Содержание

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 програмные 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 аспиранитуре 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.

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, возможно, являетесь практиком машинного обучения или исследователем данных, которому 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 книги 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 предпочитаемого 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[1].

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-

[1] См. 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[1]. 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.

[1] См. 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.

Если у вас есть технический вопрос или проблема с использованием примеров исходного кода, то, пожалуйста, отправьте электронное письмо по адресу 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 Брейнделу 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 этому.

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[1] является неотъемлемой частью инструментария исследователя данных. Ка-

[1] 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 параллелизовывать 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»[1], 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

---

[1] 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 сосредоточиваться на своей работе. Но на практике, как отмечает Джоэл Спольски (Joel Spolsky)[1], «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[2] исследовательской лаборатории Deepmind для решения задачи сворачивания белков и Codex[3] 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»[4] 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[5], который гласит, что число транзисторов в компьютерах удваивается каждые два года. Даже если вы настроены опти-

---

1 См. https://oreil.ly/mpzSe.

2 См. https://oreil.ly/RFaMa.

3 См. https://oreil.ly/vGnyh.

4 См. AI and Compute, https://oreil.ly/7huR_.

5 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[1]?

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[2]. 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

[1] 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.

[2] 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 исполнение*, который хорошо справляется с зависимостями заданий. Фреймворк Ray предоставляет 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 исполнять 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[1]. 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[2];
- набор высокоуровневых библиотек, разработанных и поддерживаемых

---

[1] 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). Если весь ваш кластер сгорит дотла, то мы боимся, что фреймворк Ray не сможет вам помочь.

[2] 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[1]. 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 состыкующийся 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*[2], то есть на выполнение набора заданий[3], которые осуществляются в узлах кластера. В частности, отдельные задания заявки выполняются в процессах-*работниках* на *узлах-работниках*[4]. 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.

---

[1] 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.

[2] Англ. job. – *Прим. перев.*

[3] Англ. task. – *Прим. перев.*

[4] Англ. worker node. – *Прим. перев.*

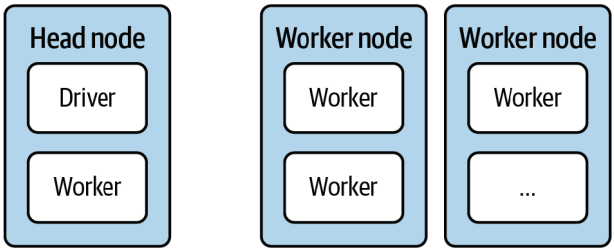

Рис. 1.1 ❖ Базовые компоненты кластера Ray

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[1]. 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[2].

---

[1] 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/j9ccz) дают возможность устанавливать все зависимости заранее, чтобы вместо этого вы могли сосредоточиваться на выполнении исходного кода.

[2] 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[1]. Подробнее об использовании клиента Ray для подключения к производственным кластерам можно прочитать в документации Ray[2].

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[3].

---

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

2 См. https://oreil.ly/nNhMt.

3 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[1].

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 данных

Несколько уклончивый термин «наука о данных»[2] за последние годы претерпел значительную эволюцию, и в интернете можно найти множество определений различной полезности[3]. 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»[4].

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:

---

1 См. https://oreil.ly/7r0Lv.

2 Англ. data science. – *Прим. перев.*

3 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.

4 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[1] – 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[2]. 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

---

[1] Англ. model serving. – *Прим. перев.*

[2] 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 Ameisen, 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 *распределимы 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.

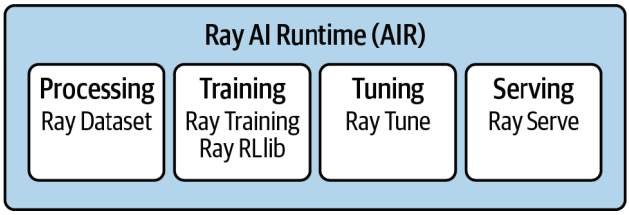

Рис. 1.2 ❖ Инструментарий Ray AIR является зонтичным компонентом для всех встроенных во фреймворк Ray текущих библиотек науки о данных

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 фреймфорками обработки данных. Абстракция `Dataset` строится поверх мощного фреймворка Arrow[1].

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

[1] 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.

pyarrow. 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[1], 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()
```

[1] 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])  ❸
sample = cubes.take(10)  ❹
print(sample)
```

❶ Мы преобразовываем (map) каждую строку в ds, оставляя только квадратное значение ее записи data.
❷ Затем мы фильтруем (filter) квадраты (squares), оставляя только четные числа (в общей сложности пять тысяч элементов).
❸ Затем мы применяем flat_map, чтобы дополнить оставшиеся значения соответствующими кубами.
❹ 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() ❶
result = pipe\
    .map(lambda x: x[“data”] ** 2)\
    .filter(lambda x: x % 2 == 0)\
    .flat_map(lambda x: [x, x**3]) ❷
result.show(10)
```

❶ Объект Dataset можно превратить в конвейер, вызвав на нем функцию .window().
❷ Шаги конвейера могут быть выстроены в цепочку и 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[1]. Эта библиотека основана на современных фреймворках машинного

[1] 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 tensorflow`[1]. 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.

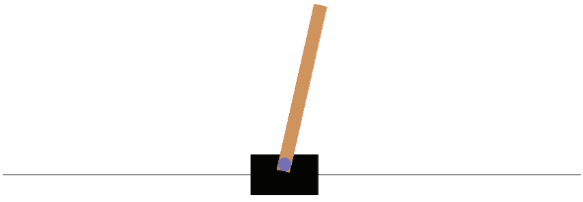

Рис. 1.3 ❖ Управление шестом, прикрепленным к тележке, путем приложения усилия влево либо вправо

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

[1] 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 с помощью 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`[1] 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[2], 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

[1] См. https://oreil.ly/YNxoz.

[2] 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 задействовать 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[1], 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 RLlib. Это не совпадение, поскольку в RLlib (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.

---

[1] 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*[1] 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 параллелизует 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 служб 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.

---

[1] 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[1]. 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[2]:

---

[1] 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`.

[2] 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[1], 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[2], вы можете использовать эти инструмен-

---

[1] См. https://oreil.ly/XUT7y.

[2] Spark был создан другой лабораторией в Беркли, AMPLab. Интернет полон блогпотов, 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[1], чтобы интегрировать рабочие нагрузки Spark с Ray. Аналогичным образом проект Modin[2] является упрощенной распределенной заменой наборов данных DataFrame[3] 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[4]. 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 практически 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-

---

1 См. https://oreil.ly/J1D5I.

2 См. https://oreil.ly/brGPJ.

3 Кадр данных. – *Прим. перев.*

4 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 гиперпарамтерической 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 выглядит 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?

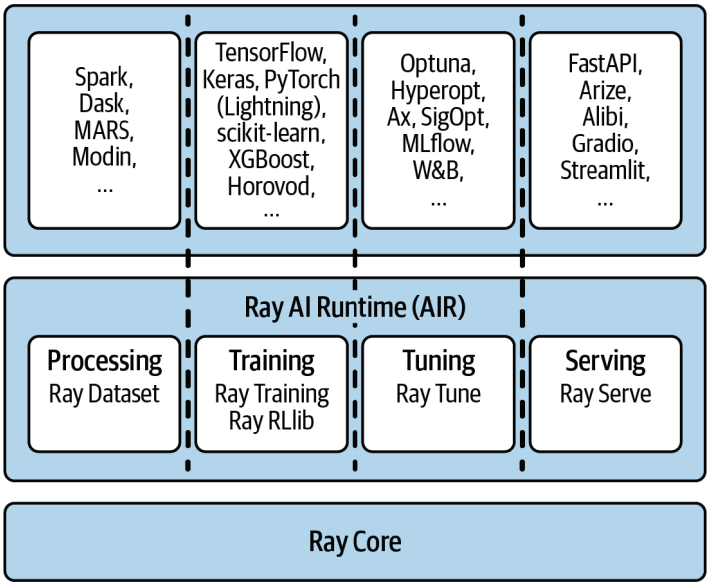

Рис. 1.4 ❖ Фреймворк Ray в трех слоях

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.