляется по функции потерь для стратегии. Затем, отключив вторые потери, сделайте то же самое для проверки функции потерь для полезностей.

10.2.8. Упрощение алгоритма

Зачастую алгоритмы глубокого RL состоят из многих компонентов, и трудно заставить их все работать одновременно. Для сложных задач существует проверенный метод — *упрощение*: начать с самого необходимого, а потом добавлять остальные компоненты. Мы видели, что различные алгоритмы глубокого RL являются расширениями и модификациями более простых алгоритмов, как показано в дереве семейства алгоритмов RL на рис. 9.1. Например, PPO — это улучшение A2C посредством изменения функции потерь для стратегии, а A2C, в свою очередь, — это преобразование REINFORCE. Мы можем с пользой для себя применить понимание связей между алгоритмами. Если реализация PPO не работает, то можно отключить функцию потерь PPO, чтобы превратить его в A2C, и сосредоточиться на том, чтобы сначала добиться работоспособности последнего. Затем мы можем снова включить расширение и отладить весь алгоритм.

Этот метод применим и при построении алгоритма с нуля. Например, можно сначала реализовать REINFORCE, а когда он заработает — расширить его до реализации A2C. Когда и тот станет рабочим, можно расширить его до реализации PPO. Это вписывается в модель наследования классов, в которой более сложный класс расширяет более простой базовый класс. Кроме того, данный подход позволяет использовать компоненты повторно, а значит, сократить количество нового кода при отладке.

10.2.9. Упрощение задачи

При реализации и отладке алгоритма часто бывает целесообразно сначала протестировать его на задачах попроще, таких как CartPole. Обычно для более простых задач требуется меньше настроек гиперпараметров, а их решение менее затратно с точки зрения вычислений. Например, обучение агента игре CartPole может быть запущено на скромном ЦПУ в течение десятков минут, тогда как для запуска обучения агента в игре Atari может понадобиться несколько графических процессоров и уйдут на это часы или даже дни. Сосредоточившись сначала на более простых задачах, можно ускорить процессы отладки и разработки, ведь каждая итерация тестов становится намного быстрее.

Но даже если все работает при решении более простых задач, то не обязательно прямо переносится на более трудные проблемы. В коде все равно могут оставаться какие-нибудь ошибки. Кроме того, для более сложных задач могут потребоваться новые преобразования. Например, в CartPole используется простое состояние с четырьмя элементами, и для решения ее задачи нужна сеть с многослойным перцептроном. Но в игре Atari состояния — это изображения, и необходима сверточная сеть, совмещенная с какой-нибудь предварительной обработкой изображений и преобразованиями среды. Эти новые компоненты все так же нужно отлаживать и тестировать, но большая часть основных компонентов будет уже протестирована, что значительно упростит отладку.

10.2.10. Гиперпараметры

Несмотря на то что глубокое RL может быть весьма чувствительным к гиперпараметрам, зачастую их настройка — это лишь малая часть процесса отладки, если мы реализуем существующий алгоритм. Нередко можно обратиться за рабочими гиперпараметрами к исследовательским статьям или реализациям и проверить, достигает ли с ними наш агент сопоставимой производительности. После того как исправлены основные ошибки, можно заняться тонкой настройкой гиперпараметров и попытаться добиться наилучших результатов. Для справки: в разделе 10.4 приведены подробные сведения об эффективных значениях гиперпараметров для разных алгоритмов и сред, обсуждаемых в этой книге.

10.2.11. Рабочий процесс в SLM Lab

Глубокое обучение с подкреплением по большей части остается эмпирической наукой, поэтому вполне естественно принять на вооружение рабочий процесс научно-исследовательской деятельности. Это подразумевает выдвижение некоторых гипотез, определение переменных и проведение экспериментов для получения результатов. На завершение каждого из экспериментов могут уйти дни или недели, поэтому, выбирая, какой эксперимент запустить, нужно руководствоваться определенной стратегией. Подумайте, какие из экспериментов могут быть приоритизированы, какие проводиться параллельно и какие наиболее перспективны. Чтобы отслеживать все эксперименты, ведите лабораторные записи с отчетами о результатах. В SLM Lab это можно делать, создавая коммиты с кодом, используемым для эксперимента, и записывая в информационный отчет результаты эксперимента вместе с SHA из Git. Все отчеты вносятся как pull request в GitHub, и их можно получить вместе с данными, необходимыми для воспроизведения эксперимента. Пример отчета из SLM Lab показан на рис. 10.4.

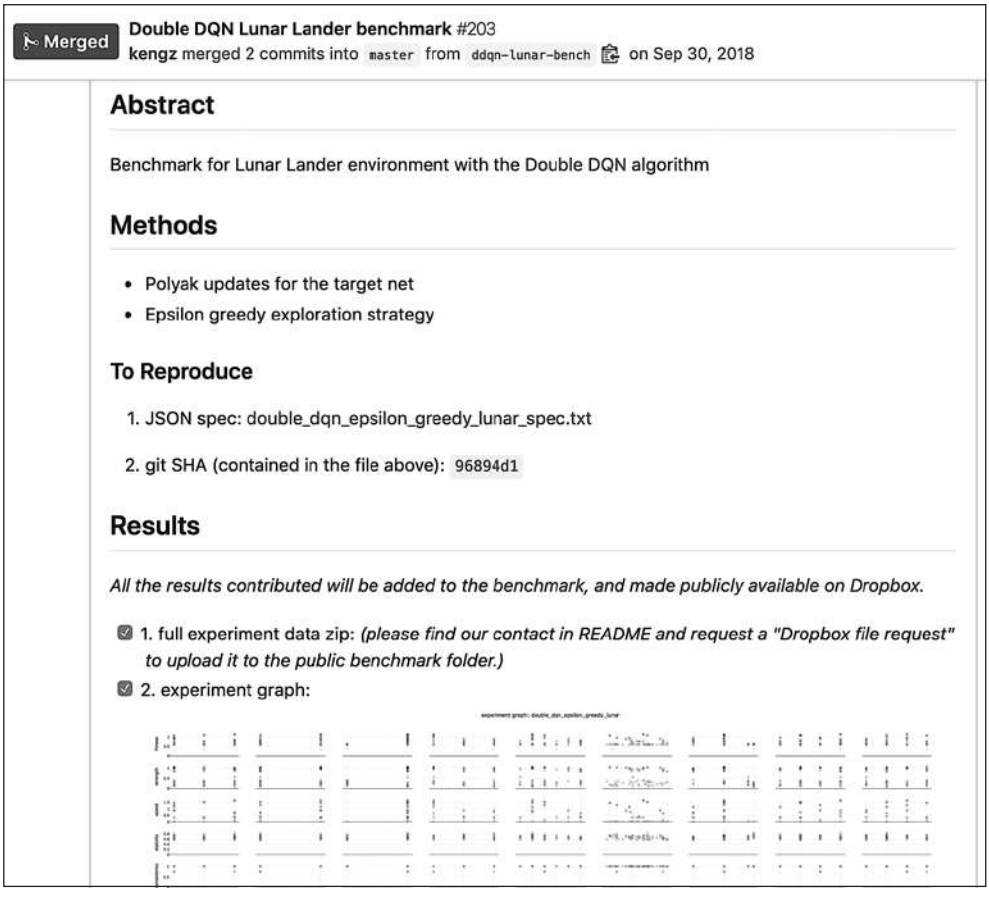


Рис. 10.4. Снимок экрана в SLM Lab с pull request с отчетом об эксперименте. В запрос входят описание методов, Git SHA, файл spec, графики и другие необходимые для воспроизведения данные

Советы, приведенные в этом разделе, не являются исчерпывающими — баги могут возникать самыми разными способами. Однако эти советы помогут начать отладку, тогда как в остальных случаях придется проводить эксперименты вручную. Не стоит также отказываться от обращения к существующим реализациям, написанным другими людьми, так как они могут пролить свет на те моменты, которые мы иначе упустили бы. Просто будьте добропорядочными гражданами страны открытого исходного кода и как следует благодарите других людей за их работу.

Для хорошей отладки необходимо продуманно подходить к построению гипотез и решению проблем, что проще сделать, придерживаясь хороших практик проектирования. Многие проблемы требуют углубленного рассмотрения кода, и нередко на то, чтобы алгоритм RL заработал, уходят недели или даже месяцы. Для человека

основные составляющие — это позитивный настрой и огромное упорство. Если верить в успех и проявлять настойчивость, то рано или поздно все наладится, и, когда это произойдет, награда будет достойной.

10.3. Практические приемы в играх Atari

В этом разделе мы рассмотрим несколько особых приемов для достижения хорошей производительности в средах Atari. Сначала обсудим разные версии сред, доступные в OpenAI Gym, а затем рассмотрим шаги предварительной обработки информации из среды, ставшие стандартной практикой.

В OpenAI Gym предлагаются многочисленные вариации сред Atari. Например, в нем есть шесть версий среды Pong.

1. Pong-v4.
2. PongDeterministic-v4.
3. PongNoFrameskip-v4.
4. Pong-ram-v4.
5. Pong-ramDeterministic-v4.
6. Pong-ramNoFrameskip-v4.

Все они разные с точки зрения внутренней реализации, но в их основе лежит одна и та же игра. Например, *NoFrameskip* подразумевает, что из среды возвращаются необработанные кадры, поэтому пользователи должны реализовать собственный механизм пропуска кадров. Среда *ram* возвращает данные из оперативной памяти как состояние, а не как изображение. В большей части исследовательской литературы для Pong используются среды с *NoFrameskip* или *PongNoFrameskip-v4*.

При использовании версии *NoFrameskip* предварительная обработка данных в основном реализуется и управляется пользователями. Часто она написана как обертка над исходной средой. Репозиторий GitHub для Baselines из OpenAI остается основным справочником по оберткам для отдельных сред. В SLM Lab многие из них также применяются в модуле `wrapper` в `slm_lab/env/wrapper.py`, как показано в листинге 10.7. Все эти обертки служат для учета каких-нибудь специфических особенностей сред Atari, которые обсуждаются в этом коротком разделе.

Листинг 10.7. В SLM Lab внедрены обертки для предварительной обработки информации среды из OpenAI Baselines. Эти методы используются для создания сред Atari, в названии которых есть часть NoFrameskip, для обучения и оценки алгоритмов. Сюда не включено большинство отрывков кода, но они есть в исходном файле в SLM Lab

```
1 # slm_lab/env/wrapper.py
2
3 ...
```

```
 4
 5 def wrap_atari(env):
 6     '''Применение типичного набора оберток для игр Atari'''
 7     assert 'NoFrameskip' in env.spec.id
 8     env = NoopResetEnv(env, noop_max=30)
 9     env = MaxAndSkipEnv(env, skip=4)
10     return env
11
12 def wrap_deepmind(env, episode_life=True, stack_len=None):
13     '''Обертка для среды Atari в стиле DeepMind'''
14     if episode_life:
15         env = EpisodicLifeEnv(env)
16     if 'FIRE' in env.unwrapped.get_action_meanings():
17         env = FireResetEnv(env)
18     env = PreprocessImage(env)
19     if stack_len is not None: # применить конкатенацию
       ➥ для изображения (1, 84, 84)
20         env = FrameStack(env, 'concat', stack_len)
21     return env
22
23 def make_gym_env(name, seed=None, frame_op=None, frame_op_len=None,
   ➥ reward_scale=None, normalize_state=False):
24     '''Общий метод для создания любой среды Gym;
       ➥ автоматически обертывает Atari'''
25     env = gym.make(name)
26     if seed is not None:
27         env.seed(seed)
28     if 'NoFrameskip' in env.spec.id: # Atari
29         env = wrap_atari(env)
30         # для мониторинга не производится усечение
           ➥ вознаграждений — они усекаются в памяти Atari
31         episode_life = not util.in_eval_lab_modes()
32         env = wrap_deepmind(env, episode_life, frame_op_len)
33     elif len(env.observation_space.shape) == 3: # среда с состояниями
       ➥ в виде изображений
34         env = PreprocessImage(env)
35         if normalize_state:
36             env = NormalizeStateEnv(env)
37         if frame_op_len is not None: # применить конкатенацию
           ➥ для изображения (1, 84, 84)
38             env = FrameStack(env, 'concat', frame_op_len)
39     else: # среда с состояниями в векторной форме
40         if normalize_state:
41            env = NormalizeStateEnv(env)
42         if frame_op is not None:
43             env = FrameStack(env, frame_op, frame_op_len)
44         if reward_scale is not None:
45             env = ScaleRewardEnv(env, reward_scale)
46         return env
```

1. `NoopResetEnv`. При перезапуске игры Atari начальное состояние, передаваемое агенту, — это случайный кадр, взятый из первых 30 кадров в исходной среде. Служит для предотвращения запоминания среды агентом.
2. `FireResetEnv`. В некоторых играх при перезапуске нужно нажимать FIRE. Например, в Breakout требуется, чтобы при старте игрок запустил мяч. Данная обертка делает это один раз при перезапуске, чтобы FIRE не являлось действием агента во время игры.
3. `EpisodicLifeEnv`. Во многих играх Atari большое количество жизней. В основном агент лучше обучается, если все жизни воспринимаются как равноценные. Эта обертка расходует по одной жизни на каждый новый эпизод.
4. `MaxAndSkipEnv`. Для того чтобы агент обучился хорошей стратегии, разница между двумя последовательными состояниями слишком мала. Для увеличения этого различия агентам задается ограничение на выбор действия на каждом четвертом временном шаге[1], а выбранное действие повторяется в промежуточных кадрах, которые пропускаются этой оберткой. То, что агент расценивает как s_t и s_{t+1}, в действительности является s_t и s_{t+4}. Дополнительно эта обертка выбирает пикселы с максимальными значениями из всех соответствующих пикселов во всех пропущенных кадрах. Для каждого пиксела берется отдельный максимум. В некоторых играх одни объекты визуализируются только в четных кадрах, а другие — только в нечетных. В зависимости от частоты пропуска кадров это может привести к тому, что некоторые объекты будут полностью исключены из состояния. Выбор из пропущенных кадров пиксела с максимальным значением решает потенциальную проблему.
5. `ScaleRewardEnv`. Масштабирует вознаграждение за временной шаг. Наиболее часто применяемый коэффициент масштабирования для игр Atari усекает вознаграждение до значений −1, 0 и 1. Знак берется как у исходного значения, это помогает привести масштаб вознаграждений для разных игр к одному стандарту.
6. `PreprocessImage`. Обертка для предварительной обработки изображений с соблюдением соглашения об изображениях PyTorch (канал цвета идет первым). Она уменьшает изображение и переводит его в оттенки серого, затем меняет порядок осей, чтобы оно соответствовало соглашению о первом канале в PyTorch.
7. `FrameStack`. Для большинства агентов каждый кадр, поступающий на вход сети, формируется совмещением четырех следующих друг за другом изображений из игры. В этой обертке реализован эффективный метод совмещения кадров, который лишь преобразует данные в совмещенное изображение во время обучения. Совмещение кадров мотивировано тем, что в отдельно взятом кадре содержится не вся полезная информация о состоянии игры. Вспомните обсуждение различий между МППР и частично наблюдаемыми МППР в примечании 1.1 в главе 1.

[1] В пространстве Invaders для того, чтобы лазеры были видны, это значение изменено на действие через каждые три шага.

Игры Atari не являются идеальными МППР, поэтому состояние игры не может быть выведено из одного наблюдаемого состояния — изображения. Например, агент не может догадаться о скорости и направлении движения объектов в игре по единственному изображению. Тем не менее знать эти значения важно, принимая решение, как действовать, и это может стать решающим фактором в достижении хорошего счета или проигрыше в игре. Для решения этой проблемы предыдущие четыре кадра совмещаются перед передачей их агенту. Он может использовать совокупности кадров из четырех изображений для получения важных свойств, таких как движение объекта. Тогда размерность передаваемых агенту состояний становится (84, 84, 4).

Кроме того, для таких алгоритмов, как актор-критик и PPO, используются векторизированные среды. Это модифицированная среда, представляющая собой вектор из многочисленных параллельных экземпляров игры, запущенных на отдельных ЦПУ. Параллелизация позволяет улучшить процесс обучения за счет того, что выборка происходит намного быстрее и становится более разнообразной. Она также входит в модуль `slm_lab.env.vec_wrapper` с обертками сред в SLM Lab.

10.4. Справочник по глубокому обучению с подкреплением

В этом разделе представлен набор гиперпараметров для алгоритмов и сред, обсуждаемых в книге. На поиск хороших начальных значений для настройки гиперпараметров алгоритма может уйти много времени. Этот раздел задумывался просто как справочная информация, которая может быть полезной при настройке.

Мы разделили гиперпараметры для нескольких сред на категории по семействам алгоритмов. Данные таблицы показывают, какие из гиперпараметров значительнее меняются при смене среды. Решая новую задачу, нужно начинать с более чувствительных (сильно меняющихся) гиперпараметров.

В конце приводится краткое сравнение производительности разных алгоритмов в нескольких средах.

Более широкий набор гиперпараметров алгоритмов и сравнение производительности для них имеются в репозитории SLM Lab по ссылке https://github.com/kengz/SLM-Lab/blob/master/BENCHMARK.md.

10.4.1. Таблицы гиперпараметров

В этом разделе приводятся три таблицы гиперпараметров, скомпонованные в соответствии с основными видами алгоритмов. В столбцах показано несколько примеров сред, к которым может быть применен данный алгоритм (дискретное

или непрерывное управление) и которые выстроены слева направо в порядке возрастания сложности. Нужно отметить, что в соответствии с соглашением по алгоритмам для всех игр Atari используются одни и те же гиперпараметры. Исключением являются REINFORCE и SARSA, поскольку они не так уж часто встречаются на практике.

Гиперпараметры приведены для конкретного алгоритма, но к его вариациям применима большая часть их значений. В табл. 10.1 показаны гиперпараметры для двойной DQN с PER, их можно задействовать и для DQN, DQN с PER и двойной DQN, за одним исключением — при использовании PER нужно снизить скорость обучения (см. подраздел 5.6.1).

Таблица 10.1. Гиперпараметры для двойной DQN с PER в разных средах

Гиперпараметр/среда	**LunarLander**	**Atari**
algorithm.gamma	0,99	0,99
algorithm.action_policy	ε-жадная	ε-жадная
algorithm.explore_var_spec.start_val	1,0	1,0
algorithm.explore_var_spec.end_val	0,01	0,01
algorithm.explore_var_spec.start_step	0	10 000
algorithm.explore_var_spec.end_step	50 000	1 000 000
algorithm.training_batch_iter	1	1
algorithm.training_iter	1	4
algorithm.training_frequency	1	4
algorithm.training_start_step	32	10 000
memory.max_size	50 000	200 000
m memory.alpha	0,6	0,6
memory.epsilon	0,0001	0,0001
memory.batch_size	32	32
net.clip_grad_val	10	10
net.loss_spec.name	SmoothL1Loss	SmoothL1Loss
net.optim_spec.name	Adam	Adam
net.optim_spec.lr	0,00025	0,00025
net.lr_scheduler_spec.name	Отсутствует	Отсутствует
net.lr_scheduler_spec.frame	—	—
net.update_type	replace	replace
net.update_frequency	100	100

Продолжение ⇩

Таблица 10.1 *(продолжение)*

Гиперпараметр/среда	LunarLander	Atari
net.gpu	False	True
env.num_envs	1	16
env.max_frame	300 000	10 000 000

В табл. 10.2 приведены гиперпараметры для A2C с GAE. Этот же набор гиперпараметров может быть применен к A2C с оценкой преимущества по отдаче за n шагов, нужно лишь заменить специфический для GAE `algorithm.lam` на `algorithm.num_step_returns` для отдачи за n шагов.

Таблица 10.2. Гиперпараметры для A2C (GAE) в разных средах

Гиперпараметр/среда	LunarLander	BipedalWalker	Atari
algorithm.gamma	0,99	0,99	0,99
algorithm.lam	0,95	0,95	0,95
algorithm.entropy_coef_spec.start_val	0,01	0,01	0,01
algorithm.entropy_coef_spec.end_val	0,01	0,01	0,01
algorithm.entropy_coef_spec.start_step	0	0	0
algorithm.entropy_coef_spec.end_step	0	0	0
algorithm.val_loss_coef	1,0	0,5	0,5
algorithm.training_frequency	128	256	32
net.shared	False	False	True
net.clip_grad_val	0,5	0,5	0,5
net.init_fn	orthogonal_	orthogonal_	orthogonal_
net.normalize	False	False	True
net.loss_spec.name	MSELoss	MSELoss	MSELoss
net.optim_spec.name	Adam	Adam	RMSprop
net.optim_spec.lr	0,002	0,0003	0,0007
net.lr_scheduler_spec.name	Отсутствует	Отсутствует	Отсутствует
net.lr_scheduler_spec.frame	—	—	—
net.gpu	False	False	True
env.num_envs	8	32	16
env.max_frame	300 000	4 000 000	10 000 000

Наконец, в табл. 10.3 показаны гиперпараметры для PPO. Они доступны как файлы `spec` для SLM Lab в папке `slm_lab/spec/benchmark/`, которая, кроме того, содержит много других настроенных файлов `spec`. Их можно запустить непосредственно с помощью команд SLM Lab, которые будут рассмотрены в подразделе 11.3.1.

Таблица 10.3. Гиперпараметры для PPO в разных средах

Гиперпараметр/среда	LunarLander	BipedalWalker	Atari
algorithm.gamma	0,99	0,99	0,99
algorithm.lam	0,95	0,95	0,70
algorithm.clip_eps_spec.start_val	0,2	0,2	0,1
algorithm.clip_eps_spec.end_val	0	0	0,1
algorithm.clip_eps_spec.start_step	10 000	10 000	0
algorithm.clip_eps_spec.end_step	300 000	1 000 000	0
algorithm.entropy_coef_spec.start_val	0,01	0,01	0,01
algorithm.entropy_coef_spec.end_val	0,01	0,01	0,01
algorithm.entropy_coef_spec.start_step	0	0	0
algorithm.entropy_coef_spec.end_step	0	0	0
algorithm.val_loss_coef	1,0	0,5	0,5
algorithm.time_horizon	128	512	128
algorithm.minibatch_size	256	4 096	256
algorithm.training_epoch	10	15	4
net.shared	False	False	True
net.clip_grad_val	0,5	0,5	0,5
net.init_fn	orthogonal_	orthogonal_	orthogonal_
net.normalize	False	False	True
net.loss_spec.name	MSELoss	MSELoss	MSELoss
net.optim_spec.name	Adam	Adam	Adam
net.optim_spec.lr	0,0005	0,0003	0,00025
net.lr_scheduler_spec.name	Отсутствует	Отсутствует	LinearToZero
net.lr_scheduler_spec.frame	—	—	10 000 000
net.gpu	False	False	True
env.num_envs	8	32	16
env.max_frame	300 000	4 000 000	10 000 000

10.4.2. Сравнение производительности алгоритмов

В этом разделе сравнивается производительность различных алгоритмов в нескольких средах, которые следуют в порядке возрастания их сложности.

Результаты, приведенные в табл. 10.4, и графики, показанные в этом разделе, дают представление о сравнительной производительности разных алгоритмов. Однако в зависимости от гиперпараметров результаты могут значительно варьироваться.

Возможна более тонкая настройка гиперпараметров, которая может привести к изменению относительного упорядочения алгоритмов, если они имеют схожую производительность. Но если разрыв в производительности большой, маловероятно, что дальнейшая настройка изменит упорядочение.

Имейте в виду, что нет алгоритма, который был бы лучшим во всех средах. PPO в целом показывает наилучшую производительность, но в некоторых средах лучшими могут оказаться другие алгоритмы.

Таблица 10.4. Окончательные значения средних вознаграждений mean_returns_ma. Это скользящее среднее по 100 контрольным точкам для полных вознаграждений, усредненных по результатам четырех сессий в испытании. Результаты получены с гиперпараметрами, приведенными в подразделе 10.4.1

Среда/алгоритм	DQN	Двойная DQN с PER	A2C (*n*-шаговый)	A2C (GAE)	PPO
LunarLander	192,4	**232,9**	68,2	25,2	214,2
BipedalWalker	—	—	187,0	15,7	**231,5**
Pong	16,3	20,5	18,6	19,2	**20,6**
Breakout	86,9	178,8	394,1	372,6	**445,4**
Qbert	2913,2	10 863,3	**13 590,4**	12 498,1	13 379,2

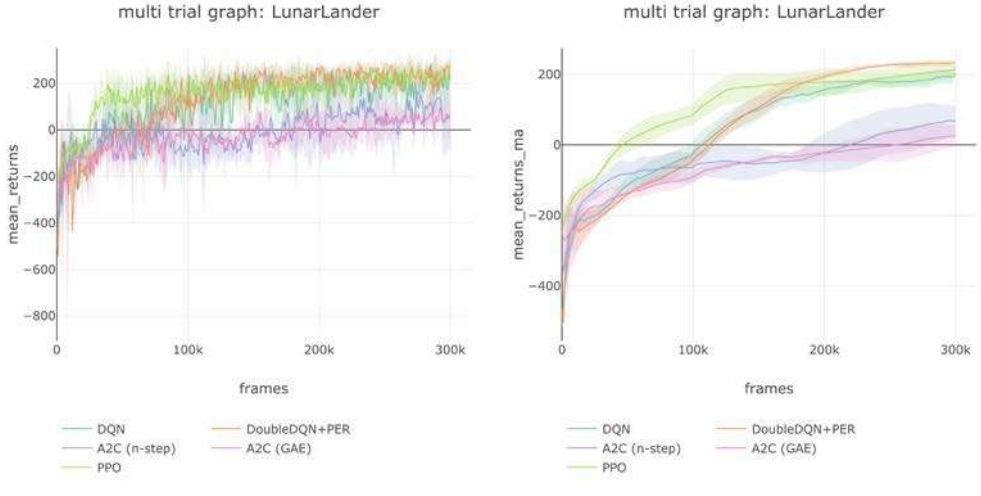


Рис. 10.5. Сравнение производительности алгоритмов в среде LunarLander

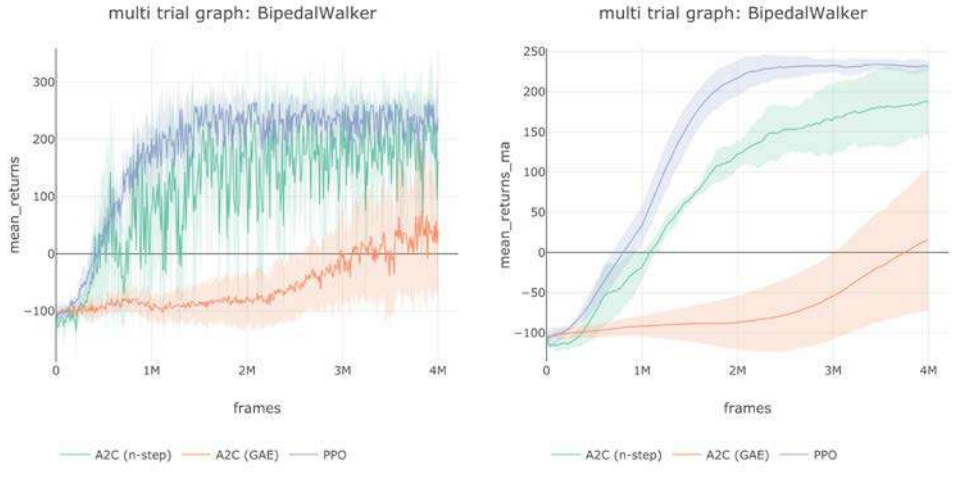


Рис. 10.6. Сравнение производительности алгоритмов в среде BipedalWalker

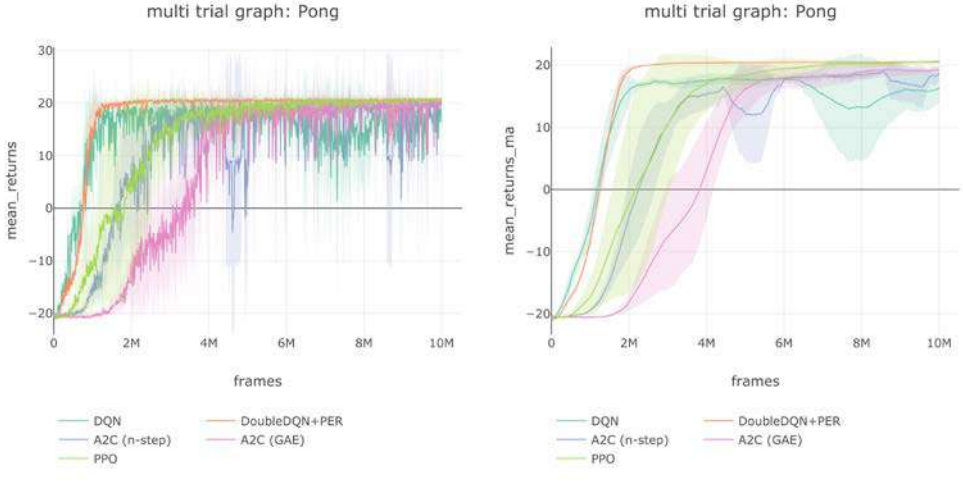


Рис. 10.7. Сравнение производительности алгоритмов в среде Pong Atari

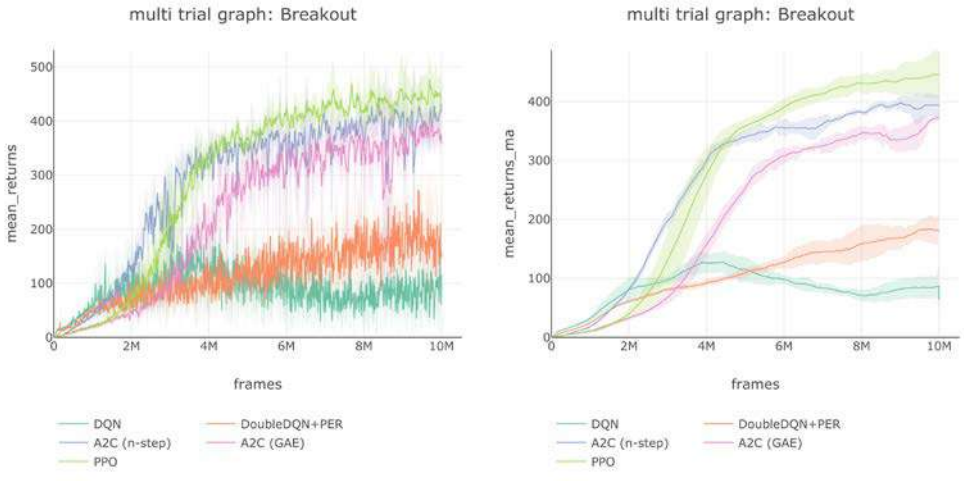


Рис. 10.8. Сравнение производительности алгоритмов в среде Breakout Atari

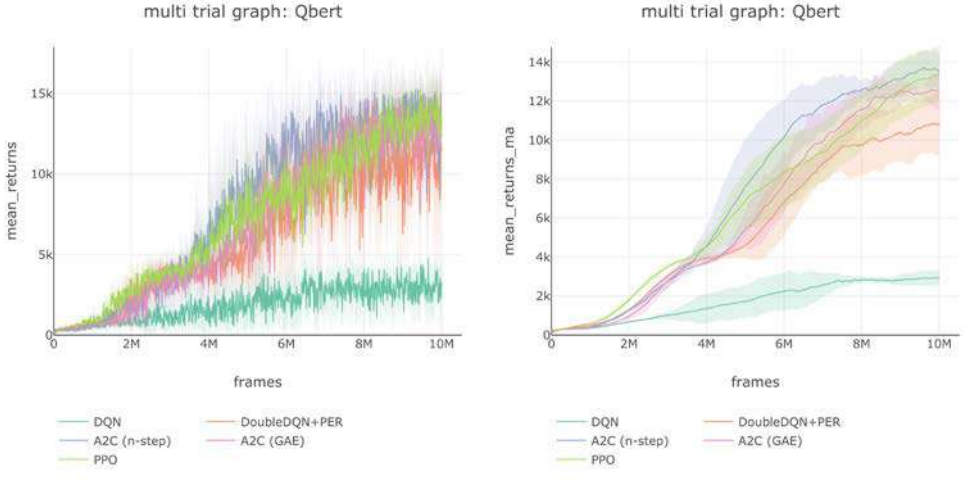


Рис. 10.9. Сравнение производительности алгоритмов в среде Qbert Atari

10.5. Резюме

Добиться того, чтобы алгоритмы глубокого RL работали, может оказаться непростой задачей. В этой главе мы обсудили некоторые эффективные практики проектирования кода, которые помогут вам заложить основы и сделать более управляемыми реализацию и отладку. Эти техники содержат модульные тесты, руководства по стилю оформления кода, автоматизированный анализ кода и рабочий процесс Git.

Затем мы перешли к практическим советам по отладке. В них вошли проверка на наличие «признаков жизни», проверка данных вручную и проверка компонентов агента, таких как препроцессор, память, нейронная сеть и сам алгоритм обучения. А также в числе общих советов — упрощение задачи и внедрение рабочих процессов научной деятельности. Мы вкратце обсудили особые приемы успешного обучения агентов играм Atari. И в конце перечислили наборы оптимальных гиперпараметров для основных алгоритмов и сред, обсуждаемых в этой книге. Надеемся, что это будет полезным руководством к действию для тех, кто столкнулся с этими задачами впервые.

Идеи, рассмотренные в этой главе, не являются исчерпывающими — сценариев и ошибок куда больше. Тем не менее данные советы — хорошая отправная точка для отладки. Для изучения других методик требуется больше практического опыта. При всем многообразии потенциальных ошибок нередко приходится тратить недели или даже месяцы на попытки заставить реализацию работать. Заниматься глубоким обучением с подкреплением непросто, но награда за то, что довел до ума что-нибудь сложное, очень мотивирует. И для того, чтобы не бросить дело на полпути, важней всего верить в успех и проявлять настойчивость.

11 SLM Lab

На протяжении всей книги мы пользовались SLM Lab для запуска испытаний и экспериментов. Эта глава задумывалась как справочник по ее основным функциям и командам.

Она начинается с обзора алгоритмов, реализованных в SLM Lab. Далее идет более подробное обсуждение файла спецификации, включая синтаксис настройки поиска по гиперпараметрам. Затем дано введение во фреймворк экспериментирования, который включает не только сессию `Session`, испытание `Trial` и эксперимент `Experiment`, но и основные команды библиотеки. Глава заканчивается разбором автоматически генерируемых при использовании SLM Lab графиков и данных.

Здесь предполагается, что SLM Lab была установлена в соответствии с инструкциями, приведенными во введении. Исходный код доступен на GitHub по ссылке https://github.com/kengz/SLM-Lab. Поскольку в библиотеку часто добавляются новые алгоритмы и функции, для этой книги выделена специальная ветвь `book`.

11.1. Алгоритмы, реализованные в SLM Lab

В SLM Lab есть реализации алгоритмов, которые мы обсуждали в этой книге:

- REINFORCE [148];
- SARSA [118];
- DQN [88];
- двойная DQN [141], приоритизированная память прецедентов [121];
- актор-критик с преимуществом (A2C);
- PPO [124];
- асинхронный актор-критик (A3C) [87].

Добавление функций и алгоритмов происходит часто, и SLM Lab активно разрабатывается. Вот несколько примеров недавно введенных расширений основных алгоритмов.

- Комбинированная память прецедентов (Combined Experience Replay, CER) [153]. Это простая модификация памяти прецедентов, в которой наиболее свежие прецеденты всегда добавляются в конец набора данных для обучения. Авторы CER показали, что это снижает чувствительность агента к размеру памяти прецедентов, уменьшая время настройки этого параметра.
- Дуэльная DQN [144]. В этом алгоритме используется преобразование типовой структуры нейронной сети, применяемой для аппроксимации Q-функции в DQN. В DQN сеть обычно выдает непосредственно Q-значения. Дуэльная DQN раскладывает эту оценку на две части: оценку функции полезности состояний V^{π} и функцию преимущества A^{π}. Эти оценки объединяются внутри модуля сети для получения итоговой оценки Q-значений, с тем чтобы форма окончательных выходных значений сети была такой же, как в DQN. Дальнейшее обучение происходит точно так же, как в самом алгоритме DQN или его вариантах. Дуэльная DQN основана на том, что в некоторых состояниях выбор действий не так уж важен и незначительно влияет на выходные данные, тогда как в других это критически важно. Следовательно, целесообразно отделить оценку полезности состояний от оценки преимущества. На момент публикации алгоритма дуэльной DQN им были достигнуты наилучшие результаты в играх Atari.
- Мягкий актор-критик (Soft Actor-Critic, SAC) [47]. Это алгоритм актора-критика, специально спроектированный как устойчивый и эффективный с точки зрения качества выборки. Его эффективность выборки выше, чем у A2C и PPO, поскольку это алгоритм обучения по отложенному опыту, так что в нем возможно повторное использование данных, хранящихся в памяти прецедентов. Еще в нем применяется подход по максимизации энтропии, прибавляющий энтропийный член к целевой функции. Посредством максимизации как энтропии, так и отдачи агент учится достигать высоких вознаграждений, действуя настолько случайно, насколько возможно. Это делает SAC менее уязвимым, и, как следствие, обучение становится более устойчивым.

В SLM Lab реализация всех алгоритмов разделена на три компонента, которым соответствуют три класса.

- `Algorithm`. Обрабатывает взаимодействия со средой, реализует стратегию выбора действий, вычисляет специфичную для алгоритма функцию потерь и запускает этап обучения. Помимо этого, класс `Algorithm` управляет другими компонентами и взаимодействиями между ними.

- `Net`. Хранит нейронные сети, которые выступают в качестве аппроксимаций функций для алгоритма.
- `Memory`. Предоставляет необходимое для обучения хранилище данных и осуществляет возврат к исходным значениям.

Организованные таким образом реализации могут пользоваться преимуществами наследования классов. Более того, в каждом компоненте имеется стандартный API, через который происходят взаимодействие и интеграция разных элементов. Наследование и стандартный API упрощают реализацию и тестирование новых компонентов. Для наглядности рассмотрим в качестве примера приоритизированную память прецедентов (Prioritized Experience Replay, PER).

Во введенном в главе 5 PER изменен механизм выборки из памяти прецедентов. Если коротко, в стандартном алгоритме DQN прецеденты выбираются из памяти случайно равномерно, тогда как в PER выборка прецедентов происходит в соответствии с распределением вероятностей, полученным из приоритетов всех прецедентов. Прецедентам необходимо присвоить приоритет, который обычно основан на абсолютной TD-ошибке. Значит, приоритеты актуальных прецедентов нужно обновлять на каждом этапе обучения агента. За исключением этого память прецедентов и процедура обучения такие же, как в алгоритме DQN. В SLM Lab это реализовано посредством класса памяти `PrioritizedReplay`, который наследует от `Replay` и которому поэтому требуется лишь около 100 строк нового кода. Исходный код находится в SLM Lab в `slm_lab/agent/memory/prioritized.py`.

При такой структуре по мере готовности компонент может быть незамедлительно применен ко всем подходящим алгоритмам. Например, двойная DQN может автоматически использовать PER, поскольку она наследует от DQN. Аналогично дуэльная DQN может им пользоваться, так как она реализована независимо преобразованием класса `Net`, который взаимодействует с памятью PER через стандартный API.

SLM Lab спроектирована для максимального повторного использования компонентов — это приносит дополнительную выгоду от сокращения кода и увеличения покрытия тестами. Основная идея в том, что мы можем доверять хорошо устоявшимся компонентам и сосредоточить внимание лишь на тех, которые находятся в процессе исследования и разработки. К тому же так проще изолировать влияние разных элементов: компоненты проектируются так, чтобы их можно было легко включать и отключать или заменять на другие — все с помощью файла `spec`. Это помогает при отладке, а также оценке новых идей, поскольку так проще установить базовые значения.

11.2. Файл spec

В этом разделе мы рассмотрим, как сформировать файл `spec` в SLM Lab.

В SLM Lab все настраиваемые гиперпараметры алгоритма указываются в файле `spec`. Это проектное решение призвано повысить воспроизводимость экспериментов в глубоком RL. Для реализованного кода, как правило, осуществляется контроль версий, и его можно отслеживать с помощью SHA с Git. Но для запуска эксперимента требуется также указать гиперпараметры, которые не отслеживаются как часть кода. Эти недостающие или скрытые гиперпараметры — один из источников проблемы воспроизведения экспериментов в глубоком RL, так как не существует стандартной практики их отслеживания при каждом запуске эксперимента. Для решения этой проблемы в SLM Lab все гиперпараметры представлены в одном файле `spec`. Они автоматически сохраняются вместе с SHA с Git и случайным начальным значением, которое используется как часть выходных значений, порождаемых для каждого запуска.

В листинге 11.1 показан пример отрывка файла `spec`, сохраненного после запуска обучения. SHA с Git (строка 22) позволяет восстановить версию кода, применявшуюся для запуска `spec`, а случайное начальное значение (строка 23) упрощает повторное моделирование стохастического процесса в агенте и среде.

Листинг 11.1. Пример файла spec, сохраненного во время сессии вместе с SHA с Git и случайным начальным значением

```
 1 {
 2     "agent": [
 3         {
 4             "name": "A2C",
 5                 ...
 6         }
 7     ],
 8     ...
 9     "meta": {
10         "distributed": false,
11         "log_frequency": 10000,
12         "eval_frequency": 10000,
13         "max_session": 4,
14         "max_trial": 1,
15         "experiment": 0,
16         "trial": 0,
17         "session": 0,
18         "cuda_offset": 0,
19         "experiment_ts": "2019_07_08_073946",
```

```
20         "prepath": "data/a2c_nstep_pong_2019_07_08_073946/
           ➥ a2c_nstep_pong_t0_s0",
21         "ckpt": null,
22         "git_sha": "8687422539022c56ae41600296747180ee54c912",
23         "random_seed": 1562571588,
24         "eval_model_prepath": null,
25         "graph_prepath":
           ➥ "data/a2c_nstep_pong_2019_07_08_073946/graph/a2c_nstep_pong_t0_s0",
26         "info_prepath":
           ➥ "data/a2c_nstep_pong_2019_07_08_073946/info/a2c_nstep_pong_t0_s0",
27         "log_prepath":
           ➥ "data/a2c_nstep_pong_2019_07_08_073946/log/a2c_nstep_pong_t0_s0",
28         "model_prepath":
           ➥ "data/a2c_nstep_pong_2019_07_08_073946/model/a2c_nstep_pong_t0_s0"
29     },
30     "name": "a2c_nstep_pong"
31 }
```

При таком способе проектирования любой эксперимент в SLM Lab можно запустить повторно с помощью файла `spec`, который содержит все гиперпараметры и полученный с Git SHA для использовавшейся версии кода. В файле `spec` есть вся информация, необходимая для того, чтобы в *полном объеме воспроизвести эксперимент в RL*. Это делается путем простой проверки кода на наличие SHA с Git и выполнения сохраненной спецификации. На протяжении всей книги мы пользовались файлами `spec` для проведения испытаний и экспериментов, теперь рассмотрим этот файл в деталях.

В SLM Lab файл `spec` используется для создания агента и среды. Его формат стандартизирован в соответствии с применяемой в библиотеке структурой модульных компонентов. Составляющие файла `spec`.

1. `agent`. Формат списка позволяет запускать нескольких агентов. Однако для наших целей мы можем предположить, что агент только один. Каждый элемент списка — это спецификация агента, содержащая спецификации для его компонентов:
   - `algorithm` — основные параметры, специфичные для конкретного алгоритма, такие как вид стратегии, параметры алгоритма, скорости уменьшения коэффициентов и порядок обучения;
   - `memory` — определяет, какую память применять в соответствии с алгоритмом, а также все специфические для памяти гиперпараметры, такие как размеры пакета и самой памяти;
   - `net` — тип нейронной сети, архитектура ее скрытых слоев, функции активации, усечение градиентов, функция потерь, оптимизатор, скорости уменьшения, метод обновления и применение CUDA.
2. `env`. Используется также формат списка для поддержки нескольких сред, но на данный момент мы можем считать, что среда только одна. Он указывает, какую

среду использовать, задает необязательное количество шагов на эпизод и полное число шагов (кадров) в `Session`. Также определяет методы предварительной обработки состояний и вознаграждений и количество сред в векторизированной среде (см. главу 8).

3. `body`. Задает способ привязки агента к средам. Можно проигнорировать для нашего случая с единственным агентом и одной средой — просто воспользуйтесь значениями по умолчанию.
4. `meta`. Высокоуровневая настройка запусков библиотеки. Задает количество испытаний `Trial` и сессий `Session` при запуске, частоту оценки и журналирования и переключение на асинхронное обучение (см. главу 8).
5. `search`. Гиперпараметры, по которым происходит поиск, и методы их выбора. Поиск можно выполнять по любым переменным в файле `spec`, включая переменные среды, хотя обычно применяется поднабор переменных агента.

11.2.1. Синтаксис поиска в spec

Синтаксис для указания гиперпараметра, по которому производится поиск, следующий: `"{key}__{space_type}": {v}`. Здесь `{key}` — это название гиперпараметра, определенного в остальной части файла. `{v}` обычно задает диапазон поиска, как при случайном поиске. Тем не менее для обеспечения более гибких стратегий поиска `v` может быть представлено также списком выбираемых значений или средним значением и нормальным отклонением для распределения вероятностей. В SLM Lab можно выбирать из четырех распределений, по два для дискретных и непрерывных переменных, и то, как интерпретируется `v`, зависит от `space_type`, в котором определен метод выборки значений.

- `space_type` для дискретных переменных:
  - `choice: str/int/float. v` — список выбираемых значений;
  - `randint: int. v = [low, high)`.
- `space_type` для непрерывных переменных:
  - `uniform: float. v = [low, high)`;
  - `normal: float. v = [mean, stdev)`.

Дополнительно в качестве `space_type` доступно `grid_search` для итерации по всему списку выбираемых значений вместо их случайной выборки, приведенной ранее. Мы встречали этот метод во всех главах об алгоритмах в книге.

Рассмотрим спецификацию поиска, которая может быть использована для запуска эксперимента в SLM Lab. В данном эксперименте мы обучаем агента DQN с прогнозной сетью играть в CartPole. Поиск выполняется по трем гиперпараметрам: коэффициенту дисконтирования γ, количеству пакетов, выбираемых из памяти

на каждом шаге обучения, и частоте обновления замещением для прогнозной сети. `spec` приведен полностью в листинге 11.2. Каждая строка снабжена комментарием с кратким описанием гиперпараметра.

Листинг 11.2. Спецификация поиска для DQN в CartPole

```
# slm_lab/spec/experimental/dqn/dqn_cartpole_search.json

{
    "dqn_cartpole": {
        "agent": [{
            "name": "DQN",
            "algorithm": {
                "name": "DQN", # название класса запускаемого алгоритма
                "action_pdtype": "Argmax", # распределение
                ➥ вероятностей действий стратегии
                "action_policy": "epsilon_greedy", # метод выборки действий
                "explore_var_spec": {
                    "name": "linear_decay", # как уменьшается
                    ➥ переменная исследования
                    "start_val": 1.0, # начальное значение
                    ➥ переменной исследования
                    "end_val": 0.1, # минимальное значение
                    ➥ переменной исследования
                    "start_step": 0, # временной шаг начала
                    ➥ уменьшения значения
                    "end_step": 1000, # временной шаг завершения
                    ➥ уменьшения значения
                },
                "gamma": 0.99, # коэффициент дисконтирования
                "training_batch_iter": 8, # обновлений параметров на пакет
                "training_iter": 4, # пакетов на шаг обучения
                "training_frequency": 4, # как часто обучать агента
                "training_start_step": 32 # временной шаг начала обучения
            },
            "memory": {
                "name": "Replay", # название класса памяти
                "batch_size": 32, # размер выбираемого из памяти пакета
                "max_size": 10000, # максимальное количество
                ➥ хранимых прецедентов
                "use_cer": false # использовать ли комбинированную память
                ➥ прецедентов примеров
            },
            "net": {
                "type": "MLPNet", # название класса сети
                "hid_layers": [64], # размер скрытых слоев
                "hid_layers_activation": "selu", # функция активации
                ➥ скрытых слоев
                "clip_grad_val": 0.5, # максимальная норма градиента
                "loss_spec": { # спецификация функции потерь
                    "name": "MSELoss"
                },
```

```
38                "optim_spec": { # спецификация оптимизатора
39                    "name": "Adam",
40                    "lr": 0.01
41                },
42                "lr_scheduler_spec": null, # спецификация
                   ➥ изменения скорости обучения
43                "update_type": "polyak", # метод обновления
                   ➥ прогнозной сети
44                "update_frequency": 32, # как часто обновлять
                   ➥ прогнозную сеть
45                "polyak_coef": 0.1, # вес параметров сети,
                   ➥ применяемых при обновлении
46                "gpu": false # использовать ли графический
                   ➥ процессор при обучении
47            }
48        }],
49        "env": [{
50            "name": "CartPole-v0", # название среды
51            "max_t": null, # максимальное количество шагов на эпизод
52            "max_frame": 50000 # максимальное количество шагов на сессию
53        }],
54        "body": {
55            "product": "outer",
56            "num": 1
57        },
58        "meta": {
59            "distributed": false, # использовать ли асинхронную параллелизацию
60            "eval_frequency": 1000, # как часто оценивать агента
61            "max_session": 4, # количество запускаемых сессий
62            "max_trial": 32 # количество запускаемых испытаний
63       },
64        "search": {
65            "agent": [{
66                "algorithm": {
67                    "gamma__uniform": [0.50, 1.0],
68                    "training_iter__randint": [1, 10]
69                },
70                "net": {
71                    "optim_spec": {
72                        "lr__choice": [0.0001, 0.001, 0.01, 0.1]
73                    }
74                }
75            }]
76        }
77    }
78 }
```

В листинге 11.2 в строках 64–76 определены три переменные, по которым происходит поиск. Это одна непрерывная переменная `gamma`, значения которой выбираются с помощью случайного равномерного распределения. Поиск по дискретной переменной `training_iter` производится с равномерной выборкой целых значений,

принадлежащих промежутку [0, 10). Скорость обучения `lr` выбирается случайным образом из списка [0,0001; 0,001; 0,01; 0,1]. Это иллюстрация некоторых методов выбора, которые могут применяться при поиске по гиперпараметрам в SLM Lab.

При запуске эксперимента большое значение имеет еще одна переменная, `max_trial` (строка 62). Она определяет, какое количество наборов гиперпараметров будет порождаться и использоваться при запуске испытаний. В данном примере при `max_trial = 32` имеются 32 случайные комбинации значений `gamma`, `training_iter` и `lr`, которые станут замещать свои значения по умолчанию в полном файле `spec`. В результате для запуска испытаний будут задействованы 32 файла `spec` с разными наборами гиперпараметров. В каждом испытании будут запущены по четыре сессии `Session` (строка 61).

Теперь, когда мы знаем, как писать файл `spec`, рассмотрим запуск эксперимента с помощью SLM Lab.

11.3. Запуск SLM Lab

Проведение эксперимента в глубоком RL обычно включает в себя тестирование разных наборов гиперпараметров. Результаты в глубоком RL отличаются высокой дисперсией даже для одного и того же набора гиперпараметров. В связи с этим хорошо зарекомендовала себя практика выполнения множества запусков с применением разных случайных начальных значений и усреднения результатов.

Эта практика внедрена во фреймворк экспериментирования в SLM Lab, который имеет иерархическую структуру из трех компонентов.

1. **Сессия** (`Session`). На самом нижнем уровне фреймворка экспериментирования в SLM Lab находится сессия, в которой выполняется цикл управления RL. В нем происходят инициализация агента и среды с помощью отдельного набора гиперпараметров и заданного случайного начального значения, а также обучение агента. По завершении сессии в папку с данными для анализа сохраняются обученный агент, файл `spec`, данные и графики.
2. **Испытание** (`Trial`). Запускаются несколько сессий с применением одного и того же набора гиперпараметров и разных случайных начальных значений. Затем результаты сессий усредняются и строятся графики для испытаний.
3. **Эксперимент** (`Experiment`). На самом высоком уровне фреймворка экспериментирования в SLM Lab эксперимент порождает различные наборы гиперпараметров и запускает испытание для каждого из них. Его можно понимать как исследование на тему «Какие значения γ и скорости обучения дают самое быстрое и наиболее устойчивое решение, если другие переменные остаются постоянными?». По завершении эксперимента испытания сравниваются по графикам для множества испытаний.

11.3.1. Команды SLM Lab

Рассмотрим основные команды в SLM Lab, составленные по шаблону `python run_lab.py {spec_file} {spec_name} {lab_mode}`. Существуют четыре основные команды для разных случаев применения.

1. `python run_lab.py slm_lab/spec/benchmark/a2c/a2c_nstep_pong.json a2c_nstep_pong dev`. Режим разработки отличается от описанного ранее режима обучения тем, что он ограничен одним сеансом, в нем визуализируется среда, проверяются обновления параметров сети и ведется подробный журнал отладки.
2. `python run_lab.py slm_lab/spec/benchmark/a2c/a2c_nstep_pong.json a2c_nstep_pong train`. Обучение агента с помощью заданной спецификации путем порождения испытания.
3. `python run_lab.py slm_lab/spec/benchmark/a2c/a2c_nstep_pong.json a2c_nstep_pong search`. Запуск эксперимента с поиском по гиперпараметрам с порождением одного эксперимента.
4. `python run_lab.py data/a2c_nstep_pong_2018_06_16_214527/a2c_nstep_pong_spec.json a2c_nstep_pong enjoy@a2c_nstep_pong_t1_s0`. Загрузка из папки `data/` в SLM Lab агента, сохраненного после завершения испытания или эксперимента. Режим `enjoy@a2c_nstep_pong_t1_s0` задает испытания и сессии файла модели.

11.4. Анализ результатов эксперимента

После окончания эксперимента данные автоматически выводятся в папку, размещенную в `SLM-Lab/data/`. В этом разделе сделан упор на использование данных для глубокого анализа эксперимента. Экспериментальные данные отражают иерархию «сессия — испытание — эксперимент», обсуждавшуюся в разделе 11.3. Мы дадим обзор данных, генерируемых SLM Lab, сопровождая их примерами.

11.4.1. Обзор экспериментальных данных

Получаемые в ходе экспериментов данные автоматически сохраняются в папке `data/{experiment_id}`. `{experiment_id}` — это конкатенация названия файла `spec` и временной метки, сгенерированной при запуске эксперимента.

Для каждой сессии создаются график кривой обучения с его скользящим средним, сохраненные параметры модели агента, файл CSV, содержащий данные сессии, и показатели на уровне сессии. Данные сессии состоят из вознаграждений, функции потерь, скорости обучения, значения переменной исследования и т. д. На рис. 11.1 приведен пример графиков для сессии эксперимента с актором-критиком из подраздела 6.7.1.

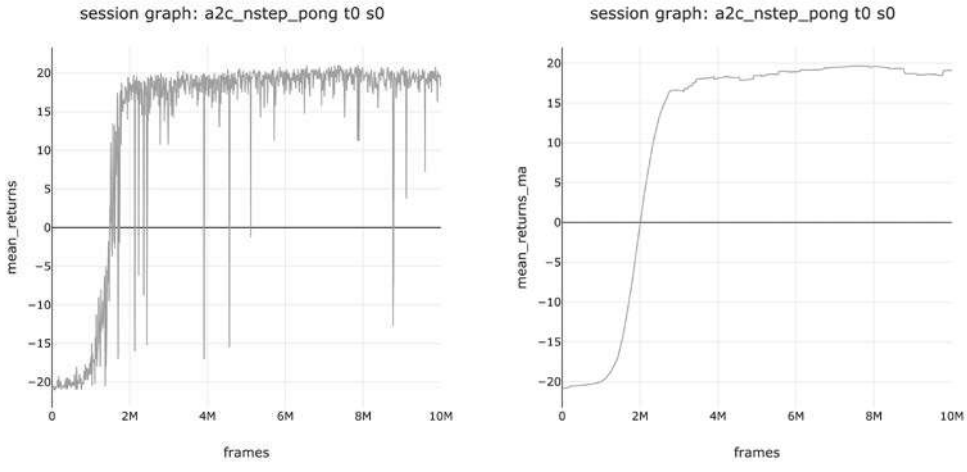


Рис. 11.1. Примеры графиков для сессии. По вертикальной оси отложены полные вознаграждения в контрольных точках, усредненные по данным из восьми эпизодов. По горизонтальной оси — все кадры обучения. Справа показан график со скользящим средним по оценкам в 100 контрольных точках

При каждом испытании создается график усредненных по сессиям кривых обучения с доверительным интервалом в пределах ±1 стандартное отклонение. Также включена версия этого графика со скользящим средним. Кроме того, выдаются показатели на уровне испытания. На рис. 11.2 продемонстрирован пример графиков для испытания.

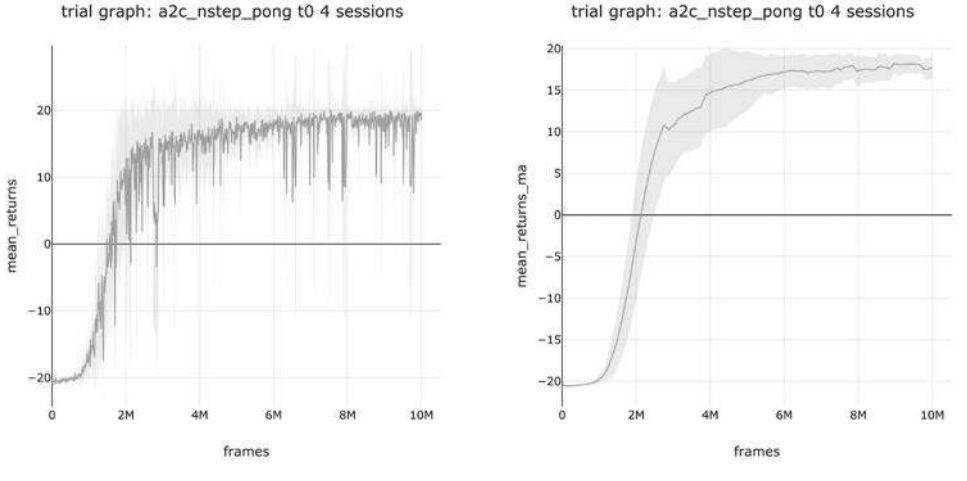


Рис. 11.2. Примеры графиков для испытаний, усредненных по данным сессий. По вертикальной оси отложены полные вознаграждения в контрольных точках, усредненные по данным из восьми эпизодов. По горизонтальной оси — все кадры обучения. Справа показан график со скользящим средним по оценкам в 100 контрольных точках

Эксперимент порождает график, на котором сравниваются результаты всех испытаний, а также версию со скользящим средним. Дополнительно создается файл CSV под названием `experiment_df`. Он содержит итоговые значения переменных и результаты эксперимента, отсортированные в порядке ухудшения производительности при испытаниях. Этот файл предназначен для того, чтобы ясно показать диапазоны значений гиперпараметров и их комбинации от самых успешных до самых неудачных. На рис. 11.3 приведены примеры графиков для множества испытаний.

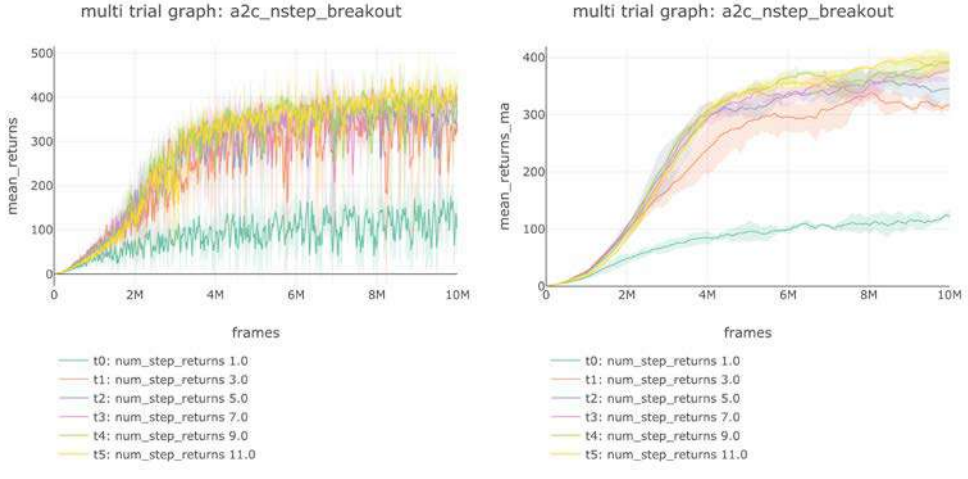


Рис. 11.3. Примеры графиков для ряда испытаний

11.5. Резюме

В этой главе более детально рассматривалась прилагаемая к книге библиотека SLM Lab. Мы обсудили реализованные в ней алгоритмы и применяемые для их настройки файлы `spec`. Познакомились с основными командами библиотеки.

Затем мы рассмотрели фреймворк экспериментирования, состоящий из классов `Session`, `Trial` и `Experiment`, каждый из которых создает графики и данные, полезные для анализа производительности алгоритма.

SLM Lab находится в процессе активной разработки, и в ее последнюю версию часто добавляются новые алгоритмы и функции. Чтобы воспользоваться ими, перейдите в основную ветвь `master` репозитория GitHub по ссылке https://github.com/kengz/SLM-Lab.

12 Архитектура сетей

Нейронные сети являются компонентами всех алгоритмов, обсуждаемых в этой книге. Однако до этого момента мы еще ни разу не рассматривали подробно ни устройство этих сетей, ни те полезные функциональные возможности, которые они предоставляют при объединении с RL. Цель этой главы — более близкое знакомство с проектированием и обучением нейронных сетей в контексте глубокого RL.

Глава начинается с краткого введения в разные семейства нейронных сетей и типы данных, на обработке которых они специализируются. Затем мы поговорим о том, как выбрать подходящую сеть, исходя из двух характеристик среды: насколько она доступна для *наблюдения* и какова природа пространства состояний. Чтобы охарактеризовать доступность среды для наблюдения, обсудим разницу между марковским процессом принятия решений (МППР) и частично наблюдаемым марковским процессом принятия решений (частично наблюдаемый МППР). Кроме того, введем три разных вида частично наблюдаемых МППР.

В остальной части главы рассматривается `Net` API из SLM Lab, в котором инкапсулирована функциональность, общая для обучения нейронных сетей в контексте глубокого RL. Мы сделаем обзор некоторых желательных свойств `Net` API и на примерах из SLM Lab покажем, как эти свойства можно реализовать.

12.1. Виды нейронных сетей

Нейронные сети могут быть объединены в семейства в соответствии с их особенностями, решаемыми ими задачами и типом входных данных, с обработкой которых они хорошо справляются. Есть три основные категории: многослойные перцептроны (multilayer perceptron, MLP), сверточные нейронные сети (convolutional neural networks, CNN) и рекуррентные нейронные сети (recurrent neural networks, RNN). Эти виды нейронных сетей можно комбинировать и получать гибриды, например CNN-RNN.

Каждое из семейств характеризуется типами слоев, составляющих сеть, и способом организации потока вычислений. Эти семейства можно также понимать в том смысле, что они включают в себя различные первоначальные знания о своих входных значениях и используют эти знания для того, чтобы лучше обучаться на данных с определенными характеристиками. Далее в этом разделе идет очень краткий обзор характеристик основных семейств нейронных сетей. Читатели, знакомые с нейронными сетями, могут его полностью пропустить или взглянуть лишь на рис. 12.4. Желающим получить более подробные сведения рекомендуем две превосходные книги: *Neural Networks and Deep Learning* Майкла Нильсена [92] и *Deep Learning* Яна Гудфеллоу, Иошуа Бенджио и Аарона Курвилля [45]. Обе были доступны онлайн на момент написания этих строк.

12.1.1. Многослойные перцептроны

Многослойные перцептроны (MLP) — самый простой и наиболее общий вид нейронных сетей. Они состоят только из *полносвязных слоев*. В полносвязном слое все выходные значения из предыдущего слоя связаны со всеми узлами текущего слоя посредством определенных весов. К выходным значениям всех плотных слоев обычно применяются нелинейные функции активации. Они придают нейронным сетям выразительность, делая возможной аппроксимацию весьма сложных нелинейных функций.

Многослойные перцептроны являются сетями общего назначения, и их входные значения представлены одним вектором с n элементами. MLP делают очень мало предположений о природе своих входных данных. Например, они не кодируют никакую информацию о связях между разными измерениями входных данных. В этом одновременно заключаются как их сила, так и ограниченность. Это позволяет многослойным прецептронам рассматривать входные значения с разных точек зрения. Они могут реагировать на глобальные структуры и шаблоны, обнаруживая характеристики, являющиеся комбинациями всех входных элементов.

Однако многослойные перцептроны могут также игнорировать структуры в данных, на которых они обучаются. Рассмотрим два изображения, приведенные на рис. 12.1, *а* и *б*. Первое — это фотография горного пейзажа, а второе выглядит как случайный шум. Значения пикселов на рис. 12.1, *а*, демонстрируют очень сильную пространственную корреляцию в двух измерениях. В большинстве случаев надежный показатель значения отдельного пиксела — это значения соседних пикселов. Значение пиксела на рис. 12.1, *б*, напротив, не имеет сильной корреляции со значениями соседних с ним пикселов. Любое реалистичное изображение в гораздо большей степени подобно рис. 12.1, *а*, чем рис. 12.1, *б*, показывая высокую степень локальной корреляции в значениях пикселов.

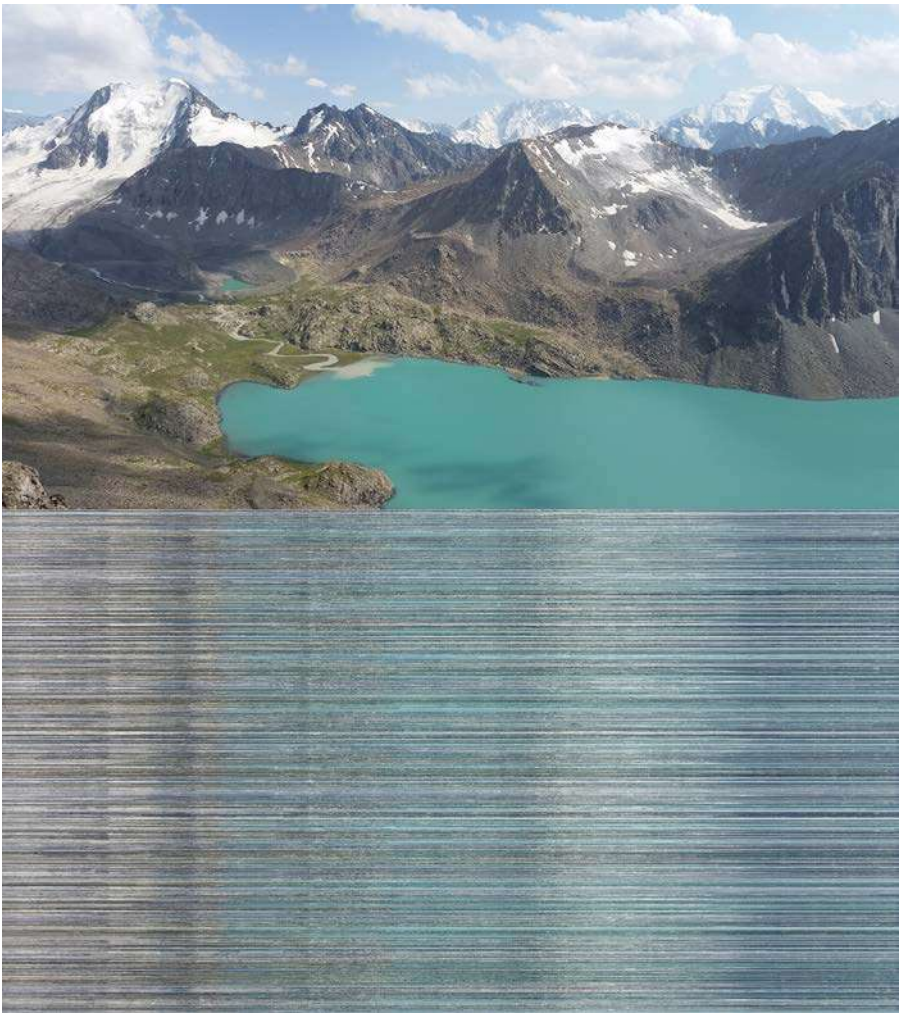


Рис. 12.1. Сравнение фотографии гор со случайным изображением

Двумерная структура изображения не является легкодоступной для MLP, так как перед подачей на вход нейронной сети каждое изображение будет преобразовано в длинный одномерный список чисел. Перед обучением может произойти перестановка значений пикселов в случайном порядке, причем будет получено эквивалентное одномерное представление изображения. Для сравнения представьте, что произойдет, если выполнить случайную перестановку пикселов двумерного изображения. Таким образом и был получен рис. 12.1, *б*, — случайной перестановкой пикселов рис. 12.1, *а*. В двумерном пространстве разница между этими фрагментами, очевидна, и по одному лишь рис. 12.1, *б*, невозможно определить, что представляют собой пикселы. Смысл этих пикселов проще выявить по их значениям в совокупности с расположением в двумерном пространстве.

Конвертация двумерного изображения в одномерный вектор — пример потери метаинформации о состоянии, которая более детально обсуждается в разделе 14.4. Изображения двумерны по своей сути, в них каждый пиксел связан с другими пикселами в двумерном пространстве. При преобразовании в одномерное представление задача усложняется. Сети нужно реконструировать двумерные взаимосвязи между пикселами, так как эта информация неочевидна при таком способе представления входных данных.

Другая особенность многослойных перцептронов — тенденция к быстрому росту количества параметров. Например, рассмотрим MLP, принимающий на входе значения из $\mathbb{R}^{784}$, с двумя скрытыми слоями с 1024 и 512 узлами и выходным слоем из десяти узлов. Функция, вычисляемая MLP, приведена в уравнении (12.1), функция активации — сигмоида $\sigma(x) = \frac{1}{1+e^{-x}}$:

$$f_{\text{MLP}}(x) = \sigma(W_2\sigma(W_1x + b_1) + b_2), \tag{12.1}$$

где W_1 и W_2 — матрицы весов, а b_1 и b_2 — векторы смещений. В W_1 содержится $1024 \cdot 784 = 802\,816$ элементов, в W_2 — $512 \cdot 1024 = 524\,288$ элементов, а в векторах смещений b_1 и b_2 — 1024 и 512 элементов соответственно. Тогда количество параметров сети составляет $802\,816 + 524\,288 + 1024 + 512 = 1\,328\,640$. По сегодняшним меркам это маленькая сеть, но и у нее есть много параметров, которые нужно настроить. Это может быть проблематично по той причине, что количество примеров, необходимых для настройки оптимальных значений для каждого из параметров, увеличивается с ростом общего числа параметров в сети. Большое количество поддающихся настройке параметров вкупе с неэффективностью выборки текущих алгоритмов глубокого RL может привести к тому, что обучение агента затянется надолго. Из-за этого многослойные перцептроны лучше всего подходят для сред с двумя свойствами: их пространство состояний имеет малую размерность и для формирования признаков требуются все элементы состояния.

Наконец, у многослойных перцептронов нет памяти, то есть они знают только о текущих входных значениях и ничего не помнят о предыдущих. В MLP на вход подаются неупорядоченные значения, предварительная обработка каждого из которых происходит независимо от других.

12.1.2. Сверточные нейронные сети

Сверточные нейронные сети (CNN) преуспели в обучении по изображениям. CNN спроектированы специально для исследования пространственной структуры данных в виде изображений, поскольку они содержат один или два сверточных слоя, каждый из которых состоит из нескольких ядер свертки.

Для получения выходных значений к подмножеству входных значений многократно применяются ядра свертки. Говорится: произошла *свертка по ядру*, что

называется операцией *свертки*. Например, можно применить свертку к двумерному изображению для получения на выходе двумерных значений. В этом случае однократное применение ядра будет соответствовать его применению к небольшому участку изображения, состоящему из нескольких пикселов: 3×3 или 5×5 пикселов. Результат этой операции — скалярное выходное значение.

Данное скалярное выходное значение может быть интерпретировано как сигнал о наличии или об отсутствии отдельно взятого признака в локальной области входных данных, применительно к которой это значение было получено. Следовательно, ядро можно рассматривать как локальный детектор признаков, так как оно выдает выходные значения на основе пространственно сопряженного подмножества входных признаков. Ядро (детектор признаков) применяется локально по всему изображению для получения *карты признаков*, в которой описаны все места, где на данном изображении встречается определенный признак.

Одиночный сверточный слой, как правило, состоит из некоторого количества ядер (обычно 8–256), каждое из которых может настраиваться определять отдельный признак. Например, одно ядро может настраиваться находить вертикальные прямолинейные ребра, второе — горизонтальные прямолинейные ребра, а третье — криволинейное ребро. При комбинировании слоев последующие слои могут настраиваться на все более сложные признаки с помощью выходных значений детекторов признаков предыдущего слоя. Например, на высокоуровневом слое сети одна функция может научиться определять ракетку в игре Pong из Atari, а другая — мячик.

Одно из преимуществ такой структуры слоев — способность ядра обнаруживать наличие признака вне зависимости от того, где он появляется на изображении. Это выгодно, так как положение полезных признаков на разных изображениях может варьироваться. Например, в игре Pong из Atari меняются как позиции ракеток игрока и его оппонента, так и мяча. Применив к изображению ядро — «детектор ракетки», получим двумерную карту расположения всех ракеток на рисунке.

С помощью сверток сеть «бесплатно» учится тому, что конкретная комбинация значений представляет один и тот же признак в разных частях изображения. С точки зрения количества параметров сверточные слои эффективнее, чем полносвязные. Одно ядро может быть применено для определения одного и того же признака в любом месте изображения — нам не нужно настраивать отдельное ядро обнаруживать этот признак на множестве других участков. В результате при применении сверточных и полносвязных слоев к входным данным с одинаковым числом элементов у первых будет значительно меньше параметров, чем у вторых. Это происходит вследствие многократного применения одного-единственного ядра к большому количеству подучастков входных данных.

Недостаток сверточных слоев — их локальность: они за раз обрабатывают лишь часть пространства входных значений и игнорируют глобальную структуру изо-

бражения. Однако глобальная структура часто имеет большое значение. Рассмотрим еще раз игру Pong из Atari. Для достижения хорошей производительности при принятии решения о действии нужно использовать взаимное местоположение мячика и ракеток. Местоположение можно задать только относительно всего пространства входных значений.

Ограниченность сверток часто нивелируется за счет увеличения рецептивного поля ядер на более высоких уровнях[1]. Это значит, что увеличивается эффективная область пространства входных значений, которую обрабатывает ядро, — такие ядра могут «видеть» большую часть пространства входных значений. Чем больше рецептивное поле ядра, тем более глобальной будет его перспектива. Альтернативный подход — добавление небольшого MLP поверх CNN, чтобы получить преимущества обоих семейств. Вернемся к примеру с Pong Atari. На выходе из CNN будут получены двумерные карты, содержащие местоположения мячика и ракеток. Эти карты передаются в качестве входных значений в MLP, который объединяет всю эту информацию и выдает действие.

Как и у многослойных перцептронов, у сверточных нейронных сетей нет памяти. Однако, в отличие от MLP, CNN идеально подходят для обучения по изображениям, так как они предполагают наличие пространственной структуры у своих входных данных.

Более того, применение ядер — эффективный способ уменьшения числа параметров в сети, когда количество элементов на входе велико, а в каждом типичном цифровом представлении изображения есть тысячи, а то и миллионы элементов. Фактически CNN значительно превосходят все остальные семейства нейронных сетей в обучении по изображениям. Поэтому главное практическое правило гласит: если предоставляемое средой состояние — это изображение, включите в сеть несколько сверточных слоев.

12.1.3. Рекуррентные нейронные сети

Рекуррентные нейронные сети (RNN) специализируются на обучении по последовательно передаваемым данным. Для них одна информационная единица состоит из последовательности элементов, каждый из которых является вектором. RNN, в отличие от MLP и CNN, исходит из предположения, что порядок получения элементов имеет значение. Предложения — один из примеров типов данных, которые RNN хорошо обрабатывают. Каждый элемент последовательности — это слово, а порядок слов влияет на смысл предложения в целом. Однако информационная единица может быть и последовательностью состояний, например упорядоченной последовательностью состояний, наблюдаемых агентом.

---

[1] Например, с помощью операций объединения, расширенных сверток, периодических сверток или больших ядер.

Отличительной чертой RNN является то, что они хранят состояния, то есть запоминают элементы, которые видели прежде. При обработке элемента последовательности x_i RNN помнит идущие перед ним элементы $x_0, x_1 ... x_{i-1}$. Это достигается с помощью специальных рекуррентных слоев со скрытым состоянием. Скрытое состояние — это настроенное представление элементов в последовательности, которые сеть видела до текущего момента, и оно обновляется при каждом получении сетью нового элемента. Память RNN соответствует длине последовательности, в начале новой последовательности скрытое состояние RNN обнуляется. Долгая краткосрочная память (Long Short-Term Memory, LSTM) [52] и рекуррентный блок с гейтами (Gated Recurrent Unit, GRU) [21] — наиболее типичные слои с такими характеристиками.

Применение механизма для запоминания прошлого целесообразно, когда информация, предоставляемая средой на шаге t, не охватывает в полной мере все, что было бы полезно знать в текущий момент. Рассмотрим среду-лабиринт, в которой агент для получения положительного вознаграждения может поднимать золотые монеты, появляющиеся в одних и тех же местах. Предположим также, что золотые монеты после того, как их поднял агент, могут снова появляться по истечении фиксированного промежутка времени. Даже если агент может видеть, где находятся все существующие на текущий момент монеты, он может лишь предполагать, когда отдельные монеты появятся снова, если он помнит, как давно их поднял. Несложно представить, что при ограниченном времени для максимизации счета нужно следить за тем, какие монеты и когда он поднимает. Для этого задания агенту потребуется память, а это основное преимущество RNN перед CNN или MLP.

К счастью, нет необходимости ограничиваться выбором исключительно сети одного типа. Наоборот, очень распространено создание гибридных сетей, состоящих из множеств подсетей, которые могут все принадлежать к разным семействам. Например, можно спроектировать сеть с модулем обработки состояний — MLP или CNN с представлением необработанного состояния[1] на выходе и с модулем временной обработки, которым является RNN. На каждом временном шаге необработанное состояние проходит через модуль обработки состояний и выходные данные передаются на вход RNN. Затем RNN использует эту информацию, чтобы выдать конечный результат на выходе общей сети. Сеть, подсетями которой являются и CNN, и RNN, называется CNN-RNN. Поскольку MLP очень часто применяются в качестве небольших дополнительных подсетей, их обычно не упоминают в этом названии.

Подведем итоги: RNN лучше всего подходят для задач, где данные представлены как упорядоченные последовательности. В контексте глубокого RL они обычно используются, когда агенту для принятия оптимальных решений необходимо помнить, что происходило на протяжении длительного времени.

---

[1] В таком представлении, как правило, меньше элементов, чем в первоначальном состоянии.

12.2. Рекомендации по выбору семейства сетей

После введения разных видов нейронных сетей может возникнуть вопрос: какие из них агент должен применять для конкретной среды? В этом разделе обсуждаются рекомендации по выбору семейства сетей на основе характеристик среды.

Все среды глубокого RL могут рассматриваться как порождающие последовательные данные. Мы видели, что RNN специализируются на обработке этого вида выходных данных. Так почему же не использовать RNN или CNN-RNN в глубоком RL всегда? Для ответа на этот вопрос необходимо обсудить различия между МППР и частично наблюдаемыми МППР.

12.2.1. Сравнение МППР и частично наблюдаемых МППР

Здесь мы вкратце повторим формальное определение МППР, которое было дано в главе 1. МППР — это математическая модель последовательного принятия решений. В основе МППР лежит функция переходов, которая моделирует переход состояния s_t в следующее состояние s_{t+1}. Функция переходов МППР приведена в уравнении (12.2):

$$s_{t+1} \sim P(st_{+1} \mid s_t, a_t). \tag{12.2}$$

Функция переходов обладает марковским свойством — переход в s_{t+1} полностью определяется текущим состоянием и действием (s_t, a_t). Ранее в эпизоде агент мог наблюдать много состояний $s_0, s_1 \ldots s_{t-1}$, но они не несут какой-либо дополнительной информации о том состоянии, в которое перейдет среда.

Понятие состояния встречается в двух случаях. Во-первых, есть состояние, которое порождается средой, наблюдается агентом и называется наблюдаемым состоянием s_t. Во-вторых, есть состояние, которое используется функцией переходов, — внутреннее состояние среды s_t^{int}.

Если среда является МППР, то она описывается как *полностью наблюдаемая*, и $s_t = s_t^{int}$. Например, обе задачи, CartPole и LunarLander, — полностью наблюдаемые среды. CartPole предоставляет на каждом временном шаге четыре единицы информации: положение тележки относительно линейной оси, скорость тележки, угол наклона стержня и скорость вершины стержня. С учетом действия — движения влево или вправо — данной информации достаточно для определения следующего состояния среды.

Вместе с тем внутреннее состояние среды может быть скрыто от агента, то есть $s_t \neq s_t^{int}$. Такие типы сред известны как *частично наблюдаемые* МППР (partially

observable MDP, POMDP). Функция переходов у частично наблюдаемых МППР такая же, как у МППР. Однако частично наблюдаемые МППР предоставляют агенту не внутреннее состояние среды s_t^{int}, а наблюдаемое состояние s_t. Это значит, что агенту больше не известно внутреннее состояние s_t^{int} и ему приходится делать вывод о нем исходя из всех наблюдаемых состояний (s_t, s_{t-1}... s_1, s_0).

Внутренние состояния полностью описывают интересующую нас систему. Например, в Pong из Atari во внутреннее состояние среды s_t^{int} будут включены местоположения и скорости ракеток и шарика. Наблюдаемые состояния s_t, как правило, содержат исходные данные, передаваемые сенсорами системы, например изображение игры. Во время игры по этому изображению мы делаем заключение о внутреннем состоянии среды.

Рассмотрим модифицированную версию среды CartPole. Предположим, что среда на каждом временном шаге предоставляет только две единицы информации: позицию тележки относительно линейной оси и угол наклона стержня[1]. Это наблюдаемое состояние s_t, так как у агента есть доступ к данной информации. При этом среда, помимо позиции тележки и угла наклона стержня, отслеживает еще и скорости тележки и стержня. Это внутреннее состояние среды, так как оно полностью описывает тележку со стержнем.

До настоящего момента не было необходимости проводить различие между наблюдаемым и внутренним состояниями, так как мы рассматривали только МППР. Внутреннее состояние — это то, что агент наблюдает в МППР. Однако в частично наблюдаемых МППР нам нужен способ, позволяющий различать между собой наблюдаемые и внутренние состояния среды.

Частично наблюдаемые МППР можно разделить на три категории:

- полностью наблюдаемые при частично известной истории;
- полностью наблюдаемые при полностью известной истории;
- никогда полностью не наблюдаемые.

Полностью наблюдаемые МППР при частично известной истории. В таких средах внутреннее состояние s_t^{int} можно вывести из нескольких последних наблюдаемых состояний (s_t, s_{t-1}... s_{t-k}), где значение k мало, например от 2 до 4. В целом считается, что большинство игр Atari обладают этой характеристикой [65]. Рассмотрим в качестве примера наблюдаемое состояние в игре Breakout Atari. На рис. 12.2 показаны платформа агента, положение мячика, неразрушенные блоки, оставшееся количество жизней агента и счет. Этого почти достаточно, чтобы определить, в каком внутреннем состоянии находится игра, за исключением одной существенной части информации — направления, в котором движется мяч. Его можно получить, найдя разницу между s_t и s_{t-1}. Зная, в каком направлении летит мяч, можно опре-

[1] Данная модификация была предложена Вьерстра и др. в Recurrent Policy Gradients [147].

делить, в каком состоянии находится среда[1], и предположить, что произойдет дальше. Если важно вывести не только скорость мяча, но и его ускорение, то для оценки этих значений можно использовать три предыдущих наблюдаемых состояния.

Рис. 12.2. Игра Breakout из Atari

Как мы уже видели в главе 5, для игр Atari общепринятой практикой являются выбор каждого четвертого кадра и объединение четырех[2] таких кадров с учетом пропущенных. Агент может воспользоваться различиями между объединенными кадрами, переданными на вход как одно значение, для выведения полезной информации о движениях объектов в игре.

Полностью наблюдаемые МППР при полностью известной истории. В этих средах всегда можно определить внутреннее состояние игры, проследив историю всех наблюдаемых состояний. Данное свойство есть, например, у Т-образного лабиринта, предложенного Бремом Бакером в *Reinforcement Learning with Long Short-Term Memory* [10]. На рис. 12.3 показана среда, которая состоит из длинного коридора с перпендикулярным ему отрезком в конце. Агент начинает всегда в одном и том же состоянии внизу Т. Он наблюдает только свое ближайшее окружение, и на каждом временном шаге ему доступны четыре действия — движения вверх, вниз, влево или вправо. Агенту нужно прийти в целевое состояние, которое всегда находится в одном из концов поперечины.

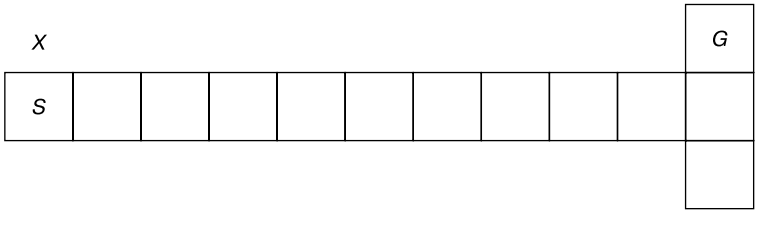


Рис. 12.3. Т-образный лабиринт [10]

В начальном состоянии игры агент наблюдает, на каком конце перекладины находится цель. В каждом эпизоде положение целевого состояния меняется, переходя

[1] В Breakout из Atari мячик движется с постоянной скоростью, поэтому мы можем игнорировать информацию более высокого порядка, такую как ускорение.

[2] Не совсем верно, здесь происходит наложение трех пропущенных кадров и того кадра, который показывается агенту. — *Примеч. науч. ред.*

на противоположный ее конец. По достижении перекрестка агент может выбрать — пойти налево или направо. Если он движется к целевому состоянию, игра завершается с вознаграждением 4, если же к неверному концу поперечины — с вознаграждением –1. При условии что агент может помнить то, что наблюдал в начальном состоянии, всегда возможно его оптимальное поведение. Следовательно, эта среда полностью наблюдаемая при полностью известной истории наблюдаемых состояний.

Рассмотрим другой пример из DMLab-30 [12] — библиотеки с открытым исходным кодом от DeepMind. Многие из ее сред спроектированы специально для тестирования памяти агента, следовательно, они попадают в категорию частично наблюдаемых МППР. `natlab_varying_map_regrowth` — это задание, в котором агент должен собирать грибы в натуралистичной среде. Наблюдаемое состояние — это изображение в формате RGBD[1], генерируемое в соответствии с текущей позицией агента. Приблизительно через минуту грибы снова вырастают в тех же самых местах, то есть агенту выгодно помнить, какие грибы он собрал и как давно это произошло. Эта среда интересна тем, что количество наблюдаемых состояний, необходимое агенту для вывода внутреннего состояния игры, варьируется в зависимости от шага и предпринятых агентом действий. По этой причине целесообразно использовать полную историю наблюдаемых состояний, чтобы агент не упустил важную информацию.

Никогда полностью не наблюдаемые МППР. В этих средах невозможно сделать вывод о внутреннем состоянии s_t^{int}, даже когда известна вся история наблюдаемых состояний ($s_0, s_1 \ldots s_{t-1}, s_t$). Покер — игра с такой характеристикой: даже если вам известны все сданные до сих пор карты, вы не знаете, какие из них на руках у игроков.

Или рассмотрим задачу навигации, в которой агент помещен в большую комнату с рядом разноцветных шаров. Расположение шаров генерируется случайным образом в начале каждого эпизода, и в комнате всегда есть только один красный шар. Для восприятия среды агент экипирован черно-белой камерой от первого лица, поэтому изображения с камеры — это наблюдаемые состояния. Предположим, задание агента состоит в том, чтобы перейти к красному шару. Задача непротиворечива, но без цветных изображений он не может воспринимать цвета шаров, то есть никогда не располагает достаточной информацией для решения задачи.

12.2.2. Выбор сетей для сред

Если дана новая среда, то как можно определить, является она МППР или одним из трех типов частично наблюдаемых МППР? Хороший подход заключается в том, чтобы уделить некоторое время тому, чтобы понять среду. Как человек выполнил бы это задание? Сам попытался бы поиграть в среде. Можем ли мы решить, какое действие лучше выбрать на каждом временном шаге, основываясь на информации, полученной из одного наблюдаемого состояния? Если да, то среда,

[1] RGB плюс глубина (Depth).

скорее всего, МППР. Если нет, то тогда сколько наблюдаемых состояний нужно запомнить для оптимального выполнения задания? Это несколько состояний или вся история? Тогда, по всей видимости, среда — это частично наблюдаемый МППР, который является полностью наблюдаемыми, когда история известна частично или полностью. Наконец, рассмотрим, отсутствует ли в истории наблюдаемых состояний какая-нибудь важная информация? Если это так, то среда может быть частично наблюдаемым МППР, который никогда не наблюдается полностью. Как это влияет на потенциальную производительность? Все еще можно получить хорошее решение или задача неразрешима? Если приемлемая производительность может быть достигнута, невзирая на отсутствие информации, то применение глубокого RL все еще возможно.

Мы охарактеризовали среды по возможности их *наблюдения* — в какой мере внутреннее состояние среды может быть выведено из наблюдаемых состояний. Этот подход можно совместить с информацией о пространстве состояний среды, чтобы получить некоторое представление о том, какая архитектура нейронной сети будет наиболее подходящей для агента.

Относительно возможности наблюдения среды наиболее важно для нейронной сети, хранит она *состояние* или нет, то есть может ли запоминать историю наблюдаемых состояний.

MLP и CNN не хранят состояния, поэтому они больше подходят для сред, которые являются МППР, так как им не нужно помнить никакой истории. Они также могут показывать хорошую производительность, будучи примененными к частично наблюдаемым МППР, которые становятся полностью наблюдаемыми при частично известной истории. Однако в этом случае необходимо так преобразовать наблюдаемые состояния, чтобы на вход сети поступала информация о k предыдущих временных шагах. Таким способом можно обеспечить передачу в MLP и CNN в виде одного входного значения информации, достаточной для получения состояния среды.

RNN хранят состояния, поэтому они лучше всего подходят для частично наблюдаемых МППР, которые являются полностью наблюдаемыми, если известна вся история. Это связано с тем, что задача требует запоминания потенциально длинной последовательности наблюдаемых состояний. RNN могут достигать хорошей производительности в частично наблюдаемых МППР, которые никогда не являются полностью наблюдаемыми. Но в этом случае высокая производительность агента не гарантируется — может оказаться, что слишком много информации отсутствует.

Как влияет пространство состояний на выбор сети? В разделе 12.1 говорится, что из трех семейств сетей CNN — наиболее подходящие для обучения по данным в виде изображений. Следовательно, если наблюдаемые состояния — это изображения, как в играх Atari, то обычно лучше всего использовать CNN. В остальных случаях достаточно MLP. Если наблюдаемое состояние представлено комбинацией данных в виде изображений и данных в других форматах, то для их обработки рассмотрите возможность применения множества подсетей, включающего MLP и CNN.

Если среда полностью наблюдаема при целиком известной истории или никогда не наблюдается полностью, то важно иметь подсеть RNN. Попробуйте гибридную сеть, которая использует CNN или MLP для обработки наблюдаемых состояний перед передачей данных в подсеть RNN.

На рис. 12.4 подводится итог обсуждения сред и архитектуры сетей. На нем приведены некоторые распространенные варианты сетей для каждого из трех семейств: MLP, CNN и RNN, а также для гибридной CNN-RNN. В прямоугольниках со сплошной рамкой показаны обязательные подсети, с пунктирной рамкой — дополнительные. Также на рисунке описаны характеристики входных данных сетей, примеры некоторых сред и несколько алгоритмов, которые достигают лучших показателей производительности при соответствующих типах сетей.

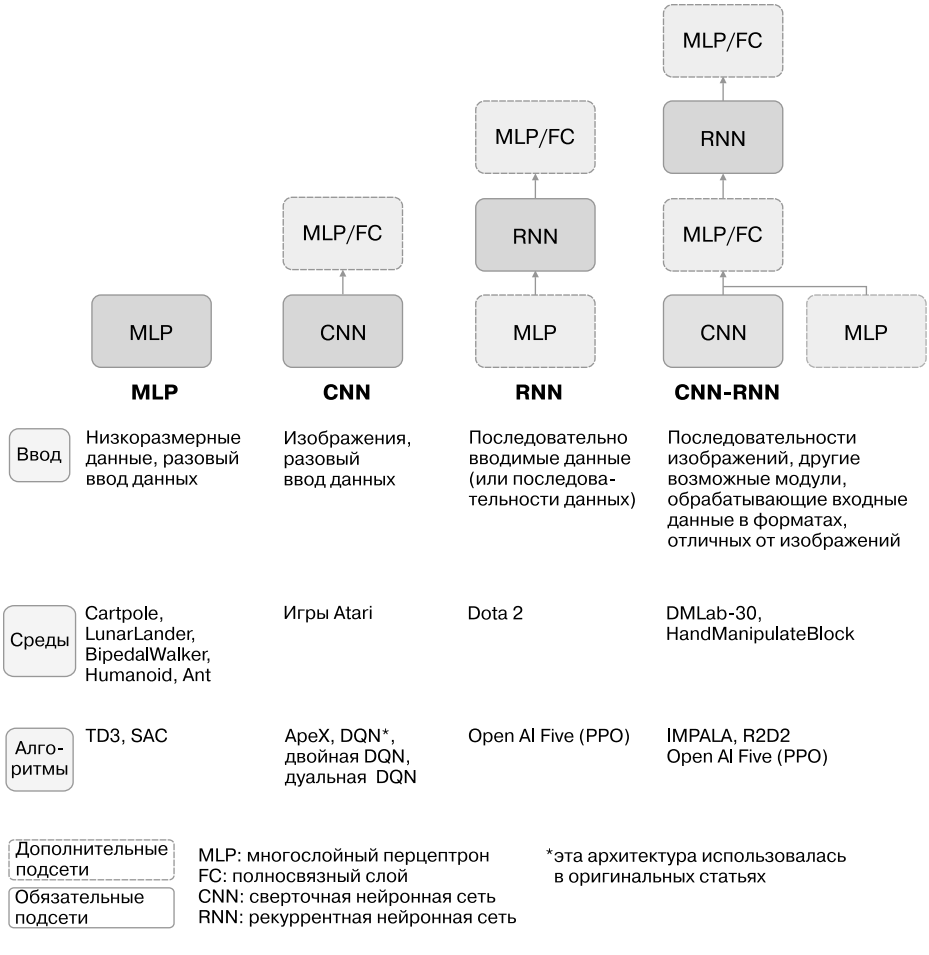


Рис. 12.4. Семейства нейронных сетей

В число сред, которые мы обсуждаем в книге, входят CartPole, LunarLander и BipedalWalker. Через OpenAI Gym доступны две более сложные среды с непрерывным управлением — Humanoid и Ant. Все они являются МППР, которые предоставляют агенту низкоразмерное наблюдаемое состояние. Таким образом, их задачи лучше всего решать с помощью MLP. На момент написания этих строк алгоритмами двойного градиента детерминированной стратегии с задержкой (Twin Delayed Deep Deterministic Policy Gradient, TD3) [42] и мягкого актора-критика (Soft Actor-Critic, SAC) [47] в средах Humanoid и Ant с применением лишь MLP были достигнуты наилучшие результаты.

На протяжении этой книги мы часто обсуждаем игры Atari. Эти среды по большей части — частично наблюдаемые МППР, которые являются полностью наблюдаемыми при частично известной истории. Наблюдаемое состояние — это изображение в формате RGB, в связи с чем для подобных задач лучше всего подходят CNN. Для таких сред за последние несколько лет наилучшие оценки были получены сетями, содержащими компонент CNN, например, DQN [88], двойная DQN [141], дуэльная DQN [144] и ApeX [54].

Dota 2 — сложная игра с большим количеством агентов, для которой требуются стратегии с длинным временным горизонтом. Она может быть охарактеризована как частично наблюдаемый МППР, который становится полностью наблюдаемым, если известна полная история. Агенты PPO из OpenAI Five [104] применяют получаемые из API игры наблюдаемые состояния в формате, отличном от изображений. Состояние содержит 20 000 элементов и включает карту игры и информацию о других агентах. Для такого состояния подходит тип сети MLP. Поскольку игра требует долгосрочных стратегий, агенту нужно помнить историю наблюдаемых состояний. Данные, полученные на выходе MLP, затем передаются в сеть LSTM с 1024 элементами. За дополнительными сведениями об архитектуре модели обращайтесь к разделу «Структура модели» в посте об OpenAI Five [104]. Объединив значительные усилия по проектированию с вычислительной мощью, они в 2019 году смогли победить лучших в мире игроков [107].

В средах из DMLab-30 наблюдаемые состояния основаны на изображениях. Наилучшая архитектура — объединение модулей CNN и RNN, чтобы сети могли хорошо обрабатывать изображения и отслеживать историю наблюдаемых состояний. IMPALA [37] и R2D2 [65] — два алгоритма, обеспечивающих наивысшую производительность в этих средах, в обоих используются гибридные сети CNN-RNN.

И наконец, еще одна предоставляемая OpenAI среда — HandManipulateBlock. Цель — изменить ориентацию блока, находящегося в роботизированной руке. Это сложная среда с непрерывным управлением, в которой робот имеет 24 степени свободы. Наблюдаемое состояние представлено комбинацией трех изображений руки с блоком и вектора, описывающего расположение кончиков пальцев робота [98]. Чтобы внести разнообразие в обучающие данные, OpenAI в каждом эпизоде задает случайные значения некоторых внутренних параметров среды, таких как вес

блока. В связи с этим для решения задания агенту требуется выводить внутренние параметры для эпизода с помощью последовательностей наблюдаемых состояний, передаваемых в RNN. В используемой для этого задания сети для обработки и объединения наблюдаемых состояний применяются CNN и MLP, последовательность обработанных состояний передается в RNN.

Резюме. Нейронные сети можно разделить на семейства по типу данных, на которых они лучше всего обучаются. Есть три основных типа сетей: MLP, CNN и RNN, которые наилучшим образом подходят для обработки неупорядоченных низкоразмерных данных, изображений и последовательностей соответственно. Кроме того, возможно создание гибридных сетей, состоящих из множества подсетей, принадлежащих к разным семействам. Мы также обсудили разницу между МППР и частично наблюдаемыми МППР. Среды в RL различаются в зависимости от пространства наблюдаемых состояний и от того, являются они МППР или частично наблюдаемыми МППР. Как мы видели, на основании этой информации можно выбрать наилучшую архитектуру сети для решения задачи в конкретной среде.

Теперь обратимся к практической части проектирования нейронных сетей для глубокого RL.

12.3. Net API

В алгоритмах глубокого RL задействуются нейронные сети. Хотя аппроксимируемые сетями функции различаются в зависимости от алгоритма, у процессов обучения этих сетей много общих процедур, таких как вычисление функции потерь и обновление параметров. Следовательно, для всех нейронных сетей, применяемых разными алгоритмами, целесообразно использовать стандартизированный Net API. Кроме того, модульные компоненты сетей позволяют уменьшить количество кода и тем самым упростить чтение и отладку реализованных алгоритмов.

К Net API для глубокого RL предъявляются следующие требования.

1. **Выведение размерности входного и выходного слоев.** Размерности входного и выходного слоев меняются в зависимости от среды и алгоритма. Они могут быть выведены автоматически, чтобы пользователям не приходилось каждый раз указывать их вручную. Эта функция экономит время и уменьшает количество ошибок в коде.
2. **Автоматическое создание сети.** Архитектура сети зависит от среды и серьезно влияет на производительность алгоритма. Поскольку в RL принято пробовать много разных видов архитектуры сети, лучше указывать ее в конфигурационном файле, вместо того чтобы менять код. Для этого нужен метод, который на основе конфигурации будет автоматически создавать соответствующую нейронную сеть.

3. **Этап обучения.** Всем нейронным сетям нужны этапы обучения, которые включают расчет функции потерь, вычисление градиентов и обновление параметров сети. Эти шаги стоит стандартизировать в рамках одной функции, которую могут повторно использовать все алгоритмы.
4. **Предоставление базовых методов.** API как оболочка над библиотекой нейронных сетей должен также предоставлять наиболее часто используемые методы, такие как функции активации, оптимизаторы, уменьшение скорости обучения и контрольные точки модели.

Приведенный в листинге 12.1 `Net` — это базовый класс, в котором реализованы несколько распространенных методов, он имеется в SLM Lab в `slm_lab/agent/net/base.py`. Этот базовый класс расширяется классами `MLPNet`, `ConvNet` и `RecurrentNet`, созданными для разных типов нейронных сетей.

Листинг 12.1. Определение методов API в базовом классе Net

```
# slm_lab/agent/net/base.py

class Net(ABC):
    '''Абстрактный класс Net для определения методов API'''

    def __init__(self, net_spec, in_dim, out_dim):
        '''
        @param {dict} net_spec — файл spec для net
        @param {int|list} in_dim — размерность(и) на входе сети.
        ➥ Обычно in_dim=body.state_dim
        @param {int|list} out_dim — размерность(и) на выходе
        ➥ сети. Обычно out_dim=body.action_dim
        '''
        ...

    @abstractmethod
    def forward(self):
        '''Следующий шаг для конкретной архитектуры сети'''
        ...

    @net_util.dev_check_train_step
    def train_step(self, loss, optim, lr_scheduler, clock, global_net=None):
        '''Выполняет одно обновление параметров сети'''
        ...

    def store_grad_norms(self):
        '''Хранит нормы градиентов для отладки'''
        ...
```

Класс `Net` поддерживается также набором функций-утилит, с помощью которых автоматически создается сеть и предоставляются полезные методы базовой библиотеки нейронных сетей. Рассмотрим каждое из требований к API детально.

12.3.1. Выведение размерностей входного и выходного слоев

Чтобы создать сеть правильно, нужно вывести размерности входного и выходного слоев. Размерность входного слоя задается пространством состояний среды. Например, если состояния среды — это векторы с 16 элементами, то во входном слое должно быть 16 узлов. А если наблюдаемые состояния — это изображения в градациях серого размером 84×84 пикселов, то входной слой должен быть определен как матрица размерности (84, 84). Кроме того, CNN должна включать несколько каналов, а для RNN нужно знать длину последовательности.

Размерность выходного слоя определяется пространством действий среды и алгоритмом, применяемым для обучения агента. Исходя из обсуждаемых в этой книге алгоритмов глубокого RL, нужно рассмотреть три варианта выходов сети. Агент может настраивать *Q*-функцию, стратегию, или стратегию и *V*-функцию. Тогда выходы сети будут представлены *Q*-значениями, вероятностями действий или вероятностями действий и *V*-значениями соответственно.

Определить размерность на выходе сложнее, чем на входе, поэтому в SLM Lab включены несколько вспомогательных методов, приведенных в листинге 12.2. В библиотеке эти методы расположены в `slm_lab/agent/net/net_util`.

Сначала метод `get_policy_out_dim` (строки 3–19) выводит размерность выходного слоя, когда сеть настраивает стратегию.

- Размерность пространства действий среды сохраняется в атрибуте агента `body.action_dim` (строка 5).
- Случаи дискретного пространства обрабатываются в строках 6–12 — когда действий много (строки 7–9) или когда оно одно (строки 10–12).
- Случаи непрерывного пространства обрабатываются в строках 13–18 — когда действий много (строки 17 и 18) или когда оно одно (строки 15 и 16).

Затем метод `get_out_dim` (строки 21–31) выводит выходную размерность из алгоритма сети. Если алгоритм настраивает *V*-функцию (использует критика), то добавляется дополнительный выходной слой с одним выходным значением (строки 24–28). В ином случае размерность на выходе — это просто размерность на выходе сети стратегии (строки 29 и 30).

Q-функцию можно рассматривать как экземпляр дискретной стратегии Argmax (для максимального *Q*-значения вероятность равна 1). Таким образом, мы можем использовать выходную размерность сети стратегии, когда алгоритм задействует *Q*-сеть.

Листинг 12.2. Вспомогательный метод для определения размерности выходного слоя сети

```
 1 # slm_lab/agent/net/net_util.py
 2
 3 def get_policy_out_dim(body):
 4     '''Вспомогательный метод для создания сети стратегии out_dim
       ➥ для основной части сети в зависимости от is_discrete, action_type'''
 5     action_dim = body.action_dim
 6     if body.is_discrete:
 7         if body.action_type == 'multi_discrete':
 8             assert ps.is_list(action_dim), action_dim
 9             policy_out_dim = action_dim
10         else:
11             assert ps.is_integer(action_dim), action_dim
12             policy_out_dim = action_dim
13     else:
14         assert ps.is_integer(action_dim), action_dim
15         if action_dim == 1: # если действие одно, используйте [loc, scale]
16             policy_out_dim = 2
17         else: # действий много — [locs], [scales]
18             policy_out_dim = [action_dim, action_dim]
19     return policy_out_dim
20
21 def get_out_dim(body, add_critic=False):
22     '''Создание out_dim класса NetClass для основной части сети,
       ➥ исходя из is_discrete, action_type и из того,
       ➥ добавляется ли модуль критика'''
23     policy_out_dim = get_policy_out_dim(body)
24     if add_critic:
25         if ps.is_list(policy_out_dim):
26             out_dim = policy_out_dim + [1]
27         else:
28             out_dim = [policy_out_dim, 1]
29     else:
30         out_dim = policy_out_dim
31     return out_dim
```

Метод `get_out_dim` используется для создания нейронных сетей внутри классов алгоритмов. В листинге 12.3 приведен пример из `Reinforce`. Когда сеть создается в методе `init_nets` (строки 7–13), выходная размерность определяется с помощью метода `get_out_dim` (строка 10).

Листинг 12.3. Создание сети в классе Reinforce

```
1 # slm_lab/agent/algorithm/reinforce.py
2
3 class Reinforce(Algorithm):
4     ...
```

```
 5
 6     @lab_api
 7     def init_nets(self, global_nets=None):
 8         ...
 9         in_dim = self.body.state_dim
10         out_dim = net_util.get_out_dim(self.body)
11         NetClass = getattr(net, self.net_spec['type'])
12         self.net = NetClass(self.net_spec, in_dim, out_dim)
13         ...
```

Выбранный класс `Net` (`MLPNet`, `ConvNet` или `RecurrentNet`) инициализируется с помощью спецификации net и определенных входной и выходной размерностей (строки 11 и 12). Теперь рассмотрим, как с помощью этих выходных значений класс `Net` создает экземпляр нейронной сети.

12.3.2. Автоматическое создание сети

Класс `Net` умеет строить нейронные сети по заданной спецификации net. В листинге 12.4 приводятся два примера спецификаций net: один для MLP (строки 2–10), а другой для CNN (строки 21–44). У MLP есть скрытый слой из 64 элементов (строки 7–8), функция активации SeLU (строка 9), норма усечения градиентов до 0,5 (строка 10), функция потерь (строки 11–13), оптимизатор (строки 14–17) и расписание уменьшения скорости обучения (строка 18). У CNN есть три скрытых сверточных слоя (строки 26–31) и один полносвязный слой (строка 32), в остальной части ее спецификации указаны стандартные компоненты, аналогичные MLP (строки 33–43).

Листинг 12.4. Пример спецификации net для построения Net

```
 1 {
 2     "reinforce_cartpole": {
 3         "agent": [{
 4             "name": "Reinforce",
 5             ...
 6             "net": {
 7                 "type": "MLPNet",
 8                 "hid_layers": [64],
 9                 "hid_layers_activation": "selu",
10                 "clip_grad_val": 0.5,
11                 "loss_spec": {
12                     "name": "MSELoss"
13                 },
14                 "optim_spec": {
15                     "name": "Adam",
16                     "lr": 0.002
17                 },
```

```
18                "lr_scheduler_spec": null
19            }
20    ...
21    "dqn_pong": {
22        "agent": [{
23            "name": "DQN",
24            ...
25            "net": {
26                "type": "ConvNet",
27                "conv_hid_layers": [
28                    [32, 8, 4, 0, 1],
29                    [64, 4, 2, 0, 1],
30                    [64, 3, 1, 0, 1]
31                ],
32                "fc_hid_layers": [256],
33                "hid_layers_activation": "relu",
34                "clip_grad_val": 10.0,
35                "loss_spec": {
36                   "name": "SmoothL1Loss"
37                },
38                "optim_spec": {
39                    "name": "Adam",
40                    "lr": 1e-4,
41                },
42                "lr_scheduler_spec": null,
43                ...
44 }
```

Внутри класса `Net` используется другая вспомогательная функция для построения его слоев с помощью класса-контейнера `Sequential` из PyTorch. Класс `MLPNet` применяет метод `build_fc_model` (листинг 12.5) для построения полносвязных слоев, а `ConvNet` дополнительно использует метод `build_conv_layers` (листинг 12.6) для сверточных слоев.

Метод `build_fc_model` принимает список `dims` размерностей слоев, которые обозначены в спецификации net как `hid_layers` для MLP или `fc_hid_layers` для CNN. Проходя в цикле по размерностям, он строит полносвязные слои `nn.Linear` (строка 10) с добавлением функций активации (строки 11 и 12). Затем, чтобы полностью сформировать нейронную сеть, эти слои помещаются в контейнер `Sequential` (строка 13).

Листинг 12.5. Автоматическое создание сети, построение полносвязных слоев

```
1 # slm_lab/agent/net/net_util.py
2
3 def build_fc_model(dims, activation):
4     '''Строит полносвязную модель, чередуя nn.Linear
      ➥ и activation_fn'''
```

```
    assert len(dims) >= 2, 'в dims должны быть по крайней мере
    ➥ размерности на входе и выходе'
    # берем попеременно элементы из dims и создаем
    ➥ пары dims (in, out) для каждого слоя
    dim_pairs = list(zip(dims[:-1], dims[1:]))
    layers = []
    for in_d, out_d in dim_pairs:
        layers.append(nn.Linear(in_d, out_d))
        if activation is not None:
            layers.append(get_activation_fn(activation))
    model = nn.Sequential(*layers)
    return model
```

В классе `ConvNet` определен специальный метод `build_conv_layers` для аналогичного построения сверточных слоев, хотя у него есть дополнительные компоненты, характерные для сверточных сетей.

Листинг 12.6. Автоматическое создание сети, построение сверточных слоев

```
'# slm_lab/agent/net/conv.py

class ConvNet(Net, nn.Module):
    ...

    def build_conv_layers(self, conv_hid_layers):
        '''
        Строит все сверточные слои в сети
        ➥ и сохраняет их в модели Sequential
        '''
        conv_layers = []
        in_d = self.in_dim[0]
        for i, hid_layer in enumerate(conv_hid_layers): # входной канал
            hid_layer = [tuple(e) if ps.is_list(e) else e for e in hid_layer]
            ➥ # ограничение на преобразование списка в кортеж
            # hid_layer = out_d, ядро, шаг, дополнение, расширение
            conv_layers.append(nn.Conv2d(in_d, *hid_layer))
            if self.hid_layers_activation is not None:
                conv_layers.append(net_util.get_activation_fn(
                ➥ self.hid_layers_activation))
            # В первый слой не включается нормализация пакета
            if self.batch_norm and i != 0:
                conv_layers.append(nn.BatchNorm2d(in_d))
            in_d = hid_layer[0] # обновление до out_d
        conv_model = nn.Sequential(*conv_layers)
        return conv_model
```

Исходный код для автоматического построения разных типов нейронных сетей находится в SLM Lab в `slm_lab/agent/net/`. Читать его не обязательно, но может быть полезно, чтобы лучше понимать классы `Net`.

12.3.3. Шаг обучения

В базовом классе `Net` реализован стандартизированный метод `train_step`, используемый всеми подклассами для обновления параметров. Он соответствует стандартной логике настройки сети в глубоком обучении, как показано в листинге 12.7. Рассмотрим основные шаги.

1. Обновление скорости обучения по расписанию изменения скорости обучения и метода `clock` (строка 8).
2. Очищение x существующих градиентов (строка 9).
3. Алгоритм рассчитает свою функцию потерь перед передачей ее значения в этот метод. Для вычисления градиента с помощью обратного распространения вызывается `loss.backward()` (строка 10).
4. При необходимости выполняется усечение градиента (строки 11 и 12). Это предотвратит слишком значительное обновление параметров.
5. С помощью оптимизатора обновляются параметры сети (строка 15).
6. Если обучение происходит асинхронно, то в этот метод будет передано `global_net`, чтобы передать локальные градиенты в глобальную сеть (строки 13 и 14). После обновления сети последние параметры глобальной сети копируются в локальную сеть (строки 16 и 17).
7. Функция-декоратор `@net_util.dev_check_training_step` применяется для проверки того, обновлены ли параметры сети. Она активна только в режиме разработки (более детально рассматривается в разделе 10.2).

Листинг 12.7. Стандартизированный метод обновления параметров сети

```
 1 # slm_lab/agent/net/base.py
 2
 3 class Net(ABC):
 4     ...
 5
 6     @net_util.dev_check_train_step
 7     def train_step(self, loss, optim, lr_scheduler, clock, global_net=None):
 8         lr_scheduler.step(epoch=ps.get(clock, 'frame'))
 9         optim.zero_grad()
10         loss.backward()
11         if self.clip_grad_val is not None:
12             nn.utils.clip_grad_norm_(self.parameters(), self.clip_grad_val)
13         if global_net is not None:
14             net_util.push_global_grads(self, global_net)
15         optim.step()
16         if global_net is not None:
17             net_util.copy(global_net, self)
18         clock.tick('opt_step')
19         return loss
```

12.3.4. Предоставление базовых методов

В листинге 12.8 приведены несколько примеров того, как с помощью короткого кода можно предоставить полезные функции PyTorch в SLM Lab.

`get_activation_fn` (строки 3–6) и `get_optim` (строки 22–27) демонстрируют применение компонентов из спецификации net для получения и инициализации соответствующих классов PyTorch с целью их использования в классах `Net`.

`get_lr_scheduler` (строки 8–20) — это просто обертка для `LRSchedulerClass` из PyTorch. Можно также применять пользовательские расписания, определенные в SLM Lab.

`save` (строки 29–31) и `load` (строки 33–36) — простые методы для сохранения и загрузки параметров сети в контрольных точках.

Листинг 12.8. Предоставление распространенной функциональности PyTorch

```
 1 # slm_lab/agent/net/net_util.py
 2
 3 def get_activation_fn(activation):
 4     '''Вспомогательный метод для порождения
       ➥ функций активации для сети'''
 5     ActivationClass = getattr(nn, get_nn_name(activation))
 6     return ActivationClass()
 7
 8 def get_lr_scheduler(optim, lr_scheduler_spec):
 9     '''Вспомогательный метод для анализа параметров
       ➥ lr_scheduler и создания optim.lr_scheduler из PyTorch'''
10     if ps.is_empty(lr_scheduler_spec):
11         lr_scheduler = NoOpLRScheduler(optim)
12     elif lr_scheduler_spec['name'] == 'LinearToZero':
13         LRSchedulerClass = getattr(torch.optim.lr_scheduler, 'LambdaLR')
14         frame = float(lr_scheduler_spec['frame'])
15         lr_scheduler = LRSchedulerClass(optim, lr_lambda=lambda
           ➥  x: 1 - x / frame)
16     else:
17         LRSchedulerClass = getattr(torch.optim.lr_scheduler,
           ➥ lr_scheduler_spec['name'])
18         lr_scheduler_spec = ps.omit(lr_scheduler_spec, 'name')
19         lr_scheduler = LRSchedulerClass(optim, **lr_scheduler_spec)
20     return lr_scheduler
21
22 def get_optim(net, optim_spec):
23     '''Вспомогательный метод для анализа параметров оптимизатора
       ➥ и создания оптимизатора для сети'''
24     OptimClass = getattr(torch.optim, optim_spec['name'])
25     optim_spec = ps.omit(optim_spec, 'name')
```

```
    optim = OptimClass(net.parameters(), **optim_spec)
    return optim

def save(net, model_path):
    '''Сохранение весов модели по указанному пути'''
    torch.save(net.state_dict(), util.smart_path(model_path))

def load(net, model_path):
    Загрузка весов модели из указанного пути в модуль сети'''
    device = None if torch.cuda.is_available() else 'cpu'
    net.load_state_dict(torch.load(util.smart_path(model_path),
    ➥ map_location=device))
```

12.4. Резюме

В этой главе мы сосредоточились на проектировании и реализации нейронных сетей для глубокого RL. Вкратце рассмотрели три основных семейства сетей: MLP, CNN и RNN и обсудили некоторые рекомендации, как, основываясь на характеристиках среды, выбрать подходящее семейство сетей.

Важная характеристика среды — является она МППР или частично наблюдаемым МППР. Существует три типа частично наблюдаемых МППР: полностью наблюдаемые при частично известной истории, полностью наблюдаемые при полностью известной истории и никогда полностью не наблюдаемые.

MLP и CNN хорошо подходят для решения МППР и частично наблюдаемых МППР, которые являются полностью наблюдаемыми при частично известной истории. RNN и RNN-CNN — для частично наблюдаемых МППР, которые являются полностью наблюдаемыми при полностью известной истории. Кроме того, RNN могут (хотя это и не гарантировано) улучшить производительность в частично наблюдаемых МППР, которые никогда полностью не наблюдаются.

Есть набор методов, которые часто используются и для повторного применения стандартизированы как часть Net API. Они упрощают реализацию алгоритмов и включают выведение размерностей входного и выходного слоев, автоматическое создание сетей и стандартизированный шаг обучения.

12.5. Рекомендуемая литература

Основная

- *Nielsen M.* Neural Networks and Deep Learning. 2015 [92].
- *Goodfellow I., Bengio Y., Courville A.* Deep Learning. 2016 [45].

По CNN

- *LeCun Y.* Generalization and Network Design Strategies. 1989 [71].
- *Cox D., Dean T.* Neural Networks and Neuroscience-Inspired Computer Vision. 2014 [28].

По RNN

- *Karpathy A.* The Unreasonable Effectiveness of Recurrent Neural Networks. 2015 [66].
- *Cho K.* Natural Language Understanding with Distributed Representation. 2015. P. 11–53 [20].
- *Bakker B.* Reinforcement Learning with Long Short-Term Memory. 2002 [10].
- OpenAI Blog. OpenAI Five. 2018 [104].

13 Аппаратное обеспечение

Своим успехом глубокое RL частично обязано появлению мощного аппаратного обеспечения. При реализации алгоритмов глубокого RL не обойтись без базовых знаний о компьютере. К тому же эти алгоритмы потребляют в больших объемах данные, память и вычислительные ресурсы. При обучении агента полезно иметь возможность оценивать потребляемую алгоритмом память и необходимые ему вычислительные ресурсы, а также управлять эффективностью использования данных.

Эта глава призвана дать вам представление о типах данных, встречающихся в глубоком RL, их размерах и об оптимизации их применения. Сначала кратко описываются работа и взаимодействие компонентов аппаратного обеспечения, таких как центральный процессор, оперативная память и графический процессор. Затем в разделе 13.2 дан обзор типов данных, а в разделе 13.3 обсуждаются распространенные в глубоком RL типы данных и приводится несколько рекомендаций по работе с ними. В конце главы для справки приведены требования, предъявляемые к аппаратному обеспечению при запуске разных типов экспериментов в глубоком RL.

Эта информация пригодится при тщательной отладке и выборе аппаратного обеспечения или управлении им в глубоком RL.

13.1. Компьютер

Сейчас компьютеры повсеместно — это наши телефоны, ноутбуки, настольные компьютеры и облачные (удаленные) сервисы. Мы прошли долгий путь с тех пор, как Ада Лавлейс написала первый алгоритм, а Алан Тьюринг изобрел универсальный компьютер — машину Тьюринга. Первые компьютеры были огромными механическими устройствами, собранными из по-настоящему движущихся частей, куда программы загружались с перфокарт и в которых ошибки были самыми настоящими жуками[1]. Нынешние компьютеры электронные, маленькие и быстрые и внешне сильно отличаются от своих предшественников. Большинство людей пользуются

[1] Забавный факт о происхождении термина «компьютерный баг» — это в буквальном смысле жук, который попал внутрь раннего механического компьютера и вызвал в нем сбой. Людям пришлось открыть машину и почистить ее.

ими, совершенно не интересуясь тем, как они работают, — это привилегия живущих в компьютерную эпоху.

Даже эволюционируя, компьютер остается просто реализацией машины Тьюринга, и это значит, что определенные элементы его конструкции не меняются. Компьютер состоит из процессора и оперативной памяти, которые соответствуют считывающей головке и ленте в машине Тьюринга. Архитектура современного компьютера значительно усложнилась в основном ради решения практических задач, таких как хранение и передача данных и ускоренные чтение и запись. Но в конечном счете компьютер все так же обрабатывает информацию и в нем всегда есть процессор и оперативная память.

Сначала рассмотрим процессор. Сейчас в нем содержится несколько вычислительных ядер — процессоров (central processing unit, CPU), например два или четыре. Каждое ядро может быть многопоточным, что позволяет ему запускать единовременно более одного потока. Поэтому на упаковке процессора может быть написано, например, «2 ядра, 4 потока» — подразумевается, что в нем два ядра, в каждом из которых по два потока.

Рассмотрим этот пример более детально. Компьютер с двумя ядрами и четырьмя потоками может выдавать количество потоков за количество процессоров, хотя число ядер меньше. Это обусловлено тем, что четыре потока дают возможность запустить четыре процесса со 100%-ной загрузкой каждый. Тем не менее при необходимости можно запустить только два процесса без гиперпоточности и максимизировать производительность двух ядер. Вот так центральный процессор и может показывать 200 % загрузки. Максимальный процент зависит от количества потоков в одном ядре.

В зависимости от режима ядра могут быть реорганизованы и действовать, как будто процессоров больше, чем на самом деле. По этой причине настоящие ядра известны как физические процессоры, а организованные потоки — как логические процессоры. Это можно проверить с помощью команды терминала `lscpu` в Linux или `SPHardwareDataType` в `system_profiler` в MacOS. В листинге 13.1 показан сервер Linux с 32 логическими процессорами, а фактически это 16 физических процессоров с двумя потоками на каждом.

Листинг 13.1. Пример вывода для команды lscpu, в котором показана информация о процессоре на сервере Linux. В машине есть 16 физических ядер и 32 логических ядра, частота процессора — 2,30 ГГц

```
1 # На Linux для вывода информации о процессоре запустите `lscpu`
2 $ lscpu
3 Architecture:                X86_64
4 CPU op-mode(s):              32-bit, 64-bit
5 Byte Order:                  Little Endian
6 CPU(s):                      32
7 On-line CPU(s) list:         0-31
8 Thread(s) per core:          2
```

```
 9 Core(s) per socket:          16
10 Socket(s):                   1
11 NUMA node(s):                1
12 Vendor ID:                   GenuineIntel
13 CPU family:                  6
14 Model:                       79
15 Model name:                  Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz
16 Stepping:                    1
17 CPU MHz:                     2699.625
18 CPU max MHz:                 3000.0000
19 CPU min MHz:                 1200.0000
20 BogoMIPS:                    4600.18
21 Hypervisor vendor:           Xen
22 Virtualization type:         full
23 L1d cache:                   32K
24 L1i cache:                   32K
25 L2 cache:                    256K
26 L3 cache:                    46080K
27 NUMA node0 CPU(s):           0-31
28 ...
```

Значит, мы можем, увеличив до максимума количество потоков, запустить больше процессов, работающих медленнее, или, максимизировав количество ядер, — меньше процессов, работающих быстрее. Первый режим прекрасно подходит для поиска по гиперпараметрам, когда параллельно запускается много сессий, а второй стоит применять для ускоренного запуска нескольких отдельных сессий обучения. Ограничением является то, что один процесс не может использовать более одного ядра, поэтому, запустив меньше процессов, чем количество физических ядер, мы не получим никакого преимущества.

В процессоре есть внутренний таймер, который совершает один такт при выполнении каждой операции, его тактовая частота является мерой частоты процессора. Из названия модели в листинге 13.1 (строка 15) очевидно, что частота — 2,30 ГГц, то есть 2,3 млн тактов в секунду. Реальная тактовая частота в определенный момент времени может варьироваться в зависимости от множества факторов, таких как безопасная рабочая температура процессора (электроника по-прежнему изготавливается из физических деталей, которые могут перегреваться).

Процессор можно разогнать, чтобы заставить его работать еще быстрее. Это можно разрешить в BIOS при загрузке, но сначала для защиты процессора, как правило, устанавливают серьезное жидкостное охлаждение. Для глубокого RL разгонять процессор не обязательно, и это невыполнимо при использовании удаленного сервера, но если вы работаете на настольном компьютере, такой вариант возможен.

Теперь перейдем к памяти компьютера. Есть несколько видов памяти, которые образуют иерархию от ближайшей к процессору до наиболее отдаленной. Чем ближе память к процессору, тем меньше задержка при передаче данных, следовательно, доступ к данным происходит быстрее. Однако она, как правило, меньше, так как в маленьком процессоре много памяти не поместится. Чтобы компенсировать это,

фрагменты данных, которые применяются чаще всего, помещаются в память, близкую к процессору, тогда как большие, но менее востребованные данные — в более дальнюю.

Внутри процессора есть регистры для хранения инструкций и обрабатываемых данных — это самая маленькая и быстрая память. Далее идет кэш, в котором во избежание повторных вычислений содержатся часто и многократно используемые данные. В листинге 13.1 (строки 23–26) приведены разные виды кэш-памяти, все они медленнее, чем регистры.

Следующая память — оперативная (Random Access Memory, RAM). Во время выполнения программ в нее загружаются и в ней находятся данные, которые обрабатываются процессором. Когда говорят о загрузке данных в память или о том, что процессу не хватает памяти, имеется в виду RAM. На материнской плате компьютера обычно установлено не менее четырех слотов для оперативной памяти, то есть, если заменить четыре карты по 4 Гбайт на столько же карт по 16 Гбайт, память увеличится с 16 до 64 Гбайт.

При запуске программы нас обычно интересует использование процессора и оперативной памяти. Информация о процессоре приведена ранее, что касается информации о загрузке RAM, то в ней отображаются два числа: `VIRT` (виртуальная память) и `RES` (резидентная память). На рис. 13.1 показан снимок экрана системной панели управления, сгенерированной с помощью `Glances` [44]. В основных столбцах списка процессов отображены процент загрузки процессора, загрузка памяти (RAM) в процентах, `VIRT` и `RES`.

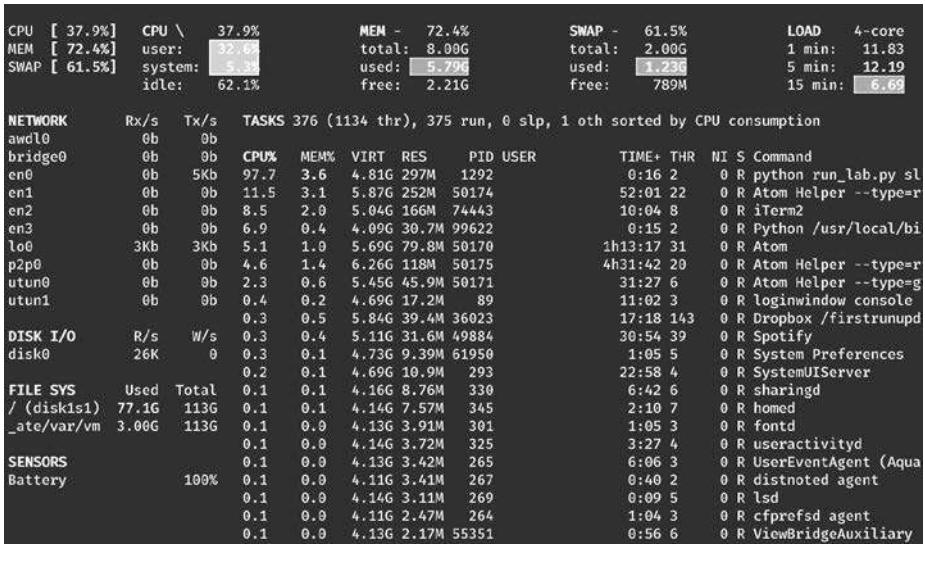


Рис. 13.1. Снимок экрана системной панели управления, сгенерированной с помощью инструмента мониторинга Glances [44]. На ней показана важная статистика загрузки процессора, использования памяти, ввода/вывода и процессов

В верхней части рисунка видно, что вся оперативная память (`MEM total`) этого компьютера составляет 8,00 Гбайт. В первой строке таблицы заданий можно видеть, что для выполнения процессов ЦП задействуется на 97,7 %.

Реальный объем RAM, занятой процессом, показан как резидентная память `RES` — он довольно мал, что добросовестно отражено в проценте `MEM`. Например, верхний процесс использует 3,6 % оперативной памяти, что соответствует 297 Мбайт резидентной памяти из всех доступных 8 Гбайт.

Однако расход виртуальной памяти значительно превосходит этот показатель — сумма значений в столбце `VIRT` превышает 40 Гбайт. Фактически виртуальная оперативная память — это просто оценка объема памяти, который может потребоваться программе. Например, в Python объявление большого пустого массива во время выполнения повысит оценку количества виртуальной памяти, но никакая память не будет занята, пока массив не заполнят реальные данные.

Размер виртуальной памяти не вызывает проблем, пока резидентная память не превышает реальный объем доступной RAM, иначе мы рискуем тем, что память закончится и работа компьютера нарушится.

Регистры, кэш и оперативная память — это быстрая рабочая память, используемая для вычислений, но она непостоянная. При перезагрузке компьютера эта память обычно очищается. В компьютере для постоянного хранения информации применяются жесткие диски или переносные накопители, такие как флеш-карты. В зависимости от аппаратного обеспечения диск может сохранять большое количество информации с довольно высокой скоростью чтения и записи, что отображено в левой части рис. 13.1, в пунктах `DISK I/O` и `FILE SYS`. Несмотря на это, данный тип памяти все равно медленнее, чем непостоянная память. Чем дальше память от процессора, тем меньше ее скорость чтения и записи, но тем больше объем. Принимая решение об управлении данными во время сессии обучения, пользуйтесь этим правилом.

Процессор вместе с оперативной памятью представляют собой универсальное устройство. Без процессора компьютер перестает быть компьютером. Согласно закону Мура его вычислительная мощность возрастает экспоненциально. Тем не менее похоже, что наша потребность в вычислительных ресурсах растет еще быстрее, ведь с увеличением мощности компьютеров расширился горизонт проблем, требующих решения. Расцвет глубокого обучения — прекрасный тому пример, он даже двигает вперед индустрию аппаратного обеспечения.

Хотя процессоры универсальные и довольно мощные, они не всегда поспевают за нашими вычислительными потребностями. К счастью, выполняемые нами вычисления зачастую состоят из особых типов операций. Для них можно спроектировать специальное аппаратное обеспечение, на котором высокая эффективность вычислений достигается за счет отказа от универсальности центрального процессора.

Один из таких примеров — матричные операции. Поскольку с помощью матриц можно кодировать и преобразовывать данные, на них основана работа с изображениями. Движения камеры, построение полигональной сетки (mesh), освещение, построение теней и определение траектории луча — вот типичные функции компьютерной графики, применяемые в любой из библиотек для визуализации, и все это матричные операции. Для воспроизведения видео с большой скоростью нужно выполнять множество таких вычислений — это обусловило развитие графических процессоров (graphics processing unit, GPU).

Сначала графические процессоры нашли применение в индустрии видеоигр. Предвестниками GPU были графические процессоры в игровых автоматах. Они развивались вместе с персональными компьютерами и видеоиграми, и в 1999 году в Nvidia изобрели GPU [95]. Параллельно творческие студии, такие как Pixar и Adobe, помогали заложить основы индустрии компьютерной графики, создавая и распространяя программное обеспечение и алгоритмы для графических процессоров. Игроки, графические дизайнеры и аниматоры стали основными потребителями GPU.

Сверточные сети, обрабатывающие изображения, также задействуют много матричных операций. Следовательно, то, что исследователи глубокого обучения начнут внедрять GPU, было лишь вопросом времени. Когда это наконец произошло, область глубокого обучения стала стремительно развиваться.

За счет того что дорогостоящие матричные вычисления производит графический процессор, центральный процессор освобождается для выполнения других заданий. Расчеты на GPU происходят параллельно и весьма эффективно из-за его узкой специализации. По сравнению с центральным процессором GPU содержит гораздо больше вычислительных ядер, и у него аналогичная архитектура памяти: регистры, кэш и RAM, хотя последняя обычно меньше. Чтобы отправить данные на обработку в графический процессор, сначала нужно загрузить их в оперативную память GPU. В листинге 13.2 приведен пример перемещения созданного на ЦП тензора PyTorch в оперативную память GPU.

Листинг 13.2. Пример создания на центральном процессоре тензора PyTorch и его дальнейшего перемещения в оперативную память графического процессора

```
 1 # пример кода для перемещения созданного в ЦП тензора в GPU(RAM)
 2 import torch
 3
 4 # указать доступное устройство
 5 device = 'cuda:0' if torch.cuda.is_available() else 'cpu'
 6
 7 # тензор сначала создается на ЦП
 8 v = torch.ones((64, 64, 1), dtype=torch.float32)
 9 # затем перемещается в GPU
10 v = v.to(device)
```

Вычисления, которые были медленными на ЦП, на GPU ощутимо ускоряются. Это открыло перед глубоким обучением новые возможности — исследователи стали тренировать большие по размеру и глубине сети на все возрастающем количестве данных. Это привело к получению самых передовых результатов во многих областях, включая компьютерное зрение, обработку естественного языка и распознавание речи. Графические процессоры способствовали быстрому развитию глубокого обучения и связанных областей, включая глубокое RL.

Несмотря на свою эффективность, графические процессоры все же оптимизированы для обработки графики, а не нейронных сетей. Для промышленных приложений даже возможностей GPU не хватает. Это стало толчком к развитию других типов специализированных процессоров. Стремясь удовлетворить эти потребности, Google в 2016 году объявила о выходе тензорного процессора (tensor processing unit, TPU). Это позволило привнести в приложения для глубокого обучения «на порядок более высокую производительность» [58]. Фактически созданный DeepMind алгоритм глубокого обучения с подкреплением AlphaGo работал на тензорных процессорах.

В этом разделе, посвященном программному обеспечению, мы обсудили основные составляющие компьютера — процессор и память. Центральный процессор универсален и может обрабатывать за раз ограниченное количество данных. В графическом процессоре есть много ядер, которые специализируются на параллельных вычислениях. Тензорные процессоры предназначены для тензорных вычислений в нейронных сетях. Независимо от специализации любой процессор нуждается в надлежащим образом спроектированной архитектуре памяти, включающей регистры, кэш и RAM.

13.2. Типы данных

Чтобы выполнять вычисления эффективно, нужно представлять себе область задачи и ее временную сложность. Для любых приложений, интенсивно использующих данные, таких как глубокое RL, полезно понимать некоторые важные детали и процессы на всех уровнях, от программного до аппаратного. Мы рассмотрели устройство компьютера, теперь разовьем интуитивное представление о данных, которые часто встречаются в глубоком RL.

Бит — наименьшая единица, применяемая для измерения количества информации. Современных данных очень много, поэтому нужен больший масштаб. Исторически так сложилось, что 8 бит — это 1 байт, или 8 бит = 1 байт. Начиная с этого места используются метрические приставки: 1000 байт — это 1 килобайт (Кбайт), 1 000 000 байт — это 1 мегабайт (Мбайт) и т. д.

Кодирование информации носит общий характер, поэтому любые данные можно закодировать как цифровые биты. Данные, если они еще не в числовом формате,

преобразуются в числа перед тем, как быть представленными в виде битов в аппаратном обеспечении. Изображение конвертируется в значения пикселов, звук — в частоты и амплитуды, категориальные данные — путем прямого кодирования в векторы, слова — в векторы и т. д. Затем на уровне аппаратного обеспечения числа преобразуются в биты.

Один байт, или 8 бит, был выбран в качестве общепринятой наименьшей единицы информации умышленно. Согласно правилу возведения в степень строка длиной 8 бит способна представлять $2^8 = 256$ различных предложений, то есть 1 байт может представлять 256 различных чисел. В библиотеках численных методов, таких как `numpy` [143], с помощью этого реализованы целые числа разных диапазонов, то есть беззнаковые 8-битовые целые `uint8`: [0, 255] — и целые 8-битовые со знаком `int8`: [−128, 127].

Байты подходят для хранения данных с небольшим диапазоном числовых значений, таких как значения пикселов изображений в градациях серого (0–255), поскольку занимаемое байтом пространство в памяти минимально. Размер одного числа формата `uint8` составляет 8 бит = 1 байт, поэтому для изображения в градациях серого, уменьшенного до размера 84×84 пикселов, нужно сохранить всего лишь 7096 чисел, то есть 7096 байт ≈ 7 Кбайт. На миллион таких изображений в памяти прецедентов понадобится 7 Кбайт · 1 000 000 = 7 Гбайт, что умещается в оперативной памяти самых современных компьютеров.

Разумеется, что такой маленький диапазон недостаточен для целей большинства вычислительных операций. Удвоив размер битовой строки, можно реализовать 16-битовые целые числа с гораздо большим диапазоном, `int16`: [−32 768, 32 768]. Кроме того, поскольку 16-битовая строка достаточно длинная, она может быть использована для реализации чисел с плавающей запятой, которые известны как формат `float` и являются десятичными числами. Это дает 16-битовые числа с плавающей запятой `float16`. Реализация десятичных чисел, которую мы не будем здесь рассматривать, несколько отличается от реализации целых чисел. Невзирая на это различие, как для целых, так и для десятичных чисел используются битовые строки одного и того же размера 16 бит = 2 байта.

Несколько раз повторив удвоение битов, можно реализовать целые числа и числа с плавающей запятой, занимающие 32 и 64 бита, что составляет 4 и 8 байт соответственно и дает `int32`, `float32`, `int64`, `float64`. Однако каждое удвоение битов будет уменьшать скорость вычислений в два раза, так как битовые операции нужно будет проводить на в два раза большем количестве элементов в битовой строке. Больше битов не всегда значит лучше, ведь за них приходится платить увеличением размеров и уменьшением скорости вычислений.

Целые числа могут быть реализованы как `int8`, `int16`, `int32`, `int64` с диапазоном, симметричным относительно 0. Смещение начала диапазона на 0 позволяет реа-

лизовать беззнаковые целые числа как `uint8`, `uint16`, `uint32`, `uint64`. Целые числа с разным количеством битов различаются размером, скоростью вычислений и представляемым диапазоном значений.

Присвоение целому числу со знаком значения за пределами его диапазона вызывает переполнение с непредсказуемым поведением и нарушает арифметические расчеты. Беззнаковые целые числа никогда не переполняются, вместо этого они молча произведут вычисления по модулю, например, `np.uint8(257) = np.uint8(1)`. Такое поведение, может быть, и желательно в некоторых приложениях, но в большинстве случаев следует аккуратно выполнять нисходящее приведение данных к меньшим форматам беззнаковых целых чисел. Если значения экстремумов нужно ограничить, выполните их усечение перед нисходящим приведением. Это показано в листинге 13.3.

Листинг 13.3. Простой скрипт, демонстрирующий сравнение размеров и методы оптимизации для разных типов данных

```
1 import numpy as np
2
3 # остерегайтесь переполнения диапазона при нисходящем приведении типов
4 np.array([0, 255, 256, 257], dtype=np.uint8)
5 # => array([  0,  255,  0,  1], dtype=uint8)
6
7 # если максимальное значение 255, выполнить усечение перед приведением
8 np.clip(np.array([0, 255, 256, 257], dtype=np.int16),
  ➥ 0, 255).astype(np.uint8)
9 # => array([  0,  255,  255,  255], dtype=uint8)
```

Начиная с 16 бит числа с плавающей запятой могут быть реализованы как `float16`, `float32` или `float64`. Они различаются размером, скоростью вычислений и точностью десятичных чисел, которые они могут представлять, — чем больше битов, тем выше точность. Поэтому `float16` известны как формат с половинной точностью, `float32` — с одинарной точностью, а `float64` — с двойной точностью (или просто double). Восьмибитовой реализации чисел с плавающей точкой нет, так как меньшая точность ненадежна для большинства приложений. Половинная точность вычислений позволяет удвоить скорость по сравнению с одинарной, и ее достаточно для расчетов там, где не требуется большого количества знаков после запятой. Она также идеальна для хранения, поскольку у большинства необработанных данных количество знаков после запятой невелико. Для большинства вычислений хватает одинарной точности, и это наиболее распространенный тип, используемый во многих программах, включая глубокое обучение. Двойная точность в основном зарезервирована для серьезных научных вычислений, таких как физические формулы с большим количеством значащих разрядов. При выборе подходящего типа для представления чисел с плавающей запятой среди прочего следует учитывать количество байтов, скорость, диапазон, точность и поведение при переполнении.

13.3. Оптимизация типов данных в RL

Мы уже получили понятие о том, как данные воплощаются на аппаратном уровне, а также об ограничениях размера, скорости вычислений, диапазона и точности. Листинг 13.4 содержит некоторые из этих типов данных в `numpy` и размеры занимаемой ими памяти. Рассмотрим теперь, как они связаны с данными, часто встречающимися в глубоком обучении и глубоком RL.

Листинг 13.4. Простой скрипт, в котором показаны разные типы данных и их размеры

```
import numpy as np

# Базовые типы данных и их размеры

# в компьютере данные кодируются как биты,
# поэтому размер данных определяется количеством битов
# например, и np.int16, и np.float16 занимают 2 байта,
  ➥ хотя числа с плавающей запятой представлены иначе, чем целые

# 8-битовые беззнаковые целые числа, диапазон [0, 255]
# размер: 8 бит = 1 байт
# применяются для хранения изображений или состояний малого размера
np.uint8(1).nbytes

# 16-битовые целые числа, диапазон: [-32768, 32768]
# размер: 16 бит = 2 байта
np.int16(1).nbytes

# 32-битовые целые числа, 4 байта
np.int32(1).nbytes

# 64-битовые целые числа, 8 байт
np.int64(1).nbytes

# числа с плавающей запятой половинной точности, 2 байта
# могут быть недостаточно точными для  вычислений,
  ➥ но подходят для хранения большинства данных
np.float16(1).nbytes

# числа с плавающей запятой одинарной точности, 4 байта
# используются по умолчанию в большинстве вычислений
np.float32(1).nbytes

# числа с плавающей запятой двойной точности, 8 байт
# в основном зарезервированы для высокоточных расчетов
np.float64(1).nbytes
```

В большинстве библиотек нейронные сети по умолчанию инициализируются и вычисляются с помощью `float32`. Значит, все их входные данные должны приводиться к `float32`. Однако `float32` не всегда оптимален для хранения данных из-за

своего большого размера в байтах (32 бита = 4 байта). Пусть один кадр данных RL составляет 10 Кбайт и в памяти прецедентов нужно хранить 1 млн кадров. Общий размер равен 10 Гбайт, что превышает объем обычной оперативной памяти. Вследствие этого к данным часто применяют нисходящее приведение типов и хранят их в формате с половинной точностью `float16`.

Большую часть данных, которые нужно хранить в глубоком RL, составляют состояния, действия, вознаграждения и булев сигнал done, означающий конец эпизода. Есть еще дополнительные переменные, необходимые для конкретных алгоритмов или для отладки, такие как *V*- и *Q*-значения, логарифмы вероятности и энтропия. Обычно данные значения — это либо небольшие целые числа, либо числа с плавающей запятой низкой точности, поэтому для хранения подойдут варианты `uint8`, `int8` или `float16`.

Примем 1 млн кадров в качестве меры стандартного количества данных, хранимых в памяти прецедентов. Большинство этих переменных, за исключением состояний, — скалярные значения. Для каждой скалярной переменной для хранения миллиона значений потребуется миллион элементов. При использовании `uint8` понадобится 1 000 000 · 1 байт = 1 Мбайт, а для `float16` — 1 000 000 · 2 байт = 2 Мбайт. Разница в размере в 1 Мбайт несущественна для большинства современных компьютеров, поэтому во избежание рисков случайной потери десятичных разрядов или ограничения диапазона значений для всех переменных повсеместно применяется `float16`.

Тем не менее большинство усилий по оптимизации памяти приходится на работу с состояниями, потому что они составляют большую часть данных, порождаемых в глубоком RL. Если состояние — относительно небольшой тензор, например вектор длиной 4 в CartPole, то можно использовать `float16`. Оптимизация нужна только тогда, когда все состояния большие, как изображения. В наши дни разрешение изображений, генерируемых камерами или игровыми движками, обычно выше, чем 1920 × 1080 пикселов, и размер таких изображений может быть несколько мегабайт. Пусть состояние — это маленькое изображение в формате RGB размером 256 × 256 пикселов со значениями `float32`. Тогда одно состояние в виде изображения составит (256 · 256 · 3) · 4 байт = 786 432 байта ≈ 786 Кбайт, а на загрузку миллиона таких изображений в оперативную память потребуется 786 Гбайт. Такие объемы не по силам даже большим серверам.

При преобразовании изображения в градации серого три цветовых канала сводятся до одного, то есть размер сокращается до 262 Гбайт, но это все равно слишком много. При понижении разрешения с 256 × 256 до 84 × 84 пиксела размер уменьшается приблизительно в девять раз — до 28 Гбайт. При таком сжатии часть информации будет потеряна, и значения пикселов после понижения разрешения будут зависеть от алгоритма сжатия. Обратите внимание также на то, что первоначальные значения в формате RGB — это целые числа с диапазоном 0–255, но при преобразовании в градации серого они конвертируются в `float32` с одинарной точностью

и диапазоном 0–255. Для большинства задач не требуются высокоточные значения пикселов, поэтому можно произвести обратное преобразование в `uint8`, что уменьшит размер еще в четыре раза.

В конечном виде изображение в градациях серого размером 84×84 пиксела со значениями `uint8` занимает лишь $84 \cdot 84 \cdot 1$ байт = 7096 байт. Миллион изображений — приблизительно 7 Гбайт, что легко уместится в оперативной памяти большинства современных компьютеров. В листинге 13.5 приведено сравнение размеров данных на всех этапах оптимизации.

Листинг 13.5. Размеры данных, предназначенных для хранения в памяти прецедентов, на различных этапах оптимизации

```
import numpy as np

# Размеры данных, удобные для отладки с использованием RAM

# память прецедентов с 1 млн данных в формате
  ➥ uint8 занимает 1 Мбайт
np.ones((1000000, 1), dtype=np.uint8).nbytes

# память прецедентов с 1 млн данных в формате
  ➥ float16 занимает 2 Мбайт
np.ones((1000000, 1), dtype=np.float16).nbytes

# уменьшенное изображение в градациях серого
  ➥ с пикселами типа uint8 занимает ~ 7 Кбайт
np.ones((84, 84, 1), dtype=np.uint8).nbytes

# память прецедентов с 1 млн изображений занимает ~ 7 Гбайт
np.ones((1000000, 84, 84, 1), dtype=np.uint8).nbytes

# размер необработанного маленького изображения
  ➥ ~ 262 Кбайт, что в 37 раз больше, чем размер
  ➥ приведенного ранее изображения
# 1 млн таких изображений занял бы 262 Гбайт —
  ➥ слишком много для стандартных компьютеров
np.ones((256, 256, 1), dtype=np.float32).nbytes
```

При запуске реального алгоритма глубокого RL потребление памяти может больше, чем для хранения данных в виде набора состояний, действий, вознаграждений и done, взятых по отдельности. В основанных на полезности алгоритмах в памяти прецедентов нужно хранить следующее состояние для кортежа (s, a, r, s'). При предварительной обработке состояний из игр Atari четыре кадра объединяются. Из этих соображений размер требуемой памяти может увеличиться в несколько раз, притом что в теории необходимый объем необработанных состояний не изменится.

Для эффективного управления памятью стоит приложить дополнительные усилия к тому, чтобы обеспечить хранение минимально необходимого количества данных.

Основная стратегия — хранить мягкие ссылки на переменные и разрешать их, только когда они необходимы для вычислений. Например, можно объединить кадры из исходных состояний, чтобы создать предварительно обработанное состояние большего размера, непосредственно перед передачей в сеть.

Помимо данных, значительным потреблением памяти отличается сама нейронная сеть. Сеть прямого распространения из двух слоев может занимать 100 Мбайт в оперативной памяти, тогда как сверточная сеть — 2 Гбайт. Объект `Tensor` может использовать RAM при накоплении градиентов для autograd, хотя обычно в зависимости от размера пакета он занимает несколько мегабайт. Все это нужно учитывать при запуске сессии обучения.

После обсуждения эффективного хранения информации в оперативной памяти ЦП и GPU рассмотрим перемещение данных из хранилища в нейронную сеть для вычислений.

Поскольку данные передаются с задержкой, обусловленной аппаратным обеспечением, к их созданию и перемещению нужно подходить стратегически. Чем меньше передается данных, тем лучше, поэтому стоит оптимизировать хранилище, как говорилось ранее. Цель — гарантировать, что передача данных не станет узким местом в процессе обучения, как когда ЦП/GPU работает вхолостую, ожидая передачи данных, или когда большая часть процессорного времени расходуется на перемещение и копирование данных.

Часть памяти компьютера, которая расположена ближе всего к процессору и которой мы обычно можем управлять программно, — это RAM. Если загрузка данных в оперативную память эффективная, то главное узкое место при передаче данных устранено.

Перед передачей в нейронную сеть данные нужно преобразовать в объекты `Tensor`, чтобы обеспечить поддержку различных дифференциальных операций, таких как autograd. Если данные уже загружены как `numpy` в оперативную память центрального процессора, то PyTorch просто ссылается на них при создании `Tensor` без дополнительного копирования. Это делает работу с ним чрезвычайно простой и эффективной. Пример операции преобразования `numpy` в `Tensor` приведен в листинге 13.6. Далее он передается из оперативной памяти ЦП непосредственно в ядра для вычислений.

Листинг 13.6. Применение PyTorch данных numpy в оперативной памяти процессора для непосредственного создания Tensor

```
import numpy as np
import torch

# Рекомендации по перемещению данных из хранилища в сеть для вычислений

# Часто данные поступают в высокоточном формате, таком как float64
```

```
 7 # но это не так уж удобно для обучения (например,
   ➥ изображения с высоким разрешением, полученные из игры)
 8 # для хранения пользуйтесь int/float низкой точности
 9 # чтобы снизить потребление оперативной памяти
   ➥ более чем в восемь раз, например, преобразование (float64 в uint8)
10
11 # для хранения подходит понижающее приведение
   ➥ данных в форматах с плавающей запятой к float16
12 state = np.ones((4), dtype=np.float16)
13
14 # для того чтобы хранить изображения в памяти прецедентов
   ➥ и чтобы они помещались в оперативной памяти,
15 # применяйте формат, оптимизированный посредством уменьшения
   ➥ размера и понижающего приведения
16 im = np.ones((84, 84, 1), dtype=np.uint8)
17
18 # непосредственно перед передачей данных в нейронную сеть для расчетов
19 # выполните их приведение к необходимому формату, обычно это float32
20 im_tensor = torch.from_numpy(im.astype(np.float32))
```

Если мы хотим использовать для вычислений графический процессор, то нужно передать тензоры из оперативной памяти ЦП в оперативную память GPU. На копирование и создание данных понадобится дополнительное время. Для крупных сетей накладные расходы на передачу данных восполняются ускорением вычислений на графическом процессоре. Однако для меньших сетей (например, с одним скрытым слоем, состоящим менее чем из 1000 единиц) GPU не дает значительного ускорения по сравнению с ЦП, поэтому обучение может замедлиться из-за издержек передачи данных. Отсюда вытекает практическое правило: не применять графический процессор для маленьких сетей.

Поскольку в глубоком обучении нейронные сети в большинстве своем относительно небольшие, то графический процессор зачастую не задействован полностью. Другое узкое место может возникнуть из-за среды, если данные генерируются медленно. Как вы помните, графические процессоры изначально были созданы для игр, так что большинство игровых движков можно ускорить с их помощью. Эту опцию можно применить и к другим физическим движкам и средам. Таким образом, возможно использование графического процессора для ускорения порождения данных в среде.

Еще одно потенциальное узкое место возникает, когда обучение алгоритма происходит параллельно с помощью множества процессов. При использовании множества узлов, распределенных на большом количестве машин, сообщение между ними может быть медленным. Предпочтительнее запускать обучение на одной мощной машине, поскольку тогда исчезают проблемы коммуникации между процессами.

Кроме того, в рамках одной машины самый быстрый способ совместного использования данных множеством параллельных рабочих процессов — это общая оперативная память. Так устраняется передача данных между процессами. Благодаря

интеграции PyTorch с модулем multiprocessing в Python именно это и происходит при вызове метода сети `share_memory()`, чтобы поделиться параметрами глобальной сети со всеми действующими процессами, как показано в листинге 8.1.

Крайний случай — когда размер данных настолько велик, что их невозможно разместить в оперативной памяти, а можно хранить только в постоянной памяти. Запись на жесткий диск и чтение с него чрезвычайно медленные по сравнению с перемещением данных внутри оперативной памяти, поэтому здесь нужен разумный подход. Например, можно запланировать загрузку фрагмента данных до того, как он понадобится для вычислений, чтобы процессору не приходилось ждать. Как только отпадает необходимость в одном фрагменте данных, удаляйте его, чтобы освободить оперативную память для следующего фрагмента.

Исходя из сказанного по поводу аппаратного обеспечения нас интересует, где производить вычисления (на ЦП или GPU), где размещать данные (оперативная память), как много данных помещается в RAM и как избежать узких мест при порождении и передаче данных.

13.4. Выбор аппаратного обеспечения

Пользуясь полученными практическими сведениями о применяемых данных, мы можем составить требования к аппаратному обеспечению для глубокого RL.

Все не основанные на изображениях среды, обсуждаемые в этой книге, можно запускать на ноутбуке. Только использующие изображения среды, такие как игры Atari, выигрывают от применения графического процессора. Для них мы рекомендуем один графический процессор и по крайней мере четыре вычислительных ядра. На этом оборудовании можно запустить для Atari одно испытание, состоящее из четырех сессий.

Стандартные технические требования для настольных компьютеров — это графический процессор GTX 1080, четыре ядра с частотой выше 3,0 ГГц и 32 Гбайт оперативной памяти. У Тима Деттмерса есть превосходное руководство по сборке настольных компьютеров для глубокого обучения, доступное по ссылке https://timdettmers.com/2018/12/16/deep-learning-hardware-guide [33]. Другой вариант — аренда удаленного сервера в облаке. Для начала хорошо подойдет сервер с графическим процессором и четырьмя ядрами.

Впрочем, для более масштабных экспериментов желательно располагать большей вычислительной мощностью. Для настольных компьютеров это обычно подразумевает увеличение количества ядер. Вариант в облаке — аренда сервера с 32 ядрами и 8 графическими процессорами. В качестве альтернативы рассмотрите сервер с 64 ядрами и без графического процессора. Как мы видели в главе 8, некоторые алгоритмы, такие как A2C, могут выполняться параллельно на множестве ядер, чтобы компенсировать отсутствие графического процессора.

13.5. Резюме

В глубоком RL полезно иметь возможность оценить требования алгоритма к памяти и объему вычислений. В этой главе мы получили представление о размерах основных типов данных, встречающихся в RL, таких как состояния в виде векторов и изображений. А также осветили причины роста потребления памяти алгоритмом.

Помимо этого мы вкратце рассмотрели разные виды процессоров — ЦП и GPU. Графические процессоры чаще всего задействуются для ускорения обучения в средах, использующих изображения.

Приведенные в этой главе рекомендации призваны помочь оптимизировать потребление памяти алгоритмами глубокого RL за счет более эффективного использования вычислительных ресурсов.

Часть IV
Проектирование сред

14 Состояния

Решение новой задачи в глубоком RL подразумевает создание среды. Поэтому теперь мы перейдем от алгоритмов к компонентам структуры среды, которыми являются состояния, действия, вознаграждения и функция переходов. При проектировании среды мы сначала моделируем задачу, а затем решаем, какую информацию и как среда должна предоставлять пользователям.

В RL крайне важно, чтобы среда предоставляла достаточно информации для решения задачи RL алгоритмом. И это одна из ключевых функций состояний, которые являются предметом этой главы.

Сначала приведем несколько примеров состояний как из реального мира, так и из сред RL. В следующих разделах мы рассмотрим важные для проектирования вопросы.

- **Полнота.** Содержит ли представление состояния достаточно информации о внешнем мире для решения данной задачи?
- **Сложность.** Насколько эффективно представление и насколько оно вычислительно затратно?
- **Потери информации.** Теряется ли в представлении какая-нибудь информация, например, при преобразовании изображения в градации серого или при уменьшении его размерности?

В последнем разделе будут рассмотрены некоторые из часто применяемых приемов предварительной обработки состояний.

14.1. Примеры состояний

Состояние — это информация, описывающая среду. Оно также может рассматриваться как нечто наблюдаемое, то есть набор параметров, поддающихся измерению[1]. Состояния — это больше чем просто числа, представляющие игру или модель, запущенную на компьютере. Аналогично среда — это больше чем просто

[1] Такое определение наблюдаемого объекта широко используется в физике и играет основную роль во многих теориях, включая квантовую механику.

компьютерная модель для задачи RL, среды включают системы из реального мира. Рассмотрим несколько примеров состояний.

Все, что мы видим, слышим и ощущаем, — это состояния окружающей среды. Это информация, которую мы можем *«измерить»* с помощью нашего восприятия посредством органов чувств, таких как глаза, уши, кожа. Наряду с необработанной чувственной информацией состояния могут содержать абстрактные сведения, такие как скорость движущегося объекта.

Также есть информация, которую мы можем выявлять косвенно, с помощью инструментов — например, магнитные поля, инфракрасное излучение, ультразвук и недавно обнаруженные гравитационные волны[1]. Животные, восприятие которых отличается от нашего, ощущают свое окружение иначе. Например, собаки не различают красный и зеленый цвета, но их обоняние намного лучше человеческого.

Все эти примеры являются состояниями — информацией о среде, измеренной с помощью различных инструментов восприятия, либо биологических, либо механических. Одна и та же среда может выдавать разные сведения в зависимости от контекста и того, что измеряется, то есть информация может меняться в соответствии с точкой зрения индивида. Нужно учитывать эту принципиально важную идею и руководствоваться ею при проектировании систем обработки информации — не только в RL, но и в повседневной жизни.

Среды из реального мира содержат много сложной информации. К счастью, с этой сложностью можно бороться, что и является темой данной главы. Но пока рассмотрим простые состояния. В видеоиграх, таких как классические игры Atari (рис. 14.1), состояния наблюдаются через видео- и звуковой потоки — графику игры на экране и звук из колонок. У роботов, как физических, так и моделируемых, состояния более высокого уровня, например углы поворота, скорость и крутящий момент сустава. Состояния в RL стараются упростить, чтобы исключить бесполезный фоновый шум и побочные эффекты и сконцентрироваться только на информации, которую проектировщик считает значимой. Например, моделью робота могут не учитываться трение, сопротивление воздуха, тепловое расширение и т. д.

Важно отличать состояния от действий и вознаграждений. Состояние — это информация о среде, даже если не было предпринято никакого действия и не получено никакого вознаграждения. Действие — это влияние, оказываемое на среду сущностью, которая средой не является, а является агентом. Вознаграждение — это форма метаинформации о переходе состояний, вызванном приложенным действием.

---

[1] Четырнадцатого сентября 2015 года LIGO (под управлением Калифорнийского и Массачусетского технологических институтов) впервые обнаружила гравитационные волны [80]. Это действительно значимое достижение в науке.

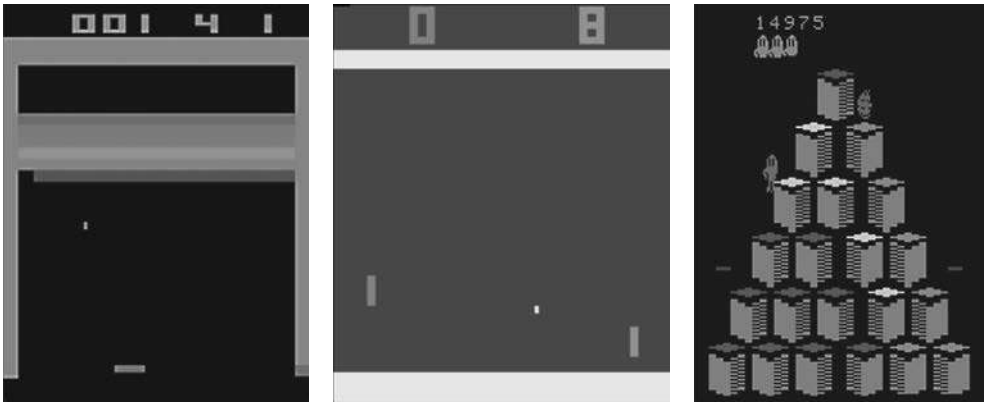


Рис. 14.1. Примеры игр Atari с цветными изображениями в формате RGB (тензоры ранга 3) в качестве состояний. Они доступны как часть предлагаемой OpenAI Gym [18] среды для обучения по аркадным играм (Arcade Learning Environment, ALE) [14]

Состояние может быть представлено с помощью любой подходящей структуры данных, например скалярной величины, вектора или порождающего тензора[1]. Если оно не является численным изначально, его всегда можно закодировать. Например, в естественном языке слова или символы в численном виде можно получить с помощью векторных представлений слов. Также информацию можно биективно отобразить[2] в список целых чисел. Например, 88 клавишам пианино можно присвоить уникальные метки с помощью целых чисел [1, 2... 88]. Состояние может иметь дискретные или непрерывные значения или их сочетания. Например, чайник может показывать, что он включен (дискретное состояние), и предоставлять свою текущую температуру (непрерывное состояние). Проектировщик по своему усмотрению использует удобное для алгоритма представление информации.

Состояние s — это элемент пространства состояний S, полностью задающего все значения переменных, которыми характеризуется состояние среды. В состоянии может быть больше одного элемента (измерения), каждый из которых может быть как дискретным, так и непрерывным и обладать любой мощностью[3]. Удобно представлять состояние с помощью тензоров с данными одного типа (целого или

---

[1] Тензор — это обобщенная информационная структура с N измерениями. Скалярная величина (одиночное число) — тензор ранга 0, у вектора (списка чисел) ранг 1, у матрицы (таблицы чисел) ранг 2, у куба ранг 3 и т. д.

[2] Биективное отображение — это математический термин для отображения один к одному элементов двух массивов, то есть установления взаимно однозначного соответствия между массивами. В своих приложениях мы обычно биективно отображаем массив в список целых чисел, начинающийся с 0.

[3] Мощность является мерой «величины» массива и используется для различения между дискретными и непрерывными массивами.

с плавающей запятой), поскольку этого ожидают большинство библиотек для вычислений, таких как `numpy`, PyTorch или TensorFlow.

Ранг и структура состояния могут быть любыми[1]:

- скалярная величина (тензор ранга 0) — температура;
- вектор (тензор ранга 1) — [позиция, скорость, угол, угловая скорость];
- матрица (тензор ранга 2) — пикселы в градациях серого из игры Atari;
- куб данных (тензор ранга 3) — цветные пикселы в формате RGB из игры Atari (см. примеры на рис. 14.1).

Помимо этого, состояние может быть комбинацией тензоров. Например, компьютерная модель робота может предоставлять поле зрения робота как изображение в формате RGB (тензор ранга 3), а углы его суставов — как отдельный вектор. Для обработки подаваемых на вход разных по структуре тензоров потребуется особая архитектура нейронной сети, в данном случае отдельные тензоры рассматриваются как подсостояния, которые вместе образуют общее состояние.

Получив общее описание состояний, рассмотрим рабочий процесс проектирования состояний, показанный на рис. 14.2. Процесс начинается с принятия решения о том, какую информацию о внешнем мире и в какой форме включить в состояние. Например, электромагнитный спектр может быть представлен разными способами: данными с радара или лидара, изображениями в формате RGB, картой глубин, тепловыми изображениями и т. д. В конечном счете выбранная информация может быть описана как *необработанное (исходное) состояние*.

Мы можем использовать знания о задаче для упрощения состояния, представляя полезную с нашей точки зрения информацию в более явном виде — например, уменьшая размеры изображений и преобразуя его в градации серого. Такое состояние может быть описано как *синтезированное состояние* среды.

Агент может взять как исходное, так и синтезированное состояние среды и дополнительно обработать его для собственных нужд. Например, ряд изображений в градациях серого может быть совмещен для получения одного входного значения. Это известно как *предварительно обработанное состояние*.

Возьмем в качестве практического примера игру Pong из Atari с рис. 14.1. Это двумерная игра в настольный теннис. В ней есть две ракетки и один шарик, ракеткой слева управляет среда, а той, что справа, — игрок. Тот, кто заставляет оппонента пропустить шарик мимо ракетки, зарабатывает очко, и игра заканчивается после 21 раунда. Чтобы выиграть, агенту нужно уметь определять положение шарика и обеих ракеток.

---

[1] Снова терминология: ранг, грубо говоря, — это количество измерений тензора, тогда как структура представляет собой размеры этих измерений. Например, структурой вектора из десяти элементов будет (10), структура матрицы 2×4 — (2, 4).

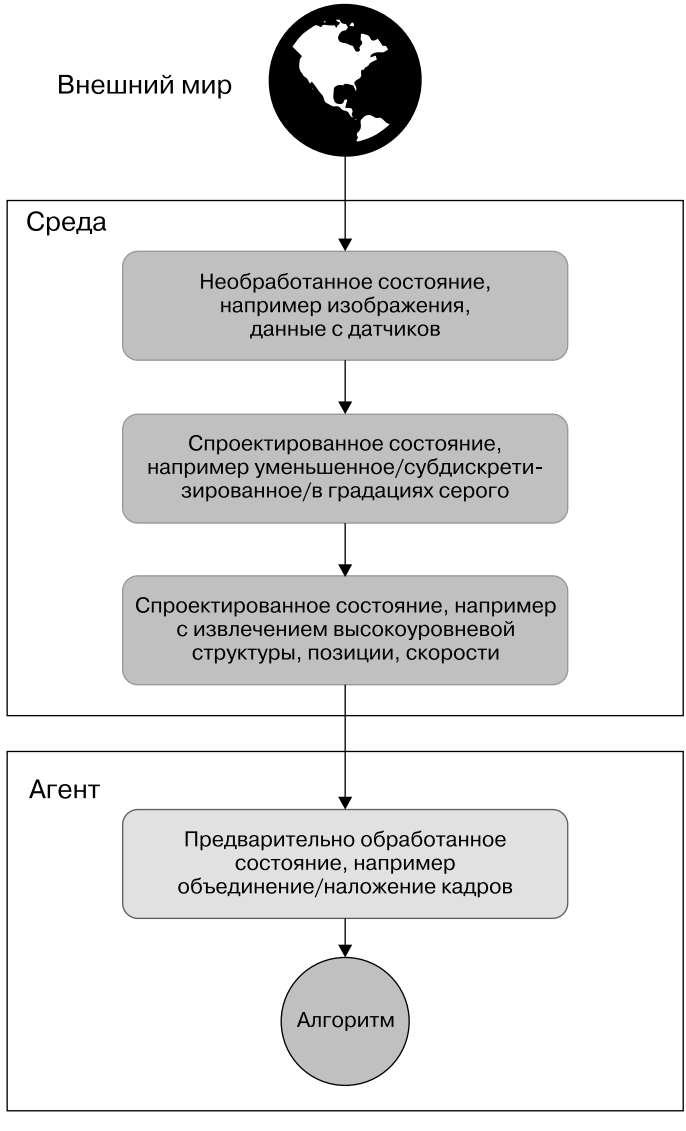


Рис. 14.2. Поток информации из внешнего мира на вход алгоритма

В Pong *необработанное состояние* — это получаемое на конкретном временном шаге с экрана компьютера цветное изображение в формате RGB, которое содержит полную, неотфильтрованную информацию об игре. Агенту, чтобы воспользоваться необработанным состоянием, пришлось бы научиться определять, какими шаблонами в пикселах представлены шарик и ракетки, и определять их движения по пикселам. Только тогда он смог бы начать вырабатывать выигрышную стратегию — по крайней мере, так поступил бы человек.

Однако нам уже известно, что полезная информация включает позиции и скорости данных объектов. Мы можем с помощью этого знания создать *синтезированное состояние*. Один из кандидатов на такое состояние — вектор, в котором числами представлены позиции и скорости обеих ракеток и шарика[1]. Это состояние содержит гораздо меньше информации, которую можно обрабатывать, чем *исходное состояние*, но алгоритму намного проще учиться на его недвусмысленном содержимом.

Рассмотрим другой пример — среду CartPole из OpenAI Gym (рис. 14.3). Ей доступны и необработанное изображение, и синтезированное состояние, как показано в листинге 14.1. Синтезированное состояние — это то, что непосредственно возвращается методами `env.reset` (строка 15) и `env.step` (строка 37) API среды. Необработанное состояние в виде изображения в формате RGB может быть получено с помощью функции `env.render(mode='rgb_array')` (строка 21).

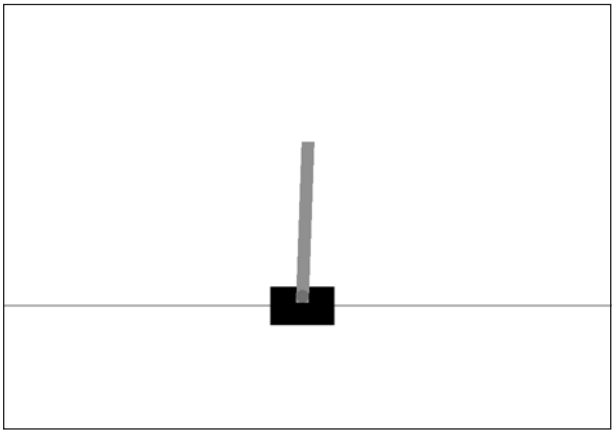


Рис. 14.3. CartPole-v0 — самая простая среда из OpenAI Gym. Цель — удерживать стержень в равновесии в течение 200 временных шагов, управляя перемещениями тележки влево-вправо

Если заменить обычное состояние необработанным в виде изображения в формате RGB, любому алгоритму будет намного труднее решить задачу, которая иначе была бы простой. Это вызвано тем, что теперь агенту приходится выполнять гораздо больше работы. Он должен догадаться, на какой набор пикселов нужно обращать внимание, выяснить, как идентифицировать объекты тележки и стержня на изображении, определить параметры движения объектов из изменений наборов пикселов между изображениями и т. д. Агент должен из всех значений пикселов отфильтровать шумы, выделить полезные сигналы, сформировать высокоуровневые понятия и затем восстановить информацию, аналогичную той, которая предоставляется синтезированным состоянием в виде вектора из позиций и скоростей. Только после этого он может сообщить соответствующие признаки своей стратегии для решения задачи.

[1] Это гипотетический пример синтезированного состояния; игровой средой предлагается лишь необработанное состояние в виде изображения.

Листинг 14.1. Пример кода для получения необработанного и синтезированного состояний из среды CartPole-v0

```
# отрывок кода для исследования среды CartPole
import gym

# инициализировать среду и проверить ее пространства состояний и действий
env = gym.make('CartPole-v0')
print(env.observation_space)
# => Box(4,)
print(env.action_space)
# => Discrete(2)
# максимальное число шагов T для нормального завершения среды
print(env.spec.max_episode_steps)
# => 200

# присвоить переменным среды начальные значения и проверить ее состояния
state = env.reset()
# пример состояния: [позиция, скорость, угол, угловая скорость]
print(state)
# => [0.04160531 0.00446476 0.02865677 0.00944443]

# получить тензор изображения из визуализатора,
➥ указав режим mode='rgb_array'
im_state = env.render(mode='rgb_array')

# трехмерный тензор, который является изображением в формате RGB
print(im_state)
# => [[[255 255 255]
# [255 255 255]
# [255 255 255]
# ...

# структура тензора изображения – (высота, ширина, канал)
print(im_state.shape)
# => (800, 1200, 3)

done = False
while not done:
    rand_action = env.action_space.sample() # случайное действие
    state, reward, done, _info = env.step(rand_action)
    print(state) # проверить, как изменилось состояние
    im_state = env.render(mode='rgb_array')
```

Необработанное состояние должно содержать всю относящуюся к задаче информацию, но по нему намного труднее обучаться. В нем зачастую много лишних сведений и фоновых шумов, поэтому за извлечение и контекстуализацию этой информации в удобной форме неизбежно приходится платить ростом затрат на обработку. Полная, необработанная информация дает бóльшую свободу, но увеличивает нагрузку на извлечение из нее полезных признаков и их интерпретацию.

Синтезированное состояние обычно содержит выделенные сигналы, которые гораздо удобнее для обучения. Но есть риск потери важной информации — более подробно это обсуждается в разделе 14.4. Среды могут предоставлять как необработанные, так и синтезированные состояния. Какое из них выбрать, зависит от цели. С помощью более сложных необработанных состояний можно проверить устойчивость и пределы применимости алгоритма. Более простые синтезированные состояния можно использовать для непосредственного решения задачи.

Исследовательские среды тяготеют к более простым состояниям, с которыми можно быстро выполнять вычисления. Отчасти это вызвано тем, что для обучения безмодельных методов глубокого RL требуется много прецедентов, поэтому что-то более сложное затруднило бы обучение. Это будет оставаться узким местом до тех пор, пока не произойдет существенного повышения эффективности выборок. В средах классического и робототехнического управления из OpenAI Gym [18] и MuJoCo [136] относительно низкоразмерные синтезированные состояния включают позиции, скорости и углы суставов. Даже основанные на изображениях игровые среды из Atari дают картинки, низкого разрешения которых достаточно для решения задачи.

Современные видеоигры зачастую довольно сложны и реалистичны и благодаря продуманным игровым движкам очень хорошо имитируют реальный мир. Состояния для этих игр могут быть отнесены к способам восприятия реального мира, таким как камера с видом от первого лица и объемный звук из компьютерной игры. Шлемы виртуальной реальности обеспечивают высокоточное стереоскопическое (трехмерное) зрение, а их консоли предоставляют обратную связь, позволяющую испытывать тактильные ощущения и вибрации. В аркадных шутерах с помощью подвижных сидений имитируются даже физические движения[1].

Эти типы сложной перцептивной информации все еще редко применяются в исследованиях ИИ. Тому есть несколько причин: существенные требования к хранилищам данных и вычислениям, а также то, что разработка платформ для сред все еще находится на начальной стадии. К счастью, за последние несколько лет в качестве надстроек над мощными игровыми движками появились более реалистичные среды RL. В их число входит Unity ML-Agents [59], включающая несколько модельных сред, надстроенных над игровым движком Unity. Кроме того, это Deepdrive 2.0 [115] (на основе Unreal Engine) с обеспечением реалистической модели управления автомобилем. А также Holodeck [46] (на основе Unreal Engine) с высокоточным моделированием виртуальных миров. Это открывает совершенно новый игровой плацдарм для еще более сложных заданий в RL.

Сложность пространства состояний соответствует сложности задачи, последняя зависит также от того, насколько хорошо спроектированы состояния. Структура

[1] Стоит побывать в зале с игровыми автоматами, чтобы испытать новые ощущения (и хорошенько повеселиться).

состояния может как обеспечить решение задачи среды RL, так и сделать ее неразрешимой.

Объекты, с которыми мы часто взаимодействуем, служат основой для проектирования состояния. Многие из них передают визуальную информацию, ведь у человека зрение — основной канал восприятия. Температура и вес измеряются по градуированным шкалам, состояние электрического прибора показывают его световые индикаторы, давление и скорость отображаются измерительными устройствами, за игрой следят по табло. Есть и невизуальные каналы, такие как звуковой сигнал автомобиля или вибрации телефона в беззвучном режиме. Это примеры того, как можно измерить и представить информацию о системе. Живя в информационную эпоху, мы так привыкли все измерять, что идеи для проектирования состояний можно найти повсюду.

Хорошая структура среды повышает скорость исследований и разработки приложений. Но ее нужно всесторонне тестировать, чтобы пользователям не приходилось с ней возиться и исправлять проблемы, не связанные с их работой. В основном OpenAI Gym был создан для того, чтобы предоставлять обширный набор стандартных надежных сред с тщательно спроектированными состояниями, действиями и вознаграждениями. Хороший дизайн вкупе с простотой применения и простой лицензией позволили OpenAI Gym стать испытательной лабораторией для исследователей и тем самым внести вклад в развитие области. С момента своего создания он стал одним из стандартов де-факто для сред в глубоком RL.

Несмотря на важность проектирования состояний в глубоком RL, формальных или исчерпывающих руководств по нему мало. Тем не менее приобретение достаточных навыков в проектировании состояний — или по крайней мере понимание этого процесса — дополняет основные знания алгоритмов RL. Без этого вам не решить новых задач.

14.2. Полнота состояния

Поговорим о структуре необработанного состояния. Самый важный вопрос: содержит ли необработанное состояние достаточно информации для решения задачи?

Основное правило: подумайте о том, какая информация необходима для решения задачи. Затем рассмотрите доступность этой информации во внешнем мире. Если информация полная, то задача *полностью наблюдаемая*, например, игру в шахматы можно полностью представить позициями всех фигур на доске. Задача с неполной информацией является *частично наблюдаемой*, пример — покер, где игрок не может видеть карты других игроков.

Состояние, которое содержит полную информацию, идеальное, но оно не всегда возможно из-за теоретических или практических ограничений. Иногда состояние,

полностью наблюдаемое в теории, не является таковым на практике из-за шумов, неидеальных условий или других неучтенных факторов. Например, при использовании роботов в реальном мире сигналу требуется время на прохождение от компьютера к двигателям, поэтому для высокоточного управления нужно учитывать подобные эффекты.

В частично наблюдаемом состоянии влияние неполной информации может проявиться в разных ситуациях. Агент может уметь компенсировать шумы и задержки в среде, если они не слишком большие. Но если значение задержки слишком велико, играть в онлайн-видеоигры, для победы в которых решения нужно принимать за доли секунды, становится невозможно.

Далее при обработке исходного состояния нужно рассмотреть вторичные признаки.

1. Какие типы данных используются: дискретные или непрерывные, плотные или разреженные? Этим определяется подходящий формат представления данных.
2. Какова мощность пространства состояний? Будет ли порождение состояний малозатратным с точки зрения вычислений? От этого зависит возможность или, наоборот, невозможность получения необходимого для обучения объема данных.
3. Сколько данных нужно для решения задачи? Одно из правил: нужно оценить, сколько данных понадобилось бы для решения этой задачи человеку, затем умножить это количество на число от 1000 до 100 000. Это лишь приблизительная оценка, реальное количество определяется характером задачи, структурой состояния и эффективностью применяемого алгоритма. Например, человеку может понадобиться от 10 до 100 эпизодов, чтобы научиться хорошо играть в игру из Atari. Алгоритму, как правило, требуется более 10 млн кадров (около 10 000 эпизодов), чтобы играть на том же уровне.

Для ответа на эти вопросы нужны экспертные знания предметной области и понимание сути задачи, особенно если до сих пор ее никто не решал. Оценивать эти моменты следует сквозь призму сложности результирующего состояния.

14.3. Сложность состояния

Состояния представлены структурами данных, и, как при всяком проектировании структур данных, нам нужно рассмотреть их вычислительную сложность. Эта сложность может проявляться в двух формах: *разрешимости* и *эффективности представления признаков*. Далее приводятся несколько рекомендаций по проектированию эффективных представлений состояний, причем они применимы как к необработанным, так и к синтезированным состояниям.

Разрешимость[1] напрямую связана с мощностью пространства состояний. Каков размер каждого из примеров данных? Сколько примеров нужно, чтобы получить хорошее представление всей задачи и решить ее? Как вы помните, кадр соотносится со всеми данными, порождаемыми за временной шаг. Мы видели, что алгоритмам RL для хорошей производительности в играх Atari обычно требуются миллионы кадров. Типичный кадр уменьшенного изображения в градациях серого занимает 7 Кбайт, тогда 10 млн кадров соответствует 10 000 000 · 7 Кбайт, что дает 7 Гбайт. Насколько высоким будет потребление памяти (RAM), зависит от того, сколько кадров агент сохраняет за раз. Для сравнения рассмотрим современную игру с высоким разрешением, размер каждого изображения которой — 1 Мбайт. Обработка 10 млн кадров равносильна вычислениям над 10 Тбайт данных. Вот потому-то понижение размерности изображений — хорошая идея[2].

Точно воспроизводящая задачу (то есть использующая высококачественные изображения из игры) модель среды может порождать необработанные данные в таких количествах, что вычисления станут трудновыполнимыми на практике. Потому необработанные состояния зачастую следует сжать и преобразовать в надлежащим образом отобранные признаки, значимые для задачи. Проектирование признаков — чрезвычайно важная часть решения задачи, особенно новой. Не все признаки равноценны, поэтому нужно рассмотреть их эффективность.

Пусть существует множество представлений возможных признаков, тогда возникает несколько вопросов. Что произойдет при запуске с ними одного и того же алгоритма? Будет ли хранилище большим? Не станет ли решение непрактичным из-за высокой вычислительной сложности? Каковы вычислительные затраты на извлечение признаков из необработанного состояния? Мы ищем минимально достаточное представление признака, которое делает вычисления разрешимыми и не порождает слишком много данных, которые нужно обрабатывать.

Первая хорошая стратегия — это сжатие необработанного состояния. Сложность пропорциональна размеру данных: чем больше битов должен обработать алгоритм, тем больше пространства и времени ему потребуется. Основанные на изображениях игры — хороший пример. При кодировании игры в высоком разрешении сохраняется вся первоначальная информация. Но чтобы играть эффективно, редко нужны изображения с высоким разрешением, которые затратно производить и обрабатывать.

Если для обработки входного изображения применяется сверточная сеть, количество пикселов будет каскадироваться вниз по всем слоям и число вычислений возрастет. Более рационально воспользоваться стандартными приемами сжатия

---

[1] В информатике разрешимой считается задача, которая может быть решена за полиномиальное время.

[2] Методы предварительной обработки обсуждаются в разделе 14.5, расчет потребляемой памяти приводится в главе 13.

изображений. Размер уменьшится до приемлемого значения — скажем, 84×84 пикселов, а вычисления пропорционально ускорятся. Даже на современном мощном аппаратном обеспечении уменьшение размера изображений может сыграть решающую роль в том, чтобы модель поместилась в памяти и, следовательно, сессия обучения сократилась с месяцев до дней.

Сложность растет также по причине комбинаторного взрыва после каждого добавления дополнительного измерения. Даже когда измерения представляют фундаментально различные типы информации, количество измерений все же можно уменьшить с помощью разумного проектирования. Снова обратимся к видеоиграм. Вместо использования полноцветного трехканального изображения в формате RGB мы можем преобразовать его в оттенки серого с одним каналом. Однако это применимо только тогда, когда цвет не имеет большого значения. Если бы фильм «Матрица» был черно-белым, Нео было бы непросто отличить красную таблетку от синей. При преобразовании изображения в оттенки серого происходит уменьшение трехмерного объемного массива пикселов (с цветовыми каналами) до двумерной матрицы пикселов в градациях серого (без цветовых каналов) и сложность снижается по закону кубического корня. Поэтому в играх Atari, помимо уменьшения размера состояний, обычно выполняется их преобразование из цветных изображений в градации серого.

Другая стратегия сжатия информации — проектирование признаков. Вспомните разницу между *необработанным* и *синтезированным* состояниями. Проектирование признаков — это процесс преобразования первого во второе, то есть переход от низкоуровневой необработанной информации к высокоуровневым представлениям. Конечно же, мы ограничены информацией, доступной в исходном состоянии. Возьмем в качестве примера CartPole. Вместо изображения среда может быть лаконично представлена всего четырьмя числами: для тележки — положением и скоростью вдоль оси X, для стержня — углом наклона и угловой скоростью. К тому же такое состояние намного более информативно, так как алгоритму не нужно выполнять дополнительную работу, чтобы из необработанного изображения уяснить понятия позиций и скоростей.

Проектировщик состояний может по своему усмотрению вручную закодировать полезную информацию. В зависимости от его выбора новое кодирование может как упростить, так и усложнить задачу. Пусть мы не используем для обучения состояние в виде изображения из Pong из Atari. Тогда какую содержательную высокоуровневую информацию можно извлечь из необработанного изображения, чтобы сформировать лаконичное представление игры по аналогии с примером для CartPole из предыдущего абзаца? Можно предположить, что такое синтезированное состояние будет включать позиции, скорости и ускорения[1] шарика и ракеток агента и его оппонента. Если не учесть информацию о ракетке оппонента, то

[1] Ускорение необходимо, так как чем дольше шарик остается в игре, тем больше его скорость.

игра усложнится, поскольку агент не сможет исследовать поведение противника и предупреждать его действия. И наоборот, если включить скорости шарика и обеих ракеток, задача упростится, так как агенту не нужно будет учиться оценивать скорости по изменению позиций с течением времени.

Проектированием признаков достигаются два результата: мощность состояния ощутимо уменьшается, а его сложность значительно снижается. При выборе вручную из необработанного состояния только подходящих признаков исключается ненужная информация и остается более компактное синтезированное состояние с меньшим количеством измерений. Помимо этого, синтезированное состояние может содержать информацию, которую можно только вывести из необработанных данных, но не наблюдать непосредственно. Хороший пример — скорости объектов. Когда эти признаки представлены в синтезированном состоянии напрямую, агенту не нужно учиться извлекать их из необработанного состояния. По сути, большая часть работы по пониманию задачи уже выполнена за агента. Получив синтезированное состояние, агенту нужно лишь воспользоваться им и сосредоточиться на непосредственном решении основной задачи.

Один из недостатков формирования признаков в том, что в ходе него полагаются на людей, которые, исходя из своих знаний о задаче, определяют значимую информацию в необработанных данных, чтобы спроектировать эффективное и целесообразное представление признаков. В этом процессе, по сравнению с необработанными данными, неизбежно кодируется больше человеческих приоритетов по той причине, что мы используем свои знания и эвристики для того, чтобы отобрать то, что считаем пригодным. Агент, который учится по такому состоянию, определенно не делает это с нуля по-настоящему, но если наша главная забота — решение задачи, то не о чем беспокоиться. В результате проектирования признаков появляются состояния, содержимое которых построено корректно с точки зрения человека, и это повышает их интерпретируемость. Правда, кто-то может возразить, что по той же самой причине такие состояния станут менее обобщенными или с большей вероятностью будут включать человеческие ошибки.

Хоть проектирование признаков и желательно, но на практике оно не всегда возможно. Например, если задание — это распознание изображений или визуальный переход по страницам, то нецелесообразно вручную создавать состояния, учитывающие все значимые признаки на изображении, не говоря уже о том, что это может противоречить цели. Как бы то ни было, даже несмотря на чрезвычайную сложность, порой это стоит приложенных усилий. Очень сложные игры, такие как Dota 2[1], служат примером того, что, когда для проектирования состояний тре-

[1] Dota 2 — популярная игра от eSports, в которой две команды по пять игроков, выбрав героев более чем из 100 персонажей, сражаются за вражеский трон. Обширная карта и умопомрачительное число комбинаций персонажей и навыков делают эту игру чрезвычайно сложной.

буются колоссальные усилия, применение необработанных изображений сильно затрудняет решение задачи.

Чтобы играть в Dota 2, OpenAI спроектировали огромное игровое состояние, содержащее ошеломляющее количество полученных из API игры элементов — 20 000. Состояние включает связанную с картой информацию, такую как ландшафт и время между возрождением героев. Состояние также содержит для каждого из героев и других юнитов набор сведений, таких как атака, здоровье, позиция, способности и предметы. Это гигантское состояние обрабатывается по отдельности, затем объединяется перед передачей в сеть LSTM из 1024 элементов. За более подробной информацией об архитектуре модели обращайтесь к разделу «Структура модели» в посте об OpenAI Five [104]. Объединив значительные инженерные усилия с вычислительной мощностью, в 2019 году они смогли победить лучших в мире игроков [107].

Таким образом, проектирование признаков возможно и приветствуется, особенно когда важные элементы игры известны и могут быть довольно просто описаны. Вероятны проблемы с тем, какие из очевидных элементов включать в синтезированное состояние. В играх нас обычно интересуют вещи, которые меняются или двигаются. В робототехнике — позиции, углы, силы и крутящие моменты. Для фондовой биржи полезными показателями торгов будут емкость рынка, изменения цен, скользящие средние и т. д.

Не существует четких и определенных правил проектирования эффективных состояний по необработанным данным. Однако никогда не будет лишним потратить время, чтобы понять задачу, включить в состояние знания о предметной области и контекст. Это особо оправданно в практических приложениях.

По сути, проектирование состояний заключается в сжатии информации и устранении шумов, поэтому стоит опробовать все соответствующие стандартные методы. В общем, рассмотрите следующее.

1. **Очевидная информация.** Что-то, что для человека кажется значимым с первого взгляда, например позиции ракетки, мячика и блоков в Breakout, просто потому что это объекты игры.
2. **Контрфактическая информация.** Если x отсутствует и задача не может быть решена, то нужно включить в нее x. Когда в CartPole недоступна скорость объекта, мы не можем предположить, куда он движется, поэтому важно предугадывать движения объектов — это означает, что скорость должна присутствовать.
3. **Инвариантность** (позаимствовано из физики). Что остается постоянным или инвариантным при преобразованиях, то есть проявляет симметрию? Например, CartPole инвариантна к горизонтальным перемещениям, пока они не уходят за экран. Поэтому абсолютное положение тележки вдоль оси X не имеет значения

и может быть исключено. А падение стержня не инвариантно к повороту по вертикали, поскольку сила тяжести действует только в одном направлении. Этот факт отражается углом наклона и угловой скоростью.

4. **Изменения** (позаимствовано из физики). Обычно нас интересует то, что меняется. Изменилась ли позиция мяча? Цена выросла или упала? Передвинулась ли роборука? Эти изменения могут быть значимыми, притом что другие — фоновые шумы — нет. Например, автопилотируемому автомобилю не нужно замечать летящие по ветру листья. Возникает заманчивая мысль, что агент должен беспокоиться лишь об изменениях, которые могут вызвать его действия. Тем не менее существуют важные исключительно наблюдаемые изменения, такие как сигнал светофора, которым мы не можем управлять, но которому должны подчиняться. В целом основная идея такова: включать в состояние значимые вещи, которые изменяются и влияют на достижение цели. В то же время признак, который агент не контролирует и который не вносит вклад в достижение цели, можно исключить.

5. **Переменные, зависящие от времени.** Многие, но не все изменения происходят во времени, а не в пространстве. Тем не менее по информации из одного кадра (временного шага) невозможно определить никакое изменение во времени. Если ожидается, что значение x будет меняться со временем, то в состояние следует включить $\frac{dx}{dt}$. Например, в CartPole имеет значение изменения позиции, следовательно, скорость важна.

6. **Причина и следствие.** Если цель или задание являются следствием других причин (а обычно это так), то должны быть включены все элементы причинно-следственной цепи.

7. **Устоявшиеся статистические или количественные показатели.** Здравый смысл подсказывает, что при техническом анализе торгов на бирже следует рассматривать такие показатели, как емкость и скользящие средние. Целесообразнее включить устоявшиеся значения, чем вводить собственные по необработанным ценовым данным с фондовой биржи. Не нужно изобретать колесо.

Наконец, нередко полезно позаимствовать приемы из областей с родственными или аналогичными задачами и типами данных. Для изображений используйте методы уменьшения размера, преобразования в градации серого, свертки и совмещения из компьютерного зрения, которые способствовали продвижению RL. Например, все современные результаты в играх Atari достигнуты с помощью сверточных нейронных сетей. Если данные разреженные, присмотритесь к представлениям слов из обработки естественного языка (natural language processing, NLP). При последовательно передаваемых данных снова обратитесь к NLP и примените рекуррентные сети. Не стесняясь, привлекайте любые приемы из широкого спектра областей, таких как теория игр и старый добрый искусственный интеллект (GOFAI) для игр, классическая механика для робототехники, классическое управление для управляющих систем и т. д.

14.4. Потеря информации о состоянии

Проектирование состояния начинается с набора доступных сведений из необработанного состояния. Затем для получения желаемого конечного набора сведений в форме синтезированного состояния могут быть применены различные методы сжатия. При этом часть информации может быть потеряна, в противном случае процесс называется сжатием без потерь.

В этом разделе мы рассмотрим распространенную ошибку при проектировании состояний по необработанным состояниям — вероятность *потери* важной информации. Ответственность за то, чтобы этого не происходило, лежит на проектировщике. Мы рассмотрим много практических примеров, включая преобразование изображения в градации серого, дискретизацию, коллизии хеширования и потери метаинформации, возникающие из-за некорректности представления.

14.4.1. Преобразование изображений в градации серого

При уменьшении размера изображения проектировщики должны визуально проверить, не стало ли оно слишком размытым — не стало ли его разрешение слишком низким, — чтобы его можно было использовать в дальнейшем. Более того, в видеоиграх некоторые элементы закодированы цветом, чтобы людям было проще играть. При сжатии цветных изображений в формате RGB до градаций серого проследите за тем, чтобы при этих изменениях не пропали ключевые элементы. Это может произойти, когда возникают коллизии хеширования при отображении большего массива чисел (трехмерного тензора, изображения в формате RGB с тремя каналами) в меньший массив (двумерный тензор, изображение в градациях серого). Тогда значения яркости разных цветов могут оказаться практически одинаковыми.

На рис. 14.4 приведен пример потери информации при преобразовании в градации серого. В OpenAI обучали агента в игре Seaquest из Atari [103], где игрок получает вознаграждения за управление подводной лодкой. На предварительно обработанном изображении с зеленым и синим цветами были сопоставлены очень близкие значения в градациях серого, что сделало их неразличимыми. Из-за этой ошибки акулы противника слились с фоном и стали невидимыми для агента (представление для алгоритма на рис. 14.4). Утрата этой критической части информации негативно сказалась на производительности. К счастью, есть много других схем преобразования в градации серого. Данной проблемы можно избежать, выбрав подходящую схему, которая не вызовет исчезновения акул, как показано на исправленном представлении на рис. 14.4. Не существует абсолютно надежной схемы преобразования в градации серого, поэтому предварительно обработанные изображения всегда нужно проверять вручную.

Рис. 14.4. Преобразование в градации серого может привести к исчезновению ключевых элементов игры, отсюда вытекает хорошее правило: всегда проверять предварительно обработанные изображения при отладке. Изображение взято из OpenAI Baselines: DQN [103]

14.4.2. Дискретизация

Другая распространенная причина потери информации возникает при дискретизации непрерывных значений, то есть отображении непрерывного или бесконечного диапазона значений в массив дискретных конечных значений. Возьмем в качестве примера задание, предполагающее считывание показаний аналоговых часов. В зависимости от того, показания какой стрелки, часовой или минутной, желательно считывать, нужно применять разные схемы дискретизации. Если задание состоит только в считывании часов, есть смысл дискретизировать угол на 12 частей, поскольку на циферблате только 12 значений часов. Если нужно считывать минуты, этого будет недостаточно. В зависимости от нужной точности показания часов должны быть дискретизированы на интервалы по 30, 10 или 1 мин. Любые меньшие значения становятся для человеческого глаза неразличимыми. Фактически деления на часах — это одна из схем дискретизации, помогающая нам считывать время с допустимой степенью точности. Это влияет и на то, как мы называем время. Владелец часов с делениями только по 30 мин на вопрос о том, который час, скорее всего, даст ответ по одной и той же схеме дискретизации, будь то 10:27, 10:28 или 10:29. Этот человек наверняка скажет: «Десять тридцать». Данный пример демонстрирует то, как дискретизация вызывает потерю информации (и тут нечему удивляться).

14.4.3. Конфликты хеширования

Два предыдущих примера — это частные случаи того, что известно как *конфликт хеширования* (hash conflict). Он возникает, когда больший массив данных сжимается до меньшего массива, который не способен представить все различающиеся элементы. Следовательно, коллизии возникают, когда разные элементы изначального массива отображаются в одно и то же значение меньшего массива.

Сжатие информации может привести к конфликтам хеширования, но они не всегда становятся проблемой. Если взглянуть на пример Seaquest из Atari, то все схемы преобразования в градации серого приводят к конфликтам хеширования, но лишь некоторые из них вызывают проблему, связанную с исчезновением акул на рис. 14.4. К несчастью, нет простых способов определения того, какие из конфликтов хеширования неблагоприятны, так как это зависит от задачи. Нужно положиться на проверку примеров данных вручную.

В повседневной жизни мы тоже испытываем конфликты хеширования. Люди используют цвета для передачи семантики, например, красный — опасность, зеленый — безопасность, привычные красно-желто-зеленые лампочки стоят в светофорах по всему миру. Существует технология, позволяющая одной лампочке светиться всеми этими цветами попеременно. И хоть ее применение оправдано экономически и технологически, вождение станет чрезвычайно опасным для людей, не различающих цвета: им будет очень трудно сказать, красный это свет или зеленый. Поэтому в светофорах продолжают устанавливать три отдельные монохромные лампочки.

14.4.4. Потери метаинформации

Некоторая часть информации предоставляет высокоуровневые знания о данных, с которыми мы имеем дело. Это метаинформация — информация об информации. Рассмотрим несколько примеров.

В шахматы играют на двумерной доске, и у игроков не возникает проблем с перемещением фигур во время игры. Отдельные действительно высококлассные игроки могут играть в уме, запоминая позиции фигур и называя вслух ходы, такие как «слон с A2 на D5». Сделать это труднее, чем на реальной доске, так как игрок должен создать мысленный образ позиций на воображаемой доске, но это вполне возможно.

В шахматах координаты отсчитываются в двух направлениях: для горизонтальных позиций используются буквы латинского алфавита, для вертикальных — числа. Для примера перейдем к одномерным координатам, отобразив двумерные координаты в набор из 64 чисел, скажем A1 → 1, B1 → 2... A2 → 9, B2 → 10... H8 → 64. Ход «слон с A2 на D5» теперь станет ходом «слон с 9 на 36».

Представьте себе игру в шахматы в этой одномерной системе. Это намного труднее, поскольку теперь нам нужно преобразовать числа 9 и 36 обратно в A2 и D5, чтобы смочь визуализировать позиции фигур на двумерной доске. Здесь предполагается, что мы играли в шахматы раньше. Пойдем еще дальше и представим, как новичков знакомят с шахматами, не показывая им двумерную доску, а лишь описав все правила с помощью новой одномерной системы координат. Более того, им даже не говорят, что это двумерная игра, так что у них нет представления

об ее пространственном характере. Вне всякого сомнения, новички будут в недоумении.

Является ли это сжатием без потерь? Все двумерные координаты отображены в 64 различных числа — конфликты хеширования отсутствуют. Так почему же играть стало намного трудней? Количественная проверка не покажет нам, что не в порядке, но качественная выявит проблему. Обратив двумерные координаты в одномерные, мы исключили сведения о двумерных связях. Мы проигнорировали тот факт, что игра рассчитана на два измерения и фигуры перемещаются в двумерном пространстве, поэтому правила были грамотно разработаны для этих двух измерений. Например, понятия «шах» и «мат», зависящие от двумерного расположения, станут очень странными в одномерных координатах.

То же самое произошло на заре распознавания цифр по изображениям (например, с помощью набора данных MNIST [73]) еще до того, как стали применяться сверточные сети. Изображение нарезалось на полоски, которые добавлялись друг за другом в вектор, и он передавался на вход MLP. Неудивительно, что сети было нелегко распознавать цифры, учитывая, что она получала одномерную версию изображения и не знала, что оно должно быть двумерным. Эта проблема была решена, когда стали применять двумерные свертки, разработанные для обработки двумерных данных. На рис. 14.5 показан пример цифры 8, взятый из набора данных MNIST [73]. Будучи сжатой до одномерного вектора, она выглядит как на рис. 14.6 (вектор очень длинный, поэтому показан частично), что значительно труднее идентифицировать как 8.

Рис. 14.5. Изображение цифры 8 из MNIST. Это естественное двумерное представление легко идентифицировать

Рис. 14.6. Вот так выглядит рис. 14.5, сжатый до одномерного вектора, при соединении всех строк вместе. Узнать в этом 8 очень сложно

Данный сценарий — пример *потери метаинформации*. Такая потеря не становится очевидной сразу при просмотре информации — нужно рассмотреть ее внешний контекст. Например, коллекция из 64 чисел в шахматах ничего не скажет нам о двумерности данных — этот контекст должен предоставляться извне. Подобная метаинформация существует не только в *пространстве*, но и во *времени*.

Рассмотрим кадр из среды CartPole при $t = 1$ (рис. 14.7, *а*). Предположим, что у нас есть доступ только к изображению, но не к синтезированному состоянию. Глядя на этот отдельно взятый кадр, сможем ли мы сказать, что произойдет при $t = 2, 3$ и т. д.? Это невозможно. Без доступа к состоянию нам неизвестны скорости тележки или стержня. Стержень может стоять неподвижно, падать влево или вправо. Только посмотрев все другие кадры на более поздних временных шагах (см. рис. 14.7, *б–г*), мы узнаем, что стержень падает влево. Это иллюстрирует отсутствие высокоуровневой информации о том, как *сама информация изменяется с течением времени*. Способ восстановления этой информации заключается в том, чтобы собрать последовательные кадры вместе и считать это одним наблюдением.

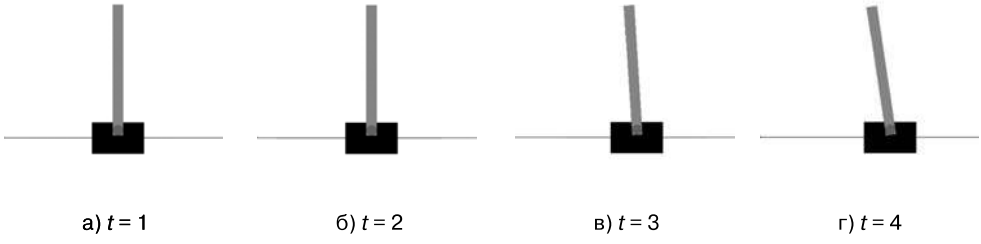


Рис. 14.7. Четыре последовательных кадра для среды CartPole

На рис. 14.7 можно наблюдать последовательность изменений, поскольку снимки были сделаны с рациональными интервалами. Наша способность восприятия этих изменений определяется частотой смены кадров. В высококачественных играх частота больше 60 кадров в секунду, поэтому для игроков видео выглядит непрерывным. Однако при высокой частоте смены кадров изменения между последовательными кадрами малы. Выбирая частоту смены кадров для обучения, нужно обеспечить различимость изменений. Что квалифицируется как различимое, зависит от воспринимающего. Люди менее чувствительны к изменениям, чем машины, так что, если объект на экране сместится на 1 пиксел, люди, скорее всего, этого не заметят. В то же время изменения будут накапливаться и с течением времени станут более различимыми. Отсюда вытекает общепринятая стратегия акцентирования последовательных изменений путем пропуска кадров, то есть выборки одного кадра из каждых k кадров. Для игр с частотой смены кадров 60 оптимальна частота пропуска $k = 4$. Этот метод упрощает обучение для агента, кроме того, он уменьшает общее количество данных, которые нужно хранить.

Если бы нам сейчас сказали, что кадры на рис. 14.7 проигрываются в обратном порядке, то мы пришли бы к заключению, что стержень на самом деле не падает влево, а поднялся и падает вправо. Должна быть дана метаинформация *о порядке следования данных*. Сохранить ее можно с помощью рекуррентной нейронной сети, другой способ — передать последовательность кадров по порядку в MLP или CNN.

Наш последний пример — текст на китайском языке, в котором символы записаны в таблице, а предложение состоит из последовательности символов. Иногда предложения должны читаться слева направо сверху вниз, порой сверху вниз слева направо, а в ряде случаев сверху вниз справа налево. Кроме того, книгу можно читать слева направо или справа налево, правда, это обычно соответствует направлению потока предложений. Если попытаться обучать нейронную сеть по двумерной таблице китайских символов, не зная, в каком направлении нужно читать, то она увидит только бессвязную последовательность отдельных символов. История из книги будет утеряна.

Примеры демонстрируют, как может происходить потеря информации явным и неявным образом. Этот вид потерь может встречаться в самых неожиданных местах. Все потери такого рода можно назвать информационной слепотой — по аналогии

с цветовой слепотой, но в других измерениях и формах информации, включая метаинформацию и внешний контекст.

Таким образом, при сжатии данных проверяйте, не произошла ли потеря информации, с помощью методов, обсуждаемых в этой главе.

- **Случайное исключение** (основная проверка). Проверьте, чтобы ключевая информация не была случайно упущена.
- **Конфликт хеширования** (численная проверка). При отображении большего набора в меньшее пространство проверьте, чтобы не произошло слияния элементов, которые должны оставаться различимыми.
- **Информационная слепота** (качественная проверка). Проверьте, чтобы с точки зрения человека не было потерь как неявной и качественной информации, так и метаинформации.

Основное практическое правило: всегда лучше выполнить отладку сжатой информации вручную. Если человек не может воспользоваться сжатым изображением или состоянием, чтобы играть в игру или решить задачу, неудивительно, что алгоритм тоже окажется неудачным. Этот принцип применим не только к играм, но и в целом к любым задачам. Если состояние — это изображение, визуализируйте его и рассмотрите, если звук — прослушайте, если это какие-то абстрактные тензоры — распечатайте их и проверьте значения. Возьмите несколько примеров и попробуйте действовать, основываясь на них. В конце концов, все, что касается среды, должно иметь смысл для человека. Может оказаться, что проблема заключается не в алгоритме, а в синтезированном состоянии, предоставляющем недостаточно информации.

Преобразование состояний из необработанных в синтезированные может быть реализовано как модуль препроцессора состояний. Он может применяться либо внутри среды, либо путем преобразования агентом состояния, произведенного средой. Где его использовать, зависит от того, чем мы управляем: например, если среда представлена третьей стороной, логично реализовать препроцессор внутри агента.

14.5. Предварительная обработка

В этом разделе мы рассмотрим распространенные реализации предварительной обработки. Помимо этого, более детально расскажем о пропуске кадров. Этот раздел при первом чтении можно пропустить без ущерба для понимания следующих разделов.

Препроцессор — это функция, преобразующая данные перед их использованием. Предварительная обработка характерна не только для глубокого RL — она широко

применяется в машинном обучении. В целом предварительная обработка данных выполняется по многим причинам, среди которых следующие.

- **Очистка.** Например, при анализе естественного языка из предложений удаляются нестандартные символы.
- **Численное представление.** Обрабатывая естественный язык, нельзя выполнять вычисления непосредственно над символами алфавита. Их нужно закодировать как числа, такие как 0, 1, 2... или векторные представления слов, которые можно будет передать в нейронную сеть.
- **Стандартизация.** Нормализация входных данных широко применяется для того, чтобы смысл и диапазон значений всех признаков были аналогичными. Сети могут уделять признакам больше или меньше внимания в зависимости от их относительного масштаба. Часто масштаб может быть произвольным (например, длина может измеряться в миллиметрах или метрах), но это не должно влиять на обучение. Нормализация позволяет предотвратить это влияние, выполняя для каждого признака вычитание среднего значения и деление на среднее квадратическое отклонение.
- **Исследование.** Зачастую неясно, какое представление данных наилучшее для конкретной задачи. Одна из составляющих процесса экспериментирования — испытание разных методов предварительной обработки, таких как векторные представления, кодирование, преобразование в градации серого, субдискретизация и объединение кадров.

14.5.1. Стандартизация

Чтобы стандартизировать набор данных, сначала рассчитывают общее среднее значение и среднее квадратическое отклонение, затем из значения для каждой единицы данных вычитают среднее и разность делят на среднее квадратическое отклонение. Теперь общее распределение будет симметричным относительно 0 со средним квадратическим отклонением, равным 1. Это показано в уравнении (14.1):

$$x_{\text{нов}} = \frac{x - \bar{x}}{\sigma} — \tag{14.1}$$

и реализовано в листинге 14.2 (строки 2–6).

Другой тесно связанный метод — это нормализация данных, служащая для масштабирования *диапазона значений* каждого признака в наборе данных до промежутка между 0 и 1. Пусть дан набор данных, тогда для каждого признака найдем минимум, максимум и разброс значений, затем из значения для каждой единицы данных

вычтем минимум и разделим результат на величину разброса значений. Это показано в уравнении (14.2):

$$x_{\text{норм}} = \frac{x - x_{\min}}{x_{\max} - x_{\min}} - \quad (14.2)$$

и реализовано в листинге 14.2 (строки 8–15).

Листинг 14.2. Метод стандартизации тензора

```
 1 # исходный код: slm_lab/agent/algorithm/math_util.py
 2 def standardize(v):
 3     '''Метод стандартизации массива numpy ранга 1'''
 4     assert len(v) > 1, 'Cannot standardize vector of size 1'
 5     v_std = (v - v.mean()) / (v.std() + 1e-08)
 6     return v_std
 7
 8 def normalize(v):
 9     '''Метод нормализации массива numpy ранга 1'''
10     v_min = v.min()
11     v_max = v.max()
12     v_range = v_max - v_min
13     v_range += 1e-08 # ограничение при делении
14     v_norm = (v - v_min) / v_range
15     return v_norm
```

Методы из листинга 14.2 работают только для данных, полученных при обучении по отложенному опыту. В RL данные порождаются при обучении по актуальному опыту, то есть образуются постепенно с течением времени через взаимодействия агента со средой. Следовательно, нужен метод стандартизации полученных при обучении по актуальному опыту данных для состояний. Реализацию онлайн-стандартизации можно найти в SLM Lab в `slm_lab/env/wrapper.py`.

Эти эффективные методы широко применяются. Однако сначала важно проверить визуально, что в процессе не была утеряна никакая критическая информация. Проиллюстрируем это на примере.

Рассмотрим разнообразную среду с фиксированными по местоположению признаками, такую как карта в Dota 2. Территориальные и ключевые объекты игры зафиксированы в конкретных локациях. Предположим, что во время обучения агент видит только стандартизированные координаты местоположений. Поскольку карта большая, велика вероятность того, что агент сможет исследовать лишь малую ее часть. Это значит, что его стандартизированные данные будут представлять только этот небольшой участок карты. Если поместить агента на неисследованную территорию, то разброс или стандартное отклонение данных внезапно изменится. Это может привести к появлению после стандартизации аномальных единиц данных с предельными значениями. Еще бóльшая проблема заключается в том, что могут сместиться значения, которым сопоставлены виденные прежде состояния, и тем самым изменится их смысл. Если стандартизация или нормализация целесообразны,

но глобальная информация тоже необходима, то можно передавать агенту и обработанные, и необработанные данные. В архитектуре агента OpenAI Five абсолютная позиция на карте становится одним из входных значений [104].

Подводя итоги, нужно сказать, что стандартизация и нормализация — чрезвычайно удобные методы очистки набора данных. Они могут быть полезными для обучения, но применять их следует осторожно. Сведения об относительных изменениях при преобразовании сохраняются, но абсолютная глобальная информация может быть утрачена. Для реализации этих методов нужно глубокое понимание набора данных и информации, необходимой для решения задачи.

14.5.2. Предварительная обработка изображений

Цветное цифровое изображение обычно представлено тремя цветовыми каналами: красным, зеленым и синим, которые сокращенно обозначаются RGB (Red, Green, Blue). Это разложение изображения на три двумерных слоя, каждое значение на которых представляет интенсивность (яркость). Следовательно, каждый слой — это интенсивность одного из цветовых каналов изображения. Совмещением двумерных слоев друг с другом образуется трехмерный тензор. Каждый пиксел будет визуализироваться как комбинация его интенсивностей в основных цветовых каналах. На современных жидкокристаллических мониторах каждый физический пиксел в действительности является тремя расположенными вплотную друг к другу крошечными светодиодами красного, зеленого и синего цветов. Издалека нам видны цветные изображения, но незаметны светодиоды.

Для каждого изображения наличие трех цветовых каналов подразумевает наличие трех наборов значений, поэтому для его хранения потребуется в три раза больше места, чем для монохромного изображения. С помощью взвешенных сумм трем значениям каждого цветного пиксела может быть сопоставлено одно скалярное значение. Это называется *преобразованием в градации серого*, а результирующее изображение является двумерным тензором, который занимает в три раза меньше места на запоминающем устройстве, чем первоначальное изображение. Веса цветов следует подбирать аккуратно, чтобы не вызвать конфликта хеширования. К счастью, в стандартных библиотеках компьютерного зрения есть встроенные методы преобразования в градации серого с постоянными коэффициентами.

Даже после преобразования в градации серого разрешение современного цифрового изображения все еще остается высоким. Из-за этого на изображении оказывается гораздо больше пикселов, чем требуется в процессе обучения. Нужно уменьшить изображение, чтобы сократить количество значений в тензоре и тем самым уменьшить число элементов, нуждающихся в хранении и обработке. Это называется *субдискретизацией* (downsampling) или уменьшением размера. С субдискретизацией связана проблема размытия изображений и частичной потери информации. Существует много алгоритмов субдискретизации, при которых размытие изображений происходит по-разному.

Порой полезна только часть изображения, а бесполезные части — обычно это рамка — можно отбросить, чтобы еще больше уменьшить размер изображения. Как правило, это делается с помощью *обрезки* (cropping) изображения. В некоторых играх Atari в верхней и нижней частях показывается счет. Эти части не связаны напрямую с состоянием игры, поэтому их можно обрезать, хотя нет ничего страшного в том, чтобы оставить.

И последнее — интенсивность пикселов на изображении представлена целым числом, которое может принимать 256 значений в интервале 0–255. Общепринято производить *нормализацию изображения* путем приведения его к тензору чисел с плавающей запятой и поэлементного деления последнего на величину интервала 255. Таким образом, окончательный тензор изображения будет содержать значения в диапазоне 0–1.

Все эти методы предварительной обработки можно комбинировать. Из соображений эффективности их применяют в следующем порядке: преобразование в градации серого, субдискретизация, обрезка и нормализация изображения. В листинге 14.3 приведены реализация в SLM Lab методов предварительной обработки изображений, а также отдельные их комбинации.

Листинг 14.3. Методы предварительной обработки изображений в SLM Lab

```
# исходный код: slm_lab/lib/util.py
import cv2
import numpy as np

def to_opencv_image(im):
    '''Преобразование в структуру изображения h,w,c в OpenCV'''
    shape = im.shape
    if len(shape) == 3 and shape[0] < shape[-1]:
        return im.transpose(1, 2, 0)
    else:
        return im

def to_pytorch_image(im):
    '''Преобразование в структуру изображения c,h,w в PyTorch'''
    shape = im.shape
    if len(shape) == 3 and shape[-1] < shape[0]:
        return im.transpose(2, 0, 1)
    else:
        return im

def crop_image(im):
    '''Обрезка не используемых нижних рамок в играх Atari'''
    return im[18:102, :]

def grayscale_image(im):
    return cv2.cvtColor(im, cv2.COLOR_RGB2GRAY)

def resize_image(im, w_h):
```

```
29     return cv2.resize(im, w_h, interpolation=cv2.INTER_AREA)
30
31 def normalize_image(im):
32     '''Нормализация изображения делением на максимальное значение 255'''
33     return np.divide(im, 255.0)
34
35 def preprocess_image(im):
36     '''
37     Предварительная обработка изображения
       ➥ с помощью методов OpenAI Baselines: grayscale, resize
38     В resize вместо обрезки применяется растягивание
39     '''
40 im = to_opencv_image(im)
41     im = grayscale_image(im)
42     im = resize_image(im, (84, 84))
43     im = np.expand_dims(im, 0)
44     return im
```

14.5.3. Предварительная обработка временных данных

Предварительная обработка временных данных используется для восстановления или концентрации получаемой на множестве временных шагов информации, которую в противном случае было бы трудно или невозможно заметить по одному изображению.

Есть два эффективных метода предварительной обработки временных данных: *объединение кадров* (frame concatenation) и *накопление кадров* (frame stacking). Они применяются для учета получаемой за много шагов информации, такой как движение объектов в Pong из Atari и порядок следования изменений во времени.

Объединение и накопление могут использоваться совместно с любыми нерекурентными сетями. Пусть s_k — состояние, подаваемое на вход сети без какого-либо объединения или накопления. Собираем в общей сложности c кадров, где c — длина при объединении или накоплении, и сводим их в один тензор с помощью объединения (например, `np.concatenate`) или накопления (например, `np.stack` [143]).

Затем передаем предварительно обработанное состояние на вход сети. Заметьте, поскольку у нового входного значения другая структура, для его обработки входной слой сети должен быть соответствующим образом модифицирован. При обучении сеть настраивается видеть позиции в предварительно обработанных входных данных представленными в порядке следования во времени — после первого кадра идет второй и т. д.

Разница между двумя этими методами заключается в структуре результирующего выходного значения. При объединении сохраняется ранг тензора, но увеличивается размер измерения, по которому происходит объединение. Например, после объединения четырех векторов будет получен вектор в четыре раза длиннее, чем первоначальный. Объединение обычно производится над

состояниями, не являющимися изображениями, чтобы их можно было легко передать в простую сеть многослойного перцептрона с соответственно увеличенным входным слоем.

При накоплении, наоборот, ранг тензора увеличивается на 1 с сохранением структуры базовых тензоров. Это равносильно упорядочению последовательности состояний в списке. Например, накопление четырех изображений в градациях серого (двумерные тензоры) дает трехмерный тензор. Это аналогично изображению с четырьмя каналами, только теперь каналы соответствуют не цветам формата RGB, а временным позициям. Накопление обычно выполняется для состояний в виде изображений в градациях серого, а предварительно обработанный трехмерный тензор подается в сверточную сеть как изображение с четырьмя каналами.

Кроме того, объединение и накопление состояний можно сочетать с другими методами предварительной обработки. Например, можно, как обычно делается для сред Atari, сначала предварительно обработать изображения с помощью методов из листинга 14.3, а затем накопить их.

Пропуск кадров (frame skipping) — это метод временной предварительной обработки, который меняет эффективную частоту кадров. Например, при значении 4 для пропуска кадров в видео будет наблюдаться лишь каждый четвертый кадр, а остальные станут отбрасываться. Это форма субдискретизации для временных данных.

Пропуск кадров чаще всего выполняется в видеоиграх. Видео показывается на экране путем быстрой визуализации ряда идущих друг за другом изображений. Частота визуализации называется частотой кадров, например, для современных видеоигр это 60 кадров в секунду (FPS). Для плавного воспроизведения видео необходима высокая частота кадров, что означает показ большого количества изображений за короткое время. Именно поэтому файлы с фильмами обычно огромные.

Если выложить в ряд все изображения, составляющие видео, то их будет много — больше, чем нужно для обучения агента хорошей стратегии. Изменения на большинстве изображений будут слишком малы, чтобы быть значимыми, — это нужно, чтобы видео было плавным, но бесполезно для обучения. Обучающемуся агенту требуется воспринимать значимые различия между кадрами. Что еще хуже, обработка всех изображений будет слишком вычислительно затратной и непроизводительной.

Пропуск большинства кадров решает проблему лишних данных в видео с высоким FPS. Задав в качестве примера частоту пропуска кадров 4, мы будем визуализировать только каждый четвертый кадр и отбрасывать все изображения между ними. К тому же среда значительно ускорится из-за того что визуализация будет потреблять меньше вычислительных ресурсов. Это стало общепринятой практикой для игр Atari с момента, когда в оригинальной статье о DQN [89] был предложен способ создания эффективной частоты кадров 15 FPS. Этот же метод может быть

использован и в других видеоиграх, хотя для каждой из сред нужно подобрать подходящую частоту пропуска кадров.

По большому счету, пропуск кадров не ограничен только видео — он может быть применен к любой форме временных данных, имеющей параметр, равнозначный частоте кадров. В качестве примеров можно привести сигналы на фондовом рынке, аудиоданные, вибрации объектов и т. д. Уменьшение частоты выборки, по сути, является методом временной субдискретизации.

Несмотря на свою эффективность, пропуск кадров сопряжен с рядом проблем. В видеоиграх часто встречаются мерцающие элементы, такие как анимированные персонажи в классическом Mario или блоки в Breakout из Atari. К сожалению, если частота пропуска кадров совпадет с частотой мерцания этих элементов, то они не будут визуализированы должным образом. Или, что еще хуже, мы вообще их не увидим, если будут получены только те кадры, где эти элементы затемняются. Следовательно, может быть потеряна какая-то важная информация.

Одно из обходных решений — взять идущие подряд пропущенные кадры и выбрать максимальное значение (максимальную яркость) на пиксел. Обработка первоначально пропущенных данных приведет к увеличению количества вычислений и, возможно, замедлит обучение. Более того, здесь предполагается, что мы заинтересованы в ярких элементах, но в некоторых играх применяется темная цветовая схема, поэтому ключевые элементы игры будут, наоборот, темными. В этом случае потребуется выбирать минимальные значения пикселов.

Более простое решение проблемы, вызванной регулярным пропуском кадров, заключается в том, чтобы делать это стохастически. Рассмотрим в качестве иллюстрации случайный пропуск кадров с частотой 4. Для этого сгенерируем и сохраним в памяти следующие четыре кадра, затем случайным образом выберем один из них для визуализации, а остальные отбросим. Недостаток этого метода в том, что он вносит стохастичность, выраженную в случайных задержках кадров, что может быть приемлемо для многих, но не для всех задач. Этот метод предварительной обработки может поставить под угрозу решение задачи среды, если требуется стратегия с точностью до кадра. Стратегии с точностью до кадра реально существуют в киберспорте. Просто посмотрите на мировой рекорд по скоростному прохождению игры Super Mario Bros, чтобы увидеть, как лучшие рекордсмены для успешного выполнения отдельных заданий действуют строго в определенных кадрах.

В общем-то, не существует идеального метода пропуска кадров, приложимого ко всем задачам. У всех сред и наборов данных свои особенности, поэтому сначала нужно понять данные, а потом выбирать подходящий метод пропуска кадров.

Пропуск кадров можно применять и к действиям. Играя, люди обычно наблюдают множество кадров до того, как предпринять действие. Это разумно, ведь нам нужно получить достаточно информации, прежде чем принять взвешенное решение. При прохождении каждой игры есть своя идеальная скорость выполнения

эффективных действий. Для шутеров необходимы быстрые рефлексы, тогда как в стратегических играх требуется медленное и долгосрочное планирование. Общепринятый показатель — количество действий в минуту (actions-per-minute, APM) в предположении, что частота смены кадров постоянна (например, 60 Гц). Для нашего обсуждения более надежным является обратный показатель — количество кадров на действие (frames-per-action, FPA), полученное делением частоты смены кадров на APM. Всем играм присуще свое распределение FPA: динамичным играм — более низкие значения FPA, а играм в медленном темпе — более высокие. Поскольку мы совершаем действие один раз за много кадров, FPA в большинстве случаев должно быть выше 1.

При реализации пропуска кадров для действий нужно учесть одну деталь. Предположим, у среды есть основная *реальная частота смены кадров*, тогда как пропуск кадров дает меньшую *эффективную частоту смены кадров*. При пропуске кадров состояния воспринимаются с эффективной частотой, и с той же частотой предпринимаются действия. По отношению к реальным кадрам FPA равноценно частоте пропуска кадров. «Под капотом» шаги в среде производятся с реальной частотой смены кадров, поэтому в каждом реальном кадре ожидается соответствующее действие, в связи с этим с точки зрения внутренней структуры FPA всегда равно 1. Следовательно, при пропуске кадров нужно реализовать логику предоставления действий с реальной частотой пропуска кадров. Обычно это выполняется алгоритмом пропуска кадров среды.

В некоторых средах допускается не выполнять действия. В этом случае ничего не нужно делать, и беспокоиться следует лишь о предоставлении действий с эффективной частотой смены кадров. В других случаях среды требуют действий с реальной частотой смены кадров. Если действия не имеют накопительного эффекта, то они могут безопасно повторяться в пропущенных кадрах. Нажатие кнопки мыши в Dota 2 обладает этим свойством: число повторных нажатий, сделанных игроком до фактического выполнения действия, не имеет значения. Тем не менее если у действий есть накопительный эффект, то они могут вступать в противоречие. Например, бросок бананом в MarioKart [84] обладает накопительным эффектом: бананы — ценный ресурс, не стоит расходовать их впустую на повторяющиеся действия. Безопасный вариант — бездействие (предоставление холостых действий или `null`) в пропускаемых кадрах.

Если FPA слишком низкое для игры, то агент становится *гиперактивным* и его действия могут стать противоречивыми. Это явление способно приобрести форму броуновского движения — случайного блуждания с усреднением по большому количеству действий. Даже когда частота пропуска кадров идеальна, FPA все равно может оказаться слишком низким.

Одно из средств повышения эффективного FPA — помимо пропуска кадров для состояний пропускать еще и действия. Чтобы получить эффективное FPA со значением 5, имея частоту пропуска кадров для состояний, равную 4, нужно пропустить 5 эффективных кадров до изменения действия. К действиям также можно повторно

применить те же самые методы пропуска кадров с постоянными и случайными интервалами. Тогда относительно внутренней реальной частоты смены кадров состояния визуализируются каждые 4 реальных кадра, а действия выполняются каждые $5 \cdot 4 = 20$ реальных кадров.

Аналогично тому, как пропуск действий решает проблему *гиперактивности агента*, пропуск кадров для состояний решает проблему *гиперчувствительности агента*. Без пропуска кадров агент будет обучаться на мало различающихся состояниях, что означает, что он должен быть чувствительным к малым изменениям. Если после обучения неожиданно ввести пропуск кадров, то приводящие к успеху изменения в состояниях будут слишком велики для агента — он окажется гиперчувствительным.

Итак, для состояний пропуск кадров может производиться как с постоянными интервалами, так и случайным образом. Это приводит к различию между реальной и эффективной частотой смены кадров, и может понадобиться особая обработка выполнения шагов в среде. Кроме того, пропуск кадров может быть применен и к действиям.

14.6. Резюме

В этой главе были рассмотрены некоторые важные свойства состояний и их разнообразные формы, такие как векторы или изображения. Также было установлено различие между необработанным и синтезированным состояниями.

Следует проанализировать, содержит ли необработанное состояние, полученное из внешнего мира, полный набор сведений, необходимых для решения задачи. Затем, исходя из этого, спроектировать представление состояния — набор выбранных вручную сведений, полезных с нашей точки зрения. При проектировании состояния нужно учитывать ряд важных факторов — прежде всего его сложность, полноту и потери информации. Для удобства использования вычислительная сложность состояний должна быть низкой. Мы также обратили внимание на некоторые потенциальные проблемы, чтобы гарантировать, что в представлении состояния не происходят потери критически важной для решения задачи информации.

В конце мы рассмотрели несколько распространенных методов предварительной обработки состояний, таких как стандартизация численных данных, уменьшение размера, преобразование в градации серого и нормализация для данных в виде изображений, а также совмещение и пропуск кадров для временных данных.

15 Действия

Действие — это получаемое на выходе агента значение, которое изменяет среду, вызывая ее переход в следующее состояние. Состояние наблюдается, действие производится.

Проектирование действий имеет большое значение, поскольку они дают агенту способность изменять среду. От того, как спроектированы действия, зависит, будет управление системой легким или трудным, следовательно, это напрямую влияет на сложность задачи. Конкретная структура управления может быть рациональной для одного человека, но лишенной смысла для другого. К счастью, одно и то же действие всегда может быть произведено разными способами. Например, коробкой передач автомобиля можно управлять вручную или автоматически, но люди обычно считают, что второй вариант проще.

Многие выводы, сделанные в главе 14, применимы и к проектированию действий. По структуре эта глава схожа с остальными: сначала даны примеры некоторых действий и рассказано об основных принципах проектирования, затем рассмотрены полнота и сложность действий.

15.1. Примеры действий

В повседневной жизни действия могут проявляться в разных формах, от знакомых до неожиданных. Все, что может производить изменения в среде, считается действием. Игровые контроллеры и музыкальные инструменты — один из самых привычных примеров элементов управления действиями. Другой — физические перемещения нашего тела. Движения роборуки внутри эмуляции — виртуальные действия.

В числе менее знакомых находятся голосовые команды виртуальному ассистенту — действия, заставляющие его выполнять задания. Кошачье мурлыканье — это действие для привлечения внимания людей. Магнитное поле порождает силу, действующую на заряженные частицы. Фактически действия могут приобретать любую форму при условии, что это информация, которую можно передать, чтобы изменить систему.

Как и реальные среды, действия тоже могут быть сложными. Например, работа рукой — это сложная форма управления с привлечением большого числа мускулов, точных координат и ловких реакций на воздействие окружающей среды. Мы, люди, пока растем, тратим много лет на то, чтобы овладеть всем этим. В моделируемых средах действия часто упрощают, например, роборука может иметь до десяти степеней свободы. Это существенно отличается от сложности человеческой руки, но не всегда является недостатком.

Подобно состояниям, действия могут быть представлены как числа, упорядоченные в виде любых подходящих структур данных. Их нужно проектировать, чтобы они должным образом представляли информацию, необходимую для использования алгоритмом. Это также является формированием признаков, и здесь с тем же успехом можно применить продуманные принципы проектирования.

Действие a — это элемент пространства действий A, определяющего все возможные действия, которые может воспринимать среда. Действия могут быть дискретными и пронумерованными с использованием целых значений, как кнопки с номерами этажей в лифте, или непрерывными с вещественными значениями, как нажатие педали газа в машине. Они также могут быть комбинацией дискретных и непрерывных значений, как в случае с чайником, у которого есть кнопка включения (дискретное значение) и регулятор уровня температуры (непрерывная величина)[1]. Действие может состоять из более чем одного элемента (измерения), и каждый элемент, как дискретный, так и непрерывный, может иметь любую мощность.

Для удобства и совместимости со стандартными библиотеками для вычислений действия также кодируются как тензоры с данными одного типа (`int`, `float`), ранг и структура которых могут быть любыми:

- скалярная величина (тензор ранга 0) — регулятор температуры;
- вектор (тензор ранга 1) — клавиши пианино;
- матрица (тензор ранга 2) — кнопки калькулятора.

Важно упомянуть различие между измерением и мощностью. У регулятора температуры есть только один элемент и, следовательно, одно измерение, но его значение непрерывно, значит, его мощность $\left|\mathbb{R}^1\right|$. У пианино 88 клавиш, каждая из которых при грубом рассмотрении является бинарной (нажата или нет), так что мощность пространства действий для пианино — 2^{88}. Ранг и структура тензора действий просто показывают, как упорядочены его элементы.

Большинство элементов управления, с которыми мы можем непосредственно взаимодействовать, имеют всего лишь ранг 2. Регуляторы, телефон, клавиатура

[1] Можно возразить, что в реальном мире все является дискретным с какой-либо минимальной степенью дробности, но на теоретическом уровне предпочтительно рассматривать некоторые вещи как непрерывные.

и пианино — это тензоры действий ранга 1. Музыкальный синтезатор и мультитач в смартфоне — тензоры ранга 2, как и кран в душе, где обычно есть одна ручка, поднимая которую, можно регулировать напор воды, а поворачивая — температуру.

Тензоры действий ранга 3 хоть и редко, но все же встречаются. Один из примеров — контроллер виртуальной реальности, позволяющий захват движения в трехмерном пространстве. Или представьте себе склад с большим количеством ячеек, образующих трехмерную сеть, дверцами которых управляют с помощью компьютера. Эти виртуальные кнопки «закрыть/открыть» формируют трехмерную схему.

Кроме того, действие может быть комбинацией тензоров. Чайником можно управлять с помощью кнопки, нажатием которой его можно включить, и ручки регулировки температуры. DJ-контроллер — пример с богатым набором из кнопок, регуляторов, ползунков и проигрывателей пластинки. В компьютерных играх, как правило, требуются комбинированные входные значения, такие как нажатия клавиш на клавиатуре и движения мышью. Вождение автомобиля также подразумевает много элементов управления: педаль газа, тормоз, руль, коробка передач, световой индикатор и т. д.

Когда действия представлены множеством тензоров с разными структурами, нужно, чтобы в нейронной сети для них были отдельные выходные слои. Эти различающиеся тензоры рассматриваются как поддействия, вместе образующие смешанное действие.

Несмотря на то что действия выполняет агент, они определяются средой, решающей, какие из них приемлемы. Агент принимает состояния и действия так, как они заданы в среде, тем не менее он может предварительно преобразовывать их для своих целей при условии, что окончательный формат подаваемых на вход сети данных будет корректным.

В целом есть три способа проектирования действий как сингулярных, сингулярно-комбинированных и составных. Рассмотрим несколько примеров.

В простых средах управление в основном *сингулярное*. Действие — это тензор ранга 0 (скалярная величина), поэтому агент может выполнять только одно действие за раз. В CartPole доступные действия — перемещения либо влево, либо вправо, но агент одновременно может выбрать только одно из них.

Однако это не всегда так. Старый игровой контроллер Atari поддерживает комбинации действий, например нажатие кнопки для выстрела и поворот джойстика для перемещения в игре. При проектировании среды данные сочетания могут быть реализованы так, чтобы представляться агенту отдельными действиями, что можно описать как сингулярные комбинации. На практике это достигается полным перечислением всех возможных комбинаций и сопоставлением им списка целых чисел. Благодаря такой структуре упрощается создание сети стратегии, поскольку нужен только один выходной слой. Агент будет выбирать на каждом временном шаге

одно из целых чисел, которому при передаче в среду будет сопоставлено обратное сочетание сигналов кнопки и джойстика.

Бывают случаи, когда из-за слишком большого массива сингулярных комбинаций задача управления усложняется. В робототехнике агентам часто приходится одновременно контролировать множество крутящих моментов или углов поворота суставов, которые представлены различными вещественными числами. Для полного перечисления всех этих сочетаний потребуется неоправданно большой массив чисел. В таких случаях измерения оставляют раздельными, так что действие *состоит* из субдействий, по одному на каждое измерение.

Действия в RL проектируются преимущественно для использования машинами или программным обеспечением, но мы можем черпать вдохновение в разнообразных изобретательных элементах управления, разработанных для человека. Ряд примеров приведен в разделе 15.5. Мы заимствуем оттуда основной принцип проектирования: элементы управления должны быть знакомыми и интуитивно понятными для пользователей, так как это позволяет им лучше понимать, какое влияние оказывают их действия. Целесообразно так проектировать элементы управления и для аппаратного, и для программного обеспечения, чтобы их было проще понимать, а значит, и легче отлаживать. Кроме того, интуитивность элемента управления дает возможность передавать рычаги управления от программного обеспечения обученному человеку. Эта функция особенно полезна в автоматических системах управления, автопилотируемых автомобилях и заводском оборудовании.

Мы, будучи людьми, в повседневной жизни контролируем многие вещи, что может служить как хорошими, так и плохими примерами проектирования действий. Проектирование элементов управления в целом подпадает под более широкую категорию интерфейсов пользователя, которая относится к области разработки взаимодействий между им и системой. Хорошие интерфейсы пользователя должны быть эффективными и простыми в применении. Плохо спроектированные элементы управления могут сделать задание более сложным, сильнее подверженным ошибкам и порой опасным.

15.2. Полнота действий

При задании пространства действий нужно обеспечить предоставление достаточно разнопланового и точного управления для решения задачи — это и есть *полнота*. Можем ли мы контролировать все, чем нам нужно управлять?

Для начала полезно подумать, что человек использовал бы для управления системой, чтобы решить задачу. Затем проанализировать, какая степень точности и какой ранг элементов управления требуются. Например, индустриальным роботам могут понадобиться элементы управления с точностью до долей градуса, но им

не нужно далеко перемещаться. В то же время человеку за рулем автомобиля необходима точность лишь до нескольких градусов, но машина должна быть способной совершать поездки на большие расстояния.

Принимая решение о том, *что* нужно контролировать, мы можем черпать вдохновение у людей, например, посмотреть на структуру видеоигры. Но агент может управлять системой совсем иначе. Например, машина может контролировать много вещей одновременно с большой точностью, а людям это очень сложно. Это лишь один пример того, насколько по-разному люди и агенты могут управлять робототехнической системой.

В видеоиграх встречаются одни из самых лучших и богатых структур элементов управления. Это и неудивительно, учитывая существующее разнообразие игр и то, что многие из них имитируют сценарии из реального мира. Как правило, проектирование элемента управления происходит следующим образом.

Сначала, исходя из игровой динамики, определите элементы, которыми должен управлять игрок для достижения цели. Хорошо если есть теоретическая модель игры, обычно представленная в виде блок-схемы, описывающей все игровые сценарии. Проверьте их работоспособность, чтобы хотя бы в теории обеспечить корректность выявленных элементов и предписанных им действий. Это гарантия того, что действия определены полностью. Как только элементы идентифицированы, выясните способы управления ими. Если игра имитирует реальность, то это должно быть относительно просто, ведь элементы управления могут имитировать свои аналоги из реальной жизни. В других случаях необходимо проектирование.

Если проектирование элемента управления осуществимо, то нужно рассмотреть следующие факторы: интуитивность и эффективность. Они не только позволяют создать оптимальный игровой процесс, но и влияют на сложность игры. Плохо спроектированные интерфейсы могут сделать задание намного более сложным, чем следует. Проектирование должно обеспечивать оптимальное соотношение между лаконичностью и многословностью. Предположим, что в игре есть 100 различных действий. Можно иметь 100 кнопок, по одной на действие, но это слишком многословно. Также это могут быть два циферблата по 10 вариантов, комбинациям которых сопоставляются эти 100 действий, но это слишком лаконично и малопонятно. Действиям может быть присуща естественная схема категоризации, например набор атак, набор заклинаний и набор примитивов движений, таких как перемещение влево или вправо. Тогда более естественным будет разделить элементы управления по этим категориям. Внутри каждой категории можно использовать более компактный дизайн с несколькими кнопками, по одной на каждый элемент из категории.

Наконец, после завершения проектирования и реализации элемента управления проектировщику нужно его протестировать. Проверьте путем верификации и сравнения с применявшейся при реализации теоретической схемой, действительно ли набор действий, доступных с помощью структуры, является полным. Затем напи-

шите ряд модульных тестов и удостоверьтесь, что реализация действий рабочая и ход игры не нарушается. Часто модульные тесты дополнительно проверяют, чтобы выходные значения находились в пределах заданного диапазона, так как предельные значения могут вызвать неожиданное поведение. Для подтверждения полноты действий проверьте, чтобы в проекте не были упущены какие-нибудь нужные действия или не производились какие-то посторонние действия, которые не были задуманы изначально. Как правило, процессы проектирования и тестирования повторяются несколько раз с улучшением и тонкой настройкой структуры, пока она не станет приемлемой для выпуска игры.

Тот же процесс проектирования, помимо игр, может быть эффективно применен к неигровым приложениям, но тогда возникает ряд вопросов. Во-первых, что мы хотим, чтобы агент делал в данной среде? Этот вопрос, ответ на который, по большому счету, дается при определении целевой функции среды, помогает задать направление, которого следует придерживаться при составлении полного набора действий. Во-вторых, каким образом агент может влиять на среду? Какими элементами агент должен управлять? А что насчет элементов, которые агент *не может* контролировать? Это касается установления существующих категорий действий. В-третьих, как агент будет ими управлять? Это определяет возможные схемы проектирования для предусмотренных действий и предлагает интуитивные и эффективные способы их кодирования. Реализованный проект управляющего элемента нужно протестировать аналогичным образом.

Обратимся к робототехнике и в качестве примера рассмотрим задание, в котором нужно поднимать объекты. Роботизированная рука движется вдоль трех осей, следовательно, у действия будет три проекции, по одной для каждой оси. Рука может свободно двигаться в сторону объекта, но у нее нет возможности его подобрать. Такая структура действий неполная, поэтому задание не может быть выполнено. Это можно исправить, прикрепив к концу руки захват для предметов. У окончательного рабочего элемента управления есть четыре измерения: три для перемещения руки и одно для захвата. Теперь роботизированная рука может выполнить полный набор задуманных действий и достичь своей цели.

Предположим, мы хотим создать интерфейс для управления роботизированной рукой. Какая структура будет оптимальной? Чем будут отличаться друг от друга интерфейсы для робота под управлением машины и человека? Один из вариантов — действия, представленные четырьмя числами, соответствующими трем осям и одному захвату. Такой структурой проще всего управлять с помощью машины, а для человека это будет не так уж легко и интуитивно. Ему сначала нужно будет ознакомиться с тем, как числа отображаются в действительные состояния роботизированной руки, не говоря уже о том, что набирать отдельные числа он будет довольно медленно. Для человека удобнее структура, включающая три поворотных диска, с которыми сопоставлено вращение вокруг осей, и один ползунок для сжатия и разжатия захвата.

Приняв решение о технической реализации действий, можно определить подходящие типы данных для их кодирования в правильных диапазонах. Если элемент управления принимает дискретные значения, то его можно воплотить в кнопке или индикаторе с дискретной шкалой и, соответственно, закодировать как целое число. Если значения непрерывные — реализовать в виде ползунка и закодировать как вещественное число.

Важно добиться равновесия между лаконичностью и выразительностью структуры пространства действий. Когда пространство действий слишком мало, агент может оказаться неспособным управлять системой с достаточной точностью. В худшем случае он вообще не сможет выполнить желаемое задание. Если пространство действий слишком обширное или сложное, обучаться управлению системой станет намного труднее. В следующем разделе мы рассмотрим некоторые методы регулирования сложности действий.

15.3. Сложность действий

Обсудим несколько стратегий решения проблемы сложности действий, включая объединение и разделение действий, применение условных действий, дискретизацию непрерывных действий и использование инвариантности в среде.

Один из способов обращения людей со сложными элементами управления — это объединение низкоуровневых действий в одно высокоуровневое.

У пианино 88 клавиш. Чтобы научиться играть на нем, нужно всю жизнь практиковаться. При управлении таким большим количеством клавиш, играя, мы не фокусируемся на одной клавише за раз, так как наш мозг этого не умеет. Кроме того, концентрироваться на отдельных клавишах неэффективно — намного проще упорядочить их последовательности в большие шаблоны. При написании музыки для пианино и игре на нем используют аккорды, гаммы и интервалы.

В компьютерных играх, таких как Dota 2 и StarCraft, много компонентов и огромное пространство возможных стратегий. Придется провести многие годы перед экраном, чтобы научиться играть в такие игры на профессиональном уровне. Обычно в игре есть много командных кнопок для управления юнитами и их действиями. Умелые игроки не играют на уровне отдельных действий — они применяют командные клавиши быстрого доступа, комбинации кнопок и повторимые макрокоманды. С опытом это кодируется в их мышечной памяти, по мере того как они находят все больше коротких путей, чтобы понизить сложность действий, реорганизуя их и создавая шаблоны более высокого уровня.

Фактически высокоуровневые шаблоны равнозначны схемам управления, сформированным из первоначальных элементов управления, — точно так же, как аккорды

являются элементарными схемами управления, составленными из отдельных клавиш. Этот метод является формой *метауправления*, и люди применяют его постоянно. Пока нам еще неизвестно, как заставить агентов RL автономно проектировать собственные схемы управления. Следовательно, заниматься метауправлением нам придется самостоятельно — мы ставим себя на их место и проектируем эти шаблоны со своей точки зрения.

Иногда действие проще представить не как одно сложное действие, а как комбинацию множества субдействий.

Например, рассмотрим игру-шутер, в которой игрок целится в экран размером 84×84 пиксела. При проектировании действия «целиться» мы можем представить его и как одномерное, и как двумерное. Выбор действия в двумерном пространстве будет происходить с учетом двух распределений мощностью 84 каждое. Если биективно отобразить это в одномерное пространство, то его мощность будет $84 \cdot 84 = 7096$. В этом случае действие будет выбираться из одномерного распределения, весьма обширного и разреженного. Разреженность может осложнить выполняемые агентом исследования среды и искусственно усложнить задачу. Представьте себе, что игроку нужно выбирать из 7096 кнопок, — скорее всего, он выучит лишь несколько и будет пользоваться ими, игнорируя остальные. По сравнению с этим выборка из двумерного пространства намного удобнее.

Если для задачи естественным является управление во множестве измерений, как в случаях с шахматной доской или игровой консолью, то при проектировании действий нужно найти баланс между выразительностью и разнообразием. Все измерения можно оставить раздельными или биективно отобразить в одно измерение. Наличие слишком большого числа измерений повышает сложность элемента управления, поскольку агенту нужно исследовать и учиться контролировать отдельные субдействия одного действия. Но мы также видели, что разделение действий в соответствии с множеством измерений может быть эффективным способом уменьшения мощности пространства действий.

Общая рекомендация: такие вещи, как сочетания кнопок, лучше кодировать как уникальные действия, потому что так их проще обнаруживать, особенно если комбинация состоит из длинной последовательности субдействий.

Мы также видели, что действия игрового контроллера Atari можно биективно отобразить в одномерный вектор, которому сопоставлены все возможные комбинации для кнопок и джойстика.

Это может быть полезным в некоторых сценариях, но не всегда. На примере с шахматами было показано, как перевод ходов фигур из двумерного пространства в одномерное может затруднить игру. В других случаях биективное отображение в одномерный массив настолько раздувает пространство действий, что его просто

нереально применять на практике. Если взять изображение из современной видеоигры при полноэкранном режиме с разрешением 1920×1080 пикселов и биективно отобразить его в одномерный вектор, то мощность вектора составит 2 073 600.

Одно из эмпирических правил — при биективном отображении пространства действий элемента управления в массив с меньшей, чем первоначальная, размерностью проанализируйте разреженность и потери информации. Если при биективном отображении теряются ключевые сведения или метаинформация, такие как пространственный характер управления, то его следует избегать. Поскольку биективное отображение в меньшее измерение повышает его мощность, контрольная выборка из этого измерения может неожиданно стать слишком разреженной.

Другой фактор, который следует учитывать, когда речь идет о понижении сложности, касается абсолютного и относительного контроля. В Dota 2 агент может управлять большим количеством юнитов на обширной игровой карте. При проектировании элемента управления для перемещения юнитов с помощью абсолютных координат x и y решение задачи становится почти невозможным просто из-за огромного размера карты. Лучшая схема управления — позволить агенту сначала выбрать юнита, получить его текущую позицию, затем выбрать ограниченные относительные смещения по x и y и применить к нему. В агенте OpenAI для Dota 2 [104] таким образом на порядок снизили мощность массива действий при перемещениях. Это работает и при любых больших размерностях действий, но с естественным порядком следования значений, таким как шкала значений. Использование относительного масштаба может сильно сократить мощность пространства действий, и в результате может быть найден подход к решению задачи, бывшей неразрешимой.

Преимущество данного подхода заключается в том, что относительные смещения по x и y можно применить к юниту, где бы он ни находился, поэтому такая схема управления обладает симметрией переноса. Обучившись полезному маневру, агент сразу же может применять его к юнитам в любом месте на карте. Один и тот же маневр можно эффективно объединять с другими субдействиями в различных сценариях и даром получать разноплановые действия, обучаясь множеству более простых субдействий.

Важно то, что возможность относительного контроля обусловлена природой данной задачи — ее локальностью. Поскольку управление перемещениями юнита имеет значение лишь для его окрестности, то передвижения юнита — это локальная задача, к которой можно применить относительный контроль. Однако недостаток этого — невосприимчивость по отношению к любым глобальным стратегиям, связанным с территорией или картой, которые не являются локальными. Чтобы решить нелокальный аспект задачи, нужно восстановить глобальный контекст и дополнительную информацию.

Упростить непрерывный контроль можно путем дискретизации. При этом нужно подумать о разрешении пространства, то есть о том, насколько большими окажутся

«пикселы», когда непрерывное пространство будет преобразовано в сетку. Если разрешение слишком грубое, агенту придется приложить немалые усилия, чтобы предоставить правильную аппроксимацию. Если оно слишком высокое, задача может стать неразрешимой с вычислительной точки зрения. Дискретизацию следует выполнять с осторожностью, чтобы не получить пространство действий со слишком большой мощностью.

Другая потенциальная проблема — это введение искусственных ограничений в пространстве, которое в противном случае было непрерывным. Пусть непрерывное пространство действий с диапазоном значений от 0 до 2 дискретизируется в целые числа 0, 1, 2. Тогда оправданно ли округление числа 1,5 до 2 при дискретизации? Если при дискретизации используется слишком грубое разрешение, мы рискуем до некоторой степени утратить точность и чувствительность дискретизированных действий.

Мы, люди, пользуемся интуитивным пониманием задачи и способностью к адаптации, чтобы учиться, тестировать наши элементы управления и настраивать их чувствительность для достижения баланса между точностью и эффективностью. Мы понимаем, насколько большим должно быть приращение температуры в термостате или насколько чувствительным должен быть регулятор громкости на колонке. Другое важное различие между людьми и машинами — время реакции. Машины могут реагировать предельно быстро, поэтому может понадобиться ввести ограничение скорости агента, чтобы он действовал каждые N временных шагов, для предотвращения его гиперактивности. Иначе из-за быстрой смены действий — нейтрализации влияния предыдущих действий в короткие промежутки времени — результирующее действие может стать случайным и импульсивным. В играх Atari применяется выборка с частотой пропуска кадров 4, чтобы заданное действие повторялось во всех пропущенных кадрах при внутреннем прохождении по реальным кадрам игры. Кроме того, можно позволить агенту адаптивно подбирать для себя идеальную частоту пропуска кадров, исходя из сценария. Люди уже так делают — например, в Tetris мы ускоряем игру, когда готовы быстро опустить блок.

Один из самых удобных методов снижения сложности, в целом применимый к действиям и состояниям, — это симметрия. В шахматах агенту не нужно учиться играть за обе стороны по отдельности. Если он на месте второго игрока, то ему нужно лишь повернуть доску (с абсолютными координатами) и задействовать уже настроенную стратегию. Таким образом, вне зависимости от того, на какой стороне находится агент, он применяет для игры одни и те же стандартные координаты. Симметрия может быть пространственной или временной. Распространенные типы симметрии — симметрия переноса, осевая, зеркальная, спиральная (вращение со смещением) и т. д. Все они могут быть описаны как функциональные преобразования. Будучи заданными однократно, преобразования могут применяться к состояниям и действиям для понижения на порядок мощности пространств состояний и действий.

Для RL особенно характерно наличие несимметричных взаимосвязей между состояниями и действиями. Пространство состояний, как правило, намного больше и сложнее, чем пространство действий. Тензоры состояний обычно содержат больше элементов и часто демонстрируют большее разнообразие, чем тензоры действий. Например, состояние в виде изображения из Pong из Atari (после предварительной обработки) содержит $84 \cdot 84 = 7096$ пикселов. А каждый пиксел может принимать одно из 256 значений, что дает мощность пространства состояний, равную 256^{7096}. Сравните это с действием, у которого есть одно измерение и которое может принимать одно из четырех значений, так что мощность пространства действий[1] $4^1 = 4$.

Когда пространство действий мало по сравнению с обширным пространством состояний со значительными вариациями, агенту нужно настроить функцию «многие к одному», в которой многие состояния отображаются в одно и то же действие. Бóльшее пространство состояний содержит много информации, помогающей делать выбор из относительно малого набора действий. Если бы было справедливо обратное, то сети, наоборот, потребовалось бы настроить функцию «один ко многим», в которой одно состояние отображается во много возможных действий. В этом случае предоставляемой состоянием информации может оказаться недостаточно для выбора между действиями.

Пусть тензор состояний s несет n_s бит информации, а тензор действий a — n_a бит. Если для корректного определения одного бита действия требуется один бит информации из состояния, то, чтобы все возможные значения a были полностью различимыми, нужно по крайней мере, чтобы $n_s = n_a$. При $n_s < n_a$ это невозможно, поэтому некоторые из значений a будут недостаточно определены.

Для Pong из Atari задача порождения одного из четырех возможных значений действия для заданного большого пространства состояний хорошо определена, и состояния содержат достаточно информации, чтобы алгоритм научился оптимальной стратегии. Представьте себе обратную ситуацию, когда нужно получить изображение, притом что заданы лишь четыре значения действий. Сгенерировать полное разнообразие изображений, с которыми мы сталкиваемся при игре в Pong, будет непросто.

Наконец, при поиске подхода к решению новой задачи начинайте с самой простой структуры действий (то же самое касается состояний). Хорошо, если первая структура действий будет ограниченной, чтобы можно было сконцентрироваться на простейших аспектах задачи. Упрощение задачи может быть весьма полезным для постепенного построения решения. Если первоначальная структура кажется многообещающей и агент может научиться решать упрощенную версию, то можно постепенно повышать сложность, делая задачу более реалистичной и приближенной к полной версии. В ходе постепенного продвижения проектирования мы сможем лучше понять природу задачи и то, как агент учится ее решать.

[1] Действия в Pong в OpenAI Gym: 0 (бездействие), 1 (подача), 2 (вверх), 3 (вниз).

Методы понижения сложности в основном касаются действий, но их можно применить и к состояниям, когда это уместно. Подводя итог, приведем рекомендации относительно сложности в проектировании состояний.

1. **Выполнить биективное отображение в меньшее количество измерений или разделение на большее число измерений.** Биективно отобразите действия в одномерное пространство, чтобы обнаружить сложные сочетания субдействий, примените разделение измерений действий для уменьшения их мощности.
2. **Переключение между абсолютным и относительным управлением.** Упростите, пользуясь преимуществом относительного действия для локальных элементов управления, применяйте абсолютный масштаб для глобальных элементов управления.
3. **Дискретизация.** Попытайтесь не отходить от изначальной природы элемента управления, но для упрощения задействуйте дискретизацию, при этом удостоверьтесь, что разрешение достаточное.
4. **Упрощение с помощью симметрии.** Попытайтесь найти симметрию в действиях и примените ее для уменьшения пространства действий.
5. **Проверка соотношения между размерами пространств состояний и действий.** С точки зрения здравого смысла пространство действий не должно быть более сложным, чем пространство состояний, — в основном намного меньше.
6. **Переход от простого к сложному.** Начинайте с самой простой структуры действий, а затем постепенно повышайте сложность.

15.4. Резюме

В начале этой главы говорилось о разработке элементов управления для людей как источнике идей, она сравнивалась с проектированием действий для агентов. Были также рассмотрены полнота и сложность действий: оптимальная структура действий должна позволять агентам эффективно управлять всеми значимыми компонентами в среде и не должна без необходимости усложнять задачу. В конце приведен ряд рекомендаций по уменьшению сложности в структуре действий.

15.5. Проектирование действий в повседневной жизни

Этот раздел описывает ряд интересных и необычных структур действий для людей и призван стать источником вдохновения.

Проектирование действий для людей больше известно как дизайн интерфейса пользователя (UI). Это обширная и разнообразная дисциплина, охватывающая множество областей, от разработки промышленных продуктов до игр, а с недавних

пор — веб-дизайн и проектирование приложений. Несмотря на то что термин возник недавно, дизайн интерфейсов существовал на протяжении всей истории человечества, начиная с первых доисторический каменных топоров, сделанных нашими предками, и заканчивая современными смартфонами. Он всегда влиял на наше взаимодействие с окружением. При поиске идей для хорошего дизайна интерфейса стоит остановиться на играх и музыкальных инструментах, повсеместно распространенных в человеческом обществе.

По сути, музыкальные инструменты — это устройства управления, взаимодействующие с воздухом с целью порождения сложных состояний в звуковой форме посредством щипков, выдувания воздуха или ударов. Для получения качественного звучания их необходимо скрупулезно изготовить вручную и тщательно настроить. Помимо этого, производителю необходимо учесть особенности пользователей: имеют значение размер их руки, сила и тренированность пальцев. Первоклассный скрипач сыграет на обычной современной скрипке лучше, чем новичок на скрипке Страдивари (скрипки, изготовленные семейством Страдивари в 1700-х годах и до сих пор считающиеся непревзойденными в мире). Тогда как выдающийся музыкант может извлечь из такого инструмента потрясающую виртуозную музыку, вызывающую сильные эмоции.

Рассмотрим классические клавишные инструменты, которые производят звук с помощью воздушных столбов (орган), щипков струн (клавесин) или ударов по струнам (пианино). Разные регистры органа и клавесина и педали пианино повышают звук при выверенном касании клавиш. Мы видели, что действия для пианино можно упростить до 88 целых чисел, но этого недостаточно для обучения робота правильной игре на нем. В реальности на каждую клавишу пианино можно нажимать с разным усилием, порождая *тихие* (piano) и *громкие* (forte) ноты. Кроме того, в пианино есть три ножные педали для преобразования звука: *левая* (soft), *средняя* (sostenuto) и *правая* (damper). В результате сложного взаимодействия этих управляющих элементов возникает богатое звучание, так что моделирование действий и состояний (звуков), производимых пианино, было бы непростым заданием. Все же это возможно и уже было реализовано в цифровых пианино (синтезаторах), хотя их звучание все еще отличается от звучания настоящего. Для обучения робота, играющего на пианино, сперва можно попытаться видоизменить программное обеспечение для синтезатора, переведя его в виртуальную среду, а после дать роботу обучаться на дорогом пианино с риском поломки последнего.

Люди не перестают изобретать новые музыкальные инструменты. Современные музыкальные жанры, такие как электронная танцевальная музыка и техно, известны тем, что ломают стереотипы и создают новое звучание с помощью современных технологий. Один из ярких примеров вновь созданных инструментов — новый вид миди-клавиатуры под названием *Seaboard* [116]. Он выводит пианино на уровень непрерывного управления: в Seaboard вместо отдельных клавиш есть одна волновая поверхность, напоминающая очертаниями клавиатуру пианино. Прикосновение

к любой точке этой поверхности производит звук из одного из многочисленных программных пресетов (гитара, струнные и т. д.). Более того, его чувствительная поверхность распознает пять типов касания (5D Touch): силу нажатия (strike), давление после нажатия (press), скорость подъема (lift), движения влево-вправо (glide), движения вверх-вниз (slide). Кроме того, для новых типов управления ROLI, производителем Seaboard, были введены новые условные обозначения для партитур. Фактически с учетом двумерной поверхности, пяти типов касания и еще одного для звуковых пресетов у этого инструмента в общей сложности восемь измерений: семь непрерывных и одно дискретное. Взглянуть на него в действии можно в видео на YouTube и на его собственном сайте https://roli.com/products/blocks/seaboard-block.

Игры — другой прекрасный источник идей для проектирования элементов управления. Существуя в человеческом сообществе с давних времен, игры развиваются вместе с доступными технологиями. Каждое поколение игр с собственными средой, трудностями проектирования и нововведениями отражает технологию своей эпохи. Когда деревянная черепица стала обычным явлением, в Китае придумали домино, доступность бумаги позволила создать игральные карты. В играх докомпьютерной эпохи управление и взаимодействие происходили с помощью физических игровых элементов, таких как домино, карты, кости, шахматные фигуры, мраморные шарики, палочки и т. д.

В раннюю компьютерную эпоху электронные игры стали популярными благодаря игровым автоматам, прежде всего Atari и Sega. Выводимые на экраны этих машин изображения соответствуют состояниям в RL. Их управляющие элементы — физические кнопки и джойстик — соотносятся с действиями. Разработчики игровых автоматов должны, кроме разработки игр, создавать еще и подходящие для них интерфейсы необходимых элементов управления. Порой это подразумевает изобретение интерфейса нового элемента управления, помимо обычных джойстика и кнопок. Многие из этих ранних инноваций и поныне наглядно представлены в залах игровых автоматов. В шутерах есть пластиковые пистолеты или джойстики, из которых можно целиться по мишеням на экране и стрелять. Игры со стрельбой и вождением, такие как Jurassic Park 1994 года для Sega, дополнены подвижными сиденьями, двигающимися, когда игроков на экране преследуют динозавры. В типичном гоночном игровом автомате есть руль, ножные педали и коробка передач. В танцевальной игре Dance Dance Revolution — коврик, на который игроки могут наступать, чтобы стрелки ритма на экране совпадали с музыкой. В других игровых автоматах имеются уникальные элементы управления, созданные специально для их игр, такие как лук и стрела, молоток, кран-машина и т. д.

Перенос технологии электронных игр породил консольные игры. Большинство из них унаследовало элементы управления своих предшественников с двумя кнопками, но джойстик из аркадных игр был заменен на четыре отдельные кнопки со стрелками. Когда консольные игры стали популярными, производители, такие как

Sony и Nintendo, начали вводить в свои устройства новые элементы управления. У них уже была привычная форма, которую мы видим сегодня, — по четыре кнопки на каждой стороне с дополнительными кнопками движения влево и вправо вверху. Современные консольные системы, такие как PSP и Xbox, имеют джойстик, позволяющий вернуться к более полному контролю над направлением. В более продвинутых приставках, в числе которых Wii и Switch от Nintendo, есть также гироскопические элементы управления, распознающие движения в трехмерном пространстве.

По мере широкого распространения персональных компьютеров быстро возросло количество компьютерных игр, и проектирование элементов управления вскоре расширилось до клавиатуры и мыши. В более простых играх джойстик и две кнопки ранних консолей обычно имитируются кнопками со стрелками и двумя другими кнопками. Введение движений и щелчков кнопкой мыши открыло новые возможности, такие как панорамирование — взгляд сверху на перемещения игрока. Благодаря этому стали более интуитивно понятными игры-шутеры от первого лица, такие как Doom, где кнопки со стрелками используются для перемещения игрока, а мышь — для того чтобы поворачивать камеру, целиться и стрелять. По мере возрастания сложности игр то же самое происходило с действиями в них: для получения комбинированных ходов может задействоваться нажатие сочетания множества кнопок. В зависимости от контекста игры действия могут оказаться перегруженными. В StarCraft и Dota 2 десятки и даже сотни объектов, которыми нужно управлять, но у человека-то всего десять пальцев. Поэтому в игре проектируются обобщенные элементы управления, применимые к разным игровым контекстам и объектам. Но даже в этом случае для таких игр требуется двадцать различных кнопок, и перевод этих сложных систем управления в среды RL затруднителен.

С изобретением смартфонов было создано новое игровое устройство и зародилась индустрия мобильных игр. Игровые состояния и элементы управления переместились на маленький сенсорный экран, где дизайнеры вольны создавать любые виртуальные интерфейсы. Некоторые смартфоны имеют гироскопические датчики, что позволяет использовать устройство в качестве руля в гоночных играх.

Мы достигли нового рубежа технологии развлечений — виртуальной реальности (virtual reality, VR). Уже созданы примитивные контроллеры движений, позволяющие пользователям действовать внутри трехмерной среды с эффектом полного присутствия. Платформы виртуальной реальности, такие как Oculus [97] и HTC Vive [55], используют портативные консоли с датчиками движения. Они также могут применять сенсор на основе обработки визуальной информации Leap Motion [69] для отслеживания движений руки и пальцев, чтобы повысить сложность и интуитивность управления. VR-перчатки — еще один способ слежения за руками — могут быть задействованы для создания пространственных виртуальных объектов и манипулирования ими. С их помощью художники создают великолепные трехмерные виртуальные скульптуры, невозможные в реальном мире. На эти

скульптуры можно смотреть посредством экрана или VR-шлема, подключенных к той же самой виртуальной реальности.

Другая действительно впечатляющая, но менее известная технология — браслет с нейроинтерфейсом от CTRL-Labs [29]. Это бесконтактный браслет, который регистрирует нервные сигналы, посылаемые из головного мозга пользователя к его руке, и затем по ним воссоздает соответствующие руку и палец. Это звучит как научная фантастика, но он уже был продемонстрирован в игре Asteroid из Atari. На момент написания этих строк браслет все еще находится в активной разработке.

Помимо игр и музыки, много креативных структур управления можно найти в современном искусстве — интерактивных инсталляциях. Они приводятся в действие необычным способом — с помощью теней, света, жестов или звука. Художник Даниэль Розин [31] специализируется на создании интерактивных зеркал, в которых используются черные и белые плитки, куклы-тролли и игрушечные пингвины, вращающиеся на большой сетке. Эти объекты по сути являются «пикселами» физического «экрана». Расположенная рядом инфракрасная камера захватывает силуэт человека и отражает его на сетке, как в зеркале. В этом творческом примере с помощью силуэта движущегося человека управляют сеткой из ярких забавных игрушек.

Наконец, что не менее важно, обычные повседневные объекты, с которыми мы взаимодействуем, также способны вдохновить на проектирование действий. Для нас привычны типичные интерфейсы управления, такие как кнопки, наборные диски, ползунки, переключатели и рычаги. Несмотря на то что они, как правило, механические, виртуальные интерфейсы стремятся их сымитировать. В качестве дополнительной литературы по проектированию интерфейсов элементов управления настоятельно рекомендуем книгу Дональда Нормана *The Design of Everyday Things* [94].

16 Вознаграждения

В этой короткой главе изучается проектирование вознаграждений, обсуждаются их роль в задачах RL и некоторые важные проектные решения. В частности рассматриваются масштаб, размер, частота вознаграждения и возможность его использования при разработке сигнала. В конце главы приводится ряд простых рекомендаций по проектированию.

16.1. Роль вознаграждений

Сигналы вознаграждения определяют целевую функцию, которую агент должен максимизировать. Вознаграждение — это скалярная величина, получаемая из среды и присваивающая коэффициент доверия конкретному переходу s, a, s', который произошел в связи с действием a агента.

Проектирование вознаграждений — одна из основных проблем в RL, известная своей сложностью по нескольким причинам. Чтобы интуитивно правильно присваивать коэффициенты доверия, то есть судить, какой переход был оптимальным (положительное вознаграждение), нейтральным (нулевое вознаграждение) или неудачным (отрицательное вознаграждение), необходимо углубленное знание среды. Даже если мы приняли решение о знаке вознаграждения, нам все еще нужно выбрать его величину.

Если одному переходу присвоено вознаграждение +1, а другому — вознаграждение +10, то, грубо говоря, последний в десять раз важнее первого. Но зачастую неясно, как определять этот масштаб. Поскольку агент, который обучается, применяя в качестве подкрепляющих сигналов вознаграждения, воспринимает их буквально, то и структура вознаграждений должна быть буквальной. Более того, агент может научиться злоупотреблять вознаграждениями, найдя неожиданную стратегию их использования, не проявляя задуманного поведения. В результате, чтобы агент стал вести себя корректно, структуру вознаграждений потребуется часто настраивать, а это подразумевает много ручной работы.

Люди зачастую опираются на интуицию, когда речь идет об оценке вклада их действий в достижение цели. Такая своего рода внутренняя способность назна-

чать коэффициенты доверия складывается из опыта и знаний. С этого хорошо начинать при проектировании вознаграждений, стимулирующих правильное поведение агента.

Вознаграждение — это обратный сигнал, информирующий агента о том, насколько хорошо или плохо он действует. Агент обычно обучается с нуля, у него нет никаких предварительных знаний или здравого смысла, поэтому он практически не ограничен в исследовании действий. Сигнал вознаграждения, оценивающий лишь результат, но не выполнение задания, не регламентирует наблюдаемые типы поведения агента независимо от того, насколько неадекватными они могут быть. Это обусловлено тем, что задача часто неуникальная — существует много путей, ведущих к множеству решений. Однако с человеческой точки зрения такие ненормальные типы поведения нежелательны и рассматриваются как ошибочные.

Можно попытаться исправить это, наложив некоторые условия на агента, однако даже серьезные ограничения свободы действий могут оказаться недостаточными. Ограниченность нашей мускулатуры помогает нам ходить оптимальным образом, но если мы захотим, наша походка может быть странной. Альтернатива — проектирование вознаграждений. Так же как люди могут ходить естественно или неестественно в зависимости от своих целей, вознаграждения можно спроектировать так, чтобы стимулировать желаемое поведение агента, по крайней мере в теории. Если бы нам пришлось в точности воспроизвести необходимый сигнал вознаграждения для человекоподобной походки, можно представить, насколько это было бы сложным, учитывая то, как много компонентов должны правильно функционировать для достижения поставленной задачи.

Для проектирования оптимального сигнала вознаграждения нужно определить, какие типы поведения желательны, затем присвоить им соответствующие вознаграждения. При этом следует проявлять осмотрительность, чтобы не исключить другие возможные хорошие действия. Затем следует протестировать и оценить агентов, чтобы проверить приемлемость разработанного сигнала вознаграждения. Для тривиальных игровых задач это выполнимо, вместе с тем остается неясным, насколько это возможно или практично для более сложных сред.

Как бы то ни было, хорошо спроектированные сигналы вознаграждения для конкретных задач могут позволить нам продвинуться довольно далеко. Это было продемонстрировано недавними достижениями в RL, такими как Dota 2 от OpenAI [104] и робототехнические манипуляторы [101]. В этих примерах функции вознаграждений были тщательно настроены людьми, и агенты смогли достичь впечатляющих результатов. Проектирование вознаграждений было важной частью успешного обучения агентов. Эти результаты наводят на мысль о том, что порог сложности, после которого ручное проектирование вознаграждений становится неприемлемым, довольно высок. Возникает естественный вопрос: существуют ли проблемы, для которых невозможно разработать оптимальные сигналы

вознаграждения? Если нет, то проектирование хороших сигналов вознаграждения было бы возможным для задания любой сложности, хотя оно может оказаться трудным и времязатратным. Рассмотрим с учетом сказанного некоторые практические рекомендации по проектированию вознаграждений.

16.2. Рекомендации по проектированию вознаграждений

Сигнал вознаграждения может быть разреженным или плотным. В первом случае на большинстве временных шагов выдается сигнал нейтрального вознаграждения (обычно $r = 0$), а положительное или отрицательное вознаграждение — только когда среда завершается или действие агента имеет решающее значение. Во втором случае, наоборот, предоставляется много не нейтральных сигналов вознаграждения, показывающих, было последнее действие хорошим или плохим. И агент таким образом получает положительное или отрицательное вознаграждение на большинстве временных шагов.

Соответствие значения вознаграждения классу «хорошее», «нейтральное» или «плохое» относительно, то есть числовая шкала для вознаграждений — это проектное решение. Именно поэтому стандартизация вознаграждений во время обучения агента оправданна. Среда может быть спроектирована так, что все значения вознаграждений отрицательные, а плохие вознаграждения еще более отрицательные, чем хорошие. Поскольку агент всегда максимизирует целевую функцию, значения вознаграждений все равно нужно упорядочивать так, чтобы у лучших вознаграждений были более высокие значения.

Хотя это и не обязательно, с математической точки зрения целесообразно установить для нейтральных вознаграждений значение 0, для плохих — отрицательные значения, а для хороших — положительные. Такая структура проста для понимания, к тому же обладает хорошими математическими свойствами. Например, если шкала вознаграждений центрирована относительно 0, то можно легко изменить ее масштаб как для положительной, так и для отрицательной сторон путем умножения на скалярную величину.

Вознаграждение определяется как $r_t = \mathcal{R}(s, a, s')$, поэтому на первом шаге проектирования вознаграждений выясняется, в чем состоят хорошие или плохие переходы внутри среды. Поскольку в переход включено действие a, вызвавшее переход из состояния s в s', полезно систематически пронумеровать все возможные переходы. Однако в сложных средах переходов может быть слишком много, чтобы их перечислить, поэтому нам нужны более разумные методы проектирования вознаграждений.

Если это возможно, разделите переходы в соответствии с правилами на хорошие и плохие, затем присвойте вознаграждения на основании этих правил. Во многих

заданиях можно просто присвоить коэффициент доверия, исходя из состояния s', в которое произошел переход, без учета предыдущего состояния. Например, в CartPole вознаграждение 0,1 присваивается всем состояниям, где стержень не падает, независимо от того, какие действия предпринимались для его удержания в вертикальном положении. В то же время вознаграждения можно давать, основываясь лишь на действии. Например, в среде LunarLander от OpenAI ценой каждого зажигания основного двигателя является небольшое отрицательное вознаграждение.

К счастью, вознаграждения широко распространены в играх, хотя называются по-другому — счет игры. Учет очков характерен для игр по той причине, что в них всегда есть победитель или задание, которое нужно выполнить. Очки могут предоставляться в явной числовой форме, обычно с отображением на экране, или в неявной нечисловой форме, такой как объявление победы или проигрыша. Снова мы можем обратиться к играм за идеями и методами для проектирования вознаграждений.

Простейшая форма вознаграждения — бинарная: победа или проигрыш, что может быть закодировано как 1 и 0 соответственно. Один из примеров — шахматы. Кроме того, среднее вознаграждение за несколько раундов такой игры прекрасно преобразуется в вероятность выигрыша. В ролевых сюжетных играх обычно нет счета, но мы по-прежнему можем добавить бинарный счет за преодоление каждого этапа, чтобы стимулировать окончание игры.

Другая форма вознаграждения — одиночное скалярное значение, которое постепенно увеличивается по мере того, как игрок накапливает больше очков. Во многих играх есть цели, которых нужно достичь, объекты, которые требуется собирать, и этапы, которые следует пройти. За каждую из этих целей и каждый из объектов присваиваются очки, по мере прохождения игры они суммируются в окончательный счет, по которому ранжируется успех всех игроков. Такие накопительные вознаграждения показываются в играх от простых до средней сложности. Это связано с тем, что до сих пор целесообразным остается присваивать очки за все значимые игровые элементы, обеспечивая при этом, чтобы их сумма корректно отображала конечную цель игры.

Наконец, в больших и сложных играх могут отслеживаться вспомогательные счета, но их не обязательно суммировать с конечной целевой функцией игры. В играх-стратегиях реального времени наподобие StarCraft и Command & Conquer отслеживаются счета, включающие здания, исследования, ресурсы и юниты, но они не обязательно соотносятся с конечным результатом игры. В Dota 2 вспомогательные счета, такие как последние удары, золото и полученный опыт, тщательно отслеживаются на протяжении игры, но в конце побеждает команда, которая разрушила вражеский трон. Эти вспомогательные счета часто помогают проинформировать о прогрессе в долго длящейся игре, показывая преимущество одной стороны перед второй, и отражают конечный результат. Однако при благоприятных показателях

победа не всегда гарантирована, поскольку игрок с меньшими вспомогательными счетами все равно может выиграть. Следовательно, вспомогательные счета — это набор вознаграждений, отдельный от целевой функции игры. Несмотря на это, они все же полезны в качестве действительных сигналов вознаграждения, так как могут помочь агенту обучаться. OpenAI Five [104] используют сочетание вспомогательных счетов с конечной целевой функцией как сигнал вознаграждения для Dota 2.

Для игр с победой или проигрышем, таких как шахматы, трудно спроектировать хорошее промежуточное вознаграждение. Причина этого заключается в том, что при астрономическом количестве вариантов расстановки фигур объективно сложно определить, насколько хорош тот, который находится на доске. Конечное вознаграждение просто составляет 1 для победителя и 0 для проигравшего, а промежуточное вознаграждение нулевое — это разреженный сигнал вознаграждения.

Напрашивается вопрос: когда задание определено и нам известен желаемый конечный результат, который может легко быть присвоен в качестве вознаграждения, почему бы просто не придерживаться разреженного вознаграждения, как в шахматах?

Существует компромисс между разреженными и плотными вознаграждениями. Хотя разреженный сигнал легко указать, по нему намного труднее учиться, так как из среды обратно поступает намного меньше информации. Агент должен ждать завершения задания, чтобы получить сигнал вознаграждения, но даже тогда у него нет простого способа сказать, какое из действий на промежуточных шагах было хорошим, а какое — плохим. Разреженные вознаграждения сильно снижают эффективность выборки в задаче, поэтому агенту для обучения потребуется на порядок больше прецедентов. Порой разреженность вознаграждений может стать причиной неразрешимости задачи ввиду того, что агенту поступает так мало информации, что он не способен обнаружить результативную последовательность действий. При плотных вознаграждениях ситуация обратная. Хотя промежуточные вознаграждения может быть трудно указать, обратная информация из среды поступает незамедлительно, поэтому агент чаще получает сигналы, по которым обучается.

Обычно применяются комбинированные плотные и разреженные вознаграждения, а их веса варьируются с течением времени так, чтобы постепенно переходить от первых ко вторым. Определим комбинированное вознаграждение в уравнении (16.1):

$$r = \delta r_{\text{dense}} + r_{\text{sparse}}, \tag{16.1}$$

где $\delta = 1 \rightarrow 0$.

В ходе обучения агента значение коэффициента δ постепенно уменьшается с 1 до 0. Небольшая деталь проектирования — мы исключили из r_{dense} конечное вознаграждение, чтобы не учитывать его дважды.

Плотные вознаграждения на ранней фазе обучения передают агенту много обратной информации, что может помочь ему с исследованием среды. Предположим, что цель агента в роботизированном задании — подойти к флагу на открытой местности. Конечное вознаграждение, получаемое в конце эпизода, — это просто 1, если он достиг флага, и −1, если нет. Пусть расстояние от агента до флага d (агент, флаг), тогда полезным плотным немедленным вознаграждением может быть отрицательное расстояние $r_{\text{dense}} = -d(\text{агент, флаг})$. Вначале это плотное вознаграждение помогает научить агента тому, что совершаемые им перемещения изменяют расстояние до флага и минимизировать это расстояние хорошо с точки зрения целевой функции. Исходя из этого, он обучается стратегии минимизации расстояния. После того как агент немного обучится, можно убрать дополнительное слагаемое, снизив значение δ до 0. На момент отключения r_{dense} та же самая стратегия будет способна правильно функционировать, используя лишь разреженные вознаграждения.

Чтобы понять, насколько важными могут быть плотные вознаграждения, просто представьте себе то же самое задание, но лишь с разреженными вознаграждениями. Вероятность того, что агент, случайно блуждающий по обширному двумерному пространству, когда-либо достигнет флага, чрезвычайно мала. Большую часть времени он будет терпеть неудачи. Даже редких случаев успеха будет слишком мало для обучения, не говоря уже об открытии концепции минимизации расстояния. Вероятность того, что агент научится решать эту задачу даже за длительное время, низкая.

Другой продвинутый прием решения проблемы разреженных вознаграждений — *перераспределение вознаграждений* (reward redistribution). Суть ее в том, чтобы взять конечное вознаграждение, разделить его на части и перераспределить некоторые из них для присвоения коэффициентов доверия значимым событиям на промежуточных шагах. Если разработчику вознаграждений известны такие значимые события, то это довольно просто, но не всегда их можно легко определить. Одно из решений — оценить их, собрав большое количество траекторий, выделив распространенные шаблоны и связав их с конечными вознаграждениями. Это форма автоматического присвоения коэффициентов доверия, ее примером является RUDDER [8], разработанный LIT AI Lab.

Противоположность перераспределения вознаграждений — *отложенное вознаграждение* (reward delay), при котором все сигналы вознаграждения откладываются на несколько временных шагов. Например, так происходит при пропуске кадров. Важно, чтобы вознаграждения из пропущенных кадров не терялись — полное вознаграждение должно оставаться равным вознаграждению в исходной среде. Это достигается за счет хранения вознаграждений из промежуточных кадров и их последующего суммирования с целью получения вознаграждения для следующего выбранного кадра. Суммированием обеспечивается и то, что сигнал

вознаграждения обладает линейностью, благодаря которой он может работать при любой частоте пропуска кадров без изменения целевой функции.

При проектировании сигнала вознаграждения следует учитывать его профиль распределения. Во многих алгоритмах вознаграждение используется в расчете функции потерь. Большой размер вознаграждения может привести к огромным значениям функции потерь и вызвать резкий рост градиентов. Создавая функцию вознаграждений, избегайте предельных значений. Таким образом, целесообразно придавать сигналу такую структуру, чтобы статистические данные были разумными — стандартизированными, с нулевым средним значением и без предельных значений. Мало того что на излишне сложную структуру вознаграждений уйдет много времени и сил, так она еще и затруднит отладку среды и агента. При этом нет никакой уверенности в том, что она будет работать значительно лучше, чем более простая структура. Сложный сигнал вознаграждения зачастую является результатом чрезмерной концентрации на отдельных сценариях, поэтому маловероятно, что их можно будет обобщить при изменениях в среде.

Еще одна деталь проектирования сигнала вознаграждения, о которой нужно знать, — это *аренда вознаграждений* (reward farming) или *взлом вознаграждений* (reward hacking). В сложных средах трудно предвидеть все возможные сценарии. Ситуация, когда игрок находит в видеоигре вредоносный код или лазейку и постоянно ими злоупотребляет для получения чрезвычайно больших вознаграждений, рассматривается как ошибка, даже если это разрешено в рамках игры. Именно это произошло, когда основанный на эволюционной стратегии агент, созданный группой исследователей из Фрайбургского университета, сумел обнаружить ошибку в игре Qbert из Atari [23]. Он выполнил специфическую последовательность действий, которая включила призовую анимацию, неограниченно увеличивающую количество очков в игре. Видео под названием *Canonical ES finds a bug in Qbert (Full)* вы найдете на YouTube, перейдя по ссылке https://youtu.be/meE5aaRJ0Zs [22].

Взлом вознаграждения в видеоиграх может быть веселым и занимательным, но в прикладных приложениях он может оказаться потенциально вредоносным и имеющим серьезные последствия. Встреча системы RL, управляющей промышленным аппаратным обеспечением, с вредоносным кодом с неуправляемым процессом, может вызвать поломку дорогостоящего оборудования или травмирование людей. Так как назначение коэффициентов доверия тесно связано с определением поведения агента и его управлением, оно является одним из основных предметов исследования в безопасности ИИ. Хорошее введение в данную тему можно найти в статье *Concrete Problems in AI Safety* [4]. Нужно ответственно подходить к проектированию сред и обучению агентов, чтобы избежать негативных побочных эффектов при развертывании системы RL в реальных условиях. Это позволит предотвратить взлом вознаграждений, обеспечить контроль, стимулировать безопасные исследования и гарантировать устойчивость системы.

В случае взлома сигнал вознаграждения рассматривается как ошибочный [100], поэтому проектировщик должен исправить ошибку в среде или перепроектировать часть этого сигнала. Нет верного способа узнать, в каком месте может произойти взлом вознаграждения, при поиске нужно полагаться на наблюдения в тестовой среде. Один из способов сделать это — записать все вознаграждения, полученные во время обучения, и проанализировать их. Вычислите среднее значение, моду, среднее квадратическое отклонение, затем просмотрите их на предмет предельных или отклоняющихся от нормы значений. Если обнаружены какие-нибудь предельные значения, определите соответствующие сценарии и проверьте их вручную, чтобы увидеть, каким было поведение агента, породившее такие необычные значения вознаграждения. Для упрощения отладки можно записать видео с примерами этих сценариев или просто наблюдать среду вживую, если существуют способы воспроизведения ошибочных сценариев.

Мы в некоторой степени охватили основы проектирования вознаграждений. Подводя итоги, приведем несколько факторов, которые следует учитывать при проектировании сигнала вознаграждения.

- **Используйте хорошие, нейтральные и плохие значения вознаграждений.** Целесообразно начать с применения положительных значений как хороших, затем перейти к нулевым — как нейтральным и отрицательным — как плохим. Внимательно относитесь к масштабу, избегайте предельных значений.
- **Выбирайте разреженные или плотные сигналы вознаграждений.** Первые легко проектировать, но часто они сильно усложняют задачу, вторые трудно проектировать, но они дают агенту намного больше обратной информации.
- **Следите за безопасностью и вероятностью взлома вознаграждений.** Постоянно оценивайте агентов, чтобы удостовериться, что взлома вознаграждения не происходит. Нужно ответственно обучать агента и проектировать среду, чтобы обеспечить безопасность развернутой в реальных условиях системы.

Сигнал вознаграждения можно рассматривать в качестве посредника для передачи накопленных человеком знаний, присущих ему ожиданий и здравого смысла агенту, который всем этим не обладает. На текущий момент в RL агент понимает и видит задание совсем не так, как мы, люди. Его единственной целью является максимизация целевой функции, а все типы его поведения проистекают из сигналов вознаграждения. Для поощрения конкретного поведения нужно понять, каким образом присваивать вознаграждения, чтобы они проявлялись как побочный эффект максимизации целевой функции. Однако не всегда возможно идеально спроектировать вознаграждения. Агент может неверно истолковать наши намерения и решить задачу совсем не так, как мы хотим, даже если при этом продолжает максимизировать полное вознаграждение. Поэтому для проектирования оптимальных сигналов вознаграждения необходимо знание человеком предметной области.

16.3. Резюме

В этой главе о проектировании вознаграждений обсуждались баланс между разреженными и плотными сигналами вознаграждения, а также важность масштаба вознаграждений. Тщательно спроектированная функция вознаграждений может быть весьма эффективной, как показали результаты OpenAI в Dota 2. Оптимальный сигнал вознаграждения не должен быть слишком обобщенным или конкретизированным и должен стимулировать желаемое поведение агента. Тем не менее ошибочная структура вознаграждений может вызывать неожиданные типы поведения. Это известно как взлом вознаграждений и может представлять угрозу для системы, развернутой в условиях промышленной эксплуатации. Отсюда следует, что при развертывании системы RL в реальных условиях нельзя забывать об обеспечении безопасности.

17 Функция переходов

После состояний, действий и вознаграждений осталось рассмотреть последний компонент, необходимый для функционирования среды RL, — функцию переходов, также известную как модель.

Модель среды может быть запрограммирована или настроена. Правила программирования общие и могут позволить создать среды с различными степенями сложности. Шахматы прекрасно описываются простым набором правил. При моделировании роботов аппроксимируют динамику робота и его окружение. Современные компьютерные игры могут быть очень сложными, но их продолжают создавать с помощью программных игровых движков.

Однако при моделировании задач, которые невозможно эффективно запрограммировать, модель среды можно настроить. Например, моделирование контактного взаимодействия в робототехнике с помощью правил программирования затруднительно, поэтому в качестве альтернативы его пытаются настроить по наблюдениям из реального мира. Для прикладных задач, где состояния и действия — абстрактные величины, сложные для понимания и моделирования, но для которых доступно много данных о переходах, функцию переходов можно настроить. У сред могут быть также гибридные модели, которые частично запрограммированы, а частично настроены.

В этой главе приводятся некоторые рекомендации по проверке осуществимости построения функции переходов и оценке того, насколько близко она аппроксимирует реальную задачу.

17.1. Проверка осуществимости

Как вы помните, функция переходов задана как $P(s_{t+1} | s_t, a_t)$ и обладает марковским свойством, что означает: переход полностью определяется текущими состоянием и действием. В теории нужно рассмотреть, можно ли построить модель в такой форме. На практике сначала анализируют, является ли потенциально возможным построение модели с помощью правил, физического движка, игрового движка или набора данных.

В этом разделе представлены несколько правил проверки на осуществимость, которые нужно выполнить до построения функции переходов. Требуется рассмотреть следующее.

- **Программирование или обучение.** Можно ли построить среду RL без использования данных, путем выяснения и программирования правил переходов? Возможно ли полное описание задачи набором правил? Примерами являются настольные игры, такие как шахматы, и физические системы, такие как роботы. Если это невозможно или непрактично, то модель можно лишь настроить на основе данных.
- **Полнота данных.** Если модель нужно настраивать по данным, то доступны ли они в достаточном количестве? То есть достаточно ли данные репрезентативны? Если нет, можно ли собрать больше данных или возместить недостающие? Все ли данные являются полностью наблюдаемыми? Если нет, модель может оказаться неточной, каков в этом случае приемлемый порог погрешности?
- **Затратность данных.** Иногда порождение данных может быть затратным: процесс занимает много времени или оказывается недешевым по той причине, что сбор реальных данных подразумевает длительные циклы и дорогостоящие ресурсы. Например, реальный робот дорогой и двигается медленнее, чем его модель, но для создания реалистичной модели может потребоваться сначала накопить данные о действительных движениях робота.
- **Эффективность выборки данных.** Для глубокого RL по-прежнему характерны низкая эффективность выборки и плохая обобщающая способность. Это значит, что модель задачи должна иметь высокую достоверность, что, в свою очередь, требует большого количества данных для построения модели и увеличивает стоимость ее обучения. Даже если функция переходов программируемая, на получение реалистичной модели все равно может быть затрачено много времени и сил. Например, у видеоигр самые сложные программируемые функции переходов, получение которых может стоить миллионы долларов.
- **Обучение по актуальному или отложенному опыту.** Когда данные собирают в отрыве от действительного агента RL, это рассматривается как обучение по отложенному опыту. Если модель строится на данных, полученных при обучении по отложенному опыту, будет ли этого достаточно, чтобы учесть влияние взаимодействия с агентом? Например, обучающийся агент может изучить новые части пространства состояний, из-за чего потребуются новые данные для заполнения пробелов в модели. Здесь необходим онлайн-подход, то есть развертывание агента в реальных условиях для взаимодействия с действительной задачей для сбора дополнительных данных. Возможно ли это?
- **Обучение или промышленная эксплуатация** (в связи с предыдущим пунктом). Прежде чем можно будет использовать агента, обученного с помощью настроенной модели, его нужно оценить с учетом действительной задачи. Безопасно ли тестирование агента в условиях промышленной эксплуатации? Каким образом

можно гарантировать, что его поведение не выйдет за разумные пределы? Если неожиданное поведение проявится в промышленной эксплуатации, то каким может оказаться финансовый ущерб?

Если эти проверки на осуществимость пройдены, то можно приступить к конструированию функции переходов. При построении модели нужно учесть несколько характеристик задачи, которыми определяются подходящие методы.

- **Детерминированные задачи с известными правилами переходов** — например, шахматы. На основе известных правил, например с помощью хранимого словаря, строится детерминированная функция отображения. Это может быть записано как $s_{t+1} \sim P(s_{t+1} \mid s_t, a_t) = 1$ или, что равнозначно, как прямая функция, где $s_{t+1} = f_{\text{deterministic}}(s_t, a_t)$. При построении модели можно использовать существующие движки с открытым исходным кодом или коммерческие инструменты. Есть множество превосходных игровых и физических движков, таких как Box2D [17], PyBullet [19], Unity [138] и Unreal Engine [139]. Их можно применять для построения ряда сред RL, от простых двумерных игрушечных задач до высокореалистичных автомобильных тренажеров.
- **Стохастические (недетерминированные) задачи с известной динамикой** — например, реалистические роботизированные модели. Здесь часть динамики стохастическая по своей природе, а остальное — это детерминированная динамика с добавлением случайных шумов. Например, физическая система может быть смоделирована по детерминированным правилам. Но чтобы сделать ее более реалистичной, принято учитывать случайные шумы, такие как трение, вибрации или шум датчика. В этом случае модель переходов имеет вид $s_{t+1} \sim P(s_{t+1} \mid s_t, a_t)$.
- **Стохастические задачи с неизвестной динамикой** — например, сложное управление запасами или оптимизация продаж со множеством ненаблюдаемых и непредсказуемых компонентов. Поскольку динамика неизвестна, ее нужно настроить по данным. Собираются все доступные данные в форме ..., s_t, a_t, s_{t+1}, a_{t+1}, ... и строится гистограмма, чтобы зафиксировать частоту s_{t+1} для заданных s_t, a_t. Затем гистограмме сопоставляется тип распределения: гауссово, Бернулли, бета-распределение и т. д. Таким образом получают распределения вероятностей в виде $P(s_{t+1} \mid s_t, a_t)$ — настроенной модели. Этот процесс можно преобразовать в задание обучения с учителем по настройке распределения вероятностей по данным с s_t, a_t в качестве входных значений и s_{t+1} — выходного целевого значения.
- **Задачи, которые не подчиняются марковскому свойству**, например сложные видеоигры. В этой ситуации для определения функции переходов требуются длительные временные горизонты. Есть две распространенные стратегии. Если функция переходов программируемая, концентрируются на построении реалистичной модели, которая не обязательно является марковской. Затем принимается решение, должны ли состояния, показываемые агенту, быть марковскими. Если важно, чтобы было именно так, может потребоваться повторно спроектировать состояние, дабы включить достаточно информации. В противном случае

состояния могут быть проще, тогда для агента задача становится частично наблюдаемым МППР. Если функция переходов настраивается, мы не можем позволить себе такую роскошь, поскольку данные s_t, a_t для обучения должны содержать достаточно информации, чтобы полностью определить следующий переход. В этом случае основная стратегия — переопределить состояние, чтобы оно стало марковским.

Как убедиться, что построенная модель является достаточно реалистичным представлением задачи? Это подводит нас к теме следующего раздела.

17.2. Проверка реалистичности

Любая модель, аппроксимирующая реальное явление, скорее всего, несовершенна. Следовательно, нужны методы оценки того, насколько хороша полученная аппроксимация. В этом разделе сначала обсуждаются некоторые из источников ошибок модели, а затем расстояние Кульбака — Лейблера как количественная оценка погрешности.

Можно выделить две основные причины несовершенства модели.

В первом случае полное моделирование каждого из аспектов задачи может оказаться неосуществимым, в связи с чем выполняются упрощения. Например, при моделировании роботизированной руки могут не учитываться трение, вибрации, тепловое расширение и деформации, вызванные столкновениями ее физических частей, которые существуют в реальном мире. На практике некоторые данные просто недоступны. Возьмем в качестве примера рекомендуемые фильмы. Полезно знать жанры, которые нравятся человеку, но получить эти сведения напрямую невозможно, их можно лишь вывести, основываясь на ранее просмотренных фильмах.

Второй случай обусловлен ограниченностью исследования среды при настроенной функции переходов. Здесь проблема в том, что некоторые из переходов могут оставаться непосещенными до того, как алгоритм будет развернут в реальных условиях для взаимодействия со средой. Это происходит, когда пространство состояний настолько велико, что настройка хорошей модели для всех состояний практически нецелесообразна. Вместо этого обычно сосредотачиваются на настройке оптимальной модели для переходов, которые, вероятнее всего, встретятся агенту. Если модель не учитывает переходы, с которыми агент сталкивается в промышленной эксплуатации, то, естественно, она может выполнять неточные переходы. Но это не означает, что агент не может обучаться на ограниченной модели.

Модель можно настраивать по данным циклически следующим образом. Сначала накапливают поднабор данных из реального мира и используют их для настройки модели. Она будет аппроксимировать реальность с некоторой погрешностью. Затем можно обучить агента, развернуть его в условиях промышленной эксплуатации, накопить больше данных о переходах и повторно настроить модель, чтобы

улучшить ее. Этот процесс повторяется до тех пор, пока ошибка не снизится до приемлемого порогового значения.

Теперь необходимо ввести понятие ошибки между настроенной моделью и задачей, которую она аппроксимирует. Определим функции переходов при обучении и промышленной эксплуатации как $P_{\text{train}}(s' \mid s, a)$ и $P_{\text{prod}}(s' \mid s, a)$ соответственно. Мы получили два распределения вероятностей и можем измерить разницу между ними с помощью стандартных методов.

Расстояние Кульбака — Лейблера[1] широко применяется для измерения того, насколько одно распределение вероятностей отклоняется от другого. Предположим, нам даны два распределения $p(s)$, $q(s)$ случайных значений переменной $s \in S$. Пусть $p(s)$ — реальное распределение, а $q(s)$ — его аппроксимация. Используя расстояние КЛ, можно определить, насколько $q(s)$ отклоняется от реального $p(s)$.

Расстояние КЛ для дискретной переменной приведено в уравнении (17.1):

$$\text{КЛ}\big(p(s) \,\|\, q(s)\big) = \sum_{s \in S} p(s) \log \frac{p(s)}{q(s)}. \tag{17.1}$$

Выражение в обобщенной форме для непрерывной переменной получено простым преобразованием суммы дискретных значений в непрерывный интеграл, как показано в уравнении (17.2):

$$\text{КЛ}\big(p(s) \,\|\, q(s)\big) = \int_{-\infty}^{\infty} p(s) \log \frac{p(s)}{q(s)} ds. \tag{17.2}$$

Расстояние КЛ — это неотрицательное число. Когда оно равно 0, аппроксимирующее распределение q не отклоняется от реального распределения p, то есть два распределения равны. Чем больше расстояние КЛ, тем больше отклонение q от p. Этот показатель к тому же асимметричен, то есть $\text{КЛ}(p \,\|\, q) \neq \text{КЛ}(q \,\|\, p)$ в общем случае. Помимо этого, расстояние КЛ из-за своих полезных свойств применяется в ряде алгоритмов RL для измерения отклонений между стратегиями на разных итерациях, как мы видели в главе 7.

Расстояние КЛ также может быть истолковано как потеря информации при использовании $q(s)$ для аппроксимации реальности $p(s)$. В нашем случае реальность — это $P_{\text{prod}}(s' \mid s, a)$, полученная при промышленной эксплуатации, а аппроксимация — $P_{\text{train}}(s' \mid s, a)$, полученная при обучении. Кроме того, P_{prod} и P_{train} — условные вероятности, но расстояние КЛ можно просто подсчитать для каждого конкретного экземпляра условных переменных. Пусть $p = P_{\text{prod}}$, $q = P_{\text{train}}$, а расстояние $\text{КЛ}_{s,a}$ между ними записано в уравнении (17.3):

$$\text{КЛ}_{s,a}\Big(P_{\text{prod}}\big(s' \mid s, a\big) \,\|\, P_{\text{train}}\big(s' \mid s, a\big)\Big) = \sum_{s' \in S} P_{\text{prod}}\big(s' \mid s, a\big) \log \frac{P_{\text{prod}}\big(s' \mid s, a\big)}{P_{\text{train}}\big(s' \mid s, a\big)}. \tag{17.3}$$

[1] Известно также как относительная энтропия.

Это дискретная форма записи, при необходимости получения формы для непрерывных переменных преобразуйте сумму в интеграл.

Уравнение (17.3) применимо к одиночной паре (s, a), однако нам нужно оценить расстояние КЛ для модели в целом по всем парам (s, a). На практике для этого используют прием, часто встречающийся в книге, — выборку методом Монте-Карло.

Чтобы улучшить модель с его помощью, рассчитывайте и отслеживайте расстояние КЛ на каждой итерации обучения модели и ее развертывания для эксплуатации. Тем самым обеспечивается то, что P_{train} не будет слишком сильно отклоняться от P_{prod}. По прошествии множества итераций обучения целью должно стать уменьшение расстояния КЛ до приемлемого порогового значения.

В этой главе мы показали, как построить $P(s' \mid s, a)$ для среды с учетом требования ее реалистичности. Теперь, имея на вооружении инструментарий для выполнения в почти буквальном смысле проверки реалистичности, можно перейти к более практическим вопросам. Каковы распределения данных, доступные для обучения и промышленной эксплуатации, и в чем различие между ними? Как будет устраняться разрыв между обучением и промышленной эксплуатацией? Эти вопросы особенно важны для промышленных приложений, где данные часто ограничены и их трудно получить, так что они будут представлять лишь поднабор из полного распределения переходов для действительной задачи. Доступные данные повлияют на качество модели, что, в свою очередь, скажется на обучении алгоритма RL. Чтобы алгоритмы RL были применимы за рамками обучения, нужно обобщить их за пределами данных для обучения при развертывании для промышленной эксплуатации. Если разрыв между данными для обучения и промышленной эксплуатации слишком велик, агент может потерпеть неудачу на наборе, отличном от обучающего. Возможно, для уменьшения этого разрыва понадобится итеративное улучшение модели.

17.3. Резюме

В этой главе мы рассмотрели функции переходов, также известные как модели сред. Модель может быть запрограммирована с помощью правил или настроена по данным. Мы представили список проверок осуществимости программирования или настройки модели. Затем рассмотрели различные формы, которые может принимать функция переходов. В конце было представлено расстояние Кульбака — Лейблера в качестве метода оценки погрешности построенной модели относительно реального распределения переходов.

Заключение

В начале книги задача RL была сформулирована как МППР. В частях I и II введены основные семейства алгоритмов глубокого RL, применимые к решению МППР — основанные на стратегии и полезности и комбинированные методы. В части III основное внимание уделялось практическим аспектам обучения агентов с охватом таких тем, как отладка, архитектура нейронной сети и аппаратное обеспечение. В нее также вошел справочник по глубокому RL, содержащий сведения о гиперпараметрах и производительности алгоритмов для отдельных задач классического управления и некоторых сред Atari из OpenAI Gym.

Проектирование сред стало подходящим завершением книги, поскольку это важная часть практического применения глубокого RL. Если нет сред, то и агенту нечего решать. Проектирование сред — обширная и интересная тема, поэтому мы смогли лишь кратко затронуть некоторые из важных идей. Часть IV следует понимать не как глубокий и детальный обзор темы, а как ряд рекомендаций высокого уровня.

Надеемся, книга послужила полезным введением в глубокое RL и хоть отчасти передала те вдохновение, любопытство и признательность, которые мы испытываем, изучая эту тему и принимая участие в ее развитии. Надеемся, что она подогрела и ваш интерес и вам стало любопытно больше узнать о глубоком RL. Так что мы завершаем книгу кратким упоминанием ряда открытых исследовательских вопросов с приведением там, где это необходимо, ссылок на последние работы в данных областях. Это не полный перечень, но отправная точка в бесконечно интересной и быстро меняющейся области исследований.

Воспроизводимость

На текущий момент алгоритмы глубокого RL известны нестабильностью, чувствительностью к гиперпараметрам и высокой дисперсией получаемого решения, что делает результаты трудновоспроизводимыми. В статье Хендерсона и др. *Deep Reinforcement Learning that Matters* содержится превосходный анализ этих проблем и предлагается несколько возможных способов решения. Один из них — лучше воспроизводимый рабочий процесс, к применению которого мы старались подтолкнуть в этой книге и прилагаемой к ней библиотеке SLM Lab.

Отрыв от реальности

Он известен также как проблема перехода из симуляционной среды в реальную (simulation-to-reality [sim-to-real] transfer problem). Нередко обучать агента глубокого RL в реальных условиях сложно из-за высокой стоимости, значительных временных затрат и требований безопасности. Зачастую агента обучают в виртуальной среде и разворачивают в реальных условиях. К несчастью, получить точную виртуальную модель реального мира трудно — это называется отрывом от реальности и порождает проблему переноса модели, связанную с обучением агента на распределении данных, отличном от тестовых данных.

Один из распространенных подходов к решению данной проблемы заключается во введении случайных шумов в модель при обучении. В *Learning Dexterous In-Hand Manipulation* от OpenAI [98] для этого задаются случайные значения физических свойств модели, таких как масса объекта, его цвет и трение.

Метаобучение и многозадачное обучение

Одно из ограничений алгоритмов из этой книги связано с тем, что они при выполнении каждого задания обучаются с нуля, что весьма неэффективно. Многие задачи естественным образом взаимосвязаны. Например, пловцы соревнуются во множестве стилей, так как эти техники тесно связаны друг с другом. При обучении новому стилю спортсменам не приходится каждый раз учиться плавать с нуля. Более общий пример — когда люди растут, они учатся управлять своим телом и применяют эти знания к овладению множеством разных двигательных навыков.

Метаобучение — это подход к решению данной задачи путем обучения тому, как нужно учиться. Алгоритмы проектируются так, чтобы учиться тому, как научиться выполнять новое задание эффективно. Их обучение происходит на наборе взаимосвязанных обучающих заданий. Один из примеров — статья Финна и др. *Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks* [40].

Многозадачное обучение связано с более эффективным обучением выполнению множества заданий путем их совместного изучения. Примеры данного подхода — алгоритм Distral[1] от Теха и др. [134] и алгоритм актора-имитатора[2] от Паризотто с коллегами [110].

---

[1] Название образовано от слов distill («общий») и transfer learning («трансферное обучение»). Это метод параллелизации, при котором рабочие сети обмениваются между собой не параметрами, а стратегией, вобравшей в себя поведение, общее для всех заданий. — *Примеч. пер.*

[2] В методе актора-имитатора одна стратегическая сеть (имитатор) учится выполнять ряд различных заданий под руководством нескольких экспертов (акторов). Обученный экземпляр сети может в дальнейшем быть применен к новым задачам без каких-либо предварительных экспертных указаний. — *Примеч. пер.*

Многоагентные задачи

Применение RL к многоагентным задачам представляется вполне естественным. Пример тому — OpenAI Five [104], где с помощью алгоритма PPO пять отдельных агентов научились играть в сетевую игру Dota 2. При простой настройке коэффициентов, уравновешивающих индивидуальные и командные вознаграждения, авторами было замечено проявление у агентов согласованного поведения. Алгоритм FTW от DeepMind [56] играл в многопользовательскую игру Quake, применяя обучение на основе естественного отбора и глубокое RL. Целью каждого агента была максимизация вероятности победы его команды.

Эффективность выборки

Низкая эффективность выборки алгоритмов глубокого RL значительно ограничивает их применение к реальным задачам. Один из способов решения этой проблемы — объединение моделей среды. Недавние примеры — алгоритмы MB-MPO[1] [24] SimPLe[2] [62] от Клаверы и Кайзера с их коллегами соответственно.

Другой вариант — ретроспективная память прецедентов (Hindsight Experience Replay, HER) [6]. В основе этого метода лежит то, что люди учатся на ошибках так же хорошо, как и на успешных попытках решения задачи. В алгоритме HER на траекториях задаются новые вознаграждения, которые предполагают, что агент сделал в эпизоде именно то, что и должен был сделать. Это помогает ему исследовать среду при разреженных вознаграждениях, повышая тем самым эффективность выборки.

Обобщение

Способность к обобщению алгоритмов глубокого RL остается низкой. Они часто обучаются и тестируются на одних и тех же средах, что делает их склонными к переобучению. Например, в статье *Natural Environment Benchmarks for Reinforcement Learning* [152] Жанг и его коллеги показали, что простого зашумления изображений

---

[1] Model-Based Meta-Policy-Optimization — основанная на моделях метаоптимизация политики. Алгоритм использует ансамбль обученных моделей для метаобучения стратегии, которая может быть быстро адаптирована к любой модели из данного ансамбля за один шаг градиента стратегии. — *Примеч. пер.*

[2] Simulated Policy Learning — метод имитационного обучения стратегии — алгоритм глубокого RL, основанный на моделях прогнозирования видео. В алгоритме SimPLe, помимо среды эмулятора Atari 2600, применяется имитирующая ее среда (модель мира). У этих сред общие пространства действий и вознаграждений, и они производят визуальные наблюдения в одном и том же формате. Цель — с помощью имитирующей среды обучиться стратегии, которая будет показывать хорошую производительность в первоначальной среде. — *Примеч. пер.*

в играх Atari достаточно, для того чтобы обученный агент потерпел полную неудачу. Для решения этой проблемы они применили при обучении агентов преобразованные среды Atari, вставив в игры фоновое видео.

В предыдущем примере изменения вносились в среду, а основное задание оставалось одним и тем же. Однако нужно, чтобы агента можно было использовать для не виденных им связанных заданий. Для тестирования можно создать среду, которая будет постепенно порождать новые, но связанные задания. Два примера — игра Obstacle Tower [60] с порождением элементов и уровней и среда Animal-AI Olympics [7] с набором не связанных друг с другом заданий для тренировки и оценки.

Исследование и структурирование вознаграждений

Обучать агентов в средах с разреженными вознаграждениями сложно по той причине, что среда дает мало информации о том, какие действия желательны. При исследовании среды агентам трудно обнаружить полезные действия, и они часто застревают в локальном минимуме.

Во многих случаях возможно проектирование дополнительных вознаграждений, которые стимулируют поведение, помогающее решить задачу. Например, в среде BipedalWalker [18] вместо того, чтобы давать агенту вознаграждение, только если он сможет дойти до цели, его поощряют за движение вперед и наказывают за приложение крутящего момента и падение. Такие плотные вознаграждения способствуют эффективному продвижению агента вперед.

Впрочем, проектирование вознаграждений занимает много времени, так как они часто обусловлены спецификой задачи. Один из способов решения этой проблемы в том, чтобы давать агенту внутреннее вознаграждение, которое будет мотивировать исследование неиспытанных состояний и развитие новых навыков. Последний пример подобного вознаграждения агента за любопытство приводится в работе *Curiosity-Driven Exploration by Self-Supervised Prediction* Патака и др. [111].

В алгоритме Go-Explore [36] Экоффет с коллегами используют иной подход, основанный на том, что исследование может быть более эффективным, если начинать его с перспективных состояний, а не с начала эпизода. Для реализации этого обучение разделяют на фазы исследования и имитации. В первой фазе агент Go-Explore исследует среду случайным образом и запоминает интересные состояния с ведущими к ним траекториями. Во второй фазе он производит исследования не с нуля, а с сохраненных состояний.

Приложения

A История глубокого обучения с подкреплением

1947 год. Выборка методом Монте-Карло.

1958 год. Перцептрон.

1959 год. Метод временных различий (TD-обучение).

1983 год. ASE-ALE — первый алгоритм актора-критика.

1986 год. Алгоритм обратного распространения ошибки.

1989 год. Сверточные нейронные сети CNN.

1989 год. Q-обучение.

1991 год. TD-Gammon.

1992 год. REINFORCE.

1992 год. Память прецедентов (Experience Replay).

1994 год. SARSA.

1999 год. В Nvidia создали графический процессор.

2007 год. Выпуск CUDA.

2012 год. Среды для обучения по аркадным играм (Arcade Learning Environment, ALE).

2013 год. DQN.

Январь 2015 года. Обобщенная оценка преимущества (Generalized Advantage Estimation).

Февраль 2015 года. DQN обеспечено управление на уровне человека в Atari.

Февраль 2015 года. TRPO.

Сентябрь 2015 года. Градиент глубокой детерминированной стратегии (Deep Deterministic Policy Gradients, DDPG) [81].

Сентябрь 2015 года. Двойная DQN.

Ноябрь 2015 года. Дуэльная DQN [144].

Ноябрь 2015 года. Приоритизированная память прецедентов (Prioritized Experience Replay).

Ноябрь 2015 года. TensorFlow.

Февраль 2016 года. A3C.

Март 2016 года. AlphaGo победил Ли Седоля со счетом 4:1.

Июнь 2016 года. OpenAI Gym.

Июнь 2016 года. Генеративно-состязательное имитационное обучение (Generative Adversarial Imitation Learning, GAIL) [51].

Октябрь 2016 года. PyTorch.

Март 2017 года. Не зависящее от модели метаобучение (Model-Agnostic Meta-Learning, MAML) [40].

Июль 2017 года. Обучение с подкреплением на распределениях (Distributional RL) [13].

Июль 2017 года. PPO.

Август 2017 года. OpenAI сыграл в Dota 2 с реальными игроками со счетом 1:1.

Август 2017 года. Встроенный модуль любопытства (Intrinsic Curiosity Module, ICM) [111].

Октябрь 2017 года. Rainbow [49].

Декабрь 2017 года. AlphaZer [126].

Январь 2018 года. Мягкий актор-критик (Soft Actor-Critic, SAC) [47].

Февраль 2018 года. IMPALA [37].

Июнь 2018 года. Qt-Opt [64].

Ноябрь 2018 года. Go-Explore решил задачу среды Montezuma's Revenge [36].

Декабрь 2018 года. AlphaZero стал сильнейшим в истории игроком в шахматы, го и сёги.

Декабрь 2018 года. AlphaStar [3] победил одного из лучших игроков в мире в игре StarCraft II.

Апрель 2019 года. OpenAI Five победил мировых чемпионов в Dota 2.

Май 2019 года. FTW достиг уровня человека в игре Quake III Arena в эпизоде Capture the Flag [56].

Б Примеры сред

На сегодняшний день в глубоком RL представлен широчайший выбор сред, предлагаемых рядом библиотек на Python. Далее для справки перечислены некоторые из них.

1. Animal-AI Olympics [7], https://github.com/beyretb/AnimalAI-Olympics — соревнования для ИИ с тестами, основанными на познавательных способностях животных.
2. Deepdrive [115], https://github.com/deepdrive/deepdrive — моделирование вождения автомобиля.
3. DeepMind Lab [12], https://github.com/deepmind/lab — набор сложных заданий по трехмерной навигации и решению головоломок.
4. DeepMind PySC2 [142], https://github.com/deepmind/pysc2 — среда StarCraft II.
5. Gibson Environments [151], https://github.com/StanfordVL/GibsonEnv — реалистичное восприятие для воплощенных агентов.
6. Holodeck [46], https://github.com/BYU-PCCL/holodeck — высокоточные модели, созданные с помощью Unreal Engine 4.
7. Microsoft Malmö [57], https://github.com/Microsoft/malmo — среды Minecraft.
8. MuJoCo [136], http://www.mujoco.org/ — физический движок с моделированием робототехнических задач.
9. OpenAI Coinrun [25], https://github.com/openai/coinrun — среда, созданная специально для количественной оценки обобщаемости в RL.
10. OpenAI Gym [18], https://github.com/openai/gym — огромный выбор сред для классического управления, Box2D, робототехники и Atari.
11. OpenAI Retro [108], https://github.com/openai/retro — ретроигры, в числе которых Atari, NEC, Nintendo и Sega.
12. OpenAI Roboschool [109], https://github.com/openai/roboschool — роботизированные среды, настроенные для исследований.
13. Stanford osim-RL [129], https://github.com/stanfordnmbl/osim-rl — скелетно-мышечная среда RL.

14. Unity ML-Agents [59], https://github.com/Unity-Technologies/ml-agents — набор сред, созданных с помощью игрового движка Unity.
15. Unity Obstacle Tower [60], https://github.com/Unity-Technologies/obstacle-tower-env — процедурно порождаемая среда, состоящая из множества уровней (этажей), на каждом из которых агент должен решить задания для перехода на следующий уровень.
16. VizDoom [150], — игровой симулятор VizDoom, который можно использовать вместе с OpenAI Gym.

В этой книге рассмотрено много сред из OpenAI Gym, включая непрерывную среду Pendulum-v0 и несколько легко конфигурируемых дискретных сред: CartPole-v0, MountainCar-v0 и LunarLander-v2. Для более сложных сред используются некоторые игры из Atari, такие как PongNoFrameskip-v4 и BreakoutNoFrameskip-v4.

Б.1. Дискретные среды

В этом разделе предоставлено детальное описание сред CartPole-v0, MountainCar-v0 и LunarLander-v2 из OpenAI Gym, а также PongNoFrameskip-v4 и BreakoutNoFrameskip-v4 из Atari. Некоторые сведения взяты из первоначальной документации Wiki по OpenAI Gym.

Б.1.1. CartPole-v0

Эта простейшая игровая задача в OpenAI Gym, широко применяемая для отладки алгоритмов, впервые была описана Барто, Саттоном и Андерсоном [11]. Стержень закреплен на тележке, которая может перемещаться по дорожке без трения (рис. Б.1).

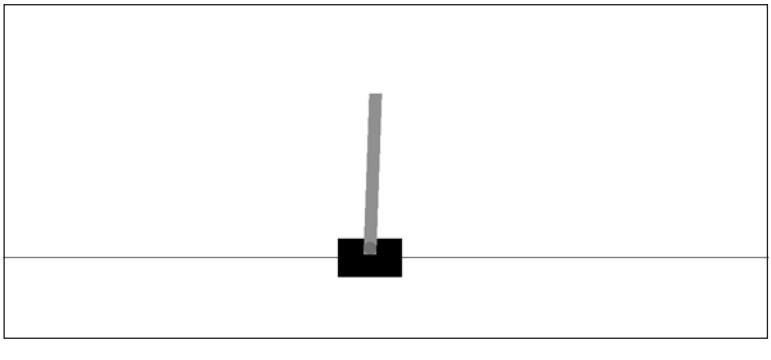


Рис. Б.1. Среда CartPole-v0, целью которой является удержание в равновесии стержня на протяжении 200 временных шагов

1. **Целевая функция** — удерживать стержень в вертикальном положении на протяжении 200 временных шагов.
2. **Состояние** — массив из четырех элементов: [позиция тележки, скорость тележки, угол наклона стержня, угловая скорость стержня], например: [–0,03474355; 0,03248249; –0,03100749; 0,03614301].
3. **Действие** — целое число из набора {0, 1}, означающее перемещение тележки влево или вправо, например: 0 — переместить влево.
4. **Вознаграждение** составляет +1 за каждый временной шаг, на котором стержень остается в вертикальном положении.
5. **Завершение** либо при падении стержня (отклонение на 12° от вертикали), либо при выходе тележки за пределы экрана, либо после максимального количества временных шагов, равного 200.
6. **Считается решенной** при среднем полном вознаграждении 195 за 100 последовательных эпизодов.

Б.1.2. MountainCar-v0

Это задача с разреженными вознаграждениями, которая была предложена Эндрю Муром [91] и в которой нужно раскачать машину без собственного двигателя так, чтобы она заехала на вершину холма, отмеченную флагом (рис. Б.2).

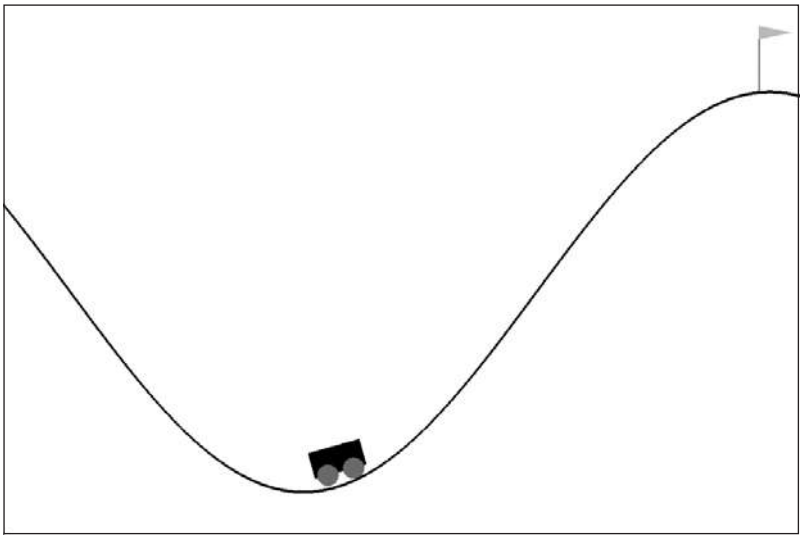


Рис. Б.2. Среда MountainCar-v0, цель которой — раскачивая машину влево и вправо, достичь вершины холма за кратчайшее время

1. **Целевая функция** — докатить машину до флага.
2. **Состояние** — массив из двух элементов: [позиция машины; скорость машины], например: [–0,59025158; 0].
3. **Действие** — целое число из набора {0, 1, 2}, соответствующее толчку влево, неподвижному стоянию и толчку вправо, например 0, чтобы толкнуть влево.
4. **Вознаграждение** — пока не будет достигнута вершина холма, на каждом временном шаге дается –1.
5. **Завершение**, когда машина достигает вершины холма или после максимального количества временных шагов, равного 200.
6. **Считается решенной** при среднем полном вознаграждении –110 за 100 последовательных эпизодов.

Б.1.3. LunarLander-v2

Это более сложная задача управления, где агент должен управлять посадочным модулем и прилунить его, не разбив. Количество топлива бесконечно, а посадочная площадка, обозначенная двумя флажками, всегда находится в центре (рис. Б.3).

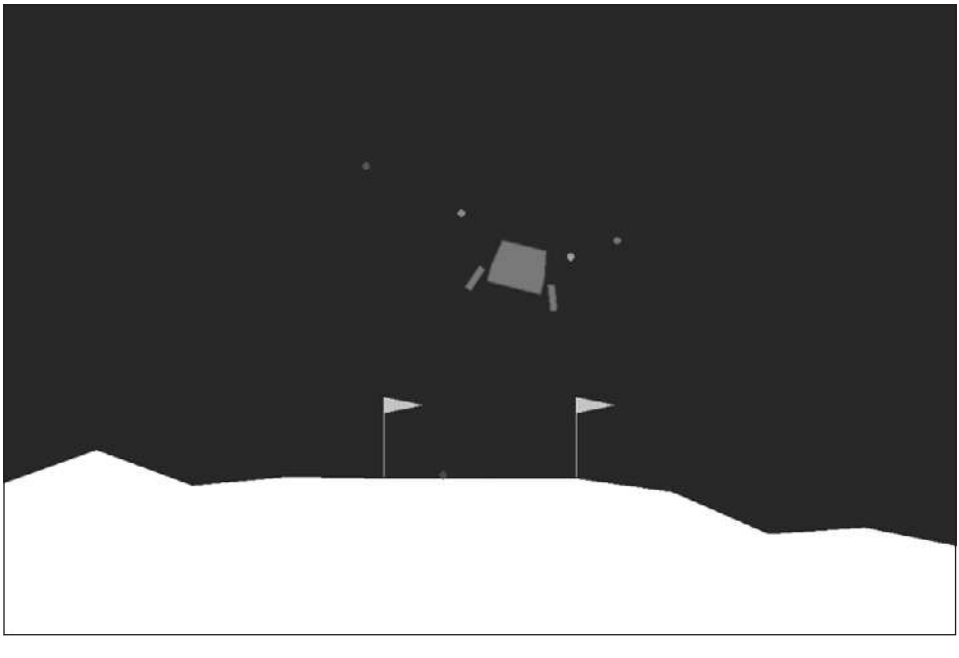


Рис. Б.3. Среда LunarLander-v2, цель — управляя посадочным модулем, прилунить его между двумя флажками, не разбив и использовав минимальное количество горючего

1. **Целевая функция** — прилунить посадочный модуль, не разбив и потратив меньше топлива. Полное вознаграждение за решение — +200.
2. **Состояние** — массив из восьми элементов: [координата *x*, координата *y*, проекция скорости на ось *X*, проекция скорости на ось *Y*, угол наклона модуля, угловая скорость модуля, контакт с поверхностью левой опоры, контакт с поверхностью правой опоры], например: [–0,00550737; 0,94806147; –0,55786095; 0,49652665; 0,00638856; 0,12636393; 0,0; 0,0].
3. **Действие** — целое число из набора {0, 1, 2, 3}, означающее незапущенные двигатели и зажигание левого, основного и правого двигателей; например, 2 соответствует запуску основного двигателя.
4. **Вознаграждение** составляет –100 за разбитый модуль, по –0,3 на временной шаг за зажигание основного двигателя, от +100 до +140 за посадку, по +10 за каждый контакт с поверхностью одной из опор.
5. **Завершение**, когда посадочный модуль прилунился или разбился либо после максимального количества временных шагов, равного 1000.
6. **Считается решенной** при среднем полном вознаграждении 200 за 100 последовательных шагов.

Б.1.4. PongNoFrameskip-v4

Это основанная на изображениях игра Atari, состоящая из мяча, левой ракетки под управлением программы и правой ракетки под управлением агента (рис. Б.4). Цель — поймать мяч и заставить оппонента его пропустить. Игра длится 21 раунд.

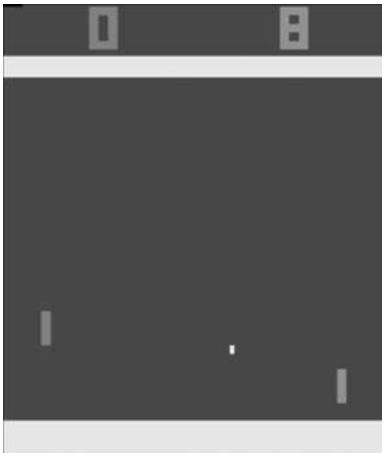


Рис. Б.4. Среда PongNoFrameskip-v4, основная цель которой — победить программного соперника слева

1. **Целевая функция** — максимизировать счет игры до +21.
2. **Состояние** — тензор изображения в формате RGB со структурой (210, 160, 3).
3. **Действие** — целое число из набора {0, 1, 2, 3, 4, 5} для управления симулятором игровой приставки.
4. **Вознаграждение** составляет –1, если мяч пропустил агент, и +1 — если оппонент.
5. **Завершение** после 21 раунда.
6. **Считается решенной** при максимизации среднего счета за 100 последовательных эпизодов, возможно достижение наилучшего счета +21.

Б.1.5. BreakoutNoFrameskip-v4

Это игра из Atari на основе изображений, в которой есть блоки, мячик, и платформа, расположенная внизу экрана (рис. Б.5). Цель — разрушить все блоки, отбивая мячик платформой. Жизнь теряется каждый раз, когда мячик падает мимо платформы.

1. **Целевая функция** — максимальный счет в игре.
2. **Состояние** — тензор изображения в формате RGB со структурой (210, 160, 3).
3. **Действие** — целое число из набора {0, 1, 2, 3} для управления симулятором игровой приставки.
4. **Вознаграждение** — определяется логикой игры.
5. **Завершение**, когда все жизни потеряны.
6. **Считается решенной** при максимизации среднего счета за 100 последовательных эпизодов.

Рис. Б.5. Среда BreakoutNoFrameskip-v4, основная цель которой — разрушение всех блоков

Б.2. Непрерывные среды

В число сред непрерывного управления в OpenAI Gym входят задачи роботизированного управления Pendulum-v0 и BipedalWalker-v2. В этом разделе представлено детальное описание данных сред, часть информации взята из оригинальной документации Wiki по OpenAI Gym.

Б.2.1. Pendulum-v0

Это задача непрерывного управления, в которой агент прикладывает крутящий момент к перевернутому маятнику без трения, чтобы раскачать его до верхнего вертикального положения и удерживать там (рис. Б.6).

1. **Целевая функция** — раскачивать маятник и удерживать его в верхнем вертикальном положении.
2. **Состояние** — массив из трех элементов, содержащий угол шарнира θ и его угловую скорость: $\left[\cos\theta, \sin\theta, \frac{d\theta}{dt}\right]$, например: [0,91450008; –0,40458573; 0,85436913].
3. **Действие** — число с плавающей запятой из промежутка [–2,0; 2,0], соответствующее приложенному к шарниру крутящему моменту, например 0 при отсутствии крутящего момента.
4. **Вознаграждение** определяется из уравнения $-\left(\theta^2 + 0{,}1\frac{d\theta^2}{dt} + 0{,}001M^2\right)$, где M — крутящий момент.

Рис. Б.6. Среда Pendulum-v0, цель — так раскачать перевернутый маятник, чтобы он принял верхнее вертикальное положение и оставался в нем

5. **Завершение** после максимального количества временных шагов, равного 200.
6. **Считается решенной** при максимизации среднего полного вознаграждения за 100 последовательных эпизодов.

Б.2.2. BipedalWalker-v2

Это задача непрерывного управления, в которой агент сканирует свое окружение с помощью датчика-лидара робота и двигается вправо без падений (рис. Б.7).

1. **Целевая функция** — идти вправо, не падая.
2. **Состояние** — массив из 24 элементов: [угол наклона корпуса, угловая скорость корпуса, скорость по оси *X*, скорость по оси *Y*, угол шарнира бедра 1, скорость шарнира бедра 1, угол шарнира колена 1, скорость шарнира колена 1, контакт с землей ноги 1, угол шарнира бедра 2, скорость шарнира бедра 2, угол шарнира колена 2, скорость шарнира колена 2, контакт с землей ноги 2... 10 показаний лидара]. Например, [2,74561788e–03; 1,18099805e–05; –1,53996013e–03; –1,60000777e–02... 7,09147751e–01; 8,85930359e–01; 1,00000000e+00; 1,00000000e+00].
3. **Действие** — вектор из четырех чисел с плавающей запятой со значениями в интервале [–1,0; 1,0]: [крутящий момент и скорость бедра 1, крутящий момент и скорость колена 1, крутящий момент и скорость бедра 2, крутящий момент и скорость колена 2], например: [0,09762701; 0,43037874; 0,20552675; 0,08976637].

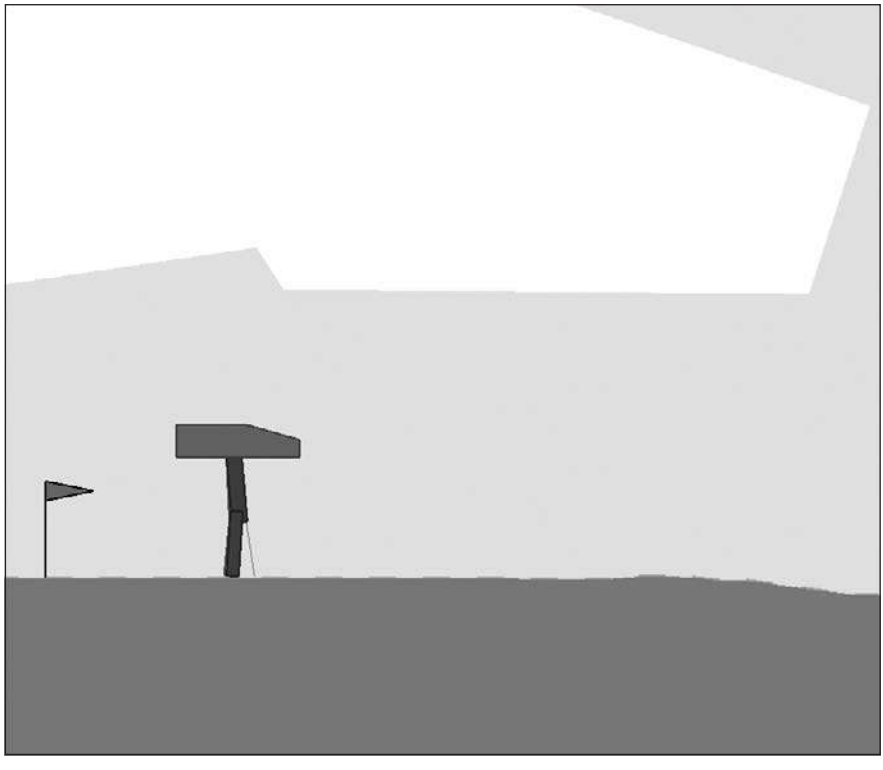


Рис. Б.7. Среда BipedalWalker-v2, цель — перемещение агента вправо без падений

4. **Вознаграждение** — за движение вправо максимально до +300 и –100 за падение робота.
5. **Завершение**, когда тело робота касается земли, либо при достижении цели, расположенной справа, либо после максимального количества временных шагов, равного 1600.
6. **Считается решенной** при среднем полном вознаграждении 300 за 100 последовательных эпизодов.

Список используемых источников

1. *Abadi M., Agarwal A., Barham P., Brevdo E., Chen Z., Citro C., Corrado G. S. et al.* TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. 2015. https://arxiv.org/abs/1603.04467.
2. *Achiam J., Held D., Tamar A., Abbeel P.* Constrained Policy Optimization. 2017. https://arxiv.org/abs/1705.10528.
3. AlphaStar Team. AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. 2019. https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii.
4. *Amodei D., Olah C., Steinhardt J., Christiano P., Schulman J., Mané D.* Concrete Problems in AI Safety. 2016. https://arxiv.org/abs/1606.06565.
5. *Anderson H. L.* Metropolis, Monte Carlo and the MANIAC // Los Alamos Science 14, 1986. http://library.lanl.gov/cgi-bin/getfile?00326886.pdf.
6. *Andrychowicz M., Wolski F., Ray A., Schneider J., Fong R., Welinder P., McGrew B., Tobin J., Abbeel P., Zaremba W.* Hindsight Experience Replay. 2017. https://arxiv.org/abs/1707.01495.
7. Animal-AI Olympics. https://github.com/beyretb/AnimalAI-Olympics.
8. *Arjona-Medina J. A., Gillhofer M., Widrich M., Unterthiner T., Brandstetter J., Hochreiter S.* RUDDER: Return Decomposition for Delayed Rewards. 2018. https://arxiv.org/abs/1806.07857.
9. *Baird L. C.* Advantage Updating. Tech. rep. Wright-Patterson Air Force Base, OH, 1993.
10. *Bakker B.* Reinforcement Learning with Long Short-Term Memory // Advances in Neural Information Processing Systems 14 (NeurIPS 2001). Ed. by Dietterich T. G., Becker S., Ghahramani Z. MIT Press, 2002. P. 1475–1482. http://papers.nips.cc/paper/1953-reinforcement-learning-with-long-short-term-memory.pdf.
11. *Barto A. G., Sutton R. S., Anderson C. W.* Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems // IEEE Transactions on Systems,

Man & Cybernetics 13.5, 1983. P. 834–846. https://ieeexplore.ieee.org/abstract/document/6313077.

12. *Beattie C., Leibo J. Z., Teplyashin D., Ward T., Wainwright M., Küttler H., Lefrancq A. et al.* DeepMind Lab. 2016. https://arxiv.org/abs/1612.03801.
13. *Bellemare M. G., Dabney W., Munos R.* A Distributional Perspective on Reinforcement Learning. 2017. https://arxiv.org/abs/1707.06887.
14. *Bellemare M. G., Naddaf Y., Veness J., Bowling M.* The Arcade Learning Environment: An Evaluation Platform for General Agents // Journal of Artificial Intelligence Research 47. June 2013. P. 253–279. https://arxiv.org/abs/1207.4708.
15. *Bellman R.* A Markovian Decision Process // Indiana University Mathematics Journal 6.5. 1957. P. 679–684. https://www.jstor.org/stable/24900506.
16. *Bellman R.* The Theory of Dynamic Programming // Bulletin of the American Mathematical Society 60.6. 1954. https://projecteuclid.org/euclid.bams/1183519147.
17. Box2D. 2006. https://github.com/erincatto/box2d.
18. *Brockman G., Cheung V., Pettersson L., Schneider J., Schulman J., Tang J., Zaremba W.* OpenAI Gym. 2016. https://arxiv.org/abs/1606.01540.
19. Bullet Physics SDK. https://github.com/bulletphysics/bullet3.
20. *Cho K.* Natural Language Understanding with Distributed Representation. 2015. https://arxiv.org/abs/1511.07916.
21. *Cho K., van Merrienboer B., Gulçehre Ç., Bahdanau D., Bougares F., Schwenk H., Bengio Y.* Learning Phrase Representations Using RNN Encoder-Decoder for Statistical Machine Translation. 2014. https://arxiv.org/abs/1406.1078.
22. *Chrabaszcz P.* Canonical ES Finds a Bug in Qbert (Full). YouTube video, 13:54. 2018. https://youtu.be/meE5aaRJ0Zs.
23. *Chrabaszcz P., Loshchilov I., Hutter F.* Back to Basics: Benchmarking Canonical Evolution Strategies for Playing Atari. 2018. https://arxiv.org/abs/1802.08842.
24. *Clavera I., Rothfuss J., Schulman J., Fujita Y., Asfour T., Abbeel P.* Model-Based Reinforcement Learning via Meta-Policy Optimization. 2018. https://arxiv.org/abs/1809.05214.
25. *Cobbe K., Klimov O., Hesse C., Kim T., Schulman J.* Quantifying Generalization in Reinforcement Learning. 2018. https://arxiv.org/abs/1812.02341.
26. Codacy. https://www.codacy.com.
27. Code Climate. https://codeclimate.com.
28. *Cox D., Dean T.* Neural Networks and Neuroscience-Inspired Computer Vision // Current Biology 24. 2014. Computational Neuroscience. P. 921–929. https://linkinghub.elsevier.com/retrieve/pii/S0960982214010392.

29. CTRL-Labs. https://www.ctrl-labs.com.

30. *Cun Y. L., Denker J. S., Solla S. A.* Optimal Brain Damage // Advances in Neural Information Processing Systems 2. Ed. by Touretzky, D. S. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990. P. 598–605. http://dl.acm.org/citation.cfm?id=109230.109298.

31. Daniel Rozin Interactive Art. http://www.smoothware.com/danny.

32. *Deng J., Dong W., Socher R., Li L.-J., Li K., Fei-Fei L.* ImageNet: A Large-Scale Hierarchical Image Database // IEEE Conference on Computer Vision and Pattern Recognition, CVPR09. IEEE, 2009. P. 248–255.

33. *Dettmers T.* A Full Hardware Guide to Deep Learning. 2018. https://timdettmers.com/2018/12/16/deep-learning-hardware-guide.

34. *Dreyfus S.* Richard Bellman on the Birth of Dynamic Programming // Operations Research 50.1. 2002. P. 48–51. https://pubsonline.informs.org/doi/abs/10.1287/opre.50.1.48.17791.

35. *Duan Y., Chen X., Houthooft R., Schulman J., Abbeel P.* Benchmarking Deep Reinforcement Learning for Continuous Control. 2016. https://arxiv.org/abs/1604.06778.

36. *Ecoffet A., Huizinga J., Lehman J., Stanley K. O., Clune J.* Go-Explore: A New Approach for Hard-Exploration Problems. 2019. https://arxiv.org/abs/1901.10995.

37. *Espeholt L., Soyer H., Munos R., Simonyan K., Mnih V., Ward T., Doron Y. et al.* IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures. 2018. https://arxiv.org/abs/1802.01561.

38. *Fazel M., Ge R., Kakade S. M., Mesbahi M.* Global Convergence of Policy Gradient Methods for the Linear Quadratic Regulator. 2018. https://arxiv.org/abs/1801.05039.

39. *Fei-Fei L., Johnson J., Yeung S.* Lecture 14: Reinforcement Learning // CS231n: Convolutional Neural Networks for Visual Recognition, Spring 2017. Stanford University, 2017. http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf.

40. *Finn C., Abbeel P., Levine S.* Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks. 2017. https://arxiv.org/abs/1703.03400.

41. *Frankle J., Carbin M.* The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks. 2018. https://arxiv.org/abs/1803.03635.

42. *Fujimoto S., van Hoof H., Meger D.* Addressing Function Approximation Error in Actor-Critic Methods. 2018. https://arxiv.org/abs/1802.09477.

43. *Fukushima K.* Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position // Biological Cybernetics 36. 1980. P. 193–202. https://www.rctn.org/bruno/public/papers/Fukushima1980.pdf.

44. Glances — An Eye on Your System. https://github.com/nicolargo/glances.

45. *Goodfellow I., Bengio Y., Courville A.* Deep Learning. MIT Press, 2016. http://www.deeplearningbook.org.

46. *Greaves J., Robinson M., Walton N., Mortensen M., Pottorff R., Christopherson C., Hancock D., Wingate D.* Holodeck — A High Fidelity Simulator for Reinforcement Learning. 2018. https://github.com/byu-pccl/holodeck.

47. *Haarnoja T., Zhou A., Abbeel P., Levine S.* Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. 2018. https://arxiv.org/abs/1801.01290.

48. *Henderson P., Islam R., Bachman P., Pineau J., Precup D., Meger D.* Deep Reinforcement Learning That Matters. 2017. https://arxiv.org/abs/1709.06560.

49. *Hessel M., Modayil J., van Hasselt H., Schaul T., Ostrovski G., Dabney W., Horgan D., Piot B., Azar M. G., Silver D.* Rainbow: Combining Improvements in Deep Reinforcement Learning. 2017. https://arxiv.org/abs/1710.02298.

50. *Hinton G.* Lecture 6a: Overview of Mini-Batch Gradient Descent // CSC321: Neural Networks for Machine Learning. University of Toronto, 2014. http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

51. *Ho J., Ermon S.* Generative Adversarial Imitation Learning. 2016. https://arxiv.org/abs/1606.03476.

52. *Hochreiter S., Schmidhuber J.* Long Short-Term Memory // Neural Computation 9.8. Nov. 1997. P. 1735–1780. https://direct.mit.edu/neco/article-abstract/9/8/1735/6109/Long-Short-Term-Memory?redirectedFrom=fulltext.

53. *Hofstadter D. R.* Gödel, Escher, Bach: An Eternal Golden Braid. New York: Vintage Books, 1980.

54. *Horgan D., Quan J., Budden D., Barth-Maron G., Hessel M., van Hasselt H., Silver D.* Distributed Prioritized Experience Replay. 2018. https://arxiv.org/abs/1803.00933.

55. HTC Vive. https://www.vive.com/us.

56. *Jaderberg M., Czarnecki W. M., Dunning I., Marris L., Lever G., Castañeda A. G., Beattie C. et al.* Human-Level Performance in 3D Multiplayer Games with Population-Based Reinforcement Learning // Science 364.6443. 2019. P. 859–865. https://www.science.org/doi/10.1126/science.aau6249.

57. *Johnson M., Hofmann K., Hutton T., Bignell D.* The Malmo Platform for Artificial Intelligence Experimentation // IJCAI'16 Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence. New York, USA: AAAI Press, 2016. P. 4246–4247. http://dl.acm.org/citation.cfm?id=3061053.3061259.

58. *Jouppi N.* Google Supercharges Machine Learning Tasks with TPU Custom Chip. Google Blog. 2016. https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html.

59. *Juliani A., Berges V.-P., Vckay E., Gao Y., Henry H., Mattar M., Lange D.* Unity: A General Platform for Intelligent Agents. 2018. https://arxiv.org/abs/1809.02627.

60. *Juliani A., Khalifa A., Berges V., Harper J., Henry H., Crespi A., Togelius J., Lange D.* Obstacle Tower: A Generalization Challenge in Vision, Control and Planning. 2019. https://arxiv.org/abs/1902.01378.

61. *Kaelbling L. P., Littman M. L., Moore A. P.* Reinforcement Learning: A Survey // Journal of Artificial Intelligence Research 4. 1996. P. 237–285. http://people.csail.mit.edu/lpk/papers/rl-survey.ps.

62. *Kaiser L., Babaeizadeh M., Milos P., Osinski B., Campbell R. H., Czechowski K., Erhan D. et al.* Model-Based Reinforcement Learning for Atari. 2019. https://arxiv.org/abs/1903.00374.

63. *Kakade S.* A Natural Policy Gradient // NeurIPS'11 Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic. Cambridge, MA, USA: MIT Press, 2001. P. 1531–1538. http://dl.acm.org/citation.cfm?id=2980539.2980738.

64. *Kalashnikov D., Irpan A., Pastor P., Ibarz J., Herzog A., Jang E., Quillen D. et al.* QT-Opt: Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation. 2018. https://arxiv.org/abs/1806.10293.

65. *Kapturowski S., Ostrovski G., Quan J., Munos R., Dabney W.* Recurrent Experience Replay in Distributed Reinforcement Learning // International Conference on Learning Representations. 2019. https://openreview.net/forum?id=r1lyTjAqYX.

66. *Karpathy A.* The Unreasonable Effectiveness of Recurrent Neural Networks. 2015. http://karpathy.github.io/2015/05/21/rnn-effectiveness.

67. *Keng W. L., Graesser L.* SLM-Lab. 2017. https://github.com/kengz/SLM-Lab.

68. *Kingma D. P., Ba J.* Adam: A Method for Stochastic Optimization. 2014. https://arxiv.org/abs/1412.6980.

69. Leap Motion. https://www.leapmotion.com/technology.

70. *LeCun Y., Boser B., Denker J. S., Henderson D., Howard R. E., Hubbard W., Jackel L. D.* Backpropagation Applied to Handwritten Zip Code Recognition // Neural Computation 1. 1989. P. 541–551. http://yann.lecun.com/exdb/publis/pdf/lecun-89e.pdf.

71. *LeCun Y.* Generalization and Network Design Strategies // Connectionism in Perspective. Ed. by Pfeifer, R., Schreter, Z., Fogelman, F., and Steels, L. Elsevier, 1989.

72. *LeCun Y.* Week 10, Unsupervised Learning Part 1 // DS-GA-1008: Deep Learning, NYU. 2017. https://cilvr.cs.nyu.edu/lib/exe/fetch.php?media=deeplearning:2017:007-unsup01.pdf.

73. *LeCun Y., Cortes C., Burges C. J.* The MNIST Database of Handwritten Digits. 2010. http://yann.lecun.com/exdb/mnist.

74. *Levine S.* Sep 6: Policy Gradients Introduction, Lecture 4 // CS 294: Deep Reinforcement Learning, Fall 2017. UC Berkeley, 2017. http://rail.eecs.berkeley.edu/deeprlcourse-fa17/index.html#lectures.

75. *Levine S.* Sep 11: Actor-Critic Introduction, Lecture 5 // CS 294: Deep Reinforcement Learning, Fall 2017. UC Berkeley, 2017. http://rail.eecs.berkeley.edu/deeprlcourse-fa17/index.html#lectures.

76. *Levine S.* Sep 13: Value Functions Introduction, Lecture 6 // CS 294: Deep Reinforcement Learning, Fall 2017. UC Berkeley, 2017. http://rail.eecs.berkeley.edu/deeprlcourse-fa17/index.html#lectures.

77. *Levine S.* Sep 18: Advanced Q-Learning Algorithms, Lecture 7 // CS 294: Deep Reinforcement Learning, Fall 2017. UC Berkeley, 2017. http://rail.eecs.berkeley.edu/deeprlcourse-fa17/index.html#lectures.

78. *Levine S., Achiam J.* Oct 11: Advanced Policy Gradients, Lecture 13 // CS 294: Deep Reinforcement Learning, Fall 2017. UC Berkeley, 2017. http://rail.eecs.berkeley.edu/deeprlcourse-fa17/index.html#lectures.

79. *Li W., Todorov E.* Iterative Linear Quadratic Regulator Design for Nonlinear Biological Movement Systems // Proceedings of the 1st International Conference on Informatics in Control, Automation and Robotics (ICINCO 1). Ed. By Araújo, H., Vieira, A., Braz, J., Encarnação, B., and Carvalho, M. INSTICC Press, 2004. P. 222–229.

80. LIGO. Gravitational Waves Detected 100 Years after Einstein's Prediction. 2016. https://www.ligo.caltech.edu/news/ligo20160211.

81. *Lillicrap T. P., Hunt J. J., Pritzel A., Heess N., Erez T., Tassa Y., Silver D., Wierstra D.* Continuous Control with Deep Reinforcement Learning. 2015. https://arxiv.org/abs/1509.02971.

82. *Lin L.-J.* Self-Improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching // Machine Learning 8. 1992. P. 293–321. http://www.incompleteideas.net/lin-92.pdf.

83. *Mania H., Guy A., Recht B.* Simple Random Search Provides a Competitive Approach to Reinforcement Learning. 2018. https://arxiv.org/abs/1803.07055.

84. MarioKart 8. https://mariokart8.nintendo.com.

85. *Metropolis N.* The Beginning of the Monte Carlo Method // Los Alamos Science. Special Issue. 1987. http://library.lanl.gov/cgi-bin/getfile?00326866.pdf.

86. Microsoft Research. Policy Gradient Methods: Tutorial and New Frontiers. YouTube video, 1:09:19. 2017. https://youtu.be/y4ci8whvS1E.

87. *Mnih V., Badia A. P., Mirza M., Graves A., Harley T., Lillicrap T. P., Silver D., Kavukcuoglu K.* Asynchronous Methods for Deep Reinforcement Learning // ICML'16 Proceedings of the 33rd International Conference on International Conference on Machine Learning. Volume 48. New York, NY, USA: JMLR.org, 2016. P. 1928–1937. http://dl.acm.org/citation.cfm?id=3045390.3045594.

88. *Mnih V., Kavukcuoglu K., Silver D., Graves A., Antonoglou I., Wierstra D., Riedmiller M. A.* Playing Atari with Deep Reinforcement Learning. 2013. https://arxiv.org/abs/1312.5602.

89. *Mnih V., Kavukcuoglu K., Silver D., Rusu A. A., Veness J., Bellemare M. G., Graves A. et al.* Human-Level Control through Deep Reinforcement Learning // Nature 518.7540. Feb. 2015. P. 529–533. http://dx.doi.org/10.1038/nature14236.

90. *Molchanov P., Tyree S., Karras T., Aila T., Kautz J.* Pruning Convolutional Neural Networks for Resource Efficient Inference. 2016. https://arxiv.org/abs/1611.06440.

91. *Moore A. W.* Efficient Memory-Based Learning for Robot Control. Tech. rep. University of Cambridge, 1990.

92. *Nielsen M.* Neural Networks and Deep Learning. Determination Press, 2015.

93. *Niu F., Recht B., Re C., Wright S. J.* HOGWILD! A Lock-Free Approach to Parallelizing Stochastic Gradient Descent // NeurIPS'11 Proceedings of the 24th International Conference on Neural Information Processing Systems. USA: Curran Associates Inc., 2011. P. 693–701. http://dl.acm.org/citation.cfm?id=2986459.2986537.

94. *Norman D. A.* The Design of Everyday Things. New York, NY, USA: Basic Books, Inc., 2002.

95. Nvidia. NVIDIA Launches the World's First Graphics Processing Unit: GeForce 256. 1999. https://www.nvidia.com/object/IO_20020111_5424.html.

96. NVIDIA. NVIDIA Unveils CUDA–The GPU Computing Revolution Begins. 2006. https://www.nvidia.com/object/IO_37226.html.

97. Oculus. https://www.oculus.com.

98. OpenAI. *Andrychowicz M., Baker B., Chociej M., Józefowicz R., McGrew B., Pachocki J. et al.* Learning Dexterous In-Hand Manipulation. 2018. https://arxiv.org/abs/1808.00177.

99. OpenAI Baselines. https://github.com/openai/baselines.

100. OpenAI Blog. Faulty Reward Functions in the Wild. 2016. https://blog.openai.com/faulty-reward-functions.

101. OpenAI Blog. Learning Dexterity. 2018. https://blog.openai.com/learning-dexterity.

102. OpenAI Blog. More on Dota 2. 2017. https://blog.openai.com/more-on-dota-2.

103. OpenAI Blog. OpenAI Baselines: DQN. 2017. https://blog.openai.com/openai-baselines-dqn.

104. OpenAI Blog. OpenAI Five. 2018. https://blog.openai.com/openai-five.

105. OpenAI Blog. OpenAI Five Benchmark: Results. 2018. https://blog.openai.com/openai-five-benchmark-results.

106. OpenAI Blog. Proximal Policy Optimization. 2017. https://blog.openai.com/openai-baselines-ppo.

107. OpenAI Five. https://openai.com/five.

108. OpenAI Retro. https://github.com/openai/retro.

109. OpenAI Roboschool. 2017. https://github.com/openai/roboschool.

110. *Parisotto E., Ba L. J., Salakhutdinov R.* Actor-Mimic: Deep Multitask and Transfer Reinforcement Learning // 4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2–4, 2016. Conference Track Proceedings. 2016. https://arxiv.org/abs/1511.06342.

111. *Pathak D., Agrawal P., Efros A. A., Darrell T.*"Curiosity-Driven Exploration by Self-Supervised Prediction // ICML. 2017.

112. *Peters J., Schaal S.* Reinforcement Learning of Motor Skills with Policy Gradients // Neural Networks 21.4. May 2008. P. 682–697.

113. *Peters J., Schaal S.* Natural Actor-Critic // Neurocomputing 71.7–9. Mar. 2008. P. 1180–1190. https://linkinghub.elsevier.com/retrieve/pii/S0925231208000532.

114. PyTorch. 2018. https://github.com/pytorch/pytorch.

115. *Quiter C., Ernst M.* deepdrive/deepdrive: 2.0. Mar. 2018. https://zenodo.org/record/1248998#.YXaWm1VBzGg.

116. ROLI. Seaboard: The future of the keyboard. https://roli.com/products/seaboard.

117. *Rumelhart D. E., Hinton G. E., Williams R. J.* Learning Representations by Back-Propagating Errors // Nature 323. Oct. 1986. P. 533–536. https://www.nature.com/articles/323533a0.

118. *Rummery G. A., Niranjan M.* On-Line Q-Learning Using Connectionist Systems. Tech. rep. University of Cambridge, 1994.

119. *Samuel A. L.* Some Studies in Machine Learning Using the Game of Checkers // IBM Journal of Research and Development 3.3. July 1959. P. 210–229. https://ieeexplore.ieee.org/document/5392560/.

120. *Samuel A. L.* Some Studies in Machine Learning Using the Game of Checkers. II–ecent Progress // IBM Journal of Research and Development 11.6. Nov. 1967. P. 601–617. https://ieeexplore.ieee.org/document/5391906.

121. *Schaul T., Quan J., Antonoglou I., Silver D.* Prioritized Experience Replay. 2015. https://arxiv.org/abs/1511.05952.

122. *Schulman J., Levine S., Moritz P., Jordan M. I., Abbeel P.* Trust Region Policy Optimization. 2015. https://arxiv.org/abs/1502.05477.

123. *Schulman J., Moritz P., Levine S., Jordan M. I., Abbeel P.* High-Dimensional Continuous Control Using Generalized Advantage Estimation. 2015. https://arxiv.org/abs/1506.02438.

124. *Schulman J., Wolski F., Dhariwal P., Radford A., Klimov O.* Proximal Policy Optimization Algorithms. 2017. https://arxiv.org/abs/1707.06347.

125. *Silver D., Huang A., Maddison C. J., Guez A., Sifre L., Van Den Driessche G., Schrittwieser J., Antonoglou I., Panneershelvam V., Lanctot M. et al.* Mastering the Game of Go with Deep Neural Networks and Tree Search // Nature 529.7587. 2016. P. 484–489.

126. *Silver D., Hubert T., Schrittwieser J., Antonoglou I., Lai M., Guez A., Lanctot M., Sifre L., Kumaran D., Graepel T. et al.* Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. 2017. https://arxiv.org/abs/1712.01815.

127. *Silver D., Schrittwieser J., Simonyan K., Antonoglou I., Huang A., Guez A., Hubert T., Baker L., Lai M., Bolton A. et al.* Mastering the Game of Go without Human Knowledge // Nature 550.7676. 2017. P. 354.

128. *Smith J. E., Winkler R. L.* The Optimizer's Curse: Skepticism and Postdecision Surprise in Decision Analysis // Management Science 52.3. 2006. P. 311–322. http://dblp.uni-trier.de/db/journals/mansci/mansci52.html#SmithW06.

129. Stanford osim-RL. https://github.com/stanfordnmbl/osim-rl.

130. *Sutton R. S.* Dyna, an Integrated Architecture for Learning, Planning, and Reacting // ACM SIGART Bulletin 2.4. July 1991. P. 160–163. https://dl.acm.org/doi/10.1145/122344.122377.

131. *Sutton R. S.* Learning to Predict by the Methods of Temporal Differences // Machine Learning 3.1. 1988. P. 9–44.

132. *Sutton R. S., Barto A. G.* Reinforcement Learning: An Introduction. Second ed. The MIT Press, 2018. https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf.

133. *Sutton R. S., McAllester D., Singh S., Mansour Y.* Policy Gradient Methods for Reinforcement Learning with Function Approximation // NeurIPS'99 Proceedings of the 12th International Conference on Neural Information Processing Systems.

Cambridge, MA, USA: MIT Press, 1999. P. 1057–1063. http://dl.acm.org/citation.cfm?id=3009657.3009806.

134. *Teh Y. W., Bapst V., Czarnecki W. M., Quan J., Kirkpatrick J., Hadsell R., Heess N., Pascanu R.* Distral: Robust Multitask Reinforcement Learning. 2017. https://arxiv.org/abs/1707.04175.

135. *Tesauro G.* Temporal Difference Learning and TD-Gammon // Communications of the ACM 38.3. Mar. 1995. P. 58–68. https://dl.acm.org/doi/10.1145/203330.203343.

136. *Todorov E., Erez T., Tassa Y.* MuJoCo: A Physics Engine for Model-Based Control // IROS. IEEE, 2012. P. 5026–5033. http://dblp.uni-trier.de/db/conf/iros/iros2012.html#TodorovET12.

137. *Tsitsiklis J. N., Van Roy B.* An Analysis of Temporal-Difference Learning with Function Approximation // IEEE Transactions on Automatic Control 42.5. 1997. http://www.mit.edu/~jnt/Papers/J063-97-bvr-td.pdf.

138. Unity. https://unity3d.com.

139. Unreal Engine. https://www.unrealengine.com.

140. *van Hasselt H.* Double Q-Learning // Advances in Neural Information Processing Systems 23. Ed. by Lafferty, J. D. et al. Curran Associates, 2010. P. 2613–2621. http://papers.nips.cc/paper/3964-double-q-learning.pdf.

141. *van Hasselt H., Guez A., Silver D.* Deep Reinforcement Learning with Double Q-Learning. 2015. https://arxiv.org/abs/1509.06461.

142. *Vinyals O., Ewalds T., Bartunov S., Georgiev P., Vezhnevets A. S., Yeo M., Makhzani A. et al.* StarCraft II: A New Challenge for Reinforcement Learning. 2017. https://arxiv.org/abs/1708.04782.

143. *Walt S. V., Colbert S. C., Varoquaux G.* The NumPy Array: A Structure for Efficient Numerical Computation // Computing in Science and Engineering 13.2. Mar. 2011. P. 22–30. https://ieeexplore.ieee.org/document/5725236.

144. *Wang Z., Schaul T., Hessel M., van Hasselt H., Lanctot M., Freitas N. D.* Dueling Network Architectures for Deep Reinforcement Learning. 2015. https://arxiv.org/abs/1511.06581.

145. *Watkins C. J.* Learning from Delayed Rewards. PhD thesis. Cambridge, UK: King's College, May 1989. http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf.

146. *Widrow B., Gupta N. K., Maitra S.* Punish/Reward: Learning with a Critic in Adaptive Threshold Systems // IEEE Transactions on Systems, Man, and Cybernetics SMC-3.5. Sept. 1973. P. 455–465. https://ieeexplore.ieee.org/document/4309272.

147. *Wierstra D., Förster A., Peters J., Schmidhuber J.* Recurrent Policy Gradients // Logic Journal of the IGPL 18.5. Oct. 2010. P. 620–634.

148. *Williams R. J.* Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning // Machine Learning 8.3–4. May 1992. P. 229–256. https://link.springer.com/article/10.1007%2FBF00992696.

149. *Williams R. J., Peng J.* Function Optimization Using Connectionist Reinforcement Learning algorithms // Connection Science 3.3. 1991. P. 241–268.

150. *Wydmuch M., Kempka M., Jaśkowski W.* ViZDoom Competitions: Playing Doom from Pixels // IEEE Transactions on Games 11.3. 2018.

151. *Xia F., R. Zamir A., He Z.-Y., Sax A., Malik J., Savarese S.* Gibson Env: Real-World Perception for Embodied Agents // 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition. IEEE. 2018. https://arxiv.org/abs/1808.10654.

152. *Zhang A., Wu Y., Pineau J.* Natural Environment Benchmarks for Reinforcement Learning. 2018. https://arxiv.org/abs/1811.06032.

153. *Zhang S., Sutton R. S.* A Deeper Look at Experience Replay. 2017. https://arxiv.org/abs/1712.01275.

Дорогие читатели!

Мы благодарим вас за доверие издательству «Питер»!

Покупая книги, вы проявляете уважение к автору и всем людям, которые трудились над их созданием. Вы вдохновляете нас на поиск нового уникального, качественного и полезного контента.

Лаура Грессер, Ван Лун Кенг

Глубокое обучение с подкреплением: теория и практика на языке Python

Перевел с английского *К. Синица*

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*А. Питиримов*
Ведущий редактор	*Н. Гринчик*
Научный редактор	*А. Панов*
Литературный редактор	*Н. Рощина*
Художественный редактор	*В. Мостипан*
Корректоры	*О. Андриевич, Е. Павлович*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 01.2022. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 23.11.21. Формат 70×100/16. Бумага офсетная. Усл. п. л. 33,540. Тираж 500. Заказ 0000.