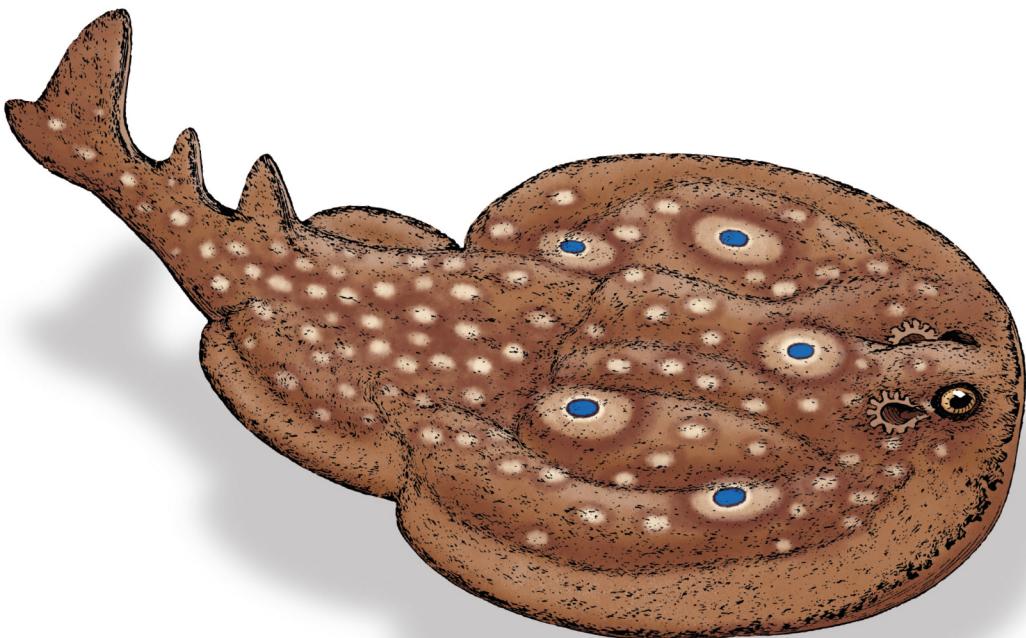


O'REILLY®

# Изучаем Ray

Гибкие распределенные  
вычисления на Python  
в машинном обучении



Макс Пумперла  
Эдвард Оукс  
Ричард Ляо



Max Pumperla, Edward Oakes, and Richard Liaw

# Learning Ray

**Flexible Distributed Python  
for Machine Learning**

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Макс Пумперла, Эдвард Оукс и Ричард Ляо

# Изучаем Ray

Гибкие распределенные вычисления  
на Python в машинном обучении



УДК 004.043  
П88 32.973.22  
П88

Пумперла М., Оукс Э., Ляо Р.  
П88 Изучаем Ray / пер. с англ. А. В. Логунова. – М.: Books.kz, 2023. – 290 с.: ил.

**ISBN 978-6-01083-430-9**

В книге излагаются основы работы с фреймворком распределенных вычислений с открытым исходным кодом Ray, который упрощает процесс масштабирования вычислительно интенсивных рабочих нагрузок на Python. Читатель научится применять фреймворк Ray локально и разворачивать вычислительные кластеры Ray, создавать распределенные приложения с помощью ядра фреймворка – Ray Core, управлять распределенным обучением с помощью библиотеки Ray Train.

Издание предназначено для программистов на Python, инженеров и исследователей данных.

УДК 004.043  
ББК 32.973.22

Copyright © 2023 Max Pumperla and O'Reilly Media, Inc. All rights reserved.

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

ISBN 978-1-098-12061-0 (англ.)

© 2023 Max Pumperla and O'Reilly Media, Inc.

ISBN 978-6-01083-430-9 (казах.)

© Перевод, оформление, издание,  
Books.kz, 2023

*Посвящается Альме*

# Содержание

<b>От издательства</b> .....	12
<b>Об авторах</b> .....	13
<b>Колофон</b> .....	14
<b>Предисловие</b> .....	15
<b>Введение</b> .....	16

## **Глава 1. Общий обзор фреймворка Ray**..... 23

Что такое Ray? .....	24
Что привело к разработке Ray?.....	25
Принципы внутреннего устройства фреймворка Ray .....	26
Простота и абстракция .....	26
Гибкость и неоднородность .....	27
Скорость и масштабируемость.....	28
Три слоя: ядро, библиотеки и экосистема .....	28
Фреймворк распределенных вычислений .....	29
Комплект библиотек науки о данных .....	32
Инструментарий Ray AIR и рабочий процесс науки о данных .....	32
Обработка данных с использованием библиотеки Ray Data.....	34
Тренировка моделей .....	36
Обучение с подкреплением с помощью библиотеки Ray RLLib.....	36
Распределенная тренировка с помощью библиотеки Ray Train .....	40
Гиперпараметрическая настройка.....	40
Подача моделей в качестве служб .....	42
Растущая экосистема.....	44
Резюме .....	45

## **Глава 2. Начало работы с инструментарием Ray Core**..... 47

Введение в инструментарий Ray Core .....	48
Первый пример использования Ray API .....	50
Функции и дистанционные задания Ray .....	52
Использование хранилища объектов с помощью put и get .....	53

---

Применение функции wait фреймворка Ray для неблокирующих вызовов .....	54
Оперирование зависимостями заданий .....	56
Из классов в акторы .....	57
Краткий обзор API инструментария Ray Core .....	60
Понимание системных компонентов фреймворка Ray .....	61
Планирование и исполнение работы на узле .....	61
Головной узел .....	63
Распределенное планирование и исполнение .....	64
Простой пример использования парадигмы MapReduce с фреймворком Ray .....	66
Отображение и перетасовка данных в документах .....	69
Редукция количеств слов .....	70
Резюме .....	72

## **Глава 3. Разработка первого распределенного приложения ..... 73**

Введение в обучение с подкреплением .....	73
Постановка простой задачи о лабиринте .....	75
Разработка симуляции .....	80
Тренировка модели обучения с подкреплением .....	83
Разработка распределенного приложения Ray .....	87
Резюмирование терминологии обучения с подкреплением .....	90
Резюме .....	92

## **Глава 4. Обучение с подкреплением с использованием библиотеки Ray RLlib ..... 93**

Краткий обзор библиотеки RLlib .....	94
Начало работы с библиотекой RLlib .....	95
Разработка среды в рамках библиотеки Gym .....	96
Работа с интерфейсом командной строки библиотеки RLlib .....	97
Использование Python API библиотеки RLlib .....	99
Тренировка алгоритмов библиотеки RLlib .....	99
Сохранение, загрузка и оценивание моделей библиотеки RLlib .....	101
Вычисление действий .....	102
Доступ к политике и модельным состояниям .....	103
Конфигурирование экспериментов с помощью библиотеки RLlib .....	106
Конфигурирование ресурсов .....	108
Конфигурирование работников розыгрыша .....	108
Конфигурирование сред .....	109
Работа со средами библиотеки RLlib .....	109
Общий обзор сред библиотеки RLlib .....	109
Работа с несколькими агентами .....	110
Работа с серверами политик и клиентами .....	115

Определение сервера .....	115
Определение клиента .....	117
Продвинутые концепции.....	118
Разработка продвинутой среды .....	118
Применение процедуры усвоения учебной программы .....	120
Работа с офлайновыми данными .....	122
Другие продвинутые темы .....	123
Резюме .....	124

## Глава 5. Гиперпараметрическая оптимизация с использованием библиотеки Ray Tune .....

125

Настройка гиперпараметров.....	126
Разработка примера случайного поиска с помощью фреймворка Ray.....	126
В чем трудность гиперпараметрической оптимизации? .....	129
Введение в библиотеку Tune .....	130
Принцип работы библиотеки Tune .....	131
Алгоритмы поиска .....	133
Планировщики.....	134
Конфигурирование и выполнение библиотеки Tune .....	136
Детализация ресурсов.....	136
Функции обратного вызова и метрики.....	137
Контрольные точки, остановка и возобновление .....	139
Конкретно-прикладные и условные пространства поиска .....	140
Машинное обучение с помощью библиотеки Tune.....	141
Использование библиотеки RLLib вместе с библиотекой Tune.....	141
Настройка моделей Keras.....	142
Резюме .....	145

## Глава 6. Обработка данных с использованием фреймворка Ray .....

147

Библиотека Ray Data .....	148
Основы библиотеки Ray Data.....	149
Создание набора данных Dataset .....	150
Чтение из хранилища и запись в него.....	150
Встроенные преобразования .....	151
Блоки и реорганизация блоков .....	152
Схемы и форматы данных .....	152
Вычисления на наборах данных Dataset .....	153
Конвейеры наборов данных Dataset .....	155
Пример: параллельная тренировка копий классификатора.....	157
Интеграции с внешними библиотеками .....	161
Разработка конвейера машинного обучения .....	164
Резюме .....	166

---

<b>Глава 7. Распределенная тренировка с использованием библиотеки Ray Train.....</b>	167
Основы распределенной тренировки моделей .....	168
Введение в библиотеку Ray Train на примере .....	169
Предсказание больших чаевых в поездках на нью-йоркском такси .....	170
Загрузка/предобработка данных и выделение признаков.....	171
Определение модели глубокого обучения .....	172
Распределенная тренировка с помощью библиотеки Ray Train.....	173
Распределенное пакетное генерирование модельных предсказаний .....	176
Подробнее о тренерах в библиотеке Ray Train .....	177
Миграция в библиотеку Ray Train с минимальными изменениями исходного кода .....	179
Горизонтальное масштабирование тренеров .....	180
Предобработка с помощью библиотеки Ray Train.....	181
Интеграция тренеров с библиотекой Ray Tune .....	183
Использование обратных вызовов для мониторинга тренировки .....	185
Резюме .....	185
<b>Глава 8. Онлайновое генерирование модельных предсказаний с использованием библиотеки Ray Serve .....</b>	187
Ключевые характеристики онлайнового генерирования модельных предсказаний.....	189
Модели машинного обучения характерны своей вычислительной интенсивностью.....	189
Модели машинного обучения бесполезны в изоляции .....	190
Введение в библиотеку Ray Serve .....	191
Архитектурный обзор .....	191
Определение базовой конечной точки HTTP .....	193
Масштабирование и ресурсное обеспечение .....	195
Пакетирование запросов .....	197
Графы генерирования многомодельных предсказаний.....	198
Ключевая функциональность: привязка нескольких развертываний.....	199
Шаблон 1: конвейеризация .....	200
Шаблон 2: широковещательная трансляция .....	201
Шаблон 3: условная логика .....	201
Сквозной пример: разработка API на базе обработки естественного языка .....	202
Доставка содержимого и предобработка .....	204
Модели обработки естественного языка.....	204
Обработка HTTP и логика драйвера.....	206
Собираем все воедино.....	208
Резюме .....	210

<b>Глава 9. Кластеры Ray .....</b>	211
Создание кластера Ray в ручном режиме.....	212
Развертывание на Kubernetes .....	214
Настройка своего первого кластера KubeRay .....	215
Взаимодействие с кластером KubeRay .....	216
Выполнение программ Ray с помощью команды kubectl.....	217
Использование сервера подачи заявок Ray на выполнение работы .....	217
Клиент Ray .....	218
Предоставление оператора KubeRay .....	219
Конфигурирование оператора KubeRay .....	219
Конфигурирование журналирования для KubeRay .....	222
Использование инструмента запуска кластеров Ray .....	223
Конфигурирование своего кластера Ray .....	224
Использование CLI-инструмента запуска кластеров .....	224
Взаимодействие с кластером Ray .....	225
Работа с облачными кластерами .....	225
AWS .....	225
Использование других облачных провайдеров .....	226
Автомасштабирование.....	227
Резюме .....	228
 <b>Глава 10. Начало работы с инструментарием Ray AI Runtime ...</b>	229
Зачем использовать инструментарий AIR? .....	229
Ключевые концепции инструментария AIR на примере.....	231
Наборы данных Dataset и предобработчики .....	232
Тренеры.....	233
Настройщики и контрольные точки .....	235
Пакетные предсказатели .....	237
Развертывания .....	238
Рабочие нагрузки, подходящие для инструментария AIR.....	241
Исполнение рабочих нагрузок AIR .....	244
Исполнение без отслеживания внутреннего состояния.....	244
Исполнение с отслеживанием внутреннего состояния.....	245
Исполнение составной рабочей нагрузки .....	245
Исполнение заданий по онлайновому генерированию модельных предсказаний .....	246
Управление памятью в инструментарии AIR.....	246
Принятая в инструментарии AIR модель сбоя .....	247
Автомасштабирование рабочих нагрузок AIR .....	248
Резюме .....	249
 <b>Глава 11. Экосистема фреймворка Ray и за ее пределами.....</b>	250
Растущая экосистема.....	251
Загрузка и предобработка данных.....	251

Тренировка моделей .....	253
Подача моделей в качестве служб .....	257
Разработка конкретно-прикладных интеграций .....	260
Обзор интеграций фреймворка Ray .....	262
Фреймворк Ray и другие системы .....	262
Фреймворки распределенных вычислений на Python .....	263
Инструментарий Ray AIR и более широкая экосистема машинного обучения.....	263
Как интегрировать инструментарий AIR в свою платформу машинного обучения .....	266
Куда отсюда двигаться дальше? .....	267
Резюме .....	269
<b>Тематический указатель .....</b>	<b>270</b>

# От издательства

## ***Отзывы и пожелания***

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

Вы можете написать отзыв на нашем сайте [www.dmkpress.com](http://www.dmkpress.com), зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу [dmkpress@gmail.com](mailto:dmkpress@gmail.com); при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу [http://dmkpress.com/authors/publish\\_book/](http://dmkpress.com/authors/publish_book/) или напишите в издательство по адресу [dmkpress@gmail.com](mailto:dmkpress@gmail.com).

## ***Список опечаток***

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу [dmkpress@gmail.com](mailto:dmkpress@gmail.com). Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

## ***Нарушение авторских прав***

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты [dmkpress@gmail.com](mailto:dmkpress@gmail.com).

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

# Об авторах

**Макс Пумперла** – преподаватель информатики и инженер-программист из Гамбурга, Германия. Активно работает с открытым исходным кодом, сопровождает несколько пакетов Python, автор книг по машинному обучению, выступает с докладами на международных конференциях. В настоящее время он работает инженером-программистом в Anyscale. Занимая должность руководителя отдела исследований продуктов в Pathmind Inc., он разрабатывал программные решения на основе обучения с подкреплением для масштабного промышленного применения, используя библиотеки Ray RLlib, Serve и Tune. Макс был основным разработчиком DL4J в Skymind, помогал развивать и расширять экосистему Keras и занимается техническим сопровождением Hyperopt.

**Эдвард Оукс** – инженер-программист и руководитель коллектива разработчиков в Anyscale, где возглавляет разработку библиотеки Ray Serve и является одним из ведущих разработчиков фреймворка Ray с открытым исходным кодом. До перехода в Anyscale учился в аспирантуре на факультете EECS Калифорнийского университета в Беркли.

**Ричард Ляо** – инженер-программист в Anyscale, работающий над инструментами с открытым исходным кодом для распределенного машинного обучения. Находится в отпуске по окончании аспирантуры при факультете вычислительных наук Калифорнийского университета в Беркли, где учился под кураторством Джозефа Гонсалеса, Ионы Стойцы и Кена Голдберга.

# Колофон

На обложке книги «Изучаем фреймворк Ray» показан мраморный электрический скат (*Torpedo marmorata*), также именуемый торпедным скатом. Мраморные электрические скаты встречаются в восточной части Атлантического океана от Африки до Норвегии, а также в Средиземном море. Они обитают на дне, предпочитая жить на мелководье или умеренно глубокой воде, на скалистых рифах, зарослях морской травы и илистых отмелях.

Мраморные электрические скаты имеют коричневые и черные крапинки, что позволяет им маскироваться в мутных водах, где они прячутся днем. Ночью скаты выходят на охоту и добывают корм в виде мелкой рыбы, такой как бычки, кефаль, макрель и ракообразные. Они могут вырасти до 2 футов в длину, расширять свои челюсти, чтобы заглатывать рыбу размером больше их рта, и убивать электрическим разрядом напряжением до 200 В.

Поскольку эти скаты способны поражать электрическим током, у них мало естественных хищников. Многие животные на обложках издательства O'Reilly находятся под угрозой исчезновения, и все они важны для нашего мира.

Иллюстрация на обложке выполнена Карен Монтгомери по мотивам стаинной линейной гравюры из «Королевской естественной истории» Лайдекера.

# Предисловие

За последнее десятилетие вычислительные потребности машинного обучения и приложений по обработке данных значительно превзошли возможности одного сервера или одного центрального процессора, включая аппаратные ускорители, такие как графические и тензорные процессоры. Данный тренд не оставляет иного выбора, кроме как выполнять эти приложения распределенно. К сожалению, создание таких распределенных приложений общеизвестно характеризуется чрезвычайной сложностью.

За последние несколько лет фреймворк распределенных вычислений Ray получал все большее предпочтение в связи со своей способностью упрощать разработку таких приложений. Ray включает в себя гибкое ядро и набор мощных библиотек, которые позволяют разработчикам легко масштабировать различные рабочие нагрузки, включая тренировку, гиперпараметрическую настройку, обучение с подкреплением, подачу моделей в качестве служб и пакетную обработку неструктурированных данных. Фреймворк Ray является одним из самых популярных проектов с открытым исходным кодом и используется тысячами компаний для внедрения широкого спектра вычислительных решений, от платформ машинного обучения до рекомендательных систем, систем обнаружения мошенничества и тренировки крупнейших моделей, в том числе ChatGPT компании Open AI.

В этой книге Макс Пумперла, Эдвард Оукс и Ричард Ляо проделали выдающуюся работу, предоставив щадящее и всестороннее введение во фреймворк Ray и его библиотеки с использованием простых для понимания примеров. К концу книги вы освоите ключевые концепции и абстракции фреймворка Ray и сможете разрабатывать и быстро масштабировать сквозные приложения машинного обучения от уровня вашего ноутбука до уровня больших локальных кластеров или облака.

– Ион Стойца

Соучредитель Anyscale и Databricks,  
а также профессор Калифорнийского университета  
в Беркли, Калифорния

Январь 2023 года

# Введение

Распределенные вычисления – увлекательнейшая тема. Оглядываясь назад на первые годы вычислительной техники, нельзя не впечатлиться тем фактом, что сегодня так много компаний вовлечены в распределение своих рабочих нагрузок по кластерам компьютеров. Впечатляет то, что для этого были найдены эффективные способы, но горизонтальное масштабирование также становится все более насущной необходимостью. Отдельные компьютеры выполняют свою работу все быстрее, и тем не менее потребность в крупномасштабных вычислениях продолжает превышать возможности отдельных машин.

Признавая, что масштабирование является одновременно необходимостью и вызовом времени, фреймворк Ray призван упростить разработчикам распределенные вычисления. Благодаря ему распределенные вычисления стали доступными для неспециалистов и стало довольно легко масштабировать скрипты Python по нескольким узлам. Фреймворк Ray хорошо зарекомендовал себя в масштабировании *вычислительно интенсивных* рабочих нагрузок и рабочих нагрузок *интенсивных по использованию данных*, таких как предобработка данных и тренировка моделей, и он непосредственно ориентирован на рабочие нагрузки машинного обучения, требующие масштабирования. Хотя сегодня эти два типа рабочих нагрузок можно масштабировать без Ray, вам, скорее всего, для каждого из них придется использовать разные API и распределенные системы. А управление несколькими распределенными системами приобретает запутанный характер и во многих отношениях становится неэффективным.

Добавление инструментария Ray AI Runtime (AIR) с выпуском Ray 2.0 в августе 2022 года еще больше расширило поддержку сложных рабочих нагрузок машинного обучения в Ray. Инструментарий AIR – это набор библиотек и инструментов, которые упрощают создание и развертывание сквозных приложений машинного обучения в единой распределенной системе. С помощью AIR даже самые сложные рабочие процессы обычно можно выражать в виде *одного-единственного скрипта Python*. Это означает, что появляется возможность сначала выполнять свои программы локально, что существенно влияет на скорость отладки и разработки.

Исследователи данных извлекают выгоду из фреймворка Ray за счет того, что они могут опираться на растущую экосистему библиотек фреймворка, связанных с машинным обучением, и сторонних интеграций. Инструментарий Ray AIR помогает быстро прототипировать идеи и легче переходить от разработки к производству. В отличие от многих других распределенных систем, фреймворк Ray также имеет встроенную поддержку графических процессоров, что бывает особенно важно для таких профессий, как инженеры машинного обучения. Поддерживая инженеров данных, Ray также имеет

тесную интеграцию с такими инструментами, как Kubernetes, и может развертываться в многооблачных конфигурациях.

Кроме того, указанный фреймворк можно использовать как единый вычислительный слой для обеспечения масштабирования, отказоустойчивости, планирования и оркестровки рабочих нагрузок. Другими словами, вам непременно стоит потратить усилия на изучение фреймворка *Ray* в рамках различных ИТ-профессий.

## Кому следует прочитать эту книгу

Вполне вероятно, что вы взяли в руки эту книгу, потому что вас интересуют некоторые аспекты фреймворка *Ray*. Возможно, вы – инженер распределенных систем, который хочет знать, как работает движок фреймворка *Ray*. Вы также можете быть разработчиком программного обеспечения, заинтересованным в освоении новой технологии. Или же инженером данных, который хочет оценить *Ray* в сравнении с аналогичными инструментами. Вы также, вероятно, являетесь практиком машинного обучения или исследователем данных, кото-рому необходимо найти способы масштабирования экспериментов.

Независимо от вашей конкретной ИТ-профессии, общим знаменателем, позволяющим извлечь максимальную пользу из этой книги, является сносное владение языком программирования Python. Точнее говоря, учитывая, что примеры исходного кода в этой книге написаны на Python, от вас требуется знание данного языка на среднем уровне. Как питонщик, вы прекрасно знаете, что явное лучше неявного. Так что давайте будем откровенны, под владением языком Python подразумевается, что вы умеете использовать командную строку в своей системе, получать помощь в случае сбоя и самостоятельно настраивать среду программирования.

Если вы никогда не работали с распределенными системами раньше, то ничего страшного. В книге будут охвачены все основы, необходимые для того, чтобы начать работать с ними. Кроме того, вы сможете выполнять большинство представленных в книге примеров исходного кода на своем ноутбуке. Знакомство с основами означает, что мы не сможем остановиться на распределенных системах слишком подробно. Эта книга в конечном счете ориентирована на разработчиков приложений, использующих фреймворк *Ray*, в особенности приложений в области науки о данных и машинного обучения.

Для усвоения последующих глав этой книги вам потребуется некоторое знакомство с машинным обучением, но мы не исходим из того, что вы работали в этой области. В частности, вы должны иметь базовое представление о парадигме машинного обучения и о том, чем она отличается от традиционного программирования. Вы также должны знать основы использования библиотек NumPy и Pandas. Кроме того, вы должны, по крайней мере, чувствовать себя комфортно, читая примеры с использованием популярных библиотек TensorFlow и PyTorch. Будет вполне достаточно уметь следить за ходом выполнения исходного кода на уровне API, но вам не нужно уметь писать свои собственные модели. Мы рассмотрим примеры с использованием

обеих доминирующих библиотек глубокого обучения (TensorFlow и PyTorch), чтобы проиллюстрировать возможности применения фреймворка Ray для рабочих нагрузок машинного обучения, независимо от предпочтаемого вами фреймворка.

Мы будем подробно останавливаться на продвинутых темах машинного обучения, но основное внимание будет уделено фреймворку Ray как технологии и тому, как его использовать. Обсуждаемые примеры машинного обучения, возможно, будут для вас новыми и потребуют повторного прочтения, но вы все равно сможете сосредоточиться на API фреймворка Ray и на том, как его использовать на практике. С учетом описанных выше технических требований вот что вы могли бы извлечь из этой книги:

- если вы – исследователь данных, то фреймворк Ray откроет для вас новые способы обдумывания и разработки распределенных приложений машинного обучения. Вы научитесь выполнять подбор гиперпараметров для масштабируемых экспериментов, получите практические знания по крупномасштабной тренировке моделей и познакомитесь с современной библиотекой обучения с подкреплением;
- если вы – инженер данных, то вы научитесь использовать библиотеку Ray Data для крупномасштабного приема данных, улучшать свои конвейеры, используя такие инструменты, как библиотека Dask on Ray, и эффективно развертывать модели в крупном масштабе;
- если вы – инженер, то вы поймете, как фреймворк Ray работает под капотом, как запускать и масштабировать кластеры Ray в облаке и как использовать Ray для разработки приложений, интегрированных с известными вам проектами.

Конечно же, вы можете изучить все эти темы независимо от вашей профессии. Мы надеемся, что к концу данной книги вы узнаете достаточно для того, чтобы оценить фреймворк Ray по достоинству за все его сильные стороны.

## ЦЕЛИ ЭТОЙ КНИГИ

Эта книга была написана в первую очередь для читателей, которые не знакомы с фреймворком Ray и хотят быстро получить от него максимальную отдачу. Мы подобрали материал таким образом, чтобы вы поняли ключевые идеи, лежащие в основе Ray, и научились использовать его главные строительные блоки. Прочитав ее, вы будете без особого труда самостоятельно ориентироваться в более сложных темах, которые выходят за рамки этого введения.

Вам также следует четко представлять, чем наша книга не является. Она написана не для того, чтобы предоставить вам как можно больше информации, например справочные материалы по API или исчерпывающие руководства. Она также написана не для того, чтобы помогать вам решать конкретные задачи, как это делается в практических руководствах или книгах рецептов. Излагаемый в этой книге материал ориентирован на изучение и понимание фреймворка Ray и дает вам интересные примеры стартового уровня.

Программное обеспечение быстро развивается и устаревает, но фундаментальные концепции, лежащие в его основе, нередко остаются стабильными даже в течение основных циклов релиза. Здесь мы пытаемся найти баланс между передачей идей и предоставлением конкретных примеров исходного кода. Идеи, которые вы найдете в нашей книге, в идеале останутся полезными даже тогда, когда исходный код в конечном итоге потребует обновления.

В то время как документация фреймворка Ray продолжает совершенствоваться, мы убеждены, что книги обладают качествами, которые трудно найти в проектной документации. Раз уж вы читаете эти строки, мы понимаем, что, возможно, заявляя это, мы ломимся в и без того уже открытые двери. Но следует признать, что некоторые самые лучшие известные нам технические книги пробуждают интерес к проекту и вызывают у читателя желание ознакомиться с краткими справочными материалами по API, к которым он никогда бы не прикоснулся в противном случае. Мы надеемся, что это одна из таких книг.

## НАВИГАЦИЯ ПО ЭТОЙ КНИГЕ

Мы организовали эту книгу таким образом, чтобы провести вас в естественном порядке от ключевых концепций к более сложным темам фреймворка Ray. Многие описываемые в ней идеи сопровождаются примерами исходного кода, которые находятся в репозитории книги на GitHub<sup>1</sup>.

В двух словах: в первых трех главах книги излагаются основы фреймворка Ray как фреймворка распределенных вычислений на Python с практическими примерами. Главы с 4 по 10 знакомят с высокоуровневыми библиотеками Ray и показывают, как разрабатывать приложения с их помощью. Последняя глава дает исчерпывающий обзор экосистемы Ray и показывает, куда двигаться дальше. Вот чего вы можете ожидать от каждой главы:

### *Глава 1. Краткий обзор фреймворка Ray*

Знакомит вас с фреймворком Ray как с системой, состоящей из трех слоев: ее ядра, библиотек и экосистемы машинного обучения. В этой главе вы выполните свои первые примеры с библиотеками Ray, чтобы получить представление о том, что вообще можно делать с фреймворком Ray.

### *Глава 2. Начало работы с инструментарием Ray Core*

Рассказывает об основах проекта Ray, а именно о его стержневом API. В ней также обсуждается, как задания и акторы Ray естественным образом расширяются из функций и классов Python. Вы также узнаете о компонентах системы Ray и о том, как они работают вместе.

### *Глава 3. Разработка первого распределенного приложения*

Проведет по реализации приложения распределенного обучения с подкреплением с помощью инструментария Ray Core. Вы реализуете это при-

---

<sup>1</sup> См. [https://oreil.ly/learning\\_ray\\_repo](https://oreil.ly/learning_ray_repo).

ложение с чистого листа и увидите на практике гибкость фреймворка Ray в распределении вашего исходного кода Python.

#### *Глава 4. Обучение с подкреплением с использованием библиотеки Ray RLlib*

Дает краткое введение в обучение с подкреплением и показывает, как в библиотеке RLlib фреймворка Ray реализованы важные концепции. Собрав вместе несколько примеров, мы также перейдем к более сложным темам, таким как усвоение агентом учебной программы или работа с офлайновыми данными.

#### *Глава 5. Гиперпараметрическая оптимизация с использованием библиотеки Ray Tune*

Рассказывает о том, почему сложно эффективно настраивать гиперпараметры, как библиотека Ray Tune работает в концептуальном плане и как ее использовать на практике для своих проектов машинного обучения.

#### *Глава 6. Обработка данных с использованием фреймворка Ray*

Знакомит с абстракцией наборов данных как объекта Dataset в библиотеке Ray Data и с тем, как они вписываются в ландшафт других систем обработки данных. Вы также научитесь работать со сторонними интеграциями, такими как библиотека Dask on Ray.

#### *Глава 7. Распределенная тренировка с использованием библиотеки Ray Train*

Знакомит с основами распределенной тренировки моделей и показывает, как использовать библиотеку Ray Train с фреймворками машинного обучения, такими как PyTorch. Мы также покажем, как добавлять конкретно-прикладных предобработчиков в свои модели, как отслеживать тренировку с помощью функций обратного вызова и как настраивать гиперпараметры своих моделей с помощью библиотеки Tune.

#### *Глава 8. Онлайновое генерирование модельных предсказаний с использованием библиотеки Ray Serve*

Научит вас основам выставления натренированных моделей машинного обучения в качестве конечных точек API, к которым можно обращаться из любого места. Мы обсудим библиотеку Ray Serve и то, как она решает сложности онлайнового генерирования модельных предсказаний, рассмотрим ее архитектуру и покажем, как ее использовать на практике.

#### *Глава 9. Кластеры Ray*

Познакомит с конфигурированием, запуском и масштабированием кластеров Ray в своих приложениях. Вы узнаете об интерфейсе командной строки (CLI) для запуска кластеров и автомасштабировщике Ray, а также о том, как настраивать кластеры в облаке. Мы также покажем, как развертывать Ray в Kubernetes и с помощью других менеджеров кластеров.

#### *Глава 10. Начало работы с инструментарием Ray AI Runtime*

Знакомит с унифицированным набором инструментов для рабочих нагрузок машинного обучения под названием Ray AIR, который предлагает целый ряд сторонних интеграций для тренировки моделей или доступа к конкретно-прикладным источникам данных.

## Глава 11. Экосистема фреймворка Ray и за ее пределами

Дает краткий обзор многих интересных расширений и интеграций, которые привлекались во фреймворке Ray на протяжении многих лет.

# КАК ИСПОЛЬЗОВАТЬ ПРИМЕРЫ ИСХОДНОГО КОДА

Весь исходный код этой книги находится в ее репозитории на GitHub<sup>1</sup>. В репозитории GitHub имеется папка *notebook* с блокнотами Jupyter каждой главы. Мы построили примеры таким образом, что вы можете либо набирать исходный код по ходу чтения, либо следовать основному изложению и выполнять исходный код из GitHub в другое время. Выбор за вами.

Если говорить о примерах из книги, то мы исходим из того, что у вас установлен Python 3.7 или более поздняя его версия. На момент написания книги поддержка Python 3.10 для фреймворка Ray является экспериментальной, поэтому в настоящее время мы можем рекомендовать только версию Python не позднее 3.9. Все примеры исходного кода основаны на допущении о том, что у вас установлен фреймворк Ray, и каждая глава добавляет свои собственные специфические требования. Примеры были протестированы на Ray версии 2.2.0, и мы рекомендуем вам придерживаться этой версии на протяжении всей книги.

# ИСПОЛЬЗУЕМЫЕ В КНИГЕ УСЛОВНЫЕ ОБОЗНАЧЕНИЯ

В книге используются следующие типографские условные обозначения:

### *курсивный шрифт*

обозначает новые термины, URL-адреса, адреса электронной почты, имена файлов и расширения файлов;

### моноширинный шрифт

используется для листингов программ, а также внутри абзацев для ссылки на элементы программ, такие как переменные или имена функций, базы данных, типы данных, переменные среды, инструкции и ключевые слова;

### моноширинный жирный шрифт

показывает команды или другой текст, который пользователь должен вводить буквально;

### <текст в угловых скобках>

должен быть заменен значениями, предоставленными пользователем, или значениями, определяемыми контекстом.

 Данный элемент обозначает общее замечание.

 Данный элемент обозначает предупреждение или предостережение.

---

<sup>1</sup> См. [https://oreil.ly/learning\\_ray\\_repo](https://oreil.ly/learning_ray_repo).

## ИСПОЛЬЗОВАНИЕ ПРИМЕРОВ ИСХОДНОГО КОДА

Дополнительные материалы (примеры исходного кода, упражнения и т. д.) доступны для скачивания по адресу [https://oreil.ly/learning\\_ray\\_repo](https://oreil.ly/learning_ray_repo).

Если у вас есть технический вопрос или проблема с использованием примеров исходного кода, то, пожалуйста, отправьте электронное письмо по адресу [bookquestions@oreilly.com](mailto:bookquestions@oreilly.com).

## БЛАГОДАРНОСТИ

Мы хотели бы поблагодарить весь коллектив издательства O'Reilly за помощь в создании этой книги. В частности, мы хотели бы поблагодарить нашего неутомимого редактора Джеффа Блейела за бесценный вклад и обратную связь. Большое спасибо Джесс Хаберман за плодотворные дискуссии и непредвзятость на ранних стадиях процесса. Мы благодарим Кэтрин Тозер, Челси Фостер и Кассандру Фуртадо, а также многих других сотрудников O'Reilly.

Большое спасибо всем рецензентам за их ценные отзывы и предложения: Марку Саруфиму, Кевину Фергюсону, Адаму Брейндalu и Хорхе Давила-Чакону. Мы хотели бы поблагодарить многих коллег из Anyscale, которые помогали нам с книгой в любом качестве, включая Свена Мику, Стефани Ванг, Антони Баума, Кристи Бергман, Дмитрия Гехтмана, Чжэ Чжана и многих других.

Вдобавок ко всему мы хотели бы от всего сердца поблагодарить коллектив разработчиков и сообщество фреймворка Ray за их поддержку и отзывы, а также многих ключевых интересантов в Anyscale, поддерживающих этот проект.

Я (Макс) также хотел бы поблагодарить коллектив Pathmind за их поддержку на ранних этапах проекта, в особенности Криса Николсона, который за эти годы был гораздо полезнее, чем я мог бы здесь описать. Особая благодарность обществу Espresso в Винтерхуде за помощь в превращении кофе в книги, а также за растущий набор инструментов на основе GPT-3, помогавших мне заканчивать на полуслове, когда действие кофеина заканчивалось. Я также хотел бы выразить свою благодарность моей семье за их поддержку и терпение. Как всегда, ничего из этого не было бы возможно без Энн, которая постоянно поддерживает меня, когда это важно, даже если я берусь за слишком большое число проектов, подобных данному.

# Глава 1

---

## Общий обзор фреймворка Ray

Одна из причин, по которой нам нужны эффективные распределенные вычисления, заключается в том, что мы собираем все больше разнообразных данных с возрастающей скоростью. Появившиеся за последнее десятилетие системы хранения данных, механизмы их обработки и анализа имеют решающее значение для успеха многих компаний. Интересно, что большинство технологий «больших данных» разрабатываются и эксплуатируются инженерами (данных), которые отвечают за задания по сбору и обработке данных. Идея состоит в том, чтобы освободить исследователей данных и дать им возможность делать то, что у них получается лучше всего. У вас, как исследователя данных, возможно, возникнет желание сосредоточиться на тренировке сложных моделей машинного обучения, эффективном подборе гиперпараметров, разработке совершенно новых конкретно-прикладных моделей или симуляций либо подаче своих моделей в качестве служб с целью их демонстрации.

В то же время потребность в масштабировании этих рабочих нагрузок до вычислительного кластера может оказаться *неизбежной*. В этой связи выбранная вами распределенная система должна поддерживать выполнение всех этих мелкозернистых заданий «больших вычислений» потенциально на специализированном оборудовании. В идеале она также должна вписываться в используемую вами цепочку инструментов больших данных и быть достаточно быстрой, чтобы соответствовать вашим требованиям к задержке. Другими словами, распределенные вычисления должны быть достаточно мощными и гибкими, соответствуя сложным рабочим нагрузкам науки о данных, – и фреймворк Ray способен в этом помочь.

Сегодня Python, по всей видимости, является самым популярным языком науки о данных; он, безусловно, считается наиболее полезным для нашей повседневной работы. Языку Python уже более 30 лет, но его активное сообщество по-прежнему растет и развивается. Богатая экосистема PyData<sup>1</sup> является неотъемлемой частью инструментария исследователя данных. Ка-

---

<sup>1</sup> См. <https://pydata.org/>.

ким образом гарантированно обеспечить масштабирование своих рабочих нагрузок, при этом используя необходимые инструменты? Это сложная задача, в особенности с учетом того, что сообщества разработчиков нельзя просто вынудить отказаться от своего набора инструментов или языка программирования. Следовательно, инструменты науки о данных для распределенных вычислений должны разрабатываться под их существующее сообщество.

## Что такое Ray?

Во фреймворке Ray нам нравится то, что он проставляет все эти галочки. Это гибкий фреймфорк распределенных вычислений, разработанный под сообщество Python в области науки о данных.

С фреймворком Ray легко начинать работу, и он не усложняет простые вещи. Его базовый API прост настолько, насколько это возможно, и помогает эффективно рассуждать о распределенных программах, которые вы хотите писать. Он позволяет эффективно параллелизовать программы Python на своем ноутбуке и выполнять локально протестированный исходный код в кластере практически без каких-либо изменений. Его высокоуровневые библиотеки легко конфигурируемы и используются бесшовно вместе. Некоторые из них, такие как библиотека обучения с подкреплением, вероятно, будут иметь блестящее будущее в качестве самостоятельных проектов, независимо от того, выполняются они распределенно или нет. Хотя ядро фреймворка Ray было разработано на C++, с самого первого дня это был фреймворк на основе стратегии «сначала Python»<sup>1</sup>, который интегрируется со многими важными инструментами науки о данных и может опираться на растущую экосистему.

Распределенные вычисления на Python не новы, и Ray – не первый фреймворк в этой области (и не последний), но его особенность в том, что именно он способен предложить. Ray особенно эффективен при комбинировании нескольких его модулей и наличии конкретно-прикладных рабочих нагрузок, связанных с машинным обучением, что в противном случае было бы трудно реализовать. За счет этого значительно упрощаются распределенные вычисления, чтобы гибко выполнять сложные рабочие нагрузки, используя инструменты Python, которые вы знаете и хотите использовать. Другими словами, *изучив Ray, вы знакомитесь с гибкими распределенными вычислениями на Python в области машинного обучения*. И эта книга учит вас тому, как это делается.

В данной главе вы получите первое представление о том, что фреймворк Ray вообще может делать. Мы обсудим три слоя, из которых состоит Ray: его стержневой движок, библиотеки высокого уровня и экосистему. В данной

---

<sup>1</sup> Под стратегией «сначала Python» (англ. Python-first) мы подразумеваем, что все библиотеки более высокого уровня пишутся на Python и что разработка новых функциональных возможностей обусловливается потребностями сообщества Python. Несмотря на это, фреймворк Ray был разработан с поддержкой привязок к нескольким языкам и, например, поставляется с Java API. Таким образом, не исключено, что Ray сможет поддерживать другие языки, являющиеся важными для экосистемы науки о данных.

главе мы сначала покажем примеры исходного кода, чтобы дать представление о фреймворке Ray. Эту главу можно просмотреть в качестве краткого ознакомления с книгой; мы отложим подробное рассмотрение API и компонентов фреймворка Ray до последующих глав.

## Что привело к разработке Ray?

Программировать распределенные системы трудно. Это требует определенных знаний и опыта, которых у вас может и не быть. В идеале такие системы не мешают и предоставляют абстракции, позволяющие вам сосредоточиться на своей работе. Но на практике, как отмечает Джоэл Спольски (Joel Spolsky)<sup>1</sup>, «все нетривиальные абстракции в той или иной степени негерметичны», и сделать так, чтобы кластеры компьютеров делали то, что вы хотите, несомненно, сложно. Многие программные системы требуют ресурсов, которые намного превышают возможности отдельных серверов. Даже если бы было достаточно одного сервера, современные системы должны быть отказоустойчивыми и обеспечивать такой функционал, как высокая доступность. Это означает, что вашим приложениям, возможно, придется работать на нескольких машинах или даже в центрах обработки данных, просто чтобы обеспечивать их надежную работу.

Даже если вы не слишком хорошо знакомы с машинным обучением или искусственным интеллектом в целом, вы, должно быть, слышали о недавних прорывных событиях в этой области. Назовем только два из них – в последнее время в новостях появились такие системы, как AlphaFold<sup>2</sup> исследовательской лаборатории Deepmind для решения задачи сворачивания белков и Codex<sup>3</sup> компании OpenAI, помогающая разработчикам программного обеспечения справляться с тяготами своей работы. Возможно, вы также слышали, что для тренировки систем машинного обучения обычно требуются большие объемы данных и что модели машинного обучения имеют тенденцию становиться все крупнее. В своей статье «Искусственный интеллект и вычисления»<sup>4</sup> разработчики из OpenAI продемонстрировали экспоненциальный рост вычислений, необходимых для тренировки моделей искусственного интеллекта. В их исследовании число операций, необходимых для систем искусственного интеллекта, измеряется в петафлопсах (тысячах триллионов операций в секунду) и с 2012 года удваивается каждые 3.4 месяца.

Сравните это с законом Мура<sup>5</sup>, который гласит, что число транзисторов в компьютерах удваивается каждые два года. Даже если вы настроены опти-

<sup>1</sup> См. <https://oreil.ly/mpzSe>.

<sup>2</sup> См. <https://oreil.ly/RFaMa>.

<sup>3</sup> См. <https://oreil.ly/vGnyh>.

<sup>4</sup> См. AI and Compute, [https://oreil.ly/7huR\\_](https://oreil.ly/7huR_).

<sup>5</sup> Закон Мура соблюдался долгое время, но имеются признаки того, что он замедляется. Некоторые даже говорят, что он больше не действует (<https://oreil.ly/fhPg->). Мы здесь не для того, чтобы оспаривать эти аргументы. Важно не то, что наши компьютеры в целом становятся все быстрее, а то, что существует соотношение с объемом необходимых нам вычислений.

мистично в отношении закона Мура, очевидно, что в машинном обучении существует явная потребность в распределенных вычислениях. Вы также должны понимать, что многие задачи машинного обучения могут быть естественным образом разложены под параллельное выполнение. Тогда почему бы не ускорить процесс, если это можно сделать<sup>1</sup>?

Распределенные вычисления обычно воспринимаются как сложные. В чем же причина? Разве не реалистично найти хорошие абстракции для выполнения своего исходного кода в кластерах без необходимости постоянно думать об отдельных машинах и о том, как они взаимодействуют между собой? Что, если сосредоточиваться непосредственно на рабочих нагрузках искусственного интеллекта?

Исследователи из лаборатории RISELab Калифорнийского университета в Беркли создали фреймворк Ray для решения этих вопросов. Они искали эффективные способы ускорить свои рабочие нагрузки путем их распределения. Рабочие нагрузки, которые они имели в виду, были довольно гибкими по своей природе и не вписывались в рамки, доступные в то время. Исследователи лаборатории RISELab также хотели разработать систему, в обязанности которой входили бы способы распределения работы. При наличии используемых по умолчанию разумных линий поведения исследователи должны иметь возможность сосредоточиваться на своей работе, независимо от специфики своего вычислительного кластера. И в идеале они должны иметь доступ ко всем своим любимым инструментам Python. По этой причине фреймворк Ray был разработан с акцентом на высокопроизводительные и разнородные рабочие нагрузки<sup>2</sup>. В целях более глубокого понимания этих моментов давайте подробнее рассмотрим философию внутреннего устройства фреймворка Ray.

## Принципы внутреннего устройства фреймворка Ray

Фреймворк Ray разработан, принимая во внимание несколько принципов внутреннего устройства. Его API сконструирован с учетом простоты и универсальности, а его вычислительная модель ориентирована на гибкость. Его системная архитектура сконструирована с учетом производительности и масштабируемости. Давайте рассмотрим каждый из этих принципов подробнее.

### *Простота и абстракция*

---

<sup>1</sup> Существует много способов ускорения тренировки моделей машинного обучения, от базовых до сложных. Например, в главе 6 мы потратим значительное количество времени на разработку распределенной обработки данных и в главе 7 – на распределенную тренировку модели.

<sup>2</sup> Компания Anyscale (<https://www.anyscale.com/>), в которой был разработан фреймворк Ray, создает управляемую платформу Ray и предлагает размещаемые на серверах решения для приложений Ray.

API фреймворка Ray не только отличается простотой, но и (как вы увидите в главе 2) интуитивно понятен в использовании. При этом не имеет значения, что вы хотите задействовать: все процессорные ядра своего ноутбука либо все машины в вашем кластере. Возможно, вам придется изменить одну или две строки исходного кода, но используемый вами исходный код Ray по сути останется прежним. И как в любой хорошей распределенной системе, Ray управляет распределением заданий и координацией незаметно для пользователя, «под капотом». Это здорово, потому что вам не приходится вязнуть в рассуждениях о механике распределенных вычислений. Хороший уровень абстракции позволяет сосредоточиваться на своей работе, и мы думаем, что Ray проделал отличную работу, предоставив его вам.

Поскольку API фреймворка Ray столь универсально применим и написан в *Python'овском стиле*, его легко интегрировать с другими инструментами. Например, акторы Ray могут вызывать существующие распределенные рабочие нагрузки Python или вызываться ими самими. В этом смысле Ray также является хорошим «склеивающим исходным кодом» для распределенных рабочих нагрузок, поскольку он достаточно производителен и гибок при взаимодействии между разными системами и фреймворками.

## Гибкость и неоднородность

Для рабочих нагрузок искусственного интеллекта, в частности при работе с такими парадигмами, как обучение с подкреплением, нужна гибкая модель программирования. API фреймворка Ray разработан таким образом, чтобы упрощать написание гибкого и компонуемого исходного кода. Проще говоря, если вы можете выразить свою рабочую нагрузку на Python, то вы сможете ее распределить с помощью фреймворка Ray. Разумеется, вам все равно нужно будет обеспечить достаточный объем доступных ресурсов и помнить о том, что вы хотите распределять. Но фреймворк Ray не ограничивает то, что вы можете с ним сделать.

Когда дело касается *неоднородности* вычислений, фреймворк Ray также обладает гибкостью. Например, допустим, вы работаете над сложной симуляцией. Симуляции обычно можно разбивать на несколько заданий, или шагов. Выполнение некоторых из этих шагов может занимать несколько часов, других – всего несколько миллисекунд, но их всегда нужно планировать и выполнять быстро. Иногда исполнение одного задания в симуляции может занимать много времени, но другие, более мелкие задания должны иметь возможность работать параллельно, не блокируя его. Кроме того, последующие задания могут зависеть от результата вышестоящего задания, поэтому нужен фреймворк, обеспечивающий *динамическое выполнение*, который хорошо справляется с зависимостями заданий. Фреймворк Ray представляет полную гибкость при выполнении подобных разнородных рабочих процессов.

Вам также необходимо обеспечивать гибкость в использовании ресурсов, и Ray поддерживает разнородное оборудование. Например, некоторые задания, возможно, придется выполнять на графическом процессоре, тогда

как другие лучше всего выполняются на паре процессорных ядер. Ray предоставляет такую гибкость.

## Скорость и масштабируемость

Еще одним принципом внутреннего устройства фреймворка Ray является скорость, с которой Ray исполняет свои задания. Он может обрабатывать миллионы заданий в секунду, и с ним вы получаете очень низкие задержки. Ray разработан так, чтобы выполнять свои задания с задержкой всего в миллисекунды.

Для того чтобы распределенная система была быстрой, она также должна хорошо масштабироваться. Фреймворк Ray эффективно распределяет и планирует ваши задания по всему вычислительному кластеру. И он также делает это в отказоустойчивом ключе. Как вы подробно узнаете в главе 9, кластеры фреймворка Ray поддерживают *автомасштабирование*, чтобы поддерживать высокоэластичные рабочие нагрузки. Автомасштабировщик Ray пытается запускать или останавливать машины в вашем кластере в соответствии с текущим спросом. Это помогает как минимизировать затраты, так и обеспечивать наличие в вашем кластере ресурсов, достаточных для выполнения вашей рабочей нагрузки.

В распределенных системах вопрос не в том, пойдет ли что-то не так, а в том, когда что-то пойдет не так. Машина может выйти из строя, прервать выполнение задания или просто воспламениться<sup>1</sup>. В любом случае, фреймворк Ray разработан с учетом быстрого восстановления после сбоев, что способствует его общей скорости.

Поскольку мы еще не говорили об архитектуре фреймворка Ray (глава 2 познакомит вас с ней), мы пока не можем рассказать о том, как эти принципы внутреннего устройства реализованы. Вместо этого давайте перенесем наше внимание на то, что фреймворк Ray может делать для вас на практике.

## Три слоя: ядро, библиотеки и экосистема

Теперь, когда вы знаете цели, с которыми был разработан фреймворк Ray, и что его создатели имели в виду, давайте рассмотрим три слоя фреймворка Ray. Такая подача – не единственный способ изложить суть дела, но именно подобный подход имеет наибольший смысл в этой книге:

- низкоуровневый фреймворк распределенных вычислений на Python со сжатым стержневым API и инструментарием для развертывания кластеров под названием Ray Core<sup>2</sup>;
- набор высокоуровневых библиотек, разработанных и поддерживаемых

---

<sup>1</sup> Это может показаться радикальным, но тут не до шуток. Вот лишь один пример: в марте 2021 года французский центр обработки данных, обслуживающий миллионы веб-сайтов, сгорел полностью ([https://oreil.ly/Nl9\\_o](https://oreil.ly/Nl9_o)). Если весь ваш кластер сгорит дотла, то мы боимся, что фреймворк Ray не сможет вам помочь.

<sup>2</sup> Это книга по Python, поэтому мы сосредоточимся исключительно на Python, но вы должны знать, что в Ray также есть Java API, который на данный момент имеет менее зрелую реализацию, чем его Python'овский эквивалент.

создателями фреймворка Ray. Сюда входит инструментарий Ray AIR, служащий для использования этих библиотек с помощью унифицированного API в обычных рабочих нагрузках машинного обучения;

- растущая экосистема интеграций и партнерских взаимодействий с другими известными проектами, которые охватывают многие аспекты первых двух слоев.

Здесь многое предстоит разобрать, и в оставшейся части этой главы мы рассмотрим каждый из этих слоев по отдельности.

Стержневой движок Ray с его API можно представить в центре, на котором строится все остальное. Библиотеки Ray для науки о данных построены поверх Ray Core и обеспечивают предметно-специфичный слой абстракции<sup>1</sup>. На практике многие исследователи данных будут использовать эти библиотеки напрямую, тогда как инженеры машинного обучения или инженеры платформ могут в значительной степени опираться на разработку своих собственных инструментов в качестве расширений API инструментария Ray Core. Инструментарий Ray AIR можно рассматривать как зонтик, который связывает библиотеки Ray и предлагает состыковывающийся фреймворк для работы с обычными рабочими нагрузками искусственного интеллекта. А растущее число сторонних интеграций для Ray является еще одной отличной отправной точкой для опытных практиков. Давайте рассмотрим каждый слой по очереди.

## ФРЕЙМВОРК РАСПРЕДЕЛЕННЫХ ВЫЧИСЛЕНИЙ

По своей сути Ray – это фреймворк распределенных вычислений. Здесь мы познакомим вас только с базовой терминологией, а в главе 2 подробно поговорим об архитектуре фреймворка Ray. Если говорить коротко, то Ray настраивает кластеры компьютеров и управляет ими таким образом, чтобы вы могли выполнять на них распределенные задания. Кластер Ray состоит из узлов, которые соединены друг с другом через сеть. Вы программируете, работая с так называемым *драйвером*, корнем программы, который находится на *головном узле*. Драйвер может исполнять *заявки на выполнение работы*<sup>2</sup>, то есть на выполнение набора заданий<sup>3</sup>, которые осуществляются в узлах кластера. В частности, отдельные задания заявки выполняются в *процессах-работниках на узлах-работниках*<sup>4</sup>. На рис. 1/1 показана базовая структура кластера Ray. Обратите внимание, что мы пока не рассматриваем связь между узлами; на этой диаграмме просто показана компоновка кластера Ray.

<sup>1</sup> Одна из причин, по которой столь много библиотек построено поверх Ray Core, заключается в том, что так очень компактно и просто рассуждать. Одна из целей данной книги – вдохновить вас на написание собственных приложений или даже библиотек с помощью Ray.

<sup>2</sup> Англ. job. – Прим. перев.

<sup>3</sup> Англ. task. – Прим. перев.

<sup>4</sup> Англ. worker node. – Прим. перев.

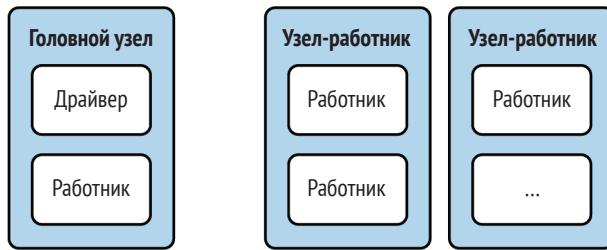


Рис. 1.1 ♦ Базовые компоненты кластера Ray

Интересно то, что кластер Ray также может быть *локальным кластером*, состоящим только из вашего собственного компьютера. В этом случае есть только один узел, а именно головной узел, в котором есть процесс драйвера и несколько процессов-работников. По умолчанию число процессов-работников равно числу центральных процессоров, имеющихся на вашем компьютере.

Имея эти знания под рукой, самое время заняться черновой работой и запустить свой первый локальный кластер Ray. Установка фреймворка Ray в любую основную операционную систему должна пройти без проблем с использованием менеджера пакетов pip:

```
pip install "ray[rllib, serve, tune]==2.2.0"
```

С помощью простой команды pip install ray вы установите только основной функционал фреймворка Ray. Поскольку мы хотим разведать некоторые расширенные функциональные возможности, мы установили «дополнительные» компоненты rllib, serve и tune, которые мы вскоре обсудим<sup>1</sup>. В зависимости от конфигурации вашей системы кавычки в этой команде установки вам, возможно, не понадобятся.

Далее продолжите и запустите сеанс Python. Вы могли бы, например, использовать интерпретатор ipython, который часто подходит для выполнения простых примеров. В своем сеансе Python теперь можно легко импортировать и инициализировать фреймворк Ray:

```
import ray
ray.init()
```

 Если вам не хочется набирать команды в терминале самостоятельно, то можете перейти к блокноту Jupyter этой главы и выполнить исходный код там. Выбор за вами, но в любом случае, пожалуйста, не забудьте использовать Python версии 3.9 или более ранней<sup>2</sup>.

<sup>1</sup> Обычно в этой книге мы вводим зависимости только тогда, когда они нам нужны, что должно облегчить дальнейшую работу. Напротив, блокноты Jupyter на GitHub (<https://oreil.ly/9ccz>) дают возможность устанавливать все зависимости заранее, чтобы вместо этого вы могли сосредоточиваться на выполнении исходного кода.

<sup>2</sup> На момент написания книги поддержка Ray в Python 3.10 отсутствовала, поэтому лучше всего придерживаться версии между 3.7 и 3.9, чтобы сверяться по ходу изложения.

С помощью этих двух строк кода вы запустили кластер Ray на своем локальном компьютере. Этот кластер может использовать все доступные на вашем компьютере ядра в качестве работников. Прямо сейчас ваш кластер Ray мало что делает, но это скоро изменится.

Функция `init`, которую вы применяете для запуска кластера, является одним из шести основных API-вызовов, о которых вы подробно узнаете в главе 2. В целом *API инструментария Ray Core* очень доступен и прост в использовании. Но поскольку этот интерфейс также является довольно низкоуровневым, для разработки с его помощью интересных примеров требуется время. В главе 2 приведен подробный первый пример, который поможет вам начать работу с API инструментария Ray Core, а в главе 3 вы увидите, как создавать более интересное приложение Ray с использованием обучения с подкреплением.

В приведенном выше исходном коде мы не предоставили функции `ray.init(...)` никаких аргументов. Если бы вы хотели запустить Ray в «реально существующем» кластере, то вам пришлось бы передать в `init` несколько аргументов. Этот вызов `init` часто называют *клиентом Ray*, и он используется для интерактивного подключения к существующему кластеру Ray<sup>1</sup>. Подробнее об использовании клиента Ray для подключения к производственным кластерам можно прочитать в документации Ray<sup>2</sup>.

Конечно, если вы имели опыт работы с вычислительными кластерами, то знаете, что существует множество подводных камней и тонкостей. Например, можно развертывать приложения Ray в кластерах, предоставляемых облачными провайдерами, такими как Amazon Web Services (AWS), Google Cloud Platform (GCP) или Microsoft Azure, и для каждого варианта требуется хороший инструментарий развертывания и технического сопровождения. Кроме того, кластер можно развернуть на своем собственном оборудовании либо использовать для развертывания своих кластеров Ray такие инструменты, как Kubernetes. В главе 9 (следующие главы посвящены конкретным приложениям Ray) мы вернемся к теме масштабирования рабочих нагрузок с помощью библиотеки Ray Clusters.

Прежде чем перейти к библиотекам Ray более высокого уровня, давайте кратко рассмотрим два основополагающих компонента Ray как фреймворка распределенных вычислений:

### *Ray Clusters*

Этот компонент отвечает за распределение ресурсов, создание узлов и обеспечение их работоспособности. Хорошим стартом в работе с библиотекой Ray Clusters является специальное краткое руководство по использованию<sup>3</sup>.

---

<sup>1</sup> Существуют и другие средства взаимодействия с кластерами Ray, такие как CLI-инструмент подачи заявок на выполнение работы, Ray Jobs CLI (<https://oreil.ly/XXnIw>).

<sup>2</sup> См. <https://oreil.ly/nNhMt>.

<sup>3</sup> См. <https://oreil.ly/rBUIl>.

### Ray Core

После того как ваш кластер запущен, вы используете API инструментария Ray Core, чтобы на нем программировать. Начать работу с Ray Core можно, следуя официальному пошаговому руководству по этому компоненту<sup>1</sup>.

## Комплект библиотек науки о данных

Перейдя в этом разделе ко второму слою фреймворка Ray, мы кратко представим все библиотеки науки о данных, с которыми данный фреймворк поставляется. Для этого сначала давайте взглянем с высоты птичьего полета на то, что, собственно говоря, значит «заниматься наукой о данных». После того как вы поймете этот контекст, вам будет намного проще ознакомиться с библиотеками Ray более высокого уровня и понять, чем они могут быть вам полезны.

## Инструментарий Ray AIR и рабочий процесс науки о данных

Несколько уклончивый термин «наука о данных»<sup>2</sup> за последние годы претерпел значительную эволюцию, и в интернете можно найти множество определений различной полезности<sup>3</sup>. Для нас это практика извлечения информации из данных и разработки реально-практических приложений путем привлечения данных. Это довольно широкое определение по своей сути практической и прикладной дисциплины, которая сосредоточена вокруг создания и понимания вещей. В этом смысле описывать практиков в данной области как «исследователей данных» примерно так же неправильно, как описывать хакеров как «специалистов по вычислительным наукам»<sup>4</sup>.

В общих чертах занятие наукой о данных – это итеративный процесс, который предусматривает разработку технических требований, сбор и обработку данных, разработку моделей и их оценивание, а также развертывание технологических решений. Машинное обучение не обязательно является частью этого процесса, но нередко выступает таковым. Если задействовано машинное обучение, то можно указать несколько дополнительных шагов:

---

<sup>1</sup> См. <https://oreil.ly/7r0Lv>.

<sup>2</sup> Англ. data science. – Прим. перев.

<sup>3</sup> Нам никогда не нравилась классификация науки о данных как пересечение нескольких дисциплин, таких как математика, программирование и бизнес. В конечном счете данное определение не говорит ровным счетом ничего о том, чем именно занимаются практикующие ее специалисты.

<sup>4</sup> В качестве увлекательного упражнения мы рекомендуем прочитать знаменитое эссе Пола Грэма «Хакеры и художники» (Paul Graham, Hackers and Painters, <https://oreil.ly/ZEDtU>) на эту тему и заменить «вычислительные науки» на «наука о данных». Каким был бы взлом 2.0?

## Обработка данных

Для тренировки моделей машинного обучения нужны данные в формате, понятном вашей модели. Процесс преобразования и выбора того, какие данные следует подавать в вашу модель, часто называют *конструированием признаков*. Этот шаг бывает запутанным. Вы получаете больше пользы, если для выполнения этой работы сможете опираться на общепринятые инструменты.

## Тренировка моделей

В машинном обучении нужно тренировать свои алгоритмы на данных, которые были обработаны на предыдущем шаге. Сюда входит выбор правильного алгоритма для работы, и вам будет играть на руку возможность выбирать алгоритм из широкого ассортимента.

## Гиперпараметрическая настройка

Модели машинного обучения имеют параметры, которые настраиваются на шаге тренировки модели. Большинство моделей машинного обучения также имеют еще один набор параметров, именуемых *гиперпараметрами*, которые можно видоизменять перед тренировкой. Эти параметры могут сильно влиять на результативность результирующей модели машинного обучения и нуждаются в надлежащей настройке. Существуют хорошие инструменты, помогающие автоматизировать этот процесс.

## Подача моделей в качестве служб

Натренированные модели необходимо развертывать. Подавать модель в качестве службы<sup>1</sup> – значит делать ее доступной для всех, кто нуждается в доступе, любыми необходимыми средствами. В прототипах часто используются простые HTTP-серверы, но подаче моделей машинного обучения в качестве служб посвящено множество специализированных программных пакетов.

Этот список ни в коем случае не является исчерпывающим, и о разработке приложений машинного обучения можно рассказать гораздо больше<sup>2</sup>. Однако перечисленные выше четыре шага действительно имеют решающее значение для успеха проекта в области науки о данных с использованием машинного обучения.

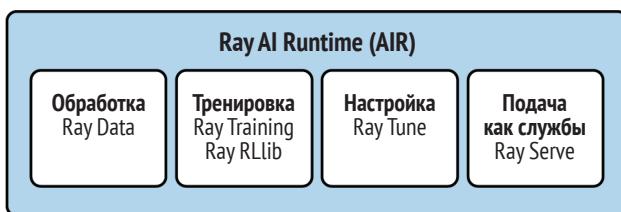
Во фреймворке Ray есть специальные библиотеки для каждого из только что перечисленных четырех шагов, специфичных для машинного обучения. В частности, вы можете обеспечивать свои потребности в обработке данных с помощью библиотеки *Ray Data*, выполнять распределенную тренировку моделей посредством библиотеки *Ray Train* и рабочие нагрузки обучения с подкреплением при содействии библиотеки *Ray RLlib*, эффективно настраивать гиперпараметры с помощью библиотеки *Ray Tune* и организовывать подачу

<sup>1</sup> Англ. *model serving*. – Прим. перев.

<sup>2</sup> Если вы хотите подробнее узнать о целостном взгляде на процесс науки о данных при разработке приложений машинного обучения, то книга Эммануэля Амейзена «Разработка приложений на базе машинного обучения» (Emmanuel Améisen, Building Machine Learning Powered Applications, O'Reilly) полностью посвящена данной теме.

своих моделей в качестве служб через библиотеку *Ray Serve*. При этом то, как построен фреймворк Ray, означает, что все эти библиотеки *распределены по своему внутреннему устройству*, и этот момент невозможно переоценить.

Более того, все эти шаги являются частью процесса и редко выполняются изолированно. У вас не только все задействованные библиотеки могут взаимодействовать бесшовно, но и есть возможность получать решающее преимущество, работая со состыкованным API на протяжении всего процесса науки о данных. Это именно то, для чего был разработан инструментарий Ray AIR: наличие общей среды выполнения и API для ваших экспериментов, а также возможность масштабировать свои рабочие нагрузки, когда вы будете готовы. На рис. 1.2 показан общий обзор всех компонентов AIR.



**Рис. 1.2** ♦ Инструментарий Ray AIR является зонтичным компонентом для всех встроенных во фреймворк Ray текущих библиотек науки о данных

Хотя знакомить с инструментарием Ray AI Runtime API в этой главе было бы уже перебором (для этого вы можете перейти к главе 10), мы познакомим вас со всеми строительными блоками, которые подаются на его вход. Давайте пройдемся поочередно по каждой встроенной во фреймворк Ray библиотеке науки о данных.

## Обработка данных с использованием библиотеки Ray Data

Первая высокоуровневая библиотека Ray, о которой мы поговорим, – это библиотека Ray Data. Указанная библиотека содержит структуру данных, как и положено, именуемую *Dataset*, множество коннекторов для загрузки данных из различных форматов и систем, API преобразования таких наборов данных, способы формирования конвейеров обработки данных с их помощью и множество интеграций с другими фреймворками обработки данных. Абстракция *Dataset* строится поверх мощного фреймворка Arrow<sup>1</sup>.

Прежде чем начать работу с библиотекой Ray Data, необходимо установить Python'овскую библиотеку Arrow, например выполнив команду `pip install`

<sup>1</sup> В главе 6 мы познакомим вас с основами того, что обеспечивает работу распределенных наборов данных Ray Dataset, включая использование в них фреймворка Arrow. На данный момент мы хотим сосредоточиться на их API и конкретных шаблонах использования.

руаггов. Следующий ниже простой пример создает распределенный набор данных `Dataset` в вашем локальном кластере Ray из структуры данных Python. В частности, вы создадите набор данных из словаря Python, содержащий 10 000 записей со строковым именем (`name`) и целочисленным значением (`data`):

```
import ray

items = [{"name": str(i), "data": i} for i in range(10000)]
ds = ray.data.from_items(items) ❶
ds.show(5) ❷
```

- ❶ Создать распределенный набор данных `Dataset` с помощью метода `from_items` модуля `ray.data`.
- ❷ Напечатать первые пять элементов набора данных `Dataset`.

Показать (`show`) набор данных `Dataset` – значит напечатать несколько его значений. Вы должны увидеть ровно пять элементов в командной строке, вот в таком виде:

```
{'name': '0', 'data': 0}
{'name': '1', 'data': 1}
{'name': '2', 'data': 2}
{'name': '3', 'data': 3}
{'name': '4', 'data': 4}
```

Отлично! Теперь у вас есть несколько строк, но что можно сделать с этими данными? В API объекта `Dataset` серьезная ставка делается на функциональное программирование, поскольку эта парадигма хорошо подходит для преобразований данных.

Несмотря на то что Python 3 постарался скрыть некоторые свои возможности функционального программирования, вы, вероятно, знакомы с такими функциями, как `map`, `filter`, `flat_map` и др. Если нет, то это достаточно просто сделать: `map` берет каждый элемент набора данных и преобразовывает его во что-то другое в параллельном режиме; `filter` удаляет точки данных в соответствии с булевой функцией фильтрации, а чуть более сложная функция `flat_map` сначала преобразовывает значения аналогично `map`, но затем она также «разглаживает» результат. Например, если бы функция `map` создала список списков, то `flat_map` разгладила бы вложенные списки и выдала бы простой список. Оснащенные этими тремя функциональными API-вызовами<sup>1</sup>, давайте посмотрим, насколько легко можно преобразовывать свой набор данных `ds`:

```
squares = ds.map(lambda x: x[“data”] ** 2) ❶
evens = squares.filter(lambda x: x % 2 == 0) ❷
evens.count()
```

---

<sup>1</sup> Мы остановимся на этом подробнее в последующих главах, в частности в главе 6, но обратите внимание, что Ray Data не является библиотекой обработки данных общего назначения. Такие инструменты, как Spark, обладают более развитой и оптимизированной поддержкой крупномасштабной обработки данных.

```
cubes = evens.flat_map(lambda x: [x, x**3]) ③
sample = cubes.take(10) ④
print(sample)
```

- ① Мы преобразовываем (`map`) каждую строку в `ds`, оставляя только квадратное значение ее записи `data`.
- ② Затем мы фильтруем (`filter`) квадраты (`squares`), оставляя только четные числа (в общей сложности пять тысяч элементов).
- ③ Потом мы применяем `flat_map`, чтобы дополнить оставшиеся значения соответствующими кубами.
- ④ Взять (`take`) в общей сложности 10 значений означает покинуть фреймворк Ray и вернуть список Python с этими значениями, которые затем можно напечатать.

Недостатком преобразований объектов `Dataset` является то, что каждый шаг выполняется синхронно. В данном примере это не проблема, но в сложных заданиях, в которых, например, сочетаются чтение файлов и обработка данных, вам, возможно, потребуется исполнение, которое может накладываться на индивидуальные задания. Объект `DatasetPipeline` делает именно это. Давайте перепишем приведенный выше пример в форме конвейера:

```
pipe = ds.window() ①
result = pipe\
    .map(lambda x: x[“data”] ** 2)\ \
    .filter(lambda x: x % 2 == 0)\ \
    .flat_map(lambda x: [x, x**3]) ②
result.show(10)
```

- ① Объект `Dataset` можно превратить в конвейер, вызвав на нем функцию `.window()`.
- ② Шаги конвейера могут быть выстроены в цепочку, и можно получить тот же результат, что и раньше.

О библиотеке Ray Data можно рассказать гораздо больше, в особенности о ее интеграции с известными системами обработки данных, но мы отложим подробное обсуждение до главы 6.

## Тренировка моделей

Перейдя к следующему набору библиотек, давайте рассмотрим возможности фреймворка Ray по распределенной тренировке. Для этого у вас есть доступ к двум библиотекам. Одна из них посвящена конкретно обучению с подкреплением; другая имеет иную сферу применения и ориентирована в первую очередь на задания, связанные с контролируемым обучением.

### *Обучение с подкреплением с помощью библиотеки Ray RLlib*

Давайте начнем с библиотеки Ray RLlib, служащей для обучения с подкреплением<sup>1</sup>. Эта библиотека основана на современных фреймворках машинного

---

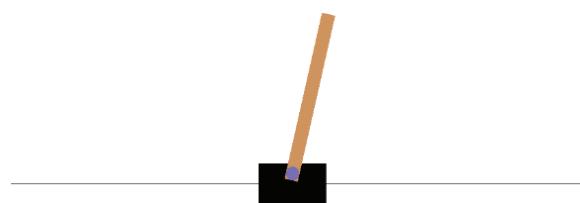
<sup>1</sup> Англ. reinforcement learning (RL); син. подкрепляемое обучение. – Прим. перев.

обучения TensorFlow и PyTorch, и вы можете использовать любой из них. Оба фреймворка, похоже, все больше концептуально сходятся, так что можно выбрать тот, который нравится больше всего, при этом не много теряя от этого. На протяжении всей книги мы используем примеры как на основе TensorFlow, так и на основе PyTorch, чтобы при использовании фреймворка Ray вы могли получить представление об обоих упомянутых фреймворках.

В рамках этого раздела прямо сейчас следует установить TensorFlow с помощью команды `pip install tensorflow1`. Прежде чем выполнить пример исходного кода, также необходимо установить библиотеку gym. Это делается с помощью команды `pip install "gym==0.25.0"`.

Одним из самых простых способов выполнения примеров с применением библиотеки RLlib является применение инструмента командной строки `rllib`, который мы уже установили неявно при выполнении команды `pip install "ray[rllib]"`. Когда в главе 4 вы будете выполнять более сложные примеры, вы будете в основном опираться на ее Python'овский API, но сейчас мы хотим получить первое представление о проведении экспериментов по обучению с подкреплением с помощью RLlib.

Мы рассмотрим довольно классическую задачу управления балансировкой шеста на тележке. Представьте, что у вас есть шест, подобный изображенному на рис. 1.3, закрепленный на стыке тележки и подверженный действию силы тяжести. Тележка может свободно перемещаться по дорожке без трения, и вы можете управлять тележкой, подталкивая ее слева либо справа с фиксированной силой. Если вы будете делать это достаточно хорошо, то шест будет оставаться в вертикальном положении. За каждый временной шаг, в течение которого шест не упал, мы получаем вознаграждение в размере 1 балла. Нашей целью является получение высокого вознаграждения, и вопрос заключается в том, сможем ли мы научить алгоритм самообучения на основе подкрепления делать это за нас.



**Рис. 1.3 ♦ Управление шестом, прикрепленным к тележке, путем приложения усилия влево либо вправо**

В частности, мы хотим натренировать агента обучения с подкреплением, который может выполнять два действия, а именно: толкать тележку влево либо вправо, наблюдать за происходящим при взаимодействии со средой

<sup>1</sup> Если вы используете Mac, то вам придется установить `tensorflow-macos`. В общем случае, если при установке фреймворка Ray или его зависимостей в вашей системе вы столкнетесь с какими-либо проблемами, следует обратиться к руководству по установке (<https://docs.ray.io/en/latest/ray-overview/installation.html>).

указанным образом и учиться на основе опыта, максимизируя получаемое вознаграждение (то есть подкрепление со стороны среды).

В целях решения этой задачи с помощью библиотеки Ray RLlib мы можем использовать так называемый *отрегулированный* пример, то есть предварительно сконфигурированный алгоритм, который хорошо работает в данной задаче. Отрегулированный пример можно выполнить с помощью одной команды. Библиотека RLlib содержит множество таких примеров, и их все можно перечислить посредством команды `rllib example list`.

Одним из таких примеров является отрегулированный пример `cartpole-ppo`, в котором для решения задачи о шесте на тележке используется алгоритм PPO, в частности среда `CartPole-v1`<sup>1</sup> библиотеки OpenAI Gym. С конфигурацией этого примера можно ознакомиться, набрав команду `rllib example get cartpole-ppo`, которая сначала скачает файл примера с GitHub, а затем распечатает его конфигурацию. Указанная конфигурация закодирована в формате YAML-файла и выглядит следующим образом:

```
cartpole-ppo:  
  env: CartPole-v1 ❶  
  run: PPO ❷  
  stop:  
    episode_reward_mean: 150 ❸  
    timesteps_total: 100000  
  config: ❹  
    framework: tf  
    gamma: 0.99  
    lr: 0.0003  
    num_workers: 1  
    observation_filter: MeanStdFilter  
    num_sgd_iter: 6  
    vf_loss_coeff: 0.01  
    model:  
      fcnet_hiddens: [32]  
      fcnet_activation: linear  
      vf_share_layers: true  
    enable_connectors: True
```

- ❶ Среда `CartPole-v1` симулирует задачу, которую мы только что описали.
- ❷ Использовать мощный алгоритм обучения с подкреплением под названием Proximal Policy Optimization<sup>2</sup>, или PPO.
- ❸ Прекратить эксперимент, как только мы получим вознаграждение в размере 150 баллов.
- ❹ Алгоритм PPO нуждается в небольшом специфичном для обучения с подкреплением конфигурировании, которое настроит его под работу с этой задачей.

Детали приведенного выше конфигурационного файла на данный момент не имеют большого значения, поэтому не следует на них отвлекаться. Важной частью является то, что вы указываете среду `Cartpole-v1` и специфичную для обучения с подкреплением конфигурацию, достаточную для обеспечения

---

<sup>1</sup> См. <https://oreil.ly/YNxoz>.

<sup>2</sup> Оптимизация политики близости расположения. – Прим. перев.

работы процедуры тренировки. Запуск этой конфигурации не требует какого-либо специального оборудования и завершается в считанные минуты. В целях тренировки этого примера нужно будет установить зависимость PyGame с помощью команды `pip install pygame`, а затем просто выполнить следующую ниже команду:

```
rllib example run cartpole-ppo
```

Если ее выполнить, то библиотека RLLib создаст именованный эксперимент и зарегистрирует в журнале важные метрики, такие как вознаграждение или среднее вознаграждение за эпизод (`episode_reward_mean`). На выходе из тренировочного прогона вы также должны увидеть информацию о машине (`loc`, означающую хост-имя и порт), а также статус тренировочных прогонов. Если прогон терминирован (`TERMINATED`), но в журнале ни разу не был показан успешно работающий (`RUNNING`) эксперимент, то, должно быть, что-то пошло не так. Вот примерный фрагмент тренировочного прогона:

Trial name	status	loc
PPO_CartPole-v0_9931e_00000	RUNNING	127.0.0.1:8683

Когда тренировочный прогон завершится и все пройдет хорошо, то вы должны увидеть следующий ниже результат:

```
Your training finished.  
Best available checkpoint for each trial:  
<checkpoint-path>/checkpoint_<number>
```

Теперь можно оценить свой натренированный алгоритм из любой контрольной точки, например выполнив:

```
rllib evaluate <checkpoint-path>/checkpoint_<number> --algo PPO
```

По умолчанию локальной папкой контрольных точек Ray является `~/ray-results`. В использованной нами конфигурации тренировки путь `<checkpoint-path>` должен иметь вид `~/ray_results/cartpole-ppo/PPO_CartPole-v1_<experiment_id>`. Во время процедуры тренировки в эту папку будут сгенерированы промежуточные и окончательные контрольные точки модели.

Теперь оценить результативность натренированного алгоритма обучения с подкреплением можно из контрольной точки, скопировав команду, напечатанную в предыдущем примере тренировочного прогона:

```
rllib evaluate <checkpoint-path>/checkpoint_<number> --algo PPO
```

Выполнение этой команды приведет к печати результатов оценивания, а именно вознаграждений, полученных с помощью обученного алгоритма в среде CartPole-v1.

С помощью библиотеки RLLib можно делать гораздо больше, и мы рассмотрим это подробнее в главе 4. Цель приведенного выше примера состояла в том, чтобы продемонстрировать то, насколько легко можно начать работу с RLLib и инструментом командной строки `rllib`, используя команды `example` и `evaluate`.

## ***Распределенная тренировка с помощью библиотеки Ray Train***

Библиотека Ray RLLib посвящена исключительно обучению с подкреплением, но что делать, если нужно тренировать модели из других типов машинного обучения, таких как контролируемое обучение? В этом случае можно воздействовать еще одну библиотеку фреймворка Ray для распределенной тренировки: *Ray Train*. На данный момент у нас недостаточно знаний о таких фреймворках, как TensorFlow, чтобы привести вам краткий и информативный пример для Ray Train. Если вас интересует распределенная тренировка, то вы можете перейти к главе 6.

## **Гиперпараметрическая настройка**

Давать названия вещам сложно, но название библиотеки *Ray Tune*, которая используется для настройки всевозможных параметров, попадает в точку. Она была разработана специально для отыскания хороших гиперпараметров для моделей машинного обучения. Типичная конфигурация выглядит следующим образом:

- вы хотите выполнить чрезвычайно дорогостоящую в вычислительном плане функцию тренировки. В машинном обучении нередко выполняются процедуры тренировки, которые занимают дни, если не недели, но давайте предположим, что вы имеете дело всего с парой минут;
- в результате тренировки вы вычисляете так называемую целевую функцию. Обычно вы хотите либо максимизировать выгоду, либо минимизировать потерю с точки зрения результативности эксперимента;
- сложность заключается в том, что функция тренировки может зависеть от определенных параметров, именуемых гиперпараметрами, которые влияют на значение целевой функции;
- у вас могут быть догадки в отношении того, какими должны быть отдельные гиперпараметры, но настроить их все бывает трудно. Даже если вы можете ограничить эти параметры разумным диапазоном, обычно тестирование широкого спектра комбинаций имеет запретительный характер. Ваша функция тренировки обходится просто слишком дорого.

Что можно сделать, чтобы эффективно брать образцы гиперпараметров и получать «достаточно хорошие» результаты на своей целевой функции? Область, связанная с решением этой задачи, называется *гиперпараметри-*

ческой оптимизацией<sup>1</sup>, и библиотека Ray Tune располагает огромным набором алгоритмов для ее решения. Давайте рассмотрим пример применения указанной библиотеки, используемый в ситуации, которую мы только что объяснили. Основное внимание снова сосредоточено на фреймворке Ray и его API, а не на конкретной задаче машинного обучения (которую мы пока просто просимулируем):

```
from ray import tune
import math
import time

def training_function(config): ❶
    x, y = config["x"], config["y"]
    time.sleep(10)
    score = objective(x, y)
    tune.report(score=score) ❷

def objective(x, y):
    return math.sqrt((x**2 + y**2)/2) ❸

result = tune.run(❹
    training_function,
    config={
        "x": tune.grid_search([-1, -.5, 0, .5, 1]), ❺
        "y": tune.grid_search([-1, -.5, 0, .5, 1])
    })

print(result.get_best_config(metric="score", mode="min"))
```

- ❶ Просимулировать дорогостоящую функцию тренировки, которая зависит от двух гиперпараметров, *x* и *y*, прочитанных из конфигурации (*config*).
- ❷ После 10-секундного сна, симулирующего тренировку и вычисление целевой функции, объекту *tune* сообщается балл.
- ❸ Целевая функция вычисляет среднее значение квадратов *x* и *y* и возвращает квадратный корень из этого члена. В машинном обучении указанный тип целевой функции довольно распространен.
- ❹ Применить функцию *tune.run*, чтобы инициализировать гиперпараметрическую оптимизацию на функции тренировки (*training\_function*).
- ❺ Ключевой частью является предоставление параметрического пространства для *x* и *y*, в котором *tune* будет выполнять поиск.

Обратите внимание, что результат этого прогона структурно похож на то, что вы видели в примере с библиотекой RLib. Это не совпадение, поскольку в RLib (как и во многих других библиотеках ФРЕЙМВОРКА Ray) под капотом используется библиотека Ray Tune. Если вы присмотритесь повнимательнее, то увидите ожидающие исполнения (PENDING) прогоны, а также работающие (RUNNING) и терминированные (TERMINATED) прогоны. Библиотека Tune выбирает, планирует и исполняет ваши тренировочные прогоны автоматически.

---

<sup>1</sup> Англ. meter optimization (HPO).

В частности, этот пример с библиотекой Tune отыскивает наилучшие возможные комбинации параметров  $x$  и  $y$  для функции тренировки (`training_function`) с заданной целевой функцией (`objective`), которую мы хотим минимизировать. Поначалу целевая функция, возможно, покажется на вид немного устрашающей, но поскольку мы вычисляем сумму квадратов  $x$  и  $y$ , все значения будут неотрицательными. Это означает, что наименьшее значение получается при  $x=0$  и  $y=0$ , и в результате вычисления целевая функция примет значение 0.

Мы выполняем так называемый *поиск в параметрической решетке*<sup>1</sup> по всем возможным комбинациям параметров. Поскольку мы в явной форме передаем 5 возможных значений как для  $x$ , так и для  $y$ , в функцию тренировки подается в общей сложности 25 комбинаций. Поскольку мы переводим функцию тренировки (`training_function`) в спящий режим на 10 секунд, последовательное тестирование всех комбинаций гиперпараметров в общей сложности займет более 4 минут. Поскольку фреймворк Ray умело паралллизует эту рабочую нагрузку, весь этот эксперимент занял у нас всего около 35 секунд, но может занять гораздо больше времени, в зависимости от того, где вы его проводите.

Теперь представьте, что каждый тренировочный прогон занял бы несколько часов, и у нас было бы не 2 гиперпараметра, а 20. Это делает поиск в параметрической решетке неосуществимым, в особенности если у вас нет обоснованных догадок о диапазоне параметров. В таких ситуациях, как обсуждается в главе 5, придется использовать более сложные методы гиперпараметрической оптимизации библиотеки Ray Tune.

## Подача моделей в качестве служб

Последняя высокоуровневая библиотека фреймворка Ray, которую мы здесь обсудим, специализируется на подаче моделей в качестве службы и называется просто *Ray Serve*. Для того чтобы увидеть пример ее работы, нужна натренированная модель машинного обучения, которая подлежит размещению в качестве службы. К счастью, в настоящее время в интернете можно найти много интересных моделей, которые уже были натренированы за вас. Например, на веб-сайте Hugging Face сообщества ИИ есть целый ряд моделей, которые можно скачать непосредственно в Python. Мы будем использовать языковую модель под названием GPT-2, которая на входе принимает текст и на выходе продуцирует текст продолжения или завершения введенного текста. Например, можно задать вопрос, и GPT-2 попытается на него ответить.

Размещение такой модели в качестве службы – хороший способ сделать ее общедоступной. Возможно, вы не знаете, как загружать и запускать модель TensorFlow на своем компьютере, но вы знаете, как задавать вопрос на простом английском языке. Подача модели в качестве службы скрывает детали реализации вычислительного решения и позволяет пользователям сосредоточиваться на передаче данных на вход модели и интерпретации данных на выходе из нее.

---

<sup>1</sup> См. grid search. – Прим. перев.

Продолжая работу, надо выполнить команду `pip install transformers`, чтобы установить библиотеку Hugging Face, в которой есть модель, которую мы хотим использовать<sup>1</sup>. Теперь можно импортировать и запустить экземпляр библиотеки `serve` фреймворка Ray, загрузить и развернуть модель GPT-2 и задать ей вопрос о смысле жизни, примерно вот так:

```
from ray import serve
from transformers import pipeline
import requests

serve.start() ❶

@serve.deployment ❷
def model(request):
    language_model = pipeline("text-generation", model="gpt2") ❸
    query = request.query_params["query"]
    return language_model(query, max_length=100) ❹

model.deploy() ❺

query = "What's the meaning of life?"
response = requests.get(f"http://localhost:8000/model?query={query}") ❻
print(response.text)
```

- ❶ Запустить `serve` локально.
- ❷ Декоратор `@serve.deployment` превращает функцию с параметром `request` в развертывание `serve`.
- ❸ Загружать языковую модель (`language_model`) внутрь функции `model` для каждого запроса – это не совсем эффективное решение, но это самый быстрый способ продемонстрировать развертывание.
- ❹ Попросить модель выдавать не более 100 символов с продолжением запроса.
- ❺ Формально развернуть модель таким образом, чтобы она могла начать получать запросы по HTTP.
- ❻ Использовать обязательную библиотеку `requests`, чтобы получать ответ на любой вопрос, который у вас может возникнуть.

В главе 9 вы научитесь надлежащим образом развертывать модели в различных сценариях, но сейчас мы рекомендуем вам поиграть с этим примером и протестировать различные запросы. Повторное выполнение последних двух строк исходного кода практически каждый раз будет давать разные ответы. Ниже приведена мрачная поэтическая жемчужина, вызывающая еще больше вопросов и слегка подвергнутая цензуре с учетом несовершеннолетних читателей, которая возникла из одного запроса<sup>2</sup>:

---

<sup>1</sup> В зависимости от используемой операционной системы, возможно, сначала потребуется установить компилятор Rust, чтобы обеспечить ее успешную работу. Например, в Mac указанный компилятор можно установить с помощью команды `brew install rust`.

<sup>2</sup> Перевод: В чем смысл жизни? Существует ли тот или иной образ жизни? Каково это – быть пойманным в ловушку взаимоотношений? Как ее изменить, пока не стало слишком поздно? Как мы ее называли в наше время? Какое место мы занимаем в этом мире и ради чего мы, собственно, собираемся жить? Моя жизнь как личности была сформирована любовью, которую я получал от других. – Прим. перев.

```
[{  
    "generated_text": "What's the meaning of life?\n\n" Is there one way or another of living?\n\n" How does it feel to be trapped in a relationship?\n\n" How can it be changed before it's too late?  
    What did we call it in our time?\n\n" Where do we fit within this world and what are we going to live for?\n\n" My life as a person has been shaped by the love I've received from others."  
}]
```

На этом мы завершаем наш головокружительный тур по второму слою фреймворка Ray – его библиотекам для науки о данных. В конечном счете все представленные в этой главе высокоуровневые библиотеки фреймворка Ray являются расширениями API инструментария Ray Core. Фреймворк Ray позволяет относительно легко разрабатывать новые расширения, и есть еще несколько библиотек, которые мы не можем полностью обсудить в этой книге. Например, относительно недавно появилось дополнение – библиотека Ray Workflows<sup>1</sup>, которая позволяет определять и выполнять с помощью Ray длительно работающие приложения.

Прежде чем завершить эту главу, давайте очень кратко рассмотрим третий слой – растущую экосистему вокруг фреймворка Ray.

## Растущая экосистема

Высокоуровневые библиотеки фреймворка Ray мощны и заслуживают гораздо более глубокого рассмотрения на протяжении всей книги. Хотя их полезность для жизненного цикла экспериментов в области науки о данных неоспорима, мы также не хотим создавать впечатление, что отныне фреймворк Ray – это все, что вам нужно. Неудивительно, что лучшие и наиболее успешные фреймворки – это те, которые хорошо интегрируются с существующими техническими решениями и идеями. Лучше сосредоточиться на своих главных сильных сторонах и использовать другие инструменты для восполнения того, чего не хватает в вашем технологическом решении, и фреймворк Ray делает это довольно хорошо.

На протяжении всей книги, и в частности в главе 11, мы будем обсуждать множество полезных сторонних библиотек, построенных поверх фреймворка Ray. Экосистема фреймворка Ray также имеет целый ряд интеграций с существующими инструментами. В качестве примера напомним, что Ray Data – это библиотека загрузки и вычислений данных фреймворка Ray. Если у вас есть существующий проект, в котором уже используются движки обработки данных, такие как Spark или Dask<sup>2</sup>, вы можете использовать эти инструмен-

---

<sup>1</sup> См. <https://oreil.ly/XUT7y>.

<sup>2</sup> Spark был создан другой лабораторией в Беркли, AMPLab. Интернет полон блог-постов, в которых утверждается, что Ray следует рассматривать как замену Spark. Их лучше трактовать как инструменты с разными сильными сторонами, которые, скорее всего, останутся надолго.

ты вместе с Ray. В частности, можно запускать всю экосистему Dask поверх кластера Ray, применяя планировщик Dask on Ray, либо использовать проект Spark on Ray<sup>1</sup>, чтобы интегрировать рабочие нагрузки Spark с Ray. Аналогичным образом проект Modin<sup>2</sup> является упрощенной распределенной заменой наборов данных DataFrame<sup>3</sup> библиотеки Pandas, в которой Ray (или Dask) используется в качестве движка распределенного исполнения (Pandas on Ray).

Общей темой здесь является то, что фреймворк Ray не пытается заменить все эти инструменты, а наоборот – интегрируется с ними, по-прежнему предоставляя вам доступ к своей нативной библиотеке Ray Data. В главе 11 мы подробно рассмотрим взаимосвязь Ray с другими инструментами в более широкой экосистеме.

Одним из важных аспектов многих библиотек фреймворка Ray является то, что они легко интегрируют общераспространенные инструменты в качестве бэкендов. Ray часто создает общие интерфейсы, вместо того чтобы пытаться создавать новые стандарты<sup>4</sup>. Эти интерфейсы позволяют выполнять задания распределенным образом, чего нет у большинства соответствующих бэкендовых приложений либо не в такой степени. Например, библиотеки RLLib и Train фреймворка Ray опираются на всю мощь фреймворков TensorFlow и PyTorch. А библиотека Ray Tune поддерживает алгоритмы практических из всех известных инструментов гиперпараметрической оптимизации, имеющихся на сегодняшний день, включая Hyperopt, Optuna, Nevergrad, Ax, SigOpt и многие другие. Ни один из этих инструментов не распределим по умолчанию, но библиотека Tune объединяет их в *общий интерфейс для распределенных рабочих нагрузок*.

## РЕЗЮМЕ

На рис. 1.4 представлен общий обзор трех слоев фреймворка Ray в том виде, в каком мы их разместили. Стержневой движок распределенного исполнения находится в центре фреймворка Ray. API инструментария Ray Core – это универсальная библиотека для распределенных вычислений, а библиотека Ray Clusters позволяет развертывать рабочие нагрузки различными способами.

В ситуациях практических рабочих процессов науки о данных библиотека Ray Data используется для обработки данных, библиотека Ray RLLib для обучения с подкреплением, библиотека Ray Train для распределенной трени-

<sup>1</sup> См. <https://oreil.ly/J1D5I>.

<sup>2</sup> См. <https://oreil.ly/brGPJ>.

<sup>3</sup> Кадр данных. – Прим. перев.

<sup>4</sup> До того, как фреймворк глубокого обучения Keras (<https://keras.io/>) стал официальной частью TensorFlow, он начинался как удобная API-спецификация для различных фреймворков более низкого уровня, таких как Theano или CNTK. В этом смысле у библиотеки Ray RLLib есть шанс стать «Keras для обучения с подкреплением», а библиотека Ray Tune может быть просто «Keras для гиперпараметрической оптимизации». Недостающим звеном для более широкого принятия на вооружение может стать чуть более элегантный API у обоих.

ровки моделей, библиотека Ray Tune для гиперпараметрической настройки и библиотека Ray Serve для подачи моделей как служб. Вы видели примеры работы с каждой из этих библиотек и имеете представление о том, как выглядят их API. Инструментарий Ray AIR предоставляет унифицированный API для всех других библиотек машинного обучения в рамках фреймворка Ray и был разработан с учетом потребностей исследователей данных.

Вдобавок ко всему экосистема фреймворка Ray имеет множество расширений, интеграций и бэкендов, которые мы подробнее рассмотрим позже. Возможно, на рис. 1.4, вы уже отметите несколько инструментов, которые вам известны и нравятся.

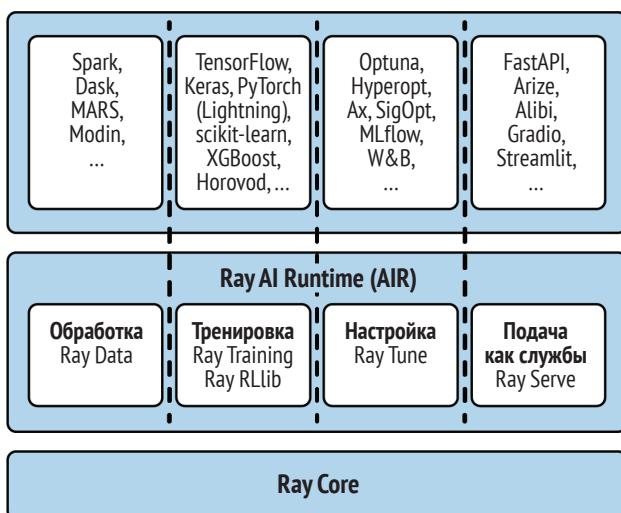


Рис. 1.4 ♦ Фреймворк Ray в трех слоях

Инструментарий API инструментария Ray Core расположен в центре рис. 1.4, в окружении библиотек RLLib, Ray Tune, Ray Train, Ray Serve, Ray Data и множества сторонних интеграций, которых слишком много, чтобы перечислять их здесь.

# Глава 2

---

## Начало работы с инструментарием Ray Core

Не лишено иронии то, что вы читаете книгу о распределенных вычислениях на Python, хотя сам по себе Python большей частью характерен своей неэффективностью в распределенных вычислениях. Его интерпретатор является практически однопоточным. Например, использование простого Python'a затрудняет применение нескольких центральных процессоров на одной машине, не говоря уже о целом кластере машин. Следовательно, нужны дополнительные инструменты, и, к счастью, в экосистеме языка Python для вас есть несколько вариантов. Такие библиотеки, как `multiprocessing`, помогают распределять работу на одной машине<sup>1</sup>, но не более того.

API инструментария Ray Core, рассматриваемый как библиотека Python, обладает достаточной мощью для того, чтобы сделать общее распределенное программирование более доступным для сообщества языка Python в целом. В порядке аналогии, некоторые компании обходятся развертыванием предварительно натренированных моделей машинного обучения для своих вариантов использования, но такая стратегия не всегда эффективна. Нередко для достижения успеха неизбежно приходится тренировать конкретно-прикладные модели. Ваши распределенные рабочие нагрузки точно так же *могли бы* просто вписываться в (потенциально ограничивающую) модель программирования существующих фреймворков, но инструментарий Ray Core благодаря своей универсальности может раскрывать весь спектр возможностей разработки распределенных приложений<sup>2</sup>. Поскольку это имеет настолько

---

<sup>1</sup> Обратите внимание, что фреймворт Ray поставляется с упрощенной заменой многопроцессорной обработки (<https://oreil.ly/UMg73>), которая может оказаться полезным для определенных рабочих нагрузок.

<sup>2</sup> Это является компромиссом между универсальностью и специализацией. Предоставляя поверх API инструментария Core специализированные, но в то же время пригодные для взаимодействия библиотеки, фреймворт Ray обеспечивает инструментарий на различных уровнях абстракции.

фундаментальное значение, мы посвящаем всю эту главу основам данного инструментария, а всю главу 3 – разработке интересного приложения с использованием его API. Благодаря такому подходу вы получите практические знания об инструментарии Ray Core и сможете использовать их в последующих главах и своих собственных проектах.

В этой главе вы поймете, как инструментарий Ray Core оперирует распределенными вычислениями, запустив локальный кластер, и научитесь использовать экономичный и мощный API фреймворка Ray для параллелизации интересующих вычислений. Например, вы разработаете пример с использованием Ray, удобным способом эффективно и асинхронно выполняющий задание с параллелизмом данных, который нелегко воспроизведется с помощью других инструментов. Мы поговорим о том, каким образом задания и акторы работают как распределенные версии функций и классов Python. Вы также научитесь помещать *объекты* в хранилище объектов Ray и извлекать их оттуда. Мы приведем вам конкретные примеры этих трех фундаментальных концепций (заданий, акторов и объектов), используя всего шесть базовых API-вызовов инструментария Ray Core. Наконец, мы поговорим о системных компонентах, лежащих в основе фреймворка Ray, и о том, как выглядит его архитектура. Другими словами, в этой главе мы дадим вам возможность заглянуть под капот двигателя Ray.

## ВВЕДЕНИЕ В ИНСТРУМЕНТАРИЙ RAY CORE

Подавляющая часть этой главы посвящена расширенному примеру, который мы разработаем вместе, с использованием инструментария Ray Core. Многие концепции фреймворка Ray можно объяснить на хорошем примере, так что именно этим мы и займемся.

Как и прежде, вы можете отслеживать работу этого примера, набирая исходный код самостоятельно (что настоятельно рекомендуется) либо следуя инструкциям в блокноте Jupiter этой главы<sup>1</sup>. В любом случае, проверьте, чтобы фреймворк Ray у вас был установлен, например с помощью команды `pip install ray`.

В главе 1 мы показали, как запускать локальный кластер, вызывая `import ray` и затем инициализируя его с помощью `ray.init()`. После выполнения этого исходного кода вы увидите результат в следующей ниже форме. Мы опустили излишнюю информацию, поскольку для ее понимания вам сначала необходимо больше разобраться во внутренних механизмах фреймворка Ray:

```
... INFO services.py:1263 -- View the Ray dashboard at http://127.0.0.1:8265
{'node_ip_address': '192.168.1.41',
...
'node_id': '...'}

---


```

<sup>1</sup> См. <https://oreil.ly/lMmfp>.

Этот результат указывает на то, что ваш кластер Ray запущен. Как видно из первой строки результата, Ray поставляется со своей собственной, предупакованной приборной панелью<sup>1</sup>. В этом можно удостовериться по адресу <http://127.0.0.1:8265>, если только ваш результат не показывает другой порт. Если вы хотите ознакомиться с приборной панелью, то лучше не торопиться и все хорошо рассмотреть. Например, вы должны увидеть список всех ядер центрального процессора и общую задействованность вашего (тривиального) приложения Ray. Для того чтобы увидеть задействованность ресурсов кластера Ray на Python, нужно вызвать `ray.cluster_resources()`. Результат должен выглядеть примерно так:

```
{'CPU': 12.0,
'memory': 14203886388.0,
'node:127.0.0.1': 1.0,
'object_store_memory': 2147483648.0}
```

Для выполнения примеров этой главы вам понадобится работающий кластер Ray, поэтому, прежде чем продолжить, проверьте, чтобы он был запущен. Цель этого раздела – дать краткое введение в API инструментария Ray Core, который с этого момента мы будем просто называть Ray API.

Для программистов на Python самое замечательное в Ray API заключается в том, что он работает «очень близко к дому». В нем используются знакомые концепции, такие как декораторы, функции и классы, тем самым обеспечивая быстрый процесс усвоения. Ray API призван обеспечивать универсальный программный интерфейс для распределенных вычислений. Это, разумеется, непростая задача, но мы думаем, что Ray в этом отношении преуспевает, поскольку предоставляет хорошие абстракции, которые можно усваивать и использовать на интуитивном уровне. Движок фреймворка Ray выполняет всю тяжелую работу за вас в фоновом режиме. Именно эта философия внутреннего устройства позволяет использовать Ray с существующими библиотеками и системами Python.

Обратите внимание, что причина, по которой в этой книге мы начинаем с инструментария Ray Core, кроется в том, что, по нашему мнению, указанная библиотека обладает огромным потенциалом повышения доступности распределенных вычислений. По сути, вся эта глава посвящена тому, чтобы заглянуть за кулисы и понять, что придает фреймворку Ray хорошую работоспособность и как овладеть его основами. Если вы – менее опытный программист на Python или просто хотите сосредоточиться на задачах более высокого уровня, то к Ray Core потребуется некоторое привыкание<sup>2</sup>. С учетом этого мы настоятельно рекомендуем изучить API инструментария Ray Core, поскольку это отличный способ освоить распределенные вычисления с использованием Python.

<sup>1</sup> Когда мы писали эти строки, приборная панель находилась в стадии реконструкции. Как бы нам ни хотелось показать вам ее скриншоты и ознакомить с ней, пока вам придется открыть ее для себя самим (<https://oreil.ly/c6vtM>).

<sup>2</sup> В случае если вы относитесь к этим категориям, возможно, вам будет приятно услышать, что многие исследователи данных редко используют инструментарий Ray Core напрямую. Вместо этого они работают непосредственно с библиотеками фреймворка Ray более высокого уровня, такими как Data, Train или Tune.

## Первый пример использования Ray API

В качестве примера возьмем следующую ниже функцию, которая извлекает данные из базы данных и их обрабатывает. Наш образец базы данных представляет собой простой список на языке Python, содержащий слова из названия этой книги. Мы действуем так, как будто извлечение отдельного элемента из этой базы данных и дальнейшая его обработка обходятся дорого, путем перевода Python в спящий режим (`sleep`):

```
import time

database = [ ❶
    "Learning", "Ray",
    "Flexible", "Distributed", "Python", "for", "Machine", "Learning"
]

def retrieve(item):
    time.sleep(item / 10.) ❷
    return item, database[item]
```

- ❶ Образец базы данных, содержащей строковые данные с названием этой книги.  
❷ Эмулировать операцию обработки данных, которая занимает много времени.

Всего в нашей базе данных насчитывается восемь элементов. Если бы мы извлекали все элементы последовательно, то сколько времени это бы заняло? В случае элемента с индексом 5 мы ждем полсекунды ( $5/10$ ) и т. д. По нашим подсчетам, в общей сложности время выполнения составит около  $(0 + 1 + 2 + 3 + 4 + 5 + 6 + 7) / 10 = 2.8$  секунды. Давайте посмотрим, действительно ли это то, что мы получаем:

```
def print_runtime(input_data, start_time):
    print(f'Runtime: {time.time() - start_time:.2f} seconds, data:')
    print(*input_data, sep="\n")

start = time.time()
data = [retrieve(item) for item in range(8)] ❶
print_runtime(data, start) ❷
```

- ❶ Используется операция включения в список, чтобы извлечь все восемь элементов.  
❷ Выполняется распаковка данных, чтобы напечатать каждый элемент в отдельной строке.

Если выполнить этот исходный код, то вы должны увидеть следующий ниже результат:

```
Runtime: 2.82 seconds, data:
(0, 'Learning')
(1, 'Ray')
(2, 'Flexible')
(3, 'Distributed')
(4, 'Python')
(5, 'for')
(6, 'Machine')
(7, 'Learning')
```

Небольшие непроизводительные издержки увеличивают общее время выполнения до 2.82 секунды. С вашей стороны это может быть немного меньше или намного больше, в зависимости от вашего компьютера. Важным выводом является то, что наша наивная реализация на Python не способна выполнять эту функцию параллельно.

Возможно, это не станет для вас неожиданностью, но вы могли бы по крайней мере полагать, что операции включения в список на Python в этом отношении эффективнее. Полученное время выполнения в значительной степени соответствует наихудшему сценарию, а именно 2.8 секунды, которые мы рассчитали перед выполнением исходного кода. Если подумать, то даже немного неприятно видеть, что программа, которая по сути спит большую часть времени выполнения, в целом работает так медленно. В конечном счете в этом можно обвинить глобальную блокировку интерпретатора<sup>1</sup>, но на нее и так уже свалено немало вины.

### ГЛОБАЛЬНАЯ БЛОКИРОВКА ИНТЕРПРЕТАТОРА РУТНОН

Глобальная блокировка интерпретатора, несомненно, является одной из самых печально известных особенностей языка Python. В двух словах: это блокировка, которая обеспечивает, что только один поток на вашем компьютере сможет исполнять ваш исходный код Python одновременно. Если вы используете многопоточность, то потоки должны управлять интерпретатором Python по очереди.

Глобальная блокировка интерпретатора была внедрена по уважительным причинам. Прежде всего она значительно упрощает управление памятью в языке Python. Еще одним ключевым преимуществом является то, что она делает однопоточные программы довольно быстрыми. Программы, которые в основном используют много системного ввода-вывода (мы говорим, что они привязаны к вводу-выводу), такого как чтение файлов или баз данных, тоже выигрывают. Одним из основных недостатков является то, что привязанные к центральному процессору программы, по сути, являются однопоточными. В действительности привязанные к центральному процессору задания могут выполняться даже быстрее, если не использовать многопоточность, поскольку последняя приводит к непроизводительным издержкам на блокировку операции записи поверх глобальной блокировки.

Учитывая все это, если верить Ларри Хастингсу (Larry Hastings)<sup>2</sup>, глобальная блокировка интерпретатора, как это ни парадоксально, может быть одной из причин популярности Python. Интересно, что Гастингс также руководил (безуспешной) попыткой ее удалить в рамках проекта под названием GILectomy, которая является такой же сложной хирургической операцией, как и само название проекта. Жюри еще не определилось, но Сэм Гросс (Sam Gross)<sup>3</sup>, возможно, только что нашел способ удалить глобальную блокировку в своей ветке nogil для версии Python 3.9. На данный момент, если вам абсолютно необходимо обойти глобальную блокировку, рассмотрите возможность использования реализации, отличной от CPython. CPython – это стандартная реализация Python, и если вы не знаете, что используете ее, вы определенно ее используете. Такие реализации, как Jython, IronPython или PyPy, не имеют глобальной блокировки, но у них есть свои недостатки.

<sup>1</sup> Англ. Global Interpreter Lock (GIL). – Прим. перев.

<sup>2</sup> См. <https://oreil.ly/UShnM>.

<sup>3</sup> См. <https://oreil.ly/J4l-q>.

## Функции и дистанционные задания Ray

Разумно допустить, что такое задание может выиграть от параллелизации. Идеально распределенное время выполнения не должно занимать намного больше времени, чем самое длинное подзадание, а именно  $7/10 = 0.7$  секунды. Итак, давайте посмотрим, как этот пример можно было бы расширить за счет работы в Ray. С этой целью начнем с использования декоратора `@ray.remote`, как показано ниже:

```
@ray.remote ❶
def retrieve_task(item):
    return retrieve(item) ❷
```

- ❶ Сделать любую функцию Python заданием Ray с помощью этого декоратора.
- ❷ Все остальное остается неизменным. Функция извлечения задания (`retrieve_task`) просто передает управление функции `retrieve`.

Благодаря декоратору функция `retrieve_task` становится так называемым заданием Ray. По сути, задание Ray – это функция, которая выполняется в процессе, отличном от того, из которого она была вызвана, потенциально на другом компьютере.

Это чрезвычайно удобное конструктивное решение, поскольку есть возможность сначала сосредоточиваться на своем исходном коде Python, и не нужно полностью менять свое мышление или парадигму программирования, чтобы использовать фреймворк Ray. Обратите внимание, что на практике в изначальную функцию `retrieve` был бы просто добавлен декоратор `@ray.remote` (в конце концов, это целевое применение декораторов), но в целях обеспечения большей понятности мы не стали касаться приведенного выше исходного кода.

Все достаточно просто. Тогда что же нужно изменить в исходном коде, который извлекает записи базы данных и измеряет производительность? Оказывается, не так уж и много. Пример 2.1 показывает, что нужно сделать<sup>1</sup>.

### Пример 2.1 ♦ Измерение производительности задания Ray

```
start = time.time()
object_references = [ ❶
    retrieve_task.remote(item) for item in range(8)
]
data = ray.get(object_references) ❷
print_runtime(data, start)
```

- ❶ Для выполнения функции `retrieve_task` в своем локальном кластере Ray применяется метод `.remote()` и, как и раньше, передаются элементы. Каждое задание возвращает объект.

---

<sup>1</sup> Строго говоря, этот первый пример является своего рода антишаблоном (<https://oreil.ly/OuOWC>), поскольку обычно нельзя делиться мутируемым состоянием между заданиями Ray через глобальные переменные. С учетом сказанного в нашем примере с игрушечными данными не следует с этим переусердствовать. Будьте уверены, что в следующем разделе мы покажем более оптимальный способ.

- ❷ Для того чтобы вернуть фактические данные, а не только указатели на объекты Ray, применяется метод `ray.get`.

Заметили ли вы различия? Вы должны выполнить свое задание Ray дистанционно, используя вызов метода `.remote()`<sup>1</sup>. Когда фреймворк Ray выполняет задания дистанционно, даже в вашем локальном кластере, он делает это асинхронно. Элементы в списке указателей на объекты (`object_references`) в последнем фрагменте исходного кода не содержат результатов напрямую. В действительности, если с помощью `type(object_references[0])` проверить Python'овский тип первого элемента, то будет видно, что на самом деле это `ObjectRef`. Эти указатели на объекты соответствуют *фьючерсам*<sup>2</sup>, результат которых вам нужно запрашивать. Это то, для чего предназначен вызов `ray.get(...)`. Всякий раз, когда вы вызываете функцию `remote` для задания Ray, она немедленно возвращает один или несколько указателей на объекты. Задания Ray следует трактовать как главенствующий метод создания объектов. В следующем далее разделе мы покажем пример, который выстраивает несколько заданий в цепочку и позволяет Ray брать на себя передачу объектов и оперирование ими между заданиями.

Мы поработаем с этим примером<sup>3</sup> побольше, но давайте отступим на шаг назад и подведем итог тому, что мы сделали до сих пор. Мы начали с функции Python и декорировали ее декоратором `@ray.remote`, в результате чего ваша функция была превращена в простое задание. Затем вместо вызова изначальной функции в исходном коде мы вызвали метод `.remote(...)` в задании Ray. Последним шагом было применение метода `.get(...)`, чтобы получить результаты из кластера Ray. Эта процедура настолько интуитивно понятна, что вы, возможно, сможете создать свое собственное задание Ray из другой функции, не оглядываясь на этот пример. Почему бы не попробовать это сделать прямо сейчас?

Возвращаясь к нашему примеру, чего мы добились с точки зрения производительности, используя задания Ray? Для нас время выполнения составило 0.71 секунды, что лишь немного больше, чем время самого длинного подзадания, которое выполняется за 0.7 секунды. Это здорово и намного лучше, чем раньше, но нашу программу можно улучшить еще больше, задействовав больше функциональных возможностей API фреймворка Ray.

## Использование хранилища объектов с помощью `put` и `get`

Вы, возможно, заметили одну вещь – в определении функции `getieve` мы обращались к элементам из базы данных напрямую. При работе с локальным

<sup>1</sup> Эта книга предназначена для практиков науки о данных, поэтому здесь мы не будем обсуждать концептуальные детали архитектуры Ray. Если вам интересно и вы хотите узнать больше о том, как выполняются задания Ray, то рекомендуем ознакомиться с технической документацией по архитектуре фреймворка Ray (<https://oreil.ly/Pe-hT>).

<sup>2</sup> Фьючерс (futures) – это объект, который инкапсулирует асинхронное исполнение вызываемого объекта типа `Callable`. – Прим. перев.

<sup>3</sup> Этот пример был взят из фантастического доклада Дина Вамплера «Что такое фреймворк Ray? (Dean Wampler, What Is Ray?, <https://oreil.ly/Pe-hT>) и адаптирован.

кластером Ray это нормально, но представьте, что вы работаете в реально действующем кластере, который включает в себя несколько компьютеров. Как все эти компьютеры могли бы обращаться к одним и тем же данным? Напомним из главы 1, что в кластере Ray есть один головной узел с процессом-драйвером (выполняющим `ray.init()`) и множество узлов-работников с процессами-работниками, выполняющими ваши задания. По умолчанию Ray будет создавать столько процессов-работников, сколько имеется процессорных ядер на вашем компьютере. В настоящее время база данных определена только в драйвере, но выполняющие ваши задания работники должны иметь к ней доступ, чтобы выполнить задание `retrieve`. К счастью, фреймворк Ray предоставляет простой способ делиться объектами между драйвером и работниками (или между самими работниками). Для размещения данных в распределенном хранилище объектов Ray можно просто использовать `put`. В нашем определении функции `retrieve_task` мы в явной форме передаем аргумент `db` и позже передадим объект `db_object_ref`:

```
db_object_ref = ray.put(database) ❶  
  
@ray.remote  
def retrieve_task(item, db): ❷  
    time.sleep(item / 10.)  
    return item, db[item]
```

- ❶ Поместить (`put`) базу данных в хранилище объектов и получить ссылку на нее. Благодаря этому позже можно передать эту ссылку в явном виде в задание Ray.  
❷ Задание Ray `retrieve_task` принимает указатель на объект в качестве аргумента.

Используя хранилище объектов в таком ключе, фреймворк Ray можно наделять возможностью оперировать доступом к данным по всему кластеру. Мы поговорим о том, как именно передаются значения между узлами и внутри работников, когда будем говорить об инфраструктуре фреймворка Ray. Хотя взаимодействие с хранилищем объектов требует некоторых непроизводительных издержек, оно дает прирост производительности при работе с крупными и более реалистичными наборами данных. Сейчас же важным моментом является то, что этот шаг необходим в действительно распределенной среде. Если хотите, попробуйте перезапустить пример 2-1 с этой новой функцией `retrieve_task` и убедитесь, что она по-прежнему выполняется надлежащим образом.

## Применение функции `wait` фреймворка Ray для неблокирующих вызовов

Обратите внимание, как в примере 2-1 для доступа к результатам мы использовали вызов `ray.get(object_references)`. Этот вызов – блокирующий, и, следовательно, драйвер должен дождаться получения всех результатов. В нашем случае это не имеет большого значения; теперь программа завершается менее чем за секунду. Но представьте, что обработка каждого элемента базы данных занимала бы несколько минут. В этом случае, вместо того чтобы

сидеть сложа руки, вы хотели бы освободить процесс драйвера для других заданий. Кроме того, было бы здорово обрабатывать результаты по мере их поступления (некоторые заканчиваются намного быстрее, чем другие), вместо того чтобы ждать обработки всех элементов. И следует иметь в виду еще один вопрос: что произойдет, если один из элементов базы данных невозможно будет извлечь должным образом? Допустим, где-то в соединении с базой данных возникла взаимоблокировка. Драйвер просто зависал бы и не смог бы извлекать все элементы. По этой причине рекомендуется работать с разумными тайм-аутами. Допустим, мы не хотим ждать дольше, чем в 10 раз больше времени, необходимого для выполнения задания по извлечению данных, после чего будем его останавливать. Вот как это можно сделать с помощью фреймворка Ray, используя функцию `wait`:

```
start = time.time()
object_references = [
    retrieve_task.remote(item, db_object_ref) for item in range(8) ❶
]
all_data = []

while len(object_references) > 0: ❷
    finished, object_references = ray.wait(❸
        object_references, num_returns=2, timeout=7.0
    )
    data = ray.get(finished)
    print_runtime(data, start) ❹
    all_data.extend(data) ❺
```

- ❶ Выполнить `remote` в функции `retrieve_task` и передать соответствующий элемент (`item`), который мы хотим извлечь, и указатель на объект нашей базы данных.
- ❷ Вместо блокировки прокрутить список незаконченных указателей на объекты (`object_references`) в цикле.
- ❸ Мы асинхронно ожидаем (`wait`) готовых данных с разумным тайм-аутом (`timeout`). Список `object_references` здесь переопределяется, чтобы предотвратить бесконечный цикл.
- ❹ Печатать результаты по мере их поступления, а именно блоками по два элемента.
- ❺ Добавлять (`append`) новые данные к `all_data` до тех пор, пока они не закончатся.

Как видите, `ray.wait` возвращает два аргумента: законченные значения и фьючерсы, которые все еще нуждаются в обработке. Мы используем аргумент `num_returns`, который по умолчанию равен 1, чтобы позволить `wait` возвращаться всякий раз, когда доступна новая пара элементов базы данных. Это приводит к следующему ниже результату:

```
Runtime: 0.11 seconds, data:
(0, 'Learning')
(1, 'Ray')
Runtime: 0.31 seconds, data:
(2, 'Flexible')
(3, 'Distributed')
Runtime: 0.51 seconds, data:
(4, 'Python')
(5, 'for')
```

```
Runtime: 0.71 seconds, data:  
(6, 'Machine')  
(7, 'Learning')
```

Обратите внимание, что в цикле `while` мы могли бы делать много других вещей, вместо того чтобы просто печатать результаты, например запускать совершенно новые задания для других работников со значениями, уже извлеченными к этому моменту.

## Оперирование зависимостями заданий

До сих пор наш пример программы был концептуально довольно прост. Он состоит из одного шага: извлечения набора элементов из базы данных. Теперь представьте, что после загрузки данных вы хотите выполнять последующее задание обработки. Если говорить конкретнее, давайте предположим, что мы хотим использовать результат первого задания по извлечению, чтобы запрашивать другие, связанные данные (претворимся, что вы запрашиваете данные из другой таблицы в той же базе данных). Пример 2.2 формирует такое задание и последовательно выполняет и наше задание `retrieve_task`, и последующее задание `follow_up_task`.

### Пример 2.2 ♦ Выполнение последующего задания, которое зависит от другого задания Ray

```
@ray.remote  
def follow_up_task(retrieve_result): ❶  
    original_item, _ = retrieve_result  
    follow_up_result = retrieve(original_item + 1) ❷  
    return retrieve_result, follow_up_result ❸  
  
retrieve_refs = [retrieve_task.remote(item, db_object_ref) for item in [0, 2, 4, 6]]  
follow_up_refs = [follow_up_task.remote(ref) for ref in retrieve_refs]  
result = [print(data) for data in ray.get(follow_up_refs)] ❹
```

- ❶ Используя результат задания `retrieve_task`, вычислить другое задание Ray поверх него.
- ❷ Используя изначальный элемент (`original_item`) из первого задания, извлечь больше данных.
- ❸ Вернуть изначальные и последующие данные.
- ❹ Передать указатели на объекты из первого задания во второе задание.

Выполнение этого исходного кода приводит к следующему ниже результату:

```
((0, 'Learning'), (1, 'Ray'))  
((2, 'Flexible'), (3, 'Distributed'))  
((4, 'Python'), (5, 'for'))  
((6, 'Machine'), (7, 'Learning'))
```

Если у вас не так много опыта работы с асинхронным программированием, то пример 2.2, возможно, вас не впечатлит. Но мы надеемся убедить вас в том, что, по меньшей мере, немногого удивительно, что этот фрагмент

исходного кода вообще работает<sup>1</sup>. Итак, в чем же проблема? В конце концов, исходный код читается как обычный Python: определение функции и несколько включений в список. Дело в том, что тело функции `follow_up_task` ожидает входного аргумента `retrieve_result` в виде Python'овского кортежа, который мы распаковываем в первой строке определения функции.

Но, вызывая включение в список `[follow_up_task.remote(ref) for ref in retrieve_refs]`, мы вообще не передаем никаких кортежей последующему заданию. Вместо этого с `retrieve_refs` мы передаем указатели на объекты Ray. Под капотом происходит следующее: фреймворк Ray знает, что `follow_up_task` требует фактических значений, поэтому внутренне в этом задании он будет вызывать `ray.get` для урегулирования фьючерсов<sup>2</sup>. Ray строит граф зависимостей для всех заданий и исполняет их в порядке, учитывающем зависимости. Вам не нужно явно указывать фреймворку Ray, когда следует дожидаться завершения предыдущего задания; он сам будет выводить эту информацию за вас. Это также демонстрирует мощный функционал хранилища объектов Ray: если промежуточные значения велики, то можно обходиться без их копирования обратно в драйвер. Можно просто передавать указатели на объекты следующему заданию и позволять фреймворку Ray брать на себя все остальное.

Последующие задания будут запланированы только после завершения индивидуальных заданий по извлечению. На наш взгляд, это невероятная отличительная особенность. На самом деле если бы мы назвали `retrieve_refs` чем-то вроде `retrieve_result`, то вы, возможно, даже не заметили бы этой важной детали. Это заложено во внутреннее устройство фреймворка. Ray хочет, чтобы вы сосредоточились на своей работе, а не на деталях кластерных вычислений. На рис. 2.1 вы увидите график зависимостей для двух визуализированных заданий.

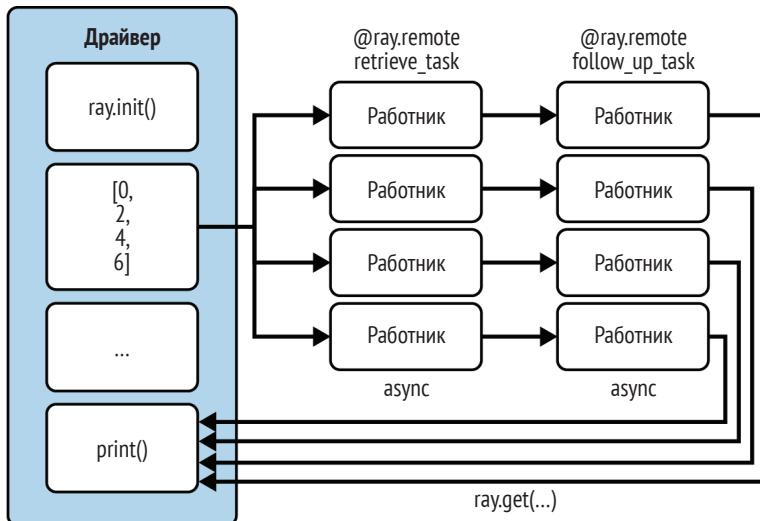
Если захотите, можете попробовать переписать пример 2-2 так, чтобы он явным образом использовал `get` в первом задании перед передачей значений в последующее задание. К сожалению, это не только привнесет больше шаблонного исходного кода, но и затруднит интуитивное понимание того, как его писать и интерпретировать.

## Из классов в акторы

Прежде чем завершить этот пример, давайте обсудим еще одну важную концепцию инструментария Ray Core. Обратите внимание, что в нашем примере все по сути является функцией. Мы использовали декоратор `ray.remote`, просто чтобы некоторые из них сделать дистанционными функциями, а в остальном применяли обычный Python.

<sup>1</sup> Согласно третьему закону Кларка ([https://oreil.ly/VHJ\\_o](https://oreil.ly/VHJ_o)), любая достаточно продвинутая технология не отличима от волшебства. Для меня в этом примере есть что-то волшебное.

<sup>2</sup> То же самое произошло ранее, когда мы передали указатель на объект дистанционному вызову задания `retrieve_task`, а затем напрямую обратились к соответствующим элементам базы данных `db`. Мы не хотели слишком сильно отвлекать вас от сути этого примера.



**Рис. 2.1** ♦ Выполнение двух зависимых заданий асинхронно и параллельно с помощью фреймворка Ray

Допустим, мы бы хотели отслеживать частоту запросов к базе данных. Конечно же, мы могли бы просто подсчитать результаты наших заданий по извлечению, но имеется ли более оптимальный способ это сделать? Мы хотим отслеживать это «распределенным» способом, который будет масштабироваться. Для этого во фреймворке Ray есть концепция *акторов*. Акторы позволяют выполнять вычисления с отслеживанием внутреннего состояния в кластере. Они также могут общаться друг с другом<sup>1</sup>. Подобно заданиям Ray, которые являются просто декорированными функциями Python, акторы Ray – это декорированные классы Python. Давайте напишем простой счетчик для отслеживания обращений к базе данных:

```

@ray.remote ①
class DataTracker:
    def __init__(self):
        self._counts = 0

    def increment(self):
        self._counts += 1

    def counts(self):
        return self._counts
  
```

- ① Сделать любой класс Python актором Ray, используя тот же декоратор `ray.remote`, что и раньше.

<sup>1</sup> Акторная модель – это устоявшаяся концепция в информатике, реализацию которой можно найти, например, в Akka или Erlang. Однако история и специфика акторов не имеют отношения к нашему изложению.

Класс `DataTracker` уже является актором, поскольку мы оснастили его декоратором `@ray.remote`. Этот актор может отслеживать состояние, здесь просто простой счетчик, а его методы являются заданиями Ray, которые вызываются точно так же, как мы делали с функциями раньше, а именно с помощью `.remote()`. Давайте поглядим, каким образом можно видоизменить существующее задание `retrieve_task`, чтобы включить этого нового актора:

```
@ray.remote
def retrieve_tracker_task(item, tracker, db): ❶
    time.sleep(item / 10.)
    tracker.increment.remote() ❷
    return item, db[item]

tracker = DataTracker.remote() ❸

object_references = [ ❹
    retrieve_tracker_task.remote(item, tracker, db_object_ref)
    for item in range(8)
]
data = ray.get(object_references)

print(data)
print(ray.get(tracker.counts.remote())) ❺
```

- ❶ Передает актора `tracker` в это задание.
- ❷ Трекер получает приращение (`increment`) при каждом вызове.
- ❸ Создает экземпляр актора `DataTracker`, вызывая `.remote()` на классе.
- ❹ Актор передается в задание по извлечению данных.
- ❺ Впоследствии от актора `tracker` можно получить состояние `counts` с помощью еще одного дистанционного вызова.

Неудивительно, что результат этого вычисления на самом деле равен 8. Для его вычисления нам не нужны акторы, но бывает полезно иметь механизм отслеживания состояния по всему кластеру, потенциально охватывающий несколько заданий. Мы могли бы передать нашего актора практически в любое зависимое задание или даже в конструктор еще одного актора. Нет никаких ограничений на то, что можно сделать, и именно эта гибкость делает Ray API столь мощным. Нечасто можно увидеть, чтобы распределенные инструменты Python допускали подобные вычисления с отслеживанием внутреннего состояния. Эта функциональная возможность может пригодиться, в особенности при выполнении сложных распределенных алгоритмов, например при использовании обучения с подкреплением.

На этом наш первый обширный пример с использованием Ray API завершается. Далее давайте кратко подытожим этот API.

- ✓ В данном введении на конкретном примере мы уделили большое внимание заданиям и акторам фреймворка Ray как распределенным версиям функций и классов Python. Но *объекты* тоже являются полноправными элементами инструментария Ray Core и должны рассматриваться как равные по статусу заданиям и акторам. Хранилище объектов является центральным компонентом фреймворка Ray.

## Краткий обзор API инструментария Ray Core

Если вы вспомните, что мы делали в предыдущем примере, то заметите, что мы применили в общей сложности всего шесть методов API<sup>1</sup>. Мы использовали `ray.init()` для запуска кластера и `@ray.remote` для преобразования функций и классов в задания и акторы. Затем мы использовали `ray.put()` для передачи значений в хранилище объектов Ray и `ray.get()` для извлечения объектов из кластера. Наконец, мы использовали `.remote()` на акторных методах либо заданиях для выполнения исходного кода в кластере и `ray.wait` для избегания блокировки вызовов.

Хотя может показаться, что этих шесть методов API не так много, они – единственные, которые, вероятно, когда-либо будут вас интересовать при применении Ray API<sup>2</sup>. Мы кратко резюмируем их в табл. 2.1, чтобы вы могли легко ссылаться на них в будущем.

**Таблица 2.1. Шесть главнейших методов API инструментария Ray Core**

API-вызов	Описание
<code>ray.init()</code>	Инициализирует кластер Ray. При передаче аргумента <code>address</code> происходит подключение к существующему кластеру
<code>@ray.remote</code>	Превращает функции в задания, а классы – в акторы
<code>ray.put()</code>	Помещает значения в хранилище объектов Ray
<code>ray.get()</code>	Получает значения из хранилища объектов. Возвращает значения, которые вы туда поместили ( <code>put</code> ) либо которые были вычислены заданием или актором
<code>.remote()</code>	Выполняет акторные методы или задания в кластере Ray и используется для создания экземпляров акторов
<code>ray.wait()</code>	Возвращает два списка указателей на объекты, один с завершенными заданиями, которых мы ожидаем, и другой с незавершенными заданиями

Теперь, когда вы увидели Ray API в действии, давайте уделим немного времени его системной архитектуре.

<sup>1</sup> Перефразируя Алана Кая (Alan Kay, <https://oreil.ly/LNdxI>): чтобы добиться простоты, нужно найти чуть более сложные строительные блоки. Ray API делает для распределенных вычислений на Python именно это.

<sup>2</sup> Ознакомьтесь со справочным руководством по API (<https://oreil.ly/k3E7H>), чтобы увидеть, что на самом деле имеется еще довольно много методов. В какой-то момент вам следует вложиться в понимание аргументов функции `init`, но все остальные методы, скорее всего, не будут вам интересны, если только вы не являетесь администратором своего кластера Ray.

# ПОНИМАНИЕ СИСТЕМНЫХ КОМПОНЕНТОВ ФРЕЙМВОРКА RAY

Вы научились использовать Ray API и поняли философию внутреннего устройства, лежащую в основе фреймворка Ray. Теперь самое время получше разобраться в опорных компонентах системы. Другими словами, в том, как фреймворт Ray работает и как он достигает того, что делает.

## Планирование и исполнение работы на узле

Вы знаете, что кластеры Ray состоят из узлов. Сначала мы посмотрим на то, что происходит на отдельных узлах, а затем взглянем пошире и проанализируем взаимодействие всего кластера.

Как мы уже говорили, узел-работник состоит из нескольких процессов-работников, или просто работников. У каждого работника есть уникальный идентификатор, IP-адрес и порт, по которому на них можно ссылаться. Работников называют «работниками» не просто так; это компоненты, которые слепо исполняют ту работу, которую вы им поручаете. Но кто говорит им, что делать и когда? Работник может уже быть занятым, у него могут отсутствовать ресурсы, необходимые для выполнения задания (например, отсутствовать доступ к графическому процессору), и у него даже могут отсутствовать значения, необходимые для выполнения данного задания. Вдобавок ко всему работники не имеют представления о том, что происходит до или после исполнения ими своей рабочей нагрузки; отсутствует координация.

Для урегулирования этих проблем у каждого узла-работника есть компонент под названием *Raylet*. Их следует рассматривать как умные компоненты узла, которые управляют процессами-работниками. Компоненты Raylet используются совместно между заявками на выполнение работы и состоят из двух элементов: *планировщика задач* и *хранилища объектов*.

Сначала давайте поговорим о хранилищах объектов. В работающем примере этой главы мы уже использовали концепцию хранилища объектов в широком смысле, не указывая ее явно. Каждый узел кластера Ray оснащен хранилищем объектов в пределах компонента Raylet этого узла, и все объекты, находящиеся в хранилище, образуют распределенное хранилище объектов кластера. Хранилище объектов управляет *совместным пулом памяти* для всех работников на одном узле и обеспечивает работникам доступ к объектам, которые были созданы на другом узле. Хранилище объектов реализовано как резидентное хранилище Plasma<sup>1</sup>, которое теперь принадлежит проекту Apache Arrow. В функциональном плане хранилище объектов берет на себя управление памятью и в конечном счете обеспечивает работникам доступ к нужным им объектам.

<sup>1</sup> См. <https://arrow.apache.org/docs/1.0/python/plasma.html>.

Вторым элементом компонента Raylet является его планировщик. Планировщик, помимо прочего, берет на себя *управление ресурсами*. Например, если заданию требуется доступ к четырем центральным процессорам, то планировщик должен удостовериться, что он может найти свободный процесс-работник, который может предоставить доступ к указанным ресурсам. По умолчанию планировщик знает и получает информацию о количестве центральных и графических процессоров, а также об объеме памяти, доступной на его узле. Если планировщик не может предоставить необходимые ресурсы, то он просто не сможет запланировать выполнение задания сразу и должен поставить его в очередь. Планировщик ограничивает число заданий, выполняемых конкурентно, чтобы обеспечивать нескончаемость физических ресурсов.

Помимо ресурсов, другой работой, которую берет на себя планировщик, является *урегулирование зависимостей*. Это означает, что он должен обеспечивать наличие у каждого работника всех объектов, необходимых для выполнения задания в локальном хранилище объектов. Для этого планировщик сначала обрабатывает локальные зависимости, выполняя поиск значений в своем хранилище объектов. Если требуемое значение в хранилище объектов этого узла недоступно, то планировщик связывается с другими узлами (мы немного расскажем о том, как это делать) и вовлекает дистанционные зависимости. После того как планировщик обеспечил достаточное для задания количество ресурсов, урегулировал все необходимые зависимости и нашел работника для задания, он может поставить выполнение задания в план.

Планирование заданий – очень сложная тема, даже если мы говорим только об отдельных узлах. Можно легко представить себе сценарии, в которых неправильно или наивно спланированное выполнение задания может «заблокировать» нижестоящие задания из-за нехватки оставшихся ресурсов. Распределение подобной работы может очень быстро стать сложной задачей, в особенности в распределенном контексте.

Теперь, когда вы знаете о компонентах Raylet, давайте кратко вернемся к процессам-работникам и завершим изложение объяснением того, как фреймворк Ray может восстанавливаться после сбоев, и необходимых для этого концепций.

Если говорить коротко, то работники хранят метаданные всех заданий, которые они вызывают, и возвращаемые этими заданиями указатели на объекты. Эта концепция, именуемая *владением*, означает, что процесс, который генерирует указатель на объект, также отвечает за его урегулирование. Другими словами, каждый процесс-работник «владеет» заданиями, которые он отправляет. А это предусматривает надлежащее исполнение и обеспечение доступности результатов. Процессы-работники должны отслеживать то, чем они владеют, например в случае сбоев, вот почему у них есть так называемая *таблица владения*. Благодаря ей, если задача завершается безуспешно и ее необходимо вычислить заново, работник уже владеет всей необходимой для этого информацией<sup>1</sup>. В качестве конкретного примера отношений владе-

---

<sup>1</sup> Это крайне ограниченное описание того, как фреймворк Ray обрабатывает сбои в целом. В конце концов, простое наличие всей информации для восстановления еще не говорит о том, как это сделать. Мы отсылаем вас к технической документации по архитектуре ([https://oreil.ly/\\_1SyA](https://oreil.ly/_1SyA)), в которой эта тема подробно обсуждается.

ния, в отличие от рассмотренной ранее концепции зависимости, давайте предположим, что у нас есть программа, которая запускает простое задание и внутренне вызывает еще одно задание:

```
@ray.remote
def task_owned():
    return

@ray.remote
def task(dependency):
    res_owned = task_owned.remote()
    return

val = ray.put("value")
res = task.remote(dependency=val)
```

Давайте бегло проанализируем концепции владения и зависимости в этом примере. В функциях `task` и `task_owned` мы определили два задания, и всего у нас три переменные: `val`, `res` и `res_owned`. Наша главная программа определяет значение переменной `val` (помещая "value" в хранилище объектов) и переменной `res`, а также вызывает `task`. Другими словами, драйвер владеет заданием `task`, переменными `val` и `res` в соответствии с определением отношения владения Ray. Напротив, переменная `res` зависит от задания `task`, но между ними нет отношения владения. При вызове задания `task` оно принимает значение переменной `val` в качестве зависимости. Затем оно вызывает `task_owned` и присваивает его переменной `res_owned` и, следовательно, владеет ими обоими. Наконец, задание `task_owned` само по себе ничем не владеет, но, безусловно, переменная `res_owned` зависит от него. Рисунок 2.2 подводит итог обсуждению темы узлов-работников, показывая все задействованные компоненты.

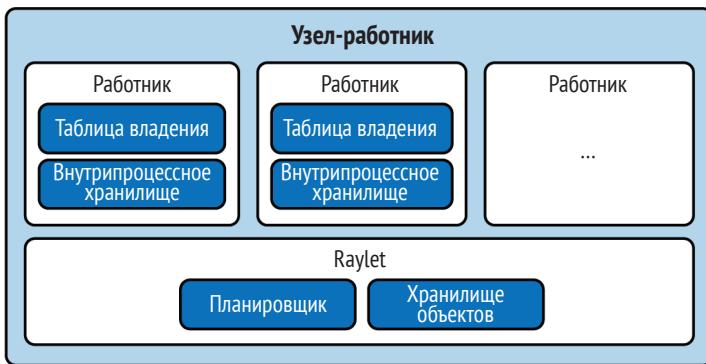


Рис. 2.2 ♦ Компоненты системы, содержащие узел-рабочник Ray

## Головной узел

В главе 1 мы уже указывали на то, что у каждого кластера Ray есть один специальный узел, именуемый *головным узлом*. Пока что вы знаете, что на этом

узле есть процесс-драйвер<sup>1</sup>. Драйверы могут сами отправлять задания, но не могут их исполнять. Вы также знаете, что на головном узле могут быть процессы-работники, что важно для возможности работы локальных кластеров, состоящих из одного-единственного узла.

Головной узел идентичен другим узлам-работникам, но на нем дополнительно выполняются процессы, ответственные за управление кластером, такие как автомасштабировщик (который мы рассмотрим в главе 9) и компонент под названием Служба глобального управления<sup>2</sup>. Это важный компонент, который несет глобальную информацию о кластере. Указанная служба представляет собой хранилище в формате ключ-значение, в котором хранится такая информация, как метаданные системного уровня. Например, в нем есть таблица с пульсирующими сигналами по каждому компоненту Raylet, чтобы обеспечивать их доступность. Компоненты Raylet, в свою очередь, посылают пульсирующие сигналы в Службу управления, сообщая о том, что они живы. В Службе глобального управления также хранятся местоположения акторов Ray. Рассмотренная выше модель владения говорит о том, что вся информация об объектах хранится в процессе-работнике их владельца, что позволяет избегать превращения Службы глобального управления в узкое место.

## Распределенное планирование и исполнение

Давайте вкратце обсудим оркестровку кластера и способы управления, планирования и исполнения заданий узлами. Говоря об узлах-работниках, мы указали, что для распределения рабочих нагрузок с помощью фреймворка Ray существует несколько компонентов. Ниже приведен обзор шагов и тонкостей, связанных с этим процессом.

### Распределенная память

Хранилища объектов в индивидуальных компонентах Raylet управляют памятью на узле. Но иногда возникает необходимость передавать объекты между узлами. Такая операция называется *распределенным трансфером объектов*<sup>3</sup>. Она необходима для дистанционного урегулирования зависимостей, чтобы у работников были объекты, необходимые им для выполнения заданий.

### Связь

Большая часть коммуникаций в кластере Ray, например трансфер объектов, происходят через фреймворк дистанционного вызова процедур gRPC<sup>4</sup>.

---

<sup>1</sup> На самом деле у него может быть несколько драйверов, но это несущественно в рамках нашего изложения. Наиболее распространенным является запуск одного драйвера на головном узле, но процессы-драйверы можно запускать и на любом узле кластера, а в одном кластере может находиться несколько драйверов.

<sup>2</sup> Англ. Global Control Service (GCS).

<sup>3</sup> Англ. distributed object transfer, син. распределенная передача объектов. – Прим. перев.

<sup>4</sup> См. <https://grpc.io/>.

## *Управление ресурсами и их реализация*

Компоненты Raylet на узле отвечают за предоставление ресурсов и сдачу процессов-работников в аренду владельцам заданий. Все планировщики на узлах формируют распределенного планировщика, что фактически означает, что узлы могут планировать задания на других узлах. Благодаря связи со Службой глобального управления локальные планировщики узнают о ресурсах других узлов.

## *Исполнение заданий*

После того как задание было передано на исполнение, все его зависимости (локальные и дистанционные данные) должны быть урегулированы, например путем извлечения больших данных из хранилища объектов, после чего можно будет начинать исполнение.

Если последние несколько разделов кажутся немного изощренными с технической точки зрения, то это потому, что так оно и есть. Важно понимать базовые шаблоны и идеи программного обеспечения, которое вы используете, но мы признаем, что разобраться в деталях архитектуры фреймворка Ray поначалу вам, возможно, будет сложновато. На самом деле одним из принципов внутреннего устройства фреймворка Ray является компромисс между удобством использования и архитектурной сложностью. Если вы хотите разобраться в архитектуре Ray поглубже, то хорошим местом для начала будет техническая документация по его архитектуре<sup>1</sup>.

На рис. 2.3 резюмируется то, что мы знаем об архитектуре фреймворка Ray.

Теперь, когда вы изучили основы API инструментария Ray Core и знакомы с основами кластерной архитектуры фреймворка Ray, давайте рассмотрим еще один сложный пример.

## **СИСТЕМЫ, СВЯЗАННЫЕ С ФРЕЙМВОРКОМ RAY**

Учитывая архитектуру и функциональность фреймворка Ray, как он соотносится с другими системами? Ниже перечислены основные моменты.

- Фреймворк Ray можно использовать в качестве фреймворка параллелизации для Python, и он может делиться свойствами с такими инструментами, как `celery` или `multiprocessing`. На самом деле в Ray реализована упрощенная замена<sup>2</sup> последнего.
- Фреймворк Ray также связан с такими платформами обработки данных, как Spark, Dask, Flink и MARS. Мы разведаем эти взаимосвязи в главе 11, когда будем говорить об экосистеме фреймворка Ray.
- Фреймворк Ray как инструмент распределенных вычислений также решает проблемы управления кластерами и оркестровки, и в главе 9 мы увидим, как Ray это делает применительно к таким инструментам, как Kubernetes.
- Поскольку фреймворк Ray реализует акторную модель конкурентности, также интересно обследовать его взаимосвязь с такими фреймворками, как Akka.
- Наконец, поскольку фреймворк Ray делает ставку на производительный низкоуровневый API в отношении связи, существует определенная взаимосвязь с фреймворками высокопроизводительных вычислений (HPC) и протоколами связи, такими как интерфейс передачи сообщений (MPI).

<sup>1</sup> См. <https://oreil.ly/tadqC>.

<sup>2</sup> См. <https://oreil.ly/4Lrgf>.

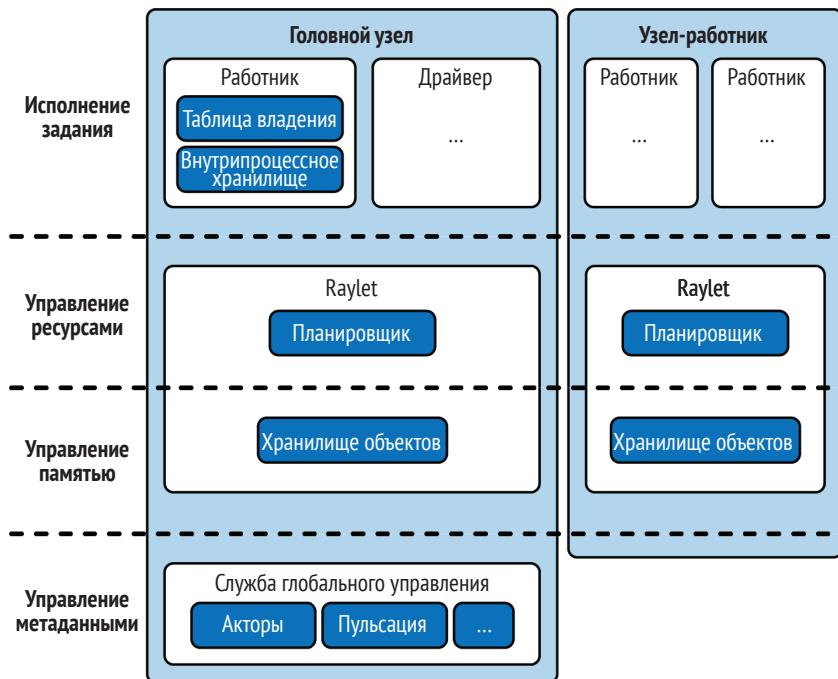


Рис. 2.3 ♦ Обзор архитектурных компонентов фреймворка Ray

## ПРОСТОЙ ПРИМЕР ИСПОЛЬЗОВАНИЯ ПАРАДИГМЫ MapReduce С ФРЕЙМВОРКОМ RAY

Мы не отпустим вас без того, чтобы обсудить пример одного из самых важных достижений в области распределенных вычислений за последние десятилетия, а именно парадигмы MapReduce. Многие успешные технологии обработки больших данных, такие как Hadoop, основаны на этой модели программирования, и к ней стоит вернуться в контексте фреймворка Ray. В целях упрощения задачи мы ограничим нашу реализацию алгоритма MapReduce одним вариантом использования – задачей подсчета вхождений слов в нескольких документах. В случае однократной обработки это почти тривиальная задача, но она становится интересной, когда задействован массивный корпус документов и вам требуется несколько вычислительных узлов, чтобы перемалывать числа.

Реализация примера подсчета слов с использованием парадигмы MapReduce, возможно, является самым известным примером, который имеется в области распределенных вычислений<sup>1</sup>, так что его стоит знать. Если вы не

<sup>1</sup> Это своего рода дрозофилы фруктовая, которая мало чем отличается от вычисления классификатора на основе повсеместно распространенного набора данных MNIST.

знаете об этой классической парадигме, то она основана на трех простых шагах:

- 1) взять набор документов и преобразовать их, или «отобразить» их элементы (например, содержащиеся в них слова), в соответствии с предоставленной вами функцией. Данная *фаза отображения* продуцирует пары *ключ-значение* по изначальному плану, в которых *ключ* представляет элемент документа, а *значение* – это просто метрика, которую нужно вычислить для этого элемента. Поскольку мы заинтересованы в подсчете слов, то всякий раз, когда мы встречаем слово в документе, наша *функция отображения* просто выдает пару (*word*, 1), тем самым указывая, что мы нашли одно его вхождение;
- 2) собрать и сгруппировать все результаты фазы отображения в соответствии с их ключом. Поскольку мы работаем в распределенной среде и один и тот же ключ может присутствовать на нескольких вычислительных узлах, это может потребовать перетасовки данных между узлами. По этой причине этот шаг часто называют *фазой перетасовки*<sup>1</sup>. Для того чтобы дать вам представление о группировке в нашем конкретном случае использования, предположим, что в общей сложности у нас есть четыре вхождения (*word*, 1), созданных на этапе отображения. Тогда перетасовка расположит все вхождения одного и того же слова совместно на одном и том же узле;
- 3) агрегировать, или «редуцировать», элементы из фазы перетасовки, именно поэтому мы называем ее *фазой редукции*. Продолжая приведенный нами пример, мы просто суммируем все вхождения слов в каждом узле, чтобы получить окончательное количество. Например, четыре вхождения (*word*, 1) были бы редуцированы до *word*: 4.

Очевидно, что MapReduce получил свое название от первой и последней фаз, но вторая фаза, возможно, не менее важна. Хотя схематически эти фазы могут выглядеть простыми, их сила заключается в том, что они могут быть массово параллелизованы на сотнях машин.

На рис. 2.4 иллюстрируется пример применения трех фаз алгоритма MapReduce к корпусу документов, который был распределен по трем разделам. При выполнении алгоритма MapReduce на распределенном корпусе документов мы сначала отображаем каждый документ в набор пар *ключ-значение*, затем перетасовываем результаты, чтобы обеспечить расположение всех пар *ключ-значение* с одинаковым ключом на одном узле, и, наконец, редуцируем пары *ключ-значение*, чтобы вычислить окончательное количество слов.

Давайте реализуем алгоритм MapReduce для нашего варианта использования с подсчетом слов на Python и параллелизуем вычисления с помощью фреймворка Ray. Сначала загрузим данные примера, чтобы вы получили более четкое представление о том, с чем мы работаем:

---

<sup>1</sup> В общем случае перетасовка – это любая операция, требующая перераспределения данных по разделам. Перетасовка бывает довольно дорогостоящей. Если фаза отображения работает с  $N$  разделами, то она будет выдавать  $N \times N$  результатов, подлежащих перетасовке.

```
import subprocess

zen_of_python = subprocess.check_output(["python", "-c", "import this"])
corpus = zen_of_python.split() ①

num_partitions = 3
chunk = len(corpus) // num_partitions
partitions = [②
    corpus[i * chunk: (i + 1) * chunk] for i in range(num_partitions)
]
```

- ① Наш текстовый корпус – это содержимое свода рекомендаций «Дзен языка Python».  
 ② Разбить корпус на три раздела.

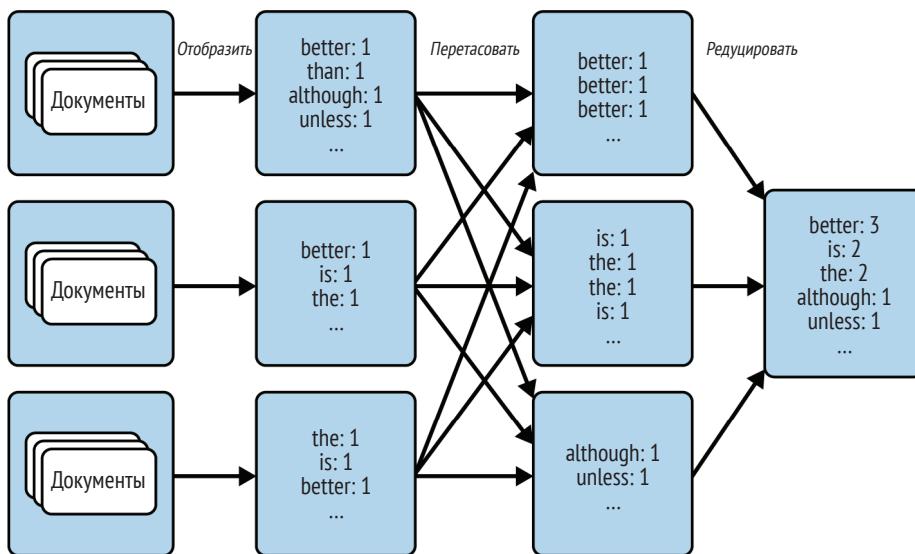


Рис. 2.4 ♦ Выполнение алгоритма MapReduce на распределенном корпусе документов

Используемые нами данные называются «Дзен языка Python». Это небольшой свод рекомендаций сообщества Python. Дзен скрыт в «пасхальном яйце» и выводится на печать при наборе команды `import this` в сеансе Python. Вам как программисту на Python стоит прочитать эти рекомендации, но в данном упражнении нас интересует только подсчет слов, которые в них содержатся. Проще говоря, мы загружаем Дзен языка Python, трактуем каждую строку как отдельный «документ» и разбиваем его на три раздела.

Начиная нашу реализацию алгоритма MapReduce, мы сначала рассмотрим фазу отображения и разберем, как фреймворк Ray нам поможет взять на себя перетасовку результатов.

## Отображение и перетасовка данных в документах

Для определения фазы отображения нужна функция отображения, которую мы применяем к каждому документу. В нашем случае мы хотим выдавать пару (`word, 1`) для каждого слова, которое находим в документе. В случае простых текстовых документов, загружаемых в виде строковых литералов Python, это выглядит следующим образом<sup>1</sup>:

```
def map_function(document):
    for word in document.lower().split():
        yield word, 1
```

Далее мы хотим применить эту функцию отображения ко всему корпусу документов. Мы делаем это, превращая следующую ниже функцию `apply_map` в задание Ray с помощью декоратора `@ray.remote`. При вызове функции `apply_map` мы применяем ее к трем разделам (`num_partitions=3`) данных документа, точно так же, как было указано на рис. 2-4. Обратите внимание, что функция `apply_map` будет возвращать три списка, по одному для каждого раздела. Как вы сейчас увидите, мы делаем это для того, чтобы фреймворк Ray мог автоматически перетасовывать результаты фазы отображения в нужные нам узлы:

```
import ray

@ray.remote
def apply_map(corporus, num_partitions=3):
    map_results = [list() for _ in range(num_partitions)] ❶
    for document in corporus:
        for result in map_function(document):
            first_letter = result[0].decode("utf-8")[0]
            word_index = ord(first_letter) % num_partitions ❷
            map_results[word_index].append(result) ❸
    return map_results
```

- ❶ Задание Ray `apply_map` возвращает по одному результату для каждого раздела данных.
- ❷ Назначить каждую пару (`word, 1`) разделу, используя функцию `ord`, чтобы сгенерировать `word_index`. За счет этого каждое вхождение слова будет перетасовано в один и тот же раздел.
- ❸ Затем пары последовательно добавляются к правильному списку.

Для текстового корпуса, который можно загрузить на одну машину, это перебор, и вместо этого мы могли бы просто подсчитать слова. Но в распределенной среде, в которой нам приходится распределять данные по нескольким узлам, эта фаза отображения имеет смысл.

---

<sup>1</sup> Обратите внимание на использование инструкции `yield` в функции `map`. Это самый быстрый способ создания генератора на Python с нужными нам данными. Вы также могли бы формировать и возвращать (`return`) список пар, если вам такой вариант понятнее.

При параллельном применении фазы отображения к корпусу документов мы используем дистанционный вызов `apply_map`, как делали много раз ранее в этой главе. Заметным отличием является то, что теперь с помощью аргумента `num_returns` мы также сообщаем фреймворку Ray, что нужно возвращать три результата (по одному для каждого раздела):

```
ap_results = [
    apply_map.options(num_returns=num_partitions) ❶
        .remote(data, num_partitions) ❷
        for data in partitions ❸
    ]

for i in range(num_partitions):
    mapper_results = ray.get(map_results[i]) ❹
    for j, result in enumerate(mapper_results):
        print(f"Отображатель {i}, возвращаемое значение {j}: {result[:2]}")
```

- ❶ Использовать `options`, чтобы сообщить фреймворку Ray, что нужно возвращать значения `num_partitions`.
- ❷ Исполнить `apply_map` дистанционно.
- ❸ Прокрутить в цикле все определенные нами разделы (`partitions`).
- ❹ Проинспектировать результаты только в целях иллюстрации. Обычно `ray.get` на этом шаге не вызывается.

Если вы выполните приведенный выше исходный код, то увидите, что результат каждой фазы отображения состоит из трех списков, в которых мы печатаем первые два элемента каждого:

```
Отображатель 0, возвращаемое значение 0: [(b'of', 1), (b'is', 1)]
Отображатель 0, возвращаемое значение 1: [(b'python', 1), (b'peters', 1)]
Отображатель 0, возвращаемое значение 2: [(b'the', 1), (b'zen', 1)]
Отображатель 1, возвращаемое значение 0: [(b'unless', 1), (b'in', 1)]
Отображатель 1, возвращаемое значение 1: [(b'although', 1), (b'practicality', 1)]
Отображатель 1, возвращаемое значение 2: [(b'beats', 1), (b'errors', 1)]
Отображатель 2, возвращаемое значение 0: [(b'is', 1), (b'is', 1)]
Отображатель 2, возвращаемое значение 1: [(b'although', 1), (b'a', 1)]
Отображатель 2, возвращаемое значение 2: [(b'better', 1), (b'than', 1)]
```

Как вы увидите, есть возможность сделать так, чтобы в фазе редукции все пары из  $j$ -го возвращенного значения оказывались на одном и том же узле<sup>1</sup>. Давайте обсудим эту фазу далее.

## Редукция количеств слов

Теперь в фазе редукции можно просто создать словарь, который суммирует все вхождения слов в каждом разделе:

---

<sup>1</sup> Согласно внутреннему устройству алгоритма, все пары с одинаковыми ключами окажутся на одном и том же узле. Например, обратите внимание, как в образце результата мы напечатали слово `is`, которое появляется в 0-м возвращенном значении двух отображателей. В фазе редукции все вхождения `is` окажутся в одном и том же разделе.

```

@ray.remote
def apply_reduce(*results): ①
    reduce_results = dict()
    for res in results:
        for key, value in res:
            if key not in reduce_results:
                reduce_results[key] = 0
            reduce_results[key] += value ②

    return reduce_results

```

- ① Редуцировать список перетасованных отображенных результатов.  
 ② Прокрутить в цикле все полученные в фазе отображения результаты (`result`) и увеличивать количество слов на единицу по каждому вхождению слова.

Теперь можно собрать  $j$ -е возвращаемое значение из каждого отображателя и передать его в  $j$ -й редуктор, как показано ниже. Обратите внимание, что здесь мы используем игрушечный набор данных, но указанный исходный код будет масштабироваться до наборов данных, которые не помещаются на одной машине. Это связано с тем, что в редукторы мы передаем не фактические данные, а указатели на объекты Ray. Фазы отображения и редукции – это задания Ray, которые могут выполняться в любом кластере Ray, и перетасовка данных также обрабатывается фреймворком Ray:

```

outputs = []
for i in range(num_partitions):
    outputs.append(①
        apply_reduce.remote(*[partition[i] for partition in map_results])
    )

counts = {k: v for output in ray.get(outputs) for k, v in output.items()} ②

sorted_counts = sorted(counts.items(), key=lambda item: item[1], reverse=True) ③
for count in sorted_counts:
    print(f'{count[0].decode('utf-8')}: {count[1]}')

```

- ① Собрать по одному результату из каждого задания отображения и передать его в `apply_reduce`.  
 ② Собрать все результаты фазы редукции в одном словаре Python `count`.  
 ③ Распечатать отсортированное количество слов по всему корпусу.

Выполнение этого примера приведет к следующему ниже результату:

```

is: 10
than: 8
better: 8
the: 6
to: 5
although: 3
...

```

Если вы хотите поглубже разобраться в том, как масштабировать задания MapReduce на несколько узлов с помощью фреймворка Ray, включая под-

робные соображения по поводу памяти, мы рекомендуем изучить отличный блог-пост<sup>1</sup> на эту тему.

Важная часть этого примера с MapReduce заключается в том, насколько гибкой на самом деле является *принятая во фреймворке Ray модель программирования*. Конечно, реализация алгоритма MapReduce производственного уровня требует немного больше усилий. Но способность быстро воспроизводить распространенные алгоритмы, подобные этому, имеет большое значение. Имейте в виду, что на более ранних этапах парадигмы MapReduce (скажем, примерно в 2010 году) она часто была единственным, что требовалось для выражения рабочих нагрузок. С помощью фреймворка Ray любому программисту на Python среднего уровня становится доступен целый ряд интересных шаблонов распределенных вычислений<sup>2</sup>.

## Резюме

В этой главе вы познакомились с основами Ray API на практике. Вы знаете, как помещать (`put`) значения в хранилище объектов и как получать (`get`) их обратно. Кроме того, вы познакомились с объявлением функций Python как заданий Ray с помощью декоратора `@ray.remote`, и вы научились выполнять их в кластере Ray с помощью вызова `.remote()`. Во многом таким же образом вы понимаете, как объявлять актора Ray из класса Python и как создавать его экземпляр и использовать для распределенных вычислений с отслеживанием внутреннего состояния.

Вдобавок ко всему вы также познакомились с основами кластеров Ray. Запуская их с помощью `ray.init(...)`, вы узнали, что можете подавать заявки на выполнение работы, состоящей из заданий, в свой кластер. Процесс-драйвер, находящийся на головном узле, затем распределяет задания по узлам-работникам. Компоненты Raylet на каждом узле планируют задания, а процессы-работники их исполняют. Вы также увидели быструю реализацию парадигмы MapReduce с помощью фреймворка Ray в качестве примера распределенного шаблона разработки приложений Ray.

Представленный в данной главе краткий обзор инструментария Ray Core должен помочь вам приступить к написанию своих собственных распределенных программ. В главе 3 мы проверим ваши знания, реализовав базовое приложение машинного обучения.

---

<sup>1</sup> См. <https://oreil.ly/ROSPr>.

<sup>2</sup> Мы рекомендуем ознакомиться с подробными шаблонами и антишаблонами фреймворка Ray как для заданий (<https://oreil.ly/dWaSg>), так и для акторов (<https://oreil.ly/s6eLw>).

# Глава 3

---

## Разработка первого распределенного приложения

Теперь, когда вы познакомились с основами Ray API на практике, давайте разработаем с его помощью что-то более реалистичное. К концу этой главы вы разработаете задачу обучения с подкреплением с чистого листа, реализуете свой первый алгоритм для ее решения и примените задания и акторы Ray для параллелизации этого решения в локальном кластере – и все это менее чем в 25 строках исходного кода.

Эта глава предназначена для читателей, у которых нет никакого опыта с обучением с подкреплением. Мы поработаем над простой задачей и разовьем необходимые навыки ее решения на практике. Поскольку глава 4 полностью посвящена этой теме, мы пропустим все продвинутые темы и язык обучения с подкреплением и просто сосредоточимся на поставленной задаче. Но даже если вы являетесь довольно продвинутым пользователем обучения с подкреплением, вы, скорее всего, выиграете от реализации классического алгоритма в распределенной среде.

Это последняя глава, которая будет посвящена работе только с инструментарием Ray Core. Мы надеемся, что вы научитесь ценить его мощь, гибкость и скорость, с которыми можно реализовывать распределенные эксперименты, масштабирование которых в противном случае потребовало бы значительных усилий.

Прежде чем перейти к какой-либо реализации, давайте чуть-чуть подробнее остановимся на парадигме обучения с подкреплением. Этот раздел можно спокойно пропустить, если вы раньше с этой парадигмой работали.

### ВВЕДЕНИЕ В ОБУЧЕНИЕ С ПОДКРЕПЛЕНИЕМ

Одно из моих (Макса) любимых мобильных приложений способно автоматически классифицировать, или «помечать», отдельные растения в нашем саду.

Работа приложения сводится к тому, что ему нужно показывать изображение соответствующего растения. Это чрезвычайно полезно; мои познания в их различии просто ужасны. (Я не хвастаюсь размерами своего сада, просто у меня это плохо получается.) За последние пару лет мы стали свидетелями всплеска подобного рода впечатляющих приложений.

В конечном счете потенциал искусственного интеллекта заключается в создании интеллектуальных агентов, которые выходят далеко за рамки классификации объектов. Представьте себе приложение с поддержкой ИИ, которое не только знает ваши растения, но и может за ними ухаживать. Такое приложение должно было бы выполнять следующие действия:

- работать в динамичных условиях (например, в условиях смены времен года);
- реагировать на изменения в окружающей среде (например, сильный шторм или вредители);
- выполнять последовательность действий (например, полив и подкормку растений);
- достигать долгосрочных целей (например, выстраивание приоритетов в отношении здоровья растений).

Наблюдая за своей средой, такой ИИ также научился бы усваивать возможные действия, которые он мог бы предпринимать, и со временем находить более оптимальные решения. Если вам кажется, что этот пример является искусственным или слишком далеким от реальности, нетрудно самостоятельно придумать примеры, которые соответствуют всем этим требованиям. Подумайте об управлении и оптимизации цепочки поставок, стратегическом пополнении запасов на складе с учетом меняющихся потребностей или организации этапов обработки на сборочной линии. Еще одним известным примером того, чего можно ожидать от ИИ, является знаменитый «Кофейный тест» Стивена Возняка (Stephen Wozniak): если вас пригласили в гости к другу или подруге, то вы можете пройти на кухню, найти кофеварку и все необходимые ингредиенты, выяснить, как сварить чашку кофе, и сесть за стол, чтобы им насладиться. Машина должна быть способна делать то же самое, за исключением того, что последняя часть, возможно, будет все-таки преувеличением. Какие еще примеры вы могли бы привести?

Все перечисленные технические требования можно сформулировать естественным образом в рамках подраздела машинного обучения, именуемого обучением с подкреплением<sup>1,2</sup>. На данный момент достаточно понять, что обучение с подкреплением всецело связано с агентами, взаимодействую-

---

<sup>1</sup> У нас пока нет роботов-садовников, и мы не знаем, какая парадигма ИИ приведет нас к этому. Обучение с подкреплением не обязательно является ответом; это просто парадигма, которая естественным образом вписывается в данное конкретное обсуждение целей ИИ.

<sup>2</sup> Обучение с подкреплением (reinforcement learning), в сущности, представляет собой гедонистическую самообучающуюся систему усвоения информации путем проб и ошибок, которая адаптирует свое поведение с целью максимизировать подкрепление со стороны среды. И тогда указанный выше термин должен переводиться как «подкрепляемое (стимулируемое средой) самообучение». – Прим. перев.

щими со своей средой путем наблюдения за ней и эмитирования действий. В данном обучении агенты оценивают свою среду, приписывая вознаграждение (например, каков уровень здоровья моего растения по линейной шкале). Термин «подкрепление» исходит из того факта, что агенты в идеале учатся придерживаться поведения, ведущего к хорошим результатам (высокому вознаграждению), и уклоняться от штрафующих ситуаций (низкого или отрицательного вознаграждения).

Взаимодействие агентов с их средой обычно моделируется путем создания его компьютерной симуляции (хотя иногда это не осуществимо). Итак, давайте разработаем пример такой симуляции, в которой агенты действуют в своей среде, чтобы дать вам представление о том, как это выглядит на практике.

## ПОСТАНОВКА ПРОСТОЙ ЗАДАЧИ О ЛАБИРИНТЕ

Как и в предыдущих главах, мы рекомендуем вам программировать материал этой главы и строить данное приложение по ходу изложения. Если вы не хотите этого делать, то можете просто отслеживать материал в блокноте Jupyter<sup>1</sup> этой главы.

Разрабатываемое нами приложение призвано дать вам общее представление. Оно структурировано следующим образом:

- реализовать простую двумерную игру в лабиринт, в которой один игрок может передвигаться в четырех главных направлениях;
- инициализировать лабиринт в виде решетки размером  $5 \times 5$ , рамками которой игрок ограничен. Одна из 25 ячеек сетки является «целью», которую игрок по имени *искатель* должен достичь;
- задействовать алгоритм обучения с подкреплением вместо жестко запрограммированного решения, чтобы *искатель* научился находить цель;
- выполнять симуляции лабиринта многократно, вознаграждая *исследователя* за то, что он нашел цель, и внимательно отслеживая решения *исследователя*, которые сработали и которые не сработали. Поскольку выполнение симуляций можно параллелизовать, и алгоритм обучения с подкреплением также можно тренировать параллельно, мы применим инструментарий Ray API, чтобы параллелизовать весь процесс.

Мы пока что не совсем готовы развернуть это приложение на реальном кластере Ray, состоящем из нескольких узлов, поэтому пока мы продолжим работать с локальными кластерами. Если вас интересуют темы инфраструктуры и вы хотите научиться настраивать кластеры Ray, то пройдите к главе 9. В любом случае проверьте, чтобы фреймворк Ray у вас был установлен с помощью команды `pip install ray`.

Давайте начнем с реализации 2D-лабиринта, который мы только что обрисовали. Идея состоит в том, чтобы реализовать на Python простую решетку,

---

<sup>1</sup> См. <https://oreil.ly/ceW4X>.

которая охватывает решетку размером  $5 \times 5$ , начинающуюся в  $(0, 0)$  и заканчивающуюся в  $(4, 4)$ , и правильно определить то, как игрок может перемещаться по решетке. Для этого сначала нужна абстракция перемещения по четырем сторонам света. Эти четыре действия, а именно перемещение вверх, вниз, влево и вправо, можно запрограммировать на Python как класс, который мы назовем `Discrete`. Абстракция перемещения в нескольких дискретных направленных действиях настолько полезна, что мы обобщим ее на  $n$  направлений, а не только на четыре. На случай если вы озадачены, тут нет никакой скоропалительности – нам действительно скоро понадобится общий класс `Discrete`:

```
import random

class Discrete:
    def __init__(self, num_actions: int):
        """ Пространство дискретных действий
            для num_actions. Discrete(4) можно
            использовать в качестве кодировки
            движения в одном из направлений
            сторон света
        """
        self.n = num_actions

    def sample(self):
        return random.randint(0, self.n - 1)

space = Discrete(4)
print(space.sample())
```

- ❶ Образец дискретного действия можно равномерно брать в диапазоне между 0 и  $n - 1$ .  
❷ Например, образец `Discrete(4)` будет давать 0, 1, 2 либо 3.

Взятие образцов из `Discrete(4)`, как в приведенном выше примере, будет случайно возвращать 0, 1, 2 либо 3. Интерпретация этих чисел зависит от нас, поэтому давайте предположим, что мы выбираем «вниз», «влево», «вверх» и «вправо» в таком порядке.

Теперь, когда мы знаем, как запрограммировать перемещение по лабиринту, давайте запрограммируем сам лабиринт, включая ячейку цели (`goal`) и позицию игрока-искателя (`seeker`), который пытается найти цель. Для этого мы собираемся реализовать класс Python под названием `Environment`. Он называется так потому, что лабиринт представляет собой среду, в которой «обитает» игрок. В целях упрощения задачи мы всегда будем ставить игрока в позицию  $(0, 0)$ , а цель – в позицию  $(4, 4)$ . Для того чтобы побудить игрока двигаться и находить цель, мы инициализируем среду (`Environment`) пространством действий (`action_space`) из `Discrete(4)`.

Нам нужно сформулировать последнюю часть информации для среды лабиринта: кодировку позиции игрока. Вся причина в том, что позже мы собираемся реализовать алгоритм, отслеживающий действия, которые привели к хорошим результатам после тех или иных позиций игрока. Закодировав позицию игрока как `Discrete(5*5)`, оно превращается в единое число,

с которым намного проще работать. На жаргоне обучения с подкреплением доступную игроку информацию об игре принято называть *наблюдением*. Таким образом, по аналогии с действиями, которые можно выполнять для нашего искателя, для него также можно определить пространство наблюдений (*observation\_space*). Вот реализация того, что мы только что изложили:

```
import os

class Environment:
    def __init__(self, *args, **kwargs):
        self.seeker, self.goal = (0, 0), (4, 4) ①
        self.info = {'seeker': self.seeker, 'goal': self.goal}

        self.action_space = Discrete(4) ②
        self.observation_space = Discrete(5*5) ③
```

- ① Искатель инициализируется в левом верхнем углу лабиринта, а цель – в его правом нижнем углу.
- ② Искатель может двигаться вниз, влево, вверх и вправо.
- ③ Он может находиться в общей сложности в 25 состояниях, по одному для каждой позиции в решетке.

Обратите внимание, что мы также определили информационную переменную (*info*), которая используется для печати информации о текущем состоянии лабиринта, например в целях отладки. Для того чтобы сыграть в настоящую игру «Найди цель» с точки зрения искателя, мы должны определить несколько вспомогательных методов. Очевидно, что игру следует считать «оконченной», когда искатель находит цель. Кроме того, мы должны вознаградить искателя за то, что он нашел цель. И когда игра окончена, мы должны иметь возможность вернуть ее в исходное состояние, чтобы сыграть снова. В довершение всего мы также определяем метод *get\_observation*, который возвращает закодированную позицию искателя (*seeker*). Продолжая реализацию класса *Environment*, это приводит к следующим ниже четырем методам:

```
def reset(self): ①
    """Обнулить позицию искателя и вернуть наблюдения"""
    self.seeker = (0, 0)
    return self.get_observation()

def get_observation(self):
    """Закодировать позицию искателя как целое число"""
    return 5 * self.seeker[0] + self.seeker[1] ②

def get_reward(self):
    """Вознаградить за отыскание цели"""
    return 1 if self.seeker == self.goal else 0 ③

def is_done(self):
    """Мы закончили, если нашли цель"""
    return self.seeker == self.goal ④
```

- ① Сбросить (*reset*) решетку в исходное состояние, чтобы начать новую игру.

- ② Конвертировать кортеж искателя в значение из пространства наблюдений (`observation_space`) среды.
- ③ Искатель получает вознаграждение только после достижения цели.
- ④ Игра окончена, если искатель находится у цели.

Последним важным методом, подлежащим реализации, является метод `step`. Представьте, что вы играете в игру в лабиринт и решаете сделать следующий шаг направо. Метод `step` выполнит это действие (а именно 3, кодировка «направо») и применит его к внутреннему состоянию игры. Отражая изменение, метод `step` затем вернет наблюдения искателя, его вознаграждение, информацию о законченности игры и значение `info` игры. Вот как работает метод `step`:

```
def step(self, action):  
    """Сделать шаг в указанном направлении и  
    вернуть всю доступную информацию"""  
    if action == 0: # переместиться вниз  
        self.seeker = (min(self.seeker[0] + 1, 4), self.seeker[1])  
    elif action == 1: # переместиться влево  
        self.seeker = (self.seeker[0], max(self.seeker[1] - 1, 0))  
    elif action == 2: # переместиться вверх  
        self.seeker = (max(self.seeker[0] - 1, 0), self.seeker[1])  
    elif action == 3: # переместиться вправо  
        self.seeker = (self.seeker[0], min(self.seeker[1] + 1, 4))  
    else:  
        raise ValueError("Недопустимое действие")  
  
    obs = self.get_observation()  
    rew = self.get_reward()  
    done = self.is_done()  
    return obs, rew, done, self.info ❶
```

- ❶ Возвращает наблюдение, вознаграждение, информацию о законченности игры и любую дополнительную информацию, которая может оказаться полезной после выполнения шага в указанном направлении.

Мы сказали, что метод `step` был последним важнейшим методом, но на самом деле мы хотим определить еще один вспомогательный метод, который чрезвычайно полезен для визуализации игры и помогает нам ее понять. Приведенный ниже метод `render` будет печатать текущее состояние игры в командную строку:

```
def render(self, *args, **kwargs):  
    """Отрисовывать среду, например  
    путем печати ее представления"""  
    os.system('cls' if os.name == 'nt' else 'clear') ❶  
  
    grid = [['|' for _ in range(5)] + ["|\n"] for _ in range(5)]  
    grid[self.goal[0]][self.goal[1]] = '|G'  
    grid[self.seeker[0]][self.seeker[1]] = '|S' ❷  
    print(''.join([''.join(grid_row) for grid_row in grid])) ❸
```

- ➊ Очистить экран.
- ➋ Нарисовать решетку и обозначить на ней цель как G, а искателя – как S.
- ➌ Затем сетка отрисовывается путем печати ее на экране.

Отлично! Теперь мы завершили реализацию класса `Environment`, который определяет игру в 2D-лабиринт. Мы можем пройти (`step`) эту игру, узнать, когда она окончится, и обнулить (`reset`) ее, чтобы начать снова. Кроме того, участник игры, искатель (`seeker`), может наблюдать за своей средой и получать вознаграждение за отыскание цели.

Давайте воспользуемся этой реализацией, чтобы сыграть в игру «Найди цель» для искателя, который просто предпринимает случайные действия. Это можно сделать, создав новую среду (`Environment`), беря образцы действий и применяя образцы к ней, а также отрисовывая среду до тех пор, пока игра не окончится:

```
import time

environment = Environment()

while not environment.is_done():
    random_action = environment.action_space.sample() ➊
    environment.step(random_action)
    time.sleep(0.1)
    environment.render() ➋
```

- ➊ Нашу среду можно тестировать, применяя взятые образцы действий до тех пор, пока игра не окончится.
- ➋ Визуализировать среду путем ее отрисовки после ожидания в течение десятой доли секунды (в противном случае исходный код будет выполняться слишком быстро, чтобы за ним можно было уследить).

Если выполнить этот фрагмент исходного кода на своем компьютере, то в конце концов вы увидите, что игра окончена и искатель нашел цель. Если вам не повезет, это может занять некоторое время.

Вы наверняка возразите, что это чрезвычайно простая задача и для ее решения нужно выполнить в общей сложности лишь восемь шагов, а именно пройти в произвольном порядке вправо и вниз по четыре раза каждый. Мы с вами и не спорим. Дело в том, что мы хотим решить эту задачу с помощью машинного обучения, чтобы позже взяться за гораздо более сложные задачи. В частности, мы хотим реализовать алгоритм, который сам определяет, как играть в игру, просто повторяя ее несколько раз: наблюдая за происходящим, решая, что делать дальше, и получая вознаграждение за свои действия.

Если хотите, сейчас самое подходящее время сделать игру сложнее. Этую игру можно модифицировать многими способами, при условии что интерфейс, который мы определили для класса `Environment`, останется без изменений. Вот несколько предложений:

- сделать ее в виде решетки размером  $10 \times 10$  или рандомизировать начальную позицию искателя;

- сделать внешние стенки решетки опасными. Всякий раз, когда вы к ним прикасаетесь, вы получаете вознаграждение в размере -100, то есть крутой штраф;
- ввести препятствия в решетке, через которые искатель не может пройти.

Если вы чувствуете себя по-настоящему предпримчивым, то также можете рандомизировать позицию цели. Это требует особой осторожности, так как в настоящее время с точки зрения метода `get_observation` у искателя нет информации о местоположении цели. Возможно, стоит вернуться к выполнению этого последнего упражнения, после того как вы закончите читать эту главу.

## РАЗРАБОТКА СИМУЛЯЦИИ

Что требуется для решения задачи «обучения» искателя хорошо играть в игру при наличии реализованного класса `Enviroment`? Как он будет последовательно находить цель за минимальное число необходимых восьми шагов? Мы снабдили среду лабиринта информацией о вознаграждении, чтобы искатель мог использовать этот сигнал для усвоения игры. В обучении с подкреплением вы постоянно играете в игры и учитесь на опыте, который приобретает по ходу. Игрока в игре часто называют *агентом*, который предпринимает *действия* в окружающей среде, наблюдает за ее *состоянием* и получает *вознаграждение*<sup>1</sup>. Чем лучше агент учится, тем лучше он интерпретирует текущее состояние игры (наблюдения) и находит действия, которые приводят к более вознаграждающим результатам.

Независимо от того, какой алгоритм обучения с подкреплением вы хотите использовать, у вас должен быть способ многократного симулирования игры для сбора данных об опыте. По этой причине мы собираемся реализовать простой класс `Simulation`.

Теперь нам нужно обратиться к еще одной полезной абстракции – политике (*Policy*), то есть способу детализации действий. Прямо сейчас, для того чтобы играть в игру, нам нужно делать всего одну вещь – подбирать искателю случайные образцы его действий. Особенность политики состоит в том, что она позволяет получать более оптимальные действия в соответствии с текущим состоянием игры. Собственно говоря, мы определяем политику как класс с методом `get_action`, который принимает состояние игры и возвращает действие.

Напомним, что в нашей игре у искателя есть в общей сложности 25 возможных состояний в решетке, и он может выполнять четыре действия. Простая идея состояла бы в том, чтобы смотреть на пары состояний и действий

---

<sup>1</sup> Как мы увидим в главе 4, обучение с подкреплением также можно выполнять в многопользовательских играх. Превращение среды лабиринта в так называемую многоагентную среду, в которой несколько искателей соревнуются за достижение цели, является интересным упражнением.

и назначать паре высокое значение, если выполнение этого действия в этом состоянии приводит к высокому вознаграждению, и низкое значение в противном случае. Например, исходя из интуитивного понимания игры, должно быть ясно, что двигаться вниз или вправо – всегда хорошая идея, в то время как двигаться влево или вверх – нет. Затем нужно создать просмотровую таблицу размером  $25 \times 4$  для всех возможных пар состояние–действие и сохранить ее в классе `Policy`. Тогда мы могли бы просто просить политику возвращать наивысшее значение любого действия с учетом конкретного состояния. Разумеется, часть, касающаяся реализации алгоритма, который находит хорошие значения для пар состояние–действие, является самой трудной. Сначала давайте реализуем описанную выше идею политики, а о подходящем алгоритме позаботимся позже:

```
import numpy as np

class Policy:
    def __init__(self, env):
        """Политика подсказывает действия,
        основываясь на текущем состоянии.
        Мы делаем это, отслеживая значение
        каждой пары состояние–действие."""
        self.state_action_table = [
            [0 for _ in range(env.action_space.n)]
            for _ in range(env.observation_space.n) ①
        ]
        self.action_space = env.action_space

    def get_action(self, state, explore=True, epsilon=0.1): ②
        """Разведывать случайным образом либо 'эксплуатировать'
        наилучшее значение, имеющееся в настоящее время"""
        if explore and random.uniform(0, 1) < epsilon: ③
            return self.action_space.sample()
        return np.argmax(self.state_action_table[state]) ④
```

- ① Определить вложенный список значений для каждой пары состояние–действие, инициализированный нулем.
- ② Разведывать (`explore`) случайные действия по требованию, чтобы мы не застревали в субоптимальном поведении.
- ③ Внести параметр `explore` в метод `get_action`, потому что мы, возможно, захотим разведывать действия в игре случайным образом. По умолчанию это происходит в 10 % случаев.
- ④ Вернуть действие с наибольшим значением в просмотровой таблице с учетом текущего состояния.

Мы включили в определение политики небольшие детали реализации, которые, возможно, будут немного сбивать с толку. Метод `get_action` имеет параметр `explore`. Без этого, если вы усвоите крайне плохую политику (например, ту, которая всегда требует, чтобы вы двигались влево), у вас не будет никаких шансов когда-либо находить более оптимальные решения. Другими словами, иногда нужно разведывать новые способы, а не «эксплуатировать»

свое текущее понимание игры<sup>1</sup>. Как указывалось ранее, мы не затрагивали вопрос о том, как учиться улучшать значения в таблице state\_action\_table политики. На данный момент просто следует иметь в виду, что политика дает действия, которым мы хотим следовать при симулировании игры в лабиринт.

Переходя к классу `Simulation`, о котором мы говорили ранее, симуляция должна брать среду (`Environment`) и вычислять действия в соответствии с указанной политикой (`Policy`) до тех пор, пока не будет достигнута цель и игра не окончится. Данные, которые мы наблюдаем при такого рода разыгрывании, или «розыгрыше»<sup>2</sup>, полной игры, мы называем приобретаемым *опытом*. Соответственно, в нашем классе `Simulation` есть метод разыгрыша (`rollout`), который вычисляет значения опыта (`experiences`) в полной игре и возвращает их. Вот как выглядит реализация класса `Simulation`:

```
class Simulation(object):
    def __init__(self, env):
        """Симулирует разыгрыши среды с учетом используемой политики"""
        self.env = env

    def rollout(self, policy, render=False, explore=True, epsilon=0.1): ①
        """Возвращает значения опыта для разыгрыша политики"""
        experiences = []
        state = self.env.reset() ②
        done = False
        while not done:
            action = policy.get_action(state, explore, epsilon) ③
            next_state, reward, done, info = self.env.step(action) ④
            experiences.append([state, action, reward, next_state]) ⑤
            state = next_state
            if render: ⑥
                time.sleep(0.05)
                self.env.render()
        return experiences
```

- ① «Разыграть» игру, следуя действиям политики (`policy`), и опционально отрисовывать симуляцию.

---

<sup>1</sup> В терминологическом плане глаголы «эксплуатировать» (`exploit`) и «разведывать» (`explore`) выполняют важную познавательную роль и используются в противопоставлении. Разведка – это поиск новых идей или новых стратегий. Эксплуатация – это использование существующих идей и стратегий, которые оказались успешными в прошлом. В частности, об этом замечательно пишет Дэн Саймон в своей культовой книге «Алгоритмы эволюционной оптимизации» (ДМК): «...Правильный баланс разведки и эксплуатации зависит от того, насколько регулярна наша среда. Если она быстро меняется, то наши знания быстро устаревают, и мы во многом не можем полагаться на эксплуатацию. Однако если наша среда весьма стабильна, то наше знание является надежным, и в разведывании слишком большого числа новых идей, возможно, не будет никакого смысла». – Прим. перев.

<sup>2</sup> Термин «розыгрыш» заимствован из компьютерной игры в нарды, где он обозначает расчет ценности позиции путем разыгрывания (`playout`), или «раскатки» (`rollout`), позиции много раз до конца игры с помощью случайно генерируемых последовательностей бросков костей, где ходы обоих игроков выполняются в соответствии с некоторой фиксированной политикой. – Прим. перев.

- ② Для верности сбрасывать среду в исходное состояние перед каждым розыгрышем.
- ③ Принятая политика (*policy*) определяет действия, которые мы предпринимаем. Параметры *explore* и *epsilon* передаются далее в сквозном порядке.
- ④ Пройти по среде, применяя действие (*action*) политики.
- ⑤ Определить опыт как квадруплет (*state, action, reward, next\_state*).
- ⑥ Опционально отрисовывать среду на каждом шаге.

Обратите внимание, что каждая запись об опыте, который мы собираем при розыгрыше, состоит из четырех значений: текущее состояние, предпринятое действие, полученное вознаграждение и следующее состояние. Вскоре мы реализуем алгоритм, который будет использовать эти значения опыта, чтобы на них учиться. Другие алгоритмы могут использовать другие значения опыта, но нам для продолжения нужны именно эти.

Теперь у нас есть политика, которая пока что ничему не научилась, но мы уже можем протестировать ее интерфейс, чтобы убедиться, что она работает. Давайте это попробуем, инициализировав объект *Simulation*, вызвав его метод *rollout* на не очень-то умной политике, а затем распечатаем ее таблицу *state\_action\_table*:

```
untrained_policy = Policy(environment)
sim = Simulation(environment)

exp = sim.rollout(untrained_policy, render=True, epsilon=1.0) ①
for row in untrained_policy.state_action_table:
    print(row) ②
```

- ① Разыграть одну полную игру с «ненатренированной» политикой, которую мы отрисовываем.
- ② Все значения состояния–действия в настоящее время равны нулю.

Если вам кажется, что мы не сильно продвинулись по сравнению с предыдущим разделом, то уверяем вас, что в следующем далее разделе все сойдется. Подготовительная работа по постановке симуляции (*Simulation*) и политики (*Policy*) была необходима для правильной постановки задачи. Теперь единственное, что осталось, – это разработать разумный способ обновления внутреннего состояния политики, основываясь на собранных нами значениях опыта, чтобы она действительно учились играть в лабиринт.

## ТРЕНИРОВКА МОДЕЛИ ОБУЧЕНИЯ С ПОДКРЕПЛЕНИЕМ

Представьте, что у нас есть набор значений опыта, которые мы собрали из пары игр. Каким был бы разумный способ обновлять значения в таблице *state\_action\_table* нашей политики? Вот одна идея. Допустим, вы сидите в позиции (3,5) и решили двинуться направо, что ставит вас в позицию (4,5), всего в одном шаге от цели. Очевидно, что тогда вы могли бы просто пойти направо и получить вознаграждение в размере 1 балла. Это должно означать, что текущее состояние, в котором вы находитесь, в сочетании с действием двигаться «направо» должно иметь высокую ценность. Другими словами,

значение этой конкретной пары состояния–действие должно быть высоким. Напротив, перемещение влево в той же ситуации ни к чему не приводит, и значение соответствующей пары состояния–действие должно быть низким.

В более общем плане, допустим, вы были в определенном состоянии (`state`), вы решили предпринять действие (`action`), ведущее к вознаграждению, и затем вы попадаете в следующее состояние (`next_state`). Напомним, что именно так мы и определили опыт. С помощью таблицы `policy.state_action_table` мы можем заглянуть немного вперед и посмотреть, не можем ли мы приобрести что-либо от действий, предпринятых из следующего состояния (`next_state`). То есть мы можем вычислить следующий максимум:

```
next_max = np.max(policy.state_action_table[next_state])
```

Каким образом сравнить значение об этом значении с текущим значением состояния–действия, которое равно `value = policy.state_action_table[state][action]`? Это можно решить разными способами, но, очевидно, мы не можем полностью отказаться от текущего значения (`value`) и слишком сильно доверять следующему максимуму (`next_max`). В конце концов, мы здесь используем всего лишь единичную порцию опыта. Поэтому, в качестве первого приближения, почему бы просто не вычислить взвешенную сумму старого и ожидаемого значений и остановиться на `new_value = 0.9 * value + 0.1 * next_max`? Здесь значения 0.9 и 0.1 были выбраны несколько произвольно; единственны важные части таковы: первое значение достаточно велико, чтобы отражать наше предпочтение оставить старое значение, и оба веса в сумме равны 1. Эта формула является неплохой отправной точкой, но проблема в том, что мы совершенно не учитываем важную информацию, которую получаем от вознаграждения (`reward`). На самом деле мы должны больше доверять текущему значению вознаграждения, чем проецируемому значению `next_max`, поэтому неплохо немного дисконтировать последнее, скажем на 10 %. Тогда обновление значения состояния–действия выглядело бы следующим образом:

```
new_value = 0.9 * value + 0.1 * (reward + 0.9 * next_max)
```

В зависимости от вашего уровня опыта в такого рода рассуждениях последние несколько параграфов, возможно, будет трудно переварить. Если вы поняли объяснения до этого места, то остальная часть данной главы, скорее всего, дастся вам легко. В математическом плане это была последняя (и единственная) сложная часть данного примера. Если вы раньше с подкрепляемым обучением работали, то наверняка заметили, что мы имеем дело с реализацией так называемого алгоритма Q-обучения<sup>1</sup>. Он называется так потому, что таблица состояний–действий может быть описана как функция `Q(state, action)`, которая возвращает значения этих пар.

Мы почти на месте, поэтому давайте формализуем процедуру с помощью функции `update_policy` для политики и собранных значений опыта:

---

<sup>1</sup> Англ. Q-Learning. – Прим. перев.

```
def update_policy(policy, experiences, weight=0.1, discount_factor=0.9):
    """Обновляет данную политику списком значений
    (state, action, reward, state) опыта"""
    for state, action, reward, next_state in experiences: ①
        next_max = np.max(policy.state_action_table[next_state]) ②
        value = policy.state_action_table[state][action] ③
        new_value = (1 - weight) * value + weight * \
                    (reward + discount_factor * next_max) ④
        policy.state_action_table[state][action] = new_value ⑤
```

- ① Прокрутить все значения опыта по порядку.
- ② Выбрать максимальное значение из всех возможных действий в следующем состоянии.
- ③ Извлечь текущее значение состояния–действия.
- ④ Новое значение – это взвешенная сумма старого значения и ожидаемого значения, которое представляет собой сумму текущего вознаграждения и дисконтированного следующего максимума (next\_max).
- ⑤ После обновления установить новое значение в таблице state\_action\_table.

Наличие этой функции в настоящее время упрощает тренировку политики, чтобы предпринимать более оптимальные решения. Мы можем использовать следующую ниже процедуру:

1. Инициализировать политику и симуляцию.
2. Выполнить симуляцию много раз, скажем в общей сложности 10 000 проходов.
3. По каждой игре сначала собирать значения опыта, выполняя розыгрыш (rollout) игры.
4. Затем обновлять политику, вызывая функцию `update_policy` для собранных данных.

Вот и все! Эта процедура реализована в следующей ниже функции `train_policy`:

```
def train_policy(env, num_episodes=10000, weight=0.1, discount_factor=0.9):
    """Тренировка политики путем ее обновления
    с учетом опыта розыгрышей"""
    policy = Policy(env)
    sim = Simulation(env)
    for _ in range(num_episodes):
        experiences = sim.rollout(policy) ①
        update_policy(policy, experiences, weight, discount_factor) ②

    return policy
```

```
trained_policy = train_policy(environment) ③
```

- ① Собрать значения опыта по каждой игре.
- ② Обновить политику с учетом этих значений опыта.
- ③ Наконец, натренировать и вернуть политику в отношении среды (`environment`), доставшейся из прошлого.

Обратите внимание, что в литературе по обучению с подкреплением полное прохождение игры в лабиринт заумно называется *эпизодом*. Вот почему в функции `train_policy` мы вызываем аргумент `num_episodes`, а не `num_games`.

## Q-обучение

Только что реализованный нами алгоритм Q-обучения часто является первым алгоритмом, преподаваемым на занятиях по обучению с подкреплением, главным образом потому, что в нем относительно легко разобраться. Вы собираете и сводите в таблицу данные об опыте, которые показывают, насколько хорошо работают пары состояние–действие, а затем обновляете таблицу в соответствии с правилом обновления алгоритма Q-обучения.

В задачах обучения с подкреплением, которые содержат огромное число состояний или действий, Q-таблица может становиться чрезмерно большой. Тогда алгоритм становится неэффективным, поскольку потребовалось бы слишком много времени, чтобы собирать достаточно данных об опыте для всех (релевантных) пар состояние–действие.

Один из способов решения этой проблемы состоит в аппроксимации Q-таблицы с использованием нейронной сети. Под ней мы подразумеваем, что для усвоения функции, которая соотносит состояния с действиями, может задействоваться глубокая нейронная сеть. Данный подход называется глубоким Q-обучением, а применяемые для его усвоения сети называются глубокими Q-сетями<sup>1</sup>. Начиная с главы 4 этой книги для решения задач обучения с подкреплением мы будем использовать исключительно глубокое обучение.

Теперь, когда у нас есть натренированная политика, давайте посмотрим, насколько хорошо она работает. В этой главе мы уже дважды выполняли случайные политики, просто чтобы получить представление о том, насколько хорошо они работают в задаче о лабиринте. Но теперь давайте оценим натренированную политику надлежащим образом на нескольких играх и посмотрим, как она работает в среднем. В частности, мы выполним симуляцию в течение пары эпизодов и подсчитаем число шагов, которое потребовалось для достижения цели в каждом эпизоде. Итак, реализуем функцию `evaluate_policy`, которая выполняет именно это:

```
def evaluate_policy(env, policy, num_episodes=10):
    """Оценить натренированную политику
    посредством розыгрышей"""
    simulation = Simulation(env)
    steps = 0

    for _ in range(num_episodes):
        experiences = simulation.rollout(policy, render=True,
                                           explore=False) ①
        steps += len(experiences) ②

    print(f"В среднем {steps / num_episodes} шагов "
          f"всего за {num_episodes} эпизодов.")

    return steps / num_episodes

evaluate_policy(environment, trained_policy)
```

- ① На этот раз установить параметр `explore` равным `False`, чтобы в полной мере эксплуатировать знания из натренированной политики.

<sup>1</sup> Англ. Deep Q-Networks (DQN).

- ② Длина значений опыта (`experiences`) – это число шагов, которые мы предприняли, чтобы закончить игру.

Помимо того что натренированная политика решает задачу с лабиринтом 10 раз подряд, как мы и надеялись, вы также должны увидеть следующее сообщение:

В среднем 8.0 шагов всего за 10 эпизодов.

Другими словами, натренированная политика способна находить оптимальные для игры в лабиринт решения. Это означает, что вы успешно реализовали свой первый алгоритм обучения с подкреплением с чистого листа!

Как вы думаете, с учетом того понимания, которое у вас сложилось, будет ли размещение искателя в произвольно отбираемых исходных позициях и последующий запуск этой функции оценивания работать по-прежнему? Почему бы вам не пойти дальше и не внести необходимые для этого изменения?

Следует задать себе еще один интересный вопрос: какие допущения были заложены в использованный нами алгоритм? Например, очевидно, что обязательным условием для алгоритма является то, что все пары состояния–действие могут быть сведены в таблицу. Как вы думаете, работало ли это так же хорошо, если бы у нас были миллионы состояний и тысячи действий?

## РАЗРАБОТКА РАСПРЕДЕЛЕННОГО ПРИЛОЖЕНИЯ RAY

Мы надеемся, что вам понравился приведенный выше пример, но вам, возможно, интересно, как проделанная нами работа относится к фреймворку Ray (отличный вопрос). Как вы вскоре увидите, все, что нужно, чтобы превратить эксперимент по обучению с подкреплением в распределенное приложение Ray, сводится к написанию трех коротких фрагментов исходного кода. И это то, что мы собираемся сделать:

1. Сделать класс `Simulation` актором Ray, используя всего несколько строк исходного кода.
2. Определить параллельную версию тренировки политики (`train_policy`), структурно похожую на ее оригинал. Для простоты мы будем параллизовывать только розыгрыши, а не обновления политики.
3. Натренировать и оценить политику, как и раньше, но используя `train_policy_parallel`.

Давайте приступим к первому шагу этого плана, реализовав актора Ray под названием `SimulationActor`:

```
import ray

ray.init()

@ray.remote
class SimulationActor(Simulation):
    """Актор Ray для класса Simulation"""

    ❶
```

```
def __init__(self):
    env = Environment()
    super().__init__(env)
```

- ❶ Этот актор Ray элементарным способом упрощает наш класс `Simulation`.

Учитывая основные навыки применения инструментария Ray Core, которые вы развили в главе 2, у вас не должно возникнуть проблем с чтением этого исходного кода. Возможно, потребуется некоторая практика, чтобы уметь писать его самостоятельно, но концептуально с этим примером вы должны быть на высоте.

Двигаясь дальше, давайте определим функцию `train_policy_parallel`, которая распределяет эту рабочую нагрузку обучения с подкреплением по вашему локальному кластеру Ray. Для этого мы создаем политику на драйвере и в общей сложности четыре экземпляра класса `SimulationActor`, которые будем использовать для распределенных розыгрышей. Затем помещаем политику в хранилище объектов с помощью `ray.put` и передаем ее дистанционным вызовам `rollout` в качестве аргумента, чтобы собирать значения опыта в течение указанного числа эпизодов тренировки. Потом мы применяем `ray.wait`, чтобы получить законченные розыгрыши (и учтываем тот факт, что некоторые розыгрыши могут завершаться раньше, чем другие) и обновляем политику собранными значениями опыта. Наконец, мы возвращаем натренированную политику:

```
def train_policy_parallel(env, num_episodes=1000, num_simulations=4):
    """Функция параллельной тренировки политики"""
    policy = Policy(env) ❶
    simulations = [SimulationActor.remote() ❷
                  for _ in range(num_simulations)] ❸

    policy_ref = ray.put(policy) ❹
    for _ in range(num_episodes):
        experiences = [sim.rollout.remote(policy_ref)
                        for sim in simulations] ❺

        while len(experiences) > 0:
            finished, experiences = ray.wait(experiences) ❻
            for xp in ray.get(finished):
                update_policy(policy, xp)

    return policy
```

- ❶ Инициализировать политику для данной среды.  
❷ Вместо одной симуляции создать четырех акторов симуляции.  
❸ Поместить политику в хранилище объектов.  
❹ Для каждого из 1000 эпизодов параллельно собирать данные об опыте, используя актеров симуляции.  
❺ Готовые розыгрыши можно извлечь из хранилища объектов и использовать для обновления политики.

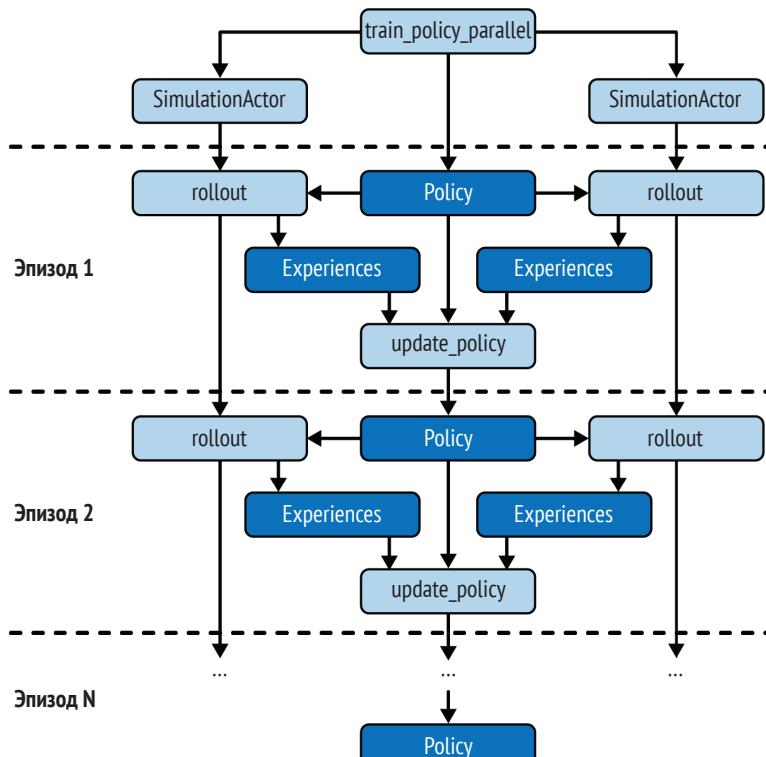
Это позволяет сделать последний шаг и выполнить процедуру тренировки в параллельном режиме, а затем оценить результат, как и раньше:

```
parallel_policy = train_policy_parallel(environment)
evaluate_policy(environment, parallel_policy)
```

Результат этих двух строк будет таким же, как и раньше, когда мы выполняли серийную версию тренировки алгоритма обучения с подкреплением для лабиринта. Мы надеемся, вы оцените по достоинству, что функция `train_policy_parallel` имеет ту же высокоуровневую структуру, что и функция `train_policy`. Неплохим упражнением будет сравнение двух вариантов построчно.

По сути, для параллелизации процесса тренировки потребовалось лишь подходящим образом применить декоратор `@ray.remote` на классе, а затем использовать правильные вызовы `remote`. Конечно, нужен опыт, чтобы все сделать правильно. Но обратите внимание, как мало времени мы потратили на размышления о распределенных вычислениях и как много времени мы смогли потратить на сам исходный код приложения. Нам не нужно было внедрять совершенно новую парадигму программирования, и мы смогли подойти к задаче наиболее естественным образом. В конечном счете это то, чего вы хотите, и фреймворк Ray отлично умеет предоставлять такую гибкость.

Подводя итог, давайте бегло взглянем на граф исполнения приложения Ray, которое мы только что разработали. На рис. 3.1 этот график заданий представлен в сжатой форме.



**Рис. 3.1** ♦ Параллельная тренировка политики с помощью фреймворка Ray в рамках алгоритма обучения с подкреплением

- ✓ Текущий пример этой главы является реализацией псевдокода, использованного для иллюстрации гибкости фреймворка Ray в первоначальной статье<sup>1</sup> его создателей. В той статье приведен рисунок, аналогичный рис. 3.1, и ее стоит прочитать для понимания контекста.

## Резюмирование терминологии обучения с подкреплением

Прежде чем завершить эту главу, давайте обсудим концепции, с которыми мы столкнулись в примере с лабиринтом, в более широком контексте. Благодаря этому вы сможете подготовиться к более сложным условиям обучения с подкреплением в следующей главе и увидите места, где мы немного упростили работу в текущем примере этой главы. Если вы достаточно хорошо знаете обучение с подкреплением, то можете этот раздел пропустить.

Каждая задача обучения с подкреплением начинается с формулирования *среды*, которая описывает динамику «игры», в которую вы хотите играть. В среде размещается игрок, или *агент*, который взаимодействует со своей средой через простой интерфейс. Агент может запрашивать информацию из среды, а именно о своем текущем *состоянии* в среде, *вознаграждении*, которое он получил в этом состоянии, и о том, *завершена* игра или нет. Наблюдая за состояниями и вознаграждениями, агент может научиться принимать решения на основе полученной информации. В частности, агент будет эмитировать *действие*, которое может исполняться средой путем выполнения следующего шага (*step*).

Механизм, используемый агентом для продуцирования действий в данном состоянии, называется *политикой*, и мы иногда говорим, что агент следует данной политике. Имея политику, можно симулировать, или *разыгрывать*, несколько шагов или целую игру с помощью этой политики. Во время разыгрыша можно собирать *опыт*, в котором аккумулируется информация о текущем состоянии и вознаграждении, следующем действии и результирующем состоянии. Вся последовательность шагов от начала до конца называется *эпизодом*, и среду можно сбрасывать (*reset*) в исходное состояние, чтобы начинать новый эпизод.

Политика, которую мы использовали в этой главе, была основана на простой идее сведения в таблицу значений состояний–действий (так называемых *Q-значений*<sup>2</sup>), а алгоритм, используемый для обновления политики на основе опыта, собранного во время разыгрышей, называется *Q-обучением*. В более общем плане реализованную нами таблицу состояний–действий можно

---

<sup>1</sup> См. <https://oreil.ly/ZKZFY>.

<sup>2</sup> Q-значение (Q-value) – это мера общего ожидаемого вознаграждения, исходя из допущения, что агент находится в состоянии  $s$  и выполняет действие  $a$ , а затем продолжает играть до конца эпизода, следуя некоторой политике. – Прим. перев.

трактовать как модель, используемую политикой. В следующей главе вы увидите примеры более сложных моделей, таких как нейронная сеть, служащая для усвоения значений состояний–действий. Политика может принимать решение *эксплуатировать* то, что она узнала о среде, выбирая наилучшее доступное значение своей модели, либо *разведывать* среду, выбирая случайное действие.

Многие представленные здесь базовые концепции применимы к любой задаче обучения с подкреплением, но мы приняли несколько упрощающих допущений. Например, в среде могло бы действовать *несколько агентов* (представьте, что несколько искателей соревнуются за достижение цели первыми), и в следующей главе мы рассмотрим так называемые многоагентные среды и многоагентное обучение с подкреплением. Кроме того, мы допустили, что *пространство действий* агента было *дискретным* и, стало быть, агент мог предпринимать только фиксированный набор действий. Конечно же, у вас также могут быть *непрерывные* пространства действий, и пример с шестом на тележке из главы 1 является одним из таких примеров. Пространства действий могут становиться более сложными, в особенности при наличии нескольких агентов, и вам могут понадобиться кортежи действий или даже необходимость вкладывать их соответствующим образом друг в друга. *Пространство наблюдений*, которое мы рассматривали в игре в лабиринт, также было довольно простым и моделировалось как дискретное множество состояний. Можно легко представить, что сложные агенты, такие как взаимодействующие со своей средой роботы, могли бы работать с изображениями или видеоданными в качестве наблюдений, что также потребовало бы более сложного пространства наблюдений.

Еще одно принятное нами важное допущение заключается в том, что среда *детерминирована*, то есть когда агент решает предпринять какое-либо действие, результирующее состояние всегда будет отражать это решение. В обычных средах это не так, и в среде могут присутствовать элементы случайности. Например, в игре в лабиринт мы могли бы реализовать подбрасывание монеты, и всякий раз, когда выпадала бы решка, агента подталкивали бы в случайном направлении. В таком сценарии мы не смогли бы планировать заранее, как это делалось в данной главе, потому что каждый раз действия не приводили бы детерминированно к одному и тому же следующему состоянию. Для того чтобы отразить это вероятностное поведение, мы должны учитывать вероятности *перехода из состояния в состояние* в наших экспериментах по обучению с подкреплением в целом.

Последнее упрощающее допущение, о котором здесь следует поговорить, заключается в том, что мы трактовали среду и ее динамику как игру, которую можно идеально просимулировать. Но дело в том, что некоторые физические системы невозможно симулировать в точности. В этом случае вы все равно могли бы взаимодействовать с такой физической средой через интерфейс, подобный тому, который мы определили в классе `Environment`, но это повлекло бы за собой некоторые непроизводительные коммуникационные издержки. На практике *рассуждение* о задачах обучения с подкреплением так, как будто они являются играми, очень мало отнимает от опыта.

## Резюме

Напомним, что мы реализовали простую задачу о лабиринте на простом Python, а затем решили задачу отыскания цели в этом лабиринте, используя простой алгоритм обучения с подкреплением. Затем мы взяли это решение и портировали его в распределенное приложение Ray примерно в 25 строках исходного кода. Мы сделали это без необходимости заранее планировать то, как работать с фреймворком Ray, – мы просто использовали инструментарий Ray API, чтобы параллелизовать наш исходный код Python. Этот пример демонстрирует то, как фреймворк Ray совершенно не мешает и позволяет сосредоточиваться на исходном коде приложения. Он также показывает, как эффективно реализовывать и распределять с помощью Ray конкретно-прикладные рабочие нагрузки, задействующие передовые технологии, такие как обучение с подкреплением.

В главе 4 вы обопретесь на то, что узнали здесь, и увидите, как задача о лабиринте легко решается непосредственно с помощью высокоуровневой библиотеки Ray RLlib.

# Глава 4

---

## Обучение с подкреплением с использованием библиотеки Ray RLlib

В главе 3 вы разработали среду обучения с подкреплением, симуляцию для розыгрыша нескольких раундов игры, алгоритм обучения с подкреплением и исходный код параллелизации тренировки алгоритма – все это полностью с чистого листа. Совсем неплохо уметь это делать, но на практике, занимаясь тренировкой алгоритмов обучения с подкреплением, единственное, что вы на самом деле хотите делать, – это первую часть, а именно описывать вашу конкретно-прикладную среду, «игру», в которую вы хотите играть<sup>1</sup>. Большая часть ваших усилий будет направлена на выбор правильного алгоритма, его наладку, отыскание для задачи наилучших параметров и, как правило, сосредоточение внимания на тренировке политики, демонстрирующей хорошую результативность.

Ray RLlib – это библиотека производственного уровня, служащая для разработки алгоритмов обучения с подкреплением в крупном масштабе. В главе 1 вы уже увидели первый пример применения библиотеки RLlib, но в этой главе мы рассмотрим ее гораздо подробнее. Самое замечательное в библиотеке RLlib то, что это зрелая библиотека для разработчиков, которая поставляется с хорошими абстракциями для работы. Как вы увидите, многие из этих абстракций вы уже знаете из предыдущей главы.

Мы начнем с того, что дадим вам краткий обзор возможностей библиотеки RLlib. Затем бегло освежим в памяти игру в лабиринт из главы 3 и покажем, как с ней обращаться с помощью интерфейса командной строки библиотеки

---

<sup>1</sup> Мы используем простую игру, чтобы проиллюстрировать процесс обучения с подкреплением. Существует целый ряд интересных производственных приложений обучения с подкреплением, которые не являются играми.

RLlib (CLI) и Python'овского API указанной библиотеки в нескольких строках исходного кода. Вы увидите, что начинать работу с библиотекой RLlib очень легко, после чего познакомитесь с ее ключевыми концепциями, такими как среды и алгоритмы библиотеки RLlib.

Мы также более подробно рассмотрим некоторые продвинутые темы обучения с подкреплением, которые чрезвычайно полезны на практике, но часто не поддерживаются должным образом в других библиотеках обучения с подкреплением. Например, вы узнаете, как создавать учебную программу для ваших агентов обучения с подкреплением, чтобы они могли усваивать простые сценарии, прежде чем переходить к более сложным. Вы также увидите, как библиотека RLlib оперирует несколькими агентами в одной среде и как задействовать данные об опыте, собранные за пределами вашего текущего приложения, чтобы повышать результативность вашего агента.

## КРАТКИЙ ОБЗОР БИБЛИОТЕКИ RLlib

Прежде чем углубиться в какие-либо примеры, давайте коротко обсудим библиотеку RLlib – что она из себя представляет и что она может делать. Библиотека RLlib, являясь частью экосистемы фреймворка Ray, наследует все преимущества фреймворка Ray по производительности и масштабируемости. В частности, библиотека RLlib по умолчанию распределена, поэтому вы можете масштабировать свою тренировку алгоритма обучения с подкреплением на столько узлов, на сколько захотите.

Еще одной выгодой от того, что библиотека RLlib построена поверх фреймворка Ray, является то, что она тесно интегрируется с другими библиотеками фреймворка. Например, как мы увидим в главе 5, гиперпараметры любого алгоритма библиотеки RLlib можно настраивать с помощью библиотеки Ray Tune. Кроме того, можно бесшовно развертывать свои модели RLlib с помощью библиотеки Ray Serve<sup>1</sup>.

Чрезвычайно полезно то, что на момент написания этой книги библиотека RLlib работает с обоими преобладающими фреймворками глубокого обучения: PyTorch и TensorFlow. Любой из них можно использовать в качестве своего бэкенда и легко переключаться между ними, нередко меняя всего одну строку исходного кода. Это огромное преимущество, поскольку компании часто привязаны к своему опорному фреймворку глубокого обучения и не могут позволить себе переключаться на другую систему и переписывать свой исходный код.

К заслугам библиотеки RLlib также относится опыт решения реальных задач с ее помощью – она является зрелой библиотекой, используемой многими компаниями, позволяющей им доводить свои рабочие нагрузки обучения с подкреплением до производства. API библиотеки RLlib привлекает многих

---

<sup>1</sup> В данной книге эта интеграция не рассматривается, но о развертывании моделей библиотеки RLlib можно узнать подробнее в практическом руководстве «Подача моделей библиотеки RLlib в качестве служб» (Serving RLlib Models, <https://oreil.ly/vsz0A>) в документации фреймворка Ray.

инженеров, поскольку предлагает правильный уровень абстракции для многих приложений, оставаясь при этом достаточно гибким для расширения.

Помимо этих более общих выгод, библиотека RLlib обладает множеством специфичных для обучения с подкреплением функциональных возможностей, которые мы рассмотрим в этой главе. На самом деле библиотека RLlib настолько многофункциональна, что заслуживала бы отдельной книги, а это значит, что здесь мы сможем затронуть лишь некоторые ее аспекты. Например, в библиотеке RLlib есть богатая коллекция продвинутых алгоритмов обучения с подкреплением на выбор. В этой главе мы сосредоточимся на нескольких избранных, но вы можете отслеживать растущий список опций на странице алгоритмов библиотеки RLlib<sup>1</sup>. Библиотека RLlib также имеет множество опций для описания сред обучения с подкреплением и отличается большой гибкостью в обращении с ними во время тренировки; обзор сред библиотеки RLlib приведен в документации<sup>2</sup>.

## НАЧАЛО РАБОТЫ С БИБЛИОТЕКОЙ RLlib

Прежде чем использовать библиотеку RLlib, сперва следует проверить, что она была установлена на ваш компьютер:

```
pip install "ray[rllib]==2.2.0"
```

-  Если вам не хочется набирать исходный код на клавиатуре самостоятельно, то рекомендуем ознакомиться с прилагаемым к этой главе блокнотом Jupyter<sup>3</sup>.

Каждая задача обучения с подкреплением начинается с обследования интересующей среды. В главе 1 мы рассмотрели классическую задачу балансировки шеста на тележке. Напомним, что мы не реализовывали среду с шестом на тележке; она поставляется вместе с библиотекой RLlib.

В противоположность этому в главе 3 мы реализовали простую игру в лабиринт самостоятельно. Проблема с той реализацией заключается в том, что мы не можем использовать ее напрямую с библиотекой RLlib либо любой другой библиотекой обучения с подкреплением, если уж на то пошло. Причина в том, что в обучении с подкреплением есть общепринятые стандарты и наши среды должны реализовывать определенные интерфейсы. Самой известной и наиболее широко используемой библиотекой для сред обучения с подкреплением является проект на Python с открытым исходным кодом под названием gym<sup>4</sup> компании OpenAI.

Давайте посмотрим, что такое библиотека Gym и как конвертировать нашу среду (Environment) лабиринта из предыдущей главы в среду библиотеки Gym, совместимую с библиотекой RLlib.

---

<sup>1</sup> См. <https://oreil.ly/14JhM>.

<sup>2</sup> См. <https://oreil.ly/Vp6xY>.

<sup>3</sup> См. <https://oreil.ly/KEhGx>.

<sup>4</sup> См. <https://gym.openai.com/>.

## Разработка среды в рамках библиотеки Gym

Если вы обратитесь к хорошо задокументированному и легко читаемому интерфейсу среды `gym.Env` на GitHub<sup>1</sup>, то заметите, что реализация этого интерфейса имеет две обязательные переменные класса и три метода, которые должны реализовываться подклассами. Вам не обязательно заглядывать в исходный код, но мы рекомендуем вам с ним ознакомиться. Возможно, вы просто удивитесь тому, как много вы уже знаете об этих средах.

Вкратце интерфейс среды библиотеки Gym выглядит как следующий ниже псевдокод:

```
import gym

class Env:

    action_space: gym.spaces.Space
    observation_space: gym.spaces.Space ①

    def step(self, action): ②
        ...

    def reset(self): ③
        ...

    def render(self, mode="human"): ④
        ...
```

- ① Интерфейс среды `gym.Env` имеет пространство действий и наблюдений.
- ② Среда `Env` может выполнять `step` и возвращать кортеж наблюдений, вознаграждение, условие завершения работы и дополнительную информацию.
- ③ Среда `Env` может сама себя сбрасывать (`reset`), в результате чего будут возвращаться первоначальные наблюдения нового эпизода.
- ④ Среду `Env` можно отрисовывать (`render`) для различных целей, например чтобы показывать человеку либо в виде строкового литерала.

Как вы помните из главы 3, это очень похоже на интерфейс среды лабиринта, который мы там сформировали. На самом деле в библиотеке Gym есть так называемое дискретное (`discrete`) пространство, реализованное в `gym.spaces`, и, следовательно, мы можем превратить нашу среду лабиринта в среду `gym.Env`, как показано ниже. Мы исходим из допущения, что вы храните этот исходный код в файле с именем `maze_gym_env.py` и что исходный код среды из главы 3 расположен в верхней части этого файла (или импортируется туда):

```
# maze_gym_env.py | Изначальное определение
# среды (Environment) помещается в верхней части

import gym
from gym.spaces import Discrete ①

class GymEnvironment(Environment, gym.Env): ②
```

---

<sup>1</sup> См. <https://oreil.ly/R3Ob1>.

```

def __init__(self, *args, **kwargs):
    """Сделать изначальную среду (Environment)
    средой gym `Env`"""
    super().__init__(*args, **kwargs)

gym_env = GymEnvironment()

```

- ❶ Заменить нашу собственную реализацию дискретного пространства (*Discrete*) реализацией в библиотеке Gym.
- ❷ Наделить класс `GymEnvironment` реализацией среды `gym.Env`. Интерфейс, по сути, остается таким же, как и раньше.

Разумеется, мы с самого начала могли наделить нашу изначальную среду реализацией среды `gym.Env`, просто наследуя от нее. Но дело в том, что интерфейс `gym.Env` в контексте обучения с подкреплением возникает настолько естественно, что будет неплохо поупражняться в его реализации без необходимости прибегать к внешним библиотекам<sup>1</sup>.

Интерфейс `gym.Env` также поставляется с полезными служебными функциональностями и множеством интересных примеров реализации. Например, среда `CartPole-v1`, которую мы использовали в главе 1, является примером из библиотеки Gym<sup>2</sup>, и для тестирования ваших алгоритмов обучения с подкреплением есть целый ряд других сред<sup>3</sup>.

## РАБОТА С ИНТЕРФЕЙСОМ КОМАНДНОЙ СТРОКИ БИБЛИОТЕКИ RLlib

Теперь, когда у нас есть среда `GymEnvironment`, реализованная в виде `gym.Env`, самое время использовать ее с помощью библиотеки RLlib. В главе 1 вы увидели интерфейс командной строки (CLI) библиотеки RLlib в действии, но на этот раз ситуация немного иная. В первой главе мы просто выполняли отрегулированный пример, используя команду `rllib example`.

На этот раз мы хотим ввести наш собственный средовой класс библиотеки `gym`, а именно класс `GymEnvironment`, который мы определили в файле Python `maze_gym_env.py`. При указании этого класса в рамках библиотеки Ray RLlib нужно полностью квалифицировать имя класса, из которого вы на него ссыл-

<sup>1</sup> Начиная с версии Ray 2.3.0 в библиотеке RLlib будет использоваться библиотека Gymnasium в качестве упрощенной замены библиотеки Gym. Это, вероятно, приведет к некоторым кардинальным изменениям, поэтому для того, чтобы отслеживать изложение в данной главе, лучше всего придерживаться фреймворка Ray версии 2.2.0.

<sup>2</sup> Библиотека Gym поставляется с большим числом интересных сред, которые стоит изучить. Например, есть целый ряд сред Atari, которые использовались в знаменитой статье разработчиков из DeepMind «Играем в игры Atari с помощью глубокого обучения с подкреплением» (Playing Atari with Deep Reinforcement Learning, <https://arxiv.org/abs/1312.5602>) или в продвинутых физических симуляциях с использованием движка MuJoCo.

<sup>3</sup> См. [https://oreil.ly/Mj6\\_t](https://oreil.ly/Mj6_t).

лается, так что в нашем случае это будет `maze_gym_env.GymEnvironment`. Если бы у вас был более сложный проект Python и ваша среда хранилась бы в другом модуле, то соответственно вы бы просто добавили имя модуля.

Следующий ниже файл Python задает минимальную конфигурацию, необходимую для тренировки алгоритма библиотеки RLlib на классе `GymEnvironment`. Для максимально точного выравнивания с нашим экспериментом из главы 3, в котором мы использовали Q-обучение, мы используем объект `DQNConfig`, чтобы описать глубокую Q-сеть<sup>1</sup> и сохранить это описание в файле с именем `maze.py`:

```
from ray.rllib.algorithms.dqn import DQNConfig

config = DQNConfig().environment("maze_gym_env.GymEnvironment")\
    .rollouts(num_rollout_workers=2)
```

Приведенный выше фрагмент исходного кода дает беглый обзор Python'овского API библиотеки RLlib, который мы рассмотрим в следующем разделе. В целях выполнения приведенного выше исходного кода с помощью библиотеки RLlib мы используем команду `rllib train`. Мы делаем это, указывая файл, который хотим выполнить: `maze.py`. В целях обеспечения контроля над временем тренировки мы сообщаем алгоритму о необходимости остановиться (`stop`) после выполнения в общей сложности 10 000 временных шагов (`timesteps_total`):

```
rllib train file maze.py --stop '{"timesteps_total": 10000}'
```

Приведенная выше единственная строка повторяет все, что мы делали в главе 3, только лучше:

- она выполняет для нас более изощренную версию Q-обучения (на основе глубокой Q-сети, или DQN)<sup>2</sup>;
- под капотом она берет на себя масштабирование до нескольких работников (в данном случае до двух);
- она даже автоматически создает за нас контрольные точки алгоритма.

Из выходных данных этого скрипта тренировки вы должны увидеть, что фреймворк Ray будет писать результаты тренировки в каталог, расположенный по адресу `~/ray_results/maze_env`. И если тренировочный прогон завершится успешно<sup>3</sup>, то на выходе вы получите контрольную точку и копируемую команду `rllib evaluate`, точно так же, как в примере из главы 1. Используя эту контрольную точку (<checkpoint>), теперь, выполнив следующую ниже команду, можно оценить натренированную политику в конкретно-прикладной среде:

```
rllib evaluate ~/ray_results/maze_env/<checkpoint> \
--algo DQN\
```

---

<sup>1</sup> Англ. Deep Q-Network (DQN). – Прим. перев.

<sup>2</sup> Если быть точным, то в библиотеке RLlib используется двойная и дуэльная (или соперничающая) глубокая Q-сеть.

<sup>3</sup> В репозиторий GitHub этой книги мы также включили эквивалентный файл `maze.yml`, который вы могли бы использовать в команде `rllib train file maze.yml` (опция `--type` не требуется).

---

```
--env maze_gym_env.Environment\
--steps 100
```

Используемый в опции `--algo` алгоритм и указанная в опции `--env` среда должны соответствовать тем, которые применялись в тренировочном про- гоне, и мы оцениваем натренированный алгоритм в общей сложности за 100 шагов. На выходе мы должны получить следующий ниже результат:

```
Episode #1: reward: 1.0
Episode #2: reward: 1.0
Episode #3: reward: 1.0
...
Episode #13: reward: 1.0
```

Не должно быть большим сюрпризом, что алгоритм глубокой Q-сети (DQN) из библиотеки RLLib получает максимальное вознаграждение в размере 1 балла за простую среду лабиринта, которую мы ему подавали в качестве задания каждый раз.

Прежде чем перейти к API Python, мы должны упомянуть, что в интерфей- се командной строки библиотеки RLLib под капотом используется библиотека Ray Tune, например для создания контрольных точек ваших алгоритмов. Подробнее об этой интеграции вы узнаете в главе 5.

## Использование Python API библиотеки RLLib

В конечном счете интерфейс командной строки (CLI) библиотеки RLLib – это всего лишь обертка вокруг опорной библиотеки Python. Поскольку большую часть своего времени вы, скорее всего, будете тратить на программирование своих экспериментов по обучению с подкреплением на Python, остальную часть этой главы мы посвятим аспектам данного API.

Вашей главной точкой входа при выполнении рабочих нагрузок обуче- ния с подкреплением с помощью библиотеки RLLib из Python является класс `Algorithm`. Для определения алгоритма следует всегда начинать с соответст- вующего класса `AlgorithmConfig`. Например, в предыдущем разделе в качестве стартовой точки мы использовали объект `DQNConfig`, а команда `rllib train` взяла на себя создание экземпляра алгоритма глубокой Q-сети (DQN). Все остальные алгоритмы библиотеки RLLib следуют тому же шаблону.

## Тренировка алгоритмов библиотеки RLLib

Каждый алгоритм (`Algorithm`) библиотеки RLLib поставляется с используемы- ми по умолчанию разумными параметрами, и, стало быть, их можно инициа- лизировать без необходимости налаживать для этих алгоритмов какие-либо конфигурационные параметры<sup>1</sup>.

---

<sup>1</sup> Разумеется, конфигурирование моделей является важнейшей частью эксперимен- тов по обучению с подкреплением. Конфигурированию алгоритмов библиотеки RLLib будет уделено особое внимание в следующем разделе.

Тем не менее, как вы увидите в следующем далее примере, стоит отметить, что алгоритмы библиотеки RLlib легко конфигурируются. Мы начинаем с создания объекта `DQNConfig`. Затем указываем его среду (`environment`) и устанавливаем число работников розыгрыша равным двум, используя метод `rollouts`. Это означает, что алгоритм глубокой Q-сети (DQN) породит двух акторов Ray, каждый из которых по умолчанию использует центральный процессор, чтобы выполнять алгоритм в параллельном режиме. Кроме того, для целей последующего оценивания мы устанавливаем значение аргумента `create_env_on_local_worker` равным `True`<sup>1</sup>:

```
from ray.tune.logger import pretty_print
from maze_gym_env import GymEnvironment
from ray.rllib.algorithms.dqn import DQNConfig

config = (DQNConfig().environment(GymEnvironment) ❶
          .rollouts(num_rollout_workers=2, create_env_on_local_worker=True))

pretty_print(config.to_dict())

algo = config.build() ❷

for i in range(10):
    result = algo.train() ❸

print(pretty_print(result)) ❹
```

- ❶ Установить значение среды (`environment`) в конкретно-прикладном классе `GymEnvironment` и сконфигурировать число работников розыгрыша, а также обеспечить, чтобы экземпляр среды был создан на локальном работнике.
- ❷ Использовать объект `DQNConfig` из библиотеки RLlib, чтобы построить (`build`) подлежащий тренировке алгоритм глубокой Q-сети (DQN). На этот раз мы используем двух работников розыгрыша.
- ❸ Вызывать метод `train` и тренировать алгоритм в течение 10 итераций.
- ❹ С помощью служебной функции `pretty_print` на выходе можно генерировать удобочитаемые результаты тренировки.

Обратите внимание, что число итераций тренировки не имеет особого значения, но этого должно быть достаточно, чтобы алгоритм научился адекватно решать задачу о лабиринте. Данный пример просто показывает, что у вас есть полный контроль над процессом тренировки.

Из распечатки словаря `config` вы можете убедиться, что параметру `num_rollout_workers` присвоено значение 2<sup>2</sup>. Переменная `result` содержит подробную информацию о состоянии алгоритма глубокой Q-сети (DQN) и результатах тренировки, которые слишком подробны, чтобы их здесь показывать. Прямо сейчас для нас наиболее важна часть, связанная с информацией о вознаграж-

---

<sup>1</sup> Данный аргумент сообщает о создании среды на локальном работнике. – Прим. перев.

<sup>2</sup> Если установить значение аргумента `num_rollout_workers` равным 0, то будет создан только локальный работник на головном узле, и все образцы из `env` будут браться там. Это особенно полезно для отладки, поскольку никакие дополнительные акторные процессы Ray порождаться не будут.

дении алгоритма, которая в идеале указывает на то, что алгоритм научился решать задачу о лабиринте. Вы должны увидеть результаты в следующем ниже виде (для наглядности мы показываем только самую релевантную информацию):

```
...
episode_reward_max: 1.0
episode_reward_mean: 1.0
episode_reward_min: 1.0
episodes_this_iter: 15
episodes_total: 19
...
training_iteration: 10
...
```

В частности, этот результат показывает, что минимальное вознаграждение, получаемое в среднем за эпизод, составляет 1.0, что, в свою очередь, означает, что агент всегда достигал цели и получал максимальное вознаграждение (1.0).

## ***Сохранение, загрузка и оценивание моделей библиотеки RLLib***

Достичь цели в этом простом примере не так уж и сложно, но давайте посмотрим, подтверждают ли результаты оценивания натренированного алгоритма, что агент также способен это делать оптимальным путем, а именно выполняя для достижения цели лишь минимальное число шагов, равное восьми.

Для этого мы задействуем еще один механизм, который вы уже увидели в интерфейсе командной строки библиотеки RLLib: *фиксирование состояний в контрольных точках*. Создание контрольных точек алгоритма полезно тем, что есть возможность восстанавливать свою работу в случае сбоя либо просто постоянно отслеживать продвижение тренировки. Контрольную точку алгоритма библиотеки RLLib можно создавать в любой момент процесса тренировки путем вызова метода `algo.save()`. Имея контрольную точку, с ее помощью можно легко восстанавливать свой `Algorithm`. Оценивание модели осуществляется так же просто и сводится к вызову `algo.evaluate(checkpoint)` с созданной вами контрольной точкой. Вот как выглядит исходный код, если все это сложить вместе:

```
from ray.rllib.algorithms.algorithm import Algorithm

checkpoint = algo.save() ❶
print(checkpoint)

evaluation = algo.evaluate() ❷
print(pretty_print(evaluation))

algo.stop() ❸
restored_algo = Algorithm.from_checkpoint(checkpoint) ❹
```

❶ Сохранить алгоритм, чтобы создать контрольную точку.

- ② Оценить алгоритм библиотеки RLlib в любой момент времени, вызывая метод `evaluate`.
- ③ Остановить алгоритм (`algo`), чтобы освободить все истребованные ресурсы.
- ④ Восстановить алгоритм (`Algorithm`) из указанной контрольной точки с помощью `from_checkpoint`.

Глядя на выходные данные этого примера, теперь можно подтвердить, что натренированный алгоритм библиотеки RLlib действительно сошелся к хорошему решению задачи о лабиринте, на что указывают эпизоды длиной 8 в результатах оценивания:

```
~/ray_results/DQN_GymEnvironment_2022-02-09_10-19-301o3m9r6d/checkpoint_000010/  
checkpoint-10 evaluation:
```

```
...  
episodes_this_iter: 5  
hist_stats:  
episode_lengths:  
- 8  
- 8  
...  
...
```

## Вычисление действий

Алгоритмы библиотеки RLlib обладают гораздо большей функциональностью, чем просто методы `train`, `evaluate`, `save` и `from_checkpoint`, которые мы увидели до сих пор. Например, можно выполнять действия напрямую с учетом текущего состояния среды. В главе 3 мы реализовали розыгрыши эпизодов путем прохождения среды и сбора вознаграждений. То же самое можно легко сделать и с библиотекой RLlib в рамках нашей среды `GymEnvironment`:

```
env = GymEnvironment()  
done = False  
total_reward = 0  
observations = env.reset()  
  
while not done:  
    action = algo.compute_single_action(observations) ❶  
    observations, reward, done, info = env.step(action)  
    total_reward += reward
```

- ❶ В целях вычисления действий для указанных наблюдений следует использовать метод вычисления одного действия (`compute_single_action`).

В случае если потребуется вычислять одновременно много действий, а не только одно, то вместо этого можно использовать метод вычисления нескольких действий (`compute_actions`), который на входе принимает словарь наблюдений и на выходе создает словарь действий с теми же словарными ключами:

```
action = algo.compute_actions(❶  
    {"obs_1": observations, "obs_2": observations}  
)
```

```
print(action)
# {'obs_1': 0, 'obs_2': 1}
```

- ❶ Для нескольких действий следует использовать метод `compute_actions`.

## **Доступ к политике и модельным состояниям**

Напомним, что каждый алгоритм обучения с подкреплением основан на политике, которая выбирает следующие действия с учетом текущих наблюдений агента в окружающей среде. Каждая политика, в свою очередь, основана на опорной модели.

В случае классического алгоритма Q-обучения, который мы обсуждали в главе 3, модель представляла собой простую просмотровую таблицу значений состояний–действий, также именуемых Q-значениями. И эта политика использовала ту модель для предсказания следующих действий, на случай если она решит *эксплуатировать* то, что модель узнала к настоящему времени, либо *разведывала* среду с помощью случайных действий в противном случае.

При использовании глубокого Q-обучения опорной моделью политики является нейронная сеть, которая, грубо говоря, ставит действия в соответствие наблюдениям. Обратите внимание, что при выборе следующих действий в среде нас в конечном счете интересуют не конкретные значения аппроксимированных Q-значений, а вероятности выполнения каждого действия. Статистическое распределение вероятностей по всем возможным действиям называется *распределением действий*.

В лабиринте, который мы используем в качестве текущего примера, можно двигаться вверх, вправо, вниз либо влево. Поэтому в нашем случае распределение действий – это вектор из четырех вероятностей, по одной для каждого действия. В случае Q-обучения алгоритм всегда будет *жадно* выбирать действие с наибольшей вероятностью этого распределения, тогда как другие алгоритмы будут брать из него образцы.

В целях прояснения ситуации давайте посмотрим на то, как получать доступ к политикам и моделям в библиотеке RLlib<sup>1</sup>:

```
policy = algo.get_policy()
print(policy.get_weights())

model = policy.model
```

Как в политике (`policy`), так и в модели (`model`) есть много полезных методов, с которыми стоит познакомиться. В этом примере мы используем метод `get_weights`, чтобы проинспектировать параметры модели, лежащей в основе политики (которые по стандартному соглашению называются *весами*).

<sup>1</sup> Класс `Policy` в сегодняшней библиотеке RLlib будет заменен в следующем ее выпуске. Новый класс `Policy`, скорее всего, по большей части будет упрощенной заменой и иметь несколько незначительных отличий. Однако идея класса остается прежней: *политика* – это класс, который инкапсулирует логику выбора действий при наличии наблюдений и предоставляет доступ к используемым опорным моделям.

Для того чтобы вас убедить в том, что здесь используется не одна модель, а фактически коллекция моделей<sup>1</sup>, мы можем получить доступ ко всем работникам, которых мы использовали при тренировке, а затем запросить у политики каждого работника их веса, используя `foreach_worker`:

```
workers = algo.workers
workers.foreach_worker(
    lambda remote_trainer: remote_trainer.get_policy().get_weights()
)
```

Благодаря этому можно получать доступ ко всем методам, имеющимся в экземпляре класса `Algorithm` на каждом работнике. В принципе, это также можно использовать для установки модельных параметров или иным образом конфигурировать своих работников. Работники библиотеки RLlib в конечном счете являются акторами Ray, поэтому их можно изменять и манипулировать ими практически любым удобным для вас способом.

Мы не говорили о конкретной реализации глубокого Q-обучения, используемой в глубокой Q-сети (DQN), но используемая модель чуть сложнее, чем то, что мы описывали до сих пор. Каждая полученная из политики модель библиотеки RLlib имеет атрибут `base_model`, который содержит замечательный метод `summary`, служащий для описания самой себя<sup>2</sup>:

```
model.base_model.summary()
```

Как видно из следующего ниже результата, эта модель принимает наши наблюдения (`observations`). Форма этих наблюдений аннотирована немного странно как `[(None, 25)]`, но, по сути, эта аннотация означает, что ожидаемые значения в решетке лабиринта размером  $5 \times 5$  закодированы правильно. Модель использует два так называемых плотных (Dense) слоя и в конце предсказывает одно-единственное значение<sup>3</sup>:

```
Model: "model"
```

---

Layer (type)	Output Shape	Param #	Connected to
observations (InputLayer)	[(None, 25)]	0	

---

<sup>1</sup> С технической точки зрения, для фактической тренировки используется только локальная модель. Две модели-работника применяются для вычисления действий и сбора данных (результатов). После каждого шага тренировки локальная модель отправляет свои текущие веса работникам для синхронизации. Полностью распределенная тренировка, в отличие от распределенного взятия образцов, будет доступна для всех алгоритмов библиотеки RLlib в будущих версиях фреймворка Ray.

<sup>2</sup> Это верно по умолчанию, поскольку под капотом мы используем TensorFlow и Keras. Если вы решите изменить спецификацию фреймворка (framework) вашего алгоритма, чтобы работать с PyTorch напрямую, то следует выполнить `print(model)`, в каковом случае моделью (`model`) будет `torch.nn.Module`. В будущем доступ к опорной модели будет унифицирован для всех фреймворков.

<sup>3</sup> Результат «value» на выходе из этой сети представляет собой Q-значение пар состояния–действие.

fc_1 (Dense)	(None, 256)	6656	observations[0][0]
fc_out (Dense)	(None, 256)	65792	fc_1[0][0]
value_out (Dense)	(None, 1)	257	fc_1[0][0]
<hr/>			
Total params: 72,705			
Trainable params: 72,705			
Non-trainable params: 0			

Обратите внимание, что эту модель вполне можно адаптировать под свои эксперименты с библиотекой RLlib. Например, если ваша среда отличается сложностью и имеет большое пространство наблюдений, то вам, возможно, понадобится модель большего размера, чтобы отразить эту сложность. Однако для этого потребуется глубокое знание опорного нейросетевого фреймворка (в данном случае TensorFlow), которого, как мы предполагаем, у вас нет<sup>1</sup>.

### ЗНАЧЕНИЯ СОСТОЯНИЕ–ДЕЙСТВИЕ И ФУНКЦИИ СОСТОЯНИЕ–ЦЕННОСТЬ

До сих пор нас в основном интересовала концепция значений состояния–действие, поскольку эта концепция занимает центральное место в формулировке Q-обучения, которую мы широко использовали до этого. Модель, с которой мы только что ознакомились, имеет специальный выходной элемент (в терминах глубокого обучения именуемый *головой*<sup>2</sup>) для предсказания Q-значений. К этой части можно обращаться и резюмировать ее посредством `model.q_value_head.summary()`.

В противоположность сказанному также есть возможность запрашивать ценность конкретного *состояния*, не указывая действие, которое с ним сопряжено. Это приводит к концепции *функций состояния–ценность*, или просто *функций ценности*, которые очень важны в литературе по обучению с подкреплением. В данном введении в библиотеку RLlib мы не сможем вдаваться в подробности, но обратите внимание, что у вас также есть доступ к *голове функции ценности* посредством `model.state_value_head.summary()`.

Далее давайте посмотрим, получится ли у нас взять несколько наблюдений из среды и передать их в модель, которую мы только что извлекли из нашей политики. Эта часть чуть сложна с технической точки зрения, потому что обращаться к моделям непосредственно в библиотеке RLlib немного сложнее. Обычно взаимодействие с моделью осуществляется только через собственную политику, которая, помимо прочего, берет на себя предобработку наблюдений. К счастью, есть возможность обращаться к используемому политикой предобработчику, преобразовывать (`transform`) наблюдения из собственной среды, а затем передавать их в модель:

<sup>1</sup> Для того чтобы узнать подробнее об адаптации своих моделей библиотеки RLlib под конкретно-прикладные задачи, следует ознакомиться с руководством по конкретно-прикладным моделям в документации Ray (<https://oreil.ly/cpRdf>).

<sup>2</sup> Англ. head. – Прим. перев.

```
from ray.rllib.models.preprocessors import get_preprocessor
env = GymEnvironment()
obs_space = env.observation_space
preprocessor = get_preprocessor(obs_space)(obs_space) ①
observations = env.reset()
transformed = preprocessor.transform(observations).reshape(1, -1) ②
model_output, _ = model({"obs": transformed}) ③
```

- ① Применить функцию `get_preprocessor`, чтобы получить доступ к используемому политической предобработчику.
- ② Любые наблюдения (`observations`), которые вы получили из вашей среды (`env`), можно преобразовать (`transform`) в форму, ожидаемую моделью. Обратите внимание, что, помимо этого, также нужно реформировать (`reshape`) наблюдения.
- ③ Вызвать модель с предобработанным словарем наблюдений и получить из нее результат.

Вычислив модельный результат (`model_output`), теперь в этом результате можно обратиться к Q-значениям и модельному распределению действий:

```
q_values = model.get_q_value_distributions(model_output) ①
print(q_values)

action_distribution = policy.dist_class(model_output, model) ②
sample = action_distribution.sample() ③
print(sample)
```

- ① Метод `get_q_value_distributions` специфичен только для моделей глубокой Q-сети (DQN).
- ② Обращаясь к `dist_class`, мы получаем принадлежащий политике класс распределения действий.
- ③ Из распределений действий можно брать образцы.

## Конфигурирование экспериментов с помощью библиотеки RLlib

Теперь, когда вы ознакомились на примере с базовым API тренировки на Python в рамках библиотеки RLlib, давайте отступим на шаг назад и подробнее остановимся на том, как конфигурировать и выполнять эксперименты с помощью библиотеки RLlib. К настоящему времени вы уже знаете, что для определения алгоритма (`Algorithm`) вы начинаете с перспективной конфигурации `AlgorithmConfig`, а затем на ее основе строите (`build`) свой алгоритм. До этого мы использовали лишь метод `rollout` класса `AlgorithmConfig`, чтобы устанавливать число работников розыгрыша равным двум и соответствующим образом задавать нашу среду (`environment`).

Если вы хотите изменить поведение вашего тренировочного прогона в рамках библиотеки RLlib, то к экземпляру `AlgorithmConfig` следует прице-

пить дополнительные служебные методы, а затем в самом конце на нем вызвать метод `build`. Поскольку алгоритмы библиотеки RLLib довольно сложны, они имеют множество опций конфигурирования. В целях упрощения общие свойства алгоритмов естественным образом сгруппированы в полезные категории<sup>1</sup>. Каждая такая категория поставляется со своим собственным методом класса `AlgorithmConfig`:

#### `training()`

Берет на себя все конфигурационные опции вашего алгоритма, связанные с тренировкой. Метод `training` – это единственное место, где алгоритмы библиотеки RLLib отличаются по своей конфигурации. Все следующие методы не зависят от алгоритма.

#### `environment()`

Конфигурирует все аспекты вашей среды.

#### `rollouts()`

Видоизменяет настройку и поведение ваших работников розыгрыша.

#### `exploration()`

Изменяет поведение вашей стратегии разведки.

#### `resources()`

Конфигурирует вычислительные ресурсы, используемые вашим алгоритмом.

#### `offline_data()`

Определяет опции тренировки с использованием так называемых офлайновых данных, тему которых мы рассмотрим в разделе «Работа с офлайновыми данными» на стр. 122.

#### `multi_agent()`

Определяет опции для тренировки алгоритмов с использованием *нескольких агентов*. В следующем разделе мы обсудим наглядный пример.

Специфичная для алгоритма конфигурация в методе `training()` становится еще более релевантной после того, как вы определились с алгоритмом и хотите увеличить его результативность. На практике библиотека RLLib предоставляет хорошие дефолтные конфигурационные настройки, чтобы сразу приступить к работе.

Более подробную информацию о конфигурировании экспериментов с помощью библиотеки RLLib можно найти в справочнике по API для алгоритмов библиотеки RLLib, обратившись к конфигурационным аргументам<sup>2</sup>. Но преж-

<sup>1</sup> Мы перечисляем только те методы, которые представляем в данной главе. Помимо тех, которые мы упомянули, вы также найдете опции для оценивания (`evaluation`) своих алгоритмов, создания отчетов (`reporting`), отладки (`debugging`), фиксации состояния в контрольных точках (`checkpointing`), добавления обратных вызовов (`callbacks`), изменения вашего фреймворка (`framework`) глубокого обучения, запроса ресурсов (`resources`) и доступа к экспериментальным функциональностям (`experimental`).

<sup>2</sup> См. <https://oreil.ly/4q1eo>.

де чем перейти к примерам, следует ознакомиться с наиболее распространенными конфигурационными опциями на практике.

## Конфигурирование ресурсов

Независимо от того, как библиотека Ray RLlib используется: локально либо в кластере, – вы можете задавать ресурсы, применяемые для процесса тренировки. Вот наиболее важные варианты, которые следует рассмотреть. Мы продолжаем использовать алгоритм глубокой Q-сети (DQN) в качестве примера, но это применимо к любому другому алгоритму библиотеки RLlib:

```
from ray.rllib.algorithms.dqn import DQNConfig

config = DQNConfig().resources(
    num_gpus=1, ①
    num_cpus_per_worker=2, ②
    num_gpus_per_worker=0, ③
)
```

- ① Указать число графических процессоров, которые будут использоваться для тренировки. Важно сначала проверить, что выбранный вами алгоритм поддерживает графические процессоры. Это значение также может быть дробным. Например, при использовании четырех работников розыгрыша в глубокой Q-сети (`num_rollout_workers=4`) можно установить `num_gpus=0.25`, чтобы упаковать всех четырех работников на одном графическом процессоре с целью извлечения выгоды из потенциального ускорения всеми работниками. Это влияет только на процесс локального ученика, а не на работников розыгрыша.
- ② Установить число центральных процессоров, используемых каждым работником.
- ③ Установить число графических процессоров, используемых каждым работником.

## Конфигурирование работников розыгрыша

Библиотека RLlib позволяет конфигурировать способ вычисления розыгрышей и их распределение:

```
from ray.rllib.algorithms.dqn import DQNConfig

config = DQNConfig().rollouts(
    num_rollout_workers=4, ①
    num_envs_per_worker=1, ②
    create_env_on_local_worker=True, ③
)
```

- ① Этот аргумент вы уже встречали. Он определяет число используемых работников Ray.
- ② Указать число оцениваемых сред на одного работника. Этот настроочный параметр позволяет выполнять «пакетное» оценивание сред. В частности, если оценивание моделей занимает много времени, то подобного рода группировка сред может ускорять тренировку.
- ③ При `num_rollout_workers > 0` драйвер («локальный работник») не нуждается в среде. Это связано с тем, что взятие образцов и оценивание выполняется работниками розыгрыша. Если вам все же нужна среда на драйвере, то этот параметр можно установить равным `True`.

## Конфигурирование сред

```
from ray.rllib.algorithms.dqn import DQNConfig

config = DQNConfig().environment(
    env="CartPole-v1", ①
    env_config={"my_config": "value"}, ②
    observation_space=None, ③
    action_space=None,
    render_env=True, ④
)
```

- ① Указать среду, которую вы хотите использовать для тренировки. Это может быть либо строковый литерал известной библиотеке Ray RLlib среды, такой как любая среда библиотеки Gym, либо имя реализованного вами класса конкретно-прикладной среды<sup>1</sup>.
- ② Опционально указать в своей среде словарь конфигурационных опций, который будет передан конструктору среды.
- ③ Также можно указать пространства наблюдений и действий в своей среде. Если вы их не укажете, то они будут сгенерированы из среды.
- ④ Это свойство, по умолчанию равное `False`, позволяет включать отрисовку среды, что требует от вас реализации метода `render` своей среды.

Обратите внимание, что мы не указали множество имеющихся конфигурационных опций для каждого из перечисленных нами типов. Вдобавок ко всему в этом введении мы не сможем затронуть аспекты, которые изменяют поведение процедуры тренировки алгоритма обучения с подкреплением (например, видоизменение подлежащей использованию опорной модели). Однако хорошая новость состоит в том, что всю необходимую информацию можно найти в документации по API тренировки в рамках библиотеки RLlib<sup>2</sup>.

## РАБОТА СО СРЕДАМИ БИБЛИОТЕКИ RLlib

До сих пор мы знакомили вас только со средой библиотеки Gym, но библиотека RLlib поддерживает широкий спектр сред. После беглого обзора всех доступных опций (см. рис. 4.1) мы покажем два конкретных примера расширенных сред библиотеки RLlib в действии.

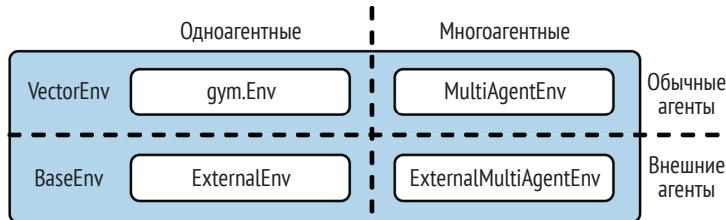
## Общий обзор сред библиотеки RLlib

Все имеющиеся в библиотеке RLlib среды расширяют общий класс `BaseEnv`. Если вы хотите работать с несколькими копиями одной и той же среды gym.

<sup>1</sup> Существует также способ *регистрировать* свои среды, чтобы иметь возможность к ним обращаться по имени, но для этого требуется использовать библиотеку Ray Tune. Об этой функциональной возможности вы узнаете в главе 5.

<sup>2</sup> См. <https://oreil.ly/mljW7>.

Env, то можете использовать обертку VectorEnv библиотеки RLlib. Векторизованные среды удобны, но они являются прямыми обобщениями того, что вы уже видели. Два других типа имеющихся в библиотеке RLlib сред более интересны и заслуживают большего внимания.



**Рис. 4.1** ♦ Общий обзор всех сред, имеющихся в библиотеке RLlib

Первый называется `MultiAgentEnv`, который позволяет тренировать модель с *несколькими агентами*. Работа с несколькими агентами бывает проблематичной. Это связано с тем, что вы должны помнить о необходимости определять своих агентов в своей среде с подходящим интерфейсом и учитывать тот факт, что у каждого агента может быть совершенно разный способ взаимодействия со своей средой.

Более того, агенты могут взаимодействовать друг с другом, и они должны учитывать действия друг друга. В более продвинутых конфигурациях даже может существовать *иерархия* агентов, которые зависят друг от друга в явной форме. Одним словом, проводить многоагентные эксперименты по обучению с подкреплением довольно трудно, и в следующем далее примере мы увидим, как библиотека RLlib с этим справляется.

Другой тип рассматриваемой нами среды называется `ExternalEnv`, его можно использовать для подсоединения внешних симуляторов к библиотеке RLlib. Например, вообразите себе, что наша предыдущая простая задача о лабиринте представляла собой симуляцию реального робота, перемещающегося по лабиринту. В таких сценариях может оказаться неприемлемо совмещать робота (или его симуляцию, реализованную в другом программном стеке) с обучающимися агентами библиотеки RLlib. Учитывая это, библиотека RLlib предоставляет простую клиент-серверную архитектуру, служащую для связи с внешними симуляторами, которая позволяет осуществлять связь по REST API. В случае если вы хотите работать как в многоагентной, так и во внешней среде, то библиотека RLlib предлагает конфигурацию многоагентной внешней среды, которая сочетает в себе и то, и другое.

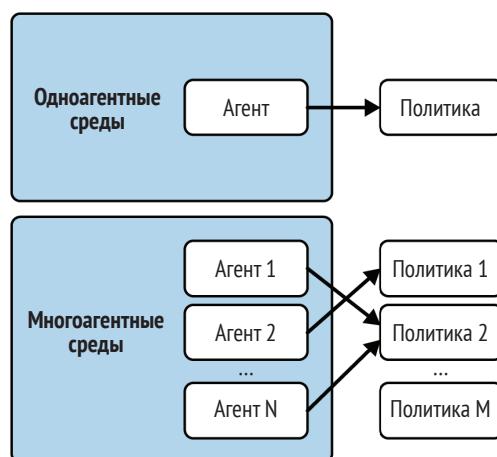
## Работа с несколькими агентами

Базовая идея определения многоагентных сред в библиотеке RLlib проста. Сначала вы присваиваете каждому агенту идентификатор. Затем все, что вы ранее определяли как одно значение в среде библиотеки Gym (наблюдения, вознаграждения и т. д.), теперь вы определяете как словарь с иденти-

фикаторами агентов в качестве ключей и значениями по каждому агенту. Разумеется, детали чуть сложнее, чем те, что на практике. Но после того, как вы определили среду, в которой размещено несколько агентов, необходимо определить то, как эти агенты должны учиться.

В одноагентной среде существует один агент, и подлежит усвоению одна политика. В многоагентной среде существует несколько агентов, и они могут соотноситься с одной или несколькими политиками. Например, если в вашей среде есть группа однородных агентов, то для всех них можно было определить одну политику. Если все они *действуют* одинаково, то их поведение можно усваивать одинаковым образом. И наоборот, у вас могут возникать ситуации с разнородными агентами, в которых каждый из них должен усваивать отдельную политику. Как показано на рис. 4.2, между этими двумя крайностями существует целый спектр возможностей.

Мы продолжим использовать игру в лабиринт в качестве примера данной главы. Благодаря этому вы сможете сами убедиться в том, насколько интерфейсы отличаются на практике. Итак, для того чтобы воплотить идеи, которые мы только что изложили в исходном коде, давайте определим многоагентную версию класса GymEnvironment. Наш класс MultiAgentEnv будет содержать ровно два агента, которых мы кодируем в словаре Python под названием agents, но в принципе это работает с любым числом агентов.



**Рис. 4.2** ♦ Соотнесенность агентов с политиками в многоагентных задачах обучения с подкреплением

Мы начинаем с инициализации и сброса нашей новой среды в исходное состояние:

```

from ray.rllib.env.multi_agent_env import MultiAgentEnv
from gym.spaces import Discrete
import os

class MultiAgentMaze(MultiAgentEnv):

```

```
def __init__(self, *args, **kwargs): ❶
    self.action_space = Discrete(4)
    self.observation_space = Discrete(5*5)
    self.agents = {1: (4, 0), 2: (0, 4)} ❷
    self.goal = (4, 4)
    self.info = {1: {'obs': self.agents[1]},
                2: {'obs': self.agents[2]}} ❸

def reset(self):
    self.agents = {1: (4, 0), 2: (0, 4)}
    return {1: self.get_observation(1), 2: self.get_observation(2)} ❹
```

- ❶ Пространства действий и наблюдений остаются точно такими же, как и раньше.
- ❷ Теперь у нас есть два искателя со стартовыми позициями (0, 4) и (4, 0) в словаре agents.
- ❸ Для объекта info мы используем идентификаторы агентов в качестве ключей.
- ❹ Наблюдения теперь являются поагентными словарями.

Обратите внимание, что мы вообще не касались пространств действий и наблюдений. Это связано с тем, что здесь мы используем два практически идентичных агента, которые могут реиспользовать одни и те же пространства. В более сложных ситуациях вам пришлось бы учитывать тот факт, что для некоторых агентов действия и наблюдения могут выглядеть по-разному<sup>1</sup>.

Продолжив, давайте обобщим наши вспомогательные методы get\_observation, get\_reward и is\_done на работу с несколькими агентами. Это делается путем передачи идентификатора действия (action\_id) их сигнатурам и оперирования каждым агентом таким же образом, как и раньше:

```
def get_observation(self, agent_id):
    seeker = self.agents[agent_id]
    return 5 * seeker[0] + seeker[1]

def get_reward(self, agent_id):
    return 1 if self.agents[agent_id] == self.goal else 0

def is_done(self, agent_id):
    return self.agents[agent_id] == self.goal
```

Далее, портируя метод step в нашу многоагентную конфигурацию, вы должны знать, что среда MultiAgentEnv теперь ожидает, что действие, переданное в метод step, тоже будет словарем, в котором ключи соответствуют идентификаторам агентов. Мы определяем шаг, просматривая всех доступных агентов и действуя от их имени<sup>2</sup>:

---

<sup>1</sup> Хороший пример, в котором для нескольких агентов определяются разные пространства наблюдений и действий, находится в документации библиотеки RLlib (<https://oreil.ly/4yyE->).

<sup>2</sup> Обратите внимание на то, как это может приводить к трудностям в принятии решения о том, какой агент должен действовать первым. В нашей простой задаче о лабиринте порядок действий не имеет значения, но в более сложных сценариях это становится важной частью правильного моделирования задачи обучения с подкреплением.

```

def step(self, action): ❶
    agent_ids = action.keys()

    for agent_id in agent_ids:
        seeker = self.agents[agent_id]
        if action[agent_id] == 0: # переместиться вниз
            seeker = (min(seeker[0] + 1, 4), seeker[1])
        elif action[agent_id] == 1: # переместиться влево
            seeker = (seeker[0], max(seeker[1] - 1, 0))
        elif action[agent_id] == 2: # переместиться вверх
            seeker = (max(seeker[0] - 1, 0), seeker[1])
        elif action[agent_id] == 3: # переместиться вправо
            seeker = (seeker[0], min(seeker[1] + 1, 4))
        else:
            raise ValueError("Недопустимое действие")
        self.agents[agent_id] = seeker ❷

    observations = {i: self.get_observation(i) for i in agent_ids} ❸
    rewards = {i: self.get_reward(i) for i in agent_ids}
    done = {i: self.is_done(i) for i in agent_ids}

    done["__all__"] = all(done.values()) ❹

    return observations, rewards, done, self.info

```

- ❶ Действия в методе `step` теперь являются поагентными словарями.
- ❷ После применения правильного действия для каждого искателя установить правильные состояния для всех агентов.
- ❸ Наблюдения (`observations`), вознаграждения (`rewards`) и завершенности (`dones`) также являются словарями, в которых идентификаторы агентов выступают ключами.
- ❹ Кроме того, библиотека RLlib должна знать, когда все агенты завершат работу.

Последним шагом является видоизменение отрисовки среды, что мы делаем, обозначая каждого агента его идентификатором при печати лабиринта на экран:

```

def render(self, *args, **kwargs):
    os.system('cls' if os.name == 'nt' else 'clear')
    grid = ['| ' for _ in range(5)] + ["|\n"] for _ in range(5)
    grid[self.goal[0]][self.goal[1]] = '|G'
    grid[self.agents[1][0]][self.agents[1][1]] = '|1'
    grid[self.agents[2][0]][self.agents[2][1]] = '|2'
    grid[self.agents[2][0]][self.agents[2][1]] = '|2'
    print(''.join([''.join(grid_row) for grid_row in grid]))

```

Случайный розыгрыш эпизода до тех пор, пока один из агентов не достигнет цели, может быть выполнен, например, с помощью следующего ниже исходного кода<sup>1</sup>:

---

<sup>1</sup> Принятие решения о завершении эпизода – важнейшая часть многоагентного обучения с подкреплением, и оно полностью зависит от поставленной задачи и того, чего вы хотите достичь.

```
import time

env = MultiAgentMaze()

while True:
    obs, rew, done, info = env.step(
        {1: env.action_space.sample(), 2: env.action_space.sample()})
    )
    time.sleep(0.1)
    env.render()
    if any(done.values()):
        break
```

Обратите внимание на необходимость обязательно передать в метод `step` два случайных образца посредством словаря Python и на проверку окончания своей работы еще каким-либо агентом. Для простоты мы используем условие `break`, потому что крайне маловероятно, что оба искателя случайно найдут свой путь к цели в одно и то же время. Но, разумеется, мы бы хотели, чтобы оба агента в конце концов прошли лабиринт.

В любом случае, оснащенная нашим классом `MultiAgentMaze`, тренировка алгоритма (`Algorithm`) библиотеки RLlib работает точно так же, как и раньше:

```
from ray.rllib.algorithms.dqn import DQNConfig

simple_trainer = DQNConfig().environment(env=MultiAgentMaze).build()
simple_trainer.train()
```

Приведенный выше фрагмент исходного кода охватывает простейший случай тренировки задачи многоагентного обучения с подкреплением<sup>1</sup>. Но если вы помните, что мы говорили ранее, при использовании нескольких агентов всегда существует соотнесенность между агентами и политиками. Без указания такой соотнесенности оба наших искателя были бы неявно привязаны к одной и той же политике. Это можно изменить, вызвав метод `.multi_agent` в нашей конфигурации `DQNConfig` и соответственно установив аргументы `policies` и `policy_mapping_fn`:

```
algo = DQNConfig()\
    .environment(env=MultiAgentMaze)\
    .multi_agent(
        policies={❶
            "policy_1": (
                None, env.observation_space,
                env.action_space, {"gamma": 0.80}
            ),
            "policy_2": (
                None, env.observation_space,
                env.action_space, {"gamma": 0.95}
            ),
        },
        policy_mapping_fn = lambda agent_id: f"policy_{agent_id}"❷
```

---

<sup>1</sup> Англ. multi-agent reinforcement learning (MARL).

```
.build()
print(algo.train())
```

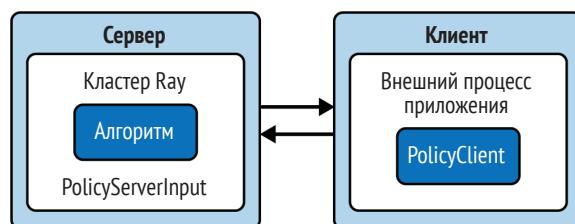
- ❶ Определить несколько политик (*policies*) для наших агентов, каждая из которых имеет свое значение "gamma".
- ❷ Затем каждого агента можно соотнести с политикой с помощью конкретно-прикладной функции *policy\_mapping\_fn*.

Как видите, проведение многоагентных экспериментов по обучению с подкреплением является полноправным элементом библиотеки RLlib, и об этом можно было бы сказать гораздо больше. Поддержка задач многоагентного обучения с подкреплением, вероятно, является одной из самых сильных функциональных особенностей библиотеки RLlib.

## Работа с серверами политик и клиентами

В последнем примере этого раздела давайте допустим, что наша изначальная среда *GymEnvironment* может быть просимулирована только на компьютере, который не может запускать библиотеку RLlib, например из-за нехватки доступных ресурсов. Мы можем запустить среду на клиенте *PolicyClient*, который может запрашивать у соответствующего *сервера* подходящие следующие действия, чтобы применять к среде. Сервер, в свою очередь, не знает о среде. Он знает только, как принимать входные данные из *PolicyClient*, и он отвечает за выполнение всего исходного кода, связанного с обучением с подкреплением, в частности определяет объект *AlgorithmConfig* библиотеки RLlib и тренирует алгоритм (*Algorithm*).

В типичной ситуации вы хотите управлять тренирующим ваш алгоритм сервером на мощном кластере Ray, и тогда соответствующий клиент будет работать за пределами этого кластера. Рисунок 4.3 схематично иллюстрирует эту конфигурацию.



**Рис. 4.3** ♦ Работа с серверами политик и клиентами в библиотеке RLlib

### Определение сервера

Прежде всего давайте начнем с определения серверной стороны такого приложения. Мы определяем так называемый *PolicyServerInput*, который работает на локальном хосте через порт 9900. Входные данные политики – это

то, что клиент предоставит позже. С помощью функции `policy_input`, определенной в качестве входных данных (`input`) в нашу алгоритмическую конфигурацию, можно определить еще одну глубокую Q-сеть (DQN) для работы на сервере:

```
# policy_server.py
import ray
from ray.rllib.agents.dqn import DQNConfig
from ray.rllib.env.policy_server_input import PolicyServerInput
import gym

ray.init()

def policy_input(context):
    return PolicyServerInput(context, "localhost", 9900) ①

config = DQNConfig() \
    .environment(
        env=None, ②
        action_space=gym.spaces.Discrete(4), ③
        observation_space=gym.spaces.Discrete(5*5)) \
    .debugging(log_level="INFO") \
    .rollouts(num_rollout_workers=0) \
    .offline_data(④
        input=policy_input,
        input_evaluation=[])

algo = config.build()
```

- ① Функция `policy_input` возвращает объект `PolicyServerInput`, работающий на локальном хосте (`localhost`) через порт 9900.
- ② Мы явно установили значение среды (`env`), равное `None`, потому что этот сервер в ней не нуждается.
- ③ Следовательно, нужно определить пространство наблюдений (`observation_space`) и пространство действий (`action_space`), поскольку сервер не в состоянии вывести их из среды.
- ④ Для запуска работы нужно подать `policy_input` на вход (`input`) эксперимента.

Определив этот алгоритм<sup>1</sup>, теперь можно начать сеанс тренировки на сервере, как показано ниже:

```
# policy_server.py
if __name__ == "__main__":
    time_steps = 0
    for _ in range(100):
        results = algo.train()
        checkpoint = algo.save() ①
        if time_steps >= 1000: ②
```

---

<sup>1</sup> По техническим причинам здесь мы должны указать пространства наблюдений и действий – в будущих выпусках библиотеки RLlib указанные пространства могут оказаться ненужными, поскольку из-за этого происходит утечка информации о среде. Также обратите внимание на то, что нужно установить значение `input_evaluation` равным пустому списку. Это приведет сервер в рабочее состояние.

```

        break
    time_steps += results["timesteps_total"]

```

- ❶ Тренировать максимум 100 итераций и сохранять контрольные точки после каждой итерации.
- ❷ Если тренировка превышает 1000 временных шагов, то мы прекращаем тренировку.

В дальнейшем мы будем исходить из того, что вы храните последние два фрагмента исходного кода в файле с именем *policy\_server.py*. Если хотите, то теперь можете запустить этот сервер политик на своем локальном компьютере, выполнив в терминале команду `python policy_server.py`.

## Определение клиента

Далее в целях определения соответствующей клиентской стороны приложения мы определяем клиента политики (*PolicyClient*), подключаемого к серверу, который только что запустили. Поскольку мы не можем исходить из того, что у вас дома есть несколько компьютеров (либо они доступны в облаке), то запустим этого клиента на одной машине вопреки тому, что мы говорили ранее. Другими словами, клиент подключится к `http://localhost:9900`, но если вы можете запустить сервер на другом компьютере, то тогда можете заменить `localhost` на IP-адрес этого компьютера, при условии что он доступен в сети.

Клиенты политик имеют довольно простой интерфейс. Они могут инициировать сервер, запускать или завершать эпизод, получать из него следующие действия и заносить в него информацию о вознаграждении (которой в противном случае у него не было бы). С учетом сказанного ниже приводится определение такого клиента:

```

# policy_client.py
import gym
from ray.rllib.env.policy_client import PolicyClient
from maze_gym_env import GymEnvironment

if __name__ == "__main__":
    env = GymEnvironment()
    client = PolicyClient("http://localhost:9900",
                          inference_mode="remote") ❶

    obs = env.reset()
    episode_id = client.start_episode(training_enabled=True) ❷

    while True:
        action = client.get_action(episode_id, obs) ❸
        obs, reward, done, info = env.step(action)

        client.log_returns(episode_id, reward, info=info) ❹

        if done:
            client.end_episode(episode_id, obs) ❺
            exit(0) ❻

```

- ① Запустить клиента политики по адресу сервера в режиме дистанционного (remote) генерирования модельных предсказаний.
- ② Сообщить серверу, чтобы он запустил эпизод.
- ③ Для заданных наблюдений за средой с сервера можно получить следующее действие.
- ④ Клиент (client) в обязательном порядке регистрирует на сервере информацию о вознаграждении.
- ⑤ Если достигнуто определенное условие, то клиентский процесс можно остановить.
- ⑥ Если работа со средой окончена (done), то мы должны сообщить серверу об окончании эпизода.

Если допустить, что этот исходный код хранится в *policy\_client.py* и вы запускаете его командой `python policy_client.py`, то запущенный нами ранее сервер начнет учиться, используя информацию о среде, получаемую исключительно от клиента.

## ПРОДВИНУТЫЕ КОНЦЕПЦИИ

До сих пор мы работали с простыми средами, которыми было достаточно легко оперировать с помощью самых базовых настроек алгоритма обучения с подкреплением в рамках библиотеки RLlib. Конечно же, на практике не всегда так везет, и, возможно, для оперирования более сложными средами придется придумывать другие идеи. В этом разделе мы представим чуть более сложную версию среды лабиринта и обсудим несколько продвинутых концепций, которые помогут вам ее решить.

## Разработка продвинутой среды

Давайте сделаем нашу среду `GymEnvironment` лабиринта чуть более вызывающей по своей трудности. Сначала мы увеличиваем ее размер с решетки  $5 \times 5$  до решетки  $11 \times 11$ . Затем мы введем в лабиринт препятствия, через которые агент может проходить, но только получая штраф, отрицательное вознаграждение в размере  $-1$ . В силу этого нашему агенту-искателю придется научиться избегать препятствий, продолжая при этом находить цель. Кроме того, мы рандомизируем стартовую позицию агента. Все это затрудняет решение задачи обучения с подкреплением. Сначала давайте рассмотрим инициализацию этой новой среды `AdvancedEnv`:

```
from gym.spaces import Discrete
import random
import os

class AdvancedEnv(GymEnvironment):

    def __init__(self, seeker=None, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.maze_len = 11
        self.action_space = Discrete(4)
```

```

self.observation_space = Discrete(self.maze_len * self.maze_len)

if seeker:
    assert 0 <= seeker[0] < self.maze_len and \
        0 <= seeker[1] < self.maze_len
    self.seeker = seeker
else:
    self.reset()

self.goal = (self.maze_len-1, self.maze_len-1)
self.info = {'seeker': self.seeker, 'goal': self.goal}

self.punish_states = [
    (i, j) for i in range(self.maze_len)
        for j in range(self.maze_len)
            if i % 2 == 1 and j % 2 == 0
]

```

- ① Установить позицию искателя (`seeker`) при инициализации.
- ② Ввести штрафные состояния (`punish_states`) в качестве препятствий для агента.

Далее при сбросе среды мы хотим обеспечить, чтобы позиция агента сбрасывалась в случайное состояние<sup>1</sup>. Мы также увеличиваем положительное вознаграждение до 5 баллов за достижение цели, чтобы компенсировать отрицательное вознаграждение за прохождение препятствия (что будет происходить задолго до того, как алгоритм обучения с подкреплением определит местоположение препятствия). Такого рода балансировка вознаграждений является важнейшей работой при калибровке экспериментов по обучению с подкреплением:

```

def reset(self):
    """Рандомно сбрасывать позицию искателя,
    возвращать наблюдения"""
    self.seeker = (
        random.randint(0, self.maze_len - 1),
        random.randint(0, self.maze_len - 1)
    )
    return self.get_observation()

def get_observation(self):
    """Кодировать позицию искателя как целое число"""
    return self.maze_len * self.seeker[0] + self.seeker[1]

def get_reward(self):
    """Вознаграждать за достижение цели и штрафовать
    за запрещенные состояния"""
    reward = -1 if self.seeker in self.punish_states else 0
    reward += 5 if self.seeker == self.goal else 0
    return reward

```

---

<sup>1</sup> В определении `reset` мы разрешаем искателю выполнять сброс, зная о цели, чтобы сделать определение проще. Допущение этого тривиального пограничного случая никак не влияет на усвоение им среды.

```
def render(self, *args, **kwargs):
    """Отрисовывать среду, например путем распечатки
    ее представления"""
    os.system('cls' if os.name == 'nt' else 'clear')
    grid = ['|' + ' ' for _ in range(self.maze_len)] +
           ["|\n"] for _ in range(self.maze_len)]
    for punish in self.punish_states:
        grid[punish[0]][punish[1]] = '|X'
    grid[self.goal[0]][self.goal[1]] = '|G'
    grid[self.seeker[0]][self.seeker[1]] = '|S'
    print(''.join([''.join(grid_row) for grid_row in grid]))
```

Приведенную выше среду можно усложнить самыми разными способами, например сделать ее намного больше, ввести отрицательное вознаграждение за каждый шаг агента в определенном направлении или наказывать агента за попытку выходить за пределы решетки. К настоящему времени вы должны достаточно хорошо понимать постановку задачи, чтобы адаптировать лабиринт дальше.

Хотя вы, возможно, добьетесь успеха в тренировке этой среды, она является отличной возможностью познакомить вас с несколькими продвинутыми концепциями, которые можно применять к другим задачам обучения с подкреплением.

## Применение процедуры усвоения учебной программы

Одной из наиболее интересных функциональных возможностей библиотеки RLlib является предоставление алгоритму (*Algorithm*) подлежащей усвоению учебной программы<sup>1</sup>. Вместо того чтобы алгоритм учился на произвольных конфигурациях среды, мы подбираем состояния, на которых гораздо легче учиться, а затем медленно, но верно вводим более сложные состояния. Формирование учебной программы – отличный способ быстрее приводить эксперименты к решениям. В целях применения процедуры усвоения учебной программы нам нужна лишь одна вещь – понимать, какие стартовые состояния легче других. Во многих средах это бывает проблемой, но для нашего продвинутого лабиринта придумать простую учебную программу очень легко. А именно в качестве меры трудности можно использовать расстояние искателя до цели. Для простоты мы будем использовать меру расстояния, выражющуюся суммой абсолютного расстояния обеих координат искателя до цели, чтобы определять трудность (*difficulty*).

Для выполнения процедуры усвоения учебной программы с помощью библиотеки RLlib мы определяем класс *CurriculumEnv*, который расширяет наш класс *AdvancedEnv* и так называемый класс *TaskSettableEnv* из библиотеки RLlib. Интерфейс класса *TaskSettableEnv* очень прост в том смысле, что нужно определить только то, как получать текущую трудность (*get\_task*) и как

---

<sup>1</sup> Англ. *curriculum*. – Прим. перев.

устанавливать требуемую трудность (`set_task`). Вот полное определение этой учебной программы:

```
from ray.rllib.env.apis.task_settable_env import TaskSettableEnv

class CurriculumEnv(AdvancedEnv, TaskSettableEnv):

    def __init__(self, *args, **kwargs):
        AdvancedEnv.__init__(self)

    def difficulty(self): ❶
        return abs(self.seeker[0] - self.goal[0]) + \
            abs(self.seeker[1] - self.goal[1])

    def get_task(self): ❷
        return self.difficulty()

    def set_task(self, task_difficulty): ❸
        while not self.difficulty() <= task_difficulty:
            self.reset()
```

- ❶ Определить трудность текущего состояния как сумму абсолютного расстояния обеих координат искателя до цели.
- ❷ Затем при определении метода `get_task` можно просто вернуть текущую трудность (`difficulty`).
- ❸ При указании трудности задания мы сбрасываем (`reset`) среду до тех пор, пока ее трудность (`difficulty`) не будет превышать указанную трудность задания (`task_difficulty`).

В целях применения этой среды для усвоения учебной программы нужно определить функцию учебной программы, которая сообщает алгоритму, когда и как задавать трудность задания. Здесь у нас есть много вариантов, но мы используем расписание, которое каждые 1000 шагов тренировки просто увеличивает трудность на единицу:

```
def curriculum_fn(train_results, task_settable_env, env_ctx):
    time_steps = train_results.get("timesteps_total")
    difficulty = time_steps // 1000
    print(f"Текущая трудность: {difficulty}")
    return difficulty
```

Для тестирования этой функции учебной программы нужно добавить ее в конфигурацию алгоритма библиотеки RLLib, установив свойство `env_task_fn` равным функции `curriculum_fn`. Обратите внимание, что перед тренировкой глубокой Q-сети (DQN) в течение 15 итераций мы также устанавливаем в нашей конфигурации `выходную` папку. Благодаря этому данные об опыте из тренировочного прогона будут сохраняться в указанной папке `temp`<sup>1</sup>:

```
from ray.rllib.algorithms.dqn import DQNConfig
import tempfile
```

---

<sup>1</sup> Обратите внимание, что если выполнить блокнот Jupyter этой главы (<https://oreil.ly/KEhGx>) в облаке, то завершение процесса тренировки может занять некоторое время.

```
temp = tempfile.mkdtemp() ①  
  
trainer = (  
    DQNConfig()  
    .environment(env=CurriculumEnv, env_task_fn=curriculum_fn) ②  
    .offline_data(output=temp) ③  
    .build()  
)  
  
for i in range(15):  
    trainer.train()
```

- ① Создать временный файл *temp*, чтобы хранить там тренировочные данные для их последующего использования.
- ② Установить CurriculumEnv в качестве среды в части `environment` нашей конфигурации и назначить функцию `curriculum_fn` свойству `env_task_fn`.
- ③ Использовать метод `offline_data` для сохранения выходных данных во временной папке *temp*.

Выполнив этот алгоритм, вы должны увидеть, как с течением времени трудность задания будет увеличиваться, тем самым сначала предоставляя алгоритму простые примеры, чтобы тот мог на них учиться и переходить к более трудным заданиям.

Усвоение учебной программы – отличный технический прием, о котором следует знать, и библиотека RLlib позволяет легко его внедрять в свои эксперименты посредством API учебной программы, который мы только что рассмотрели.

## Работа с офлайновыми данными

В предыдущем примере усвоения учебной программы мы сохраняли тренировочные данные во временной папке. Самое интересное, что из главы 3 вы уже знаете, что в Q-обучении можно сначала собирать данные об опыте, а затем решать, когда их использовать на этапе тренировки. Такое разделение сбора данных и тренировки открывает целый ряд возможностей. Например, вероятно, у вас есть хорошая эвристика, которая может решить вашу задачу неидеальным, но разумным образом. Либо у вас есть записи о взаимодействии человека со средой, демонстрирующие, как решать задачу, на примере.

Тема сбора данных об опыте для последующей тренировки часто называется работой с *оффлайновыми данными*. Такие данные называются «оффлайновыми», потому что они не генерируются напрямую политикой, взаимодействующей со средой в онлайновом режиме. Алгоритмы, которые не опираются на тренировку на данных, получаемых на выходе из их собственной политики, называются *внеполитическими алгоритмами*<sup>1</sup>, и Q-обучение, в частности глубокая Q-сеть (DQN), является лишь одним из таких примеров. Алгоритмы, которые не обладают этим свойством, называются политическими алгоритмами.

---

<sup>1</sup> Англ. off-policy algorithms.

мами<sup>1</sup>. Другими словами, внеполитические алгоритмы могут использоваться для тренировки на офлайновых данных<sup>2</sup>.

Для использования данных, которые мы сохранили во временной папке `temp`, можно создать новую конфигурацию `DQNConfig`, которая на входе принимает эту папку. Мы также установим значение `explore` равным `False`, поскольку в процессе тренировки мы хотим лишь эксплуатировать ранее собранные данные – алгоритм не будет выполнять разведку в соответствии со своей собственной политикой.

Использование результирующего алгоритма библиотеки `RLLib` работает точно так же, как и раньше, что мы и демонстрируем, тренируя его в течение 10 итераций и затем его оценивая:

```
imitation_algo = (
    DQNConfig()
    .environment(env=AdvancedEnv)
    .evaluation(off_policy_estimation_methods={})
    .offline_data(input_=temp)
    .exploration(explore=False)
    .build())

for i in range(10):
    imitation_algo.train()

imitation_algo.evaluate()
```

Обратите внимание, что мы назвали алгоритм имитационным алгоритмом (`imitation_algo`). Это связано с тем, что данная процедура тренировки призвана *имитировать* поведение, отраженное в данных, которые мы собрали ранее. Поэтому в обучении с подкреплением данный тип самообучения посредством демонстрации часто называют *имитационным самообучением*, или *клонированием поведения*.

## Другие продвинутые темы

Прежде чем завершить эту главу, давайте взглянем на несколько других продвинутых тем, которые библиотека `RLLib` может предложить. Вы уже увидели гибкость библиотеки `RLLib`: вы можете работать с целым рядом разных сред, конфигурировать свои эксперименты, тренировать по учебной программе и применять имитационное самообучение. Этот раздел дает вам представление о том, что можно делать еще.

С помощью библиотеки `RLLib` можно полностью адаптировать используемые под капотом модели и политики. Если вы раньше работали с глубоким обучением, то знаете, насколько важно иметь хорошую модельную архитектуру. В обучении с подкреплением это зачастую не так важно, как

---

<sup>1</sup> Англ. *on-policy algorithms*.

<sup>2</sup> Обратите внимание, что библиотека `RLLib` также имеет широкий спектр политических алгоритмов, таких как `PPO`.

в контролируемом обучении, но все равно является жизненно важной частью успешного проведения продвинутых экспериментов.

Кроме того, можно изменять способ предобработки своих наблюдений, предоставляя конкретно-прикладные предобработчики. В наших простых примерах с лабиринтом ничего не требовалось предварительно обрабатывать, но при работе с изображениями или видеоданными предобработка часто является решающим шагом.

В наш средовой класс `AdvancedEnv` были введены состояния, которых следует избегать. Агентам пришлось учиться это делать, но в библиотеке `RLLib` есть функциональная возможность, позволяющая автоматически их избегать с помощью так называемых *параметрических пространств действий*. Грубо говоря, можно «маскировать» все нежелательные действия из пространства действий по каждому моменту времени. В некоторых случаях также может потребоваться наличие переменных пространств наблюдений, что тоже полностью поддерживается библиотекой `RLLib`.

Мы вкратце затронули тему офлайновых данных. Библиотека `RLLib` имеет полноценный Python API для чтения и записи данных об опыте, которые можно использовать в различных ситуациях.

Для простоты мы здесь работали исключительно с глубокой Q-сетью (DQN), но библиотека `RLLib` обладает впечатляющим набором тренируемых алгоритмов. Приведем лишь один из них – алгоритм MARWIL. Это сложный гибридный алгоритм, с помощью которого можно выполнять имитационное самообучение на офлайновых данных, а также примешивать обычную тренировку на данных, сгенерированных в «онлайновом» режиме.

## Резюме

В этой главе вы познакомились с подборкой интересных функциональных возможностей библиотеки `RLLib`. Мы рассмотрели тренировку многоагентных сред, работу с офлайновыми данными, генерируемыми другим агентом, конфигурирование клиент-серверной архитектуры для разделения симуляций и тренировки алгоритма обучения с подкреплением, а также использование процедуры усвоения учебной программы, чтобы определять все более сложные задания.

Мы также дали краткий обзор главных концепций, лежащих в основе библиотеки `RLLib`, и того, как использовать ее интерфейс командной строки (CLI) и Python API. В частности, мы показали, как конфигурировать алгоритмы и среды библиотеки `RLLib` в соответствии с вашими потребностями. Поскольку мы рассмотрели лишь малую часть возможностей библиотеки `RLLib`, то настоятельно рекомендуем вам ознакомиться с ее документацией и обследовать ее API<sup>1</sup>.

В следующей главе вы узнаете, как настраивать гиперпараметры ваших моделей и политик библиотеки `RLLib` с помощью библиотеки Ray Tune.

---

<sup>1</sup> См. <https://oreil.ly/OmQYE>.

# Глава 5

---

## Гиперпараметрическая оптимизация с использованием библиотеки Ray Tune

В главе 4 вы научились разрабатывать и проводить различные эксперименты по обучению с подкреплением. Проведение таких экспериментов бывает дорогостоящим как с точки зрения вычислительных ресурсов, так и времени, необходимого для их проведения. Эти затраты только увеличиваются, по мере того как вы переходите к более сложным заданиям, поскольку маловероятно, что у вас получится просто выбрать готовый к работе алгоритм и выполнить его, получив хороший результат. Другими словами, в какой-то момент вам нужно будет настроить гиперпараметры ваших алгоритмов, чтобы получить наилучшие результаты. Как мы увидим в этой главе, настраивать модели машинного обучения непросто, но библиотека Ray Tune представляет собой прекрасный выбор, который поможет вам справиться с этим заданием.

Библиотека Ray Tune – это мощный инструмент гиперпараметрической оптимизации<sup>1</sup>. Она не только по умолчанию работает распределенно (и работает в любой другой библиотеке фреймворка Ray, обсуждаемой в этой книге), но и является одной из самых многофункциональных библиотек гиперпараметрической оптимизации, имеющейся на сегодняшний день. В довершение всего библиотека Tune интегрируется с несколькими наиболее известными библиотеками гиперпараметрической оптимизации, такими как Hyperopt, Optuna и многими другими. Это делает ее идеальным кандидатом для распределенных экспериментов с гиперпараметрической оптимизацией, независимо от того, используете ли вы другие библиотеки или начинаете с чистого листа.

---

<sup>1</sup> Англ. hyperparameter optimization (HPO). – Прим. перев.

В этой главе сначала мы чуть-чуть подробнее рассмотрим причины, по которым оптимизировать гиперпараметры трудно, и способы, при помощи которых вы могли бы реализовывать гиперпараметрическую оптимизацию наивно сами с использованием фреймворка Ray. Затем мы познакомим вас с ключевыми концепциями библиотеки Ray Tune и с тем, как ее использовать для настройки моделей библиотеки RLlib, построенных в предыдущей главе. В заключение мы также рассмотрим способы применения библиотеки Tune в задачах контролируемого обучения, используя такие фреймворки, как Keras. Попутно продемонстрируем интеграцию библиотеки Tune с другими библиотеками гиперпараметрической оптимизации и познакомим вас с некоторыми ее более продвинутыми функциональными возможностями.

## Настройка гиперпараметров

Давайте вкратце рассмотрим основы гиперпараметрической оптимизации. Если вы с ней знакомы, то можете пропустить этот раздел, но поскольку мы также обсуждаем аспекты распределенной гиперпараметрической оптимизации, вам все же будет полезно продолжить чтение. Как всегда, блокнот Jupyter этой главы можно найти в репозитории книги на GitHub<sup>1</sup>.

В нашем первом эксперименте по обучению с подкреплением, представленном в главе 3, мы определили очень простой алгоритм Q-обучения, внутренние значения состояний–действий которого обновлялись в соответствии с явным правилом обновления. После инициализации мы ни разу не касались этих модельных параметров напрямую; они были усвоены алгоритмом. Напротив, при настройке алгоритма перед тренировкой мы специально выбрали параметры веса (`weight`) и фактора дисконтирования (`discount_factor`). В той главе мы вам не рассказали о том, как решили установить значения этих параметров; мы просто согласились с тем, что они были достаточно хороши для решения данной задачи.

Аналогичным образом в главе 4 мы инициализировали алгоритм библиотеки RLlib конфигурацией (`config`), в которой для алгоритма глубокой Q-сети (DQN) использовалось два работника розыгрыша путем установки `num_rollout_workers=2`. Подобные параметры называются гиперпараметрами, и правильный вариант их значений может иметь решающее влияние на успешность экспериментов. Область гиперпараметрической оптимизации посвящена эффективному поиску таких хороших вариантов.

## Разработка примера случайного поиска с помощью фреймворка Ray

Гиперпараметры, такие как вес (`weight`) и фактор дисконтирования (`discount_factor`) нашего алгоритма Q-обучения, являются непрерывными параметрами,

---

<sup>1</sup> См. <https://oreil.ly/-afF8>.

поэтому протестировать все их комбинации невозможно. Более того, выбор этих параметров, возможно, не будет независимым друг от друга. Если мы хотим, чтобы они были выбраны за нас, нам также нужно указать *диапазон значений* каждого из них (в данном случае оба гиперпараметра должны быть в диапазоне от 0 до 1). Так как же определять хорошие или даже оптимальные гиперпараметры?

Давайте рассмотрим пример, в котором реализован наивный, но эффективный подход к настройке гиперпараметров. Этот пример также позволит ввести некоторую терминологию, которой мы будем пользоваться позже. Ключевая идея заключается в том, что можно попытаться *брать случайные образцы* гиперпараметров, выполнять алгоритм для каждого образца, а затем на основе результатов выбирать наилучший вариант выполнения. Но, отдавая должное теме данной книги, мы не просто хотим выполнить это в по-следовательном цикле – мы хотим вычислять наши прогоны параллельно, используя фреймворк Ray.

В целях упрощения задачи мы вернемся к нашему простому алгоритму Q-обучения из главы 3. Мы определили сигнатуру главной функции тренировки как `train_policy(env, num_episodes=10000, weight=0.1, discount_factor=0.9)`. Следовательно, мы можем настроить параметры `weight` и `discount_factor` нашего алгоритма, передавая в функцию `train_policy` разные значения и глядя на результативность алгоритма. Для этого давайте определим для указанных гиперпараметров так называемое *пространство поиска*. Мы бере-рем образцы значений обоих рассматриваемых параметров равномерно между 0 и 1, в общей сложности 10 вариантов.

Бот как это выглядит:

```
import random

search_space = []

for i in range(10):
    random_choice = {
        'weight': random.uniform(0, 1),
        'discount_factor': random.uniform(0, 1)
    }
    search_space.append(random_choice)
```

Далее мы определяем *целевую функцию*, или просто *цель*<sup>1</sup>. Роль целевой функции заключается в оценивании результативности указанного набора гиперпараметров для желаемого задания. В нашем случае мы хотим натре-нировать алгоритм обучения с подкреплением и оценить натренированную политику. Напомним, что именно для этой цели в главе 3 мы также определи-ли функцию `evaluate_policy`. Функция `evaluate_policy` была определена, что-

---

<sup>1</sup> Для справки: целевая функция (objective function, где objective – это плановое задание, ориентир) – это либо функция потери, и тогда она должна быть миними-зирована, либо ее противоположность (в определенных областях она называется функцией вознаграждения, прибыли, полезности, пригодности и т. д.), в каковом случае она должна быть максимизирована. – Прим. перев.

бы возвращать среднее число шагов, которое требовалось агенту для достижения цели в опорной среде лабиринта. Другими словами, мы хотим найти набор гиперпараметров, который минимизирует результат целевой функции. В целях параллелизации целевой функций мы будем использовать декоратор `ray.remote`, чтобы сделать целевую функцию (`objective`) заданием Ray:

```
import ray

@ray.remote
def objective(config): ❶
    environment = Environment()
    policy = train_policy(❷
        environment,
        weight=config["weight"],
        discount_factor=config["discount_factor"])
    )
    score = evaluate_policy(environment, policy) ❸
    return [score, config] ❹
```

- ❶ Передать словарь с образцом гиперпараметров в целевую функцию.
- ❷ Натренировать политику обучения с подкреплением, используя выбранные гиперпараметры.
- ❸ После этого можно оценить политику, чтобы извлечь балл, который мы хотим минимизировать.
- ❹ Вернуть балл и подборку гиперпараметров для последующего анализа.

Наконец, мы можем выполнить целевую функцию в параллельном режиме с помощью фреймворка Ray, совершив итеративный обход пространства поиска и собрав результаты:

```
result_objects = [objective.remote(choice) for choice in search_space]
results = ray.get(result_objects)

results.sort(key=lambda x: x[0])
print(results[-1])
```

Фактические результаты этого прогона с гиперпараметрами не очень интересны, поскольку задача легко решается (большинство прогонов будут возвращать оптимум из восьми шагов, независимо от выбранных гиперпараметров). Здесь более интересно то, насколько легко целевая функция параллелизуется с помощью фреймворка Ray. На самом деле мы хотели бы призвать вас переписать приведенный выше пример, просто прокручивая пространство поиска и вызывая целевую функцию для каждого образца, чтобы подтвердить, насколько мучительно медленным может быть такой последовательный цикл.

В концептуальном плане три предпринятых нами для выполнения этого примера шага являются репрезентативными в том, что они показывают, как гиперпараметрическая настройка работает в целом. Сначала определяется пространство поиска, затем целевая функция, и, наконец, проводится анализ, чтобы найти наилучшие гиперпараметры. В гиперпараметрической оптимизации об одном оценивании целевой функции в расчете на образец

гиперпараметров принято говорить как о *попытке*<sup>1</sup>, и все попытки формируют основу анализа. То, как образцы параметров берутся из пространства поиска (в нашем случае случайным образом), зависит от *алгоритма поиска*. На практике отыскание хороших гиперпараметров – это легче сказать, чем сделать, поэтому давайте подробнее рассмотрим причины, по которым эта задача является такой трудной.

## В чем трудность гиперпараметрической оптимизации?

Если взглянуть на приведенный выше пример шире, то можно увидеть несколько тонкостей, связанных с обеспечением хорошей работы процесса гиперпараметрической настройки. Вот краткий обзор наиболее важных из них:

- пространство поиска может состоять из большого числа гиперпараметров. Эти параметры могут иметь разные типы данных и диапазоны. Некоторые параметры могут быть коррелированы или даже зависеть от других. Взятие хороших кандидатов из сложных многомерных пространств – трудная работа;
- подбор параметров в случайном порядке может работать на удивление хорошо, но это не всегда самый лучший вариант. В целом, для того чтобы найти наилучшие параметры, приходится тестировать более сложные алгоритмы поиска;
- в частности, даже если параллелизовать гиперпараметрический поиск, как мы только что сделали, выполнение одной целевой функции может занимать много времени. И следовательно, нельзя позволять себе выполнять слишком много поисковых запросов в целом. Например, тренировка нейронных сетей может занимать несколько часов, поэтому гиперпараметрический поиск должен быть эффективным;
- при распределении поиска необходимо иметь в наличии достаточное количество вычислительных ресурсов, чтобы эффективно выполнять поиск по целевой функции. Например, для в меру быстрого вычисления целевой функции может понадобиться графический процессор, поэтому все поисковые прогоны должны иметь доступ к графическому процессору. Выделение ресурсов, необходимых для каждой попытки, имеет решающее значение для ускорения поиска;
- нужны удобные инструменты для экспериментов по гиперпараметрической оптимизации, такие как досрочная остановка неуспешных прогонов, сохранение промежуточных результатов, перезапуск из предыдущих попыток или приостановка и возобновление прогонов.

Библиотека Ray Tune, являясь зрелым распределенным фреймворком гиперпараметрической оптимизации, затрагивает все эти темы и предоставляет простой интерфейс для проведения экспериментов по гиперпараметриче-

---

<sup>1</sup> Англ. trial. – Прим. перев.

ской настройке. Прежде чем взглянуть на работу библиотеки Tune, давайте перепишем наш предыдущий пример, чтобы задействовать библиотеку Tune.

## ВВЕДЕНИЕ В БИБЛИОТЕКУ TUNE

В качестве первого впечатления от библиотеки Tune портирование нашей наивной реализации случайного поиска на основе инструментария Ray Core в библиотеку Tune выполняется просто и подчиняется тем же трем шагам, что и раньше. Сначала мы определяем пространство поиска, но на этот раз вместо библиотеки `random` используем функцию `tune.uniform`:

```
from ray import tune

search_space = {
    "weight": tune.uniform(0, 1),
    "discount_factor": tune.uniform(0, 1),
}
```

Далее можно определить целевую функцию, которая выглядит почти так же, как и раньше. Мы сконструировали ее, как показано ниже. Единственное, отличия заключаются в том, что на этот раз мы возвращаем балл в виде словаря, и нам не нужен декоратор `ray.remote`, потому что библиотека Tune возьмет на себя распределение этой целевой функции за нас на внутреннем уровне:

```
def tune_objective(config):
    environment = Environment()
    policy = train_policy(
        environment,
        weight=config["weight"],
        discount_factor=config["discount_factor"])
    score = evaluate_policy(environment, policy)

    return {"score": score}
```

Определив функцию `tune_objective`, можно передать ее функции `tune.run` вместе с заданным нами пространством поиска. По умолчанию библиотека Tune выполнит для вас случайный поиск, но, как вы скоро увидите, можно указывать и другие алгоритмы поиска<sup>1</sup>. Вызов функции `tune.run` генерирует попытки случайного поиска для вашей целевой функции и возвращает объект `analysis`, содержащий информацию о гиперпараметрическом поиске. Вызвав функцию `get_best_config` и указав аргументы `metric` и `mode`

---

<sup>1</sup> В библиотеке Tune используется та же ресурсная модель, что и в инструментарии Ray Core. По умолчанию каждый прогон `tune_objective` будет исполняться на другом ядре центрального процессора. Если хотите, то также можете указывать (дробный) графический процессор, который будет использоваться в каждой попытке.

(мы хотим минимизировать балл), можно получить наилучшие найденные гиперпараметры:

```
analysis = tune.run(tune_objective, config=search_space)
print(analysis.get_best_config(metric="score", mode="min"))
```

Этот краткий пример охватывает основы библиотеки Tune, но здесь еще много чего предстоит разобрать. Функция `tune.run` отличается достаточной мощностью и принимает много аргументов, давая вам возможность конфигурировать свои прогоны. Для того чтобы разобраться в этих разных опциях конфигурации, сначала нужно ознакомиться с ключевыми концепциями библиотеки Tune.

## Принцип работы библиотеки Tune

В целях эффективной работы с библиотекой Tune необходимо понимать шесть ключевых концепций, четыре из которых вы использовали в предыдущем примере. Вот краткий обзор компонентов библиотеки Ray Tune и того, как вам следует о них думать.

### *Пространства поиска*

Пространства поиска определяют то, какие параметры следует отбирать. Они описывают диапазон значений каждого параметра и способ взятия из них образцов. Задаются как словари и используют принадлежащие библиотеке Tune функции взятия образцов, чтобы детализировать допустимые значения гиперпараметров. Вы уже встречали функцию `tune.uniform`, но есть еще много других вариантов на выбор<sup>1</sup>.

### *Тренируемые объекты*

Тренируемый объект (`Tunable`) – это формальное представление библиотекой Tune целевой функции, которую вы хотите «настроить». В библиотеке Tune также есть API, основанный на классах, но в этой книге мы будем использовать только API, основанный на функциях. Для нас `Tunable` – это функция с единственным аргументом: пространством поиска, который сообщает библиотеке Tune о баллах. Самый простой способ сообщить о балле – вернуть словарь с интересующим вас значением.

### *Попытки*

После запуска `tune.run(...)` библиотека Tune настроит попытки и запланирует их исполнение в вашем кластере. Попытка содержит всю необходимую информацию об одном прогоне целевой функции с учетом набора гиперпараметров.

### *Анализ*

В завершение вызова функции `tune.run` возвращается объект `ExperimentAnalysis` с результатами всех попыток. Этот объект можно использовать для детализации результатов попыток.

---

<sup>1</sup> См. <https://oreil.ly/6beij>.

### Алгоритмы поиска

Библиотека Tune поддерживает большое разнообразие алгоритмов поиска, которые лежат в основе настройки гиперпараметров. До сего момента вы неявно сталкивались с алгоритмом поиска, применяемым в библиотеке Tune по умолчанию, который отбирает гиперпараметры из пространства поиска в случайном порядке.

### Планировщики

Последним, решающим компонентом эксперимента по настройке является планировщик. Планировщики составляют расписание и исполняют то, что отбирает алгоритм поиска. По умолчанию библиотека Tune планирует попытки, отобранные вашим алгоритмом поиска, по принципу «первым вошел – первым вышел» (FIFO). На практике планировщики можно рассматривать как способ ускорения ваших экспериментов, например путем досрочной остановки неуспешных попыток.

На рис. 5.1 резюмированы приведенные выше главнейшие компоненты библиотеки Tune и их взаимосвязи.



**Рис. 5.1** ♦ Ключевые компоненты библиотеки Ray Tune



В этой главе мы используем функцию `tune.gup` исключительно для иллюстрации функциональности библиотеки Tune. В рамках релиза Ray 2.0 в библиотеку Tune также был добавлен API под названием Tuner как часть инструментария Ray AIR, о котором вы узнаете больше в главе 7 и будете использовать в рамках Ray AIR в главе 10.

На момент написания этой книги API функции `tune.gup` по-прежнему остается более зрелым. Например, эксперименты с использованием вызова `tune.gup(...)` возвращают объект `ExperimentAnalysis`, являющийся мощным инструментом анализа ваших результатов. Вместо этого аналогичные вызовы с использованием API Tuner возвращают так называемый объект `ResultGrid`. В долгосрочной перспективе объект `ResultGrid` станет преемником объекта `ExperimentAnalysis`, но пока что он еще не достиг паритета характеристик.

Рекомендуем ознакомиться с документацией по API библиотеки Tune, чтобы узнать все подробности по этой теме<sup>1</sup>.

Обратите внимание, что на внутреннем уровне прогоны Tune запускаются в процессе-драйвере вашего кластера Ray, порождающем несколько

<sup>1</sup> См. <https://oreil.ly/pITtJ>.

процессов-работников (использующих акторов Ray), которые исполняют индивидуальные попытки вашего эксперимента по гиперпараметрической оптимизации. Определенные в драйвере тренируемые объекты должны отправляться работникам, а результаты попыток должны сообщаться драйверу, выполняющему вызов `tune.run(...)`.

Пространства поиска, тренируемые объекты, попытки и анализ не нуждаются в дополнительном объяснении, и в остальной части этой главы мы увидим больше примеров каждого из этих компонентов. Но алгоритмы поиска, сокращенно *поисковики*, и планировщики нуждаются в относительно большей проработке.

## Алгоритмы поиска

Все предоставляемые библиотекой Tune продвинутые алгоритмы поиска и множество сторонних библиотек гиперпараметрической оптимизации, с которыми она интегрируется, подпадают под зонтик *байесовой оптимизации*. К сожалению, углубление в подробности конкретных байесовых алгоритмов поиска выходит далеко за рамки этой книги. Базовая идея заключается в обновлении своих убеждений о том, какие диапазоны гиперпараметров стоит разведывать, основываясь на результатах предыдущих попыток. Методы, в которых этот принцип используется, позволяют принимать более информированные решения и, следовательно, имеют тенденцию быть эффективнее, чем взятие образцов параметров независимым образом (например, случайно).

Помимо базового случного поиска, который мы уже встречали выше, и *поиска в параметрической решетке*, который выбирает гиперпараметры из предопределенной «решетки» вариантов, библиотека Tune интегрируется с широким спектром байесово-оптимационных поисковиков. Например, библиотека Tune интегрируется с популярными библиотеками Hyperopt и Optuna<sup>1</sup>, и посредством обеих этих библиотек с помощью библиотеки Tune можно использовать популярный поисковик под названием Древесно-структурированный парценовский оценщик<sup>2</sup>. Мало того, библиотека Tune также интегрируется с такими инструментами, как Ax, BlendSearch, FLAML, Dragonfly, scikit-Optimize, Bayesian optimization, HpBandSter, Nevergrad, ZOOpt, SigOpt и НЕВО. Если вам требуется провести эксперименты по гиперпараметрической оптимизации с любым из этих инструментов в кластере или вы хотите легко переключаться между ними, то библиотека Tune – это то, что вам нужно.

В целях конкретизации ситуации давайте перепишем приведенный ранее базовый пример случного поиска на основе библиотеки Tune под использование библиотеки байесовой оптимизации (Bayesian optimization). Для

<sup>1</sup> В программном обеспечении с открытым исходным кодом важно знать ответственных за поддержание интеграции. Мы обсудим это подробнее в главе 11, которая охватывает экосистему Ray в целом. В случае библиотеки Tune, среди перечисленных здесь интеграций, интеграции Hyperopt и Optuna поддерживаются коллективом разработчиков библиотеки Ray Tune; остальные спонсируются сообществом.

<sup>2</sup> Англ. Tree-structured Parzen Estimator (TPE). – Прим. перев.

этого сначала проверьте, чтобы эта библиотека была установлена в вашу среду Python, например с помощью команды `pip install bayesian-optimization`:

```
from ray.tune.suggest.bayesopt import BayesOptSearch

algo = BayesOptSearch(random_search_steps=4)

tune.run(
    tune_objective,
    config=search_space,
    metric="score",
    mode="min",
    search_alg=algo,
    stop={"training_iteration": 10},
)
```

Обратите внимание, что вначале мы «разогреваем» нашу байесову оптимизацию четырьмя случайными шагами и явным образом останавливаем (`stop`) прогоны попыток после 10 итераций тренировки.

Поскольку мы не просто случайно отбираем параметры с помощью `BayesOptSearch`, используемый в прогоне библиотеки Tune алгоритм поиска (`search_alg`) должен знать, для какой метрики (`metric`) проводить оптимизацию и следует ли минимизировать либо оптимизировать эту метрику. Как мы уже говорили ранее, мы хотим получить минимальный ("min") балл ("score").

## Планировщики

Далее давайте займемся использованием *планировщиков попыток* в библиотеке Tune с целью придания вашим прогонам большей эффективности. Мы также используем этот раздел, чтобы ввести несколько иной способ информирования библиотеки Tune о ваших метриках внутри целевой функции.

Итак, давайте предположим, что, вместо того чтобы вычислять балл напрямую, как мы делали в предыдущих примерах, мы вычисляем промежуточный балл в цикле. Такая ситуация часто возникает в сценариях контролируемого машинного обучения при тренировке модели в течение нескольких итераций (конкретные применения тому мы увидим в разделе «Машинное обучение с помощью библиотеки Tune» на стр. 141). При правильном подборе гиперпараметров этот промежуточный балл может застаиваться задолго до цикла, в котором он вычисляется. Другими словами, если мы не видим достаточноного количества постепенных изменений, то почему бы не остановить попытку досрочно? Это как раз один из случаев, для которых в библиотеке Tune созданы планировщики.

Ниже приведен краткий пример такой целевой функции. Это игрушечный пример, но он поможет нам думать об оптимальных гиперпараметрах, которые мы хотим получить от библиотеки Tune, с большей легкостью, чем если бы мы начали со сложного примера:

```
def objective(config):
    for step in range(30): ❶
        score = config["weight"] * (step ** 0.5) + config["bias"]
```

```
tune.report(score=score) ❷

search_space = {"weight": tune.uniform(0, 1),
                "bias": tune.uniform(0, 1)}
```

- ❶ Нередко вам, возможно, потребуется вычислять промежуточные баллы, например в «цикле тренировки».
- ❷ Функцию `tune.report` можно использовать, чтобы сообщать библиотеке Tune об этих промежуточных баллах.

Здесь мы хотим минимизировать балл, который сводится к квадратному корню из положительного числа, умноженного на вес (`weight`) плюс член смещения (`bias`). Ясно, что оба этих гиперпараметра должны быть как можно меньше, чтобы минимизировать балл (`score`) для любого положительного  $x$ . Учитывая, что функция квадратного корня «разглаживает», нам, возможно, не придется вычислять все 30 проходов по циклу, чтобы найти достаточно хорошие значения двух гиперпараметров. Если бы каждое вычисление балла занимало час, то досрочная остановка могла бы значительно ускорить проведение экспериментов.

Давайте проиллюстрируем эту идею, применив популярный алгоритм Hyperband в качестве планировщика попыток. Указанному планировщику необходимо передать метрику и режим (опять же, мы минимизируем наш балл). Мы также обеспечиваем выполнение прогона для 10 образцов, чтобы не останавливаться преждевременно:

```
from ray.tune.schedulers import HyperBandScheduler

scheduler = HyperBandScheduler(metric="score", mode="min")

analysis = tune.run(
    objective,
    config=search_space,
    scheduler=scheduler,
    num_samples=10,
)

print(analysis.get_best_config(metric="score", mode="min"))
```

Обратите внимание, что в данном случае мы не указали алгоритм поиска, а значит, Hyperband будет работать на параметрах, отобранных случайным поиском. Вместо этого мы также могли бы скомбинировать этот планировщик с еще одним алгоритмом поиска. Это позволило бы подбирать более оптимальные гиперпараметры попыток и останавливать неуспешные попытки досрочно. При этом обратите внимание, что не каждый планировщик можно комбинировать с алгоритмами поиска. Обратитесь к матрице совместимости планировщиков Tune, чтобы получить дополнительную информацию<sup>1</sup>.

В завершение обсуждения этой темы, помимо алгоритма поиска Hyperband, библиотека Tune включает в себя распределенные реализации алгорит-

---

<sup>1</sup> См. <https://docs.ray.io/en/latest/tune/key-concepts.html?highlight=scheduler%20compatibility%20matrix#tune-schedulers>.

мов досрочной остановки, таких как Медианное правило остановки<sup>1</sup>, ASHA, Популяционно-ориентированная тренировка<sup>2</sup> и Популяционно-ориентированные бандиты<sup>3</sup>.

## Конфигурирование и выполнение библиотеки Tune

Прежде чем рассматривать более конкретные примеры машинного обучения с использованием библиотеки Ray Tune, давайте углубимся в несколько полезных тем, которые помогут вам извлечь больше пользы из экспериментов с данной библиотекой, таких как надлежащая задействованность ресурсов, остановка и возобновление попыток, добавление функций обратного вызова к прогонам библиотеки Tune или определение конкретно-прикладных и условных пространств поиска.

### Детализация ресурсов

По умолчанию каждая попытка библиотеки Tune будет выполняться на одном центральном процессоре и задействовать столько центральных процессоров, сколько доступно для конкурентных попыток. Например, если вы выполняете библиотеку Tune на ноутбуке с 8 центральными процессорами, то любой эксперимент, вычисленный до этого в данной главе, вызовет восемь конкурентных попыток и выделит по одному процессору для каждой попытки. Изменением этого поведения можно управлять с помощью аргумента `resources_per_trial` прогона Tune.

Кроме того, можно определять число используемых в каждой попытке графических процессоров. Плюс к этому библиотека Tune позволяет использовать *дробные ресурсы*, то есть можно делиться ресурсами между попытками. Так, предположим, что у вас есть машина с 12 центральными процессорами и 2 графическими процессорами, и для своей целевой функции вы запрашиваете следующие ниже ресурсы:

```
from ray import tune

tune.run(
    objective,
    config=search_space,
    num_samples=10,
    resources_per_trial={"cpu": 2, "gpu": 0.5}
)
```

Это означает, что библиотека Tune может планировать и исполнять на вашем компьютере до четырех конкурентных попыток, поскольку это исчерпывает задействованность графического процессора на данной машине (в то время как у вас все еще будет четыре незанятых центральных процессора

---

<sup>1</sup> Англ. Median Stopping Rule.

<sup>2</sup> Англ. Population Based Training (PBT).

<sup>3</sup> Англ. Population Based Bandits (PB2).

для других заданий). Если хотите, вы также можете указать используемый попыткой объем памяти ("memogy"), передав число байтов в `resources_per_trial`. Также обратите внимание на то, что если вам нужно ограничить число конкурентных попыток в явной форме, то это можно сделать, передав в вызове `tune.run(...)` параметр `max_concurrent_trials`. Если в приведенном выше примере вы хотите всегда держать один графический процессор доступным для других заданий, то можете ограничить число конкурентных попыток до двух, установив `max_concurrent_trials = 2`.

Обратите внимание, что все, что мы только что проиллюстрировали в отношении ресурсов на одной машине, естественным образом распространяется на любой кластер Ray и его доступные ресурсы. В любом случае фреймворк Ray всегда будет пытаться планировать следующие попытки, но он будет ждать и проверять наличие достаточного количества ресурсов, прежде чем их исполнять.

## Функции обратного вызова и метрики

Если вы провели некоторое время за обследованием результатов прогонов Tune, которые мы запустили в этой главе, то наверняка заметили, что каждая попытка по умолчанию содержит много информации, такой как идентификатор попытки, дата ее исполнения и многое другое. Самое интересное, что библиотека Tune не только позволяет адаптировать метрики, о которых вы хотите сообщать, вы также можете вклиниваться в функцию `tune.run`, представляя *функции обратного вызова*. Давайте вычислим краткий и репрезентативный пример, в котором выполняется и то, и другое.

Слегка видоизменив предыдущий пример, допустим, мы хотим зарегистрировать конкретное сообщение всякий раз, когда попытка возвращает результат. Для этого нужно реализовать лишь метод `on_trial_result` на объекте `Callback` из пакета `ray.tune`<sup>1</sup>. Вот как это выглядело бы для целевой функции, которая сообщает балл (`score`):

```
from ray import tune
from ray.tune import Callback
from ray.tune.logger import pretty_print

class PrintResultCallback(Callback):
    def on_trial_result(self, iteration, trials, trial, result, **info):
        print(f"Результат попытки {trial} в итерации {iteration} "
              f": {result['score']}")

def objective(config):
    for step in range(30):
        score = config["weight"] * (step ** 0.5) + config["bias"]
        tune.report(score=score, step=step, more_metrics={})
```

---

<sup>1</sup> Если вы хотите узнать подробнее о том, как использовать функции обратного вызова в библиотеке Tune или создавать свои собственные функции обратного вызова, то рекомендуем ознакомиться с руководством пользователя по функциям обратного вызова и метрикам в библиотеке Tune (<https://oreil.ly/1CDj2>).

Обратите внимание, что, помимо балла, мы также информируем библиотеку Tune о шаге (`step`) и дополнительных метриках (`more_metrics`). По сути дела, вы могли бы выставлять там любую другую метрику, которую хотели бы отслеживать, и библиотека Tune добавила бы ее в свои метрики попытки. Вот как вы могли бы провести эксперимент с нашей конкретно-прикладной функцией обратного вызова и распечатать конкретно-прикладные метрики, которые мы только что определили:

```
search_space = {"weight": tune.uniform(0, 1),
                 "bias": tune.uniform(0, 1)}

analysis = tune.run(
    objective,
    config=search_space,
    mode="min",
    metric="score",
    callbacks=[PrintResultCallback()])

best = analysis.best_trial
print(pretty_print(best.last_result))
```

Выполнение этого фрагмента исходного кода приведет к следующим ниже результатам (дополнительным к тому, что вы увидите при любом другом прогоне Tune). Обратите внимание, что здесь необходимо явно указать режим (`mode`) и метрику (`metric`), чтобы библиотека Tune знала о том, что мы подразумеваем под наилучшим результатом (`best_result`). Прежде всего вы должны видеть результат нашей функции обратного вызова, по мере того как выполняются попытки:

```
...
Результат попытки objective_85955_00000 в итерации 57: 1.5379782083952644
Результат попытки objective_85955_00000 в итерации 58: 1.5539087627537493
Результат попытки objective_85955_00000 в итерации 59: 1.569535794562848
Результат попытки objective_85955_00000 в итерации 60: 1.5848760187255326
Результат попытки objective_85955_00000 в итерации 61: 1.5999446700996236
...
```

Затем, в самом конце программы, мы печатаем метрики наилучшей попытки, включающей в себя три заданные нами конкретно-прикладные метрики. В следующем ниже результате несколько дефолтных метрик было опущено, чтобы сделать его более удобочитаемым. Мы рекомендуем вам выполнить подобный пример самостоятельно, в частности для того, чтобы привыкнуть к чтению результатов попыток Tune (которые могут быть немного перегружены информацией из-за их конкурентной природы):

```
Result logdir: /Users/maxpumperla/ray_results/objective_2022-05-23_15-52-01
...
done: true
experiment_id: ea5d89c2018f483183a005a1b5d47302
experiment_tag: 0_bias=0.73356,weight=0.16088
hostname: mac
iterations_since_restore: 30
```

```
more_metrics: []
score: 1.5999446700996236
step: 29
trial_id: '85955_00000'
...
```

Мы использовали `on_trial_result` в качестве примера метода в реализации конкретно-прикладного класса `Callback`, но у вас есть много других полезных опций, которые относительно понятны сами по себе. Не очень практично перечислять их все здесь, но некоторые особенно полезные методы обратного вызова таковы: `on_trial_start`, `on_trial_error`, `on_experiment_end` и `on_checkpoint`. Последний намекает на важный аспект прогонов в библиотеке Tune, который мы обсудим далее.

## **Контрольные точки, остановка и возобновление**

Чем большему числу попыток Tune вы даете старт и чем дольше каждая из них выполняется по отдельности, в особенности в распределенных условиях, тем больше вам нужен механизм защиты от аварийных сбоев, остановки прогона или повторного прогона с предыдущими результатами. Библиотека Tune делает это возможным, периодически создавая для вас *контрольные точки*. Каденция контрольных точек динамически корректируется библиотекой Tune, чтобы обеспечивать трату по меньшей мере 95 % времени на выполнение попыток, и чтобы на хранение контрольных точек выделялось не слишком много ресурсов.

В только что вычисленном нами примере дефолтный каталог контрольных точек, или `logdir`, имеет вид `~/ray_results/<your-objective>_<date>_<time>`. Если вы знаете этот каталог вашего эксперимента, то можете легко возобновить (`resume`) эксперимент следующим образом:

```
analysis = tune.run(
    objective,
    name="<your-logdir>",
    resume=True,
    config=search_space)
```

Точно таким же образом свои попытки можно остановить (`stop`), определив условия остановки и передав их в свой вызов функции `tune.run` в явной форме. Для этого самым простым вариантом будет предоставление словаря с условием остановки. Вот как остановить выполнение анализа целевой функции (`objective`) после достижения значения метрики `training_iteration`, равного 10 (это встроенная метрика всех прогонов Tune):

```
tune.run(
    objective,
    config=search_space,
    stop={"training_iteration": 10})
```

Одним из недостатков спецификации условия остановки в таком ключе является то, что исходно делается допущение об увеличении рассматри-

ваемой метрики. Например, вычисляемый нами балл (`score`) начинается с высокого значения, а как раз его-то мы и хотим минимизировать. В целях формулировки гибкого условия остановки для нашего балла лучше всего предоставить функцию остановки, как показано ниже:

```
def stopper(trial_id, result):
    return result["score"] < 2

tune.run(
    objective,
    config=search_space,
    stop=stopper)
```

В ситуациях, когда требуется условие остановки с более широким контекстом или явным состоянием, также есть возможность определять конкретно-прикладной класс `Stopper`<sup>1</sup>, чтобы передавать его в аргумент `stop` вашего прогона Tune, но здесь этот случай рассматриваться не будет.

## **Конкретно-прикладные и условные пространства поиска**

Последняя более продвинутая тема, которую мы здесь рассмотрим, – это сложные пространства поиска. До сих пор мы рассматривали только те гиперпараметры, которые были независимы друг от друга, но на практике некоторые из них нередко зависят от других. Кроме того, несмотря на то что встроенные в библиотеку Tune пространства поиска могут предлагать довольно много возможностей, иногда возникает потребность брать образцы параметров из более экзотического распределения либо ваших собственных модулей.

Вот как можно гармонично справиться с обеими ситуациями. Продолжая наш простой пример с целевой функцией (`objective`), давайте предположим, что для вашего параметра веса (`weight`) вместо функции `tune.uniform` из библиотеки Tune вы хотите использовать функцию взятия образцов `random.uniform` из пакета `numpy`. И тогда ваш параметр смещения (`bias`) должен быть равен весу, умноженному на стандартную нормальную величину. С этой ситуацией (или с более сложными и вложенными) можно справиться с помощью функции `tune.sample_from`, как показано ниже:

```
from ray import tune
import numpy as np

search_space = {
    "weight": tune.sample_from(
        lambda context: np.random.uniform(low=0.0, high=1.0)
    ),
    "bias": tune.sample_from(
        lambda context: context.config.weight * np.random.normal()
    )
}

tune.run(objective, config=search_space)
```

---

<sup>1</sup> См. <https://oreil.ly/1GBqm>.

В библиотеке Ray Tune есть еще много интересных функциональных возможностей, которые обязательно стоит обследовать, но здесь давайте сменим приоритеты и рассмотрим несколько приложений машинного обучения с использованием библиотеки Tune.

## МАШИННОЕ ОБУЧЕНИЕ С ПОМОЩЬЮ БИБЛИОТЕКИ TUNE

Как мы уже видели, библиотека Tune отличается своей универсальностью и позволяет настраивать гиперпараметры для любой поставленной вами целевой функции. В частности, вы можете использовать ее с любым фреймворком машинного обучения, который вас интересует. В этом разделе приведены два примера. Сначала мы собираемся использовать библиотеку Tune для оптимизации параметров эксперимента с применением библиотеки RLLib, а затем посредством библиотеки Tune настроим модель Keras с помощью библиотеки Hyperopt.

### Использование библиотеки RLLib вместе с библиотекой Tune

Библиотеки RLLib и Tune были сконструированы так, чтобы работать вместе, поэтому вы можете довольно легко настраивать эксперимент по гиперпараметрической оптимизации под ваш существующий исходный код RLLib. По сути дела, тренеры библиотеки RLLib можно передавать в первый аргумент функции `tune.run` как тренируемые объекты. При этом можно выбирать между фактическим классом-тренером, например `DQNTuner`, либо его строковым представлением, например "DQN". В качестве метрики библиотеки Tune можно передавать любую метрику, отслеживаемую вашим экспериментом на основе RLLib, например среднее вознаграждение эпизода ("episode\_reward\_mean"). А аргумент `config` в функции `tune.run` – это всего лишь ваша конфигурация тренера RLLib, но вы можете использовать всю мощь API пространства поиска библиотеки Tune, чтобы брать образцы таких гиперпараметров, как скорость самообучения или размер тренировочного пакета<sup>1</sup>. Вот полный пример того, что мы только что описали, а именно выполнение отрегулированного эксперимента на основе RLLib в среде `CartPole-v0` библиотеки Gym:

```
from ray import tune

analysis = tune.run(
    "DQN",
```

---

<sup>1</sup> В случае если вам интересно узнать причину, по которой аргумент «`config`» в функции `tune.run` не был назван `search_space`, то исторически это связано с обеспечением совместимости с объектами `config` библиотеки RLLib.

```
metric="episode_reward_mean",
mode="max",
config={
    "env": "CartPole-v1",
    "lr": tune.uniform(1e-5, 1e-4),
    "train_batch_size": tune.choice([10000, 20000, 40000]),
},
)
```

## Настройка моделей Keras

В завершение этой главы давайте рассмотрим немного более сложный пример. Как мы уже упоминали, эта книга никоим образом не является книгой по машинному обучению, а представляет собой введение во фреймворк Ray и его библиотеки. Следовательно, мы не можем ни познакомить вас с основами машинного обучения, ни потратить много времени на детальное ознакомление с фреймворками машинного обучения. Поэтому в данном разделе мы исходим из вашего знакомства с Keras и его API, а также из наличия некоторых базовых знаний о контролируемом обучении. Если у вас нет этого обязательного минимума знаний и умений, то вы все равно сможете проследить за изложением и сосредоточиться на деталях, специфичных для библиотеки Ray Tune. На следующий далее пример можно смотреть как на более реалистичный сценарий применения библиотеки Tune к рабочим нагрузкам машинного обучения.

Глядя с высоты птичьего полета, мы предпримем следующие ниже шаги:

- 1) загрузим общий набор данных;
- 2) подготовим его к заданию машинного обучения;
- 3) определим для библиотеки Tune целевую функцию, создав модель глубокого обучения с помощью Keras, которая сообщает библиотеке Tune о метрике точности;
- 4) применим интеграцию библиотеки Tune с библиотекой Hyperopt, чтобы определить алгоритм поиска, который настраивает набор гиперпараметров модели Keras.

Рабочий процесс библиотеки Tune остается прежним: мы определяем целевую функцию и пространство поиска, а затем используем функцию `tune.gip` с нужной нам конфигурацией. На высоком уровне процесс использования библиотеки Tune с любым фреймворком машинного обучения работает так, как показано на рис. 5.2.

Для того чтобы определить набор данных, на котором будет выполняться тренировка, давайте напишем простую служебную функцию `load_data`, которая загружает поставляемые с фреймворком Keras знаменитые данные MNIST. Набор данных MNIST состоит из изображений рукописных цифр размером  $28 \times 28$  пикселов. Мы нормализуем значения пикселов так, чтобы они находились в диапазоне от 0 до 1, и создаем метки для этих 10-значных категориальных переменных. Вот как это можно сделать исключительно с помощью встроенной в Keras функциональности (перед выполнением следует применить команду установки `pip install tensorflow`):

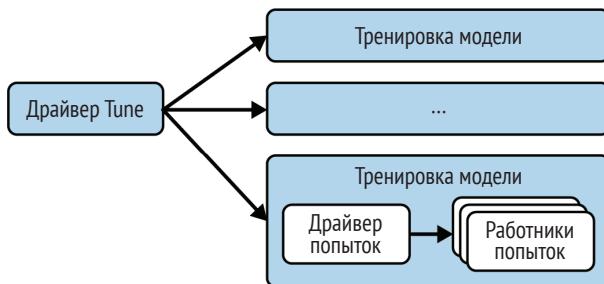
```

from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical

def load_data():
    (x_train, y_train), (x_test, y_test) = mnist.load_data()
    num_classes = 10
    x_train, x_test = x_train / 255.0, x_test / 255.0
    y_train = to_categorical(y_train, num_classes)
    y_test = to_categorical(y_test, num_classes)
    return (x_train, y_train), (x_test, y_test)

load_data()

```



**Рис. 5.2** ❖ Библиотека Tune настраивает распределенную гиперпараметрическую оптимизацию для ваших моделей машинного обучения, выполняя попытки на работниках Ray в вашем кластере и отправляя обратно в драйвер информацию о метриках

Обратите внимание, что после определения функции `load_data` мы вызываем ее один раз, чтобы данные скачались локально. Это связано с тем, что когда вызывается функция `mnist.load_data()`, она сначала ищет локально кешированную копию. Если бы данные не были сперва загружены, то несколько работников Tune попытались бы скачать данные параллельно, что может привести к проблемам<sup>1</sup>.

Далее мы определяем целевую функцию (`objective`) для Tune, или тренируемый объект, путем загрузки только что определенных нами данных, настройки последовательной модели Keras, гиперпараметры которой были отобраны из конфигурации (`config`), передаваемой нами в целевую функцию, а затем компилируем и выполняем подгонку модели. В целях определения модели глубокого обучения мы сначала разглаживаем входные изображения MNIST в векторы, а затем добавляем два полносвязных слоя (именуемых в Keras плотными слоями, `Dense`) и слой отсева (`Dropout`) между ними.

<sup>1</sup> Это даже может привести к редкому состоянию, в котором один работник начинает скачивать данные, а другой проверяет наличие локальной копии и видит ее. Но поскольку скачивание еще не завершено, второй работник попытается открыть потенциально поврежденный файл. Это показывает, что, хотя фреймворк Ray многое улаживает в фоновом режиме, вам все равно следует помнить о том, как вы пишете свой исходный код.

Подлежащими настройке гиперпараметрами являются активационная функция первого плотного слоя, частота отсева и число «скрытых» выходных узлов первого слоя. Любой другой гиперпараметр этой модели можно было бы настроить таким же образом; эта подборка гиперпараметров – всего лишь пример.

Мы могли бы сообщить интересующую метрику вручную точно так же, как делали в других примерах этой главы (например, возвратив словарь в функции `objective` либо используя вызов `tune.report(...)`). Но поскольку библиотека Tune поставляется с надлежащей интеграцией с Keras, мы можем использовать так называемый объект `TuneReportCallback` в качестве конкретно-прикладного обратного вызова Keras, который мы передаем в метод подгонки (`fit`) нашей модели. Вот как выглядит целевая функция (`objective`) Keras:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense, Dropout
from ray.tune.integration.keras import TuneReportCallback

def objective(config):
    (x_train, y_train), (x_test, y_test) = load_data()
    model = Sequential()
    model.add(Flatten(input_shape=(28, 28)))
    model.add(Dense(config["hidden"], activation=config["activation"]))
    model.add(Dropout(config["rate"]))
    model.add(Dense(10, activation="softmax"))

    model.compile(loss="categorical_crossentropy", metrics=["accuracy"])
    model.fit(x_train, y_train, batch_size=128, epochs=10,
              validation_data=(x_test, y_test),
              callbacks=[TuneReportCallback({"mean_accuracy": "accuracy"})])
```

Далее давайте задействуем конкретно-прикладной алгоритм поиска, чтобы настроить эту целевую функцию. В частности, мы используем алгоритм `HyperOptSearch`, который дает нам доступ к алгоритму ТРЕ (Древесно-структурированный парценовский оценщик) библиотеки Hyperopt посредством библиотеки Tune. Для использования этой интеграции следует установить библиотеку Hyperopt на свой компьютер (например, с помощью команды `pip install hyperopt==0.2.7`). Объект `HyperOptSearch` позволяет определять список перспективных начальных вариантов гиперпараметров, чтобы провести их обследование.

Хотя это совершенно необязательно, но иногда у вас могут быть хорошие догадки, с которых можно начинать. В нашем случае мы изначально останавливаемся на проценте ("rate") отсева 0.2, на 128 скрытых ("hidden") узлах и на активационной функции ("activation") под названием выпрямленный линейный узел<sup>1</sup> (ReLU). Кроме этого, с помощью служебного инструмента `tune` можно определить пространство поиска, поступив точно так же, как мы делали это раньше. Наконец, можно получить объект `analysis`, чтобы

---

<sup>1</sup> Англ. rectified linear unit (ReLU). – Прим. перев.

определить наилучшие найденные гиперпараметры, передав все в вызов функции `tune.run`:

```
from ray import tune
from ray.tune.suggest.hyperopt import HyperOptSearch

initial_params = [{"rate": 0.2, "hidden": 128, "activation": "relu"}]
algo = HyperOptSearch(points_to_evaluate=initial_params)

search_space = {
    "rate": tune.uniform(0.1, 0.5),
    "hidden": tune.randint(32, 512),
    "activation": tune.choice(["relu", "tanh"])
}

analysis = tune.run(
    objective,
    name="keras_hyperopt_exp",
    search_alg=algo,
    metric="mean_accuracy",
    mode="max",
    stop={"mean_accuracy": 0.99},
    num_samples=10,
    config=search_space,
)
print("Наилучшими гиперпараметрами были: ", analysis.best_config)
```

Обратите внимание, что здесь мы используем всю мощь библиотеки Hyperopt, без необходимости изучать какие-либо ее особенности. Сама библиотека Hyperopt (по умолчанию) не распределема. Применяя указанную библиотеку посредством API библиотеки Tune, ее можно использовать для распределенной гиперпараметрической оптимизации в кластере Ray.

Мы выбрали комбинацию Keras и Hyperopt в качестве примера использования библиотеки Tune с продвинутым фреймворком машинного обучения и сторонней библиотекой гиперпараметрической оптимизации. Но мы могли бы выбрать любую другую библиотеку машинного обучения и практически любую другую библиотеку гиперпараметрической оптимизации, поддерживающую библиотекой Tune. Если вы заинтересованы в более глубоком знакомстве с любой из многочисленных интеграций, которую предлагает библиотека Tune, то рекомендуем ознакомиться с примерами из документации библиотеки Ray Tune<sup>1</sup>.

## РЕЗЮМЕ

Библиотека Tune, пожалуй, является одним из самых универсальных инструментов гиперпараметрической оптимизации, который можно выбрать

---

<sup>1</sup> См. <https://oreil.ly/rKtZr>.

сегодня. Ее отличительной особенностью является насыщенность функциональными возможностями – она предлагает множество алгоритмов поиска, продвинутые планировщики, сложные пространства поиска, конкретно-прикладные объекты остановки и многие другие функциональные возможности, которые мы не смогли охватить в этой главе. Кроме того, она легко интегрируется с большинством известных инструментов гиперпараметрической оптимизации, таких как Optuna или Hyperopt, что упрощает переход с этих инструментов или просто использование их функциональностей посредством библиотеки Tune. Библиотеку Ray Tune можно рассматривать как гибкий фреймворк распределенной гиперпараметрической оптимизации, *расширяющий* другие, которые могут работать только на одиночных компьютерах.

# Глава 6

---

## Обработка данных с использованием фреймворка Ray

Эдвард Оукс

В главе 5 вы научились настраивать гиперпараметры для ваших экспериментов по машинному обучению. И конечно же, ключевым компонентом применения машинного обучения на практике являются данные. В этой главе мы разведаем стержневой набор возможностей обработки данных во фреймворке Ray: библиотеку Ray Data и ее распределенные наборы данных Dataset<sup>1</sup>.

Хотя библиотека Ray Data не предназначена для замены более общих систем обработки данных, таких как Apache Spark или Apache Hadoop, она предлагает базовые возможности обработки данных и стандартный способ загрузки, преобразования и передачи данных в различные части приложения Ray. За счет этого у экосистемы библиотек в рамках фреймворка Ray появляется возможность говорить на одном языке, благодаря чему пользователи могут смешивать и сочетать функциональные средства независимо от фреймворка, чтобы удовлетворять свои потребности.

Центральный компонент экосистемы библиотеки Ray Data, наборы данных Dataset, предлагает ключевые абстракции для загрузки, преобразования и передачи указателей на данные в кластере Ray. Библиотека Ray Data – это «склеивающее звено», которое позволяет разным библиотекам взаимодействовать поверх фреймворка Ray. Вы увидите этот процесс в действии в разделе «Интеграции с внешними библиотеками» на стр. 161, в котором мы показываем, каким образом можно выполнять обработку наборов данных DataFrame, используя полную выразительность API библиотеки Dask с помощью библиотеки

---

<sup>1</sup> Распределенный набор данных реализован в ней как класс Dataset (`ray.data.Dataset`). – Прим. перев.

Dask on Ray, и преобразовывать результат в набор данных Dataset. Главными выгодами, получаемыми за счет библиотеки Ray Data, являются:

#### Гибкость

Она поддерживает широкий спектр форматов данных, легко интегрируется с библиотеками, такими как Dask on Ray, и ее набор данных Dataset может передаваться между заданиями и акторами Ray без копирования данных.

#### Производительность для рабочих нагрузок машинного обучения

Она предлагает такие важные функциональности, как поддержка ускорителей, конвейеризация и глобальная случайная перетасовка, которые ускоряют тренировку моделей машинного обучения и рабочие нагрузки генерирования модельных предсказаний.

Данная глава познакомит вас с ключевыми концепциями обработки данных во фреймворке Ray и поможет вам понять, как использовать распространенные шаблоны, а также причины, по которым вы будете выбирать разные элементы для выполнения задания. Мы исходим из базового знакомства с концепциями обработки данных, такими как `map`, `filter`, `groupby` и `partition`, но настоящая глава не призвана быть учебным пособием по науке о данных в целом или глубоким погружением во внутренние механизмы реализации этих операций. У читателей с ограниченными знаниями в области науки о данных не должно возникнуть проблем с проследиванием логики изложения материала.

Мы начнем знакомить вас со стержневым строительным блоком: библиотекой Ray Data и ее наборами данных Dataset. Здесь будут рассмотрены архитектура, основы API и пример того, как наборы данных Dataset могут обеспечивать разработку сложных приложений с интенсивным использованием данных. Затем мы кратко рассмотрим интеграции внешних библиотек во фреймворк Ray, сосредоточив внимание на библиотеке Dask on Ray. Наконец, мы сведем все это воедино, создав масштабируемый сквозной конвейер машинного обучения в одном скрипте Python.

Блокнот Jupyter этой главы вместе с данными, использованными в сквозном примере<sup>1</sup>, находится в репозитории книги на GitHub<sup>2</sup>.

## БИБЛИОТЕКА RAY DATA

Главной целью библиотеки Ray является поддержка масштабируемой, гибкой абстракции для обработки данных во фреймворке Ray. Указанная библиотека призвана быть стандартным способом чтения, записи и передачи данных во всей экосистеме библиотек фреймворка Ray. Одним из наиболее эффективных применений библиотеки Ray Data является ее использование в качестве слоя приема и предобработки данных для рабочих нагрузок

---

<sup>1</sup> См. <https://oreil.ly/5Ga8->.

<sup>2</sup> См. <https://oreil.ly/CjHSJ>.

машинного обучения, что позволяет эффективным образом вертикально масштабировать тренировку с помощью библиотек Ray Train и Ray Tune. Мы разведаем эту тему подробнее в разделе «Разработка конвейера машинного обучения» на стр. 164.

Если вы в прошлом работали с другими API распределенной обработки данных, такими как Resilient Distributed Datasets (RDD)<sup>1</sup> в Apache Spark, API библиотеки Ray Data будет вам хорошо знаком. Ядро API основано на функциональном программировании и предлагает стандартные функции, такие как чтение и запись различных источников данных; выполнение базовых преобразований, таких как `map`, `filter` и `sort`, а также выполнение некоторых простых агрегаций, таких как `groupby`.

Под капотом в библиотеке Ray Data реализован распределенный формат Apache Arrow<sup>2</sup>. Apache Arrow – это унифицированный столбчатый формат данных для библиотек и приложений обработки данных. Интеграция с Apache Arrow означает, что библиотека Ray Data получает совместимость со многими наиболее популярными библиотеками обработки, такими как NumPy и Pandas, прямо «из коробки».

Наборы данных Dataset состоят из списка указателей на объекты Ray, каждый из которых указывает на «блок» данных. Эти блоки представляют собой либо таблицы Arrow, либо списки Python (для данных, которые не поддерживаются форматом Arrow) в хранилище объектов Ray с совместной памятью, а вычисления на данных, например операции отображения (`map`) или фильтрации (`filter`), происходят в заданиях Ray (а иногда и в акторах)<sup>3</sup>.

Поскольку библиотека Ray Data опирается на ключевые примитивы заданий и объектов Ray в хранилище объектов с совместной памятью, она наследует ключевые преимущества фреймворка Ray: масштабируемость до сотен узлов, эффективное потребление памяти за счет совместного использования памяти процессами на одном узле, а также восстановление объектов после их потери с целью плавного урегулирования аппаратных сбоев. В дополнение к этому, поскольку наборы данных Dataset представляют собой просто списки указателей на объекты, их также можно эффективно передавать между заданиями и акторами без необходимости создания копий данных, что крайне важно для обеспечения масштабируемости приложений и библиотек с интенсивным использованием данных.

## Основы библиотеки Ray Data

В этом разделе будет дан краткий обзор библиотеки Ray Data, рассказывающий о том, как приступить к чтению, записи и преобразованию наборов

---

<sup>1</sup> Отказоустойчивые распределенные наборы данных. – Прим. перев.

<sup>2</sup> См. <https://arrow.apache.org/>.

<sup>3</sup> Согласно документации, класс `Dataset` библиотеки Ray Data создает последовательность распределенных блоков данных и предоставляет методы для крупномасштабного чтения, преобразования и приема данных. `Dataset` оперирует последовательностью указателей на блоки. Каждый блок содержит набор записей в таблице Arrow или наборе данных `DataFrame` библиотеки Pandas. – Прим. перев.

данных Dataset. Это не исчерпывающий справочник, а просто введение в базовые концепции, чтобы позже иметь возможность перейти к интересным примерам, показывающим, что, собственно говоря, делает библиотеку Ray Data и ее наборы данных Dataset мощными. Для получения актуальной информации о том, что конкретно поддерживается, и о точном синтаксисе рекомендуем обратиться к документации по библиотеке Ray Data<sup>1</sup>.

Для отслеживания работы примеров этого раздела следует проверить, что-бы функциональность библиотеки Ray Data была установлена локально:

```
pip install "ray[data]==2.2.0"
```

## ***Создание набора данных Dataset***

Сначала давайте создадим простой набор данных Dataset и выполним на нем несколько базовых операций:

```
import ray

# Создать набор данных, содержащий
# целые числа в диапазоне [0, 10000].
ds = ray.data.range(10000)

# Базовые операции:
# показать размер набора данных,
# получить несколько образцов,
# распечатать схему.
print(ds.count())    # -> 10000
print(ds.take(5))   # -> [0, 1, 2, 3, 4]
print(ds.schema())   # -> <class 'int'>
```

Здесь мы создали набор данных Dataset, содержащий числа от 0 до 10 000, а затем напечатали немного базовой информации о нем: общее число записей, несколько образцов и схему.

## ***Чтение из хранилища и запись в него***

Разумеется, в реальных рабочих нагрузках у вас нередко будет возникать потребность в чтении из постоянного хранилища и записи в него, чтобы загружать свои данные и записывать результаты. Запись и чтение наборов данных Dataset выполняются просто; например, запись набора данных Dataset в CSV-файл и затем его загрузка обратно в память сводятся к применению встроенных служебных функций `write_csv` и `read_csv`:

```
# Сохранить набор данных в локальном файле
# и загрузить его обратно
ray.data.range(10000).write_csv("local_dir")
ds = ray.data.read_csv("local_dir")
print(ds.count())
```

---

<sup>1</sup> См. <https://oreil.ly/aRTsX>.

Библиотека Ray Data поддерживает ряд распространенных форматов сериализации, таких как CSV, JSON и Parquet, и может читать с локального диска или писать на него, а также в дистанционное хранилище, такое как HDFS или AWS S3.

В приведенном выше примере мы указали только путь к локальному файлу ("local\_dir"), поэтому набор данных Dataset был записан в каталог на локальном компьютере. Если вместо этого мы бы хотели писать в S3 и читать из него, то мы бы указали путь типа "s3://my\_bucket/", и библиотека Ray Data автоматически эффективным образом обработала бы чтение и запись в дистанционное хранилище<sup>1</sup>, параллелизая запросы по нескольким задачам с целью повышения пропускной способности.

Обратите внимание, что библиотека Ray Data также поддерживает конкретно-прикладные источники данных, которые можно использовать для записи в любую внешнюю систему хранения данных, которая не поддерживается прямо «из коробки».

## Встроенные преобразования

Теперь, когда мы разобрались с базовыми API, связанными с созданием и инспектированием наборов данных Dataset, давайте взглянем на несколько встроенных операций, которые можно на них выполнять. В следующем ниже примере исходного кода показаны три базовые операции, которые поддерживаются наборами данных Dataset:

```
ds1 = ray.data.range(10000)
ds2 = ray.data.range(10000)
ds3 = ds1.union(ds2) ❶
print(ds3.count()) # -> 20000

# Отфильтровать комбинированный набор данных,
# оставив только четные элементы
ds3 = ds3.filter(lambda x: x % 2 == 0) ❷
print(ds3.count()) # -> 10000
print(ds3.take(5)) # -> [0, 2, 4, 6, 8]

# Отсортировать фильтрованный набор данных
ds3 = ds3.sort()
print(ds3.take(5)) # -> [0, 0, 2, 2, 4]
```

- ❶ Объединить (union) два набора данных Dataset. Результатом является новый Dataset, содержащий все записи обоих наборов.
- ❷ Отфильтровать (filter) элементы набора данных Dataset, чтобы оставить только четные целые числа, предоставив конкретно-прикладную функцию фильтрации.
- ❸ Отсортировать (sort) набор данных Dataset.

---

<sup>1</sup> Если вам интересно проследить работу примера, который читает фактические данные из корзины S3, то рекомендуем обратиться к примеру пакетного генерирования модельных предсказаний в документации фреймворка Ray (<https://oreil.ly/9C82a>).

В дополнение к этим операциям наборы данных Dataset также поддерживают распространенные агрегации, которые можно было бы ожидать, такие как `groupby`, `sum`, `min` и т. д. Кроме того, есть возможность передавать пользовательскую функцию для конкретно-прикладных агрегаций.

## Блоки и реорганизация блоков

При использовании наборов данных Dataset важно учитывать концепцию блоков. Блоки – это базовые порции данных, составляющие набор данных Dataset; операции применяются к опорным данным по одному блоку за раз. Если число блоков в Dataset слишком велико, то каждый блок будет малым, и для каждой операции потребуется много непроизводительных издержек. Если число блоков будет слишком мало, то выполнять параллелизацию операций эффективно будет невозможно.

Если заглянуть под капот приведенного выше примера, то мы увидим, что в каждом созданном нами исходном наборе данных Dataset по умолчанию было по 200 блоков. Когда мы их объединили, результирующий Dataset состоял из 400 блоков. Учитывая важность числа блоков для эффективности, мы, возможно, захотим перетасовать данные, чтобы они соответствовали нашим изначальным 200 блокам, и оставить тот же параллелизм. Этот процесс изменения числа блоков называется *реорганизацией блоков*, и для достижения этой цели наборы данных Dataset предлагают простой API `.repartition(num_blocks)`. Давайте воспользуемся этим API, чтобы реорганизовать наш результирующий набор данных Dataset обратно на 200 блоков:

```
ds1 = ray.data.range(10000)
print(ds1.num_blocks()) # -> 200
ds2 = ray.data.range(10000)
print(ds2.num_blocks()) # -> 200
ds3 = ds1.union(ds2)
print(ds3.num_blocks()) # -> 400

print(ds3.repartition(200).num_blocks()) # -> 200
```

Блоки также осуществляют контроль над числом файлов, которые создаются, когда мы пишем набор данных Dataset в хранилище (поэтому если вы хотите, чтобы все данные были объединены в один выходной файл, то перед его записью следует вызвать `.repartition(1)`).

## Схемы и форматы данных

До этого момента мы оперировали простыми наборами данных Dataset, состоящими только из целых чисел. Однако для более сложной обработки данных нередко требуется схема, позволяющая легче воспринимать данные и применять типы к каждому столбцу.

Учитывая, что наборы данных Dataset библиотеки Ray Data призваны быть центральным местом взаимодействия приложений и библиотек в рамках фреймворка Ray, они разработаны таким образом, чтобы не зависеть от конкретного типа данных и обеспечивать гибкость при чтении, записи и преоб-

разовании между многими популярными форматами данных. Наборы данных Dataset поддерживают столбчатый формат Arrow, который позволяет выполнять преобразование между различными типами структурированных данных, таких как словари Python, наборы данных DataFrame и сериализованные файлы Parquet.

Самый простой способ создать набор данных Dataset со схемой состоит в его создании из списка словарей Python:

```
ds = ray.data.from_items([{"id": "abc", "value": 1},
                           {"id": "def", "value": 2}])
print(ds.schema()) # -> id: string, value: int64
```

В данном случае схема была выведена из ключей в переданных нами словарях. Мы также можем конвертировать в типы данных и из типов данных популярных библиотек, таких как Pandas:

```
# pandas_df унаследует схему из нашего Dataset
pandas_df = ds.to_pandas()
```

Здесь мы перешли от набора данных Dataset библиотеки Ray Data к набору данных DataFrame библиотеки Pandas, но это работает и в обратном порядке: если создать набор данных Dataset из набора данных DataFrame, то он автоматически унаследует схему из набора данных DataFrame.

## Вычисления на наборах данных Dataset

В предыдущем разделе мы представили некоторые функциональные возможности, встроенные в наборы данных Dataset, такие как фильтрация, сортировка и объединение. Однако одной из самых мощных частей наборов данных Dataset является то, что они позволяют использовать гибкую вычислительную модель фреймворка Ray и эффективно выполнять вычисления с большими объемами данных.

Первейшим способом выполнения конкретно-прикладного преобразования на наборе данных Dataset является использование метода `.map()`. Он позволяет передавать конкретно-прикладную функцию, которая будет применена к записям набора данных Dataset. Базовым примером может быть возвведение записей набора данных Dataset в квадрат:

```
ds = ray.data.range(10000).map(lambda x: x ** 2)
ds.take(5) # -> [0, 1, 4, 9, 16]
```

В данном примере мы передали простую лямбда-функцию, и данные, на которых мы оперировали, были целыми числами, но здесь мы могли бы передать любую функцию и оперировать на структурированных данных, поддерживающих формат Arrow.

Кроме того, с помощью метода `.map_batches()` можно применять преобразование не отдельных записей, а целых пакетов данных. Некоторые типы вычислений намного эффективнее, когда они *векторизованы*, то есть в них

используется алгоритм или реализация, которые эффективнее оперируют на наборе элементов, чем на одном элементе за один раз.

Возвращаясь к простому примеру возвведения значений в квадрат в наборе данных Dataset, его можно переписать так, чтобы он выполнялся пакетно, и вместо наивной реализации на Python использовать реализацию, оптимизированную под numpy.square:

```
import numpy as np

ds = ray.data.range(10000).map_batches(
    lambda batch: np.square(batch).tolist())
ds.take(5) # -> [0, 1, 4, 9, 16]
```

Векторизованные вычисления особенно полезны на графических процессорах при выполнении тренировки моделей глубокого обучения или генерирования модельных предсказаний<sup>1</sup>. Однако, как правило, выполнение вычислений на графических процессорах также сопряжено со значительными фиксированными затратами из-за необходимости загружать веса моделей или другие данные в оперативную память графического процессора. Для этой цели наборы данных Dataset поддерживают отображение данных с использованием акторов Ray. Акторы Ray долговечны и могут содержать состояние, в отличие от заданий Ray, характерных отсутствием внутреннего состояния, поэтому мы можем кешировать дорогостоящие операционные затраты, выполняя их в конструкторе актора (например, загружая модель в графический процессор).

Например, для того чтобы выполнить пакетное генерирование модельных предсказаний с использованием набора данных Dataset, нужно вместо функции передать класс, указать, что это вычисление должно выполняться с использованием акторов, и применить метод .map\_batches(), в результате чего мы получаем возможность векторизованного генерирования модельных предсказаний. Библиотека Ray Data будет автомасштабировать группу акторов под выполнение операции отображения:

```
def load_model():
    # Этот пример возвращает фиктивную модель.
    # В реальности данная функция, скорее всего, загрузила
    # бы в графический процессор некие модельные веса.
    class DummyModel:
        def __call__(self, batch):
            return batch

    return DummyModel()

class MLModel:
    def __init__(self):
        # load_model() будет выполняться всего
        # один раз для каждого запущенного актора
        self._model = load_model()
```

---

<sup>1</sup> Англ. inference. См. <https://www.anyscale.com/blog/model-batch-inference-in-ray-actors-actorpool-and-datasets>. – Прим. перев.

```

def __call__(self, batch):
    return self._model(batch)

ds.map_batches(MLModel, compute="actors")

```

Для того чтобы сгенерировать модельные предсказания на графическом процессоре, мы бы передали `num_gpus=1` в вызове `map_batches`, тем самым указав, что выполняющим функцию `map` акторам требуется графический процессор.

## Конвейеры наборов данных Dataset

По умолчанию операции на наборе данных `Dataset` являются блокирующими, а это значит, что они выполняются синхронно от начала до конца, и за один раз выполняется только одна операция. Однако в некоторых рабочих нагрузках этот шаблон бывает очень неэффективным. Например, рассмотрим следующую ниже серию преобразований набора данных `Dataset` на данных `Parquet`, которые могут быть использованы для выполнения пакетного генерирования предсказаний моделью машинного обучения<sup>1</sup>:

```

ds = (ray.data.read_parquet("s3://my_bucket/input_data") ❶
      .map(cpu_intensive_preprocessing) ❷
      .map_batches(cuda_intensive_inference, compute="actors", num_gpus=1) ❸
      .repartition(10)) ❹

ds.write_parquet("s3://my_bucket/output_predictions") ❽

```

Этот процесс состоит из пяти этапов, и на каждом из них особое внимание уделяется разным частям системы:

- ❶ чтение из дистанционного хранилища требует входную пропускную способность в кластер и может быть ограничено пропускной способностью системы хранения. На этом этапе порождается группа заданий Ray, которые будут читать данные из дистанционного хранилища в параллельном режиме, а результирующие блоки данных – сохраняться в хранилище объектов Ray;
- ❷ предобработка входных данных требует ресурсов центрального процессора. Объекты из первой фазы передаются в группу заданий, которые будут исполнять функцию предобработки `cpu_intensive_processing` на каждом блоке;
- ❸ векторизованное генерирование модельных предсказаний требует ресурсов графического процессора. Тот же процесс, что и на втором этапе, повторяется для `gpu_intensive_inference`, за исключением того,

<sup>1</sup> Parquet – это структурированный столбчатый формат, который обеспечивает эффективное сжатие, хранение и извлечение данных. Указанный формат используется во многих примерах и реально существующих наборах данных, и поэтому он применяется в приведенном выше примере. Однако исходный код можно легко изменить под использование другого формата, поменяв вызовы `read_parquet` и `write_parquet`.

что на этот раз функция выполняется на акторах, каждому из которых выделен графический процессор, и в каждый вызов функции (пакетно) передается несколько блоков. Акторы используются в этом шаге, чтобы избегать неоднократной перезагрузки применяемой для генерирования предсказаний модели на графический процессор;

- ④ для реорганизации блоков требуется сетевая пропускная способность внутри кластера. После завершения этапа 3 порождаются дополнительные задания по реорганизации данных на 10 блоков и записи каждого из этих 10 блоков в дистанционное хранилище;
- ⑤ запись в дистанционное хранилище требует выходной пропускной способности из кластера и может быть снова ограничена пропускной способностью хранилища.

На рис. 6.1 показана базовая реализация, в которой каждый этап выполняется последовательно. Эта наивная реализация приводит к простаиванию ресурсов, потому что каждый этап является блокирующим и выполняется последовательно. Например, поскольку ресурсы графического процессора используются только на заключительном этапе, они будут простаивать в ожидании загрузки и предобработки всех данных.

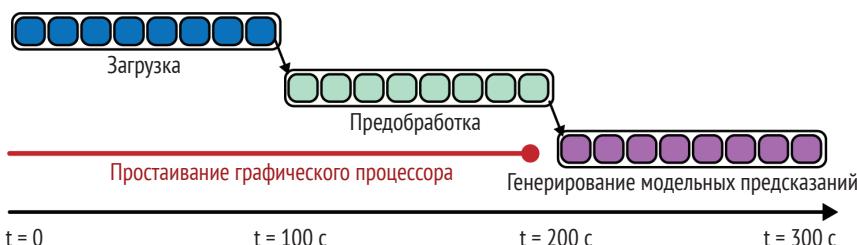


Рис. 6.1 ♦ Наивное вычисление набора данных Dataset, приводящее к простаиванию ресурсов между этапами

В данном сценарии вместо этого было бы эффективнее передавать этапы по конвейеру и давать им возможность накладываться друг на друга, как показано на рис. 6.2. Это означает, что после того, как некоторые данные были прочитаны из хранилища, они передаются на этап предобработки, затем на этап генерирования модельных предсказаний и т. д.

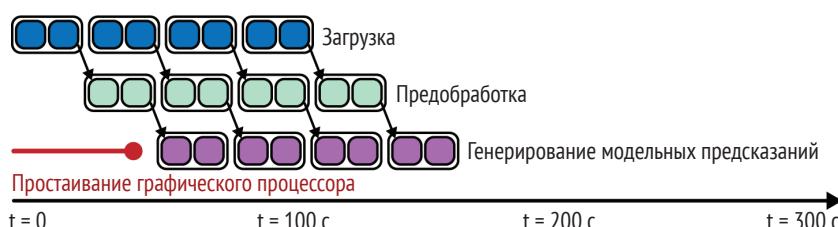


Рис. 6.2 ♦ Оптимизированный конвейер набора данных Dataset, который обеспечивает дублирование вычислений между этапами и сокращает количество незанятых ресурсов

Такая конвейеризация будет улучшать общее потребление ресурсов сквозной рабочей нагрузки, повышая пропускную способность и, следовательно, снижая затраты, необходимые для выполнения вычислений (чем меньше пристаивающих ресурсов, тем лучше!).

Наборы данных Dataset можно конвертировать в конвейеры наборов данных Dataset (DatasetPipeline)<sup>1</sup> с помощью метода `ds.window()`, что позволяет использовать поведение конвейеризации, которое мы хотим в этом сценарии. В окне указывается число блоков, которые будут пропускаться через этап в конвейере перед передачей на следующий этап. Это поведение можно отрегулировать с помощью параметра `blocks_per_window`, значение которого по умолчанию равно 10.

Давайте перепишем неэффективный псевдокод, чтобы вместо него использовать конвейер набора данных Dataset (DatasetPipeline):

```
ds = (ray.data.read_parquet("s3://my_bucket/input_data")
    .window(blocks_per_window=5)
    .map(cpu_intensive_preprocessing)
    .map_batches(gpu_intensive_inference, compute="actors", num_gpus=1)
    .repartition(10))
ds.write_parquet("s3://my_bucket/output_predictions")
```

Единственным внесенным изменением было добавление вызова метода `.window()` после `read_parquet` и перед этапом предобработки. Теперь набор данных Dataset преобразован в конвейер набора данных Dataset (DatasetPipeline), и его этапы будут выполняться параллельно в пятиблочных окнах, сокращая пристаивающие ресурсы и повышая эффективность.

Конвейеры наборов данных Dataset также можно создавать с помощью функции `ds.repeat()`, чтобы повторять этапы в конвейере конечное или бесконечное число раз. Это будет рассмотрено далее в следующем разделе, где мы будем использовать указанную функциональность для рабочей нагрузки тренировки. Разумеется, конвейеризация бывает в равной степени полезна для результативности тренировки, как и для генерирования модельных предсказаний.

## Пример: параллельная тренировка копий классификатора

Одним из ключевых преимуществ наборов данных Datasets является то, что их можно передавать между заданиями и акторами. В этом разделе мы рассмотрим, как эту функциональность можно использовать для написания эффективных реализаций сложных распределенных рабочих нагрузок, таких как распределенная настройка гиперпараметров и тренировка моделей машинного обучения. В данном разделе мы рассмотрим пример распределенной тренировки моделей машинного обучения и указанную тему разбе-

---

<sup>1</sup> См. [https://oreil.ly/Hr2d\\_](https://oreil.ly/Hr2d_).

рем гораздо подробнее в главе 7, когда будем знакомить вас с библиотекой Ray Train.

Как обсуждалось в главе 5, распространенным шаблоном тренировки моделей машинного обучения является разведывание диапазона гиперпараметров, чтобы найти те, которые приводят к наилучшей модели. Возможно, мы захотим работать с широким спектром гиперпараметров, и делать это наивно бывает очень дорого. Библиотека Ray Data позволяет легко делиться одинаковыми резидентными данными между разными параллельными прогонами тренировки в одном скрипте Python: мы можем загрузить и предобработать данные один раз, а затем передать на них указатель многим нижестоящим акторам, которые могут читать данные из совместной памяти.

В дополнение к этому при работе с очень крупными наборами данных иногда не представляется возможным загружать полные тренировочные данные в память за один процесс или на одной машине. В этом случае обычно используется *сегментирование данных*, что означает предоставление каждому работнику его собственного подмножества данных, которые могут поместиться в памяти. Это локальное подмножество данных называется *сегментом данных*<sup>1</sup>. После того как каждый работник параллельно натренируется на своем сегменте данных, результаты комбинируются синхронно либо асинхронно с использованием сервера параметров. Два важных соображения могут затруднить правильное понимание этого процесса:

- во многих алгоритмах распределенной тренировки используется синхронный подход, требуя от работников синхронизировать свои веса после каждой эпохи тренировки. Это означает, что между работниками должна быть некоторая координация, чтобы поддерживать согласованность между пакетами данных, на которых они оперируют;
- важно, чтобы каждый работник получал случайный образец данных в течение каждой эпохи. Было доказано, что глобальная случайная перетасовка работает лучше, чем локальная перетасовка или отсутствие перетасовки.

На рис. 6.3 показано, как пакет `ray.data` используется для создания сегментов данных, формирующих набор данных Dataset из указанного входного набора данных.

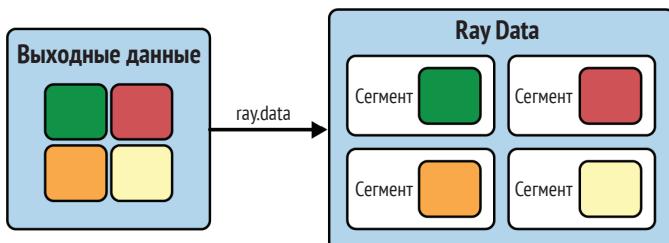


Рис. 6.3 ♦ Создание набора данных Dataset из входных данных с помощью пакета `ray.data`

<sup>1</sup> Англ. data shard. – Прим. перев.

Давайте рассмотрим пример того, каким образом можно реализовывать этот тип шаблона, используя наборы данных Dataset. В приведенном ниже примере мы будем параллельно тренировать несколько копий модели машинного обучения, используя разные гиперпараметры для разных работников.

Мы будем тренировать алгоритм `SGDClassifier` библиотеки scikit-learn на сгенерированном наборе данных двоичной классификации, и настроим гиперпараметр, которым будет регуляризационный член этого классификатора<sup>1</sup>. В этом примере фактические детали задания и модели машинного обучения не слишком важны: модель и данные можно заменить любым числом примеров. Главное, на чем здесь следует сосредоточиться, – это способ организации загрузки данных и вычислений с использованием наборов данных Dataset.

Для отслеживания работы примеров этого раздела следует проверить, чтобы у вас были локально установлены библиотеки Ray Data и scikit-learn<sup>2</sup>:

```
pip install "ray[data]==2.2.0" "scikit-learn==1.0.2"
```

Сначала давайте определим работника тренировки (`TrainingWorker`), который будет тренировать копию классификатора на данных:

```
from sklearn import datasets
from sklearn.linear_model import SGDClassifier
from sklearn.model_selection import train_test_split

@ray.remote
class TrainingWorker:
    def __init__(self, alpha: float):
        self._model = SGDClassifier(alpha=alpha)

    def train(self, train_shard: ray.data.Dataset):
        for i, epoch in enumerate(train_shard.iter_epochs()):
            X, Y = zip(*list(epoch.iter_rows()))
            self._model.partial_fit(X, Y, classes=[0, 1])

    return self._model

    def test(self, X_test: np.ndarray, Y_test: np.ndarray):
        return self._model.score(X_test, Y_test)
```

Тут следует обратить внимание на три важные вещи в отношении работника тренировки `TrainingWorker`:

- 
- <sup>1</sup> Аббревиатура SGD расшифровывается как «стохастический градиентный спуск» (Stochastic Gradient Descent). Это распространенный алгоритм оптимизации, используемый в машинном обучении, в частности в глубоком обучении.
  - <sup>2</sup> В этой и следующих двух главах мы обсуждаем более продвинутые примеры машинного обучения, для которых требуются дополнительные зависимости. Мы закрепляем версии этих зависимостей здесь и в репозитории книги на GitHub ([https://oreil.ly/learning\\_ray\\_repo](https://oreil.ly/learning_ray_repo)), чтобы обеспечить надлежащую работу примеров. С учетом сказанного, примеры, скорее всего, будут работать с относительно широким спектром версий, при условии что вы будете использовать не слишком старые.

- это простая обертка вокруг классификатора `SGDClassifier`, которая создает его экземпляр с заданным значением `alpha`;
- главная функция тренировки выполняется в методе `train`. Она тренирует классификатор на имеющихся данных в каждую эпоху;
- у нас также есть метод `test`, который можно использовать для выполнения натренированной модели на тестовом наборе.

Теперь давайте создадим несколько экземпляров работника тренировки (`TrainingWorker`) с разными гиперпараметрами (значениями `alpha`):

```
ALPHA_VALS = [0.00008, 0.00009, 0.0001, 0.00011, 0.00012]  
print(f"Запуск {len(ALPHA_VALS)} работников тренировки.")  
workers = [TrainingWorker.remote(alpha) for alpha in ALPHA_VALS]
```

Затем мы генерируем тренировочные и валидационные данные и конвертируем тренировочные данные в набор данных `Dataset`. Здесь используется функция `.repeat()`, чтобы создать конвейер набора данных `Dataset` (`DatasetPipeline`). В нем определяется число эпох, в течение которых будет проходить тренировка. В каждую эпоху последующие операции будут применяться к набору данных `Dataset`, и работники смогут прокручивать результирующие данные в цикле. Мы также перетасовываем данные в случайном порядке и их сегментируем, чтобы затем передать работникам тренировки, каждый из которых получает равную порцию данных:

```
X_train, X_test, Y_train, Y_test = train_test_split(❶  
    *datasets.make_classification()  
)  
  
train_ds = ray.data.from_items(list(zip(X_train, Y_train))) ❷  
shards = (train_ds.repeat(❸  
    .random_shuffle_each_window() ❹  
    .split(len(workers), locality_hints=workers)) ❺  
  
ray.get([  
    worker.train.remote(shard)  
    for worker, shard in zip(workers, shards  
)]) ❻
```

- ❶ Генерировать тренировочные и валидационные данные для задачи классификации.
- ❷ Конвертировать тренировочные данные в набор данных `Dataset` с помощью `from_items`.
- ❸ Определить конвейер набора данных `Dataset` (`DatasetPipeline`) с помощью метода `.repeat`. Это похоже на использование метода `.window`, как мы показывали ранее, но он позволяет прокручивать один и тот же набор данных несколько раз (в данном случае 10).
- ❹ Перетасовывать данные в случайном порядке всякий раз, когда они повторяются.
- ❺ Мы хотим, чтобы у каждого работника был свой собственный локальный сегмент данных, поэтому разбиваем (`split`) конвейер набора данных `Dataset` на несколько меньших сегментов, которые можно передать каждому работнику.
- ❻ Дождаться завершения тренировки на всех работниках.

При выполнении тренировки на работниках мы вызываем их метод `train` и передаем каждому по одному сегменту конвейера `DatasetPipeline`. Затем

блокируем, ожидая завершения тренировки на всех работниках. Давайте подведем итог тому, что происходит на этом этапе:

- 1) каждую эпоху каждый работник получает случайный сегмент данных;
- 2) работник тренирует свою локальную модель на назначенному ему сегменте данных;
- 3) после того как работник завершает тренировку на текущем сегменте, он блокирует до тех пор, пока другие работники не завершат тренировку;
- 4) предыдущие три шага повторятся для оставшихся эпох (в данном случае всего 10).

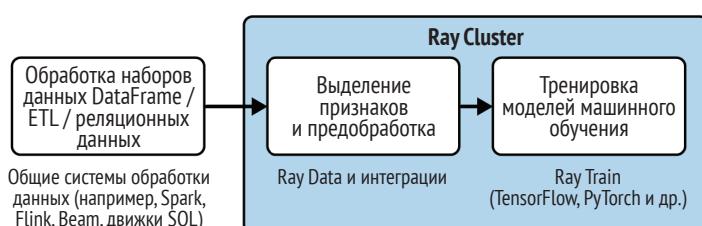
Наконец, натренированные модели от каждого работника можно протестировать на неких тестовых данных, чтобы определить, какое значение  $\alpha$  дало наиболее точную модель:

```
# Получить результаты валидации от каждого работника
print(ray.get([worker.test.remote(X_test, Y_test) for worker in workers]))
```

В реальности для такого типа рабочей нагрузки следует использовать библиотеку Ray Tune или Ray Train, о которых мы поговорим в следующей главе, но этот пример демонстрирует возможности библиотеки Ray Data и ее наборов данных Dataset для рабочих нагрузок машинного обучения. Всего в нескольких фрагментах исходного кода Python мы реализовали сложный распределенный рабочий процесс настройки гиперпараметров и тренировки, который можно легко масштабировать до сотен машин и который не зависит от какого-либо фреймворка или конкретной задачи машинного обучения.

## ИНТЕГРАЦИИ С ВНЕШНИМИ БИБЛИОТЕКАМИ

Хотя библиотека Ray Data поддерживает ряд распространенных функциональностей обработки данных прямо «из коробки», как мы уже обсуждали, она не является заменой полноценным системам обработки данных. Вместо этого, как показано на рис. 6.4, она больше ориентирована на выполнение обработки на «последней миле», такой как базовая загрузка данных, их очистка и выделение признаков перед тренировкой моделей машинного обучения или генерируением модельных предсказаний.



**Рис. 6.4** ♦ Типичный рабочий процесс с использованием фреймворка Ray для машинного обучения: использование внешних систем для первичной обработки данных и ETL, использование библиотеки Ray Data для предобработки на последней миле

Однако с фреймворком Ray интегрируется ряд других, более полнофункциональных систем обработки наборов данных DataFrame и реляционных данных, таких как:

- Dask on Ray;
- RayDP (Spark on Ray);
- Modin (Pandas on Ray);
- MARS on Ray.

Это автономные библиотеки обработки данных, с которыми вы, возможно, знакомы вне контекста фреймворка Ray. Каждый из этих инструментов интегрируется с ядром Ray (Ray Core), что обеспечивает более выразительную обработку данных, чем при использовании встроенной библиотеки Ray Data, при этом по-прежнему остается возможность использовать инструментарий развертывания, масштабируемого планирования и хранения объектов Ray с совместной памятью для обмена данными. Как показано на рис. 6.5, это дополняет Ray Data и обеспечивает сквозную обработку данных во фреймворке Ray.

На рис. 6.5 показаны преимущества интеграций экосистемы библиотеки Ray Data, обеспечивающих более выразительную обработку данных во фреймворке Ray. Приведенные на нем библиотеки интегрируются с библиотекой Ray Data для подачи в нижестоящие библиотеки, такие как Ray Train.



**Рис. 6.5 ♦ Интеграции с экосистемой библиотеки Ray Data**  
обеспечивают более выразительную обработку данных во фреймворке Ray

Для целей этой книги мы рассмотрим библиотеку Dask on Ray чуть подробнее, чтобы дать вам представление о том, как эти интеграции выглядят. Если вас интересуют подробности конкретной интеграции, то рекомендуем ознакомиться с последней документацией фреймворка Ray, чтобы получить актуальную информацию<sup>1</sup>.

Для отслеживания работы примеров этого раздела следует установить библиотеки Ray и Dask:

```
pip install "ray[data]==2.2.0" "dask==2022.2.0"
```

Dask<sup>2</sup> – это библиотека Python для параллельных вычислений, специально предназначеннная для масштабирования аналитических и научных вычисли-

<sup>1</sup> См. <https://oreil.ly/5MpG->.

<sup>2</sup> См. <https://dask.org/>.

тельных нагрузок в кластере. Одной из самых популярных функциональных особенностей библиотеки Dask являются ее компонент для работы с наборами данных DataFrame<sup>1</sup>, который предлагает подмножество API библиотеки Pandas для работы с ее наборами данных DataFrame, масштабируемое до кластера машин в случаях невозможности обработки в памяти на одном узле. Указанный компонент для работы с DataFrame работает путем создания *графа заданий*, который передается планировщику на исполнение. Наиболее типичным способом исполнения операций с компонентом DataFrame библиотеки Dask является использование распределенного планировщика Dask, однако еще существует подключаемый API, который позволяет и другим планировщикам исполнять эти графы заданий.

Фреймворк Ray поставляется в комплекте с бэкендом планировщика Dask, позволяющим исполнять графы заданий с наборами данных DataFrame как задания Ray и, следовательно, использовать планировщик Ray и хранилище объектов с совместной памятью. Это вообще не требует видоизменения ключевого исходного кода компонента DataFrame; наоборот, для выполнения с использованием фреймворка Ray сначала нужно лишь подключиться к работающему кластеру Ray (или запустить фреймворк Ray локально), а затем активировать бэкенд планировщика Ray:

```
import ray
from ray.util.dask import enable_dask_on_ray

ray.init() # Запустить или подключиться к Ray
enable_dask_on_ray() # Активировать бэкенд планировщика Ray для Dask
```

Теперь можно выполнить обычный исходный код с использованием компонента для работы с наборами данных DataFrame библиотеки Dask и масштабировать его по всему кластеру Ray. Например, мы могли бы провести некий анализ временного ряда, используя стандартные операции на наборах данных DataFrame, такие как filter и groupby, и вычислить стандартное отклонение (пример взят из документации библиотеки Dask):

```
import dask

df = dask.datasets.timeseries()
df = df[df.y > 0].groupby("name").x.std()
df.compute() # Запустить подлежащий оцениванию граф заданий
```

Если вы привыкли к библиотеке Pandas или другим библиотекам обработки наборов данных DataFrame, то, возможно, зададитесь вопросом, зачем нужно вызывать df.compute(). Это связано с тем, что библиотека Dask по умолчанию ленива и будет вычислять результаты только по запросу, что позволяет ей оптимизировать график заданий, который будет исполняться во всем кластере.

Одним из самых мощных аспектов библиотеки Dask on Ray является то, что она очень хорошо интегрируется с библиотекой Ray Data. Используя встроен-

---

<sup>1</sup> См. <https://oreil.ly/k3yJb>.

ные служебные функции, набор данных Dataset библиотеки Ray Data можно конвертировать в набор данных DataFrame библиотеки Dask и наоборот:

```
import ray

ds = ray.data.range(10000)

# Конвертировать Dataset в DataFrame библиотеки Dask
df = ds.to_dask()
print(df.std().compute()) # -> 2886.89568

# Конвертировать DataFrame библиотеки Dask обратно в Dataset
ds = ray.data.from_dask(df)
print(ds.std()) # -> 2886.89568
```

Возможно, этот простой пример не впечатлит, потому что стандартное от клонение можно вычислить, используя либо наборы данных DataFrame библиотеки Dask, либо наборы данных Dataset библиотеки Ray Data. Однако, как вы увидите в следующем далее разделе, в котором мы разработаем сквозной конвейер машинного обучения, за счет этого обеспечиваются мощные рабочие процессы. Например, полную выразительность наборов данных DataFrame можно использовать для выделения признаков и предобработки, а затем передавать данные непосредственно в последующие операции, такие как распределенная тренировка или генерирование модельных предсказаний, держа все в памяти. Это показывает, как библиотека Ray Data обеспечивает широкий спектр вариантов использования поверх фреймворка Ray и как такие интеграции, как с библиотекой Dask on Ray, делают экосистему еще мощнее.

## РАЗРАБОТКА КОНВЕЙЕРА МАШИННОГО ОБУЧЕНИЯ

Хотя в предыдущем разделе мы смогли создать простое приложение распределенной тренировки с чистого листа, было много пограничных случаев, возможностей для оптимизации производительности и возможностей по улучшению удобства использования, которые мы хотели бы урегулировать при разработке реального приложения. Как вы узнали из глав 4 и 5, фреймворк Ray располагает экосистемой библиотек, которые позволяют разрабатывать готовые к работе приложения машинного обучения. В этом разделе мы разведаем возможности по использованию библиотеки Ray Data в качестве «склеивающего слоя», чтобы разработать конвейер машинного обучения от начала до конца.

Для успешного внедрения модели машинного обучения в производстве необходимо собрать и каталогизировать данные, используя стандартные процессы ETL<sup>1</sup>. Однако это еще не конец истории: тренировка модели нередко предусматривает выделение признаков из данных перед их подачей

---

<sup>1</sup> Extract, Transform, Load (извлечь, преобразовать, загрузить). – Прим. перев.

в процесс тренировки, и то, как мы подаем данные на вход тренировки, может сильно влиять на затраты и результативность. После тренировки модели у нас также нередко возникает потребность генерировать модельные предсказания на множестве разных наборов данных – в конце концов, в этом весь смысл тренировки модели!

Хотя это, возможно, покажется всего лишь цепочкой шагов, на практике рабочий процесс обработки данных для машинного обучения представляет собой итеративный процесс экспериментирования по определению правильного набора признаков и тренировки на них высокорезультативной модели. Эффективная загрузка, преобразование и подача данных на вход тренировки и генерирование модельных предсказаний также имеют решающее значение для результативности, что напрямую отражается на затратах на вычислительно-интенсивные модели. Реализация таких конвейеров машинного обучения нередко означает сшивание нескольких систем вместе и материализацию промежуточных результатов в дистанционном хранилище между этапами. У этого подхода есть два главнейших недостатка:

- он требует оркестровки множества разных систем и программ в единый рабочий процесс, что может оказаться не под силу любому практику машинного обучения, поэтому многие люди используют системы оркестровки рабочего процесса, такие как инструмент Apache Airflow<sup>1</sup>. Хотя Airflow имеет свои преимущества, его внедрение также сопряжено с большими сложностями (в особенности в разработке);
- выполнение рабочего процесса машинного обучения в нескольких системах означает, что между каждым этапом необходимо читать из хранилища и писать в него<sup>2</sup>. Это влечет за собой значительные непроизводительные издержки из-за передачи данных и сериализации.

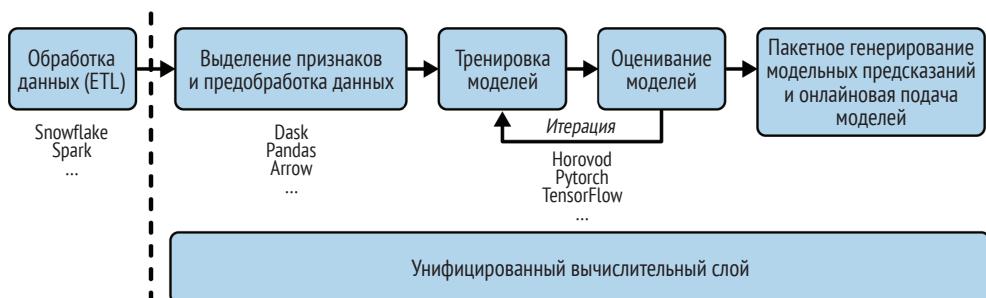
Напротив, используя фреймворк Ray, можно разрабатывать полный конвейер машинного обучения в виде одного-единственного приложения, которое может выполняться в виде одного-единственного скрипта Python, как показано на рис. 6-6. Экосистема встроенных и сторонних библиотек позволяет смешивать и сочетать функциональность, нужную для конкретного варианта использования, и разрабатывать масштабируемые, готовые к производству конвейеры. Важно отметить, что библиотека Ray Data действует как склеивающий слой, который обеспечивает эффективную загрузку данных, их предобработку и вычисление, избегая при этом дорогостоящих затрат на сериализацию и держа промежуточные данные в совместной памяти.

На рис. 6.6 показана упрощенная версия типичного рабочего процесса машинного обучения и место фреймворка Ray в этом процессе. Многочисленные этапы машинного обучения часто требуют итераций; без фреймворка Ray это означает сшивание вместе множества независимых систем в один сквозной процесс. Фреймворк Ray действует как унифицированный вычислительный слой, обеспечивающий выполнение большей части рабочего про-

<sup>1</sup> См. <https://airflow.apache.org/>.

<sup>2</sup> Еще одна проблема, связанная с опорой на многочисленные инструменты, является культурной. Передача знаний о лучших образцах практики бывает дорогостоящей, в особенности в крупных компаниях.

цесса как одно-единственное приложение.



**Рис. 6.6** ♦ Фреймворк Ray, действующий как унифицированный вычислительный слой для сложных рабочих процессов машинного обучения

В следующей главе мы собираемся показать конкретный практический пример разработки сквозного конвейера машинного обучения с использованием библиотеки Ray Data и других библиотек экосистемы Ray.

## РЕЗЮМЕ

В этой главе был представлен ключевой строительный блок во фреймворке Ray, библиотека Ray Data и ее наборы данных Dataset. Библиотека Ray Data предлагает встроенную функциональность по распределенной обработке данных, но ее истинная мощь заключается в интеграциях как с собственными, так и со сторонними библиотеками. Мы рассмотрели лишь малую часть ее функциональности. Для получения более подробной информации, справочных материалов по API библиотеки Ray Data и примеров рекомендуем обратиться к документации<sup>1</sup>.

Мы также показали простой пример распределенной тренировки классификатора библиотеки scikit-learn с использованием набора данных Dateset библиотеки Ray Data и обсудили интеграции с внешними библиотеками, такими как Dask on Ray. Наконец, мы указали на ценность разработки сквозных конвейеров машинного обучения с использованием экосистемы фреймворка Ray, которая позволяет выполнять весь рабочий процесс в одном-единственном скрипте Python. Для исследователей данных и инженеров машинного обучения это означает более быстрое время итераций, более качественные модели машинного обучения и в конечном счете более высокую ценность для предприятия.

---

<sup>1</sup> C.M., <https://oreil.ly/fmokZ>.

# Глава 7

---

# Распределенная тренировка с использованием библиотеки Ray Train

Эдвард Оукс и Ричард Ляо

В главе 6 мы затронули тему тренировки копий простой модели на сегментах данных с использованием набора данных Dataset библиотеки Ray Data, но распределенная тренировка представляет собой гораздо большее, чем это. Как мы указывали в главе 1, во фреймворке Ray есть библиотека, специально предназначенная для распределенной тренировки, – Ray Train. Она поставляется с обширным комплектом интеграций тренировки моделей машинного обучения и позволяет бесшовно масштабировать ваши эксперименты на кластерах Ray.

Мы начнем эту главу с того, что покажем причины, по которым вам, возможно, потребуется масштабировать свою тренировку моделей машинного обучения, а затем познакомим вас с разными способами это делать. После этого представим библиотеку Ray Train и рассмотрим расширенный сквозной пример. Мы также разберем несколько ключевых концепций, которые необходимо знать, чтобы использовать библиотеку Ray Train, такие как предобработчики, тренеры и контрольные точки. Наконец, мы рассмотрим несколько более продвинутых функциональных возможностей, предоставляемых библиотекой Ray Train. Как всегда, вы можете использовать блокнот Jupyter этой главы<sup>1</sup>, чтобы отслеживать изложение материала.

---

<sup>1</sup> См. <https://oreil.ly/vc5ej>.

## Основы распределенной тренировки моделей

Машинное обучение нередко требует большого объема вычислений. В зависимости от типа тренируемой вами модели, будь то градиентно-бустированное дерево или нейронная сеть, при тренировке моделей машинного обучения вы можете столкнуться с несколькими распространенными проблемами:

- время, необходимое для завершения тренировки, слишком велико;
- данные слишком велики, чтобы поместиться на одном компьютере;
- сама модель слишком велика, чтобы поместиться в одной машине.

В первом случае тренировка может быть ускорена за счет обработки данных с увеличенной пропускной способностью. Некоторые алгоритмы машинного обучения, такие как нейронные сети, могут параллелизовать части вычислений с целью ускорения тренировки<sup>1</sup>.

Во втором случае выбранный вами алгоритм может потребовать, чтобы вы поместили все имеющиеся данные из набора в память, но выделенной одноузловой памяти может оказаться недостаточно. Если это окажется так, то вам нужно будет разделить данные по нескольким узлам и выполнять тренировку распределенно. С другой стороны, иногда алгоритм, возможно, не требует распределения данных, но если вы изначально используете систему распределенных баз данных, то вам все равно будет нужен фреймворк тренировки, который может задействовать ваши распределенные данные.

Если ваша модель не помещается в одну машину, то вам, возможно, потребуется ее разбить на несколько частей, разбросанных по нескольким машинам. Подход на основе разбики моделей по нескольким машинам называется *модельным параллелизмом*. Для столкновения с этой проблемой сначала нужна модель, которая была бы достаточно большой, чтобы не помещаться в одну машину. Обычно крупным компаниям, таким как Google или Meta, требуется модельный параллелизм, и они также опираются на собственные технологические решения, чтобы управлять распределенной тренировкой.

Первые две проблемы часто возникают гораздо раньше на этапе разработки модели машинного обучения, чем третья. Технологические решения, которые мы только что набросали для урегулирования этих проблем, подпадают под зонтик тренировки с параллелизмом данных. Вместо того чтобы разбивать модель на несколько машин, упор делается на распределенных данных, чтобы ускорять тренировку.

Что касается первой проблемы, то если вы можете ускорить свой процесс тренировки (надо надеяться, с минимальной потерей точности или вообще без нее), и вы можете сделать это эффективно с точки зрения затрат, то почему бы не пойти на это? И если у вас есть распределенные данные, будь то в силу необходимости вашего алгоритма либо способа хранения данных, то вам нужно некое решение по тренировке, которое с этим справляется. Как вы увидите, библиотека Ray Train была разработана в целях эффективной тренировки с параллелизмом данных. На рис. 7.1 кратко представлены два базовых типа распределенной тренировки.

---

<sup>1</sup> В частности, это относится к вычислению градиента в нейронных сетях.

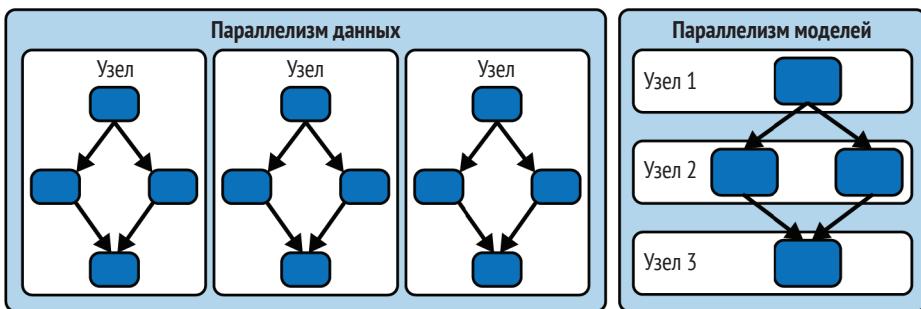


Рис. 7.1 ♦ Параллелизм данных в сравнении с параллелизмом моделей в распределенной тренировке

## ВВЕДЕНИЕ В БИБЛИОТЕКУ RAY TRAIN НА ПРИМЕРЕ

Ray Train – это библиотека, служащая для распределенной тренировки с параллелизмом данных в рамках фреймворка Ray. Она предлагает ключевые инструменты для различных этапов процесса тренировки, от обработки признаков до масштабируемой тренировки, интеграции с инструментами отслеживания моделей машинного обучения и механизмов экспорта моделей.

В базовом конвейере тренировки модели машинного обучения вы будете использовать следующие ниже ключевые компоненты библиотеки Ray Train:

### Тренеры

В библиотеке Ray Train есть несколько классов-тренеров, которые позволяют проводить распределенные тренировки. Тренеры – это классы-обертки вокруг сторонних фреймворков тренировки, таких как XGBoost, Pytorch и TensorFlow, обеспечивающие интеграцию с ключевыми акторами Ray (с целью распределения), библиотекой Ray Tune и библиотекой Ray Data.

### Предсказатели

Получив натренированную модель, вы сможете ее использовать для получения предсказаний. Для пакетов входных данных вы используете так называемые пакетные предсказатели, которые также используются для оценивания результативности модели на валидационном наборе.

В дополнение к этому библиотека Ray Train предоставляет несколько распространенных объектов-предобработчиков и служебных инструментов для обработки объектов Dataset в пригодные для потребления тренерами признаки. Наконец, библиотека Ray Train предоставляет класс – *контрольную точку* (Checkpoint), который позволяет сохранять и восстанавливать состояние тренировочного прогона. В нашем первом пошаговом примере мы не будем использовать никаких предобработчиков, но позже рассмотрим их подробнее.

Библиотека Ray Train разработана с первоклассной поддержкой тренировки на крупных наборах данных. Следуя той же философии, согласно которой

вам не нужно думать о том, как параллелизовать свой исходный код, вы можете просто «подсоединить» свой крупный набор данных к библиотеке Ray Train, не думая о том, как принимать и передавать ваши данные в разные параллельные работники.

Давайте применим эти компоненты на практике, пройдя в пошаговом режиме наш первый пример с библиотекой Ray Train. В части загрузки тренировочных данных мы собираемся использовать наши знания из главы 6 и широко использовать библиотеку Ray Data.

## Предсказание больших чаевых в поездках на нью-йоркском такси

В этом разделе пошагово рассматривается практический сквозной пример разработки конвейера глубокого обучения с использованием фреймворка Ray. Мы построим модель двоичной классификации, чтобы предсказывать, не приведет ли поездка на такси к получению больших чаевых (> 20 % от стоимости поездки), используя данные о поездках, собранных Общественной комиссией Нью-Йорка по такси и лимузинам (TLC)<sup>1</sup>. Наш рабочий процесс будет очень похож на рабочий процесс типичного практика машинного обучения:

- 1) загрузить данные, выполнить базовую предобработку и вычислить признаки, которые мы будем использовать в модели;
- 2) определить нейронную сеть и натренировать ее с помощью распределенной тренировки с параллелизмом данных;
- 3) применить натренированную нейронную сеть к свежему пакету данных.

В примере будут использоваться библиотека Dask on Ray и тренировка нейронной сети PyTorch, но обратите внимание, что здесь нет ничего специфичного ни для одной из этих библиотек: библиотеки Ray Data и Ray Train можно использовать с широким спектром популярных инструментов машинного обучения. Для отслеживания работы примера исходного кода этого раздела следует установить библиотеки Ray, PyTorch и Dask:

```
pip install "ray[data,train]==2.2.0" "dask==2022.2.0" "torch==1.12.1"  
pip install "xgboost==1.6.2" "xgboost-ray>=0.1.10"
```

В следующих далее примерах мы будем загружать данные с локального диска, чтобы упростить выполнение примеров на вашем компьютере. Данные доступны в репозитории книги на GitHub<sup>2</sup>. Пути к файлам основаны на том, что вы клонировали репозиторий и выполняете примеры из его каталога верхнего уровня.

---

<sup>1</sup> См. <https://oreil.ly/nrJgK>.

<sup>2</sup> См. <https://oreil.ly/DhcUB>.

## Загрузка/предобработка данных и выделение признаков

Первым шагом в тренировке модели являются загрузка и предобработка данных. Для этого мы будем использовать библиотеку Dask on Ray, первый пример которой вы уже встречали в главе 6. Библиотека Dask on Ray предоставляет удобный API компонента для работы с наборами данных DataFrame, возможность вертикально масштабировать предобработку в кластере и эффективно ее использовать в операциях тренировки и генерирования модельных предсказаний. Вот наш исходный код предобработки данных и построения признаков для нашей модели, определенный в одной-единственной функции `load_dataset`:

```
import ray
from ray.util.dask import enable_dask_on_ray

import dask.dataframe as dd

LABEL_COLUMN = "is_big_tip"
FEATURE_COLUMNS = ["passenger_count", "trip_distance", "fare_amount",
                    "trip_duration", "hour", "day_of_week"]

enable_dask_on_ray()

def load_dataset(path: str, *, include_label=True):
    columns = ["tpep_pickup_datetime", "tpep_dropoff_datetime",
               "tip_amount", "passenger_count",
               "trip_distance", "fare_amount"]
    df = dd.read_parquet(path, columns=columns) ❶

    df = df.dropna() ❷
    df = df[(df["passenger_count"] <= 4) &
            (df["trip_distance"] < 100) &
            (df["fare_amount"] < 1000)]

    df["tpep_pickup_datetime"] = dd.to_datetime(df["tpep_pickup_datetime"])
    df["tpep_dropoff_datetime"] = dd.to_datetime(df["tpep_dropoff_datetime"])

    df["trip_duration"] = (df["tpep_dropoff_datetime"] -
                           df["tpep_pickup_datetime"]).dt.seconds
    df = df[df["trip_duration"] < 4 * 60 * 60] # 4 hours.
    df["hour"] = df["tpep_pickup_datetime"].dt.hour
    df["day_of_week"] = df["tpep_pickup_datetime"].dt.weekday ❸

    if include_label:
        df[LABEL_COLUMN] = df[“tip_amount”] > 0.2 * df[“fare_amount”] ❹

    df = df.drop( ❺
                 columns=["tpep_pickup_datetime",
                           "tpep_dropoff_datetime", "tip_amount"]
             )

    return ray.data.from_dask(df).repartition(100) ❻
```

- ① Исключить неиспользуемые столбцы первоначального набора данных DataFrame библиотеки Dask, загруженного из файлов Parquet.
- ② Выполнить базовую очистку и удалить нулевые и аномальные (выбросные) значения.
- ③ Добавить три новых признака: продолжительность поездки, час начала поездки и день недели.
- ④ Рассчитать столбец метки: составляли ли чаевые больше или меньше 20 % от стоимости поездки.
- ⑤ Исключить все неиспользуемые столбцы.
- ⑥ Вернуть реорганизованный набор данных Dataset библиотеки Ray Data, созданный из набора данных DataFrame библиотеки Dask.

Исходный код содержит базовую загрузку и очистку данных, а также преобразование некоторых столбцов в формат, который можно использовать в качестве признаков в модели машинного обучения. Например, мы преобразовываем даты и время подбора пассажира, которые предоставляются в виде строкового литерала, в три числовых параметра: продолжительность поездки (`trip_duration`), час (`hour`) и день недели (`day_of_week`). Это упрощается благодаря встроенной поддержке библиотекой Dask служебных функций Python по обработке даты/времени<sup>1</sup>. Если эти данные будут использоваться для тренировки, то нам также нужно будет вычислить столбец метки.

Наконец, после того как мы вычислили предобработанный набор данных DataFrame библиотеки Dask, мы преобразовываем его в набор данных Dataset библиотеки Ray Data, чтобы позже использовать его в наших процессах тренировки модели и генерирования модельных предсказаний.

## Определение модели глубокого обучения

Теперь, когда мы очистили и подготовили данные, необходимо определить модельную архитектуру, которую мы будем использовать для этой модели. На практике указанная работа, скорее всего, будет итеративным процессом и потребует исследования современного состояния аналогичных задач. Ради нашего примера мы упростим работу и будем использовать базовую нейронную сеть PyTorch, которую назовем предсказателем чаевых за проезд (`FarePredictor`). Нейронная сеть имеет три линейных преобразования, начиная с размерности нашего вектора признаков, а затем она выводит значение от 0 до 1, используя сигмоидную активационную функцию. В целях улучшения тренировки в сети также используются слои *пакетной нормализации*. Выходное значение будет округляться, производя двоичное предсказание в отношении того, приведет ли поездка к большим чаевым:

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class FarePredictor(nn.Module):
    def __init__(self):
```

---

<sup>1</sup> См. <https://oreil.ly/sZPhd>.

```

super().__init__()

self.fc1 = nn.Linear(6, 256)
self.fc2 = nn.Linear(256, 16)
self.fc3 = nn.Linear(16, 1)

self.bn1 = nn.BatchNorm1d(256)
self.bn2 = nn.BatchNorm1d(16)

def forward(self, x):
    x = F.relu(self.fc1(x))
    x = self.bn1(x)
    x = F.relu(self.fc2(x))
    x = self.bn2(x)
    x = torch.sigmoid(self.fc3(x))

    return x

```

## Распределенная тренировка с помощью библиотеки Ray Train

Теперь, когда мы определили нейросетевую архитектуру, нам нужен способ эффективно натренировать ее на наших данных. Этот набор данных очень велик, поэтому лучше всего проводить *тренировку с параллелизмом данных*.

Это означает, что мы тренируем модель параллельно на нескольких машинах, на каждой из которых есть копия модели и подмножество данных. Мы будем использовать библиотеку Ray Train, чтобы определить масштабируемый процесс тренировки, в котором под капотом будет использоваться механизм DataParallel<sup>1</sup> библиотеки PyTorch. Здесь мы не будем вдаваться в концептуальные детали процесса тренировки, но обсудим их в разделах, следующих за этим сквозным примером.



В следующем ниже примере используется импорт из модуля `ray.air`. Мы упоминали инструментарий Ray AIR в главе 1 и формально представим его в главе 10. На данный момент рассматривайте этот модуль как полезный служебный инструмент для определения и выполнения ваших процессов распределенной тренировки.

В частности, мы используем так называемый сеанс (`session`) AIR, который можно применять для сообщения (`report`) о метриках, собранных в процессе тренировки. Это соответствует шаблону использования, аналогичному API `tune.report`, который мы обсуждали в главе 5.

Первым делом нужно определить ключевую логику тренировки на пакете данных на каждом работнике в каждую эпоху. Она будет заключаться в принятии локального сегмента полного набора данных, его пропускании через локальную копию модели и выполнении обратного распространения, чтобы обновлять веса модели. После каждой эпохи работник будет использовать

---

<sup>1</sup> См. [https://oreil.ly/A\\_xcS](https://oreil.ly/A_xcS).

служебные функции библиотеки Ray Train, чтобы сообщать о результате и сохранять текущие веса модели для последующего использования:

```
from ray.air import session
from ray.air.config import ScalingConfig
import ray.train as train
from ray.train.torch import TorchCheckpoint, TorchTrainer

def train_loop_per_worker(config: dict): ❶
    batch_size = config.get("batch_size", 32)
    lr = config.get("lr", 1e-2)
    num_epochs = config.get("num_epochs", 3)

    dataset_shard = session.get_dataset_shard("train") ❷

    model = FarePredictor()
    dist_model = train.torch.prepare_model(model) ❸

    loss_function = nn.SmoothL1Loss()
    optimizer = torch.optim.Adam(dist_model.parameters(), lr=lr)

    for epoch in range(num_epochs): ❹
        loss = 0
        num_batches = 0
        for batch in dataset_shard.iter_torch_batches(❺
            batch_size=batch_size, dtypes=torch.float
        ):
            labels = torch.unsqueeze(batch[LABEL_COLUMN], dim=1)
            inputs = torch.cat([
                torch.unsqueeze(batch[f], dim=1)
                for f in FEATURE_COLUMNS,
                dim=1
            ])
            output = dist_model(inputs)
            batch_loss = loss_function(output, labels)
            optimizer.zero_grad()
            batch_loss.backward()
            optimizer.step()

            num_batches += 1
            loss += batch_loss.item()

        session.report(❻
            {"epoch": epoch, "loss": loss},
            checkpoint=TorchCheckpoint.from_model(dist_model)
        )
    )
```

- ❶ Передать словарь config в цикл тренировки, чтобы задать несколько параметров времени выполнения.
- ❷ Извлечь сегмент данных для текущего работника, используя служебную функцию библиотеки Ray Train get\_data\_shard.
- ❸ Подготовить модель PyTorch для распределенной тренировки, применив функцию prepare\_model.

- ④ Определить стандартный для PyTorch цикл тренировки, прокручивающий пакеты данных в цикле и выполняющий обратное распространение.
- ⑤ Единственной нестандартной частью является использование `iter_torch_batches` для прокручивания сегмента данных в цикле.
- ⑥ После каждой эпохи сообщать о вычисленной потере (`loss`) и контрольной точке модели с использованием сеанса (`session`) фреймворка Ray.

В случае если вы незнакомы с библиотекой PyTorch, обратите внимание, что исходный код между определением функции потери (`loss_function`) и агрегированием пакетной потери (`batch_loss`) с нашей потерей (`loss`) является стандартным циклом тренировки модели PyTorch (за исключением специфичного для фреймворка Ray итеративного прокручивания пакетов сегмента набора данных).

Теперь, когда процесс тренировки определен, нужно загрузить тренировочные и валидационные данные для их подачи работникам тренировки. Здесь мы вызываем определенную ранее функцию `load_dataset`, которая будет выполнять предобработку и выделять признаки<sup>1</sup>.

Этот набор данных передается в `TorchTrainer` вместе с несколькими конфигурационными параметрами, такими как размер пакета, число эпох и число используемых работников. Каждый работник будет иметь доступ к сегменту данных локально и сможет прокручивать его в цикле. После завершения тренировки мы можем извлечь окончательную натренированную контрольную точку из возвращенного объекта `result`:

```
trainer = TorchTrainer(
    train_loop_per_worker=train_loop_per_worker, ❶
    train_loop_config={❷
        "lr": 1e-2, "num_epochs": 3, "batch_size": 64
    },
    scaling_config=ScalingConfig(num_workers=2), ❸
    datasets={❹
        "train":
            load_dataset("nyc_tlc_data/yellow_tripdata_2020-01.parquet")
    },
)
result = trainer.fit() ❺
trained_model = result.checkpoint
```

- ❶ Каждый тренер `TorchTrainer` на входе ожидает цикл тренировки в расчете на работника (`train_loop_per_worker`).
- ❷ Если в вашем цикле тренировки используется словарь `config`, то его дополнительно можно указать как конфигурацию цикла тренировки (`train_loop_config`).
- ❸ Каждому тренеру (`Trainer`) из библиотеки Ray Train требуется так называемая конфигурация масштабирования (`ScalingConfig`), чтобы знать, как масштабировать тренировку в вашем кластере Ray.

---

<sup>1</sup> Для тестирования исходный код загружает только подмножество данных; в целях крупномасштабного выполнения мы бы использовали все разделы данных при вызове функции `load_dataset` и увеличили бы число работников (`num_workers`) при тренировке модели.

- ④ Еще одним обязательным аргументом каждого тренера является словарь `datasets`. Здесь мы определяем набор данных `Dataset` под названием "`train`", и именно его мы и используем в нашем цикле тренировки.
- ⑤ Для запуска тренировки нужно просто вызвать метод `.fit()` на тренере `TorchTrainer`.

Последняя строка экспортирует натренированную модель в качестве контрольной точки для последующего использования в нижестоящих приложениях, таких как подача модели в качестве службы и генерирование модельных предсказаний. Библиотека Ray Train генерирует эти контрольные точки, чтобы сериализовывать промежуточное состояние тренировки. Контрольные точки могут включать в себя как модели, так и другие артефакты тренировки, такие как предобработчики.

## Распределенное пакетное генерирование модельных предсказаний

После того как мы натренировали модель и получили наилучшую точность, следующим шагом будет ее применение на практике. Иногда это означает запуск службы с низкой задержкой, что мы рассмотрим в главе 8, но нередко работа заключается в применении модели к пакетам данных по мере их поступления.

Давайте возьмем веса из натренированной модели (`trained_model`) и применим их к новому пакету данных (в данном случае это будет просто еще один сегмент того же публичного набора данных). Для этого сначала нужно загрузить данные, предварительно их обработать и выделить из них признаки таким же образом, как мы это делали перед тренировкой. Затем мы загрузим нашу модель и отобразим (`map`) ее на весь набор данных целиком. Библиотека Ray Data позволяет выполнять это эффективно с помощью акторов Ray, даже используя графические процессоры, путем изменения одного параметра. Мы лишь загружаем контрольную точку натренированной модели и вызываем на ней функцию `.predict_pipelined()`. Она будет использовать набор данных `Dataset` для распределенного пакетного генерирования модельных предсказаний по всем данным:

```
from ray.train.torch import TorchPredictor
from ray.train.batch_predictor import BatchPredictor

batch_predictor = BatchPredictor(trained_model, TorchPredictor)
ds = load_dataset(
    "nyc_tlc_data/yellow_tripdata_2021-01.parquet", include_label=False)

batch_predictor.predict_pipelined(ds, blocks_per_window=10)
```

Приведенный выше пример показал применение библиотек Ray Train и Ray Data для реализации сквозного рабочего процесса машинного обучения в виде одного-единственного приложения. Мы смогли структурировать набор данных, натренировать и валидировать модель машинного обучения,

а затем применить эту модель к другому набору данных в одном-единственном скрипте Python. Библиотека Ray Data с ее набором данных Dataset действовала как склеивающий слой, соединяющий разные этапы и избегающий дорогостоящих затрат на сериализацию между ними. Мы также использовали контрольные точки, чтобы сохранять модели, и выполнили заявку на выполнение работы по пакетному предсказанию в рамках библиотеки Ray Train, применив модель к новому набору данных.

Теперь, когда вы увидели первый пример с библиотекой Ray Train, давайте подробнее рассмотрим его главнейшую абстракцию – объект Trainer.

## ПОДРОБНЕЕ О ТРЕНЕРАХ В БИБЛИОТЕКЕ RAY TRAIN

Как вы увидели в примере использования TorchTrainer, тренеры – это специфичные для фреймворка классы, которые выполняют тренировку модели распределенным образом. Все классы библиотеки Ray Trainer имеют общий интерфейс. На данный момент достаточно знать о двух аспектах этого интерфейса, а именно:

- методе .fit(), который выполняет подгонку заданного тренера (Trainer) к заданным наборам данных, конфигурации и желаемым свойствам масштабирования;
- свойстве .checkpoint, которое возвращает объект Checkpoint для этого тренера.

Тренеры интегрируются с распространенными фреймворками машинного обучения, такими как PyTorch, Hugging Face, TensorFlow, Horovod, scikit-learn и др. Есть даже тренер специально для моделей библиотеки RLLib, который мы здесь не рассматриваем. Давайте обсудим еще один пример с PyTorch, чтобы выделить конкретные аспекты API Trainer, с акцентом на то, как выполнять миграцию существующей модели PyTorch в библиотеку Ray Train.

### ФРЕЙМВОРКИ ГРАДИЕНТНОГО БУСТИРОВАНИЯ

Библиотека Ray Train также предлагает поддержку фреймворков градиентно-бустированых деревьев решений.

XGBoost – это библиотека оптимизированного распределенного градиентного бустирования, разработанная с учетом высокой эффективности, гибкости и переносимости. В ней реализованы алгоритмы машинного обучения в рамках фреймворка градиентного бустирования. XGBoost обеспечивает параллельное бустирование дерева, которое быстро и точно решает многие задачи в области обработки данных.

LightGBM – это фреймворк градиентного бустирования, основанный на древовидных самообучающихся алгоритмах. По сравнению с XGBoost, это относительно новый фреймворк, но он быстро становится популярным как в академических, так и в производственных вариантах применения.

Указанные фреймворки можно использовать с помощью классов соответственно XGBoostTrainer и LightGBMTrainer.

В этом примере мы хотим сосредоточиться на деталях самой библиотеки Ray Train, поэтому будем использовать гораздо более простой набор тренировочных данных и небольшую нейронную сеть, которая на входе принимает случайный шум. Мы определяем трехслойную нейронную сеть (NeuralNetwork), явно заданный цикл тренировки (training\_loop), аналогичный представленному в предыдущем разделе, который можно использовать для тренировки модели. Для наглядности мы извлекаем исходный код тренировки, выполняемый для каждой эпохи, во вспомогательную функцию, именуемую train\_one\_epoch:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from ray.data import from_torch

num_samples = 20
input_size = 10
layer_size = 15
output_size = 5
num_epochs = 3

class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(input_size, layer_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(layer_size, output_size)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

    def train_data():
        return torch.randn(num_samples, input_size) ①

    input_data = train_data()
    label_data = torch.randn(num_samples, output_size)
    train_dataset = from_torch(input_data) ②

    def train_one_epoch(model, loss_fn, optimizer): ③
        output = model(input_data)
        loss = loss_fn(output, label_data)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    def training_loop(): ④
        model = NeuralNetwork()
        loss_fn = nn.MSELoss()
        optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
        for epoch in range(num_epochs):
            train_one_epoch(model, loss_fn, optimizer)
```

- ❶ Использовать случайно сгенерированный набор данных.
- ❷ Создать набор данных Dataset из этих данных с помощью `from_torch`.
- ❸ Извлечь исходный код PyTorch тренировки в течение одной эпохи во вспомогательную функцию.
- ❹ Этот цикл тренировки можно выполнять как есть для тренировки вашей модели PyTorch на одной машине.

Если вы хотите распределить свою тренировку без библиотеки Ray Train, то, как правило, вам нужно будет сделать две следующие вещи:

- создать бэкенд, который координирует межпроцессное взаимодействие;
- создать экземпляры нескольких параллельных процессов на каждом узле, на который вы хотите распределить свою тренировку.

Для сравнения, давайте посмотрим, насколько просто использовать библиотеку Ray Train для распределения процесса тренировки.

## Миграция в библиотеку Ray Train с минимальными изменениями исходного кода

С помощью библиотеки Ray Train можно внести одностороннее изменение в исходный код, чтобы обеспечить под капотом как межпроцессное взаимодействие, так и создание экземпляров процесса.

Изменение исходного кода производится путем вызова функции `prepare_model` в модели PyTorch, которую планируется тренировать. Это изменение буквально является единственным отличием между определенным ранее циклом тренировки (`training_loop`) и следующим ниже циклом распределенной тренировки (`distributed_training_loop`):

```
from ray.train.torch import prepare_model

def distributed_training_loop():
    model = NeuralNetwork()
    model = prepare_model(model) ❶
    loss_fn = nn.MSELoss()
    optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
    for epoch in range(num_epochs):
        train_one_epoch(model, loss_fn, optimizer)
```

- ❶ Подготовить модель для распределенной тренировки, вызвав функцию `prepare_model`.

Затем можно создать экземпляр модели `TorchTrainer`, которая имеет три обязательных аргумента:

```
train_loop_per_worker
```

Эта функция тренирует вашу модель в каждом работнике, имеет доступ к предоставленному словарю `datasets` и может принимать optionalный словарь `config`, который можно передавать тренеру в качестве конфигурации цикла тренировки (`train_loop_config`). Указанная функция обычно сообщает о метриках, например через `session`.

**datasets**

Этот словарь может содержать несколько ключей с наборами данных Dataset в качестве значений. При этом он является гибким, давая возможность иметь тренировочные данные, валидационные данные или любые другие виды данных, необходимые для вашего цикла тренировки.

**scaling\_config**

Этот объект ScalingConfig задает способ масштабирования вашей тренировки. Например, с помощью num\_workers можно указать число работников тренировки, а с помощью флага use\_gpu сообщить о необходимости задействовать графические процессоры. Мы остановимся на этом подробнее в следующем разделе.

Ниже приводится настройка тренера в нашем примере:

```
from ray.air.config import ScalingConfig
from ray.train.torch import TorchTrainer

trainer = TorchTrainer(
    train_loop_per_worker=distributed_training_loop,
    scaling_config=ScalingConfig(
        num_workers=2,
        use_gpu=False
    ),
    datasets={"train": train_dataset}
)
result = trainer.fit()
```

После инициализации тренера можно вызвать функцию fit(), которая исполнит тренировку в вашем кластере Ray. На рис. 7.2 представлена краткая информация о работе тренера TorchTrainer.



**Рис. 7.2** ♦ Работа с TorchTrainer требует, чтобы вы указали цикл тренировки, наборы данных и конфигурацию масштабирования

## Горизонтальное масштабирование тренеров

Философия библиотеки Ray Train заключается в том, что пользователю не нужно думать о том, как параллелизировать свой исходный код. Детализа-

ция конфигурации масштабирования (`scaling_config`) позволяет масштабировать тренировку без написания распределенной логики и декларативно указывать используемые тренером *вычислительные ресурсы*. Самое приятное в этой детализации то, что вам не нужно думать об опорном оборудовании. В частности, соответствующим образом указав в своей конфигурации масштабирования (`ScalingConfig`) параметры узлов своего кластера, можно использовать сотни работников:

```
import ray
from ray.air.config import ScalingConfig
from ray.train.xgboost import XGBoostTrainer

ray.init(address="auto") ①

scaling_config = ScalingConfig(num_workers=200, use_gpu=True) ②

trainer = XGBoostTrainer(③
    scaling_config=scaling_config,
    # ...
)
```

- ① Подсоединиться здесь к существующему большому кластеру Ray.
- ② Определить конфигурацию масштабирования (`ScalingConfig`) в соответствии с доступными ресурсами кластера.
- ③ Затем можно задействовать интеграцию библиотеки Ray Train с библиотекой XGBoost с целью тренировки модели в этом кластере.

## Предобработка с помощью библиотеки Ray Train

Предварительная обработка данных – распространенный метод преобразования сырых данных в признаки для модели машинного обучения, и мы видели много примеров тому в этой и предыдущей главах. До сего момента мы предобрабатывали наши данные «вручную», разрабатывая конкретно-прикладные функции преобразования данных в нужный формат. Но библиотека Ray Train имеет несколько встроенных предобработчиков, предназначенных для распространенных случаев использования, а также предоставляет интерфейсы для определения своей собственной конкретно-прикладной логики.

`Processor` – это стержневой класс, предлагаемый библиотекой Ray Train для предобработки данных. Каждый предобработчик имеет следующие API:

`.transform()`

Используется для обработки и применения обрабатывающего преобразования к набору данных.

`.fit()`

Используется для вычисления и сохранения агрегированного состояния набора данных в предобработчике. Возвращает `self` для выстраивания в цепочку.

**.fit\_transform()**

Синтаксический сахар для выполнения преобразований, требующих агрегированного состояния. В случае конкретных предобработчиков может быть оптимизирован на уровне реализации.

**.transform\_batch()**

Используется для применения одного и того же преобразования на пакетах данных с целью предсказания.

На практике часто возникает потребность в использовании одних и тех же операций предобработки данных во время тренировки модели и во время ее подачи в качестве службы. Перекос между тренировкой и подачей<sup>1</sup> – это главнейшая проблема в развертывании моделей машинного обучения, он описывает ситуацию, когда существует разница между результативностью во время тренировки модели и результативностью во время ее подачи как службы. Этот перекос часто вызван несоответствием между способами обработки данных в конвейерах тренировки и подачи. Следовательно, есть потребность в наличии обработки данных, согласующейся между тренировкой и подачей.

В связи с этим можно использовать предыдущие предобработчики, передавая их конструктору тренера (`trainer`). Это означает, что после того, как был создан предобработчик (`preprocessor`), нет необходимости применять его к своему набору данных `Dataset` вручную. Вместо этого можно передать его тренеру, и библиотека Ray Train позаботится о его применении в распределенном режиме. Вот как это работает схематично:

```
from ray.data.preprocessors import StandardScaler
from ray.train.xgboost import XGBoostTrainer

trainer = XGBoostTrainer(
    preprocessor=StandardScaler(...),
    # ...
)
result = trainer.fit()
```

Некоторые операторы предобработки, такие как кодировщики в одно активное состояние<sup>2</sup>, легко выполняются при тренировке модели и переносятся на ее подачу как службы. Однако с другими операторами, такими как операторы стандартизации, немного сложнее, поскольку вы же не будете выполнять перемалывание больших объемов данных (чтобы найти среднее значение в определенном столбце) во время подачи.

К счастью, предоставляемые в библиотеке Ray Train предобработчики поддаются сериализации, вследствие чего можно легко добиваться согласованности от тренировки модели до ее подачи как службы, просто сериализуя эти операторы. Например, предобработчик можно просто консервировать (`pickle`), как показано ниже:

---

<sup>1</sup> Англ. Training-serving skew. – Прим. перев.

<sup>2</sup> Англ. one-hot encoder. – Прим. перев.

```
import pickle
from ray.data.preprocessors import StandardScaler

preprocessor=StandardScaler(...)
pickle.dumps(preprocessor)
```

Далее давайте обсудим конкретный пример процедуры тренировки с использованием предобработчиков, а также покажем, как настраивать гиперпараметры тренера.

## Интеграция тренеров с библиотекой Ray Tune

Библиотека Ray Train обеспечивает интеграцию с библиотекой Ray Tune, которая позволяет выполнять гиперпараметрическую оптимизацию всего в нескольких строках исходного кода. Библиотека Tune создает одну попытку на каждую гиперпараметрическую конфигурацию. В каждой попытке инициализируется новый тренер, который выполняет функцию тренировки со своей генерированной конфигурацией.

В следующем ниже исходном коде мы создаем тренера XGBoostTrainer и указываем диапазоны для распространенных гиперпараметров. В частности, в сценарии тренировки мы собираемся выбрать один из двух разных *предобработчиков*. Если быть точным, мы будем использовать стандартный шкалировщик (StandardScaler), который транслирует и шкалирует каждый указанный столбец на его среднее значение и стандартное отклонение (следовательно, результирующие столбцы будут подчиняться стандартному нормальному распределению), и минимаксный шкалировщик (MinMaxScaler), который просто шкалирует каждый столбец в диапазон [0, 1].

Вот соответствующее параметрическое пространство, в котором мы будем выполнять поиск далее:

```
import ray

from ray.air.config import ScalingConfig
from ray import tune
from ray.data.preprocessors import StandardScaler, MinMaxScaler

dataset = ray.data.from_items(
    [{"X": x, "Y": 1} for x in range(0, 100)] +
    [{"X": x, "Y": 0} for x in range(100, 200)])
)

prep_v1 = StandardScaler(columns=["X"])
prep_v2 = MinMaxScaler(columns=["X"])

param_space = {
    "scaling_config": ScalingConfig(
        num_workers=tune.grid_search([2, 4]),
        resources_per_worker={
            "CPU": 2,
            "GPU": 0,
```

```
        },
    ),
    "preprocessor": tune.grid_search([prep_v1, prep_v2]),
    "params": {
        "objective": "binary:logistic",
        "tree_method": "hist",
        "eval_metric": ["logloss", "error"],
        "eta": tune.loguniform(1e-4, 1e-1),
        "subsample": tune.uniform(0.5, 1.0),
        "max_depth": tune.randint(1, 9),
    },
}
```

Теперь можно создать тренера (`Trainer`), как и раньше, на этот раз используя `XGBoostTrainer`, а затем передав его экземпляру настройщика (`Tuner`) из библиотеки Ray Tune, подгонку (`.fit()`) которого можно выполнить точно так же, как и самого тренера (`trainer`):

```
from ray.train.xgboost import XGBoostTrainer
from ray.air.config import RunConfig
from ray.tune import Tuner

trainer = XGBoostTrainer(
    params={},
    run_config=RunConfig(verbose=2),
    preprocessor=None,
    scaling_config=None,
    label_column="Y",
    datasets={"train": dataset}
)

tuner = Tuner(
    trainer,
    param_space=param_space,
)
results = tuner.fit()
```

Обратите внимание, что здесь мы используем другой компонент тренеров Ray, который вы раньше не встречали, – конфигурацию выполнения (`RunConfig`). Указанная конфигурация используется для всех опций выполнения тренера, в нашем случае для детализации журналирования эксперимента (0 означает отсутствие; 1 выдает только обновления статуса; 2 по умолчанию выдает обновления статуса и краткие результаты; 3 выдает подробные результаты).

По сравнению с другими тенническими решениями по распределенной настройке гиперпараметров, библиотеки Ray Tune и Ray Train обладают некоторыми уникальными особенностями. Техническое решение фреймворка Ray является отказоустойчивым и обладает способностью задавать набор данных и предобработчик в качестве параметра, а также корректировать число работников во время тренировки.

## Использование обратных вызовов для мониторинга тренировки

Обследуя еще одну функциональную возможность библиотеки Ray Train, вероятно, вы захотите подключить свой тренировочный исходный код к своему любимому фреймворку управления экспериментами. Библиотека Ray Train предоставляет интерфейс, чтобы получать промежуточные результаты и выполнять обратные вызовы для их обработки или регистрации в журнале. Указанный интерфейс сопровождается встроенными обратными вызовами для популярных фреймворков отслеживания, но есть возможность реализовывать свой собственный обратный вызов через интерфейс `LoggerCallback` в библиотеке Tune.

Например, журналировать результаты можно в формате JSON с помощью `JsonLoggerCallback`, в TensorBoard с помощью `TBXLoggerCallback` или в MLflow с помощью обратного вызова `MLflowLogger`<sup>1</sup>. В следующем ниже примере показано, как использовать все три в одном тренировочном прогоне, указав список обратных вызовов:

```
from ray.air.callbacks.mlflow import MLflowLoggerCallback
from ray.tune.logger import TBXLoggerCallback, JsonLoggerCallback

training_loop = ...
trainer = ...

trainer.fit(
    training_loop,
    callbacks=[
        MLflowLoggerCallback(),
        TBXLoggerCallback(),
        JsonLoggerCallback()
])

```

## РЕЗЮМЕ

В этой главе мы обсудили основы распределенной тренировки моделей и показали, как выполнять тренировку с параллелизмом данных с помощью библиотеки Ray Train. Мы познакомили вас с подробным примером, в котором на интересном наборе данных использовались как библиотека Ray Data, так и библиотека Ray Train. В частности, продемонстрировали, как применять библиотеку Dask on Ray для загрузки, предобработки ваших наборов данных

<sup>1</sup> MLflow и TensorBoard – это проекты с открытым исходным кодом для отслеживания и визуализации экспериментов по машинному обучению. Они очень полезны для мониторинга последовательного изменения вашей модели машинного обучения.

и выделения из них признаков, а затем использовать библиотеку Ray Train для запуска цикла распределенной тренировки модели PyTorch. Затем мы подробнее остановились на тренерах Ray и показали, как они интегрируются с библиотекой Ray Tune посредством настройщиков, как их применять с предобработчиками и как использовать функции обратного вызова для мониторинга тренировки.

# Глава 8

---

# Онлайновое генерирование модельных предсказаний с использованием библиотеки Ray Serve

Эдвард Оукс

В главах 6 и 7 вы научились использовать фреймворк Ray для обработки данных, тренировки моделей машинного обучения и их применения в режиме пакетного генерирования модельных предсказаний. Однако многие наиболее интересные варианты использования машинного обучения связаны с *онлайновым генерированием модельных предсказаний*<sup>1</sup>.

Онлайновое генерирование модельных предсказаний – это процесс использования моделей машинного обучения, служащий для усиления конечных точек API, с которыми пользователи взаимодействуют прямо либо косвенно. Это важно в ситуациях, когда решающее значение имеет задержка: бывает невозможно просто брать модели, применять их к данным за кулисами и подавать результаты по запросу. Существует множество реальных примеров использования, когда онлайнное генерирование модельных предсказаний может приносить большую пользу, например:

## *Рекомендательные системы*

Предоставление рекомендаций продуктов (например, интернет-магазинам) или контента (например, социальным сетям) является простым

---

<sup>1</sup> Англ. online inference. – Прим. перев.

способом использования машинного обучения. Хотя это можно делать и в офлайновом режиме, рекомендательные системы часто выигрывают за счет реагирования на предпочтения пользователей в реальном времени. Для этого в качестве ключевой функциональности требуется онлайновое генерирование модельных предсказаний с использованием недавнего поведения.

### Чат-боты

Онлайновые сервисы часто имеют окна для ведения чата в реальном времени, чтобы оказывать поддержку клиентам, не покидая клавиатуры. Традиционно в этих окнах работали сотрудники службы поддержки клиентов, но недавний тренд на снижение трудозатрат и улучшение времени на решение проблем привел к их замене чат-ботами на базе машинного обучения, которые могут находиться онлайн 24/7. Такие чат-боты требуют тщательно продуманного сочетания нескольких техник машинного обучения и должны уметь реагировать на вводимые пользователем данные в реальном времени.

### Оценивание времени прибытия

Службы совместных поездок, навигации и доставки еды – все они зависят от возможности предоставлять точную оценку времени прибытия (например, водителя, вас самих или вашего ужина). Давать точные оценки очень трудно, поскольку для этого требуется учитывать объективно существующие факторы, такие как схема дорожного движения, погода и дорожные происшествия. Кроме того, в течение одной поездки оценки неоднократно обновляются.

Это всего лишь несколько примеров того, как применение машинного обучения в онлайновой обстановке может приносить большую пользу в прикладных областях, которые традиционно отличаются своей трудностью (представьте, что вы пишете логику оценивания времени прибытия вручную!). Список приложений можно продолжать: ряд зарождающихся областей, таких как самоуправляемые автомобили, робототехника и конвейеры видеоОбработки, также получают новое определение за счет машинного обучения.

Все эти приложения имеют одно общее важное требование: задержка. В случае онлайновых сервисов низкая задержка имеет первостепенное значение для обеспечения хорошего пользовательского опыта. В случае приложений, взаимодействующих с реальным миром (таких как робототехника или самоуправляемые автомобили), более высокая задержка может иметь еще более серьезные последствия для безопасности или точности.

В этой главе будет представлено щадящее введение во встроенную во фреймворк Ray библиотеку Ray Serve, которая позволяет разрабатывать приложения для онлайнового генерирования модельных предсказаний поверх фреймворка Ray. Сначала мы обсудим сложности онлайнового генерирования модельных предсказаний, которые библиотека Ray Serve решает. Затем рассмотрим архитектуру указанной библиотеки и представим ее ключевую функциональность. Наконец, мы применим библиотеку Ray Serve для разра-

ботки API сквозного онлайнового генерирования модельных предсказаний, состоящего из нескольких моделей обработки естественного языка. Изложение материала можно отслеживать вместе с исходным кодом в блокноте Jupyter этой главы<sup>1</sup>.

## КЛЮЧЕВЫЕ ХАРАКТЕРИСТИКИ ОНЛАЙНОВОГО ГЕНЕРИРОВАНИЯ МОДЕЛЬНЫХ ПРЕДСКАЗАНИЙ

В предыдущем разделе мы упомянули, что главной целью онлайнового генерирования модельных предсказаний является взаимодействие с моделями машинного обучения с низкой задержкой. Однако это уже давно является ключевым требованием к API бэкендов и веб-серверов, поэтому возникает естественный вопрос: чем отличается подача моделей машинного обучения как служб?

### Модели машинного обучения характерны своей вычислительной интенсивностью

Многие проблемы, возникающие при онлайновом генерировании модельных предсказаний, являются результатом одной ключевой характеристики: модели машинного обучения характерны очень высокой вычислительной интенсивностью. По сравнению с традиционным веб-сервисом, где запросы в основном обрабатываются с помощью интенсивных по вводу/выводу запросов к базе данных или других вызовов API, большинство моделей машинного обучения сводятся к выполнению множества линейно-алгебраических вычислений: выдаче рекомендации, оцениванию времени прибытия или обнаружению объекта на изображении. Это особенно верно для недавнего тренда «глубокого обучения», являющегося ответвлением машинного обучения, характеризующегося нейронными сетями, которые со временем становятся все больше. Нередко модели глубокого обучения также могут значительно выигрывать от использования специализированного оборудования, такого как графические и тензорные процессоры<sup>2</sup>, которые имеют специальные инструкции, оптимизированные под вычисления машинного обучения, и позволяют выполнять векторизованные вычисления параллельно по нескольким входным данным.

Многие приложения онлайнового генерирования модельных предсказаний должны работать в режиме 24/7. В сочетании с тем фактом, что модели машинного обучения являются вычислительно интенсивными, эксплуатация служб онлайнового генерирования модельных предсказаний бывает

<sup>1</sup> См. <https://oreil.ly/k9Vll>.

<sup>2</sup> Англ. Tensor Processing Unit (TPU). – Прим. перев.

очень дорогостоящей, требуя постоянного выделения множества центральных и графических процессоров. Первостепенные проблемы, связанные с онлайновым генерированием модельных предсказаний, сводятся к подаче моделей как служб таким способом, который минимизирует сквозную задержку при одновременном снижении затрат. Ключевые свойства систем онлайнового генерирования модельных предсказаний, предоставляемые для удовлетворения этих требований, таковы:

- поддержка специализированного оборудования, такого как графические и тензорные процессоры;
- возможность масштабировать используемые для модели ресурсы вверх и вниз в ответ на запросную нагрузку;
- поддержка пакетирования запросов, чтобы пользоваться преимуществами векторизованных вычислений.

## Модели машинного обучения бесполезны в изоляции

Когда машинное обучение обсуждается в академической или исследовательской среде, зачастую основное внимание уделяется отдельной, изолированной задаче, такой как распознавание объектов или классификация. Однако реально-практические приложения обычно не являются столь четкими и хорошо определенными. Вместо этого для решения задачи от начала до конца требуется комбинация нескольких моделей машинного обучения и бизнес-логики. Например, рассмотрим вариант использования рекомендации продуктов. Хотя для решения ключевой задачи составления рекомендации мы могли бы применить целый ряд известных техник машинного обучения, существует множество не менее важных проблем на периферии, многие из которых будут специфичны для каждого варианта использования:

- валидация входных и выходных данных с целью обеспечения смысла возвращаемого пользователю результата в семантическом плане. Нередко у нас бывают какие-то ручные правила, например не возвращать одну и ту же рекомендацию пользователю несколько раз подряд;
- получение актуальной информации о пользователе и имеющихся продуктах и ее конвертирование в признаки модели (в некоторых случаях это может выполняться онлайновым хранилищем признаков);
- комбинирование результатов нескольких моделей с использованием ручных правил, таких как фильтрация с оставлением лучших результатов или отбор модели с наибольшей метрикой уверенности.

Реализация API онлайнового генерирования модельных предсказаний требует способности интегрировать все эти компоненты в одну унифицированную службу. Поэтому важно обладать гибкостью, чтобы компоновать несколько моделей наряду с конкретно-прикладной бизнес-логикой. Эти части нельзя рассматривать изолированно: логика «склеивания» часто должна эволюционировать параллельно с самими моделями.

## ВВЕДЕНИЕ В БИБЛИОТЕКУ RAY SERVE

Библиотека Ray Serve – это масштабируемый вычислительный слой, предназначенный для поставки моделей машинного обучения как служб поверх фреймворка Ray. Указанная библиотека не зависит от фреймворка и, стало быть, не привязана к конкретной библиотеке машинного обучения; наоборот, она трактует модели как обычный исходный код Python. В дополнение к этому она позволяет гибко комбинировать обычную бизнес-логику Python с моделями машинного обучения. За счет этого обеспечивается возможность полной разработки служб онлайнового генерирования модельных предсказаний: приложение на основе Serve может выполнять валидацию вводимых пользователем данных, опрашивать базу данных, масштабируемо генерировать предсказания из нескольких моделей машинного обучения, а также комбинировать, фильтровать и валидировать выходные данные – и все это в процессе обработки одного запроса на генерирование модельного предсказания. И действительно, как вы увидите в разделе «Графы генерирования многомодельных предсказаний» на стр. 198, комбинирование результатов нескольких моделей машинного обучения является одной из ключевых сильных сторон библиотеки Ray Serve.

Несмотря на гибкость, библиотека Ray Serve обладает функциональными возможностями, специально разработанными для вычислительно интенсивных моделей машинного обучения, позволяющими динамически масштабировать и обеспечивать ресурсами, с тем чтобы запросная нагрузка эффективно обрабатывалась многочисленными центральными и/или графическими процессорами. Здесь библиотека Serve унаследовала множество выгод от того, что была построена поверх фреймворка Ray: она масштабируема на сотни машин, предлагает гибкие политики планирования и обеспечивает характерную низкими непроизводительными издержками связь между процессами с использованием ключевых API фреймворка Ray.

В этом разделе постепенно вводится ключевая функциональность библиотеки Ray Serve с акцентом на то, как она помогает решать описанные ранее сложности онлайнового генерирования модельных предсказаний. Для отслеживания работы примеров исходного кода этого раздела необходимо, чтобы следующие ниже пакеты Python были установлены локально:

```
pip install "ray[serve]==2.2.0" "transformers==4.21.2" "requests==2.28.1"
```

Выполнение примеров основано на том, что исходный код хранится локально в файле с именем `app.py` в текущем рабочем каталоге.

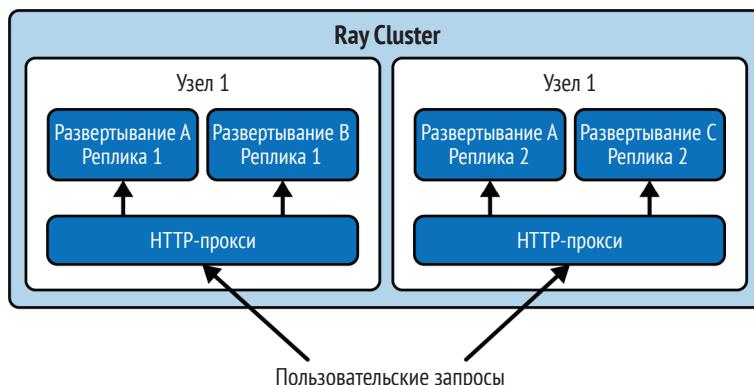
## Архитектурный обзор

Библиотека Ray Serve построена поверх фреймворка Ray, поэтому она наследует множество выгод, таких как масштабируемость, характерную низкими непроизводительными издержками связь, хорошо приспособленный для параллелизма API и возможность использовать совместную память посред-

ством хранилища объектов. Стержневым примитивом в библиотеке Ray Serve является *развертывание*, рассматриваемое как управляемая группа акторов Ray, к которым можно обращаться вместе и которые будут оперировать запросами между ними сбалансированным по нагрузке способом. В библиотеке Ray Serve каждый актор развертывания называется *репликой*. Часто развертывание соотносится с моделью машинного обучения один к одному, но развертывания могут содержать произвольный исходный код Python, поэтому в них также может содержаться бизнес-логика.

Библиотека Ray Serve позволяет предоставлять доступ к развертываниям по HTTP и определять синтаксический разбор входных данных и логику выходных данных. Однако одной из наиболее важных особенностей библиотеки Ray Serve является то, что развертывания также могут взаимодействовать друг с другом напрямую, используя нативный Python'овский API, который транслирует прямые акторные вызовы между репликами. За счет этого обеспечивается гибкая, высокопроизводительная компоновка моделей и бизнес-логики; вы увидите это в действии позже в данном разделе.

На рис. 8.1 представлена базовая схема того, как приложение библиотеки Ray Serve выполняется поверх кластера Ray: в кластере Ray размещается несколько развертываний. Каждое развертывание состоит из еще одной «реплики», каждая из которых является актором Ray. Входящий трафик маршрутизируется через HTTP-прокси, который обеспечивает балансировку связанной с запросами нагрузки между репликами<sup>1</sup>.



**Рис. 8.1** ♦ Архитектура приложения Ray Serve

По сути, составляющие приложение Ray Serve развертывания управляются централизованным актором-контроллером. Это отдельный актор, управляемый фреймворком Ray, который будет перезапускаться в случае аппаратного сбоя. Контроллер отвечает за создание и обновление акторов-реплик, широ-

<sup>1</sup> Развертывания также могут отправлять трафик напрямую друг другу. За счет этого появляется возможность разрабатывать более сложные приложения, включающие компоновку моделей или смешивание моделей машинного обучения с бизнес-логикой.

ковещательную трансляцию обновлений другим акторам в системе, а также за проверку работоспособности и восстановление после аппаратного сбоя. Если реплика или весь узел Ray по какой-либо причине выйдет из строя, то контроллер обнаружит сбои и обеспечит, чтобы акторы были восстановлены и могли продолжить раздавать трафик.

## Определение базовой конечной точки HTTP

В этом разделе мы познакомимся с библиотекой Ray Serve, определив простую конечную точку HTTP, выступающую как обертка для одной модели машинного обучения. Разворачиваемая нами модель представляет собой классификатор настроений: при вводе текста он будет предсказывать, было на выходе настроение позитивным либо негативным. Мы будем использовать предварительно натренированный классификатор настроений из библиотеки Transformers сообщества ИИ Hugging Face<sup>1</sup>, являющейся простым Python'овским API для предварительно натренированных моделей, который абстрагируется от деталей модели и позволяет сосредоточиваться на логике подачи модели как службы.

Для разворачивания этой модели с помощью библиотеки Ray Serve нужно определить класс Python и превратить его в *развертывание* Serve с помощью декоратора `@serve.deployment`. Указанный декоратор позволяет передавать ряд полезных опций для конфигурирования развертывания; мы обследуем некоторые из этих опций в разделе «Масштабирование и ресурсное обеспечение» на стр. 195:

```
from ray import serve

from transformers import pipeline

@serve.deployment
class SentimentAnalysis:
    def __init__(self):
        self._classifier = pipeline("sentiment-analysis")

    def __call__(self, request) -> str:
        input_text = request.query_params["input_text"]
        return self._classifier(input_text)[0]["label"]
```

Здесь следует отметить несколько важных моментов. Во-первых, мы создаем экземпляр модели в конструкторе класса. Эта модель может быть очень большой, поэтому ее загрузка в память может быть медленной (до нескольких минут). В библиотеке Ray Serve исходный код внутри конструктора будет выполняться при запуске всего один раз в каждой реплике, и любые свойства могут кешироваться для использования в будущем. Во-вторых, мы определяем логику обработки запроса в методе `__call__`. На входе она принимает HTTP-запрос Starlette и на выходе может возвращать любой сериализуемый

---

<sup>1</sup> См. <https://huggingface.co/>.

в формат JSON результат. В данном случае на выходе из нашей модели мы будем возвращать один строковый литерал: "POSITIVE" либо "NEGATIVE".

После того как развертывание было определено, мы используем API `.bind()`, чтобы создать (инстанцировать) его копию. Именно здесь конструктор можно передавать опциональные аргументы для конфигурирования развертывания (например, дистанционный путь к скачиванию модельных весов). Обратите внимание, что здесь развертывание не выполняется фактически – оно упаковывается со своими аргументами (это станет важнее позже, когда мы скомбинируем несколько моделей):

```
basic_deployment = SentimentAnalysis.bind()
```

Привязанное развертывание можно выполнять с помощью Python API `serve.run` либо соответствующей команды CLI `serve run`. Предполагая, что вы сохранили приведенный выше исходный код в файле с именем `app.py`, его можно выполнить локально посредством следующей ниже команды:

```
serve run app:basic_deployment
```

В результате будет создан экземпляр единственной реплики развертывания, который будет размещен за локальным HTTP-сервером. Для его тестирования можно задействовать пакет Python `requests`:

```
import requests

print(requests.get(
    "http://localhost:8000/", params={"input_text": "Hello friend!"}
).json())
```

Тестируя классификатор настроений на примере входного текста "Hello friend!", он правильно отклассифицирует текст как позитивный!

Приведенный выше пример фактически является классическим примером «Привет, мир!» для библиотеки Ray Serve: за базовой конечной точкой HTTP мы развернули одну модель. Однако обратите внимание, что нам пришлось делать синтаксический разбор входного HTTP-запроса вручную и вводить его в модель. В случае данного базового примера это была всего лишь одна строка исходного кода, но в реально-практических приложениях на входе нередко используется более сложная схема, а написание HTTP-логики вручную бывает утомительно и подвержено ошибкам. Для обеспечения возможности написания более выразительных API на основе HTTP библиотека Serve интегрируется с Python'овским фреймворком FastAPI<sup>1,2</sup>.

Развертывание в рамках библиотеки Serve может оберывать приложение FastAPI, используя его выразительные API для синтаксического разбора

---

<sup>1</sup> См. <https://oreil.ly/fLnjY>.

<sup>2</sup> Под капотом библиотека Ray Serve сериализует предоставленный пользователем объект приложения FastAPI. Затем при выполнении каждой реплики развертывания библиотека Ray Serve десериализует и выполняет приложение FastAPI так же, как оно выполнялось бы на обычном веб-сервере. Во время выполнения в каждой реплике Ray Serve будет работать независимый сервер FastAPI.

входных данных и конфигурирования поведения HTTP. В следующем ниже примере мы опираемся на FastAPI в части работы по синтаксическому разбору параметра `input_text` запроса, что позволяет удалять стереотипный исходный код синтаксического разбора:

```
from fastapi import FastAPI

app = FastAPI()

@serve.deployment
@serve.ingress(app)
class SentimentAnalysis:
    def __init__(self):
        self._classifier = pipeline("sentiment-analysis")

    @app.get("/")
    def classify(self, input_text: str) -> str:
        return self._classifier(input_text)[0]["label"]

fastapi_deployment = SentimentAnalysis.bind()
```

Видоизмененное развертывание должно вести себя точно так же, как в этом примере (попробуйте с помощью команды `serve run!`), но оно будет плавно улаживать недопустимые входные данные. В данном простом примере это видоизменение, возможно, покажется незначительным с точки зрения выгод, но в более сложных API это будет иметь огромное значение. Мы не будем здесь углубляться в детали FastAPI, но для получения дополнительной информации о его функциональных возможностях и синтаксисе рекомендуем ознакомиться с их превосходной документацией<sup>1</sup>.

## Масштабирование и ресурсное обеспечение

Как уже упоминалось, модели машинного обучения нередко требуют больших вычислительных ресурсов. Поэтому важно уметь выделять правильное количество ресурсов для своего приложения машинного обучения, чтобы оперировать запросной нагрузкой, минимизируя затраты. Библиотека Ray Serve позволяет корректировать выделяемые для развертывания ресурсы двумя способами: путем настройки числа `replicas` развертывания и настройки ресурсов, выделяемых каждой реплике. По умолчанию развертывание состоит из одной реплики, использующей один центральный процессор, но эти параметры можно корректировать в декораторе `@serve.deployment` (либо с помощью соответствующего API `deployment.options`).

Давайте видоизменим пример с классификатором настроений (`SentimentClassifier`), масштабировав его горизонтально до нескольких реплик и скорректировав ресурсное обеспечение таким образом, чтобы в каждой реплике использовалось два центральных процессора вместо одного (на практике для правильной установки этого параметра потребуется выполнить профилиро-

---

<sup>1</sup> См. <https://oreil.ly/vrxao>.

вание и понимать свою модель). Мы также добавим инструкцию `print`, чтобы журналировать идентификатор обрабатывающего каждый запрос процесса, показывая, что запросы теперь сбалансированы по нагрузке между двумя репликами:

```
app = FastAPI()

@serve.deployment(num_replicas=2, ray_actor_options={"num_cpus": 2})
@serve.ingress(app)
class SentimentAnalysis:
    def __init__(self):
        self._classifier = pipeline("sentiment-analysis")

    @app.get("/")
    def classify(self, input_text: str) -> str:
        import os
        print("from process:", os.getpid())
        return self._classifier(input_text)[0]["label"]

scaled_deployment = SentimentAnalysis.bind()
```

Выполнив эту новую версию классификатора с помощью команды `serve run app:scaled_deployment` и опросив ее с помощью библиотеки `requests`, как мы делали ранее, вы должны увидеть, что теперь запросы обрабатываются двумя копиями модели! Мы могли бы легко это отмасштабировать до десятков или сотен реплик, просто отрегулировав число реплик (`num_replicas`) в таком же ключе: фреймворк Ray позволяет масштабировать до сотен машин и тысяч процессов в одном кластере.

В данном примере мы масштабировали до статического числа реплик, при этом каждая реплика потребляла два полных центральных процессора, но библиотека Serve также поддерживает более выразительные политики ресурсообеспечения. Например:

- для придания развертыванию возможности использовать графические процессоры просто требуется установить `num_gpus` вместо `num_cpus`. Библиотека Serve поддерживает те же типы ресурсов, что и инструментарий Ray Core, поэтому при развертывании также можно использовать тензорные процессоры или другие конкретно-прикладные ресурсы;
- ресурсы могут быть дробными, что позволяет эффективно упаковывать реплики по корзинам. Например, если одна реплика не насыщает весь графический процессор, то ей можно выделить `num_gpus=0.5` и мультиплексировать с еще одной моделью;
- в случае приложений с варьирующейся запросной нагрузкой развертывание можно конфигурировать под динамическое автомасштабирование числа реплик в зависимости от числа выполняемых в данный момент.

Для получения более подробной информации о вариантах ресурсного обеспечения рекомендуем обратиться к последней версии документации библиотеки Ray Serve<sup>1</sup>.

---

<sup>1</sup> См. <https://oreil.ly/zuyND>.

## Пакетирование запросов

Многие модели машинного обучения можно эффективно векторизовывать, то есть несколько вычислений могут выполняться параллельно более эффективно, чем последовательно. Это особенно полезно при выполнении моделей на графических процессорах, специально разработанных под эффективное параллельное выполнение множества вычислений. В контексте онлайнового генерирования модельных предсказаний это открывает путь для оптимизации: параллельное обслуживание нескольких запросов (возможно, из разных источников) может значительно повышать пропускную способность системы (и, следовательно, снижать затраты).

Две стратегии высокого уровня пользуются преимуществами пакетирования запросов: пакетирование на стороне клиента и пакетирование на стороне сервера. При *пакетировании на стороне клиента* сервер принимает несколько элементов входных данных в одном запросе, а клиенты содержат логику отправки их пакетами, а не по одному за раз. Это бывает целесообразно в ситуациях, когда один клиент часто отправляет много запросов на генерирование модельных предсказаний. *Пакетирование на стороне сервера*, напротив, позволяет серверу выполнять пакетирование нескольких запросов, не требуя каких-либо изменений на клиенте. Оно также может использоваться для пакетирования запросов от нескольких клиентов, что обеспечивает эффективное пакетирование даже в ситуациях с большим числом клиентов, каждый из которых отправляет относительно малое число запросов.

Библиотека Ray Serve предлагает встроенный служебный компонент пакетирования на стороне сервера, декоратор `@serve.batch`, который требует всего нескольких изменений в исходном коде. В указанной поддержке пакетирования используются возможности Python `asyncio` по постановке нескольких запросов в очередь в рамках одного вызова функции. Функция должна принимать список элементов входных данных и возвращать соответствующий список элементов выходных данных.

Давайте еще раз обратимся к представленному ранее классификатору настроений и на этот раз видоизменим его под пакетирование на стороне сервера. Опорный конвейер (`pipeline`) Hugging Face поддерживает векторизованное генерирование модельных предсказаний: нам лишь нужно передать список элементов входных данных, и он вернет соответствующий список элементов выходных данных. Мы выделим вызов классификатора в новый метод `classify_batched`, который на входе будет принимать список элементов входных текстов, генерировать на них модельные предсказания и возвращать элементы выходных данных в виде отформатированного списка. В методе `classify_batched` будет использоваться декоратор `@serve.batch`, чтобы наделить его возможностью автоматически выполнять пакетирование. Поведение можно конфигурировать с помощью параметров `max_batch_size` и `batch_timeout_wait_s`. Здесь мы установим максимальный размер пакета равным 10 и будем ждать до 100 мс:

```
app = FastAPI()
@serve.deployment
```

```
@serve.ingress(app)

class SentimentAnalysis:
    def __init__(self):
        self._classifier = pipeline("sentiment-analysis")

    @serve.batch(max_batch_size=10, batch_wait_timeout_s=0.1)
    async def classify_batched(self, batched_inputs):
        print("Got batch size:", len(batched_inputs))
        results = self._classifier(batched_inputs)
        return [result["label"] for result in results]

    @app.get("/")
    async def classify(self, input_text: str) -> str:
        return await self.classify_batched(input_text)

batched_deployment = SentimentAnalysis.bind()
```

Обратите внимание, что в обоих методах `classify` и `classify_batched` теперь используется Python'овский синтаксис `async` и `await` и, стало быть, многие вызовы могут выполняться конкурентно в одном и том же процессе.

Для тестирования этого поведения мы будем использовать Python'овский API `serve.run`, чтобы отправлять запросы с использованием нативного для Python дескриптора развертывания:

```
import ray
from ray import serve
from app import batched_deployment

handle = serve.run(batched_deployment) ❶
ray.get([handle.classify.remote("sample text") for _ in range(10)])
```

- ❶ Получить дескриптор развертывания, чтобы иметь возможность отправлять запросы параллельно.

Возвращаемый вызовом `serve.run` дескриптор можно использовать для параллельной отправки нескольких запросов: здесь мы отправляем 10 запросов параллельно и ждем до тех пор, пока все они не вернутся. Без пакетирования каждый запрос обрабатывался бы последовательно, но поскольку мы активировали пакетирование, мы должны увидеть, что все запросы обрабатываются сразу (о чем свидетельствует размер пакета, напечатанный в методе `classify_batched`). При выполнении на центральном процессоре это, возможно, будет лишь незначительно быстрее, чем при последовательном выполнении, но при выполнении того же дескриптора на графическом процессоре мы бы наблюдали значительное ускорение в пакетированной версии.

## Графы генерирования многомодельных предсказаний

До сих пор мы развертывали и опрашивали одно-единственное развертывание `Serve`, выступающее в качестве обертки вокруг одной-единственной

модели машинного обучения. Как описывалось ранее, модели машинного обучения нередко бесполезны в изоляции: многие приложения требуют компоновки нескольких моделей и переплетения бизнес-логики с машинным обучением. Реальная мощь библиотеки Ray Serve заключается в ее способности компоновать несколько моделей вместе с обычной логикой Python в одном-единственном приложении. Это возможно путем создания множества экземпляров разных развертываний и передачи указателя между ними. Каждое такое развертывание может использовать все функциональные возможности, которые мы обсуждали до этого момента: они могут независимо масштабироваться, выполнять пакетирование запросов и гибко обеспечивать ресурсами.

Этот раздел иллюстрирует распространенные шаблоны многомодельной подачи, но на самом деле пока не содержит никаких моделей машинного обучения. В центре внимания находятся ключевые возможности, которые предоставляются библиотекой Serve.

## **Ключевая функциональность: привязка нескольких развертываний**

Все типы графов генерирования многомодельных предсказаний в библиотеке Ray Serve основаны на возможности передавать указатель на одно развертывание в конструктор другого. Для этого мы используем еще одну функциональность API `.bind()`: привязанное развертывание можно передавать еще одному вызову `.bind()`, и во время выполнения оно будет увязываться с «дескриптором» развертывания. За счет этого обеспечивается возможность независимого развертывания и создания экземпляров развертываний, а затем вызова друг друга во время выполнения. Вот самый базовый пример приложения Serve с несколькими развертываниями:

```
@serve.deployment
class DownstreamModel:
    def __call__(self, inp: str):
        return "Привет из нижестоящей модели!"

@serve.deployment
class Driver:
    def __init__(self, downstream):
        self._d = downstream

    async def __call__(self, *args) -> str:
        return await self._d.remote()

downstream = DownstreamModel.bind()
driver = Driver.bind(downstream)
```

В приведенном выше примере нижестоящая модель передается в «драйверное» развертывание. Затем во время выполнения драйверное развертывание вызывает нижестоящую модель. Драйвер может использовать любое число переданных моделей, а нижестоящая модель может даже принимать другие собственные нижестоящие модели.

## Шаблон 1: конвейеризация

Первым распространенным многомодельным шаблоном среди приложений машинного обучения является «конвейеризация»: последовательный вызов нескольких моделей, где входные данные одной модели зависят от выходных данных предыдущей. Обработка изображений, например, часто состоит из конвейера с несколькими этапами преобразований, таких как обрезка, сегментация и распознавание объектов либо оптическое распознавание символов (OCR). Каждая такая модель может иметь разные свойства, причем некоторые из них являются легковесными преобразованиями, которые могут выполняться на центральном процессоре, а другие – тяжеловесными моделями глубокого обучения, которые выполняются на графическом процессоре.

Такие конвейеры можно легко выражать с помощью API библиотеки Serve. Каждый этап конвейера определяется как независимое развертывание, и каждое развертывание передается в «конвейерный драйвер» верхнего уровня. В следующем ниже примере мы передаем два развертывания в драйвер верхнего уровня, и драйвер вызывает их последовательно. Обратите внимание, что многочисленные запросы к драйверу могут происходить конкурентно; следовательно, можно эффективно насыщать все этапы конвейера:

```
@serve.deployment
class DownstreamModel:
    def __init__(self, my_val: str):
        self._my_val = my_val

    def __call__(self, inp: str):
        return inp + " | " + self._my_val

@serve.deployment
class PipelineDriver:
    def __init__(self, model1, model2):
        self._m1 = model1
        self._m2 = model2

    @async def __call__(self, *args) -> str:
        intermediate = self._m1.remote("input")
        final = self._m2.remote(intermediate)
        return await final

m1 = DownstreamModel.bind("val1")
m2 = DownstreamModel.bind("val2")
pipeline_driver = PipelineDriver.bind(m1, m2)
```

Для тестирования этого примера можно еще раз применить API `serve run`. Отправка тестового запроса на вход конвейера возвращает значение `"input|val1|val2"` на его выходе: каждая нижестоящая «модель» добавила свое собственное значение, в итоге построив окончательный результат. На практике каждое такое развертывание может оберывать свою собственную модель машинного обучения, и один запрос может передаваться через множество физических узлов в кластере.

## Шаблон 2: широковещательная трансляция

В дополнение к последовательному выстраиванию моделей в цепочку нередко бывает полезно транслировать входные данные или промежуточный результат широковещательно нескольким моделям в параллельном режиме. Это делается для «ансамблования» или комбинирования результатов нескольких независимых моделей в единый результат либо используется в ситуации, когда разные модели могут работать лучше на разных входных данных. Нередко результаты моделей необходимо каким-либо образом комбинировать в окончательный результат: просто конкатенировать либо, возможно, выбирать один результат из множества.

В выразительном плане это очень похоже на пример конвейеризации: ряд нижестоящих моделей передается в драйвер верхнего уровня. В этом случае важно вызывать модели параллельно: ожидание результата каждой из них перед вызовом следующей значительно увеличило бы совокупную задержку системы:

```
@serve.deployment
class DownstreamModel:
    def __init__(self, my_val: str):
        self._my_val = my_val

    def __call__(self):
        return self._my_val

@serve.deployment
class BroadcastDriver:
    def __init__(self, model1, model2):
        self._m1 = model1
        self._m2 = model2

    async def __call__(self, *args) -> str:
        output1, output2 = self._m1.remote(), self._m2.remote()
        return [await output1, await output2]

m1 = DownstreamModel.bind("val1")
m2 = DownstreamModel.bind("val2")
broadcast_driver = BroadcastDriver.bind(m1, m2)
```

Тестирование этой конечной точки после ее повторного выполнения с помощью `serve run` возвращает `['val1", "val2"]'`, то есть комбинированный результат на выходе из двух вызываемых параллельно моделей.

## Шаблон 3: условная логика

Наконец, многие приложения машинного обучения примерно вписываютя в один из предыдущих шаблонов, и тем не менее нередко статический поток управления очень ограничивает. Возьмем как образец пример разработки службы извлечения номерных знаков из загруженных пользователем изображений. В этом случае нам, вероятно, потребуется сформировать конвейер обработки изображений, как обсуждалось ранее, но при этом мы не хотим загружать в конвейер любое изображение вслепую. Если пользователь загру-

жает что-то другое, кроме автомобиля, либо изображения низкого качества, то мы, скорее всего, захотим пойти напролом, избежать использования тяжеловесного и дорогостоящего конвейера и предоставить полезное сообщение об ошибке. Аналогично этому, в случае рекомендации продукта мы, возможно, захотим выбирать нижестоящую модель на основе введенных пользователем данных или результата промежуточной модели. Каждый такой пример требует встраивания в модели машинного обучения конкретно-прикладной логики.

Этого можно достичь, используя многомодельный API библиотеки Serve, потому что наш вычислительный граф определен как обычная логика Python, а не как статически определенный график. Например, в следующем ниже примере мы используем простой генератор случайных чисел, чтобы решить, к какой из двух нижестоящих моделей обращаться. В реальном примере генератор случайных чисел можно заменить бизнес-логикой, запросом к базе данных либо результатом из промежуточной модели:

```
@serve.deployment
class DownstreamModel:
    def __init__(self, my_val: str):
        self._my_val = my_val

    def __call__(self):
        return self._my_val

@serve.deployment
class ConditionalDriver:
    def __init__(self, model1, model2):
        self._m1 = model1
        self._m2 = model2

    @async def __call__(self, *args) -> str:
        import random
        if random.random() > 0.5:
            return await self._m1.remote()
        else:
            return await self._m2.remote()

m1 = DownstreamModel.bind("val1")
m2 = DownstreamModel.bind("val2")
conditional_driver = ConditionalDriver.bind(m1, m2)
```

Каждый вызов этой конечной точки возвращает либо "val1", либо "val2" с вероятностью 50/50.

## Сквозной пример: разработка API на базе обработки естественного языка

В этом разделе мы будем использовать библиотеку Ray Serve для разработки сквозного конвейера обработки естественного языка, размещаемого на сер-

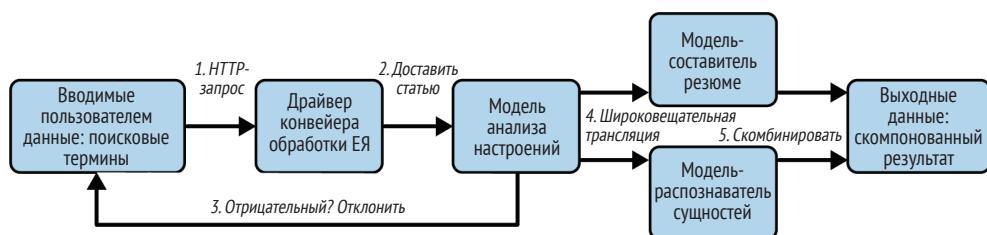
вере для онлайнового генерирования модельных предсказаний. Нашей целью будет предоставить конечную точку резюмирования статей Википедии, которая будет использовать несколько моделей обработки естественного языка и конкретно-прикладную логику для предоставления краткого описания наиболее релевантной страницы Википедии по заданному поисковому запросу.

Эта задача соединит вместе многие концепции и функциональности, которые мы обсуждали ранее:

- мы скомбинируем конкретно-прикладную бизнес-логику с несколькими моделями машинного обучения;
- граф генерирования модельных предсказаний будет состоять из всех трех многомодельных шаблонов: конвейеризации, широковещательной трансляции и условной логики;
- каждая модель будет размещена в виде отдельного развертывания Serve с целью их независимого масштабирования и обеспечения собственными ресурсами;
- одна из моделей будет использовать векторизованные вычисления посредством пакетирования;
- API будет определен с использованием FastAPI библиотеки Ray Serve для синтаксического разбора входных данных и определения выходной схемы.

Наш конвейер онлайнового генерирования модельных предсказаний будет структурирован так, как показано на рис. 8.2:

- 1) пользователь вводит ключевое слово для поиска;
- 2) мы доставляем содержимое наиболее релевантной статьи Википедии по поисковому запросу;
- 3) к статье будет применена модель анализа настроений. Все, что имеет «негативный» настрой, будет отклонено, и мы вернемся досрочно;
- 4) содержимое статьи будет широковещательно транслироваться в модели резюмирования и распознавания именованных сущностей;
- 5) мы вернем скомпонованный результат, основанный на результатах резюмирования и распознавания именованных сущностей.



**Рис. 8.2** ♦ Архитектура конвейера обработки естественного языка для резюмирования статей Википедии

Этот конвейер будет доступен по HTTP и будет возвращать результаты в структурированном формате. К концу данного раздела у нас будет конвей-

ер, работающий локально от начала до конца и готовый к масштабированию в кластере. Давайте начнем!

Прежде чем углубиться в исходный код, нужно будет установить локально следующие ниже пакеты Python:

```
pip install "ray[serve]==2.2.0" "transformers==4.21.2"  
pip install "requests==2.28.1" "wikipedia==1.4.0"
```

Вдобавок в этом разделе мы будем исходить из того, что все примеры исходного кода расположены локально в файле с именем *app.py* с тем, чтобы иметь возможность выполнять развертывания с помощью `serve run` из того же каталога<sup>1</sup>.

## Доставка содержимого и предобработка

Первый шаг – доставить наиболее релевантную страницу Википедии по заданному пользователем поисковому запросу. Для этого мы задействуем пакет *wikipedia* из PyPI, чтобы поручить ему выполнять тяжелую работу. Сначала выполним поиск по термину, а затем выберем верхний результат и вернем содержимое страницы. Если результаты не будут найдены, то мы вернем `None` – этот пограничный случай будет обработан позже, когда мы определим API:

```
from typing import Optional  
  
import wikipedia  
  
def fetch_wikipedia_page(search_term: str) -> Optional[str]:  
    results = wikipedia.search(search_term)  
    # Если результатов нет, то вернуться в вызывающий объект  
    if len(results) == 0:  
        return None  
  
    # Получить страницу для вехнего результата  
    return wikipedia.page(results[0]).content
```

## Модели обработки естественного языка

Далее нужно определить модели машинного обучения, которые будут выполнять тяжелую работу с нашим API. Мы будем использовать библиотеку *Transformers* сообщества ИИ *Hugging Face*<sup>2</sup>, поскольку она предоставляет

---

<sup>1</sup> Мы создали файл *app.py* за вас в репозитории книги на GitHub (<https://oreil.ly/E0Z-zg>). После клонирования репозитория и установки всех зависимостей у вас должна появиться возможность запускать `serve run app:<deployment_name>` непосредственно из каталога блокнота для каждого развертывания, на которое мы ссылаемся в вызове `serve run` в этой главе.

<sup>2</sup> См. <https://huggingface.co/>.

удобные API для предварительно натренированных современных моделей машинного обучения, позволяя нам сосредоточиться на логике подачи.

Первой используемой нами моделью будет тот же самый классификатор настроений, который мы применяли в предыдущих примерах. При развертывании этой модели будут задействованы преимущества векторизованных вычислений с использованием встроенного в библиотеку Serve API пакетирования:

```
from ray import serve
from transformers import pipeline
from typing import List

@serve.deployment
class SentimentAnalysis:
    def __init__(self):
        self._classifier = pipeline("sentiment-analysis")

    @serve.batch(max_batch_size=10, batch_timeout_s=0.1)
    async def is_positive_batched(self, inputs: List[str]) -> List[bool]:
        results = self._classifier(inputs, truncation=True)
        return [result["label"] == "POSITIVE" for result in results]

    async def __call__(self, input_text: str) -> bool:
        return await self.is_positive_batched(input_text)
```

Мы также будем использовать модель резюмирования текста, чтобы предоставлять краткое изложение выбранной статьи. Указанная модель принимает опциональный аргумент `max_length`, чтобы ограничивать длину резюме. Поскольку мы знаем, что это самая вычислительно дорогостоящая модель, мы устанавливаем `num_replicas=2`; если будет поступать много запросов одновременно, то за счет этого пропускная способность модели будет поддерживаться наравне с пропускной способностью других моделей. На практике, для того чтобы оставаться наравне с входной нагрузкой, может понадобиться больше реплик, но это можно узнать только из профилирования и мониторинга:

```
@serve.deployment(num_replicas=2)
class Summarizer:
    def __init__(self, max_length: Optional[int] = None):
        self._summarizer = pipeline("summarization")
        self._max_length = max_length

    def __call__(self, input_text: str) -> str:
        result = self._summarizer(
            input_text, max_length=self._max_length, truncation=True)
        return result[0]["summary_text"]
```

Заключительной моделью в нашем конвейере будет модель распознавания именованных сущностей: она будет пытаться извлекать именованные сущности из текста. Каждый результат будет иметь балл уверенности, поэтому можно установить порог, позволяющий принимать только те результаты, которые превышают определенную границу. Возможно, мы также захотим

ограничить общее число возвращаемых сущностей. Обработчик запросов для этого развертывания вызывает модель, а затем использует некую базовую бизнес-логику, которая обеспечивает соблюдение предоставленного порога уверенности и ограничение числа сущностей:

```
@serve.deployment
class EntityRecognition:
    def __init__(self, threshold: float = 0.90, max_entities: int = 10):
        self._entity_recognition = pipeline("ner")
        self._threshold = threshold
        self._max_entities = max_entities

    def __call__(self, input_text: str) -> List[str]:
        final_results = []
        for result in self._entity_recognition(input_text):
            if result["score"] > self._threshold:
                final_results.append(result["word"])
            if len(final_results) == self._max_entities:
                break
        return final_results
```

## Обработка HTTP и логика драйвера

Определив предобработку входных данных и модели машинного обучения, мы готовы определить API HTTP и логику драйвера. Сначала определяем схему ответа, который будем возвращать из API с помощью библиотеки валидации данных Pydantic<sup>1</sup>. В ответе указывается об успешности/неуспешности выполненного запроса и сообщение о статусе наряду с нашим резюме и именованными сущностями. Благодаря этому мы сможем возвращать полезный ответ в условных конструкциях ошибки, например когда результат не найден либо когда анализ настроений возвращает негативный результат:

```
from pydantic import BaseModel

class Response(BaseModel):
    success: bool
    message: str = ""
    summary: str = ""
    named_entities: List[str] = []
```

Далее нужно определить фактическую логику потока управления, которая будет выполняться в драйверном развертывании. Сам драйвер не будет выполнять никакой реальной тяжелой работы; вместо этого он будет обращаться к трем нижестоящим модельным развертываниям и интерпретировать их результаты. В нем также будет размещено определение приложения FastAPI, выполняющее синтаксический разбор входных данных и возвращающее правильную модель Response, основываясь на результатах конвейера:

---

<sup>1</sup> См. <https://oreil.ly/BM7rt>.

```

from fastapi import FastAPI

app = FastAPI()

@serve.deployment
@serve.ingress(app)
class NLPPipelineDriver:
    def __init__(self, sentiment_analysis, summarizer, entity_recognition):
        self._sentiment_analysis = sentiment_analysis
        self._summarizer = summarizer
        self._entity_recognition = entity_recognition

    @app.get("/", response_model=Response)
    async def summarize_article(self, search_term: str) -> Response:
        # Доставить содержимое верхней страницы для
        # поискового термина, если будет найдена
        page_content = fetch_wikipedia_page(search_term)
        if page_content is None:
            return Response(success=False,
                           message="Страницы не найдены.")

        # Условно продолжить, основываясь на анализе настроений
        is_positive = await self._sentiment_analysis.remote(page_content)
        if not is_positive:
            return Response(success=False,
                           message="Разрешены только положительные!")

        # Параллельно опросить модели резюмирования текста и
        # распознавания именованных сущностей
        summary_result = self._summarizer.remote(page_content)
        entities_result = self._entity_recognition.remote(page_content)
        return Response(
            success=True,
            summary=await summary_result,
            named_entities=await entities_result
        )

```

Мы доставляем содержимое страницы из тела главного обработчика, используя нашу логику `fetch_wikipedia_page` (если результат не найден, то возвращается ошибка). Затем обращаемся к модели анализа настроений. Если результат будет негативный, то мы завершаем работу досрочно и возвращаем ошибку, чтобы избежать вызова других долгостоящих моделей машинного обучения. Наконец, мы параллельно транслируем содержимое статьи как в модель резюмирования, так и в модель распознавания именованных сущностей. Результаты двух моделей сшиваются вместе в окончательный ответ, и мы возвращаем успех. Напомним, что вызовов этого обработчика может быть много, и все они будут выполняться конкурентно: вызовы к нежестоящим моделям не блокируют драйвер, и он сможет координировать вызовы к многочисленным репликам тяжеловесных моделей.

## Собираем все воедино

На данный момент мы определили всю ключевую логику. Все, что осталось, – это связать граф развертываний воедино и его выполнить:

```
sentiment_analysis = SentimentAnalysis.bind()
summarizer = Summarizer.bind()
entity_recognition = EntityRecognition.bind(threshold=0.95, max_entities=5)
nlp_pipeline_driver = NLPPipelineDriver.bind(
    sentiment_analysis, summarizer, entity_recognition)
```

Сначала нужно создать экземпляр каждого развертывания с любыми соответствующими входными аргументами. Например, здесь мы передаем порог и ограничение в модель распознавания сущностей. Самая важная часть, которую мы передаем, – это указатель на каждую из трех моделей в драйвер, чтобы тот мог координировать вычисления. Теперь, когда мы определили полный конвейер обработки естественного языка, можно его выполнить с помощью команды `serve run`<sup>1</sup>:

```
serve run app:nlp_pipeline_driver
```

Эта команда позволит локально развернуть все четыре развертывания и сделает драйвер доступным по адресу `http://localhost:8000`. Конвейер можно опрашивать, используя библиотеку `requests`, чтобы увидеть его в действии. Сначала давайте попробуем запросить запись по библиотеке Ray Serve:

```
import requests

print(requests.get(
    "http://localhost:8000/", params={"search_term": "rayserve"})
).text

'{"success":false,"message":"Страницы не найдены.",
 "summary":"","named_entities":[]}'
```

К сожалению, эта страница пока не существует! Вклинивается первая порция валидационной бизнес-логики и возвращает сообщение «Страницы не найдены». Давайте попробуем поискать что-нибудь более распространённое:

```
print(requests.get(
    "http://localhost:8000/", params={"search_term": "war"})
).text
```

---

<sup>1</sup> Используемые в этом примере модели – довольно крупные, поэтому имейте в виду, что при первом запуске данного примера их загрузка, скорее всего, займет несколько минут.

```
'{"success":false,"message":"Only positivitiy allowed!,"  
"summary":"","named_entities":[]}'
```

Возможно, нам просто было интересно узнать об истории, но эта статья была для нашего классификатора настроений слишком негативной. Давайте на этот раз попробуем что-нибудь более нейтральное – как насчет науки<sup>1</sup>?

```
print(requests.get(  
    "http://localhost:8000/", params={"search_term": "physicist"}  
) .text)
```

```
'{"success":true,"message":"","summary":" Physics is the natural science that studies  
matter, its fundamental constituents, its motion and behavior through space and time, and  
the related entities of energy and force . During the Scientific Revolution in the 17th  
century these natural sciences emerged as unique research endeavors in their own right .  
Physics intersects with many interdisciplinary areas of research, such as  
biophysics and quantum chemistry . ","named_entities":["Scientific",  
"Revolution", "Ancient", "Greek", "Egyptians"]}'
```

Приведенный выше пример успешно прошел полный конвейер: API ответил убедительным резюме статьи и списком соответствующих именованных сущностей.

Напомним, что в данном разделе мы создали API онлайновой обработки естественного языка с использованием библиотеки Ray Serve. Граф генерирования модельных предсказаний состоял из нескольких моделей машинного обучения в дополнение к конкретно-прикладной бизнес-логике и динамическому потоку управления. Каждая модель может масштабироваться независимо и иметь свое собственное ресурсное обеспечение, и мы можем использовать векторизованные вычисления, используя пакетирование на стороне сервера. Поскольку мы смогли протестировать API локально, следующим шагом будет развертывание в производстве. Библиотека Ray Serve упрощает развертывание в Kubernetes или других предложениях от облачных провайдеров с помощью инструмента запуска кластеров Ray, и мы легко смогли выполнить вертикальное масштабирование, чтобы справляться с многочисленными пользователями, отрегулировав ресурсное обеспечение наших развертываний.

---

<sup>1</sup> Перевод резюме по поисковому термину «физик»: физика – это естественная наука, предметом изучения которой является материя, ее фундаментальные составляющие, ее движение и поведение в пространстве и времени, а также связанные с ними сущности энергии и силы. Во время научной революции XVII века эта естественная наука возникла как уникальное исследовательское направление само по себе. Физика пересекается со многими междисциплинарными областями исследований, такими как биофизика и квантовая химия. Именованные сущности: научный, революция, древний, греки, египтяне. – Прим. перев.

## Резюме

В этой главе мы познакомились с Ray Serve, нативной для фреймворка Ray библиотекой, служащей для разработки API онлайнового генерирования модельных предсказаний. Библиотека Ray Serve сосредоточена на решении уникальных задач, связанных с подачей моделей машинного обучения как служб в производственных условиях, предлагая функциональность по эффективному масштабированию моделей и ресурсообеспечению, а также компоновке нескольких моделей наряду с бизнес-логикой. В дополнение к этому, как и все компоненты фреймворка Ray, библиотека Serve разработана как универсальное технологическое решение, позволяющее избегать привязки к поставщику.

Несмотря на то что мы рассмотрели пример сквозного многомодельного конвейера, в этой главе была разобрана лишь малая часть функциональных возможностей библиотеки Ray Serve и лучших образцов практики ее применения в реально-практических приложениях. Для получения более подробной информации и примеров рекомендуем ознакомиться с документацией библиотеки Ray Serve<sup>1</sup>.

---

<sup>1</sup> См. <https://oreil.ly/-fg0u>.

# Глава 9

---

## Кластеры Ray

*Ричард Ляо*

До сих пор мы сосредоточивались на том, чтобы обучать вас основам фреймворка Ray по разработке приложений машинного обучения. Вы научились параллелизировать свой исходный код Python с помощью инструментария Ray Core и проводить эксперименты по обучению с подкреплением с помощью библиотеки RLLib. Вы также увидели, как предобрабатывать данные с использованием библиотеки Ray Data, настраивать гиперпараметры с помощью библиотеки Ray Tune и тренировать модели посредством библиотеки Ray Train. Но одной из ключевых функциональных особенностей, предлагаемых фреймворком Ray, является возможность бесшовного масштабирования на несколько машин. За пределами лабораторной среды или крупной технологической компании настраивать несколько машин и соединять их в единый кластер Ray бывает довольно трудно. Данная глава всецело посвящена тому, как это делать<sup>1</sup>.

Облачные технологии превратили доступ к дешевым машинам в товар для любого человека. Но нередко для работы с инструментами облачного провайдера бывает довольно сложно подобрать правильные API. Коллектив разработчиков фреймворка Ray предоставил пару инструментов, которые позволяют абстрагироваться от сложностей. Существует три основных способа запуска, или развертывания, кластера Ray. Это можно сделать вручную, либо с помощью оператора Kubernetes, либо с помощью CLI-инструмента запуска кластеров.

---

<sup>1</sup> Фреймворк Ray сам по себе не индоктринирован в отношении того, как вы должны настраивать свой кластер. На самом деле у вас есть достаточно вариантов, многие из которых описаны в этой главе. Помимо технических решений с открытым исходным кодом, которые мы описываем здесь, также существуют полностью управляемые коммерческие решения, такие как те, которые предлагаются платформой Anyscale или Domino Data Lab.

В первой части этой главы мы рассмотрим эти три метода подробно<sup>1</sup>. Мы объясним создание кластера в ручном режиме и работу с CLI-инструментом запуска кластеров лишь вкратце и большую часть времени потратим на объяснение того, как использовать оператор Kubernetes. После этого расскажем о том, как запускать кластеры Ray в облаке и как автомасштабировать их вверх и вниз.

## Создание кластера Ray в ручном режиме

Давайте начнем с самого простого способа создания кластера Ray. Рассматривая построение кластера Ray в ручном режиме, мы исходим из того, что у вас есть список машин, которые могут взаимодействовать друг с другом, и на них установлен фреймворк Ray<sup>2</sup>.

Начнем с того, что в качестве головного узла вы можете выбрать любую машину. На этом узле выполните следующую ниже команду:

```
ray start --head --port=6379
```

Эта команда выведет IP-адрес запущенного сервера Службы глобального управления (GCS) фреймворка Ray, а именно IP-адрес локального узла плюс указанный вами номер порта<sup>3</sup>:

```
...
Next steps
To connect to this Ray runtime from another node, run
  ray start --address='<head-address>:6379'

If connection fails, check your firewall settings and network configuration.
```

Адрес головного узла (<head-address>) нужен для подключения других ваших узлов к кластеру, поэтому обязательно его скопируйте. Если вы опустите аргумент `--port`, то фреймворк Ray будет использовать случайный порт.

Затем к головному узлу можно подключить все другие узлы в вашем кластере, выполнив одну команду на каждом узле:

- 
- <sup>1</sup> Хотя технически к этой главе прилагается блокнот Jupyter (<https://oreil.ly/eGru2>), представленный здесь материал не очень хорошо подходит для разработки в интерактивном сеансе Python. Мы рекомендуем поработать с этими примерами в командной строке. Независимо от того, где вы решите работать, проверьте, чтобы фреймворк Ray был установлен с помощью команды `pip install "ray==2.2.0"`.
  - <sup>2</sup> В зависимости от ваших условий или рабочей ситуации для вас это может оказаться нереалистичным допущением. Не волнуйтесь, мы расскажем о способах создания кластеров Ray, которые не требуют, чтобы у вас были какие-либо машины. В любом случае полезно знать шаги, необходимые для создания кластера Ray в ручном режиме.
  - <sup>3</sup> Если у вас уже есть дистанционные экземпляры Redis, вы можете использовать их, указав средовую переменную `RAY_REDIS_ADDRESS=ip1:port1,ip2:port2....` Фреймворк Ray будет использовать первый адрес в качестве первичного, а остальные – в качестве сегментов.

```
ray start --address=<head-address>
```

Проверьте, чтобы был указан правильный адрес головного узла (<head-address>), который должен выглядеть примерно как 123.45.67.89:6379. Выполнив эту команду, вы должны увидеть следующий ниже результат:

```
-----
Ray runtime started.
-----
To terminate the Ray runtime, run
ray stop
```

Если вы хотите указать, что машина имеет 10 центральных процессоров и 1 графический процессор, то это можно сделать с помощью флагов `--num-cpus=10` и `--num-gpus=1`. Если вы увидите сообщение `Ray runtime started.`, то узел был успешно подключен к головному узлу по адресу `--address`. Теперь вы должны получить возможность подключиться к кластеру с помощью вызова `ray.init(address='auto')`.

 Если головной узел и новый узел, к которому вы хотите подключиться, находятся в отдельной подсети с трансляцией сетевого адреса<sup>1</sup>, то вы не сможете использовать адрес головного узла <head-address>, печатаемый командой, запускающей головной узел как `--address`. В этом случае нужно найти адрес, который достигнет головного узла с нового узла. Если головной узел имеет доменный адрес, подобный `compute04.berkeley.edu`, то можно использовать его вместо IP-адреса и опереться на DNS.

Если вы видите сообщение `Unable to connect to GCS at ...`, то оно означает, что головной узел недоступен по указанному адресу `--address`. Это может быть вызвано несколькими причинами. Например, возможно, головной узел на самом деле не запущен, по указанному адресу запущена другая версия фреймворка Ray, указанный адрес неверен либо настройки брандмауэра препятствуют доступу.

Если соединения нет, то для проверки возможности достичь каждого порта из узла можно использовать такие инструменты, как `nmap` или `ps`. Вот пример того, как выполнить проверку с помощью обоих инструментов в случае успеха<sup>2</sup>:

```
$ nmap -sV --reason -p $PORT $HEAD_ADDRESS
Nmap scan report for compute04.berkeley.edu (123.456.78.910)
Host is up, received echo-reply ttl 60 (0.00087s latency).
rDNS record for 123.456.78.910: compute04.berkeley.edu
PORT      STATE SERVICE REASON
VERSION
6379/tcp open  redis?  syn-ack
Service detection performed. Please report any incorrect
results at https://nmap.org/submit/ .
```

<sup>1</sup> Англ. Network Address Translation (NAT). – Прим. перев.

<sup>2</sup> Если вы хотите побольше узнать о конкретных флагах, используемых в последующих примерах, то рекомендуем ознакомиться с официальным справочным руководством (<https://nmap.org/book/man.html>).

```
$ nc -vv -z $HEAD_ADDRESS $PORT  
Connection to compute04.berkeley.edu 6379 port [tcp/...] succeeded!
```

Если ваш узел не может получить доступ к указанному порту и IP-адресу, то вы увидите:

```
$ nmap -sV --reason -p $PORT $HEAD_ADDRESS  
Nmap scan report for compute04.berkeley.edu (123.456.78.910)  
Host is up (0.0011s latency).  
rDNS record for 123.456.78.910: compute04.berkeley.edu  
PORT      STATE SERVICE REASON          VERSION  
6379/tcp  closed  redis   reset ttl 60  
Service detection performed. Please report any incorrect  
results at https://nmap.org/submit/ .  
$ nc -vv -z $HEAD_ADDRESS $PORT  
nc: connect to compute04.berkeley.edu port 6379 (tcp) failed: Connection refused
```

Теперь если вы хотите остановить процессы Ray на любом узле, то надо просто выполнить команду `ray stop`. Это был ручной способ создания кластера Ray. Давайте перейдем к обсуждению развертывания кластеров Ray с помощью популярного фреймворка оркестровки Kubernetes.

## Развертывание на Kubernetes

Kubernetes – это соответствующая промышленным стандартам платформа по управлению кластерными ресурсами. Она позволяет коллективам разработчиков программного обеспечения беспрепятственно развертывать, управлять и масштабировать свои бизнес-приложения в самых разнообразных производственных средах. Первоначально она была разработана для компании Google, но сегодня Kubernetes внедряется уже немалым числом организаций в качестве своего технологического решения по управлению кластерными ресурсами.

Поддерживаемый сообществом проект KubeRay<sup>1</sup> – это стандартный способ развертывания кластеров Ray и управления ими в Kubernetes. Оператор KubeRay помогает развертывать кластеры Ray поверх Kubernetes и ими управлять (рис. 9.1). Кластеры определяются как конкретно-прикладной ресурс `RayCluster` и управляются отказоустойчивым контроллером Ray. Указанный оператор автоматизирует резервирование, управление, автомасштабирование и операции кластеров Ray, развернутых в Kubernetes. Главные функциональные особенности данного оператора таковы:

- управление первоклассным `RayCluster` с помощью конкретно-прикладного ресурса;
- поддержка разнородных типов работников в одном кластере Ray;
- встроенный мониторинг посредством мониторинговой системы Prometheus;
- использование `podTemplate` для создания модулей Ray;

---

<sup>1</sup> См. <https://oreil.ly/LwUPr>.

- обновленный статус на основе запущенных модулей;
- автоматическое заполнение переменных среды в контейнерах;
- автоматическая префиксация вашей контейнерной команды командой запуска Ray;
- автоматическое добавление точки монтирования тома в */dev/shm* для совместной памяти;
- использование *ScaleStrategy* для удаления заданных узлов в заданных группах.



Рис. 9.1 ♦ Общий обзор оператора KubeRay

## Настройка своего первого кластера KubeRay

Оператор можно развернуть, склонировав репозиторий KubeRay<sup>1</sup> и вызвав следующую ниже команду:

```
export KUBERAY_VERSION=v0.3.0

kubectl create -k "github.com/ray-project/kuberay/manifests/\
cluster-scope-resources?ref=${KUBERAY_VERSION}&timeout=90s"

kubectl apply -k "github.com/ray-project/kuberay/manifests/\
base?ref=${KUBERAY_VERSION}&timeout=90s"
```

В том, что оператор был развернут, можно убедиться, применив вот эту команду:

```
kubectl -n ray-system get pods
```

После развертывания оператор будет отслеживать события Kubernetes (создание/удаление/обновление) в отношении обновлений ресурсов гаупл-кластер. После этих событий оператор сможет создать кластер, состоящий из головного модуля и нескольких модулей-работников, удалять кластер или обновлять кластер, добавляя либо удаляя модули-работники. Теперь давайте развернем новый кластер Ray, используя дефолтную конфигурацию кластера (мы вернемся к этому файлу YAML чуть позже):

<sup>1</sup> См. <https://oreil.ly/a8M1>.

```
wget "https://raw.githubusercontent.com/ray-project/kuberay/\\
${KUBERAY_VERSION}/ray-operator/config/samples/ray-cluster.complete.yaml"
kubectl create -f ray-cluster.complete.yaml
```

Оператор KubeRay конфигурирует службу Kubernetes, ориентированную на головной модуль Ray. В целях идентификации службы следует выполнить:

```
kubectl get service --selector=ray.io/cluster=raycluster-complete
```

Результат выполнения этой команды должен иметь следующую ниже структуру:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP
raycluster-complete-head-svc	ClusterIP	xx.xx.xxx.xx	<none>
PORT(S)	AGE		

Три указанных в результате порта соответствуют следующим ниже службам устройства головного модуля Ray:

6379

Служба глобального управления (GCS) головного узла Ray. Модули-работники Ray подключаются к этой службе при присоединении к кластеру.

8265

Предоставляет приборную панель Ray и службу подачи заявок Ray на выполнение работы<sup>1</sup>.

10001

Предоставляет сервер клиента Ray.

Вы должны обратить внимание на то, что используемые нами образы Docker имеют довольно крупный размер и их скачивание может занимать некоторое время. Кроме того, даже если вы видите ожидаемый результат команды `kubectl get service`, это еще не означает, что ваш кластер готов к использованию. Вы должны посмотреть на статус модуля и убедиться, что все они действительно находятся в рабочем (Running) состоянии.

## Взаимодействие с кластером KubeRay

Возможно, вы удивлены тем, что мы тратим так много времени на Kubernetes, поскольку вы, скорее всего, просто хотите научиться выполнять на нем скрипты Ray. Это понятно, и мы вернемся к данному вопросу через мгновение.

Сначала же давайте воспользуемся следующим ниже скриптом Python в качестве скрипта, который вы хотели бы выполнить в кластере. Ради прос-

---

<sup>1</sup> Англ. Ray Job Submission. Во фреймворке Ray заявка (job) на выполнение работы состоит из заданий (tasks). – Прим. перев.

тоты мы назовем его *script.py*. Он подключится к кластеру Ray и выполнит пару стандартных команд Ray:

```
import ray

ray.init(address="auto")
print(ray.cluster_resources())

@ray.remote
def test():
    return 12

ray.get([test.remote() for i in range(12)])
```

Приведенный выше скрипт можно выполнить тремя главными способами: с помощью команды `kubectl exec`, посредством подачи заявки Ray на выполнение работы либо посредством клиента Ray. Мы рассмотрим их в следующих далее разделах.

## ***Выполнение программ Ray с помощью команды kubectl***

Начнем с того, что взаимодействовать с головным модулем можно напрямую посредством команды `kubectl exec`. Эту команду следует применять, чтобы получать интерпретатор Python в головном модуле:

```
kubectl exec `kubectl get pods -o custom-columns=POD:metadata.name | \
grep raycluster-complete-head` -it -c ray-head -- python
```

С помощью данного терминала Python можно подключить и выполнить свое собственное приложение Ray:

```
import ray
ray.init(address="auto")
...
```

Взаимодействовать с этими службами можно и другими способами, без команды `kubectl`, но они потребуют налаживания сетевого взаимообмена. Далее мы воспользуемся самым простым способом – переадресацией портов.

## ***Использование сервера подачи заявок Ray на выполнение работы***

С помощью сервера подачи заявок Ray на выполнение работы можно выполнять скрипты в кластере. Указанный сервер можно использовать для отправки скрипта либо комплекта зависимостей и выполнения конкретно-прикладных скриптов с этим комплектом зависимостей. Для начала нужно будет переадресовать порт сервера подачи заявок на выполнение работы:

```
kubectl port-forward service/raycluster-complete-head-svc 8265:8265
```

Теперь нужно передать скрипт, установив значение переменной RAY\_ADDRESS равной конечной точке сервера подачи заявок и используя CLI подачи заявки Ray на выполнение работы:

```
export RAY_ADDRESS="http://localhost:8265"

ray job submit --working-dir=. -- python script.py
Вы увидите результат, который выглядит следующим образом:
Job submission server address: http://127.0.0.1:8265
2022-05-20 23:35:36,066 INFO dashboard_sdk.py:276
-- Uploading package gcs://_ray_pkg_533a957683abeba8.zip.
2022-05-20 23:35:36,067 INFO packaging.py:416
-- Creating a file package for local directory '.'.

-----
Job 'raysubmit_U5hfr1rqJZwJmLP' submitted successfully
-----
```

#### Next steps

Query the logs of the job:

```
ray job logs raysubmit_U5hfr1rqJZwJmLP
```

Query the status of the job:

```
ray job status raysubmit_U5hfr1rqJZwJmLP
```

Request the job to be stopped:

```
ray job stop raysubmit_U5hfr1rqJZwJmLP
```

Tailing logs until the job exits (disable with --no-wait):

```
{'memory': 47157884109.0, 'object_store_memory': 2147483648.0,
'CPU': 16.0, 'node:127.0.0.1': 1.0}
```

```
-----
Job 'raysubmit_U5hfr1rqJZwJmLP' succeeded
-----
```

Для выполнения работ в фоновом режиме используется опция --no-wait.

## Клиент Ray

Для того чтобы подключиться к кластеру посредством клиента Ray со своего локального компьютера, сначала следует проверить, чтобы локальная установка фреймворка Ray и младшая версия Python соответствовали версиям Ray и Python, работающим в кластере Ray. Для этого в обоих экземплярах выполняются команды ray --version и python --version. На практике вы будете использовать контейнер, в каком случае можно просто следить за тем, чтобы все выполнялось в одном контейнере. Кроме того, если версии не совпадают, то вы увидите предупреждающее сообщение, информирующее вас о проблеме.

Далее выполните следующую ниже команду:

```
kubectl port-forward service/raycluster-complete-head-svc 10001:10001
```

Эта команда будет блокировать. Локальный порт 10001 теперь будет направляться на сервер клиента Ray головного узла Ray.

В целях выполнения рабочей нагрузки Ray на своем дистанционном кластере Ray следует открыть локальную оболочку Python и запустить соединение с клиентом Ray:

```
import ray

ray.init(address="ray://localhost:10001")
print(ray.cluster_resources())

@ray.remote
def test():
    return 12

ray.get([test.remote() for i in range(12)])
```

С помощью этого метода можно выполнить программу Ray непосредственно на своем ноутбуке (вместо необходимости отправлять исходный код посредством команды `kubectl` или подачи заявки на выполнение работы).

## Предоставление оператора KubeRay

В предыдущих примерах мы использовали переадресацию портов как простой способ доступа к службам головного узла Ray. В случае производственных вариантов использования у вас, возможно, возникнет желание рассмотреть другие способы предоставления этих служб. Следующие ниже примечания являются общими для служб, работающих в Kubernetes.

По умолчанию служба Ray доступна из любой точки *внутри* кластера Kubernetes, где работает оператор Ray. Например, для того чтобы использовать клиента Ray из модуля в том же пространстве имен Kubernetes, что и кластер Ray, следует применять вызов `ray.init("ray://raycluster-complete-head-svc:10001")`.

Для того чтобы подключиться из еще одного пространства имен Kubernetes, следует использовать вызов `ray.init("ray://raycluster-complete-head-svc.default.svc.cluster.local:10001")`. (Если кластер Ray является недефолтным пространством имен, то следует использовать именно это пространство вместо дефолтного (`default`).)

Если вы пытаетесь получить доступ к службе за пределами кластера, то следует использовать контроллер входа. Любой стандартный контроллер входа должен работать с клиентом Ray и приборной панелью Ray. Рекомендуем выбрать решение, совместимое с вашими требованиями к сети и безопасности, – дальнейшие рекомендации выходят за рамки данной книги.

## Конфигурирование оператора KubeRay

Давайте подробнее рассмотрим конфигурацию кластера Ray, работающего на Kubernetes. Образец файла `kubera/ray-operator/config/samples/ray-cluster.`

*complete.yaml* будет неплохим ориентиром. Вот краткое описание наиболее характерных особенностей конфигурации кластера Ray:

```
apiVersion: ray.io/v1alpha1
kind: RayCluster
metadata:
  name: raycluster-complete
spec:
  headGroupSpec:
    rayStartParams:
      port: '6379'
      num-cpus: '1'
    ...
    template: # Шаблон модуля
      metadata: # Метаданные модуля
      spec: # Спецификация модуля
        containers:
          - name: ray-head
            image: rayproject/ray:1.12.1
            resources:
              limits:
                cpu: "1"
                memory: "1024Mi"
              requests:
                cpu: "1"
                memory: "1024Mi"
            ports:
              - containerPort: 6379
                name: gcs
              - containerPort: 8265
                name: dashboard
              - containerPort: 10001
                name: client
            env:
              - name: "RAY_LOG_TO_STDERR"
                value: "1"
            volumeMounts:
              - mountPath: /tmp/ray
                name: ray-logs
            volumes:
              - name: ray-logs
                emptyDir: {}
  workerGroupSpecs:
    - groupName: small-group
      replicas: 2
      rayStartParams:
        ...
      template: # Шаблон модуля
        ...
    - groupName: medium-group
      ...
    ...
```



Когда это возможно, лучше всего определять размер каждого модуля Ray так, чтобы он занимал весь узел Kubernetes, на котором он запланирован. Другими словами, лучше всего запускать один большой модуль Ray на каждом узле Kubernetes; выполнение нескольких модулей Ray на одном узле Kubernetes приводит к ненужным непроизводительным издержкам. Тем не менее выполнение нескольких модулей Ray на одном узле Kubernetes имеет смысл в некоторых ситуациях, например если:

- много пользователей используют кластеры Ray в кластере Kubernetes с лимитированными вычислительными ресурсами;
- вы либо ваша организация не управляете узлами Kubernetes напрямую (например, при развертывании на автопилоте GKE).

Вот несколько первостепенных конфигурационных значений, которые можно использовать:

#### `headGroupSpec` и `workerGroupSpecs`

Кластер Ray состоит из головного модуля и нескольких модулей-работников. Конфигурация головного модуля указывается в секции `headGroupSpec`. Конфигурация модулей-работников указывается в секции `workerGroupSpecs`. Может существовать несколько групп работников, каждая из которых имеет свой собственный шаблон (`template`) конфигурации. В поле `replicas` группы работников (`workerGroup`) задается число модулей-работников каждой группы, которые будут находиться в кластере.

#### `rayStartParams`

Это соотнесенность в формате «строка–строка» аргументов к точке входа `ray start` модуля Ray. Полный список аргументов приведен в документации по `ray start`<sup>1</sup>. Мы особо обращаем внимание на аргументы полей `num-cpus` и `num-gpus`:

#### `num-cpus`

Это поле сообщает планировщику Ray число центральных процессоров, доступных модулю Ray. Число центральных процессоров может определяться из ресурсных лимитов Kubernetes, указанных в шаблоне (`template`) модуля в секции групповой спецификации. Иногда это автообнаруживаемое значение бывает полезно переопределять. Например, установка `num-cpus:"0"` будет предотвращать планирование рабочих нагрузок Ray с ненулевыми требованиями к центральному процессору на головном узле.

#### `num-gpus`

Это поле задает число графических процессоров, доступных для модуля Ray. На момент написания книги данное поле не обнаруживается из шаблона (`template`) модуля в секции групповой спецификации. Следовательно, число графических процессоров должно задаваться для рабочих нагрузок графических процессоров в явной форме.

#### `template`

Именно сюда направляется основная часть конфигурации групп `headGroup` или `workerGroup`. Ключевое слово `template` представляет шаблон модуля Kubernetes, в котором определяется конфигурация модулей в группе.

---

<sup>1</sup> См. <https://oreil.ly/O5gE>.

### **resources**

Очень важно указывать запросы к процессору контейнера и памяти, а также лимиты для каждой групповой спецификации. Для рабочих нагрузок графических процессоров также можно указывать лимиты по графическим процессорам, например `nvidia.com/gpu`: 1 при использовании подключаемого устройства в виде графического процессора Nvidia.

### **nodeSelector и tolerations**

Планированием модулей Ray в рамках группы работников можно управлять, устанавливая значения полей `nodeSelector` и `tolerations` спецификации модуля. В частности, в этих полях определяется то, на каких узлах Kubernetes могут планироваться модули. Обратите внимание, что оператор KubeRay работает на уровне модулей – KubeRay не зависит от настройки опорных узлов Kubernetes. Работа по конфигурированию узлов Kubernetes остается за администраторами вашего кластера Kubernetes.

### *Образы контейнеров Ray*

Очень важно указывать образы, используемые контейнерами Ray вашего кластера. Головной узел и узлы-работники кластеров должны использовать одну и ту же версию фреймворка Ray. В большинстве случаев для головного узла и всех узлов-работников данного кластера Ray имеет смысл применять в точности одинаковый образ контейнера. В целях детализации конкретно-прикладных зависимостей вашего кластера следует создать образ на основе одного из официальных образов `gaurogo/jest/ray`.

### *Точки монтирования томов*

Точки монтирования томов можно использовать для поддержания журналов или других прикладных данных, хранящихся в ваших контейнерах Ray. (См. «Конфигурирование журналирования для KubeRay» ниже.)

### *Переменные контейнерной среды*

Переменные контейнерной среды могут использоваться для видоизменения поведения фреймворка Ray. Например, `RAY_LOG_TO_STDERR` будет перенаправлять журналы в `STDERR`, а не записывать их в файловую систему контейнера.

## **Конфигурирование журналирования для KubeRay**

Процессы кластера Ray обычно пишут журналы в каталог `/tmp/ray/session_latest/logs` в модуле. Эти журналы также видны на приборной панели Ray. В целях сохранения журналов Ray после жизненного цикла модуля можно использовать один из следующих ниже методов.

### *Агрегировать журналы из файловой системы контейнера*

В случае этой стратегии нужно с помощью команды `mountPath /tmp/ray/` смонтировать в контейнере Ray том с пустым каталогом (смотрите преды-

дущий образец конфигурации). Вы можете смонтировать журнальный том в прицепной контейнер, выполняющий инструмент агрегации журналов, такой как Promtail<sup>1</sup>.

#### *Перенаправление журналирования в STDERR контейнера*

Альтернативой является перенаправление журналирования в STDERR. Для этого надо установить средовую переменную RAY\_LOG\_TO\_STDERR=1 для всех контейнеров Ray. С точки зрения конфигурации Kubernetes, это означает добавление записи в поле env контейнера Ray в каждой групповой спецификации Ray groupSpec:

```
env:
  ...
  - name: "RAY_LOG_TO_STDERR"
    value: "1"
  ...
  ...
```

Затем можно использовать инструмент журналирования Kubernetes, предназначенный для агрегации из потоков STDERR и STDOUT.

## ИСПОЛЬЗОВАНИЕ ИНСТРУМЕНТА ЗАПУСКА КЛАСТЕРОВ Ray

Предназначение инструмента запуска кластеров Ray – упрощать развертывание кластера Ray в любом облаке. И вот что он делает:

- резервирует новый экземпляр/машину с помощью SDK облачного провайдера;
- исполняет команды оболочки, чтобы настраивать фреймворк Ray с предоставленными параметрами;
- при необходимости выполняет любые определенные пользователем конкретно-прикладные команды настройки. Это бывает полезно для настройки переменных среды и установки пакетов<sup>2</sup>;
- инициализирует кластер Ray за вас;
- разворачивает процесс автомасштабирования.

Мы познакомим вас с подробностями автомасштабирования в разделе «Автомасштабирование» на стр. 227. А пока давайте сосредоточимся на использовании инструмента запуска кластеров, чтобы развернуть кластер Ray. Для этого необходимо предоставить файл конфигурации кластера.

---

<sup>1</sup> См. <https://oreil.ly/LyEGy>.

<sup>2</sup> Для динамической настройки сред после развертывания кластера можно использовать среду выполнения.

## Конфигурирование своего кластера Ray

Для того чтобы запустить свой кластер Ray, необходимо указать требования к ресурсам в файле конфигурации кластера.

Вот наша спецификация «монтажного» кластера. Это примерно тот минимум, который нужно будет указывать для запуска кластера. Дополнительную информацию о кластерных файлах YAML можно получить из приведенного ниже большого примера (назовем его *cluster.yaml*):

```
# Уникальный идентификатор головного узла и
# работников этого кластера
cluster_name: minimal

# Максимальное число узлов-работников для запуска
# в дополнение к головному узлу. Значение
# min_workers по умолчанию равно 0
max_workers: 1

# Специфичная для облачного провайдера конфигурация
provider:
    type: aws
    region: us-west-2
    availability_zone: us-west-2a

# Как Ray будет аутентифицировать
# на недавно запущенных узлах
auth:
    ssh_user: ubuntu
```

## Использование CLI-инструмента запуска кластеров

Теперь, когда у вас есть файл конфигурации кластера, можно задействовать CLI-инструмент запуска кластеров, чтобы развернуть указанный кластер:

```
ray up cluster.yaml
```

Приведенная выше единственная строка исходного кода позаботится обо всем, что будет делаться посредством ручной настройки кластера. Она будет взаимодействовать с облачным провайдером, чтобы зарезервировать головной узел и запустить надлежащие службы Ray или процессы на этом узле.

Указанная единственная строка кода не приведет к автоматическому запуску всех заданных узлов. На самом деле она запустит только один «головной» узел и выполнит команду `ray start --head ...` на этом головном узле. Затем процесс автомасштабирования Ray будет использовать предоставленную конфигурацию кластера, чтобы запустить узлы-работники в качестве фонового потока после запуска головного узла.

## Взаимодействие с кластером Ray

После запуска кластера часто возникает потребность во взаимодействии с ним, используя различные действия:

- выполнить скрипт в кластере;
- переместить файлы, журналы и артефакты за пределы кластера;
- подключиться по SSH к узлам, чтобы проинспектировать детали машины.

Для взаимодействия с кластерами, запущенными инструментом запуска кластеров Ray, есть интерфейс командной строки (CLI). Если у вас уже имеется скрипт (например, упомянутый ранее *script.py*), то данный скрипт можно выполнить в кластере посредством команды `ray job submit` после того, как вы переадресовали порт конечной точки подачи заявки на выполнение работы:

```
# Выполнить в одном терминале:  
ray attach cluster.yaml -p 8265  
# Выполнить в отдельном терминале:  
export RAY_ADDRESS=http://localhost:8265  
ray job submit --working-dir=. -- python script.py
```

Опция `--working-dir` переместит ваши локальные файлы в кластер, а команда `python train.py` будет выполнена в оболочке в кластере.

Допустим, после выполнения этого скрипта вы генерируете артефакты, например файл *results.log*, который вы хотите проинспектировать. Для этого следует применить команду `ray rsync-down`, чтобы переместить файл обратно:

```
ray rsync-down cluster.yaml /path/on/cluster/results.log ./results.log
```

## РАБОТА С ОБЛАЧНЫМИ КЛАСТЕРАМИ

В этом разделе демонстрируются способы развертывания кластеров Ray в AWS и других облачных провайдерах.

### AWS

Сначала установите пакет средств разработки `boto` (`pip install boto3`) и сконфигурируйте свои учетные данные AWS в `$HOME/.aws/credentials`, как описано в документации `boto`<sup>1</sup>.

После того как `boto` будет сконфигурирован под управление ресурсами в вашей учетной записи AWS, вы должны быть готовы к запуску своего кластера. Приведенный файл конфигурации кластера `example-full.yaml`<sup>2</sup> создаст

<sup>1</sup> См. <https://oreil.ly/WnU8N>.

<sup>2</sup> См. <https://oreil.ly/rywrB>.

небольшой кластер с головным узлом m5.large (по требованию), сконфигурированным под автомасштабирование до двух работников спотовых экземпляров m5.large<sup>1</sup>.

Проверьте, что все работает как надо, выполнив следующие ниже команды из своей локальной машины:

```
# Создать либо обновить кластер. Когда команда завершится,
# она напечатает команду, которую можно использовать для
# подключения по SSH к головному узлу кластера
$ ray up ray/python/ray/autoscaler/aws/example-full.yaml

# Получить дистанционный экран на головном узле
$ ray attach ray/python/ray/autoscaler/aws/example-full.yaml
# Попробуйте выполнить программу Ray
# с помощью 'ray.init(address="auto")'

# Свернуть кластер
$ ray down ray/python/ray/autoscaler/aws/example-full.yaml
```

## Использование других облачных провайдеров

Кластеры Ray можно разворачивать в большинстве главнейших облаков, включая GCP и Azure. Вот шаблон начала работы с Google Cloud:

```
# Уникальный идентификатор для головного узла и
# работников этого кластера
cluster_name: minimal

# Максимальное число запускаемых узлов-работников
# в дополнение к головному узлу. Значение min_workers
# по умолчанию равно 0
max_workers: 1

# Специфичная для облачного провайдера конфигурация
provider:
  type: gcp
  region: us-west1
  availability_zone: us-west1-a
  project_id: null # Глобально уникальный ИД проекта

# Как Ray будет аутентифицировать
# на недавно запущенных узлах
auth:
  ssh_user: ubuntu
```

Вот шаблон начала работы с Azure:

```
# Уникальный идентификатор для головного узла и
# работников этого кластера
cluster_name: minimal
```

---

<sup>1</sup> См. <https://oreil.ly/fPFar>.

```

# Максимальное число запускаемых узлов-работников
# в дополнение к головному узлу. Значение min_workers
# по умолчанию равно 0
max_workers: 1

# Специфичная для облачного провайдера конфигурация
provider:
    type: azure
    location: westus2
    resource_group: ray-cluster

# Как Ray будет аутентифицировать
# на недавно запущенных узлах

auth:
    ssh_user: ubuntu
    # Необходимо указать пути к соответствующим файлам
    # пар приватных и публичных ключей.
    # Используйте `ssh-keygen -t rsa -b 4096`, чтобы
    # сгенерировать новую пару ключей ssh.
    ssh_private_key: ~/.ssh/id_rsa
    # Изменения здесь должны совпадать с тем,
    # что указано в file_mounts.
    ssh_public_key: ~/.ssh/id_rsa.pub

```

Более подробную информацию по этой теме можно получить в документации фреймворка Ray<sup>1</sup>.

## АВТОМАШТАБИРОВАНИЕ

Фреймворк Ray разработан с учетом поддержки высокояэластичных рабочих нагрузок, которые наиболее эффективны в кластере с автоматическим масштабированием. На высоком уровне автомасштабировщик пытается запускать и терминировать работу узлов, чтобы обеспечивать рабочим нагрузкам наличие ресурсов, достаточных для выполнения работы, при этом минимизируя количество простаивающих ресурсов. Он это делает, учитывая:

- заданные пользователем жесткие лимиты (минимальное/максимальное число работников);
- заданные пользователем типы узлов (узлы в кластере Ray не обязательно должны быть однородными);
- информацию со слоя планирования ядра Ray о текущем ресурсопотреблении / ресурсных потребностях кластера;
- подсказки по программному автомасштабированию.

Планировщик ресурсных потребностей автомасштабировщика будет просматривать ожидающие задачи, акторов и группы размещения, а также ре-

---

<sup>1</sup> См. <https://oreil.ly/2Eog5>.

сурсные потребности из кластера. Затем он попытается добавить минимальный список узлов, которые могут удовлетворить эти потребности.

Когда узлы-работники пристаивают более `idle_timeout_minutes`, они будут удаляться. Головной узел никогда не удаляется до тех пор, пока кластер не будет свернут.

Автомасштабировщик использует простой алгоритм упаковки в корзины<sup>1</sup>, чтобы упаковывать потребности пользователя в имеющиеся кластерные ресурсы. Оставшиеся невыполнеными потребности размещаются в наименьшем списке узлов, который удовлетворяет потребность при максимальной задействованности (начиная с самого малого узла). Подробнее об алгоритме автомасштабирования можно узнать в разделе «Автомасштабирование» технической документации по архитектуре фреймворка Ray<sup>2</sup>.

Фреймврк Ray также предоставляет документацию и инструменты для других менеджеров кластеров, таких как YARN, SLURM и LFS. Подробнее об этом можно прочитать в документации фреймворка Ray<sup>3</sup>.

## Резюме

В этой главе вы научились создавать собственные кластеры Ray, чтобы развертывать на них свои приложения Ray. Помимо ручной настройки и сворачивания кластеров, мы более подробно рассмотрели развертывание кластеров Ray в Kubernetes с помощью KubeRay. Мы также подробно разобрали инструмент запуска кластеров Ray и обсудили вопрос о том, как работать с облачными кластерами в таких облаках, как AWS, GCP и Azure. Наконец, мы обсудили тему применения автомасштабировщика Ray для масштабирования своих кластеров Ray.

Теперь, когда вы знаете о масштабировании кластеров Ray больше, в главе 10 мы вернемся к прикладной стороне дела и познакомимся с тем, как все встретившиеся нам библиотеки Ray для машинного обучения эффективно сходятся вместе, формируя инструментарий Ray AI Runtime для приложений машинного обучения.

---

<sup>1</sup> Англ. binpacking algorithm. – Прим. перев.

<sup>2</sup> См. <https://oreil.ly/u0kzS>.

<sup>3</sup> См. <https://oreil.ly/t5M9i>.

# Глава 10

---

## Начало работы с инструментарием Ray AI Runtime

С тех пор, как в главе 1 вы прочитали об инструментарии Ray AI Runtime (Ray AIR), мы прошли долгий путь. Помимо основ кластеров Ray и API ядра фреймворка Ray (Ray Core), вы получили хорошее представление обо всех библиотеках Ray более высокого уровня, которые можно использовать в рабочих нагрузках искусственного интеллекта, а именно библиотеках Ray RLlib, Tune, Train, Data и Serve в главах, предшествующих этой. Главная причина, по которой мы отложили более глубокое изложение инструментария Ray AIR до настоящего времени, заключается в том, что гораздо проще обдумывать его концепции и вычислять сложные примеры, если вы знаете его составные части.

В этой главе мы познакомим вас с ключевыми концепциями инструментария Ray AIR и с тем, как его использовать для разработки и развертывания распространенных рабочих процессов. Мы разработаем приложение AIR, в котором используются многие библиотеки Ray для науки о данных, о которых вы уже знаете. Мы также расскажем о том, когда и зачем использовать инструментарий AIR, и дадим краткий обзор его технических основ. Подробное обсуждение взаимосвязи инструментария AIR с другими системами, такими как интеграции и ключевые различия, будет рассмотрено в главе 11, когда мы будем говорить об экосистеме фреймворка Ray применительно к инструментарию AIR.

### ЗАЧЕМ ИСПОЛЬЗОВАТЬ ИНСТРУМЕНТАРИЙ AIR?

За последние пару лет выполнение рабочих нагрузок машинного обучения с помощью фреймворка Ray претерпевало постоянную эволюцию. Ray RLlib и Tune стали первыми библиотеками, которые были построены поверх ядра

фреймворка Ray (Ray Core). Вскоре после этого появились такие компоненты, как Ray Train, Serve и совсем недавно Ray Data с наборами данных Dataset. Добавление инструментария Ray AIR в качестве «зонтика» для всех других библиотек машинного обучения в рамках фреймворка Ray является результатом активных обсуждений с сообществом машинного обучения и обратной связи с ним. Фреймворк Ray как инструмент, основанный на Python, с хорошей поддержкой графических процессоров и примитивами с отслеживанием внутреннего состояния (акторы Ray) для сложных рабочих нагрузок машинного обучения, является естественным кандидатом для разработки похожего на среду выполнения инструментария AIR.

Ray AIR – это унифицированный инструментарий для ваших рабочих нагрузок машинного обучения, который предлагает целый ряд сторонних интеграций для тренировки моделей или доступа к конкретно-прикладным источникам данных. В духе других библиотек машинного обучения, построенных поверх ядра фреймворка Ray, инструментарий AIR скрывает абстракции более низкого уровня и предоставляет интуитивно понятный API, который был навеян распространенными шаблонами из таких инструментов, как scikit-learn.

По своей сути инструментарий Ray AIR был разработан как для исследователей данных, так и для инженеров машинного обучения. Как исследователь данных вы можете использовать его для разработки и масштабирования своих сквозных экспериментов или индивидуальных подзадач, таких как предобработка, тренировка, настройка, оценивание или подача моделей машинного обучения как служб. Как инженер машинного обучения вы можете зайти настолько далеко, что разработаете конкретно-прикладную платформу машинного обучения поверх инструментария AIR или просто воспользуетесь ее унифицированным API для его интеграции с другими библиотеками из вашей экосистемы. И фреймворк Ray всегда предоставляет вам гибкость в том, чтобы шагнуть уровнем ниже и углубиться в низкоуровневый API ядра Ray.

Являясь частью экосистемы фреймворка Ray, инструментарий AIR может использовать все его преимущества, включая бесшовный переход от экспериментов на ноутбуке к производственным процессам в кластере. Можна часто увидеть, как коллективы, в чьи обязанности входит исследование данных, передают свой исходный код машинного обучения «по эстафете» коллективам, ответственным за производственные системы. На практике это может обходиться дорого и отнимать много времени, поскольку указанный процесс нередко предусматривает видоизменение или даже переписывание частей исходного кода. Как мы увидим, инструментарий Ray AIR поможет вам в этом транзите, поскольку он берет на себя такие аспекты, как масштабируемость, надежность и безотказность.

На сегодняшний день инструментарий Ray AIR уже имеет приличное число интеграций, но он также полностью расширяем. И как мы покажем в следующем далее разделе, его унифицированный API обеспечивает плавный рабочий процесс, который позволяет легко заменять многие его компоненты. Например, с инструментарием AIR можно использовать тот же самый интер-

фейс для определения тренера XGBoost либо PyTorch, что делает удобным экспериментирование с различными моделями машинного обучения.

В то же время, выбрав инструментарий AIR, можно избегать проблемы работы с несколькими (распределенными) системами и написания для них склеивающего исходного кода, с которым трудно иметь дело. Коллективы разработчиков, работающие с большим количеством движущихся частей, нередко сталкиваются с быстрым устареванием интеграций и высокой нагрузкой на техническое сопровождение. Эти проблемы могут приводить к *усталости от миграций*, нежеланию перенимать новые идеи из-за ожидаемой сложности системных изменений.

-  Как и в каждой главе, с примерами исходного кода можно ознакомиться в прилагаемом к данной главе блокноте Jupyter<sup>1</sup>.

## КЛЮЧЕВЫЕ КОНЦЕПЦИИ ИНСТРУМЕНТАРИЯ AIR НА ПРИМЕРЕ

Философия внутреннего устройства инструментария AIR заключается в том, чтобы предоставлять вам возможность *справляться со своими рабочими нагрузками машинного обучения с помощью одного-единственного скрипта, выполняемого одной-единственной системой*. Давайте начнем с AIR и его важнейших концепций, рассмотрев расширенный пример использования. Вот что мы собираемся сделать:

- 1) загрузим набор данных о раке молочной железы в виде набора данных Dataset и применим инструментарий AIR для его предобработки;
- 2) определим модель XGBoost для тренировки классификатора на этих данных;
- 3) настроим так называемого настройщика для процедуры тренировки, чтобы настроить ее гиперпараметры;
- 4) будем сохранять контрольные точки натренированных моделей;
- 5) будем выполнять пакетное предсказание с помощью AIR;
- 6) развернем предсказателя как службу с помощью AIR.

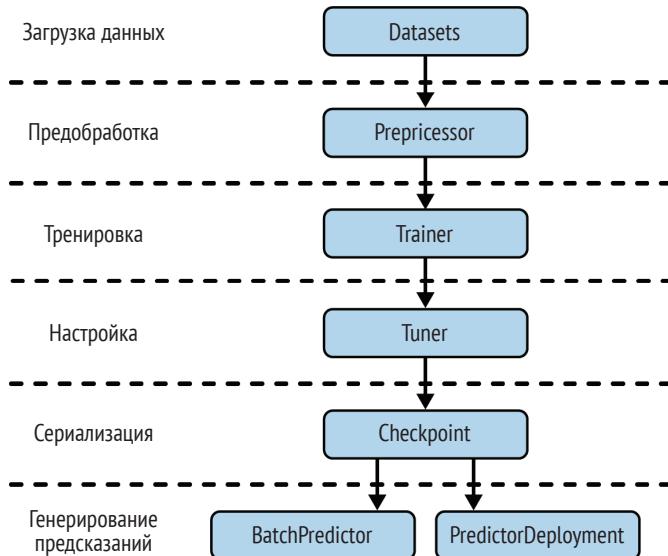
Указанные выше шаги будут выполнены путем разработки масштабируемых конвейеров с помощью API инструментария AIR. Для отслеживания работы этого примера следует проверить, чтобы у вас были установлены следующие ниже необходимые для него библиотеки:

```
pip install "ray[air]==2.2.0" "xgboost-ray>=0.1.10" "xgboost>=1.6.2"
pip install "numpy>=1.19.5" "pandas>=1.3.5" "pyarrow>=6.0.1" "aiorwlock==1.3.0"
```

На рис. 10.1 резюмированы шаги, которые мы собираемся предпринять в следующем ниже примере, наряду с компонентами инструментария AIR, которые мы будем использовать.

---

<sup>1</sup> См. <https://oreil.ly/ZrC5L>.



**Рис. 10.1** ❖ От загрузки данных до генерирования предсказаний с помощью инструментария AIR как единой распределенной системы

## Наборы данных Dataset и предобработчики

Стандартный способ загрузки данных в инструментарий Ray AIR состоит в использовании библиотеки Ray Data и ее наборов данных Dataset. Предобработчики AIR используются для преобразования входных данных в признаки для экспериментов по машинному обучению. Мы уже кратко касались предобработчиков в главе 7, но еще не обсуждали их в контексте инструментария AIR.

Поскольку предобработчики Ray AIR работают с наборами данных Dataset и используют экосистему фреймворка Ray, они позволяют эффективно масштабировать шаги предобработки. Во время тренировки предобработчик AIR подгоняется к указанным тренировочным данным и впоследствии может использоваться как для тренировки, так и для подачи моделей как службы<sup>1</sup>. Инструментарий AIR поставляется в комплекте со многими распространенными предобработчиками, которые охватывают множество вариантов использования. Если вы не найдете того, который вам нужен, то легко можете определить конкретно-прикладной предобработчик самостоятельно.

В нашем примере мы хотим сначала прочитать CSV-файл из корзины S3 в столбчатый набор данных, используя служебную функцию `read_csv`. Затем мы делим наш набор данных на тренировочный и тестовый наборы данных

<sup>1</sup> За счет этого обеспечивается паритет между конвейерами тренировки модели и подачи модели как службы, что привносит удобство в работу с инструментарием AIR, поскольку не приходится переопределять конвейеры для разных вариантов использования.

и определяем предобработчик AIR, `StandardScaler`, который нормализует все указанные столбцы набора данных так, чтобы они имели среднее значение, равное 0, и дисперсию, равную 1. Обратите внимание, что простая детализация предобработчика еще не преобразовывает данные. Вот как это реализуется:

```
import ray
from ray.data.preprocessors import StandardScaler

dataset = ray.data.read_csv(
    "s3://anonymous@air-example-data/breast_cancer.csv"
) ❶

train_dataset, valid_dataset = dataset.train_test_split(test_size=0.2)
test_dataset = valid_dataset.drop_columns(cols=[“target”]) ❷

preprocessor = StandardScaler(columns=[“mean radius”, “mean texture”]) ❸
```

- ❶ Загрузить CSV-файл о раке молочной железы из S3, используя библиотеку Ray Data.
- ❷ После определения тренировочного и тестового наборов данных мы добавляем целевой (`target`) столбец в тестовые данные.
- ❸ Определить предобработчик AIR, чтобы прошкалировать две переменные набора данных, сделав их нормально распределенными.

Обратите внимание, что для простоты в дальнейшей тренировке мы используем тестовый набор данных также в качестве валидационного набора данных, отсюда и принятые условности в именовании.

Прежде чем перейти к шагу тренировки в рабочем процессе AIR, давайте рассмотрим разные типы доступных вам предобработчиков AIR (табл. 10.1). Если вы хотите узнать обо всех имеющихся предобработчиках подробнее, то рекомендуем ознакомиться с руководством пользователя по этому вопросу<sup>1</sup>. В данной книге мы используем предобработчики только для шкалирования признаков, но другие типы предобработчиков Ray AIR также бывают очень полезны.

**Таблица 10.1. Предобработчики в инструментарии Ray AIR**

Тип предобработчика	Примеры
Шкалировщики признаков	MaxAbsScaler, MinMaxScaler, Normalizer, PowerTransformer, StandardScaler
Обобщенные препроцессоры	BatchMapper, Chain, Concatenator, SimpleImputer
Категориальные кодировщики	Categorizer, LabelEncoder, OneHotEncoder
Текстовые кодировщики	Tokenizer, FeatureHasher

## Тренеры

После того как будут готовы тренировочный и тестовый наборы данных и определены предобработчики, можно перейти к детализации тренера, ко-

<sup>1</sup> См. <https://oreil.ly/WcV6W>.

торый будет выполнять алгоритм машинного обучения на ваших данных. Тренеры из библиотеки Ray Train были представлены в главе 7; они обеспечивают состыковывающуюся обертку вокруг фреймворков тренировки, таких как TensorFlow, PyTorch или XGBoost. В этом примере мы сосредоточимся на последнем, но важно отметить, что с точки зрения API инструментария Ray AIR все остальные интеграции с фреймворками работают точно так же.

Давайте определим одну из многих специальных реализаций тренера (`Trainer`), так называемого `XGBoostTrainer`, которые поставляются с инструментарием Ray AIR. Определение такого тренера требует, чтобы были указаны следующие ниже аргументы:

- конфигурация масштабирования AIR (`ScalingConfig`), описывающая, как вы хотите масштабировать тренировку в своем кластере Ray;
- столбец метки (`label_column`), который указывает на столбец вашего набора данных, используемый в качестве метки в контролируемой тренировке с помощью XGBoost;
- аргумент `datasets`, содержащий по меньшей мере ключ `train` и опциональный ключ `valid`, чтобы указывать соответственно на тренировочный и валидационный наборы данных;
- предобработчик AIR (`preprocessor`) для вычисления признаков вашей модели машинного обучения;
- специфичные для фреймворка параметры (например, число раундов бустирования в XGBoost), а также общий набор параметров, именуемый `params`, визуализированный на рис. 10.2:

```
from ray.air.config import ScalingConfig
from ray.train.xgboost import XGBoostTrainer

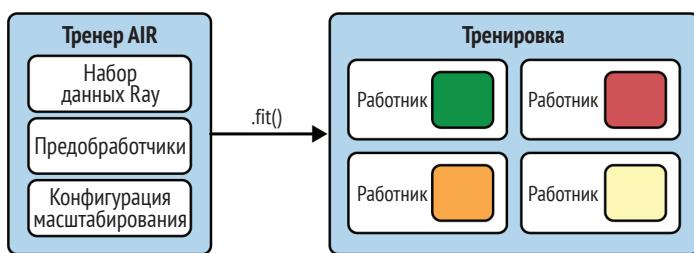
trainer = XGBoostTrainer(
    scaling_config=ScalingConfig(❶
        num_workers=2,
        use_gpu=False,
    ),
    label_column="target",
    num_boost_round=20, ❷
    params={
        "objective": "binary:logistic",
        "eval_metric": ["logloss", "error"],
    },
    datasets={"train": train_dataset, "valid": valid_dataset}, ❸
    preprocessor=preprocessor, ❹
)
result = trainer.fit() ❺
print(result.metrics)
```

- ❶ Каждый тренер (`Trainer`) сопровождается конфигурацией масштабирования. Здесь мы используем двух работников Ray и никаких графических процессоров.
- ❷ Объект `XGBoostTrainers` нуждается в конкретной конфигурации, а также в целевой функции (`objective`) тренировки и подлежащих отслеживанию метриках оценивания.

- ③ Тренер задает наборы данных, на которых он должен оперировать<sup>1</sup>.
- ④ Точно таким же образом можно предоставить предобработчиков AIR, которые тренер должен использовать.
- ⑤ После того как все было определено, достаточно просто вызвать метод `fit`, чтобы начать процедуру тренировки.

Тренеры обеспечивают масштабируемую тренировку моделей машинного обучения, основанную на наборах данных `Dataset` и предобработчиках. В добавок ко всему, как мы увидим далее, они также разработаны таким образом, чтобы хорошо интегрироваться с библиотекой Ray Tune с целью гиперпараметрической оптимизации.

Подводя итог этому разделу, на рис. 10.2 показано, как тренеры AIR выполняют подгонку моделей машинного обучения к наборам данных `Dataset` фреймворка Ray с учетом предобработчиков AIR и конфигурации масштабирования.



**Рис. 10.2** ♦ Тренеры AIR оперируют на наборах данных `Dataset` фреймворка Ray, а также используют предобработчики AIR и конфигурации масштабирования

## Настройщики и контрольные точки

Представленные во фреймворке Ray версии 2.0 в качестве составной части инструментария AIR *настройщики* предлагают масштабируемую гиперпараметрическую настройку с помощью библиотеки Ray Tune. Настройщики бесшовно взаимодействуют с тренерами AIR, но также поддерживают произвольные функции тренировки. В нашем примере вместо вызова метода `fit()` на экземпляре тренера (`trainer`) из предыдущего раздела можно передать тренера в настройщик. Для этого необходимо создать экземпляр настройщика с параметрическим пространством, в котором будет производиться поиск, так называемую конфигурацию настройки (`TuneConfig`). Эта конфигурация содержит все специфичные для библиотеки Tune конфигурации, такие как метрика, которую вы хотите оптимизировать, и опциональную конфигурацию выполнения (`RunConfig`), которая позволяет настраивать специфичные

<sup>1</sup> Технически говоря, не каждому тренеру обязательно нужно указывать аргумент `datasets`. Также есть возможность использовать специфичные для фреймворка загрузчики данных, хотя здесь мы не сможем показать никаких примеров.

для среды выполнения аспекты, такие как регистрируемый в журнале уровень детализации прогона настройщика.

Продолжая пример с тренером XGBoostTrainer, который мы определили ранее, вот как этот экземпляр тренера (trainer) обертывается в настройщик, чтобы откалибровать параметр max\_depth модели XGBoost:

```
from ray import tune

param_space = {"params": {"max_depth": tune.randint(1, 9)}}
metric = "train-logloss"

from ray.tune.tuner import Tuner, TuneConfig
from ray.air.config import RunConfig

tuner = Tuner(
    trainer, ❶
    param_space=param_space, ❷
    run_config=RunConfig(verbose=1),
    tune_config=TuneConfig(num_samples=2, metric=metric, mode="min"), ❸
)
result_grid = tuner.fit() ❹

best_result = result_grid.get_best_result()
print("Наилучший результат:", best_result)
```

- ❶ Инициализировать настройщика экземпляром тренера, который, в свою очередь, задает конфигурацию масштабирования прогона.
- ❷ Вашему настройщику также требуется параметрическое пространство (param\_space) для поиска по нему.
- ❸ Ему также требуется специальная конфигурация настройки (TuneConfig), чтобы указать настройщику, как оптимизировать тренера с учетом его параметрического пространства.
- ❹ Прогоны настройщика запускаются так же, как и тренеры, а именно вызовом метода .fit().

Всякий раз, когда вы выполняете тренеров или настройщиков AIR, они генерируют специфичные для фреймворка контрольные точки. Указанные контрольные точки можно использовать для загрузки моделей, чтобы их применять в нескольких библиотеках AIR, таких как Tune, Train или Serve. Контрольную точку можно получить, обратившись к результату вызова метода .fit() на тренере либо на настройщике. В нашем примере это означает, что можно просто обратиться к checkpoint объекта best\_result или любой другой записи из result\_grid, как показано ниже:

```
checkpoint = best_result.checkpoint
print(checkpoint)
```

Другой главный способ работы с контрольными точками состоит в их создании из специфичной для фреймворка существующей модели. Для этого можно использовать любой фреймворк машинного обучения, поддерживаемый инструментарием AIR, но поскольку с его помощью проще всего определять простую модель, мы вам покажем, как это выглядит для последовательной модели Keras в рамках TensorFlow:

```
from ray.train.tensorflow import TensorflowCheckpoint
import tensorflow as tf

model = tf.keras.Sequential([
    tf.keras.layers.InputLayer(input_shape=(1,)),
    tf.keras.layers.Dense(1)
])

keras_checkpoint = TensorflowCheckpoint.from_model(model)
```

Наличие контрольных точек хорошо тем, что в рамках инструментария AIR они являются нативным форматом обмена моделями. Их также можно использовать для подбора натренированных моделей на более позднем этапе, не беспокоясь о конкретно-прикладных способах хранения и загрузки соответствующих моделей. На рис. 10.3 схематично показано, как настройщики AIR работают с тренерами AIR.

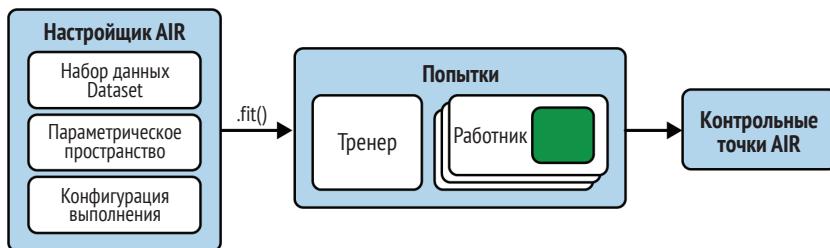


Рис. 10.3 ♦ Настройщик AIR калибрует гиперпараметры тренеров AIR

## Пакетные предсказатели

После того как вы натренировали модель с помощью инструментария AIR, то есть путем подгонки тренера (`Trainer`) или настройщика (`Tuner`), вы можете использовать результатирующую контрольную точку AIR для предсказания на пакетах данных, используя Python. Для этого из своей контрольной точки вы создаете пакетного предсказателя (`BatchPredictor`), а затем используете его метод `predict` на своем наборе данных `Dataset`. В нашем случае нужно применить специфичный для фреймворка класс-предсказатель, а именно `XGBoostPredictor`, чтобы сообщить инструментарию AIR, как правильно загружать контрольную точку (`checkpoint`):

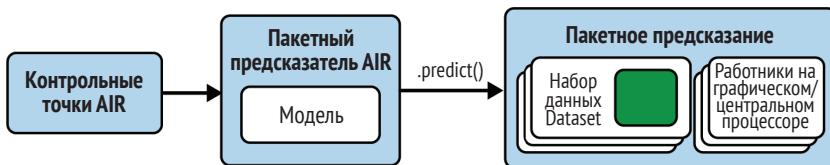
```
from ray.train.batch_predictor import BatchPredictor
from ray.train.xgboost import XGBoostPredictor

checkpoint = best_result.checkpoint
batch_predictor = BatchPredictor.from_checkpoint(checkpoint,
                                                 XGBoostPredictor) ❶

predicted_probabilities = batch_predictor.predict(test_dataset) ❷
predicted_probabilities.show()
```

- ① Загрузить модель XGBoost из контрольной точки в объект BatchPredictor.
- ② Выполнить пакетное предсказание на тестовом наборе данных, чтобы получить предсказанные вероятности.

Это можно наглядно представить на рис. 10.4.



**Рис. 10.4** ♦ Использование пакетных предсказателей AIR BatchPredictor из контрольных точек AIR Checkpoints для выполнения пакетного предсказания на наборе данных Dataset

## Развертывания

Вместо использования пакетного предсказателя (BatchPredictor) и непосредственного взаимодействия с рассматриваемой моделью можно задействовать библиотеку Ray Serve, чтобы развернуть службы генерирования модельных предсказаний, которую можно опрашивать по HTTP. Это делается с помощью класса развертывания предсказателя (PredictorDeployment) и его развертывания (deploy) с использованием контрольной точки (checkpoint). Единственной небольшой загвоздкой в этом является то, что наша модель оперирует на наборах данных DataFrame, которые невозможно отправлять по HTTP напрямую. Это означает, что необходимо указать службе предсказания в явной форме, как получать и преобразовывать определенную нами полезную нагрузку и создавать из нее набор данных DataFrame. Это делается путем указания для развертывания его адаптера<sup>1</sup>:

```

from ray import serve
from fastapi import Request
import pandas as pd
from ray.serve import PredictorDeployment

@async def adapter(request: Request): ❶
    payload = await request.json()
    return pd.DataFrame.from_dict(payload)

serve.start(detached=True)
deployment = PredictorDeployment.options(name="XGBoostService") ❷

deployment.deploy(
    XGBoostPredictor,
  
```

<sup>1</sup> Если вы выполните следующий ниже пример из блокнота Jupyter, то вам не придется беспокоиться о том, что он заблокирует блокнот, – все будет работать на ура. Подобного рода запуск сервера часто реализуется как блокирующий вызов, но класс PredictorDeployment не блокирует.

```

    checkpoint,
    http_adapter=adapter
)
print(deployment.url)

```

- ❶ Адаптер принимает объект HTTP-запроса и возвращает данные в формате, принимаемом моделью.
- ❷ После запуска Serve можно создать развертывание для модели.
- ❸ Для того чтобы развернуть (deploy) объект развертывания фактически, необходимо передать модельную контрольную точку (checkpoint), адаптерную (adapter) функцию и класс XGBoostPredictor, чтобы корректно загрузить модель.

В целях тестирования этого развертывания давайте создадим несколько образцов из тестовых данных, которые можно затем подать на вход нашей службы. Для простоты мы берем первый элемент тестового набора данных и преобразовываем его в словарь Python, чтобы вездесущая библиотека requests смогла отправить запрос на URL-адрес нашего развертывания вместе с ним:

```

import requests

first_item = test_dataset.take(1)
sample_input = dict(first_item[0])

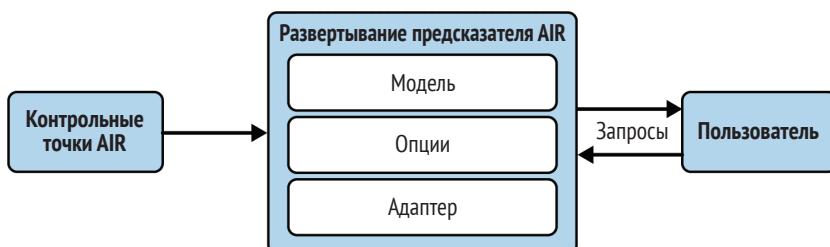
result = requests.post(❶
    deployment.url,
    json=[sample_input]
)
print(result.json())

```

```
serve.shutdown() ❷
```

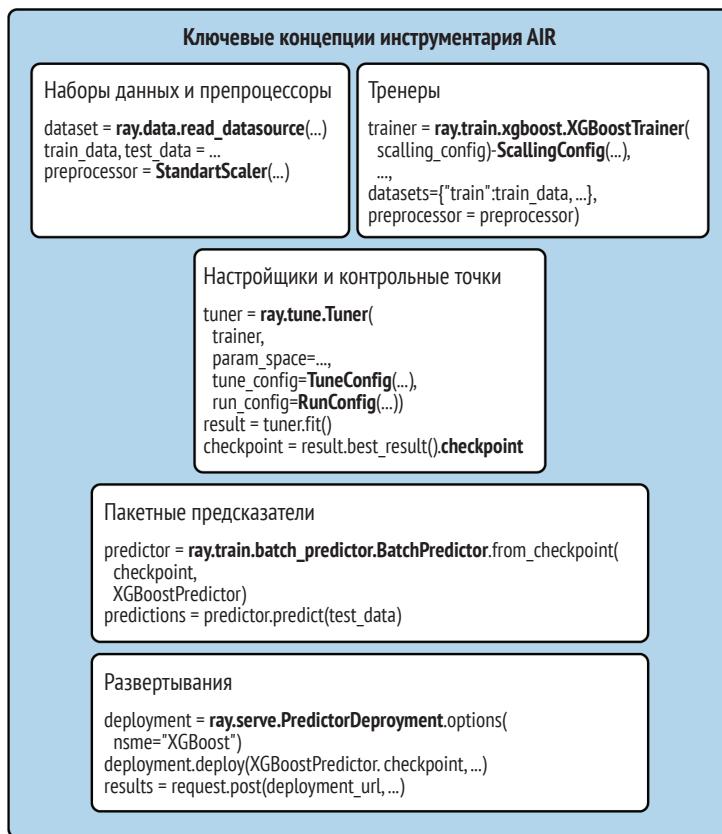
- ❶ Отправляет образец входных данных (sample\_input) по адресу `deployment.url` с помощью библиотеки requests.
- ❷ После завершения работы со службой можно безопасно отключить библиотеку Ray Serve.

На рис. 10.5 резюмируется процедура работы развертываний AIR с классом PredictorDeployment.



**Рис. 10.5** ♦ Создание развертываний предсказаний (PredictorDeployment) из контрольных точек AIR, чтобы расположенные в них модели, с которыми пользователи могут взаимодействовать, генерировали предсказания по запросу

На рис. 10.6 представлен общий вид всех компонентов и концепций, задействованных в инструментарии Ray AIR, включая псевдокод для всех главных компонентов инструментария AIR, которые мы рассмотрели в этой главе.



**Рис. 10.6** ♦ Инструментарий AIR комбинирует в себе множество библиотек фреймворка Ray, предоставляя унифицированный API для распространенных рабочих нагрузок в области науки о данных

Важно еще раз подчеркнуть, что в этом примере мы использовали *один-единственный скрипт Python* и одну-единственную распределенную систему в Ray AIR, чтобы выполнить всю тяжелую работу. На самом деле этот образец скрипта можно отмасштабировать горизонтально до крупного кластера, который использует центральные процессоры для предобработки и графические процессоры для тренировки, и отдельно сконфигурировать развертывание, просто изменив в данном скрипте параметры конфигурации масштабирования и аналогичные опции. Это не так просто или распространено, как может показаться, и нет ничего необычного в том, что исследователям данных приходится использовать несколько фреймворков (например, один для загрузки и обработки данных, один для тренировки и один для подачи модели как службы).



Инструментарий Ray AIR также можно использовать с библиотекой RLlib, но интеграция все еще находится на ранней стадии. Например, в целях интеграции библиотеки RLlib с тренерами AIR можно было бы использовать `RLTrainer`, который позволяет передавать все аргументы, которые вы бы передали стандартному алгоритму в библиотеке RLlib. После тренировки результирующую модель обучения с подкреплением можно сохранить в контрольной точке AIR, как и в случае с любым другим тренером AIR. В целях развертывания натренированной модели обучения с подкреплением можно использовать класс `PredictorDeployment` библиотеки Serve, передав свою контрольную точку вместе с классом `RLPredictor`.

Указанный API может подвергнуться изменению, но в документации AIR<sup>1</sup> можноувидеть пример того, как это работает.

## РАБОЧИЕ НАГРУЗКИ, ПОДХОДЯЩИЕ ДЛЯ ИНСТРУМЕНТАРИЯ AIR

Теперь, когда мы рассмотрели примеры работы с инструментарием AIR и его фундаментальные концепции, давайте взглянем чуть пошире и обсудим в принципе вопрос о том, какие виды рабочих нагрузок можно выполнять с его помощью. Мы уже рассматривали все эти рабочие нагрузки на протяжении всей книги, но полезно подвести их систематический итог. Как следует из названия, инструментарий AIR (AI Runtime) разработан для охвата распространенных заданий в проектах искусственного интеллекта. Эти задания можно примерно классифицировать следующим образом.

### *Вычисления без отслеживания внутреннего состояния*

Такие задания, как предобработка данных или вычисление модельных предсказаний на пакете данных, не имеют состояний<sup>2</sup>. Рабочие нагрузки без отслеживания внутреннего состояния могут вычисляться независимо в параллельном режиме. Если вы помните нашу трактовку заданий Ray из главы 2, именно для вычислений без отслеживания внутреннего состояния они и были разработаны. Инструментарий AIR использует задания Ray в основном для рабочих нагрузок без отслеживания внутреннего состояния<sup>3</sup>. В эту категорию попадают многие инструменты обработки больших данных.

### *Вычисление с отслеживанием внутреннего состояния*

Напротив, тренировка моделей и гиперпараметрическая настройка являются операциями с отслеживанием внутреннего состояния, поскольку

<sup>1</sup> См. <https://oreil.ly/gB-wg>.

<sup>2</sup> Разумеется, модель, используемая для генерирования предсказаний, сначала должна быть загружена, но поскольку во время предсказания ее параметры не меняются, в этом случае натренированные модели можно считать статическими данными. Такую ситуацию иногда принято называть *мягким состоянием*.

<sup>3</sup> В инструментарии AIR акторы Ray иногда используются для заданий без отслеживания внутреннего состояния. Это делается по соображениям производительности, таким как кеширование моделей при пакетном генерировании предсказаний.

они обновляют состояние модели во время соответствующей процедуры тренировки. Обновление работников с отслеживанием внутреннего состояния в такой распределенной тренировке является сложной темой, с которой фреймворк Ray справляется за вас. В инструментарии AIR для вычислений с отслеживанием внутреннего состояния используются акторы Ray.

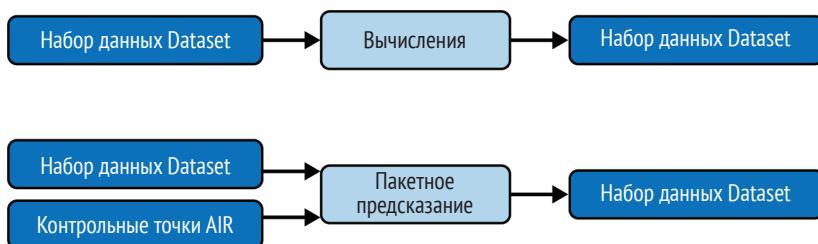
### *Составные рабочие нагрузки*

Комбинирование вычислений без отслеживания внутреннего состояния и вычислений с отслеживанием внутреннего состояния, например сперва путем обработки признаков, а затем путем тренировки модели, получило довольно широкое распространение в рабочих нагрузках искусственного интеллекта. На самом деле в сквозных проектах редко используется исключительно тот или иной подход. Выполнение таких сложных составных рабочих нагрузок в распределенном режиме можно охарактеризовать как *тренировка на больших данных*, и инструментарий AIR разработан таким образом, чтобы эффективно обрабатывать как части без отслеживания внутреннего состояния, так и части с отслеживанием внутреннего состояния.

### *Онлайновое генерирование модельных предсказаний*

Наконец, инструментарий AIR был разработан для масштабируемого онлайнового генерирования (много)модельных предсказаний. Переход от предыдущих трех рабочих нагрузок к подаче моделей как служб осуществляется без проблем по изначальному внутреннему устройству, поскольку вы по-прежнему работаете в рамках одной и той же экосистемы инструментария AIR.

На рис. 10.7 показаны типичные задания Ray AIR без отслеживания внутреннего состояния.



**Рис. 10.7 ♦** Задания AIR без отслеживания внутреннего состояния

Перечисленные выше четыре задания напрямую связаны с библиотеками фреймворка Ray. Например, в этой главе мы обсудили несколько способов использования библиотеки Ray Data и ее наборов данных Dataset для вычислений без отслеживания внутреннего состояния. Вы можете выполнять задание по пакетному генерированию предсказаний на заданном наборе данных Dataset, передавая его в `BatchPredictor`, загруженный из контрольной

точки AIR. Либо предобрабатывать набор данных Dataset, чтобы производить признаки для последующей тренировки<sup>1</sup>.

Аналогично с этим в инструментарии AIR есть три библиотеки, которые предназначены для вычислений с отслеживанием внутреннего состояния, а именно Train, Tune и RLLib. Как мы уже видели, библиотеки Train и RLLib легко интегрируются с библиотекой Tune в инструментарии AIR путем передачи соответствующих объектов Trainer в Tuner.

Когда дело доходит до продвинутых составных рабочих нагрузок, то инструментарий Ray AIR и его способность комбинированно использовать как задания, так и акторов реально проявляют себя во всей красе. Например, некоторые процедуры тренировки моделей машинного обучения требуют выполнения сложных заданий по обработке данных во время тренировки. В других случаях может требоваться перетасовка тренировочного набора данных перед каждой эпохой. Поскольку в тренировочных библиотеках инструментария Ray AIR (основанных на акторах) легко используются операции обработки данных (главным образом основанные на заданиях), в инструментарии AIR могут быть отражены даже самые сложные варианты использования.

Кроме того, поскольку любую контрольную точку AIR можно использовать с библиотекой Ray Serve, инструментарий AIR позволяет легко переключаться с рабочих нагрузок тренировки на рабочие нагрузки подачи моделей как служб, используя ту же инфраструктуру. Мы увидели, каким образом можно использовать PredictorDeployment для размещения модели за конечной точкой HTTP, которая оптимизирована под низкую задержку и высокую пропускную способность. Используя инструментарий AIR, можно масштабировать свои службы предсказания до нескольких реплик и использовать возможности фреймворка Ray по автомасштабированию, чтобы корректировать свой классер в соответствии с входящим трафиком<sup>2</sup>.

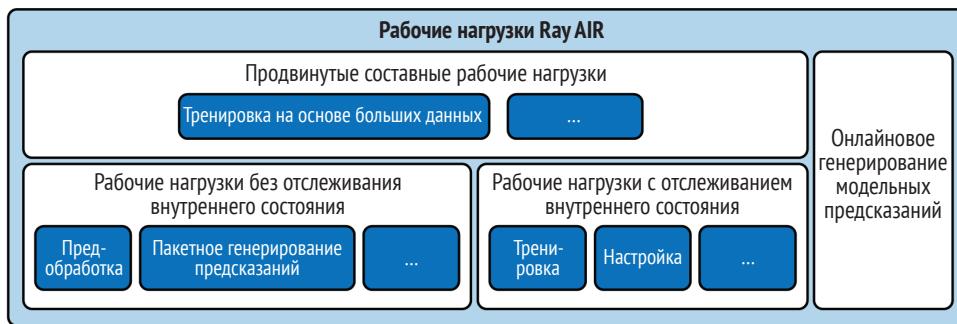
Эти типы рабочих нагрузок также можно использовать в разных сценариях. Например, инструментарий AIR можно использовать для замены и горизонтального масштабирования одиночного компонента существующего конвейера. Либо создавать свои собственные комплексные приложения машинного обучения с помощью AIR, как мы указывали в этой главе. Наконец, инструментарий AIR также можно использовать для разработки своей собственной платформы искусственного интеллекта, тему которой мы рассмотрим в главе 11.

На рис. 10.8 резюмируется то, как инструментарий AIR охватывает эти четыре типа рабочих нагрузок искусственного интеллекта в рамках экосистемы фреймворка Ray.

---

<sup>1</sup> Обратите внимание, что предобработчики могут отслеживать внутреннее состояние, но здесь мы не обсуждали никаких примеров этого сценария.

<sup>2</sup> Помимо нескольких реплик для одиночных моделей, инструментарий AIR также поддерживает развертывание нескольких моделей. За счет этого вы получаете возможность компоновать несколько моделей или выполнять A/B-тестирование.



**Рис. 10.8** ♦ Четыре типа рабочих нагрузок AIR, связанных с искусственным интеллектом, обеспечивают возможность вашей работы в кластерах Ray

Далее мы более подробно обсудим несколько аспектов каждого из этих четырех типов рабочих нагрузок AIR. В частности, мы обследуем вопрос о том, как фреймворк Ray исполняет такие рабочие нагрузки внутри. Кроме того, мы собираемся немного глубже погрузиться в технические аспекты инструментария Ray AIR, такие как его подходы к управлению памятью или к решению проблем аппаратных сбоев. Мы можем дать лишь краткий обзор этих тем, но по ходу работы будем предоставлять ссылки на более продвинутые материалы.

## Исполнение рабочих нагрузок AIR

Давайте подробнее рассмотрим принятую в инструментарии AIR модель исполнения.

### Исполнение без отслеживания внутреннего состояния

В библиотеке Ray Data задания Ray либо акторы Ray используются для того, чтобы выполнять преобразования. Предпочтение отдается заданиям, поскольку они обеспечивают более простое и гибкое планирование. В указанной библиотеке используется стратегия планирования, которая равномерно уравновешивает задания и их выходные данные по всему кластеру.

Акторы применяются, если преобразование имеет состояние или нуждается в долгостоящей настройке, например при загрузке крупномодельных контрольных точек. В этом случае однократная загрузка больших моделей в актор для их реиспользования в заданиях генерирования модельных предсказаний может повышать совокупную производительность. Когда библиотека Ray Data использует акторов, эти акторы создаются первыми, и необходимые данные (в нашем примере загруженная модель) передаются им до исполнения рассматриваемого преобразования.

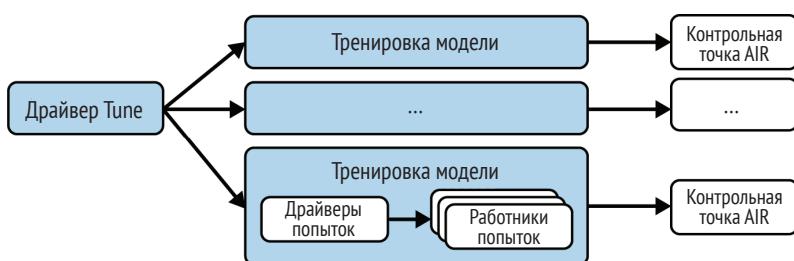
Как правило, наборы данных хранятся в памяти хранилища объектов Ray, тогда как крупные наборы данных переносятся на диск. Но в случае преобразований без отслеживания внутреннего состояния часто нет необходимости

держать промежуточные результаты в памяти. Как мы показали в главе 6, вместо этого, используя конвейеризацию на наборах данных Dataset, данные можно передавать в хранилище и из хранилища в потоковом режиме, тем самым повышая производительность<sup>1</sup>. Идея состоит в загрузке только части данных, необходимых в данный момент для преобразования. За счет этого появляется возможность значительно сокращать занимаемый преобразованием объем памяти и нередко ускорять исполнение в целом.

## **Исполнение с отслеживанием внутреннего состояния**

Библиотеки Ray Train и RLlib порождают акторов для своих работников распределенной тренировки. Как уже неоднократно демонстрировалось, обе библиотеки также легко интегрируются с библиотекой Tune. В главе 5 мы подробно описали, как библиотека Tune запускает попытки, которые, по сути дела, представляют собой группы акторов, исполняющих определенную рабочую нагрузку. Если вы используете библиотеку Train или RLlib с библиотекой Tune, это, в свою очередь, означает, что создается дерево акторов, а именно актор для каждой попытки Tune и субакторы для работников параллельной тренировки, запрошенных библиотекой Train либо RLlib.

С точки зрения исполнения рабочей нагрузки, такой подход естественным образом создает внутренний и внешний слои. Каждый субактор в попытке Tune обладает полной автономией в отношении своей рабочей нагрузки, как это требуется в указанном вами соответствующем тренировочном прогоне библиотек Train или RLlib. Это является внутренним слоем исполнения. Во внешнем слое библиотеке Tune нужно отслеживать статус индивидуальных попыток, что она и делает, периодически сообщая метрики драйверу попыток. Рисунок 10.9 иллюстрирует описанное выше вложенное создание и исполнение акторов Ray в указанном сценарии.



**Рис. 10.9 ♦** Выполнение распределенной тренировки на основе инструментария AIR вместе с библиотекой Tune

## **Исполнение составной рабочей нагрузки**

В составных рабочих нагрузках одновременно задействованы вычисления как на основе заданий, так и на основе акторов. Это может приводить к ин-

<sup>1</sup> Все важнейшие компоненты инструментария AIR, такие как Trainer или BatchPredictor, поддерживают эту функциональность конвейеризации.

тересным проблемам, связанным с ресурсообеспечением. Акторы попыток должны резервировать свои ресурсы заранее, но в заданиях без отслеживания внутреннего состояния это не делается. Вы можете столкнуться с проблемой, когда все доступные ресурсы могут быть зарезервированы для акторов тренировки, а для заданий загрузки данных ничего не остается.

Инструментарий AIR ставит этому заслон, позволяя библиотеке Type резервировать максимум 80 % центрального процессора узла. Указанный параметр можно отрегулировать, но по умолчанию такое значение является разумным, так как обеспечивает базовую доступность ресурсов для вычислений без отслеживания внутреннего состояния. В обычном сценарии, в котором в ваших операциях тренировки используются графические процессоры, а шаги обработки данных – нет, это вообще не является проблемой.

## ***Исполнение заданий по онлайновому генерированию модельных предсказаний***

В случае заданий по онлайновому генерированию модельных предсказаний библиотека Ray Serve управляет пулом акторов без отслеживания внутреннего состояния, чтобы обслуживать ваши запросы. Некоторые акторы прослушивают входящие запросы и вызывают других акторов, чтобы делать предсказания. Запросы автоматически распределяются по нагрузке с использованием алгоритма кругового обслуживания<sup>1</sup> по пулу акторов, размещающих у себя модели. Метрики загрузки отправляются в компонент Serve для выполнения процедуры автомасштабирования.

## **Управление памятью в инструментарии AIR**

В этом разделе мы немного углубимся в специфические для инструментария AIR методы управления памятью. Мы обсудим вопрос о том, в какой степени инструментарий AIR использует хранилище объектов Ray. Если этот раздел покажется вам слишком техническим, то его можно пропустить. Из него можно сделать вывод о том, что во фреймворке Ray задействованы умные технические приемы, которые обеспечивают надлежащее распределение и планирование ваших данных и вычислений.

Когда вы загружаете данные, используя наборы данных Dataset, вы уже знаете, что в вашем кластере указанные наборы данных внутренне разделены на блоки данных. Блок – это просто коллекция объектов Ray. Сделать правильный выбор размера блока довольно трудно, и данный выбор предусматривает принятие компромисса между непроизводительными издержками, связанными с необходимостью управлять слишком большим числом малых блоков, и риском возникновения исключений из-за нехватки памяти<sup>2</sup> вследствие слишком крупного размера блоков. Инструментарий AIR при-

---

<sup>1</sup> Англ. round-robin algorithm. – Прим. перев.

<sup>2</sup> Англ. out-of-memory (OOM). – Прим. перев.

держивается прагматичного подхода и старается распределять блоки таким образом, чтобы они не превышали 512 Мб. В случае если это обеспечить невозможно, будет выдано предупреждение. Если блок не помещается в память, то инструментарий AIR выплеснет ваши данные на локальный диск.

Ваши рабочие нагрузки с отслеживанием внутреннего состояния будут использовать хранилище объектов Ray в разной степени. Например, объекты Ray используются библиотекой RLLib для широковещательной трансляции весов модели индивидуальным работникам розыгрыша и для сбора данных об опыте. Они используются библиотекой Tune для настройки попыток путем отправки и извлечения контрольных точек AIR. Акторы рискуют столкнуться с проблемами нехватки памяти по техническим причинам, если требуется слишком много памяти по сравнению с выделенными ресурсами<sup>1</sup>. Если вы заранее знаете свои потребности в памяти, то сможете соответствующим образом адаптировать объем памяти (`memogy`) в своей конфигурации масштабирования (`ScalingConfig`) или просто запросить дополнительные ресурсы центрального процессора (`cput`).

В составных рабочих нагрузках акторы с отслеживанием внутреннего состояния (например, в случае тренировки) должны получать доступ к данным, созданным заданиями без отслеживания внутреннего состояния (например, в случае предобработки), что усложняет распределение памяти. Давайте рассмотрим два сценария:

- если у ответственных за тренировку акторов достаточно места (в хранилище объектов), чтобы поместить все тренировочные данные в память, то ситуация проста. Сначала выполняются шаги предобработки, а затем все блоки данных скачиваются на соответствующие узлы. Затем акторы тренировки просто прокручивают в цикле данные, которые хранятся в памяти;
- в противном случае обработка данных требует *конвейеризированного исполнения*, то есть данные будут обрабатываться заданиями «на лету» и впоследствии будут скачиваться по требованию акторами тренировки. Если соответствующий актор тренировки совмещен с узлом, который выполнял обработку, данные будут извлекаться из совместной памяти.

## Принятая в инструментарии AIR модель сбоя

Инструментарий AIR обеспечивает отказоустойчивость для большинства вычислений без отслеживания внутреннего состояния за счет *восстановления линии наследования*. Это означает, что фреймворк Ray будет восстанавливать блоки набора данных, если они будут потеряны из-за сбоев узлов, путем повторной передачи заявок на выполнение необходимых заданий, что позволяет масштабировать рабочие нагрузки до больших кластеров. Обратите

---

<sup>1</sup> В рабочих нагрузках с отслеживанием внутреннего состояния используется динамическая память Python (в виде кучи), которой фреймворк Ray не управляет.

внимание, что отказоустойчивость не относится к отказам головного узла. А сбой Службы глобального управления (GCS), хранящей метаданные кластера, будет приводить к уничтожению всех ваших заданий в кластере<sup>1</sup>.

Задания, включающие вычисления с отслеживанием внутреннего состояния, в первую очередь опираются на отказоустойчивость, основанную на контрольных точках. Библиотека Tune будет перезапускать распределенные попытки из их последней контрольной точки, как указано в конфигурации сбоя. При сконфигурированном интервале контрольных точек это означает, что библиотека Tune может эффективно выполнять попытки в кластерах, состоящих из «спотовых экземпляров». Кроме того, в случае сбоя всего кластера есть возможность возобновлять все эксперименты Tune из контрольной точки масштаба всего эксперимента.

Составные рабочие нагрузки наследуют стратегии отказоустойчивости как рабочих нагрузок без отслеживания внутреннего состояния, так и рабочих нагрузок с отслеживанием внутреннего состояния, поддерживая лучшее из обоих миров. Это означает, что восстановление линии наследования относится к части рабочей нагрузки без отслеживания внутреннего состояния, а фиксация состояния в контрольных точках уровня приложения по-прежнему относится ко всем вычислениям.

## Автомасштабирование рабочих нагрузок AIR

Библиотеки инструментария AIR могут работать на автомасштабируемых кластерах Ray, которые мы представили в главе 9. В случае рабочих нагрузок без отслеживания внутреннего состояния фреймворк Ray будет автомасштабировать при наличии поставленных в очередь заданий (или поставленных в очередь акторов вычислений на наборах данных Dataset). В случае рабочих нагрузок с отслеживанием внутреннего состояния фреймворк Ray будет автомасштабировать вертикально, если в кластере есть еще не запланированные ожидающие размещения группы (т. е. попытки Tune). Фреймворк Ray будет автомасштабировать вниз, когда узлы простаивают. Узел считается простаивающим, когда ресурсы на узле не задействованы, а также отсутствуют объекты Ray в памяти, или они не вынесены на диск в узле. Поскольку в большинстве библиотек инструментария AIR задействуются объекты, узлы будут удерживаться, если они содержат объекты, на которые указывают работники на других узлах (например, блок набора данных Dataset, используемый другой попыткой).

Вы должны знать, что автомасштабирование может приводить к неидеальной балансировке данных в кластере, поскольку узлы, которые были запущены раньше, естественным образом выполняют больше заданий в течение своего жизненного цикла. Для того чтобы оптимизировать эффективность

---

<sup>1</sup> Во избежание этого Служба глобального управления (Global Control Service, аббр. GCS) может развертываться в режиме высокой доступности, но обычно это выгодно только для рабочих нагрузок онлайнового генерирования модельных предсказаний.

рабочих нагрузок с интенсивным использованием данных, следует рассмотреть возможность ограничения (например, начиная с определенного минимального размера кластера) или отключения автомасштабирования.

## РЕЗЮМЕ

В этой главе вы увидели, как все представленные библиотеки фреймворка Ray сходятся вместе, формируя похожий на среду выполнения инструментарий Ray AIR (Ray AI Runtime). Вы узнали обо всех ключевых концепциях, которые позволяют создавать масштабируемые проекты машинного обучения, от экспериментов до производства. В частности, вы увидели, как библиотека Ray Data используется для вычислений без отслеживания внутреннего состояния, таких как предобработка признаков, и как библиотеки Ray Train, Tune и RLlib используются для вычислений с отслеживанием внутреннего состояния, таких как тренировка моделей. Бесшовное комбинирование этих типов вычислений в сложных рабочих нагрузках искусственного интеллекта и масштабирование их до больших кластеров являются ключевой мощью инструментария AIR. Разворачивание ваших проектов AIR выходит практически бесплатно, поскольку инструментарий AIR также полностью интегрируется с библиотекой Ray Serve.

В главе 11 мы покажем вам, как фреймворк Ray и, в частности, инструментарий AIR вписываются в более широкий спектр связанных инструментов. Знакомство с богатым набором интеграций и расширений экосистемы фреймворка Ray поможет вам понять, как использовать фреймворк Ray в своих собственных проектах.

# Глава 11

---

## Экосистема фреймворка Ray и за ее пределами

На протяжении этой книги вы увидели множество примеров экосистемы фреймворка Ray. Теперь самое время применить более системный подход и показать вам весь спектр имеющихся сегодня интеграций с фреймворком Ray. Мы сделаем это, разбирая указанную экосистему с точки зрения инструментария Ray AIR, чтобы иметь возможность ее обсудить в контексте презентативного рабочего процесса AIR.

Очевидно, что мы не сможем привести вам конкретные примеры исходного кода для большинства библиотек из экосистемы фреймворка Ray. Вместо этого нам придется довольствоваться демонстрацией еще одного примера работы с инструментарием Ray AIR, показывающего несколько интеграций и затрагивающего вопрос наличия других интеграций и способов их использования. Если потребуется, мы будем направлять вас к более продвинутым ресурсам, чтобы углубить ваше понимание.

Теперь, когда вы знаете о фреймворке Ray и его библиотеках гораздо больше, эта глава также станет подходящим местом, где можно сравнить предлагаемый фреймворком Ray функционал с аналогичными системами. Как вы уже видели, экосистема фреймворка Ray довольно сложна, ее можно рассматривать под разными углами и использовать для разных целей. Это означает, что многие аспекты фреймворка Ray можно сравнить с другими инструментами, которые представлены на рынке.

Мы также прокомментируем способы интегрирования фреймворка Ray в более сложные рабочие процессы в существующих платформах машинного обучения. В заключение дадим вам представление о том, как продолжить изучение фреймворка Ray после прочтения этой книги.



Блокнот Jupyter данной главы доступен в репозитории книги на GitHub<sup>1</sup>.

---

<sup>1</sup> См. <https://oreil.ly/mUrmi>.

## РАСТУЩАЯ ЭКОСИСТЕМА

Для того чтобы дать вам общее представление об экосистеме Ray на конкретном примере<sup>1</sup>, мы покажем, как использовать инструментарий Ray AIR с данными и моделями из экосистемы PyTorch, как регистрировать прогоны гиперпараметрической настройки в MLflow и как развертывать натренированные модели с помощью интеграции Ray Gradio. Попутно мы дадим краткий обзор и обсудим шаблоны использования других заслуживающих внимания интеграций на каждом соответствующем этапе.

Для отслеживания работы примеров исходного кода этой главы следует проверить, чтобы в вашей среде Python были установлены следующие ниже зависимости:

```
pip install "ray[air, serve]==2.2.0" "gradio==3.5.0" "requests==2.28.1"
pip install "mlflow==1.30.0" "torch==1.12.1" "torchvision==0.13.1"
```

Мы загрузим и преобразуем набор данных с помощью служебных функций из фреймворка PyTorch, а затем конвертируем эти данные в набор данных Dataset для работы с ними в инструментарии Ray AIR. Затем мы определим стандартную модель PyTorch и простой цикл тренировки, который можно использовать в библиотеке Ray Train. Далее обернем тренера TorchTrainer в настройщик и будем регистрировать результаты тестирования в MLflow с помощью регистратора MLflowLogger, который поставляется вместе с библиотекой Ray Tune. Наконец, мы выставим натренированную модель в качестве службы с помощью интеграции Gradio, работающей на основе библиотеки Ray Serve.

Другими словами, рассматриваемый нами пример берет распространенные библиотеки Python для науки о данных, которые вы, возможно, уже используете, и обертывает их в рабочий процесс инструментария AIR, задействуя интеграции в рамках экосистемы фреймворка Ray. В фокусе внимания будут эти интеграции и то, как они взаимодействуют с инструментарием AIR, и в меньшей степени конкретный вариант использования.

## Загрузка и предобработка данных

В главе 6 вы узнали об основах библиотеки Ray Data и ее наборов данных Dataset, о том, как их создавать из распространенных структур данных Python, как загружать файлы Parquet из систем хранения данных, таких как S3, и как использовать интеграцию фреймворка Ray с библиотекой Dask on Ray для взаимодействия с Dask.

<sup>1</sup> Текущий список интеграций в экосистему AIR можно найти в документации фреймворка Ray ([https://oreil.ly/S7\\_eb](https://oreil.ly/S7_eb)). В более общем плане список интеграций находится на странице экосистемы фреймворка Ray (<https://oreil.ly/Wiqi1>). В последнем случае вы увидите гораздо больше интеграций, чем те, которые мы обсуждаем здесь, такие как ClassyVision, Intel Analytics Zoo, NLU от John Snow Labs, Ludwig AI, PyCaret и SpaCy.

В качестве еще одного примера возможностей наборов данных Dataset давайте обратимся к работе с изображениями, загруженными в PyTorch посредством расширения `torchvision`. Идея проста. Мы загрузим имеющийся в PyTorch всем известный набор данных CIFAR-10 посредством пакета `torchvision.datasets`, а затем сделаем из него набор данных Dataset. В частности, определим функцию с именем `load_cifar`, которая возвращает данные CIFAR-10 для тренировки или тестирования:

```
from torchvision import transforms, datasets

def load_cifar(train: bool):
    transform = transforms.Compose([
        ❶ transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])

    return datasets.CIFAR10(
        ❷ root='./data',
        download=True,
        train=train, ❸
        transform=transform
    )
```

- ❶ Использует трансформанту PyTorch, чтобы вернуть нормализованные тензорные данные.
- ❷ Загружает набор данных CIFAR-10 с помощью модуля `datasets` пакета `torchvision`.
- ❸ Сигнализирует о том, что функция-загрузчик (`loader`) возвращает тренировочные либо тестовые данные.

Обратите внимание, что пока что мы не касались ни одной библиотеки Ray, и в точно таком же ключе можно было бы использовать любой другой набор данных или преобразование из PyTorch. В целях получения наборов данных Dataset для использования с инструментарием AIR мы подаем функции-загрузчики `load_cifar` в служебную функцию `from_torch` из инструментария Ray AIR:

```
from ray.data import from_torch

train_dataset = from_torch(load_cifar(train=True))
test_dataset = from_torch(load_cifar(train=False))
```

Набор данных CIFAR-10 используется для задач классификации изображений; он состоит из квадратных изображений размером 32 пикселя и поставляется с метками из 10 категорий. Пока что мы загрузили этот набор данных только в той форме, что предоставлена библиотекой PyTorch, но нам все еще нужно его конвертировать, чтобы использовать с тренером AIR. Это делается путем создания столбцов изображений (`image`) и меток (`label`), на которые затем можно ссылаться в тренере. Лучше всего это сделать путем отображения пакетов тренировочных и тестовых данных в словарь массивов NumPy именно с этими двумя столбцами:

```
import numpy as np

def to_labeled_image(batch): ❶
```

```

    return {
        "image": np.array([image.numpy() for image, _ in batch]),
        "label": np.array([label for _, label in batch]),
    }

train_dataset = train_dataset.map_batches(to_labeled_image) ②
test_dataset = test_dataset.map_batches(to_labeled_image)

```

- ① Преобразовывает каждый пакет данных, возвращая массивы NumPy `image` и `label`.  
 ② Применяет `map_batches` для преобразования исходных наборов данных.

Прежде чем перейти к тренировке модели, давайте взглянем на табл. 11.1, в которой показаны форматы входных данных, поддерживаемые библиотекой Ray Data<sup>1</sup>.

**Таблица 11.1. Экосистема библиотеки Ray Data**

Интеграция	Тип	Описание
Текстовые и двоичные файлы, файлы изображений, CSV, JSON	Базовые форматы данных	Поддержка таких базовых форматов, строго говоря, не должна рассматриваться как <i>интеграция</i> , однако стоит знать о том, что библиотека Ray Data может загружать и сохранять эти форматы
NumPy, Pandas, Arrow, Parquet, объекты Python	Продвинутые форматы данных	Библиотека Ray Data поддерживает работу с распространенными библиотеками данных машинного обучения, такими как NumPy и Pandas, но также может читать конкретно-прикладные объекты Python или файлы Parquet
Spark, Dask, MARS, Modin	Продвинутые сторонние интеграции	Фреймворк Ray взаимодействует с более сложными системами обработки данных посредством интеграций, разработанными сообществом, такими как Spark on Ray (Ray DP), Back on Ray, MEARS on Ray или Pandas on Ray (Modin)

Мы поговорим о взаимосвязи фреймворка Ray с такими системами, как Dask или Spark, подробнее в разделе «Фреймворки распределенных вычислений на Python» на стр. 263.

## Тренировка моделей

Подразделив набор данных CIFAR-10 надлежащим образом на тренировочные и тестовые данные с использованием библиотеки Ray Data, теперь можно определить классификатор, чтобы натренировать его на указанных данных. Поскольку это, вероятно, наиболее естественный сценарий, мы собираемся определить модель PyTorch, чтобы определить тренера AIR. Однако

<sup>1</sup> С постоянно обновляемым списком поддерживаемых форматов можно ознакомиться в документации библиотеки Ray Data (<https://oreil.ly/7pmuc>). Поддерживаемые форматы выходных данных библиотеки Ray Data в значительной степени совпадают с форматами входных данных.

стоит напомнить о том, что вы узнали из главы 7: на этом этапе можно легко переключаться с фреймворка на фреймворк и работать с Keras, Hugging Face, scikit-learn или любой другой библиотекой, поддерживаемой инструментарием AIR.

Мы выполним три шага: определим модель PyTorch, зададим цикл тренировки, который AIR должен выполнять с использованием этой модели, и определим тренера AIR, которого можно использовать на тренировочных данных. Для начала давайте определим простую сверточную нейронную сеть со сведением на основе максимума<sup>1</sup> и выпрямленными линейными активациями (`relu`) с помощью библиотеки PyTorch, которая создана для работы с набором данных CIFAR-10. Если вы знакомы с библиотекой PyTorch, то следующее ниже определение нейронной сети (`Net`) должно быть простым. Если вы с ней незнакомы, то достаточно знать, что для определения `torch.nn.Module` вам нужно предоставить всего одну вещь: определение прямого (`forward`) прохода по нейронной сети:

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Вы уже знаете, что для определения тренера AIR `TorchTrainer` для сети нужны тренировочный набор данных, масштабирование и опциональная конфигурация выполнения<sup>2</sup>. Инструментарию AIR также необходимо указать, что именно каждый работник должен делать при вызове метода `fit`, определив явный цикл тренировки, который обеспечивает максимальную гибкость процесса тренировки. В функции тренировки используется словарь `config`, который можно использовать во время выполнения, чтобы указывать свойства.

<sup>1</sup> Англ. max pooling. – Прим. перев.

<sup>2</sup> По поводу преобразования любой модели PyTorch в Ray AIR следует обратиться к руководству пользователя (<https://oreil.ly/k2QKV>) по данной теме.

Используемый здесь цикл тренировки – это как раз то, чего вы и ожидаете: мы загружаем модель и данные в каждого работника, а затем тренируем на пакетах данных в течение заданного числа эпох (при этом сообщая о ходе тренировки). Это довольно стандартный цикл тренировки, но есть пара важных моментов, в отношении которых следует соблюдать осторожность:

- для загрузки модели в работника следует использовать служебную функцию подготовки модели (`prepare_model`) из `ray.train.torch` на сети (`Net()`);
- для того чтобы обратиться к доступному для работника сегменту данных, следует обратиться к `get_dataset_shard` в текущем сеансе `ray.air.session`. В случае тренировки мы используем ключ "train" этого сегмента и преобразовываем его в пакеты нужного размера, используя функцию `iter_torch_batches`;
- для передачи информации об интересующих метриках тренировки следует использовать `session.report` из инструментария AIR.

Вот полное определение цикла тренировки PyTorch для каждого работника:

```
from ray import train
from ray.air import session, Checkpoint

def train_loop(config):
    model = train.torch.prepare_model(Net()) ❶
    loss_fct = nn.CrossEntropyLoss()
    optimizer = torch.optim.SGD(model.parameters(),
                                lr=0.001, momentum=0.9)

    train_batches = \
        session.get_dataset_shard("train").iter_torch_batches(❷
            batch_size=config["batch_size"],
        )

    for epoch in range(config["epochs"]):
        running_loss = 0.0
        for i, data in enumerate(train_batches):
            inputs, labels = data["image"], data["label"] ❸

            optimizer.zero_grad() ❹
            forward_outputs = model(inputs)
            loss = loss_fct(forward_outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item() ❺
            if i % 1000 == 0:
                print(f"[{epoch + 1}, {i + 1:4d}] loss: "
                      f"{running_loss / 1000:.3f}")
                running_loss = 0.0

        session.report(❻
            dict(running_loss=running_loss),
            checkpoint=Checkpoint.from_dict(
```

```
        dict(model=model.module.state_dict())
    ),
)
```

- ➊ Цикл тренировки задает модель, потерю и оптимизатор, используемые в самом начале. Обратите внимание на применение здесь функции `reparam_model`.
- ➋ Загрузить *сегмент* тренировочного набора данных в этого работника и создать итератор, содержащий пакеты данных из него.
- ➌ Согласно предыдущему определению, нашими данными (`data`) являются данные с типом `DataFrame` (кадр данных библиотеки Pandas), которые имеют столбцы "image" и "label".
- ➍ В цикле тренировки вычислить прямой и обратный проходы по сети, как мы сделали бы с любой моделью PyTorch.
- ➎ Отслеживать текущую потерю по каждому из 1000 тренировочных пакетов.
- ➏ Наконец, сообщить (`report`) об этой потере, используя сеанс AIR (`session`), передав контрольную точку (`checkpoint`) с текущим состоянием модели.

Определение этой функции, возможно, покажется немного длинным, учитывая, что мы выполняем довольно стандартную процедуру тренировки. Хотя в таких случаях, безусловно, можно было бы создать простую обертку вокруг моделей PyTorch, определение своего собственного цикла тренировки дает полную адаптируемость под решение более сложных сценариев. Мы передаем цикл тренировки (`training_loop`) в аргумент `train_loop_per_worker` тренера AIR и указываем конфигурацию этого цикла, передав словарь `train_loop_config` с необходимыми ключами.

Делая процесс еще более интересным и демонстрируя еще одну интеграцию фреймворка Ray, мы будем заносить результаты тренировочного про- гона `TorchTrainer` в MLflow путем передачи функции обратного вызова в кон- фигурацию выполнения (`RunConfig`), а именно функцию обратного вызова `MLflowLoggerCallback`:

```
from ray.train.torch import TorchTrainer
from ray.air.config import ScalingConfig, RunConfig
from ray.air.callbacks.mlflow import MLflowLoggerCallback

trainer = TorchTrainer(
    train_loop_per_worker=train_loop,
    train_loop_config={"batch_size": 10, "epochs": 5},
    datasets={"train": train_dataset},
    scaling_config=ScalingConfig(num_workers=2),
    run_config=RunConfig(callbacks=[
        MLflowLoggerCallback(experiment_name="torch_trainer")
    ])
)
result = trainer.fit()
```

Кроме того, можно использовать и другие сторонние библиотеки журна- лирования, такие как Weights & Biases или CometML, передавая аналогичные функции обратного вызова тренеру либо настройщику AIR<sup>1</sup>.

---

<sup>1</sup> Например, используя `WandbLoggerCallback`, можно не только регистрировать резуль- таты тренировок в Weights & Biases, но и, как показано в руководстве фреймворка Ray (<https://oreil.ly/2jWOj>), автоматически загружать свои контрольные точки.

В табл. 11.2 резюмированы все интеграции фреймворка Ray, связанные с тренировкой моделей машинного обучения, которые охватывают как библиотеку Ray Train, так и библиотеку RLlib.

**Таблица 11.2. Экосистема библиотек Ray Train и RLlib**

Интеграция	Тип
TensorFlow, PyTorch, XGBoost, LightGBM, Horovod	Интеграции библиотеки Train, сопровождаемые коллективом разработчиков фреймворка Ray
scikit-learn, Hugging Face, Lightning	Интеграции библиотеки Train, сопровождаемые сообществом
TensorFlow, PyTorch, OpenAI gym	Интеграции библиотеки RLlib, сопровождаемые коллективом разработчиков фреймворка Ray
AX, Unity	Интеграции библиотеки RLlib, сопровождаемые коллективом разработчиков фреймворка Ray

Мы проводим различие между интеграциями, спонсируемыми сообществом, и интеграциями, которые технически сопровождаются самим коллективом разработчиков фреймворка Ray. Большинство интеграций, о которых мы говорили в этой книге, были нативными, но из-за коллаборативной природы программного обеспечения с открытым исходным кодом часто не ощущается разницы в зрелости между нативными и сторонними интеграциями.

Подводя итог интеграциям AIR, связанным с тренировкой, в табл. 11.3 представлена общая картина экосистемы библиотеки Tune.

**Таблица 11.3. Экосистема библиотеки Ray Tune**

Интеграция	Тип
Optuna, Hyperopt, Ax, BayesOpt, BOHB, Dragonfly, FLAML, HEBO, Nevergrad, SigOpt, skopt, ZOOpt	Библиотека гиперпараметрической оптимизации
TensorBoard, MLflow, Weights & Biases, CometML	Управление журналированием и проведением экспериментов

## Подача моделей в качестве служб

Пакет *Gradio* – это популярный среди практиков способ демонстрировать свои модели машинного обучения, и он предоставляет целый ряд простых примитивов, служащих для создания элементов графического пользовательского интерфейса простым их описанием с помощью библиотеки *gradio* на Python. Как вы увидите, определять и развертывать интерфейсы Gradio совсем несложно, но еще проще затем их обертывать в так называемый сервер *GradioServer* из библиотеки Ray Serve, который позволяет масштабировать любое приложение Gradio в кластере Ray.

В целях демонстрации выполнения приложения Gradio с помощью библиотеки Ray Serve на модели, которую мы только что натренировали, давайте сначала сохраним результат процедуры тренировки на диске. Это делается путем сохранения соответствующей контрольной точки AIR в локальной

папке по нашему выбору, чтобы иметь возможность восстанавливать эту модель из контрольной точки в другом скрипте:

```
CHECKPOINT_PATH = "torch_checkpoint"
result.checkpoint.to_directory(CHECKPOINT_PATH)
```

Далее для простоты давайте создадим файл с именем *gradio\_demo.py* рядом с путем "torch\_checkpoint". В этом скрипте мы снова загрузим модель PyTorch, сначала восстановив контрольную точку TorchCheckpoint, а затем применим эту контрольную точку и определение сети (*Net()*), чтобы создать предсказателя TorchPredictor, который можно будет использовать для генерирования модельных предсказаний:

```
# gradio_demo.py
from ray.train.torch import TorchCheckpoint, TorchPredictor

CHECKPOINT_PATH = "torch_checkpoint"
checkpoint = TorchCheckpoint.from_directory(CHECKPOINT_PATH)
predictor = TorchPredictor.from_checkpoint(
    checkpoint=checkpoint,
    model=Net()
)
```

Обратите внимание, что для этого требуется импортировать или иным образом сделать доступным определение сети (*Net*) в скрипте *gradio\_demo.py*<sup>1</sup>.

Далее необходимо определить интерфейс Gradio (*Interface*), который мы определяем для приема изображений на входе и генерирования меток на выходе. Кроме того, необходимо указать, каким образом нужно преобразовывать входное изображение для генерирования метки (*Label*). По умолчанию Gradio представляет изображения в виде массивов NumPy, поэтому надо проверить, что этот массив имеет правильную форму и тип данных, а затем передать его нашему предсказателю (*predictor*). Поскольку указанный предсказатель, а именно *TorchPredictor*, производит распределение вероятностей, мы берем *argmax* его предсказания, чтобы получить целочисленный результат, который можно будет использовать в качестве метки. Поместите следующий ниже исходный код в свой скрипт Python *gradio\_demo.py*:

```
from ray.serve.gradio_integrations import GradioServer
import gradio as gr
import numpy as np

def predict(payload): ❶
    payload = np.array(payload, dtype=np.float32)
    array = payload.reshape((1, 3, 32, 32))
    return np.argmax(predictor.predict(array))

demo = gr.Interface(❷
    fn=predict,
```

---

<sup>1</sup> Для простоты мы продублировали определение сети в файле *gradio\_demo.py* на GitHub.

```

    inputs=gr.Image(),
    outputs=gr.Label(num_top_classes=10)
)

app = GradioServer.options( ③
    num_replicas=2,
    ray_actor_options={"num_cpus": 2}
).bind(demo)

```

- ❶ Функция `predict` соотносит данные Gradio на входе с данными Gradio на выходе, за- действуя предсказатель (`predictor`).
- ❷ Интерфейс Gradio имеет один вход (изображение), один выход (метку для 1 из 10 кате- горий набора данных CIFAR-10) и их соединяющую функцию `fn`.
- ❸ Мы привязываем (`bind`) демонстрацию Gradio (`demo`) к объекту `GradioServer` библиотеки Ray Serve, который развертывается на двух репликах с двумя центральными процессо- рами в каждой в качестве ресурсов.

Для выполнения этого приложения теперь можно просто набрать в оболочке следующую ниже команду<sup>1</sup>:

```
serve run gradio_demo:app
```

Указанная команда запускает поддерживаемую библиотекой Serve демон-страцию Gradio, к которой можно обратиться по адресу `localhost:8000`. При этом можно закачивать или перетаскивать изображения в соответствующее поле ввода и запрашивать предсказания, которые будут появляться в поле вывода результата приложения<sup>2</sup>.

Важно отметить, что этот пример на самом деле является всего лишь тон-кой оберткой вокруг Gradio и будет работать с любым другим приложением Gradio по вашему выбору. По сути дела, если вместо определения приложе-ния Serve (`app`) вы бы вызвали в своем скрипте `demo.launch()`, то с помощью команды `python gradio_demo.py` смогли бы его запустить как обычное при-ложение Gradio.

Есть еще одна примечательная деталь, которую легко упустить из виду, – мы подавали на вход предсказателя массив NumPy. Если вы проверите опре-деление формата данных, который мы использовали для тренировки, то вспомните, что предсказатель (`predictor`) должен работать на экземплярах `Dataset` библиотеки Ray Data. Экземпляр `predictor` достаточно умен, что-бы сделать вывод о том, что один элемент входных данных NumPy дол-

---

<sup>1</sup> При использовании интеграции Gradio библиотека Ray Serve под капотом будет автоматически выполнять приложение Gradio. Приложение Gradio оборудовано доступом к подсказкам типов на каждом развертывании Serve. Когда пользователь подает запрос, приложение Gradio отображает выходные данные каждого развер-тывания, используя Gradio-блок, который ставит им в соответствие тип выходных данных.

<sup>2</sup> Приложение ожидает изображения нужного размера, так как мы не хотели делать предобработку в `predict` более сложной, чем это необходимо. Вы можете использо-вать изображения, предоставленные в репозитории этой книги (<https://oreil.ly/7fS-l>), либо просто отыскать изображения CIFAR-10 в интернете, чтобы протестировать приложение самостоятельно.

жен содержать часть полных входных данных, связанную с изображениями ("image") (для генерирования модельных предсказаний нам не нужна метка ("label").

В завершение данного раздела ознакомьтесь с табл. 11.4, в которой перечислены текущие интеграции библиотеки Serve.

**Таблица 11.4. Экосистема библиотеки Ray Serve**

Библиотека	Описание
Фреймворки и приложения генерирования модельных предсказаний	FastAPI, Flask, Streamlit, Gradio
Объяснимость и наблюдаемость	Arize, Seldon Alibi, WhyLabs

## Разработка конкретно-прикладных интеграций

Прежде чем подробнее объяснить взаимосвязь фреймворка Ray с другими сложными программными фреймворками, давайте поговорим о том, как разрабатывать свои собственные интеграции для инструментария Ray AIR. Поскольку инструментарий AIR был сконструирован с учетом расширяемости, подходящие интерфейсы можно найти для всех заданий, для которых вы хотите разрабатывать конкретно-прикладные интеграции.

Например, допустим, вы хотите прочитать данные из Snowflake, натренировать на них модель JAX и записать результаты настройки в Neptune<sup>1</sup>. На момент написания этой книги такие интеграции недоступны, но, вероятно, в будущем все изменится. Мы выбрали эти интеграции (Snowflake, JAX, Neptune) не для того, чтобы продемонстрировать какие-либо предпочтения; они просто оказались интересными инструментами экосистемы. В любом случае стоит знать, как такие интеграции разрабатываются.

При загрузке данных из Snowflake в набор данных Dataset необходимо создать новый источник данных (Datasource). Указанный источник данных задается путем детализации того, как его настраивать (`create_reader`), как писать в источник (`do_writes`) и что происходит при успешных и безуспешных попытках записи (`on_write_complete` и `on_write_failed`). При наличии конкретной реализации источника данных `SnowflakeDatasource` можно прочитать свои данные в набор данных Dataset:

```
from ray.data import read_datasource, datasource

class SnowflakeDatasource(datasource.Datasource):
    pass

dataset = read_datasource(SnowflakeDatasource(), ...)
```

<sup>1</sup> В этой главе мы будем исходить из того, что вы знаете об этих совершенно разных инструментах экосистемы. Если это не так, то в данном разделе достаточно понимать, что Snowflake – это решение для работы с базами данных, с которым вы, возможно, захотите интегрироваться, JAX – это фреймворк машинного обучения, а Neptune можно использовать для отслеживания экспериментов.

Далее, предположим, у вас есть интересная модель JAX, которую вы хотите масштабировать, используя возможности библиотеки Ray Train. В частности, давайте допустим, что вы хотите взять указанную модель и выполнить ее тренировку с параллелизмом данных, то есть натренировать эту однушкую модель параллельно на нескольких сегментах данных. Для этой цели фреймворк Ray поставляется с так называемым тренером с параллелизмом данных (`DataParallelTrainer`). При определении одного из них для фреймворка тренировки нужно создать цикл тренировки в расчете на одного работника (`train_loop_per_worker`) и определить то, как JAX должен обрабатываться библиотекой Train на внутреннем уровне<sup>1</sup>. С помощью реализации тренера `JaxTrainer` можно использовать тот же интерфейс `Trainer`, который мы использовали во всех примерах с применением инструментария AIR:

```
from ray.train.data_parallel_trainer import DataParallelTrainer

class JaxTrainer(DataParallelTrainer):
    pass

trainer = JaxTrainer(
    ...,
    scaling_config=ScalingConfig(...),
    datasets=dict(train=dataset),
)
```

Наконец, для того чтобы использовать Neptune для журналирования и визуализации попыток Tune, можно определить обратный вызов регистратора (`LoggerCallback`), который передается в конфигурацию выполнения настройщика. В целях определения указанного обратного вызова нужно указать то, как создавать регистратор (`setup`), что должно происходить в начале и конце попыток (`log_trial_start` и `log_trial_end`) и как регистрировать свои результаты (`log_trial_result`). Если вы реализовали такой класс, например `NeptuneCallback`, то сможете его использовать так же, как мы использовали обратный вызов `MLflowLogger` в разделе «Тренировка моделей» на стр. 253:

```
from ray.tune import logger, tuner
from ray.air.config import RunConfig

class NeptuneCallback(logger.LoggerCallback):
    pass

tuner = tuner.Tuner(
    trainer,
    run_config=RunConfig(callbacks=[NeptuneCallback()])
)
```

Хотя разрабатывать интеграции не всегда так трудно, как кажется на первый взгляд, стороннее программное обеспечение является движущейся

---

<sup>1</sup> Если быть точным, вы должны определить бэкенд (`Backend`) для JAX вместе с конфигурацией бэкенда (`BackendConfig`). Затем ваш тренер с параллелизмом данных (`DataParallelTrainer`) должен быть инициализирован этим бэкендом и вашим циклом тренировки.

целью, и задача технического сопровождения интеграций бывает непростой. Тем не менее теперь вам известны три наиболее распространенных сценария интеграции новых компонентов инструментария AIR, и, возможно, вы почувствуете желание поработать над интеграцией своего любимого инструмента, спонсируемого сообществом.

## Обзор интеграций фреймворка Ray

Давайте резюмируем все упомянутые в этой главе (и на протяжении всей книги) интеграции на одной краткой диаграмме. На рис. 11.1 перечислены все интеграции, доступные на момент написания книги.

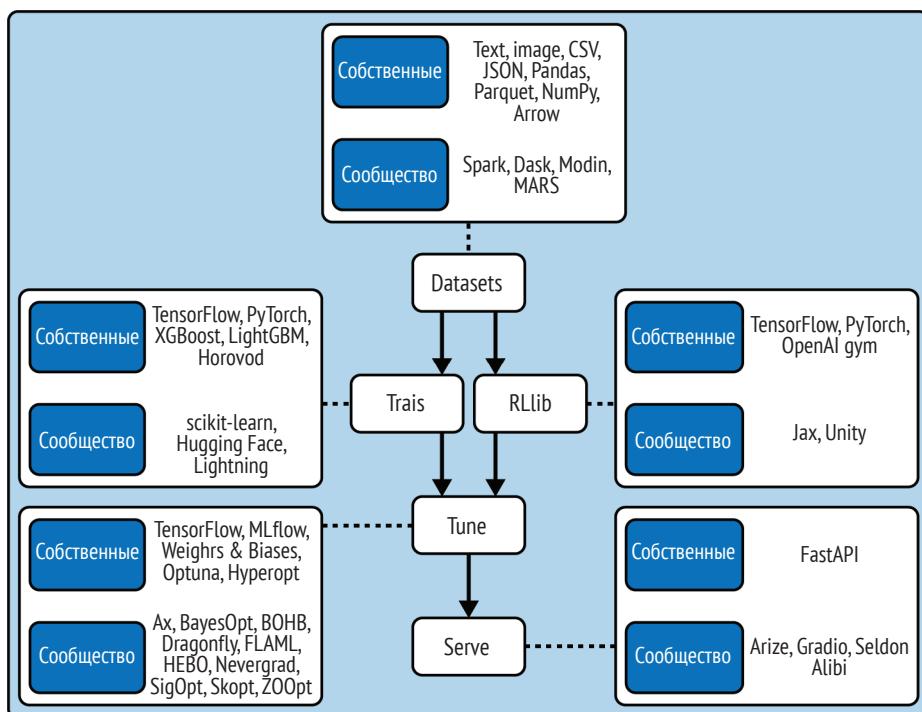


Рис. 11.1 ♦ Экосистема инструментария Ray AIR в обобщенном виде

## ФРЕЙМВОРК RAY И ДРУГИЕ СИСТЕМЫ

До этого момента мы не проводили никаких прямых сравнений с другими системами по той простой причине, что нет смысла сравнивать фреймворк Ray с чем-то еще, если вы еще не очень хорошо понимаете, что он из себя представляет. Поскольку фреймворк Ray достаточно гибок и поставляется

с большим количеством компонентов, его можно сравнить с разными типами инструментов в более широкой экосистеме машинного обучения.

Давайте начнем со сравнения наиболее очевидных кандидатов, а именно фреймворков на основе Python для кластерных вычислений.

## Фреймворки распределенных вычислений на Python

Если рассмотреть фреймворки распределенных вычислений, которые предлагают полную поддержку Python и не привязывают вас к какому-либо облачному предложению, то текущей «большой тройкой» являются Dask, Spark и Ray. Хотя между этими фреймворками существуют определенные технические и зависящие от контекста различия в производительности, лучше всего их сравнивать с точки зрения рабочих нагрузок, которые вы хотите на них выполнять. В табл. 11.5 сравниваются наиболее распространенные типы рабочих нагрузок.

**Таблица 11.5. Сравнение поддержки типов рабочих нагрузок в Ray, Dask и Spark**

Тип рабочей нагрузки	Dask	Spark	Ray
Структурированная обработка данных	Первоклассная поддержка	Первоклассная поддержка	Поддерживается посредством библиотеки Ray Data и интеграций, но не на первоклассном уровне
Низкоуровневый параллелизм	Первоклассная поддержка посредством заданий	Отсутствует	Первоклассная поддержка посредством заданий и акторов
Рабочие нагрузки глубокого обучения	Поддерживается, но не на первоклассном уровне	Поддерживается, но не на первоклассном уровне	Первоклассная поддержка посредством нескольких библиотек машинного обучения

## Инструментарий Ray AIR и более широкая экосистема машинного обучения

Инструментарий Ray AIR сфокусирован в первую очередь на вычислениях с использованием искусственного интеллекта, например предоставляя любой вид распределенной тренировки с помощью библиотеки Ray Train, но он не предназначен для охвата всех аспектов рабочей нагрузки с использованием искусственного интеллекта. Например, AIR предпочитает интегрироваться с инструментами отслеживания и мониторинга экспериментов по машинному обучению и с технологическими решениями по хранению данных, а не предоставлять нативные решения. В табл. 11.6 указаны взаимодополняющие компоненты экосистемы.

**Таблица 11.6. Взаимодополняющие компоненты экосистемы**

Категория	Примеры
Отслеживание и наблюдаемость машинного обучения	MLflow, Weights & Biases, Arize и др.
Фреймворки тренировки	PyTorch, TensorFlow, Lightning, JAX и др.
Хранилища признаков моделей машинного обучения	Feast, Tecton и др.

На другой стороне спектра находятся категории инструментов, для которых инструмент Ray AIR можно считать альтернативой. Например, существует целый ряд специфичных для фреймворка наборов инструментов, таких как TorchX или TFX, которые тесно связаны с соответствующими фреймворками. В отличие от них, инструментарий AIR не зависит от фреймворка, тем самым предотвращая привязку к поставщику, и предлагает аналогичный инструментарий<sup>1</sup>.

Помимо этого, интересно вкратце остановиться на том, как инструментарий Ray AIR соотносится с конкретными облачными предложениями. Некоторые крупные облачные сервисы предлагают тщательно продуманные наборы инструментов, служащие для решения рабочих нагрузок машинного обучения на Python. Назовем лишь один из них: AWS Sagemaker является отличным пакетом «все в одном», который позволяет надежно подключаться к вашей экосистеме AWS. Инструментарий AIR не стремится заменить такие инструменты, как SageMaker. Вместо этого он нацелен на предоставление альтернатив для таких ресурсоемких компонентов, как тренировка моделей, их оценивание и подача моделей как служб<sup>2</sup>.

Инструментарий AIR также представляет собой приемлемую альтернативу фреймворкам рабочего процесса машинного обучения, таким как KubeFlow или Flyte. В отличие от многих технологических решений на основе контейнеров, AIR предлагает интуитивно понятный высокогорлевый API на основе Python и нативную поддержку распределенных данных. В табл. 11.7 приводится резюме указанных альтернатив.

**Таблица 11.7. Альтернативные компоненты экосистемы**

Категория	Примеры
Специфичные для фреймворка наборы инструментов	TorchX, TFX и пр.
Фреймворки рабочего процесса машинного обучения	KubeFlow, Flyte, FB Learner FFlow

<sup>1</sup> Этот факт представляет собой очевидный компромисс, поскольку специфичные для фреймворка инструменты сильно адаптированы и предлагают множество преимуществ.

<sup>2</sup> Платформа Anyscale сама по себе предоставляет управляемый сервис с функциональными возможностями уровня предприятия по поддержанию разработки приложений машинного обучения поверх фреймворка Ray.

Иногда ситуация не столь однозначна, и инструментарий Ray AIR можно рассматривать или использовать как альтернативный либо дополняющий компонент экосистемы машинного обучения.

Например, фреймворк Ray и, в частности, инструментарий AIR как системы с открытым исходным кодом можно использовать внутри размещенных платформ машинного обучения, таких как SageMaker, но с их помощью также можно строить свои собственные платформы машинного обучения<sup>1</sup>. Кроме того, как уже упоминалось, AIR не всегда может конкурировать со специализированными системами обработки больших данных, такими как Spark или Dask, но часто бывает так, что библиотеки Ray Data с ее распределенными наборами данных Dataset будет достаточно для удовлетворения ваших потребностей в обработке.

Как мы упоминали в главе 10, центральное место в философии внутреннего устройства инструментария AIR занимает возможность выражать ваши рабочие нагрузки машинного обучения в одном-единственном скрипте и исполнять его во фреймворке Ray как одной-единственной распределенной системе. Поскольку Ray самостоятельно обрабатывает размещение и исполнение всех заданий в вашем кластере, обычно нет необходимости в оркестровке ваших рабочих нагрузок в явной форме (или в сшивании множества сложных распределенных систем). Конечно же, эту философию не следует воспринимать слишком буквально – иногда требуется несколько систем либо разделять задания на несколько этапов. С другой стороны, специальные инструменты оркестровки рабочего процесса, такие как Argo или AirFlow, бывают очень полезны в комплементарном плане<sup>2</sup>. Например, у вас может возникнуть желание выполнять фреймворк Ray в качестве шага во фреймфорке Lightning MLOps. В табл. 11.8 представлен обзор компонентов, которые могут использоваться наряду с инструментарием AIR или для которых AIR может быть альтернативой.

**Таблица 11.8. Компоненты экосистемы, которые инструментарий AIR может дополнять или заменять**

Категория	Примеры
Платформы машинного обучения	SageMaker, Azure ML, Vertex AI, Databricks
Системы обработки данных	Spark, Dask
Оркестровщики рабочих процессов	Argo, AirFlow, Metaflow
Фреймворки MLOps	ZenML, Lightning

<sup>1</sup> В следующем разделе мы дадим приблизительный набросок того, как это сделать.

<sup>2</sup> Во фреймворке Ray есть библиотека под названием Ray Workflows, которая в настоящее время находится в альфа-версии. По сравнению с такими инструментами, как AirFlow, библиотека Workflows является более низкоуровневой, но позволяет выполнять длительные рабочие процессы приложений нативно во фреймворке Ray. Более подробная информация о библиотеке Workflows находится в документации фреймворка Ray (<https://oreil.ly/XUT7y>).

Если у вас уже есть платформа машинного обучения, такая как Vertex или SageMaker, то вы можете использовать любое подмножество инструментария Ray AIR для расширения своей системы<sup>1</sup>. Другими словами, AIR может дополнять существующие платформы машинного обучения, интегрируясь с существующими оркестровщиками конвейеров и рабочих процессов, службами хранения и отслеживания, не требуя замены всей вашей платформы машинного обучения.

## Как интегрировать инструментарий AIR в свою платформу машинного обучения

Теперь, когда у вас есть более глубокое понимание взаимосвязи фреймворка Ray, и инструментария AIR в частности, с другими компонентами экосистемы, давайте подведем итог тому, что требуется для разработки своей собственной платформы машинного обучения и интеграции фреймворка Ray с другими компонентами экосистемы.

Ядро вашей системы машинного обучения, построенной с помощью инструментария AIR, состоит из набора кластеров Ray, каждый из которых отвечает за разные наборы заданий. Например, один кластер может выполнять предобработку данных, тренировать модель PyTorch и генерировать модельные предсказания; другой может просто подбирать ранее натренированные модели для пакетного генерирования предсказаний и подачи моделей как служб и так далее. Вы можете задействовать автомасштабировщик Ray для удовлетворения своих потребностей в масштабировании и развернуть всю систему в Kubernetes с помощью KubeRay. Затем вы можете дополнить эту стержневую систему другими компонентами по своему усмотрению, например:

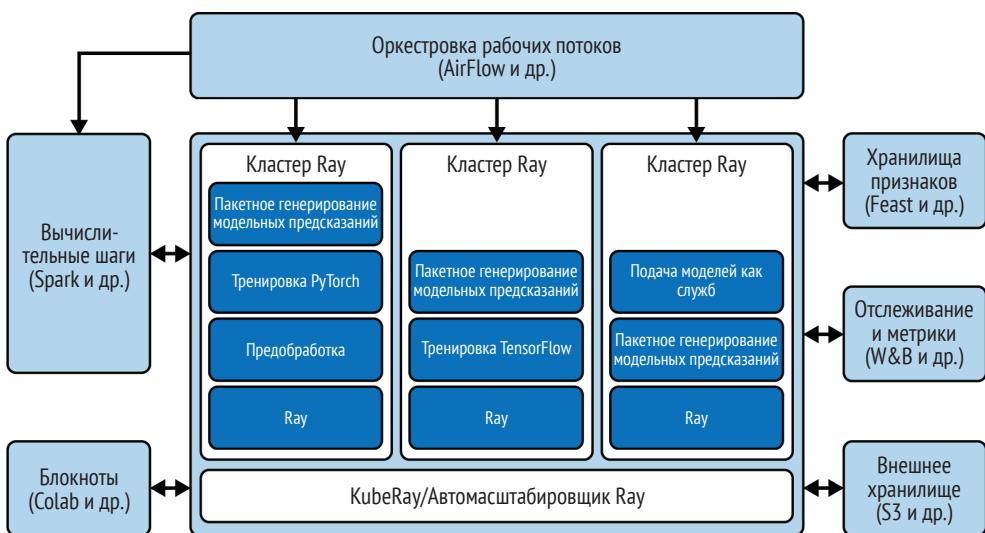
- возможно, вы захотите добавить в свою среду другие вычислительные шаги, такие как выполнение интенсивных по использованию данных задачий предобработки с помощью Spark;
- вы можете использовать оркестровщик рабочих процессов, такой как конвейеры AirFlow, Oozie или SageMaker, чтобы планировать и создавать кластеры Ray и выполнять приложения и службы Ray AIR. Каждое приложение AIR может быть частью более крупного оркестрированного рабочего процесса, например путем привязки к заданию Spark ETL из первого пункта списка<sup>2</sup>;
- вы также можете создавать свои кластеры Ray AIR для интерактивного использования с блокнотами Jupyter, например размещенными в Google Colab или Databricks Notebook;

<sup>1</sup> Мы используем термин «платформа машинного обучения» в самом широком смысле, какой только возможен, а именно для обозначения любой системы, которая отвечает за выполнение сквозных рабочих нагрузок машинного обучения.

<sup>2</sup> Оркестровка графов заданий может выполняться полностью в инструментарии Ray AIR. Внешние оркестровщики рабочих процессов будут хорошо интегрироваться, но они необходимы только при выполнении шагов, не связанных с фреймворком Ray.

- если вам нужен доступ к хранилищу признаков, такому как Feast или Tecton, то библиотеки Ray Train, Data и Serve имеют интеграцию для таких инструментов<sup>1</sup>;
- для отслеживания экспериментов или хранения метрик библиотеки Ray Train и Tune обеспечивают интеграцию с такими инструментами, как MLflow и Weights & Biases;
- как показано на рисунке, вы также можете извлекать и сохранять свои данные и модели из внешних технологических решений по хранению данных, таких как S3.

На рис. 11.2 все части объединены в одну сжатую диаграмму.



**Рис. 11.2 ♦** Разработка своей собственной платформы машинного обучения с помощью инструментария Ray AI Runtime и других компонентов экосистемы машинного обучения

## КУДА ОТСЮДА ДВИГАТЬСЯ ДАЛЬШЕ?

Мы прошли долгий путь от общего обзора фреймворка Ray в главе 1 до обсуждения его экосистемы в данной главе. Но поскольку эта книга является вводной, мы только прикоснулись к хрестоматийной верхушке айсберга функциональных возможностей фреймворка Ray. Хотя теперь вы должны хорошо разбираться в основах ядра фреймворка – Ray Core – и в том, как работают кластеры Ray, и знать, когда использовать инструментарий AIR и входящие в его состав библиотеки Data, Train, RLLib, Tune и Serve, вам еще

<sup>1</sup> Коллектив разработчиков фреймворка Ray написал демонстрационный пример (<https://oreil.ly/Pi3XF>) эталонной архитектуры, иллюстрирующий интеграцию с Feast.

многое предстоит узнать о каждом аспекте.

Начнем с того, что обширные руководства пользователя по Ray Core<sup>1</sup> дадут вам гораздо более глубокое представление о заданиях, акторах и объектах Ray, их размещении на узлах кластера и о том, как оперировать зависимостями ваших приложений. В частности, вы найдете интересные шаблоны и антишаблоны, позволяющие хорошо разрабатывать ваши программы Ray Core и избегать распространенных ошибок. Для того чтобы узнать о внутренних механизмах фреймворка Ray побольше, рекомендуем ознакомиться с несколькими продвинутыми статьями о фреймворке Ray, такими как техническая документация по архитектуре<sup>2</sup>.

Мы полностью пропустили одну интереснейшую тему – это инструменты фреймворка Ray, связанные с *наблюдаемостью*. Официальная документация по наблюдаемости в рамках фреймворка Ray<sup>3</sup> является хорошей отправной точкой для данной темы. Там вы научитесь отлаживать и профилировать свои приложения Ray, регистрировать информацию из своих кластеров Ray, отслеживать их поведение и экспортировать важные метрики. Там же вы найдете введение в приборную панель Ray<sup>4</sup>, которая поможет вам разобраться в своих программах Ray.

В центре внимания этой книги было знакомство практиков машинного обучения с ключевыми идеями фреймворка Ray и получение ими практических отправных точек для решения своих рабочих нагрузок с помощью Ray. Однако темой, заслуживающей большего внимания, чем одна глава, которую можно было бы включить в эту книгу, является ответ на вопрос, как создавать, масштабировать и сопровождать кластеры Ray. В настоящее время самым лучшим введением в расширенные темы кластеров Ray является документация фреймворка Ray<sup>5</sup>. Там вы сможете узнать гораздо подробнее о том, как развертывать кластеры у всех основных облачных провайдеров, как масштабировать кластеры в Kubernetes и как направлять задания Ray в кластер, чем мы могли бы рассказать здесь.

В этой главе мы смогли лишь вскользь упомянуть о большинстве интеграций Ray. Если вы хотите узнать о сторонних интеграциях Ray подробнее, то хорошей отправной точкой будет страница экосистемы фреймворка Ray<sup>6</sup>. Кроме того, не менее важно упомянуть, что есть еще другие библиотеки Ray<sup>7</sup>, которые просто не вошли в эту книгу, например Ray Workflows и распределенная, упрощенная замена библиотеки Python `multiprocessing`.

Наконец, если вы заинтересованы в том, чтобы стать частью сообщества фреймворка Ray, то это можно сделать несколькими способами. Вы можете

---

<sup>1</sup> См. <https://oreil.ly/cX6vj>.

<sup>2</sup> См. [https://oreil.ly/lfW\\_h](https://oreil.ly/lfW_h).

<sup>3</sup> См. <https://oreil.ly/xDWtK>.

<sup>4</sup> На момент написания этой книги приборная панель находилась в стадии капитального ремонта. Внешний вид и функциональность могут кардинально измениться, поэтому мы не включили сюда ни описания, ни скриншоты.

<sup>5</sup> См. <https://oreil.ly/SGa9w>.

<sup>6</sup> См. <https://oreil.ly/5wqYZ>.

<sup>7</sup> См. <https://oreil.ly/Yt5hX>.

присоединиться к Ray Slack<sup>1</sup>, чтобы быть на связи с разработчиками фреймворка Ray и другими членами сообщества, либо присоединиться к дискуссионному форуму Ray<sup>2</sup>, чтобы получать ответы на свои вопросы. Если вы хотите помочь в разработке фреймворка Ray, будь то внесение вклада в документацию, добавление нового варианта использования или помочь сообществу с открытым исходным кодом новыми функциональными возможностями или исправлениями ошибок, то вам следует ознакомиться с официальным руководством для участников проекта Ray<sup>3</sup>.

## РЕЗЮМЕ

В этой главе вы узнали больше об экосистеме фреймворка Ray с точки зрения ее инструментария AI Runtime. Вы ознакомились с полным спектром интеграций библиотек инструментария Ray AIR и примером тренировки и подачи модели машинного обучения как службы с использованием трех разных интеграций: PyTorch для загрузки данных и тренировки модели, MLflow для журналирования и Gradio для подачи модели как службы. Теперь вы должны быть в состоянии отправиться в путь и провести свои собственные эксперименты с использованием инструментария AIR вместе со всеми инструментами, которые вы уже применяете или собираетесь применять в будущем. Мы также обсудили ограничения фреймворка Ray, сравнили его с различными родственными системами и способы применения фреймворка Ray с другими инструментами с целью обогащения или разработки своих собственных платформ машинного обучения.

На этом завершается данная глава и вся книга. Мы надеемся, что она пробудила ваш интерес к фреймворку Ray и помогла вам начать свое путешествие вместе с ним. Внедрение инструментария AIR привнесло много новых функциональных возможностей в экосистему фреймворка Ray, и, безусловно, на дорожной карте есть еще много интересного. Нет сомнений, что теперь вы можете погрузиться в любой продвинутый материал по фреймворку Ray – возможно, у вас уже есть планы по разработке своего первого собственного приложения на базе фреймворка Ray.

<sup>1</sup> См. <https://oreil.ly/a83QM>.

<sup>2</sup> См. <https://discuss.ray.io/>.

<sup>3</sup> См. <https://oreil.ly/6AgiE>.

# Тематический указатель

## Символы

.bind, API  
создание экземпляра-копии  
развертывания, 194  
.predict\_pipelined, функция, 176  
@serve.batch, декоратор, 197  
@serve.deployment, декоратор, 43, 193  
настройка реплик  
и ресурсообеспечение  
развертывания, 195

## А

adapter, функция, 239  
AdvancedEnv, класс, 120  
Airflow, 165  
AI Runtime. См. *Ray AIR*  
Algorithm, класс, 99  
доступ ко всем экземплярам класса  
Algorithm в работниках, 104  
предоставление учебной программы  
в рамках библиотеки RLlib, 120  
тренировка в многоагентной среде, 114  
AlgorithmConfig, класс, 99  
AlphaFold, 25  
Amazon, веб-сервисы (AWS), 225  
Apache Airflow, 165  
Apache Arrow, 34  
проект Plasma, 61  
распределенность и применение  
в наборах данных Dataset фреймворка  
Ray, 149  
Apache Hadoop, 147  
Apache Spark, 147, 263  
API на базе обработки ЕЯ  
разработка примера, 202  
архитектура конвейера обработки  
ЕЯ по резюмированию статей  
Википедии, 203

доставка содержимого  
и предобработка, 204  
модели обработки ЕЯ, 204  
обработка HTTP и логика  
драйвера, 206  
собираем все воедино, 208  
Arrow для Python, установка, 34  
asyncio, возможности (Python), 197, 198  
Atari, среды (gym), 97  
await, синтакс (Python), 198  
AWS (веб-сервисы Amazon), 225  
Azure, 226

## Б

BaseEnv, класс, 109  
base\_model, 104  
batch\_timeout\_wait\_s, 197  
bayesian-optimization, библиотека, 133  
BayesOptSearch, класс, 134

## С

cartpole-ppo, отрегулированный  
пример, 38  
CartPole-v1, среда, 38  
CIFAR-10, набор данных, 252  
classify, метод, 197  
classify\_batched, метод, 197  
CLI (интерфейс командной строки)  
использование CLI-инструмента  
запуска кластеров Ray для  
развертывания кластера, 223  
работа с RLlib CLI, 97  
Codex, 25  
compute\_actions, метод, 102  
CSV-файл  
запись/чтение из наборов данных  
Dataset фреймворка Ray, 150  
 чтение из корзины S3 в столбчатый  
набор данных, 232

**D**

Dask, 44, 263  
 библиотека Dask on Ray, 251  
 встроенная поддержка служебных функций Python по работе с датой/временем, 172  
**Dask on Ray**  
 использование для тренировки нейронной сети PyTorch, 171  
 пример, 162  
**Data**, библиотека, экосистема, 253  
**DataFrame**, наборы данных  
 внешние системы обработки, 161  
 вызовы df.compute, 163  
 конвертирование полезной нагрузки службы предсказания, 238  
 Dask, 162  
 Dask on Ray, 171  
**DataFrame**, наборы данных библиотеки Pandas, 153  
**DataParallel** (PyTorch), 173  
**DataParallelTrainer**, 261  
**Data (Ray)**, библиотека, 34. См. *Ray Data, библиотека*  
 преобразование наборов данных, 34  
 распределенные наборы данных, 34  
**Dataset**, наборы данных  
 выгоды, 148, 149  
 вычисление на наборах данных  
**Dataset**, 153  
 загрузка данных в Ray AIR, 232  
 интеграции с внешними библиотеками, 161  
 использование в распределенном пакетном генерировании модельных предсказаний, 176  
 использование для вычислений без отслеживания внутреннего состояния, 241  
 использование с помощью библиотеки Ray Train для реализации полного рабочего процесса машинного обучения как одно-единственное приложение, 176  
 использование тренерами Ray AIR, 235  
 конвейеры Dataset, 155  
 основы, 149  
 блоки и реорганизация блоков, 152  
 встроенные преобразования, 151  
 создание набора данных, 150  
 схемы и форматы данных, 152  
 чтение и запись в хранилище, 150

пример возможностей, 252  
 пример параллельной тренировки копий классификатора, 157  
 разработка конвейера машинного обучения, 164  
 стратегия планирования, 244  
**DatasetPipeline**, 36  
 конверсия наборов данных Dataset, 157  
 создание с использованием функции ds.repeat, 157, 160  
**Dataset (Ray)**, 147. См. *Набор данных Dataset фреймворка Ray*  
 преобразование наборов данных Dataset, 35  
 содержимое наборов данных Dataset, 149  
 создание набора данных Dataset, 150  
**datasets**, аргумент (XGBoostTrainer в AIR), 234  
**datasets**, словарь (тренеры в библиотеке Ray Train), 176, 180  
**Deepmind**, AlphaFold, 25  
**Dense**, слои, 104, 143  
**deployment.options**, API, 195  
**discount\_factor**, параметр, 126  
**Discrete**, класс, 76  
**DQN**. См. *Q-сеть глубокая*  
**DQNConfig**, объект, 99  
 метод multi\_agent, 114  
**ds.repeat**, функция, 157, 160  
**ds.window**, функция, 36

**E**

**Environment**, класс, 76  
 применение, чтобы сыграть в 2-мерный лабиринт, 79  
 реализованные методы, 77  
**environment**, метод (AlgorithmConfig), 107  
**evaluate**, команда (rllib), 39, 98  
**example**, команда (rllib), 39  
**ExperimentAnalysis**, объект, 131  
**exploration**, метод (AlgorithmConfig), 107  
**ExternalEnv**, класс, 110

**F**

**FastAPI**, фреймворк, 194  
**fetch\_wikipedia\_page**, логика драйвера, 204, 207  
**filter**, операция, 149  
**filter**, функция, 35  
**fit**, метод (тренер), 180

`fit`, метод (`Trainer`), 177  
`flat_map`, функция, 35

**G**

`GCS` (Служба глобального управления), 64, 65  
`get_best_config`, функция, 131  
`GitHub`-репозиторий книги, 19  
`Google Cloud`, 226  
`GPT-2`, модель, 42  
`gpu_intensive_inference`, 155  
`Gradio`, пакет, 257  
`groupby`, 149  
`gRPC`, фреймворк, 64  
`gym`, библиотека, 95  
  разработка среды `gym`, 96  
  установка, 37  
`gym.Env`, 109  
  обертка `VectorEnv`, 110  
`GymEnvironment`, класс  
  определение многоагентной версии, 111  
  реализация среды `gym.Env`, 96  
  использование библиотеки `RLLib`, 98  
`gymnasium`, библиотека, 97

**H**

`Hadoop`, 147  
`HPO`. См. *Оптимизация гиперпараметрическая*  
`HTTP`, опрашиваемая служба генерирования модельных предсказаний, 238  
`HTTP`-запрос, определение логики, 193  
`HTTP API` и логика драйвера, определение для API на базе обработки ЕЯ, 206  
`Hugging Face`, модели на языке Python, 42  
`Hugging Face Transformers`, библиотека, 193, 204  
`HyperOptSearch`, алгоритм, 144

**I**

`init`, функция, 30  
`ipython`, интерпретатор, 30  
`iter_torch_batches`, функция, 175

**J**

`JAX`, 260  
`JsonLoggerCallback`, 185

**K**

`Keras`, 45, 104  
  настройка моделей `Keras` в библиотеке `Ray Tune`, 142  
`kubectl`, команда, выполнение программ `Ray`, 217  
`KubeRay`, проект, 214  
`Kubernetes`, 31, 65, 209  
  развертывание кластера `Ray`, 214  
  взаимодействие с кластером  
`KubeRay`, 216  
  конфигурирование журналирования для `KubeRay`, 222  
  конфигурирование оператора `KubeRay`, 219  
  настройка первого кластера `KubeRay`, 215  
  оператор `KubeRay`, 214

**L**

`label_column`, аргумент (`XGBoostTrainer` в AIR), 234  
`lambda`, функции, 153  
`LightGBM`, фреймворк, 177  
`LightGBMTrainer`, класс, 177  
`load_dataset`, функция (`Ray Train`), 171, 175  
`LoggerCallback`, интерфейс, 185

**M**

`map`, операция, 149  
`map`, функция, 35  
  выполнение конкретно-прикладных преобразований на наборах данных `Dataset`, 153  
`MapReduce`  
  пример использования `Ray`, 66  
  выполнение `MapReduce` на распределенном корпусе документов, 67  
  отображение и перетасовка данных документа, 69  
  редукция количеств слов, 70  
`MARL` (многоагентное обучение с подкреплением)  
  поддержка со стороны `RLLib`, 114  
  тренировка задачи, 114  
`max_batch_size`, 197  
`max_depth`, параметр (модель `XGBoost`), 236  
`min`, агрегация, 152

MinMaxScaler, 183  
 MLFlow, 251, 256  
 MLFlowLogger, 251  
 MLFLowLoggerCallback, 256  
 MNIST, данные, 142  
 model.state\_value\_head.summary, метод, 105  
 multi\_agent, метод (AlgorithmConfig), 107  
 MultiAgentEnv, определение с двумя агентами, 111  
 MultiAgentEnv, класс, 110

**N**

NAT (Трансляция сетевых адресов), 213  
 nc, инструмент, 213  
 Neptune, 260  
 nmap, инструмент, 213  
 num\_gpus, 196  
 numpy.square, оптимизированная реализация, 154  
 num\_replicas, 196, 205

**O**

offline\_data, метод (AlgorithmConfig), 107  
 OpenAI  
     искусственный интеллект и вычисления, 25  
     Codex, 25

**P**

Pandas on Ray, 45  
 Parquet, данные, 151, 153  
     Dataset-преобразования на данных  
     Parquet, 155  
 PodTemplate, 214  
 Policy, класс, 81  
     замена в будущих выпусках библиотеки Rllib, 103  
     обновление значений в таблице state\_action\_table, 83  
     таблица state\_action\_table, 82  
 PolicyClient, 117  
 PolicyServerInput, объект, 115  
 PredictorDeployment, класс, 238  
 prepare\_model, функция, 174  
 Prometheus, 214  
 Pydantic, 206  
 Python  
     версии и поддержка Ray, 30

глобальная блокировка интерпретатора, 51  
 Дзен языка, 68  
 поддержка библиотекой Dask служебных функций Python по работе с датой/временем, 172  
 применение для науки о данных, 23  
 фреймворки распределенных вычислений, 263  
 Ray для сообщества науки о данных, 24  
 Rllib API. См. *Rllib Python API*  
 Python как гибкий распределенный язык для машинного обучения, 24  
 PyTorch, 94, 104  
     выполнение миграции существующей модели в библиотеку Ray Train, 177, 179  
     загрузка и преобразование набора данных, 251  
     поддержка библиотек Ray Rllib и Train, 45  
     расширение torchvision, 252  
     тренировка нейронной сети PyTorch с использованием Dask on Ray, 171  
 DataParallel, 173  
 loss\_function и batch\_loss, 175

**Q**

Q-значение, 90, 103. См. *Значение состояния–действия*  
     получение для моделей DQN, 106  
 Q-обучение, алгоритм, 84, 90  
 Q-обучение глубокое, 86, 103  
 Q-сеть глубокая (DQN), 86  
     использование объекта DQNConfig для определения DQN-сети, 98

**R**

random.uniform, функция взятия образцов (numpy), 140  
 Ray, 24  
     истоки, 25  
     по отношению к другим системам, 65  
     принципы внутреннего устройства, 26  
         гибкость (Ray), 27, 72  
         динамическое исполнение, 27  
         простота, 27  
         разнородность (Ray), 27  
     техническая документация по архитектуре, 53  
     установка, 30

- экосистема, 44. См. Экосистема
- Ray AIR (AI Runtime)**, 29, 173, 229
- и рабочий процесс науки о данных, 32
  - ключевые концепции инструментария AIR на примере
    - настройщики и контрольные точки, 235
    - от загрузки данных до генерирования модельных предсказаний с помощью AIR, 231
    - пакетные предсказатели, 237
    - тренеры, 233
  - ключевые концепции инструментария AIR на примере, 231
  - наборы данных и предобработчики, 232
  - подходящие рабочие нагрузки
    - автомасштабирование рабочих нагрузок AIR, 248
    - исполнение рабочих нагрузок, 244
    - принятая в AIR модель сбоя, 247
    - специфичное для AIR управление памятью, 246
  - применения, 229
  - рабочие нагрузки, подходящие для инструментария, 241
  - расширяемость, 230
  - сторонние интеграции, 230
  - экосистема фреймворка Ray и за ее пределами, 250
    - дополнение или замена компонентами AIR экосистемы, 265
    - загрузка и предобработка данных, 251
    - инструментарий Ray AIR и более широкая экосистема машинного обучения, 263
    - интегрирование инструментария AIR в платформы машинного обучения, 266
    - куда отсюда двигаться дальше, 267
    - подача моделей как служб, 257
    - разработка конкретно-прикладных интеграций, 260
    - растущая экосистема, 251
    - тренировка моделей, 253
    - фреймворки распределенных вычислений на Python, 263
    - фреймворк Ray и другие системы, 262
- Ray AIR (AI Runtime), инструментарий**, зонтик для текущих библиотек науки о данных в Ray, 34
- Ray Client**, 31
- RayCluster**, 214
- Ray Clusters**, 29, 31
- Ray Core**, 32, 47
- введение, 48
  - главнейшие API-методы, 60
  - первый пример использования Ray API, 50
    - из классов в акторы, 57
    - использование функции `wait` для неблокирующих вызовов, 54
    - использование хранилища объектов с помощью `put` и `get`, 53
    - оперирование зависимостями заданий, 56
    - функции и дистанционные задания Ray, 52
  - понимание системных компонентов Ray, 61
  - головной узел, 63
  - планирование и исполнение работы на узле, 61
  - распределенное планирование и исполнение, 64
  - простой пример MapReduce, 66
    - отображение и перетасовка данных документа, 66
    - редукция количеств слов, 70
  - разработка первого распределенного приложения, 73
    - введение в обучение с подкреплением, 73
    - постановка простой задачи о лабиринте, 75
    - разработка симуляции, 80
    - сборка приложения, 87
- Ray Data**, библиотека, 34, 147
- возможность обмениваться резидентными данными по всем параллельным прогонам тренировки, 157
  - загрузка данных Snowflake, 260
  - обработка данных, 34
  - экосистема интеграций, обеспечивающая более качественную обработку данных, 161
- ray.get**, функция, 53, 60
- использование хранилища объектов, 53
- ray.init**, функция, 60
- Raylet**, планировщик, 62
- Raylet**, компонент, 61
- ray.put**, функция, 53, 60

- размещение данных в распределенном хранилище объектов, 53  
**ray.put**, функция, 60  
**ray.remote**, декоратор, 52, 60  
**ray.remote**, функция, 60  
**Ray RLlib**, 36  
 использование с библиотекой Ray Tune, 141  
 конфигурирование экспериментов, 106  
   конфигурация работника розыгрыша, 108  
 конфигурирование ресурсов, 108  
 конфигурирование сред, 108  
 краткий обзор, 94  
 начало работы, 95  
   использование RLlib Python API, 99  
   работа с RLlib CLI, 97  
   разработка среды gym, 96  
 обучение с подкреплением, 36  
 продвинутые концепции, 118  
   другие продвинутые темы, 123  
   применение процедуры усвоения учебной программы, 120  
   работа с офлайновыми данными, 122  
 работа со средами, 109  
   использование нескольких агентов, 110  
   общий обзор сред библиотеки RLlib, 109  
   сервер политик и клиенты, 115  
**Ray RLlib**, библиотека, 93  
   экосистема, 257  
**Ray Serve**, 42  
**Ray Serve**, библиотека, 188, 191  
   введение, 191  
   архитектурный обзор, 191  
   граф генерирования многомодельных предсказаний, 198  
   масштабирование и ресурсообеспечение, 195  
   определение базовой конечной точки HTTP, 193  
   пакетирование запросов, 197  
   функциональности, специально разработанные для вычислительно интенсивных моделей машинного обучения, 191  
 подача моделей в качестве служб, 42  
 развертывание опрашиваемой по HTTP службы генерирования модельных предсказаний, 238  
 разработка сквозного примера API на базе обработки ЕЯ, 202  
 доставка содержимого и предобработка, 204  
 модели обработки ЕЯ, 204  
 определение HTTP API и драйверной логики, 206  
 собираем все воедино, 208  
 экосистема, 260  
**GradioServer**, 257  
**ray start --head ...**, команда, 224  
**ray stop**, команда, 214  
**Ray Train**, 40  
**Ray Train**, библиотека, 149, 167  
   введение, 169  
   компоненты библиотеки в базовых конвейерах машинного обучения, 169  
   пример предсказания больших чаевых в поездках в нью-йоркском такси, 170  
   распределенная тренировка с помощью библиотеки Train, 173  
   распределенное пакетное генерирование модельных предсказаний, 176  
   загрузка, предобработка и выделение признаков, 171  
   использование для горизонтального масштабирования модели JAX, 260  
   поддержка фреймворков градиентно-бустированных деревьев решений, 177  
   стандартная модель PyTorch и цикл тренировки, 251  
   тренеры, 177  
     выполнение миграции в библиотеку Ray Train с минимальными изменениями в исходном коде, 179  
     горизонтальное масштабирование тренеров, 180  
     интеграция тренеров с библиотекой Ray Tune, 183  
     использование обратных вызовов для мониторинга тренировки, 185  
     предобработка с помощью библиотеки Ray Train, 181  
     экосистема, 257  
**Ray Tune**, 40  
   введение, 130  
   гиперпараметрическая настройка, 40  
   интеграция тренеров (Ray Train), 183  
   конфигурирование и выполнение, 136  
     детализация ресурсов, 136

конкретно-прикладные и условные пространства поиска, 140  
 контрольные точки, остановка и возобновление, 139  
 обратные вызовы и метрики, 137  
 машинное обучение, 141  
     использование библиотеки RLlib вместе с библиотекой Tune, 141  
     настройка моделей Keras, 142  
 поддержка алгоритмов из известных инструментов гиперпараметрической оптимизации, 45  
 применение библиотекой RLlib, 99  
 принципы работы, 131  
     алгоритм поиска, 133  
     интеграция с другими фреймворками гиперпараметрической оптимизации, 133  
     общий обзор компонентов, 131  
     планировщики, 134  
 Ray Tune, библиотека, 125, 149  
     интеграция тренеров Ray AIR, 235  
     поставляемый с ней класс MLFlowLogger, 251  
     косистема, 257  
 ray.wait, функция, 60  
 Ray Workflows, библиотека, 44, 268  
 read\_csv, служебная функция, 232  
 Redis, использование дистанционного экземпляра, 212  
 ReLU (Узел выпрямленный линейный), активационная функция, 144  
 remote, функция, 60  
 repeat, функция (Dataset), 160  
 requests, библиотека, 43  
 requests, пакет, 191  
     использование для тестирования классификатора настроений, 194  
 resources, метод (AlgorithmConfig), 107  
 ResultGrid (Tuner API), 132  
 RISELab (Калифорнийский университет в Беркли), 26  
 rllib, инструмент командной строки, 37  
 rllib evaluate, команда, 98  
 RLlib Python API, 99  
     доступ к политике и статусу модели, 103  
     сохранение, загрузка и оценивание моделей библиотеки RLlib, 101  
     тренировка алгоритмов библиотеки RLlib, 99  
 rllib train, команда, 98

rollouts, метод (AlgorithmConfig), 107  
 RunConfig, 184, 235  
 Rust, компилятор, установка, 43

## S

ScaleStrategy, 215  
 ScalingConfig, 175  
     адаптирование памяти, 247  
     определение для XGBoostTrainer в Ray AIR, 234  
     тренеры в библиотеке Ray Train, 175, 180  
     тренеры Ray AIR, 234  
     указание параметров узлов кластеров, 180  
 scikit-learn, 254  
     локальная установка, 159  
     SGDClassifier, алгоритм, 159  
 serve.run, 194  
 serve run, команда CLI, 194, 208  
 serve run app, scaled\_deployment, команда, 196  
 SGDClassifier, алгоритм, 159  
     обертка TrainingWorker, 160  
 Simulation, класс, 82  
     конвертирование в актора Ray, 87  
     реализация, 82  
 SimulationActor, экземпляр, 87  
 Snowflake, 260  
 Spark, 44, 147, 263  
 StandardScaler, 183, 233  
 state\_action\_table (пример политики), 82  
     обновление значений, 83  
 step, метод, 78  
 sum, агрегация, 152

## T

TaskSettableEnv, класс, 120  
 TensorBoard, 185  
 TensorFlow, 94, 104  
     поддержка библиотек Ray RLlib и Trains, 45  
     установка, 37  
 TensorFlow Keras, модель (последовательная), 236  
 TorchTrainer, 174, 175  
     обертывание в настройщика, 251  
     создание экземпляра и работа с ним, 180  
 torchvision, расширение, 252  
 TPE (Древесно-структурированный парценовский оценщик), поисковик, 133

train, команда (rllib), 98  
 training, метод (AlgorithmConfig), 107  
 training\_loop, 178  
 TrainingWorker, 159  
 train\_loop\_config, 179  
 train\_loop\_per\_worker, функция, 179  
 train\_one\_epoch, вспомогательная функция, 178  
 transformers, пакет, 191  
 Tune. Ray Tune  
 tune, объект, 41  
 TuneConfig, 235  
 tune\_objective, функция, 130  
 Tuner, API, 184  
 tune.report, функция, 137  
 TuneReportCallback, объект, 144  
 tune.run, функция, 41, 130, 132  
 передача аргументов библиотеки RLlib, 141

**U**

use\_gpu, флаг, 180

**W**

wait, функция, использование для неблокирующих вызовов, 54  
 weight, параметр, 126, 135  
 wikipedia, пакет на PyPI, 204  
 window, функция, 36

**X**

XGBoost, библиотека, 177  
 XGBoostPredictor, 237  
 XGBoostTrainer, определение для Ray AIR, 234  
 обертывание экземпляра настройщиком, 236  
 XGBoostTrainer, класс, 177  
 создание и указание диапазонов гиперпараметров, 183

**A**

Абстракция  
 абстракции, предоставляемые фреймворком Ray, 27, 28  
 закон негерметичных абстракций, 25  
 Автомасштабирование  
 кластер, 227  
 поддержка библиотекой Ray Clusters, 28

рабочих нагрузок AIR, 248  
 реплик Ray Serve, 196  
 Автомобиль самоуправляемый, 188  
 Агент (обучение с подкреплением), 90, 94  
 несколько агентов, 91  
 работа с несколькими агентами, 110  
 соотнесенность агентов с политиками, 111, 114  
 Агрегация, поддержка наборами данных Dataset фреймворка Ray, 149  
 Актор, 57  
 актор-контроллер для развертываний Ray Serve, 192  
 в развертывании Ray Serve, 192  
 доступ к данным из заданий без отслеживания внутреннего состояния и с отслеживанием внутреннего состояния в составных рабочих нагрузках, 247  
 использование для преобразования с отслеживанием внутреннего состояния, 245  
 использование наборов данных Dataset фреймворка Ray в распределенном пакетном генерировании модельных предсказаний, 176  
 использование Ray AIR в сочетании с заданиями для продвинутых составных рабочих нагрузок, 243  
 конвертирование класса Simulation в актора, 87  
 отображение данных с помощью акторов и поддержка наборов данных Dataset, 154  
 шаблоны и антишаблоны Ray, 72  
 Актор-контроллер, 192  
 Алгоритм поиска, 129, 132, 133  
 поддержка библиотекой Tune, 132, 133  
 Алгоритм упаковки в корзины (binpacking), 228  
 Анализ (Tune), 128, 131  
 получение результатов при тренировке модели Keras, 144  
 Ансамблирование, 201  
 Архитектура (Ray)  
 краткий обзор компонентов, 65  
 техническая документация, 62

**Б**

Балл (гиперпараметрическая оптимизация), 128

возвращение в качестве словаря, 130  
 промежуточные баллы, 134  
 Балл промежуточный, 134  
 Библиотека (Ray), 28  
     библиотека науки о данных, 32  
     особые библиотеки для этапов машинного обучения, 32  
     тренировка моделей, 36  
 Блок  
     в наборах данных Dataset, 149  
     параметр blocks\_per\_window, 157  
     реорганизация, 152  
 Блокировка, 54  
     операции наборов данных Dataset, 155  
 Блокировка интерпретатора глобальная (GIL), 51  
 Блокнот Jupyter, 30  
 Бэкенд, 45

**В**

Вероятность выполнения действий, 103  
 Вероятность перехода из состояния в состояние, 91  
 Вес  
     использование функции взятия образцов random.uniform из numpy, 140  
     получение для моделей библиотеки RLLib в Python API, 103  
 Включение в список, 50  
 Владение, 62  
     в сравнении с зависимостями, 63  
 Вознаграждение (обучение с подкреплением), 90  
     в многоагентной среде, 113  
 Возобновление прогонов Ray Tune, 139  
 Восстановление линии наследования, 247  
 Восстановление после сбоя, 28  
 Выбор действия жадный, 103  
 Выделение признаков  
     в распределенном пакетном генерировании модельных предсказаний, 176  
     построение признаков с использованием функции load\_dataset в библиотеке Ray Train, 171  
 Вызов неблокирующий, использование функции Ray wait, 54  
 Вычисление  
     без отслеживания внутреннего состояния, 241  
     векторизованное, 153, 189, 197

генерирование модельных предсказаний, 154  
 использование встроенного в библиотеку Serve API пакетирования, 205  
 на наборах данных Dataset, 153  
 распределенное  
     возможности инструментария Ray Core, 48  
     трудности, 26  
     фреймворк Ray, 29  
     Ray API как универсальный интерфейс, 49  
 с отслеживанием внутреннего состояния, 58, 241  
     опора на отказоустойчивость на основе контрольных точек, 247  
 Вычисления облачные, 211  
     работа с облачными кластерами, 225  
     другие облачные провайдеры, 226  
     AWS, 225

**Г**

Генератор случайных чисел, 202  
 Генерирование модельных предсказаний, 154  
     векторизованное, 154, 197  
     поддержка конвейера Hugging Face, 197  
     граф генерирования многомодельных предсказаний, 198  
 Генерирование модельных предсказаний онлайновое, 187, 242  
     варианты использования, 187  
     конвейер в примере API на базе обработки ЕЯ, 202  
     разница в подаче моделей машинного обучения, 189  
     вычислительно интенсивные модели машинного обучения, 189  
     модели машинного обучения бесполезны в изоляции, 190  
     разработка служб с помощью библиотеки Ray Serve, 191. См. *Ray Serve, библиотека*  
 Генерирование модельных предсказаний пакетное, 241  
     использование наборов данных Dataset, 154  
     пример в документации Ray, 154

- распределенное генерирование с использованием библиотеки Ray Train, 176
- Генерирование модельных предсказаний распределенное пакетное (Ray Train), 176
- Гиперпараметр, 126
- зависящий от других гиперпараметров, 140
  - указание диапазонов для XGBoostTrainer, 183
  - TrainingWorker, 159
- Голова (в глубоком обучении), 105
- Голова функции ценности, 105
- Граф генерирования многомодельных предсказаний, 198
- в примере API на базе обработки ЕЯ, 202
  - привязка нескольких развертываний как ключевая особенность библиотеки Ray Serve, 199
  - шаблон конвейеризации, 200
  - шаблон условной логики, 201
  - шаблон широковещательной трансляции, 201
- Граф заданий, 163
- ## Д
- Данные офлайновые
- работа с ними, 122
  - Python API в библиотеке RLlib, 124
- Действие (обучение с подкреплением), 90
- вероятности выполнения каждого действия, 103
  - вычисление в Python RLlib API, 103
  - передаваемое в steps в многоагентной среде, 112
  - упрощающие допущения, 91
- Дзен языка Python, 68
- Документация по API тренировки в рамках библиотеки RLlib, 109
- Доставка содержимого и предобработка, 204
- Драйвер, 29, 64
- ## Ж
- Журналирование
- использование Neptune, 261
  - конфигурирование для KubeRay, 222
  - результатов попыток в MLFlow, 251
- ## 3
- Завершенность
- в обучении с подкреплением, 90
  - вспомогательная функция `is_done` для работы с несколькими агентами, 112
  - многоагентной среде, 113
  - условие завершения работы в среде gym.Env, 96
  - состояние, когда игра считается оконченной, 77
- Зависимости, динамическое исполнение заданий, 27
- Зависимость
- в сравнении с владением, 63
  - оперирование ими для заданий, 56
  - урегулирование планировщиком Raylet, 62
  - урегулирования для заданий, 64
  - установка зависимостей для Ray, 30
- Загрузка данных, 251
- в распределенном пакетном генерировании модельных предсказаний, 176
  - загрузка модели в библиотеку Ray Train, 171
  - тренировочные и валидационные данные для работников тренировки, 175
- Задание, 29
- видоизменение существующего задания путем включения актора, 59
  - данные из заданий без отслеживания внутреннего состояния, получаемые акторами с отслеживанием внутреннего состояния в составных рабочих нагрузках, 247
  - исполнение, 65
  - использование Ray AIR с акторами для продвинутых составных рабочих нагрузок, 243
  - оперирование зависимостями заданий, 56
  - установка трудности, 121
  - шаблоны и антишаблоны Ray, 72
- Задание Ray дистанционное и функции, 52
- Задание Ray AIR без отслеживания внутреннего состояния, 242
- Задача о лабиринте, постановка, 75
- Задержка
- онлайновое генерирование модельных предсказаний, 187, 188

онлайновые сервисы и приложения машинного обучения, 188

**Закон**  
Мура, 25  
негерметичных абстракций, 25  
третий Кларка, 57

Заявка на выполнение работы, 29

Значение состояния–действия (обучение с подкреплением), 90, 103, 105, 126  
функции состояния–действие, 105

**И**

Иерархия агентов, 110

**ИИ.** См. Интеллект искусственный

ИИ и вычисления, 25

Инструмент обработки больших данных, 241

Интеграция  
конкретно-прикладная, 260  
обзор интеграций с фреймворком Ray, 262

Интеграция с Hyperopt и Optuna (Ray Tune), 133

Интеллект искусственный (ИИ)  
недавние достижения, 25  
типы рабочих нагрузок ИИ, которые инструментарий AIR позволяет кластерам Ray выполнять, 244  
фокусировка инструментария Ray AIR на вычислениях, связанных с искусственным интеллектом, 263

Исполнение асинхронное, 53  
выполнение зависимых заданий асинхронно и параллельно, 58

Исполнение без отслеживания внутреннего состояния, 244

Исполнение заданий по онлайновому генерируанию предсказаний, 246

Исполнение конвейеризированное, 247

Исполнение параллельное зависимых заданий, 58

Исполнение рабочих нагрузок (Ray AIR), 244  
исполнение без отслеживания внутреннего состояния, 244  
исполнение запросов на онлайновое генерируание предсказаний, 246  
исполнение составных рабочих нагрузок, 245  
исполнение с отслеживанием внутреннего состояния, 245

Исполнение синхронное  
операции наборов данных Dataset, 155  
подход алгоритмов распределенной тренировки, 158

Исполнение с отслеживанием внутреннего состояния, 245

Иследователь данных, 23  
применения инструментария Ray AIR, 230

**К**

Класс  
задания и акторы, работающие как распределенные версии, 48  
конвертирование классов Python в акторы, 58

Классификатор, параллельная тренировка копий, 157

Кластер локальный, 30

Кластер (Ray), 29, 211, 266. См. *Ray Clusters*  
базовые компоненты, 29  
запуск локального кластера, 48  
определение конфигурации ScalingConfig, 180  
процессы головного узла для управления кластером, 64

Кластер Ray  
использование инструмента запуска кластеров Ray, 223  
взаимодействие с кластером Ray, 225  
конфигурирование кластера Ray, 223  
CLI, 223  
работа с облачными кластерами, 225  
AWS, 225  
Azure, 226  
Google Cloud, 226  
развертывание в Kubernetes, 214  
взаимодействие с кластером KubeRay, 216  
конфигурирование журналирования для KubeRay, 222  
конфигурирование оператора KubeRay, 219  
настройка первого кластера KubeRay, 215  
создание в ручном режиме, 212  
типы рабочих нагрузок ИИ, которые AIR обеспечивает, 244

Клиент  
клиент Ray, 218  
определение клиента политики в среде библиотеки RLLib, 117

- PolicyClient, 115  
**Клиент Ray**, 31  
 использование для соединения с кластером KubeRay, 218  
**Клонирование поведения**, 123  
**Конвейер**  
 онлайновое генерирование модельных предсказаний, 203  
 разработка конвейера машинного обучения с использованием наборов данных Dataset фреймворка Ray, 164  
 Dataset, 155  
 DatasetPipeline, 36  
**Конвейер видеоОбработки**, 188  
**Конвейеризация**, 148  
 использование на наборах данных Dataset фреймворка Ray, 245  
 многомодельный шаблон в приложении машинного обучения, 200  
**Конвейер Hugging Face с поддержкой векторизованного генерирования модельных предсказаний**, 197  
**Конструирование признаков**, 33  
**Контейнер**, 218, 222  
 операции KubeRay, 214  
 переменные среды, 222  
 указание ресурсов, 222  
**Кортеж**, 57
- Л**  
**Логика условная**, 201
- М**  
**Масштабирование**  
 в библиотеке Ray Serve, 195  
 горизонтальное масштабирование тренеров в библиотеке Ray Train, 180  
 наборы данных Dataset фреймворка Ray, 149  
 службы онлайнового генерирования модельных предсказаний, 190  
 Ray, 28  
**Матрица совместимости планировщиков Tune**, 135  
**Метрика**  
 конфигурирование для информирования в Ray Tune, 137  
 оптимизация в BayesOptSearch, 134  
 передача в планировщик Ray Tune, 135  
 передача метрик библиотеки RLLib в библиотеку Ray Tune, 141
- получение наилучших гиперпараметров, 131  
**МО. См. Обучение машинное**  
**Модель**  
 адаптирование под эксперименты с RLLib, 105  
 в глубоком Q-обучении, используемая в DQN, 104  
 в контрольных точках Checkpoints библиотеки Ray Train, 176  
 доступ к состоянию в Python RLLib API, 103  
 контрольные точки как нативный для Ray AIR обмен моделями, 236  
 настройка моделей Keras в библиотеке Ray Tune, 142  
 обработки ЕЯ, 204  
 определение модели глубокого обучения, 172  
 распознавания сущностей, 205  
 распределенная тренировка моделей машинного обучения, 158  
 резюмирования текста, 205  
 таблица state\_action\_table политики, 90  
**Модель классификатора настроений**, 193, 205  
 видоизменение под пакетирование на стороне сервера, 197  
 горизонтальное масштабирование на несколько реплик и корректировка ресурсообеспечения, 195  
 использование в примере API на базе обработки ЕЯ, 205  
 тестирование с использованием пакета requests, 194  
**Модель Keras последовательная в рамках TensorFlow**, 236  
**Модуль**  
 взаимодействие с головным модулем кластера KubeRay, 216  
 головной, 215  
 головной модуль и модули-работники в кластере KubeRay, 215  
 конфигурирование кластера Ray в операторе KubeRay, 221  
**Модуль-работник**, 215
- Н**  
**Наблюдаемость**, 268  
**Наблюдение (в обучении с подкреплением)**  
 в многоагентной среде, 113

- вычисление действий для указанных наблюдений, 103  
 преобразование в ожидаемую моделью форму, 106  
 принимаемые моделью DQN, 104
- Н**  
**Н**аблюдения (в обучении с подкреплением), 77  
 Набор данных Dataset фреймворка Ray, 147, 148  
 Нагрузка полезная (служба предсказания), 238  
 Нагрузка рабочая без отслеживания внутреннего состояния, автомасштабирование, 248  
 Нагрузка рабочая составная, 242  
     доступ к данным из заданий без отслеживания внутреннего состояния и с отслеживанием внутреннего состояния, 247  
     исполнение, 245  
     использование инструментарием Ray AIR акторов и заданий, 243  
     стратегии отказоустойчивости, 248  
 Нагрузка рабочая с отслеживанием внутреннего состояния, автомасштабирование, 248  
 Настройка гиперпараметрическая, 33  
     с использованием Ray Tune, 40  
 Настройщик, 235  
     работа настройщиков AIR с тренерами AIR, 236  
 Наука о данных, 32  
     Ray AIR и рабочий процесс науки о данных, 32  
 Неспособность подключиться к GCS, 213
- О**
- Обработка данных, 33  
 Обработка естественного языка (ЕЯ), 202  
 Обработка изображений, 200  
 Образ контейнера, 222  
 Обучение глубокое  
     определение модели, 172  
     тренировка, 154  
 Обучение машинное (МО)  
     взаимодействие онлайнового генерирования модельных предсказаний с моделью машинного обучения, 187  
     вычислительная интенсивность моделей, 189
- гибкий распределенный Python, 24  
 инструментарий Ray AIR и более широкая экосистема машинного обучения, 263  
 инструментарий Ray AIR как зонтик для всех других библиотек машинного обучения в Ray, 230  
 интегрирование инструментария Ray AIR в платформы машинного обучения, 266  
 использование библиотеки Ray Tune, 141  
     настройка моделей Keras, 142  
     использование библиотеки RLLib вместе с библиотекой Tune, 141  
     используемые процессы науки о данных, 32  
     компоненты библиотеки Ray Train в базовых конвейерах, 169  
     модели бесполезны в изоляции, 190  
     недавние достижения, 25  
     несколько моделей в примере API на базе обработки ЕЯ, 202  
     обучение с подкреплением, 73  
     перекос между тренировкой модели и ее подачей в качестве службы в развертываниях, 182  
     применения инструментария Ray AIR инженерами машинного обучения, 230  
     производительность и наборы данных Dataset фреймворка Ray, 148  
     разработка конвейера машинного обучения с использованием наборов данных Dataset фреймворка Ray, 164  
     распределенная тренировка моделей машинного обучения, 158  
     урегулирование рабочих нагрузок с помощью прогона  
     одного-единственного скрипта  
     одной-единственной системой, 231  
     условная логика для потока управления, 201  
     широковещательная параллельная трансляция в несколько моделей, 201  
 Обучение с подкреплением. См. *Ray RLLib*  
     алгоритм Q-обучения, 86  
     с использованием Ray RLLib, 36  
     терминология, 90  
     тренировка модели, 83  
 Обучение с подкреплением многоагентное (MAR), тренировка задачи, 114

- Объединение, операция, 151
- Объект
- потеря и восстановление в наборах данных Dataset фреймворка Ray, 149
  - распределенной передачей объектов, 64
  - эквивалентность с заданиями и акторами в Ray Core, 59
- Объект тренируемый, 131
- тренеры RLLib, передаваемые как аргументы в функцию tune.run, 141
- Окно, 36, 157
- Определение модели глубокого обучения, 172
- Оптимизация байесова, 133
- Оптимизация гиперпараметрическая (HPO), 40, 125
- введение в библиотеку Ray Tune, 130
  - интеграция библиотеки Ray Train с библиотекой Ray Tune, 183
  - интеграция тренеров Ray AIR с библиоткой Ray Tune, 235
  - инструменты, поддержка алгоритмов библиотекой Ray Tune, 45
  - машинное обучение с помощью библиотеки Ray Tune, 141
  - настройка гиперпараметров, 126
  - разработка примера случайного поиска с помощью фреймворка Ray, 126
  - трудности, 129
- Оптимизация политики близости расположения (Proximal Policy Optimization, PPO), алгоритм, 38
- Опыт (в обучении с подкреплением), 90
- Организация блоков, 152, 156
- и реорганизация в наборах данных Datasets, 152
- Остановка
- остановка прогона Ray Tune, 139
  - остановка процессов Ray в узле, 214
- Отбор гиперпараметров случайный, 126
- Ответ (HTTP), определение схемы, 206
- Отказоустойчивость и владение, 62
- Отображение
- отображение модели на весь набор данных, 176
  - поддержка наборов данных Dataset с использованием акторов Ray, 154
- Отображение
- данных, 149
  - пакетов, 252
- Отрисовка среды обучения с подкреплением, 96, 109
- видоизменение в многоагентной среде, 113
- Отчетность с использованием сеанса Ray AIR, 173
- Оценивание времени прибытия, 188
- Оценивание моделей библиотеки RLLib в Python API, 101
- ## П
- Пакетирование
- запросов, 190, 197
  - на стороне клиента, 197
  - на стороне сервера, 197
- Память
- детализация для попыток в Ray Tune, 137
  - распределенная, 64
  - специфичное для AIR управление памятью, 246
  - хранилище объектов Raylet
  - управление совместным пулом памяти, 61
- эффективное потребление наборами данных Dataset фреймворка Ray, 149
- Пара ключ-значение, производимая в фазе отображения алгоритма MapReduce, 67
- Параллелизация
- параллелизация исходного кода с помощью библиотеки Ray Train, 169
  - тренировка с параллелизмом данных, 173
- Параллелизм данных, 168
- движки Spark и Dask, 44
  - интеграции внешних библиотек с наборами данных Dataset фреймворка Ray, 161
  - обработка данных, 27, 35. См. *Ray Data*; *Ray Dataset*
  - применение библиотеки Ray Data, 34
- Параллелизм модельный, 168
- Параметр непрерывный, 126
- Передача объектов распределенная, 64
- Перекос между тренировкой модели и ее подачей в качестве службы, 182
- Переменная категориальная, 142
- Перетасовка данных, 67, 69
- Планирование и исполнение работы на узле, 61
- распределенное планирование
  - и исполнение, 64
- Планировщик, 132

- планировщик Dask в пакете с Ray, 163  
 Ray Tune, 132, 134  
     комбинирование с алгоритмом поиска, 135
- Планировщик заданий, 61  
 Планировщики попыток, 134  
 Планировщик распределенный, 64  
 Платформа машинного обучения, 266  
     интегрирование инструментария Ray AIR, 266  
     применение фреймворка Ray и инструментария AIR внутри размещенных платформ, 265
- Подача моделей библиотеки RLLib в качестве служб, практическое руководство, 94  
 Подача моделей в качестве служб, 33, 257  
     использование библиотеки Ray Serve, 42
- Подача модели как службы с использованием Gradio, 251
- Подгонка предобработчика AIR к тренировочным данным, 232
- Поддержка типов рабочих нагрузок Spark, Dask и Ray, 263
- Поиск в параметрической решетке, 42, 133
- Поисковик, 133. См. *Алгоритм поисковый*
- Политика (обучение с подкреплением), 90  
     в многоагентной среде  
         соотнесенность агентов с политиками, 114  
     в многоагентных средах обучения с подкреплением, 111  
     доступ к состоянию политик в Python RLLib API, 103  
     работа с сервером политик и клиентами в среде библиотеки RLLib, 115  
         определение клиента, 117  
         определение сервера, 115
- Попытка (гиперпараметрическая оптимизация), 129, 131  
     детализация ресурсов в Ray Tune, 136
- Попытка конкурентная (Ray Tune), 136
- Потеря, 175
- Предобработка  
     в распределенном пакетном генерировании модельных предсказаний, 176  
     использование библиотеки Ray Train, 181
- использование функции `load_dataset` в библиотеке Ray Train, 171  
 содержимое в примере API на базе обработки ЕЯ, 204
- Предобработка данных, 251  
 Предобработчик  
     встроенные в библиотеку Ray Train предобработчики, 181  
     выбор предобработчика для интеграции библиотеки Ray Train с библиотекой Ray Tune, 183  
     Ray AIR, 232  
         в контрольных точках Checkpoint (Ray Train), 181  
         детализация для XGBoostTrainer, 234  
         разные типы, 233
- Предобработчики, предоставляемые библиотекой Ray Train, 169
- Предсказатели, 169  
 Предсказатель, 258, 259  
     пакетный, 237, 241
- Преобразование  
     выполнения конкретно-прикладного преобразования на наборах данных Dataset, 153  
     набора данных с использованием наборов данных Dataset, 35  
     наборы данных Dataset  
         встроенные преобразования, 151  
         на данных Parquet, 155  
     наборы данных Dataset фреймворка Ray, 149  
     столбцов в формат, используемый как признаки в модели машинного обучения, 172, 181
- Привязка нескольких развертываний, 199
- Пример отрегулированный, 38  
 Пример поиска (случайного)  
     разработка с помощью фреймворка Ray, 126
- Принцип внутреннего устройства (Ray), 26
- Прогон (Tune), 132, 134
- Программирование функциональное с наборами данных Dataset, 35
- Производительность, измерение задания Ray, 52
- Пространство действий дискретное (обучение с подкреплением), 91
- Пространство действий непрерывное (обучение с подкреплением), 91

- Пространство действий (обучение с подкреплением), 91, 112  
 определение сервера политик в среде библиотеки RLlib, 116  
 параметрическое пространство действий в библиотеке RLlib, 124
- Пространство действий параметрическое, 124
- Пространство наблюдений (обучение с подкреплением), 91, 112  
 в среде gym, 96  
 определение сервера политик в среде библиотеки RLlib, 116
- Пространство поиска, 129, 130, 131  
 конкретно-прикладное и условное, 140  
 определение с помощью библиотеки random, 127  
 определение с помощью функции tune.uniform, 130
- Процессор графический  
 выделение двух на реплику в библиотеке Ray Serve, 196  
 затраты на сервисы онлайнового генерирования модельных предсказаний, 189  
 указание для машин в кластере Ray, 213  
 флаг use\_gpu, 180
- Процессор центральный  
 выделение двух на реплику в библиотеке Ray Serve, 196  
 указание для машин в кластере Ray, 213  
 функция cpu\_intensive\_preprocessing, 155
- Процесс-работник, 29  
 отказоустойчивость и владение, 62  
 сдача в аренду владельцам заданий, 65
- Пул памяти совместный, 61
- P**
- Работа как коллекция заданий, 29
- Работник  
 библиотека RLlib, получение политики и модельных весов, 104  
 загрузка тренировочных и валидационных данных для работников тренировки, 175  
 конфигурация работника розыгрыша под эксперимент с RLlib, 108  
 определение TrainingWorker для тренировки копии классификатора, 159  
 розыгрыш (rollout), 100
- Развертывание  
 в библиотеке Ray Serve  
 настройка реплик и ресурсообеспечение, 195  
 привязка нескольких развертываний, 199  
 в инструментарии Ray AIR, 238  
 в Ray AIR  
 с помощью PredictorDeployment, 238  
 в Ray Serve, 192  
 кластеры Ray, 211  
 кластеры Ray в Kubernetes, 214
- Развертывание драйвера, определение логики потока управления, 206, 209
- Развертывание драйверное, 199
- Распределение действий, 106  
 статистическое, 103
- Режим  
 детализация для попыток в Ray Tune, 138  
 передача в планировщик Ray Tune, 138  
 получение наилучших гиперпараметров, 131
- Режим высокой доступности (GCS), 248
- Результат  
 запись результата тренировки в каталог, 98  
 состояние и результаты тренировки алгоритма RLlib DQN, 100
- Результат оценивания натренированного алгоритма обучения с подкреплением, 39
- Результат JSON-сериализуемый (HTTP-запрос), 193
- Реорганизация блоков, 152
- Реплика (Ray Serve)  
 выполнение сервера FastAPI в каждой реплике, 194  
 настройка для модели резюмирования текста, 205  
 настройка для развертывания, 196  
 создание экземпляра реплики развертывания, 194
- Ресурс  
 более выразительные политики, 196  
 выделение в библиотеке Ray Serve, 195  
 вычислительные ресурсы, используемые тренером, 181  
 детализация для попыток Ray Tune, 136  
 конфигурирование для кластера KubeRay, 222

- конфигурирование под эксперименты с RLLib, 108  
 обзор применения библиотекой Ray Cluster, 49  
 Ресурс дробный, 136, 196  
 Робототехника, 188  
 Розыгрыш (в обучении с подкреплением), 90  
     законченные розыгрыши для обновления политики, 88  
     конфигурация работника розыгрыша под эксперимент с RLLib, 108  
     метод rollouts интерфейса Python RLLib API, 100
- C**
- Самообучение имитационное, 123  
 Сбой, принятая в Ray AIR модель сбоя, 247  
 Сброс в начальное состояние (среды обучения с подкреплением), 96  
 Связь в Ray Clusters, 64  
 Сдача в аренду процессов-работников владельцам заданий, 65  
 Сеанс  
     Ray, 175  
     Ray AIR, 173  
 Сегмент данных, 158  
     прокручивание в цикле с помощью `iter_torch_batches`, 175  
     служебная функция `get_data_shard`, 174  
 Сегментирование данных, 158, 255, 256  
     служебная функция `get_data_shard`, 174  
 Сервер подачи заявок Ray на выполнение работы, 217, 219, 225  
 Сервер GCS (Ray)  
     нет соединения с GCS, 213  
     распечатка IP-адреса, 213  
     сбои, 248  
 Сеть нейронная, 103  
     в глубоких Q-сетях (DQN), 86  
     определение и тренировка в библиотеке Ray Train, 170  
     параллелизация вычислений с целью ускорения тренировки, 168  
     сеть FarePredictor в терминах PyTorch, 172  
 Симуляция, невозможность точно симулировать некоторые физические системы, 91  
 Система обработки реляционных данных, интеграция с Ray, 161
- Система рекомендательная, 187  
 сложности на периферии, 190  
 Система, связанная с Ray, 65  
 Скорость (Ray), 28  
 Словарь  
     словарь config для передачи тренеру в качестве `train_loop_config`, 179  
     словарь datasets в тренерах, 176  
     создание из словаря Python со схемой, 153  
 Слой  
     в Ray, 28  
     пакетной нормализации, 172  
     стержневой (Ray), 28. См. *Ray Core*  
 Служба глобального управления (GCS), 64, 65  
 Смещение, 135, 140  
 Сортировка данных, 149  
 Сортировка, операция, наборы данных Dataset фреймворка Ray, 151  
 Состояние  
     доступ к состоянию для модели и политики в Python RLLib API, 103  
     среды в обучении с подкреплением, 90  
 Состояние мягкое, 241  
 Состояние (обучение с подкреплением), 90  
 Сохранение алгоритмов библиотеки RLLib в Python API, 101  
 Справочник по API для алгоритмов библиотеки RLLib, 107  
 Спуск градиентный стохастический, 159  
 Среда (в обучении с подкреплением), 90  
     детерминированная среда, 91  
     использование среды gym в рамках библиотеки RLLib, 97  
     конфигурирование сред под эксперименты с RLLib, 108  
     описание для DQNConfig с использованием Python RLLib API, 99  
     работа со средами библиотеки RLLib, 109  
     использование нескольких агентов, 110  
     общий обзор сред библиотеки RLLib, 109  
     сервер политик и клиенты, 115  
     разработка среды gym, 96  
 Среда детерминированная (в обучении с подкреплением), 91  
 Страница алгоритмов библиотеки RLLib (документация фреймворка Ray), 95

**Схема**, 152  
определение для HTTP-ответов, 206

**T**

Такси нью-йоркское, предсказание больших чаевых в поездках (пример), 170  
Талица владения, 62  
Точка конечная HTTP как обертка модели машинного обучения, 193  
Точка контрольная  
    вычисления с отслеживанием внутреннего состояния с опорой на отказоустойчивость на основе контрольных точек, 247  
    генерируемая тренерами или настройщиками Ray AIR, 236  
    информирование о модельной контрольной точке, 175  
    класс Checkpoint в библиотеке Ray Train, 169  
    оценивание натренированного алгоритма из нее, 39  
    свойство checkpoint тренеров (Ray Train), 177  
    создание библиотекой Ray Tune, 139  
    создание библиотекой Ray Tune для библиотеки RLLib, 98  
    создание командой rllib, 98  
    создание контрольных точек алгоритма обучения с подкреплением, 101  
    создание специфичных для фреймворков моделей из существующих контрольных точек, 236  
    экспортирование натренированной модели в качестве контрольной точки библиотеки Ray Train, 176  
Ray AIR  
    настройщики и контрольные точки, 235  
    развертывание PredictorDeployment, 238  
    создание пакетного предсказателя, 237  
Точка монтировки тома, 222  
Трансляция сетевых адресов (NAT), 213  
Трансляция широковещательная, 201  
    в примере API на базе обработки ЕЯ, 203  
Тренер, 169  
    библиотека Ray Train  
        словарь datasets, 176  
        ScalingConfig, 180

детализация для Ray AIR, 233  
подробнее о тренерах в библиотеке Ray Train, 177  
    выполнение миграции в библиотеку Ray Train, 179  
    горизонтальное масштабирование тренеров, 180  
    интеграция тренеров с библиотекой Ray Tune, 183  
    использование обратных вызовов для мониторинга тренировки, 185  
    классы-тренеры в Ray Train с общим интерфейсом, 177  
    предобработка с помощью библиотеки Ray Train, 181  
Ray AIR  
    генерирование контрольных точек, 236  
    работа с настройщиками, 235  
TorchTrainer, 175  
Тренировка алгоритмов  
    библиотеки RLLib, 99  
        в многоагентной среде, 114  
    впечатляющий спектр в библиотеке RLLib, 124  
Тренировка моделей, 33, 253  
    библиотеки Ray, 36  
        Ray RLLib, 36  
        основы распределенной тренировки моделей, 168  
        параллелизация с помощью Ray, 89  
        пример параллельной тренировки копий классификатора с использованием наборов данных Dataset, 157  
        тренировка модели подкрепления, 83  
Тренировка моделей распределенная, 242  
    основы, 168  
    тренировка модели машинного обучения с использованием наборов данных Dataset фреймворка Ray, 157  
Тренировка на основе больших данных, 242  
Тренировка с параллелизмом данных, 173  
Трудность, 120  
    установки трудности задания, 121

**У**

Узел  
    нет доступа к указанному порту и IP-адресу, 213

остановка процессов Ray, 214  
 подключение к головному узлу в кластере Ray, 212  
 проверка на возможность достичь каждого порта из узла, 213  
**Узел выпрямленный линейный (ReLU),**  
 активационная функция, 144  
**Узел головной,** 29, 63, 212  
 адрес с трансляцией сетевого адреса, 213  
 подключение ко всем остальным узлам в кластере Ray, 212  
**Узел-работник,** 29  
 как вместилище системных компонентов, 63  
 планирование и исполнение работы, 61. См. *Raylet*  
**Узел скрытый,** 144  
**Указатель на объект,** 57  
 возврат дистанционных заданий Ray, 53  
**Управление ресурсами**  
 головным узлом, 65  
 планировщиком Raylet, 62  
**Усвоение учебной программы,**  
 применение с помощью библиотеки RLLib, 120  
**Усталость от миграций,** 231  
**Установка Ray,** 30

## Ф

**Фаза**  
 отображения (MapReduce), 67  
 перетасовки (MapReduce), 67  
 редукции (MapReduce), 67  
 редукция количеств слов, 70  
**Философия внутреннего устройства,** 231  
**Философия AIR,** 265  
**Фильтрация данных,** 149  
 операции фильтрации в наборах данных Dataset фреймворка Ray, 151  
**Формат данных**  
 гибкость внутри наборов данных Dataset, 152  
 форматы сериализации, поддерживаемые наборами данных Dataset, 151  
**Формат сериализации,** 151  
**Формат столбчатый (Arrow),** 153  
**Фреймворк**  
 сторонние фреймворки тренировки, 169

указание фреймворка для алгоритмов библиотеки RLLib, 104  
**Фреймворк глубокого обучения**  
 работа с библиотекой RLLib, 94  
**Фреймворк градиентно-бустированных деревьев решений,** 177  
**Фреймворк машинного обучения**  
 интеграция тренеров (Ray Train), 177  
**Фреймворк на основе стратегии,** 24  
**Фреймворк тренировки, тренеры**  
 в качестве оберток сторонних фреймворков тренировки, 169  
**Функциональность, в распределенных вычислениях,** 25  
**Функция**  
 задания и акторы, работающие как распределенные версии, 48  
 конвертирование функции Python в задание Ray, 52  
**Функция взятия образцов (Tune),** 130, 140  
**Функция обратного вызова**  
 использование для мониторинга тренировки в библиотеке Ray Train, 185  
 конфигурирование для Ray Tune, 137  
 объект TuneReportCallback как конкретно-прикладной обратный вызов фреймворка Keras, 144  
 передача LoggerCallback в конфигурацию выполнения настройщика, 261  
 MLFlowLoggerCallback, 256  
**Функция целевая,** 40, 41, 127  
 определение, 130  
 определение для вычисления промежуточных баллов, 134  
 остановка анализа целевой функции Ray Tune, 139  
 целевая функция Keras в библиотеке Tune, 144  
**Фьючерс,** 53, 55, 57  
 обработка функцией ray.get в follow\_up\_task, 57

## Х

**Хранилище в формате,** 64  
**Хранилище объектов,** 48  
 использование с помощью put и get, 53  
 размещение базы данных, 54  
 размещение политики, 88  
 элемент компонентов Raylet, 61

**Ц**

Цель, 127

**Ч**

Чат-бот, 188

Чтение из хранилища и запись в него  
(наборы данных Dataset фреймворка  
Ray), 150

**Ш**

Шаг (в обучении с подкреплением), 90, 96  
передаваемые действия  
в многоагентной среде, 112

Шкалировщик  
минимаксный (MinMaxScaler), 183  
стандартный (StandardScaler), 183

**Э**

Экосистема (Ray), 28, 250  
загрузка и предобработка данных, 251  
инструментарий Ray AIR и более  
широкая экосистема машинного

обучения, 263

интегрирование инструментария  
Ray AIR в платформы машинного  
обучения, 266

куда отсюда двигаться дальше, 267

подача моделей как служб, 257

разработка конкретно-прикладных  
интеграций, 260

рост экосистемы, 251

тренировка моделей, 253

фреймворки распределенных

вычислений на Python, 263

Ray и другие системы, 262

Эксплуатация либо разведка среды  
в обучении с подкреплением, 91

Элемент, извлечение из базы данных, 50

Эпизод (в обучении с подкреплением), 85,  
90

Эпоха

в эпохе и ключевая логика тренировки  
на пакете данных на каждом работнике  
в каждую эпоху, 173

функция train\_one\_epoch, 178

число эпох, 175

Книги издательства «ДМК ПРЕСС»  
можно купить оптом и в розницу  
в книготорговой компании «Галактика»  
(представляет интересы издательств  
«ДМК ПРЕСС», «СОЛООН ПРЕСС», «КТК Галактика»).  
Адрес: г. Москва, пр. Андропова, 38, оф. 10;  
тел.: (499) 782-38-89, электронная почта: [books@aliants-kniga.ru](mailto:books@aliants-kniga.ru).  
При оформлении заказа следует указать адрес (полностью),  
по которому должны быть высланы книги;  
фамилию, имя и отчество получателя.

Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: <http://www.galaktika-dmk.com/>.

Макс Пумперла, Эдвард Оукс и Ричард Ляо

## Изучаем Ray

Главный редактор *Мовчан Д. А.*  
*dmkpress@gmail.com*  
Зам. главного редактора *Сенченкова Е. А.*  
Перевод *Логунов А. В.*  
Корректор *Синяева Г. И.*  
Верстка *Чаннова А. А.*  
Дизайн обложки *Мовчан А. Г.*

Гарнитура РТ Serif. Печать цифровая.  
Усл. печ. л. 23,56. Тираж 200 экз.

Веб-сайт издательства: [www.dmkpress.com](http://www.dmkpress.com)

Данная книга поможет программистам на Python, инженерам и исследователям данных научиться применять фреймворк распределенных вычислений с открытым исходным кодом Ray и разворачивать вычислительные кластеры Ray.

Ray может использоваться для структурирования и выполнения крупномасштабных программ машинного обучения. Распределенные вычисления отличаются своей сложностью, но с помощью Ray вы легко приступите к работе.

### Прочитав книгу, вы научитесь:

- создавать свои первые распределенные приложения с помощью ядра фреймворка – Ray Core;
- оптимизировать гиперпараметры с помощью библиотеки Ray Tune;
- применять библиотеку Ray RLlib для обучения с подкреплением;
- управлять распределенной тренировкой моделей с помощью библиотеки Ray Train;
- применять Ray для обработки данных с помощью библиотеки Ray Data;
- работать с кластерами Ray и подачей моделей в качестве служб с помощью библиотеки Ray Serve;
- создавать сквозные приложения машинного обучения с помощью инструментария Ray AIR.

*«Фантастическое введение в Ray. Авторы изящно разложили сложные темы распределенных вычислений и машинного обучения на ряд простых для понимания примеров».*

*Патрик Эймс,  
главный инженер, Amazon*

*«Лучшая книга по распределенным системам применительно к машинному обучению. Представляет собой доступное введение в создание приложений с массовым распределением данных, не выходя за пределы вашего блокнота Jupyter».*

*Марк Саруфим,  
инженер по искусственно  
му интеллекту, PyTorch, Meta*

**Макс Пумперла** – преподаватель науки о данных и инженер-программист в Anyscale.  
**Эдвард Оукс** – инженер-программист и руководитель коллектива разработчиков в Anyscale.  
**Ричард Ляо** – инженер-программист в Anyscale.

---

ISBN 978-6-01083-430-9



9 786010 834309 >