O'REILLY®

Learning Ray

Flexible Distributed Python for Machine Learning

Early Release
Raw & Unedited
Compliments of
anyscale

Max Pumperla,
Edward Oakes
& Richard Liaw

Learning Ray

Flexible Distributed Python for Machine Learning

Max Pumperla, Edward Oakes, and Richard Liaw

Learning Ray

by Max Pumperla, Edward Oakes, and Richard Liaw

Printed in the United States of America.

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

- Editors: Jeff Bleiel and Jessica Haberman
- Production Editor: Katherine Tozer
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea
- April 2023: First Edition

Revision History for the Early Release

- 2022-01-21: First Release
- 2022-03-11: Second Release
- 2022-04-21: Third Release
- 2022-06-06: Fourth Release
- 2022-07-13: Fifth Release

- 2022-08-24: Sixth Release
- 2022-09-23: Seventh Release
- 2022-11-08: Eighth Release
- 2022-12-07: Ninth Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098117221 for release details.

978-1-098-11716-0

Dedication

Für Alma

Preface

Distributed computing is a fascinating topic. Looking back at the early days of computing, one can't help but be impressed by the fact that so many companies today distribute their workloads across clusters of computers. It's impressive that we have figured out efficient ways to do so, but scaling out is also becoming more and more of a necessity. Individual computers keep getting faster, and yet our need for large scale computing keeps exceeding what single machines can do.

Recognizing that scaling is both a necessity and a challenge, Ray aims to make distributed computing simple for developers. It makes distributed computing accessible to non-experts and makes it possible to scale your Python scripts across multiple nodes fairly easily. Ray is good at scaling both *data and compute heavy workloads*, such as data preprocessing and model training — and it explicitly targets machine learning (ML) workloads with the need to scale. While it is possible today to scale these two types of workloads without Ray, you would likely have to use different APIs and distributed systems for each. And managing several distributed systems can be messy and inefficient in many ways.

The addition of the Ray AI Runtime (AIR) with the release of Ray 2.0 in August 2022 increased the support for complex ML workloads in Ray even further. AIR is a collection of libraries and tools that make it easy to build and deploy end-to-end ML applications in a single distributed system. With AIR, even the most complex workflows can usually be expressed as a *single Python script*. That means you can run your programs locally first, which can make a big difference in terms of debugging and development speed.

Data scientists benefit from Ray, as they can rely on a growing ecosystem of Ray ML libraries and third party integrations. Ray AIR

helps you to quickly prototype ideas and go more easily from development to production. Unlike many other distributed systems, Ray has native support for GPUs as well, which can be particularly important to roles like ML engineers. To support data engineers, Ray also has tight integrations with tools like Kubernetes and can be deployed in multi-cloud setups. And you can use it as unified compute layer to provide scaling, fault tolerance, scheduling and orchestration of your workloads. In other words, it's well worth investing in *learning Ray* for a variety of roles.

Who Should Read This Book

It's likely that you picked up this book because you're interested in some aspects of Ray. Maybe you're a distributed systems engineer who wants to know how Ray's engine works. You might also be a software developer interested in picking up a new technology. Or you could be a data engineer who wants to evaluate how Ray compares to similar tools. You could also be a machine learning practitioner or data scientist who needs to find ways to scale his experiments.

No matter what your concrete role is, the common denominator to get the most out of this book is to feel comfortable programming in Python. This is a book written in Python and an intermediate knowledge of the language is a requirement. Explicit is better than implicit, as you know full well as Pythonista. So, let us be explicit by saying that knowing Python implies to me that you know how to use the command line on your system, how to get help when stuck, and how to set up a programming environment on your own.

If you've never worked with distributed systems before, that's ok. We cover all the basics you need to get started with this topic in the book. On top of that, you can run most code examples presented here on your laptop alone. At the same time, covering the basics means that we can't go into too much detail about distributed systems in this book. This book is ultimately focused on application

developers using Ray, specifically for data science and machine learning (ML).

If you want to follow the later chapters of this book, you need some familiarity with ML, but we don't expect you to have worked in the field. In particular, you should have a basic understanding of the machine learning paradigm and how it differs from traditional programming. You should also know the basics of using NumPy and Pandas. On top of that, you should at least feel comfortable *reading* examples using the popular TensorFlow and PyTorch libraries. It's enough to follow the flow of the code, on the API level, but you don't need to know how to write your own models. We cover examples using both dominant deep learning libraries (TensorFlow and PyTorch) to illustrate how you can use Ray for your ML workloads, regardless of your preferred framework.

We cover a lot of ground in advanced machine learning topics, but the main focus is on Ray as a technology and how to use it. The ML examples we discuss might be new to you and could require a second reading, but you can still focus on Ray's API and how to use it in practice.

Knowing the requirements, let's discuss what you might get out of this book.

- As a data scientist, Ray will open up new ways for you to think about and build distributed ML applications. You will know how to do hyperparameter selection for your experiments at scale, gain practical knowledge on large-scale model training and get to know a state-of-the-art reinforcement learning library.
- As a data engineer you will learn to use Ray Datasets for large-scale data ingest, how to improve your pipelines by leveraging tools such as Dask on Ray, and how to effectively deploy models at scale.

- As an engineer you will understand how Ray works under the hood, how to run and scale Ray clusters in the cloud, and how Ray can be used to build applications that integrate with projects you know.

You can learn all of these topics above regardless of your role, of course. Our hope is that by the end of this book you've learned to appreciate Ray for all its strengths.

Goals of This Book

This book is primarily written for readers who are new to Ray and want to get the most out of it quickly. We've chosen the material in such a way that you will understand the core ideas behind Ray and learn to use its main building blocks. Having read it, you will feel comfortable navigating more complex topics on your own that go beyond this introduction.

We should also be clear about what this book is not. It's not built to give you the most information possible, like API references or definitive guides. It's also not crafted to help you tackle concrete tasks, like how-to guides or cookbooks do. This book is focused on learning and understanding Ray, and giving you interesting examples to start with.

Software develops and deprecates quickly, but the fundamental concepts underlying software often remain stable even across majors release cycles. We're trying to strike a balance here between conveying ideas and providing you with concrete code examples. The ideas you find in this book hopefully remain useful even when the code eventually needs updating.

While Ray's documentation keeps getting better, we do believe that books can offer qualities that are difficult to match in a project's documentation. Since you're reading these lines, we realise we might be knocking down open doors with this statement. But some of

the best tech books we know spark your interest about a project and make you want to dig through terse API references that you'd never have touched otherwise. We hope this one of those books.

Navigating This Book

We organized this book to guide you from core concepts to more sophisticated topics of Ray in a natural way. Many of the ideas explained come with example code that you can find all in the book's GitHub repo.

In a nutshell, the first three chapters of the book teach the basics of Ray as a distributed Python framework with practical examples. Chapters four to ten introduce Ray's high-level libraries and show how to build applications with them. The last chapter gives you a conclusive overview of Ray's ecosystem and show you where to go next. Here's what you can expect from each chapter.

- Chapter 1, "An Overview of Ray"
  Introduces you to Ray as a system composed of three layers: its core, its ML libraries and its ecosystem. You'll run your first examples with Ray's libraries in this chapter, to give you a glimpse of what you can do with Ray.
- Chapter 2, "Getting Started with Ray Core"
  Walks you through the foundations of the Ray project, namely its core API. It also discusses how Ray Tasks and Actors naturally extend from Python functions and classes. You will also learn about Ray's system components and how they work together.
- Chapter 3, "Building Your First Distributed Application"
  Guides you through implementing a distributed reinforcement learning application with Ray Core. You will implement this app from scratch and will see Ray's flexibility in distributing your Python code in action.

- Chapter 4, “Reinforcement Learning with Ray RLlib”
  Gives you a quick introduction to reinforcement learning and shows how Ray implements important concepts in RLlib. After building some examples together, we’ll also dive into more advanced topics like curriculum learning or working with offline data.
- Chapter 5, “Hyperparameter Optimization with Ray Tune”
  Covers why efficiently tuning hyperparameters is hard, how Ray Tune works conceptually, and how you can use it in practice for your machine learning projects.
- Chapter 6, “Data Processing with Ray”
  Introduces you to the Ray Datasets abstraction of Ray and how it fits into the landscape of other data processing systems. You will also learn how to work with third-party integrations such as Dask on Ray.
- Chapter 7, “Distributed Training with Ray Train”
  Provides you with the basics of distributed model training and shows you how to use Ray Train with ML frameworks like PyTorch. We also show you how to add custom preprocessors to your models, how to monitor training with callbacks, and how to tune the hyperparameters of your models with Tune.
- Chapter 8, “Online Inference with Ray Serve”
  Teaches you the basics of exposing your trained ML models as API endpoints that can be queried from anywhere. We discuss how Ray Serve addresses the challenges of online inference, cover its architecture, and show you how to use it in practice.
- Chapter 9, “Ray Clusters”
  This chapter is all about how you configure, launch and scale Ray clusters for your applications. You’ll learn about Ray’s cluster launcher CLI and autoscaler, as well as how to set up

clusters in the cloud. We'll also show you how to deploy Ray on Kubernetes and with other cluster managers.

- Chapter 10, "Getting Started with the Ray AI Runtime"
  Introduces you to Ray AIR, a unified toolkit for your ML workloads that offers many third party integrations for model training or accessing custom data sources.
- Chapter 11, "Ray's Ecosystem and Beyond"
  Gives you an overview of the many interesting extensions and integrations that Ray has attracted over the years.

How to Use the Code Examples

You can find all the code for this book in the following GitHub repository. In the GitHub repo you find a `notebook` folder with notebooks for each chapter. We built the examples in such a way that you can either type along as you read, or follow the main text and run the code from GitHub at another time. The choice is yours.

For the examples we assume that you have Python 3.7 or later installed. At the time of this writing, support for Python 3.10 for Ray is experimental, so we can currently only recommend a Python version no later than 3.9. All code examples assume that you have Ray installed, and each chapter adds its own specific requirements. The examples have been tested on Ray version 2.2.0, and we recommend to stick to this version for the whole book.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

`Constant width italic`

Shows text that should be replaced with user-supplied values or by values determined by context.

> **TIP**
>
> This element signifies a tip or suggestion.

> **NOTE**
>
> This element signifies a general note.

> **WARNING**
>
> This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/oreillymedia/title_title*.

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Learning Ray* by Max Pumperla, Edward Oakes, and Richard Liaw (O'Reilly). Copyright 2023 Max Pumperla and O'Reilly Media, Inc., 978-1-098-11722-1."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

> **NOTE**
>
> For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning

platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/learning-ray*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and information about our books and courses, visit *https://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly-media*.

Follow us on Twitter: *https://twitter.com/oreillymedia*.

Watch us on YouTube: *https://youtube.com/oreillymedia*.

Acknowledgments

We'd like to acknowledge the whole team at O'Reilly for helping us make this book possible. In particular, we'd like to thank our tireless editor, Jeff Bleiel, for invaluable input and feedback. Many thanks to Jess Haberman for many fruitful discussions and having an open mind in the early stages of the process. We'd also like to thank Katherine Tozer, Chelsea Foster, and Cassandra Furtado, among many others at O'Reilly.

Many thanks to all the reviewers for their valuable feedback and suggestions: Mark Saroufim, Kevin Ferguson, Adam Breindel, and Jorge Davila-Chacon. We'd also like to thank the many colleagues at Anyscale who helped us with the book in any capacity, including: Sven Mika, Stephanie Wang, Antoni Baum, Christy Bergman, Dmitri Gekhtman, Zhe Zhang, and many others.

On top of that, we'd like to wholeheartedly thank the Ray contributors team and the community for their support and feedback, as well as many key stakeholders at Anyscale supporting this project.

Max would also like to thank the team at Pathmind for their support in the early phases of the project, especially Chris Nicholson, who has been more helpful over the years than I could describe here. Special thanks go out to the Espresso Society in Winterhude for helping me turn coffee into books, and an increasing array of GPT-3-based tools helping me finish half-sentences when the caffeine wore off. I would also like to express my gratitude to my family for their encouragement and patience. As always, none of this would have been possible without Anne, who always supports me when it counts — even if I take on one-too-many projects such as this one.

Chapter 1. An Overview of Ray

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

One of the reasons we need efficient distributed computing is that we're collecting ever more data with a large variety at increasing speeds. The storage systems, data processing and analytics engines that have emerged in the last decade are crucially important to the success of many companies. Interestingly, most "big data" technologies are built for and operated by (data) engineers, that are in charge of data collection and processing tasks. The rationale is to free up data scientists to do what they're best at. As a data science practitioner you might want to focus on training complex machine learning models, running efficient hyperparameter selection, building entirely new and custom models or simulations, or serving your models to showcase them.

At the same time, it might be *inevitable* to scale these workloads to a compute cluster. To do that, the distributed system of your choice needs to support all of these fine-grained "big compute" tasks, potentially on specialized hardware. Ideally, it also fits into the big data tool chain you're using and is fast enough to meet your latency requirements. In other words, distributed computing has to be powerful and flexible enough for complex data science workloads — and Ray can help you with that.

Python is likely the most popular language for data science today, and it's certainly the one we find the most useful for our daily work.

By now it's over 30 years old, but has a still growing and active community. The rich PyData ecosystem is an essential part of a data scientist's toolbox. How can you make sure to scale out your workloads while still leveraging the tools you need? That's a difficult problem, especially since communities can't be forced to just toss their toolbox, or programming language. That means distributed computing tools for data science have to be built for their existing community.

What Is Ray?

What I like about Ray is that it checks all the above boxes. It's a flexible, distributed computing framework build for the Python data science community.

Ray is easy to get started and keeps simple things simple. Its core API is as lean as it gets and helps you reason effectively about the distributed programs you want to write. You can efficiently parallelize Python programs on your laptop, and run the code you tested locally on a cluster practically without any changes. Its high-level libraries are easy to configure and can seamlessly be used together. Some of them, like Ray's reinforcement learning library, would likely have a bright future as standalone projects, distributed or not. While Ray's core is built in C++, it's been a Python-first framework since day one,[1] integrates with many important data science tools, and can count on a growing ecosystem.

Distributed Python is not new, and Ray is not the first framework in this space (nor will it be the last), but it is special in what it has to offer. Ray is particularly strong when you combine several of its modules and have custom, machine learning heavy workloads that would be difficult to implement otherwise. It makes distributed computing easy enough to run your complex workloads flexibly by leveraging the Python tools you know and want to use. In other words, by *learning Ray* you get to know *flexible distributed Python*

for machine learning. And showing you how is what this book is all about.

In this chapter you'll get a first glimpse at what Ray can do for you. We will discuss the three layers that make up Ray, namely its core engine, its high-level libraries and its ecosystem. Throughout the chapter we'll show you first code examples to give you a feel for Ray. You can view this chapter as a quick preview of the book, and we defer any in-depth treatment of Ray's APIs and components to later chapters.

What Led to Ray?

Programming distributed systems is hard. It requires specific knowledge and experience you might not have. Ideally, such systems get out of your way and provide abstractions to let you focus on your job. But in practice "all non-trivial abstractions, to some degree, are leaky" (Spolsky), and getting clusters of computers to do what you want is undoubtedly difficult. Many software systems require resources that far exceed what single servers can do. Even if one server was enough, modern systems need to be failsafe and provide features like high availability. That means your applications might have to run on multiple machines, or even datacenters, just to make sure they're running reliably.

Even if you're not too familiar with machine learning (ML) or more generally artificial intelligence (AI) as such, you must have heard of recent breakthroughs in the field. To name just two, systems like Deepmind's AlphaFold for solving the protein folding problem, or OpenAI's Codex that's helping software developers with the tedious parts of their job, have made the news lately. You might also have heard that ML systems generally require large amounts of data to be trained, and that ML models tend to get larger. OpenAI has shown exponential growth in compute needed to train AI models in their paper "AI and Compute". The operations needed for AI systems in

their study is measured in petaflops (thousands of trillion operations per second), and has been *doubling every 3.4 months* since 2012.

Compare this to Moore's Law,[2] which states that the number of transistors in computers would double every two years. Even if you're bullish on Moore's law, you can see how there's a clear need for distributed computing in ML. You should also understand that many tasks in ML can be naturally decomposed to run in parallel. So, why not speed things up if you can?[3]

Distributed computing is generally perceived as hard. But why is that? Shouldn't it be realistic to find good abstractions to run your code on clusters, without having to constantly think about individual machines and how they interoperate? What if we specifically focused on AI workloads?

Researchers at RISELab at UC Berkeley created Ray to address these questions. They were looking for efficient ways to speed up their workloads by distributing them. The workloads they had in mind were quite flexible in nature and didn't fit into the frameworks available at the time. RISELab also wanted to build a system that took care of how the work was distributed. With reasonable default behaviors in place, researchers should be able to focus on their work, regardless of the specifics of their compute cluster. And ideally they should have access to all their favorite tools in Python. For this reason, Ray was built with an emphasis on high-performance and heterogeneous workloads.[4] To understand these points better, let's have a closer look at Ray's design philosophy next.

Ray's Design Principles

Ray is built with several design principles in mind. Its API is designed for simplicity and generality, and its compute model aims for flexibility. And its system architecture is designed for performance and scalability. Let's look at each of these in more detail.

Simplicity and abstraction

Ray's API does not only bank on simplicity, it's also intuitive to pick up (as you'll see in Chapter 2). It doesn't matter whether you just want to use all the CPU cores on your laptop or leverage all the machines in your cluster. You might have to change a line of code or two, but the Ray code you use stays essentially the same. And as with any good distributed system, Ray manages task distribution and coordination under the hood. That's great, because you're not bogged down by reasoning about the mechanics of distributed computing. A good abstraction layer allows you to focus on your work, and I think Ray has done a great job of giving you one.

Since Ray's API is so generally applicable and *pythonic*, it's easy to integrate with other tools. For instance, Ray actors can call into or be called by existing distributed Python workloads. In that sense Ray makes for good "glue code" for distributed workloads, too, as its performant and flexible enough to communicate between different systems and frameworks.

Flexibility and heterogeneity

For AI workloads, in particular when dealing with paradigms like reinforcement learning, you need a flexible programming model. Ray's API is designed to make it easy to write flexible and composable code. Simply put, if you can express your workload in Python, you can distribute it with Ray. Of course, you still need to make sure you have enough resources available and be mindful of what you want to distribute. But Ray doesn't limit you in what you can do with it.

Ray is also flexible when it comes to *heterogeneity* of computations. For instance, let's say you work on a complex simulation. Simulations can usually be decomposed into several tasks or steps. Some of these steps might take hours to run, others just a few milliseconds, but they always need to be scheduled and executed quickly. Sometimes a single task in a simulation can take a long

time, but other, smaller tasks should be able to run in parallel without blocking it. Also, subsequent tasks may depend on the outcome of an upstream task, so you need a framework to allow for *dynamic execution* that deals well with task dependencies. Ray gives you full flexibility running heterogeneous workflows like that.

You also need to ensure you are flexible in your resource usage, and Ray supports heterogeneous hardware. For instance, some tasks might have to run on a GPU, while others run best on a couple of CPU cores. Ray provides you with that flexibility.

Speed and scalability

Another of Ray's design principles is the speed at which Ray executes its tasks. It can handle millions of tasks per second, and you only incur very low latencies with it. Ray is built to execute its tasks with just milliseconds of latency.

For a distributed system to be fast, it also needs to scale well. Ray is efficient at distributing and scheduling your tasks across your compute cluster. And it does so in a fault-tolerant way, too. As you'll learn in detail in Chapter 9, Ray clusters support *auto-scaling* to support highly elastic workloads. Ray's autoscaler tries to launch or stop machines in your cluster to match the current demand. This helps both minimising costs and ensuring that your cluster has enough resources to run your workload.

In distributed systems it's not a question of if, but when things go wrong. A machine might have an outage, abort a task or simply go up in flames.[5] In any case, Ray is built to recover quickly from failures, which contributes to its overall speed.

As we haven't talked about Ray's architecture (Chapter 2 will introduce you to it), we can't tell you how these design principles are realized just yet. Let's instead shift our attention to what Ray can do for you in practice.

Three Layers: Core, Libraries, and Ecosystem

Now that you know why Ray was built and what its creators had in mind, let's look at the three layers of Ray. This presentation is not the only way to slice it, but it's the way that makes most sense for this book.

1. A low-level, distributed computing framework for Python with a concise core API and tooling for cluster deployment called Ray Core.[6]
2. A set of high-level libraries for built and maintained by the creators of Ray. This includes the so-called Ray AI Runtime (AIR) to use these libraries with a unified API in common machine learning workloads.
3. A growing ecosystem of integrations and partnerships with other notable projects, which span many aspects of the first two layers.

There's a lot to unpack here, and we'll look into each of these layers individually in the remainder of this chapter.

You can imagine Ray's core engine with its API at the center of things, on which everything else builds. Ray's data science libraries build on top of Ray Core and provide a domain-specific abstraction layer.[7] In practice, many data scientists will use these libraries directly, while ML or platform engineers might heavily rely on building their tools as extensions of the Ray Core API. Ray AIR can be seen as an umbrella that links Ray libraries and offers a consistent framework for dealing with common AI workloads. And the growing number of third-party integrations for Ray is another great entrypoint for experienced practitioners. Let's look into each one of the layers one by one.

A Distributed Computing Framework

At its core, Ray is a distributed computing framework. We'll provide you with just the basic terminology here, and talk about Ray's architecture in depth in Chapter 2. In short, Ray sets up and manages clusters of computers so that you can run distributed tasks on them. A Ray cluster consists of nodes that are connected to each other via a network. You program against the so-called *driver*, the program root, which lives on the *head node*. The driver can run *jobs*, that is a collection of tasks, that are run on the nodes in the cluster. Specifically, the individual tasks of a job are run on *worker* processes on *worker nodes*. Figure Figure 1-1 illustrates the basic structure of a Ray cluster. Note that we're not concerned with communication between nodes just yet, this diagram merely shows the layout of a Ray cluster.

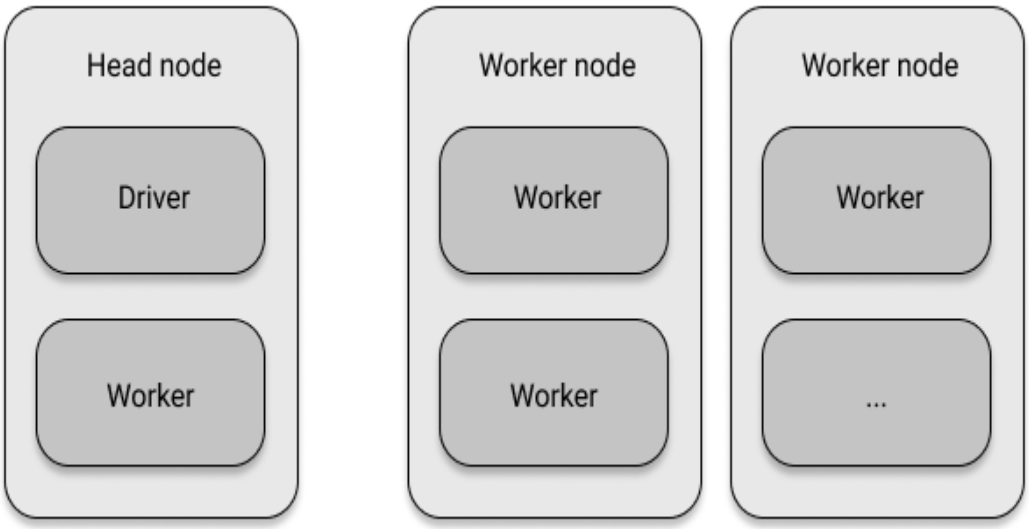


Figure 1-1. The basic components of a Ray cluster.

What's interesting is that a Ray cluster can also be a *local cluster*, that is a cluster consisting just of your own computer. In this case, there's just one node, namely the head node, which has the driver process and some worker processes. The default number of worker processes is the number of CPUs available on your machine.

With that knowledge at hand, it's time to get your hands dirty and run your first local Ray cluster. Installing Ray on any of the major operating systems should work seamlessly using `pip`:

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

With a simple `pip install ray` you would have installed just the very basics of Ray. Since we want to explore some advanced features, we installed the "extras" `rllib`, `serve` and `tune`, which we'll discuss in a bit.[8] Depending on your system configuration you may not need the quotation marks in the above installation command.

Next, go ahead and start a Python session. You could for instance use the `ipython` interpreter, which is often suitable for following along simple examples. If you don't feel like typing in the commands yourself, you can also jump into the jupyter notebook for this chapter and run the code there. The choice is up to you, but in any case please remember to use Python version `3.7` or later.[9] In your Python session you can now easily import and initialize Ray as follows:

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

With those two lines of code you've started a Ray cluster on your local machine. This cluster can utilize all the cores available on your computer as workers. Right now your Ray cluster doesn't do much, but that's about to change in the next section.

The `init` function you use to start the cluster is one of the just six fundamental API calls that you will learn about in depth in Chapter 2. Overall, the *Ray Core API* is very accessible and easy to use. But since it is also a rather low-level interface, it takes time to build interesting examples with it. Chapter 2 has an extensive first example to get you started with the Ray core API, and in Chapter 3

you'll see how to build a more interesting Ray application for reinforcement learning.

In the above example you didn't provide any arguments to the `ray.init(...)` function. If you wanted to run Ray on a "real" cluster, you'd have to pass more arguments to `init`. This `init` call is often called the Ray Client, and it is used to interactively connect against an existing Ray cluster.[10] You can read more about using the Ray Client to connect against your production clusters in the Ray documentation.

Of course, if you've ever worked with compute clusters, there are many pitfalls and intricacies. For instance, you can deploy Ray applications on clusters hosted by cloud providers such as Amazon Web Services (AWS), Google Cloud Platform (GCP), or Microsoft Azure — and each choice needs good tooling for deployment and maintenance. You can also spin up a cluster on your own hardware, or use tools such as Kubernetes to deploy your Ray clusters. In Chapter 9 (after you've seen some concrete Ray applications in prior chapters), we'll come back to the topic of scaling workloads with *Ray Clusters*.

Before moving on the Ray's higher level libraries, let's briefly summarize the two foundational components of Ray as a distributed computation framework as follows:

- *Ray Clusters*: This component is in charge of allocating resources, creating nodes, and ensuring they are healthy. A good way to get started with Ray Clusters is its dedicated quick start guide.
- *Ray Core*: Once your cluster is up and running, you use the Ray Core API that to program against it. You can get started with Ray Core by following the official walk-through for this component.

A Suite of Data Science Libraries

Moving on to the second layer of Ray, in this section we'll briefly introduce all the data science libraries that Ray comes with. To do so, let's first take a bird's eye view on what it means to do data science. Once you understand this context, it's much easier to place Ray's higher-level libraries and see how they can be useful to you.

Ray AIR and the Data Science Workflow

The somewhat elusive term "data science" (DS) evolved quite a bit in recent years, and you can find many definitions of varying usefulness online.[11] To us, it's *the practice of gaining insights and building real-world applications by leveraging data*. That's quite a broad definition of an inherently practical and applied field that centers around building and understanding things. In that sense, describing practitioners of this field as "data scientists" is about as bad of a misnomer as describing hackers as "computer scientists."[12]

In broad strokes, doing data science is an iterative process that entails requirements engineering, data collection and processing, building models and evaluating them, and deploying solutions. Machine learning is not necessarily part of this process, but often is. If ML is involved you can further specify some steps:

- *Data Processing*: To train machine learning models, you need data in a format that is understood by your ML model. The process of transforming and selecting what data should be fed into your model is often called *feature engineering*. This step can be messy. You'll benefit a lot if you can rely on common tools to do the job.
- *Model Training*: In ML you need to train your algorithms on data that got processed in the last step. This includes selecting the right algorithm for the job, and it helps if you can choose from a wide variety.

- *Hyperparameter Tuning*: Machine learning models have parameters that are tuned in the model training step. Most ML models also have another set of parameters, called *hyperparameters* that can be modified prior to training. These parameters can heavily influence the performance of your resulting ML model and need to be tuned properly. There are good tools to help automate that process.
- *Model Serving*: Trained models need to be deployed. To serve a model means to make it available to whoever needs access by whatever means necessary. In prototypes, you often use simple HTTP servers, but there are many specialized software packages for ML model serving.

This list is by no means exhaustive and there's a lot more to be said about building ML applications.[13] However, it is true that these four steps are crucial for the success of a data science project using ML.

Ray has dedicated libraries for each of the four ML-specific steps above. Specifically, you can take care of your data processing needs with *Ray Datasets*, run distributed model training with *Ray Train*, run your Reinforcement Learning workloads with *Ray RLlib*, tune your hyperparameters efficiently with *Ray Tune*, and serve your models with *Ray Serve*. And the way Ray is built, all these libraries are *distributed by design*, a point that can't be stressed enough.

What's more is that all of these steps are part of a process and are rarely tackled in isolation. Not only do you want all the libraries involved to seamlessly interoperate, it can also be a decisive advantage if you can work with a consistent API throughout the whole data science process. This is exactly what the Ray AI Runtime (AIR) was built for: having a common runtime and API for your experiments and the ability to scale your workloads when you're ready. Figure Figure 1-2 shows you a quick overview all the components of AIR.

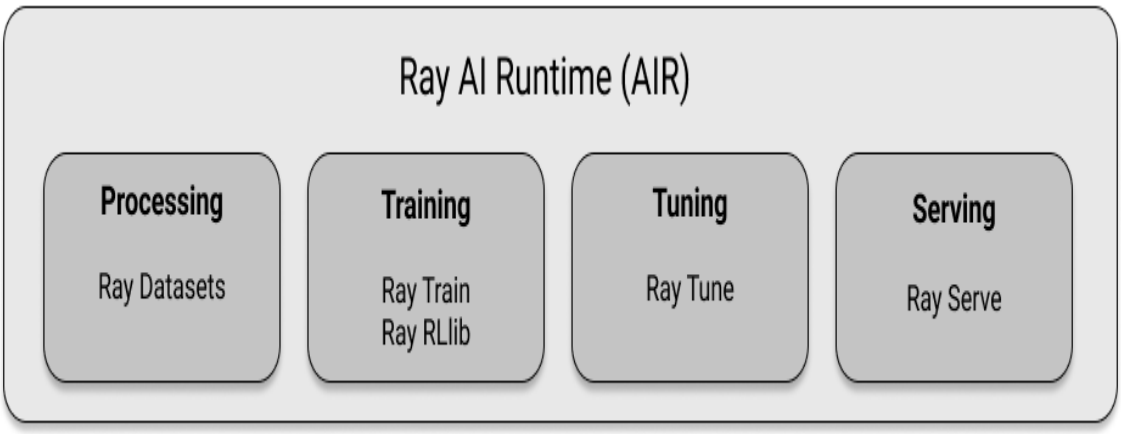


Figure 1-2. Ray AIR as an umbrella of all current data science libraries of Ray.

While introducing the Ray AI Runtime API would be too much for this chapter (you can jump ahead to Chapter 10 for more on AIR), we can introduce you to all the building blocks that feed into it. Let's go through each of Ray's DS libraries one by one.

Data Processing with Ray Datasets

The first high-level library of Ray we talk about is called "Ray Datasets". This library contains a data structure aptly called `Dataset`, a multitude of connectors for loading data from various formats and systems, an API for transforming such datasets, a way to build data processing pipelines with them, and many integrations with other data processing frameworks. The `Dataset` abstraction builds on the powerful Arrow framework.[14]

To use Ray Datasets, you need to install Arrow for Python, for instance by running `pip install pyarrow`. We'll now discuss a simple example that creates a distributed `Dataset` on your local Ray cluster from a Python data structure. Specifically, you'll create a dataset from a Python dictionary containing a string `name` and an integer-valued `data` for `10000` entries:

```
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 Creating a `Dataset` by using `from_items` from the `ray.data` module.
❷ Printing the first 5 items of the `Dataset`.

To `show` a `Dataset` means to print some of its values. You should see precisely `5` so-called `ArrowRow` elements on your command line, like this:

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

Great, now you have some rows, but what can you do with that data? The `Dataset` API bets heavily on functional programming, as this paradigm is well-suited for data transformations.

Even though Python 3 made a point of hiding some of its functional programming capabilities, you're probably familiar with functionality such as `map`, `filter` and others. If not, it's easy enough to pick up. `map` takes each element of your dataset and transforms is into something else, in parallel. `filter` removes data points according to a boolean filter function. And the slightly more elaborate `flat_map` first maps values similarly to `map`, but then also "flattens" the result. For instance, if `map` would produce a list of lists, `flat_map` would flatten out the nested lists and give you just a list. Equipped with these three functional API calls,[15] let's see how easily you can transform your dataset `ds`:

```
squares = ds.map(lambda x: x["data"] ** 2)  ❶
```

```
evens = squares.filter(lambda x: x % 2 == 0)  ❷
evens.count()

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

❶ We `map` each row of `ds` to only keep the square value of its `data` entry.
❷ Then we `filter` the `squares` to only keep even numbers (a total of 5000 elements).
❸ We then use `flat_map` to augment the remaining values with their respective cubes.
❹ To `take` a total of `10` values means to leave Ray and return a Python list with these values that we can print.

The drawback of `Dataset` transformations is that each step gets executed synchronously. In this example that is a non-issue, but for complex tasks that e.g. mix reading files and processing data, you want an execution that can overlap individual tasks. `DatasetPipeline` does exactly that. Let's rewrite the last example into a pipeline.

```
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 You can turn a `Dataset` into a pipeline by calling `.window()` on it.
❷ Pipeline steps can be chained to yield the same result as before.

There's a lot more to be said about Ray Datasets, especially its integration with notable data processing systems, but we'll have to defer an in-depth discussion until Chapter 6.

Model Training

Moving on to the next set of libraries, let's look at the distributed training capabilities of Ray. For that, you have access to two libraries. One is dedicated to reinforcement learning specifically, the other one has a different scope and is aimed primarily at supervised learning tasks.

Reinforcement learning with Ray RLlib

Let's start with *Ray RLlib* for reinforcement learning. This library is powered by the modern ML frameworks TensorFlow and PyTorch, and you can choose which one to use. Both frameworks seem to converge more and more conceptually, so you can pick the one you like most without losing much in the process. Throughout the book we use both TensorFlow and PyTorch examples, so you can get a feel for both frameworks when using Ray. Go ahead and install it with `pip install tensorflow` right now.[16]

One of the easiest ways to run examples with RLlib is to use the command line tool `rllib`, which we've already implicitly installed earlier when running `pip install "ray[rllib]"`. Once you run more complex examples in Chapter 4, you will mostly rely on its Python API, but for now we just want to get a first taste of running RL experiments with RLlib.

We'll look at a fairly classical control problem of balancing a pole on a cart. Imagine you have a pole like the one in figure Figure 1-3, fixed at a joint of a cart, and subject to gravity. The cart is free to move along a frictionless track, and you can manipulate the cart by giving it a push from the left or the right with a fixed force. If you do this well enough, the pole will ideally remain in an upright position. For each time step the pole didn't fall over, we get a reward of `1`. Collecting a high reward is our goal - and the question is whether we can teach a reinforcement learning algorithm to do so for us.

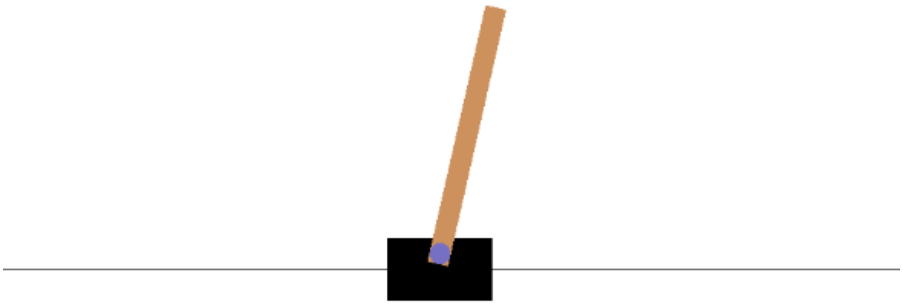


Figure 1-3. Controlling a pole attached to a cart by asserting force to the left or the right.

Specifically, we want to train a reinforcement learning agent that can carry out two actions, namely push to the left or to the right, observe what happens when interacting with the environment in that way, and learn from the experience by maximizing the reward.

To tackle this problem with Ray RLlib, we can use a so-called *tuned example*, which is a pre-configured algorithm that runs well for a given problem. You can run them with a single command. RLlib comes with many such examples, and you can list them all with `rllib example list`.

One of the available examples is `cartpole-ppo`, a tuned example that uses the PPO algorithm to solve the cartpole problem, specifically the `CartPole-v1` environment from OpenAI Gym. You can take a look at the configuration of this example by typing `rllib example get cartpole-ppo`, which will first download the example file from GitHub and then print its configuration. This configuration is encoded in YAML file format, and reads as follows:

```
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 The `CartPole-v1` environment simulates the problem we just described.
❷ We use a powerful RL algorithm called Proximal Policy Optimization, or PPO.
❸ Once we reach a reward of `150` , we stop the experiment.
❹ The PPO needs some RL-specific configuration to make it work for this problem.

The details of this configuration file don't matter much at this point, so don't get distracted by them. The important part is that you specify the `Cartpole-v1` environment and sufficient RL-specific configuration to ensure the training procedure works. Running this configuration doesn't require any special hardware and finishes in a matter of minutes. To train this example, you'll have to install the PyGame dependency with `pip install pygame`, and then simply run:

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

If you run this, RLlib creates a named experiment and logs important metrics such as the `reward`, or the `episode_reward_mean` for

you. In the output of the training run, you should also see information about the machine (`loc`, meaning host name and port), as well as the status of your training runs. If your run is `TERMINATED`, but you've never seen a successfully `RUNNING` experiment in the log, something must have gone wrong. Here's a sample snippet of a training run:

```
+-----------------------------+----------+-----------------+
| Trial name                  | status   | loc             |
|-----------------------------+----------+-----------------|
| PPO_CartPole-v0_9931e_00000 | RUNNING  | 127.0.0.1:8683 |
+-----------------------------+----------+-----------------+
```

When the training run finishes and things went well, you should see the following output:

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

You can now evaluate your trained algorithm from any
checkpoint,
e.g. by running:

╭──────────────────────────────────────────────────────────────
──────────────╮
│   rllib evaluate <checkpoint-path>/checkpoint_<number> --
algo PPO       │
╰──────────────────────────────────────────────────────────────
──────────────╯
```

Your local Ray checkpoint folder is `~/ray-results` by default. For the training configuration we used, your `<checkpoint-path>` should be of the form `~/ray_results/cartpole-ppo/PPO_CartPole-v1_<experiment_id>`. During training procedure, your intermediate and final model checkpoints get generated into this folder.

To evaluate the performance of your trained RL algorithm, you can now evaluate it *from checkpoint* by copying the command the previous example training run printed:

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

Running this command will print evaluation results, namely the rewards achieved by your trained RL algorithm on the `CartPole-v1` environment.

There's much more that you can do with RLlib, and we'll cover more of it in Chapter 4. The point of this example was to show you how easily you can get started with RLlib and the `rllib` command line tool, just by leveraging the `example` and `evaluate` commands.

Distributed training with Ray Train

Ray RLlib is dedicated to reinforcement learning, but what do you do if you need to train models for other types of machine learning, like supervised learning? You can use another Ray library for distributed training in this case, called *Ray Train*. At this point, we don't have built up enough knowledge of frameworks such as `TensorFlow` to give you a concise and informative example for Ray Train. If you're interested in distributed training, you can jump ahead to Chapter 6.

Hyperparameter Tuning

Naming things is hard, but *Ray Tune*, which you can use to tune all sorts of parameters, hits the spot. It was built specifically to find good hyperparameters for machine learning models. The typical setup is as follows:

- You want to run an extremely computationally expensive training function. In ML, it's not uncommon to run training procedures

that take days, if not weeks, but let's say you're dealing with just a couple of minutes.

- As result of training, you compute a so-called objective function. Usually you either want to maximize your gains or minimize your losses in terms of performance of your experiment.
- The tricky bit is that your training function might depend on certain parameters, hyperparameters, that influence the value of your objective function.
- You may have a hunch what individual hyperparameters should be, but tuning them all can be difficult. Even if you can restrict these parameters to a sensible range, it's usually prohibitive to test a wide range of combinations. Your training function is simply too expensive.

What can you do to efficiently sample hyperparameters and get "good enough" results on your objective? The field concerned with solving this problem is called *hyperparameter optimization* (HPO), and Ray Tune has an enormous suite of algorithms for tackling it. Let's look at a first example of Ray Tune used for the situation we just explained. The focus is yet again on Ray and its API, and not on a specific ML task (which we simply simulate for now).

```
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 We simulate an expensive training function that depends on two hyperparameters `x` and `y`, read from a `config`.

❷ After sleeping for 10 seconds to simulate training and computing the objective we report back the score to `tune`.

❸ The objective computes the mean of the squares of `x` and `y` and returns the square root of this term. This type of objective is fairly common in ML.

❹ We then use `tune.run` to initialize hyperparameter optimization on our `training_function`.

❺ A key part is to provide a parameter space for `x` and `y` for `tune` to search over.

First of all, notice how the output of this run is structurally similar to what you've seen in the RLlib example. That's no coincidence, as RLlib (like many other Ray libraries) uses Ray Tune under the hood. If you look closely, you will see `PENDING` runs that wait for execution, as well as `RUNNING` and `TERMINATED` runs. Tune takes care of selecting, scheduling, and executing your training runs for you automatically.

Specifically, this Tune example finds the best possible choices of parameters `x` and `y` for a `training_function` with a given `objective` we want to minimize. Even though the objective function might look a little intimidating at first, since we compute the sum of squares of `x` and `y`, all values will be non-negative. That means the smallest value is obtained at `x=0` and `y=0` which evaluates the objective function to `0`.

We do a so-called *grid search* over all possible parameter combinations. As we explicitly pass in five possible values for both `x` and `y` that's a total of `25` combinations that get fed into the training function. Since we instruct `training_function` to sleep for `10` seconds, testing all combinations of hyperparameters sequentially would take more than four minutes total. Since Ray is smart about parallelizing this workload, on my laptop this whole experiment only takes about `35` seconds, but it might take much longer, depending on where you run it.

Now, imagine each training run would have taken several hours, and we'd have 20 instead of two hyperparameters. That makes grid search infeasible, especially if you don't have educated guesses on the parameter range. In such situations you'll have to use more elaborate HPO methods from Ray Tune, as discussed in Chapter 5.

Model Serving

The last of Ray's high-level libraries we'll discuss specializes on model serving and is simply called *Ray Serve*. To see an example of it in action, you need a trained ML model to serve. Luckily, nowadays, you can find many interesting models on the internet that have already been trained for you. For instance, *Hugging Face* has a variety of models available for you to download directly in Python. The model we'll use is a language model called *GPT-2* that takes text as input and produces text to continue or complete the input. For example, you can prompt a question and GPT-2 will try to complete it.

Serving such a model is a good way to make it accessible. You may not know how to load and run a TensorFlow model on your computer, but you do now how to ask a question in plain English. Model serving hides the implementation details of a solution and lets users focus on providing inputs and understanding outputs of a model.

To proceed, make sure to run `pip install transformers` to install the Hugging Face library that has the model we want to use.[17] With that we can now import and start an instance of Ray's `serve` library, load and deploy a GPT-2 model and ask it for the meaning of life, like so:

```
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 We start `serve` locally.

❷ The `@serve.deployment` decorator turns a function with a `request` parameter into a `serve` deployment.

❸ Loading `language_model` inside the `model` function for every request is inefficient, but it's the quickest way to show you a deployment.

❹ We ask the model to give us at most `100` characters to continue our query.

❺ Then we formally deploy the model so that it can start receiving requests over HTTP.

❻ We use the indispensable `requests` library to get a response for any question you might have.

In Chapter 9 you will learn how to properly deploy models in various scenarios, but for now I encourage you to play around with this example and test different queries. Running the last two lines of code repeatedly will give you different answers practically every time. Here's a darkly poetic gem, raising more questions, that I queried on my machine and slightly censored for underaged readers:

```
[{
    "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."
}]
```

This concludes our whirlwind tour of Ray's data science libraries, the second of Ray's layers. Ultimately, all high level Ray libraries presented in this chapter are extensions of the Ray Core API. Ray makes it relatively easy to build new extensions, and there are a few more that we can't discuss in full in this book. For instance, there is the relatively recent addition of Ray Workflows, which allows you to define and run long-running applications with Ray.

Before we wrap up this chapter, let's have a very brief look at the third layer, the growing ecosystem around Ray.

A Growing Ecosystem

Ray's high-level libraries are powerful and deserve a much deeper treatment throughout the book. While their usefulness for the data science experimentation lifecycle is undeniable, we also don't want to give off the impression that Ray is all you need from now on. To no surprise, the best and most successful frameworks are the ones that

integrate well with existing solutions and ideas. It's better to focus on your core strengths and leverage other tools for what's missing in your solution, and Ray does this quite well.

Throughout the book, and Chapter 11 in particular, we will discuss many useful third-party libraries built on top of Ray. The Ray ecosystem also has a lot of integrations with existing tools. To give you an example of that, recall that Ray Datasets is Ray's data loading and compute library. If you happen to have an existing project that already uses data processing engines like Spark or Dask,[18] you can use those tools together with Ray. Specifically, you can run the entire Dask ecosystem on top of a Ray cluster using the Dask-on-Ray scheduler, or use the Spark on Ray project to integrate your Spark workloads with Ray as well. Likewise, the Modin project is a distributed drop-in replacement for Pandas dataframes that uses Ray (or Dask) as distributed execution engine ("Pandas on Ray").

The common theme here is that Ray doesn't try to replace, but rather integrate with all these tools, while still giving you access to its native Ray Datasets library. We'll go into much more detail about the relationship of Ray with other tools in the broader ecosystem in Chapter 11.

One important aspect of many Ray libraries is that they seamlessly integrate common tools as *backends*. Ray often creates common interfaces, instead of trying to create new standards.[19] These interfaces allow you to run tasks in a distributed fashion, a property most of the respective backends don't have, or not to the same extent.

For instance, Ray RLlib and Train are backed by the full power of TensorFlow and PyTorch. And Ray Tune supports algorithms from practically every notable HPO tool available, including Hyperopt, Optuna, Nevergrad, Ax, SigOpt and many others. None of these tools are distributed by default, but Tune unifies them in a *common interface for distributed workloads*.

Summary

To sum up what we've discussed in this chapter, Figure 1-4 gives you an overview of the three layers of Ray as we laid them out. Ray's core distributed execution engine sits at the center of the framework. The Ray Core API is a versatile library for distributed computing, and Ray Clusters allow you to deploy your workloads in a variety of ways.

For practical data science workflows you can use Ray Datasets for data processing, Ray RLlib for reinforcement learning, Ray Train for distributed model training, Ray Tune for hyperparameter tuning and Ray Serve for model serving. You've seen examples for each of these libraries and have an idea of what their APIs entail. The Ray AI Runtime (AIR) provides a unified API for all other Ray ML libraries and was built with the needs of data scientists in mind.

On top of that, Ray's ecosystem has many extensions, integrations and backends that we'll look more into later on. Maybe you can already spot a few tools you know and like in Figure 1-4?

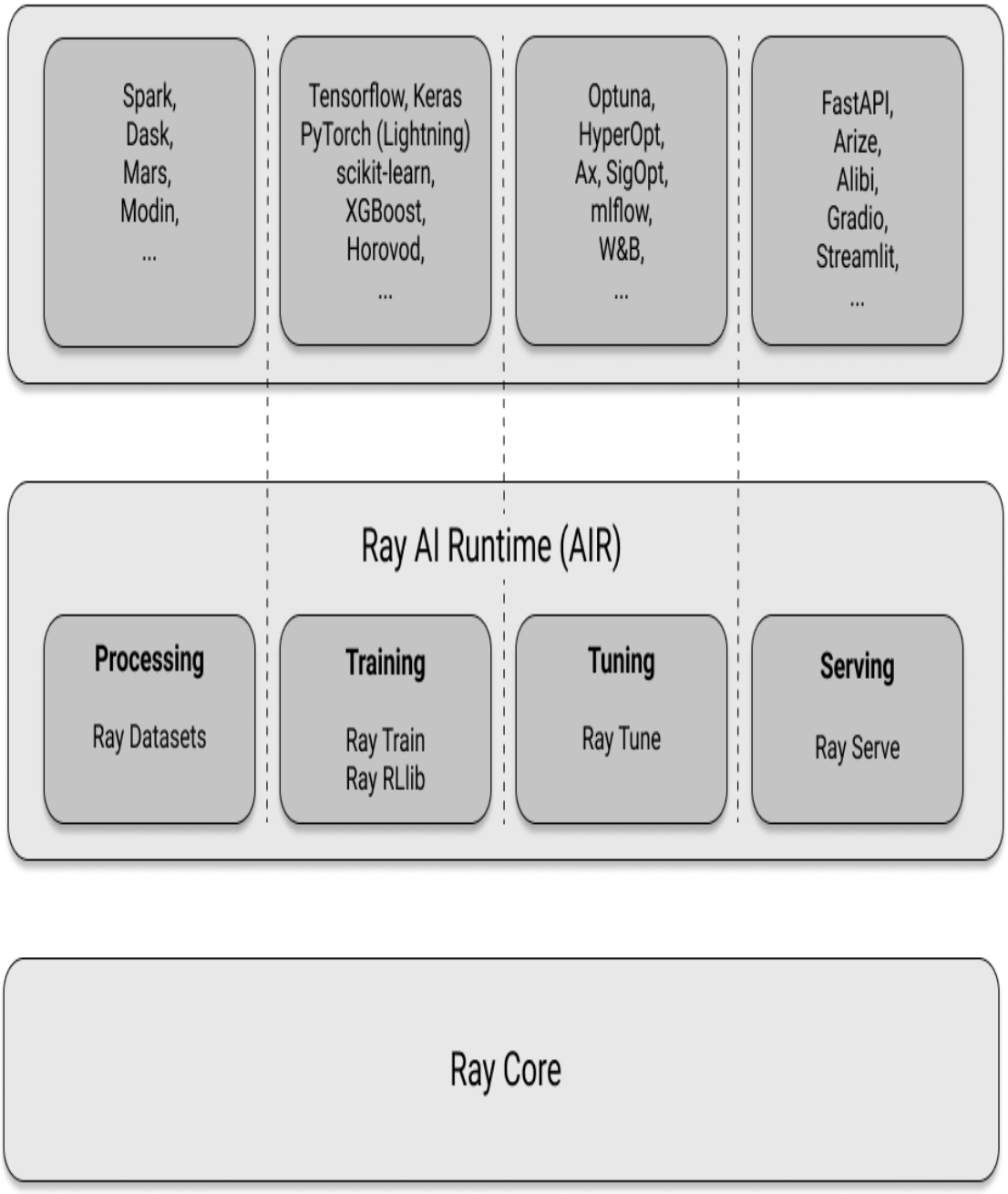


Figure 1-4. Ray in three layers. Its core API sits at the center, surrounded by the libraries RLlib, Tune, Ray Train, Ray Serve, Ray Datasets and the many third-party integrations we can't all list here.

---

1 By Python-first we mean that all higher level libraries are written in Python, and that the development of new features is driven by the needs of the

Python community. Having said that, Ray has been designed to support multiple language bindings and e.g. comes with a Java API. So, it's not out of the question that Ray might support other languages that are important to the data science exosystem.

2 Moore's Law held for a long time, but there might be signs that it's slowing down. Some even say it's dead. We're not here to argue these points. What's important is not that our computers generally keep getting faster, but the relation to the amount of compute we need.

3 There are many basic to sophisticated ways to speeding up ML training. For instance, we'll spend considerable amount of time elaborating of distributed data processing in Chapter 6 and distributed model training in Chapter 7.

4 Anyscale, the company behind Ray, is building a managed Ray platform and offers hosted solutions for your Ray applications.

5 This might sound drastic, but it's not a joke. To name just one example, in March 2021 a French data center powering millions of websites burnt down completely, which you can read about in this article. If your whole cluster burns down, we're afraid Ray can't help you.

6 This is a Python book, so we'll exclusively focus on it. But you should at least know that Ray also has a Java API, which at this point is less mature than its Python equivalent.

7 One of the reasons there are so many libraries built on top of Ray Core is that it's so lean and straightforward to reason about. And it's one of the goals of this book is to inspire you to write your own applications, or even libraries, with Ray.

8 We generally introduce dependencies in this book only when we need them, which should make it easier to follow along. In contrast, the notebooks you find on GitHub give you the option to install all dependencies upfront, so that you can focus on running the code instead.

9 At the time of this writing, there's no Python `3.10` support for Ray yet, so sticking to a version between `3.7` and `3.9` should work best to follow this book.

10 There are other means of interacting with Ray clusters, such as the Ray Jobs CLI

11 We never liked the categorization of data science as an intersection of disciplines, like maths, coding and business. Ultimately, that doesn't tell you what practitioners *do*.

12 As a fun exercise, I recommend reading Paul Graham's famous "Hackers and Painters" essay on this topic and replace "computer science" with "data science". What would hacking 2.0 be?

13 If you want to understand more about the holistic view of the data science process when building machine learning applications, the book Building Machine Learning Powered Applications is dedicated to it entirely.

14 In Chapter 6 we will introduce you to the fundamentals of what makes Ray Datasets work, including its use of Arrow. For now, we want to focus on its API and concrete usage patterns.

15 We'll elaborate more on this in later chapters, specifically in Chapter 6, but note that Ray Datasets is not meant as a general-purpose data processing library. Tools such as Spark have more mature and optimised support for large-scale data processing.

16 If you're on a Mac, you'll have to install `tensorflow-macos`. In general, if you encounter any issues installing Ray or its dependencies on your system, please refer to the installation guide.

17 Depending on the operating system you're using, you may need to install the Rust compiler first to make this work. For instance, on a Mac, you can install it with `brew install rust`.

18 Spark has been created by another lab in Berkeley, AMPLab. The internet is full of blog posts claiming that Ray should therefore be seen as a replacement of Spark. It's better to think of them as tools with different strengths that are both likely here to stay.

19 Before the deep learning framework Keras became an official part of TensorFlow, it started out as a convenient API specification for various lower-level frameworks such as Theano or CNTK. In that sense Ray RLlib has the chance to become "Keras for RL", and Ray Tune might just be "Keras for HPO". The missing piece for more adoption might just be a more elegant API for both.

Chapter 2. Getting Started with Ray Core

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

For a book on distributed Python, it's not without a certain irony that Python on its own is largely ineffective for distributed computing. Its interpreter is effectively single threaded. For instance, this makes it difficult to leverage multiple CPUs on the same machine, let alone a whole cluster of machines, using plain Python. That means you need extra tooling, and luckily the Python ecosystem has some options for you. Libraries like `multiprocessing` can help you distribute work on a single machine,[1] but not beyond.

Seen as a Python library, the Ray Core API is powerful enough to make general distributed programming more accessible to the Python community as a whole. By way of analogy, some companies get by with deploying pretrained ML models for their use cases, but that strategy is not always effective. It's often inevitable to train custom models to be successful. In the same way, your distributed workloads *might* just fit into the (potentially limiting) programming model of existing frameworks, but Ray Core can unlock the full spectrum of building distributed applications, due to its generality.[2] As it is so fundamental, we dedicate this whole chapter to the basics of Ray Core and spend all of Chapter 3 on building an interesting application with the Core API. This way you're equipped with

practical knowledge about Ray Core and can utilize it in later chapters, and your own projects.

In this chapter you'll understand how Ray core handles distributed computing by spinning up a local cluster, and you'll learn how to use Ray's lean and powerful API to parallelize some interesting computations. For instance, you'll build an example that runs a data-parallel task efficiently and asynchronously on Ray, in a convenient way that's not easily replicable with other tooling. We discuss how *tasks* and *actors* work as distributed versions of functions and classes in Python. You'll also learn how to put *objects* to Ray's object store, and how to retrieve them. We give you concrete examples for these three fundamental concepts (tasks, actors and objects), using just six basic API calls of the Ray Core API. Lastly, we'll also discuss the system components underlying Ray and what its architecture looks like. In other words, in this chapter we'll give you a look under the hood of Ray's engine.

An Introduction to Ray Core

The bulk of this chapter is an extended Ray Core example that we'll build together. Many of Ray's concepts can be explained with a good example, so that's exactly what we'll do. As before, you can follow this example by typing the code yourself (which is highly recommended), or by following the notebook for this chapter. In any case, please make sure you have Ray installed, for instance with `pip install ray`.

In Chapter 1 we showed you how start a local cluster simply by calling `import ray` and then initializing it with `ray.init()`. After running this code you will see output of the following form. We omit a lot of information in this example output, as that would require you to understand more of Ray's internals first.

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

This output indicates that your Ray cluster is up and running. As you can see from the first line of the output, Ray comes with its own, pre-packaged dashboard.[3] You can check it out at *http://127.0.0.1:8265*, unless your output shows a different port. If you want you can take your time to explore the dashboard for a little. For instance, you should see all your CPU cores listed and the total utilization of your (trivial) Ray application. To see the resource utilization of your Ray cluster in Python, you can simply call `ray.cluster_resources()`, which on my computer returns the following output:

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

You'll need a running Ray cluster to run the examples in this chapter, so make sure you've started one before continuing. The goal of this section is to give you a quick introduction to the Ray Core API, which we'll simply refer to as the Ray API from now on.

As a Python programmer, the great thing about the Ray API is that it hits so close to home. It uses familiar concepts such as decorators, functions and classes to provide you with a fast learning experience. The Ray API aims to provide a universal programming interface for distributed computing. That's certainly no easy feat, but I think Ray succeeds in this respect, as it provides you with good abstractions that are intuitive to learn and use. Ray's engine does all the heavy lifting for you in the background. This design philosophy is what enables Ray to be used with existing Python libraries and systems.

Note that the reason we start out with Ray Core in this book is that we believe that it has massive potential to make distributed computing more accessible. In essence, this chapter is all about getting a peak behind the curtains of what makes Ray work so well and how you can pick up its fundamentals. If you're a less experienced Python programmer or just want to focus on higher level tasks, Ray Core might take some getting used to.[4] Having said that, we emphatically recommend learning the Ray Core API, as it's a great way to get into distributed computing with Python.

A First Example Using the Ray API

To give you an example, take the following function which retrieves and processes data from a database. Our dummy `database` is a plain Python list containing the words of the title of this book. We act as if retrieving an individual `item` from this database and further processing it is expensive by letting Python `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 A dummy database containing string data with the title of this book.

❷ We emulate a data-crunching operation that takes a long time.

Our database has eight items in total. If we were to retrieve all items sequentially, how long should that take? For the item with index `5` we

wait for half a second (`5 / 10.`) and so on. In total, we can expect a runtime of around `(0+1+2+3+4+5+6+7)/10. = 2.8` seconds. Let's see if that's what we actually get:

```
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 We use a list comprehension to retrieve all eight items.
❷ Then we unpack the data to print each item on its own line.

If you run this code, you should see the following output:

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

There's a little overhead that brings the total runtime to `2.82` seconds. On your end this might be slightly less, or much more, depending on your computer. The important take-away is that our naive Python implementation is not able to run this function in parallel.

This may not come as a surprise to you, but you could at least have suspected that Python list comprehensions are more efficient in that regard. The runtime we got is pretty much the worst case scenario, namely the `2.8` seconds we calculated prior to running the code. If

you think about it, it might even be a bit frustrating to see that a program that essentially sleeps most of its runtime is that slow overall. Ultimately you can blame the *Global Interpreter Lock* (GIL) for that, but it gets enough of it already.

PYTHON'S GLOBAL INTERPRETER LOCK

The Global Interpreter Lock or GIL is undoubtedly one of the most infamous features of the Python language.[5] In a nutshell it's a lock that makes sure only one thread on your computer can ever execute your Python code at a time. If you use multi-threading, the threads need to take turns controlling the Python interpreter.

The GIL has been implemented for good reasons. For one, it makes memory management that much easier in Python. Another key advantage is that it makes single-threaded programs quite fast. Programs that primarily use lots of system input and output (we say they are I/O-bound), like reading files or databases, benefit as well. One of the major downsides is that CPU-bound programs are essentially single-threaded. In fact, CPU-bound tasks might even run *faster* when not using multi-threading, as the latter incurs write-lock overheads on top of the GIL.

Given all that, the GIL might somewhat paradoxically be one of the reasons for Python's popularity, if you believe Larry Hastings. Interestingly, Hastings also led (unsuccessful) efforts to remove it in a project called *GILectomy*, which is exactly the kind of complicated surgery that it sounds like. The jury is still out, but Sam Gross might just have found a way to remove the GIL in his `nogil` branch of Python 3.9. For now, if you absolutely have to work around the GIL, consider using an implementation different from CPython. CPython is Python's standard implementation, and if you don't know that you're using it, you're definitely using it. Implementations like Jython, IronPython or PyPy don't have a GIL, but come with their own drawbacks.

Functions and Remote Ray Tasks

It's reasonable to assume that such a task can benefit from parallelization. Perfectly distributed, the runtime should not take much longer than the longest subtask, namely `7/10. = 0.7` seconds. So, let's see how you can extend this example to run on Ray. To do so, you start by using the `@ray.remote` decorator as follows:

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

❶ With just this decorator we make any Python function a Ray task.
❷ All else remains unchanged. `retrieve_task` just passes through to `retrieve`.

In this way, the function `retrieve_task` becomes a so-called Ray task. In essence, a Ray task is a function that gets executed on a different process that it was called from, potentially on a different machine.

That's an extremely convenient design choice, as you can focus on your Python code first, and don't have to completely change your mindset or programming paradigm to use Ray. Note that in practice you would have simply added the `@ray.remote` decorator to your original `retrieve` function (after all, that's the intended use of decorators), but we didn't want to touch previous code to keep things as clear as possible.

Easy enough, so what do you have to change in the code that retrieves the database entries and measures performance? It turns out, not much. Let's have a look at how you would do that:[6]

Example 2-1. Measuring performance of your Ray task.

```
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 To run `retrieve_task` on your local Ray cluster, you use `.remote()` and pass in your items as before. Each task returns an object.

❷ To get back actual data, and not just Ray object references, you use `ray.get`.

Did you spot the differences? You have to execute your Ray task remotely using a `.remote()` call.[7] When tasks get executed remotely, even on your local cluster, Ray does so *asynchronously*. The list items in `object_references` in the last code snippet do not contain the results directly. In fact, if you check the Python type of the first item with `type(object_references[0])` you'll see that it's in fact an `ObjectRef`. These object references correspond to *futures* which you need to ask the result of. This is what the call to `ray.get(...)` is for. Whenever you call `remote` on a Ray task, it will immediately return one or more object references. In fact, while we introduced `ray.put(...)` first, you should consider Ray tasks as the primary way of creating objects. In the next section we'll show you an example that chains multiple tasks together and lets Ray take care of passing and resolving the objects between them.

We still want to work more on this example,[8] but let's take a step back here and recap what we did so far. You started with a Python function and decorated it with `@ray.remote`. This made your function a Ray task. Then, instead of calling the original function in your code straight-up, you called `.remote(...)` on the Ray task. The last step was to `.get(...)` the results back from your Ray cluster. I think this procedure is so intuitive that I'd bet you could already create your own Ray task from another function without having to look back at this example. Why don't you give it a try right now?

Coming back to our example, by using Ray tasks, what did we gain in terms of performance? On my machine the runtime clocks in at

`0.71` seconds, which is just slightly more than the longest subtask, which comes in at `0.7` seconds. That's great and much better than before, but we can further improve our program by leveraging more of Ray's API.

Using the object store with `put` and `get`

One thing you might have noticed is that in the definition of `retrieve` we *directly* accessed items from our `database`. Working on a local Ray cluster this is fine, but imagine you're running on an actual cluster comprising several computers. How would all those computers access the same data? Remember from Chapter 1 that in a Ray cluster there is one head node with a driver process (running `ray.init()`) and many worker nodes with worker processes executing your tasks. By default, Ray will create as many worker processes as there are CPU cores on your machine. Our `database` is currently defined on the driver only, but the workers running your tasks need to have access to it to run the `retrieve` task. Luckily, Ray provides an easy way to share *objects* between the driver and workers (or between workers). You can simply use `put` to place your data into Ray's *distributed object store*. In our definition of `retrieve_task` we explicitly pass in a `db` argument, to which we later on will pass our `db_object_ref` object.

```
db_object_ref = ray.put(database)  ❶


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

❶ You `put` your `database` into the object store and receive a reference to it. This way we can explicitly pass this reference to our Ray task later.

❷ The Ray task `retrieve_task` takes the object reference as an argument.

By using the object store this way, you can let Ray handle data access across the whole cluster. We'll talk about how exactly values are passed between nodes and within workers when talking about Ray's infrastructure. While the interaction with the object store requires some overhead, it gives you performance gains when working with larger, more realistic datasets. For now, the important part is that this step is essential in a truly distributed setting. If you like, try to re-run Example 2-1 with this new `retrieve_task` function and confirm that it still runs, as expected.

Using Ray's `wait` function for non-blocking calls

Note how in Example 2-1 we used `ray.get(object_references)` to access results. This call is *blocking*, which means that our driver has to wait for all the results to be available. That's not a big deal in our case, the program now finishes in under a second. But imagine processing of each database item would take several minutes. In that case you would want to free up the driver process for other tasks, instead of sitting idly by. Also, it would be great to process results as they come in (some finish much quicker than others), rather than waiting for all items to be processed. One more question to keep in mind is what happens if one of the database items can't be retrieved as expected. Let's say there's a deadlock somewhere in the database connection. In that case, the driver will simply hang and never retrieve all items. For that reason it's a good idea to work with reasonable timeouts. In our scenario, let's say we don't want to wait longer than ten times the longest data retrieval task before stopping the task. Here's how you can do that with Ray by using `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 We run `remote` on our `retrieve_task` and pass the respective `item` we want to retrieve, and the object reference to our database.

❷ Instead of blocking, we loop through unfinished `object_references`.

❸ We asynchronously `wait` for finished data with a reasonable `timeout`. `object_references` gets overridden here, to prevent an infinite loop.

❹ We print results as they come in, namely in blocks of two.

❺ Then we `append` new `data` to the `all_data` until finished.

As you can see `ray.wait` returns two arguments, namely finished values and futures that still need to be processed. We use the `num_returns` argument, which defaults to `1`, to let `wait` return whenever a new pair of database items is available. On my laptop this results in the following output:

```
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')
```

Note how in the `while` loop, instead of just printing results, we could have done many other things, like starting entirely new tasks on other workers with the values already retrieved up to this point.

Handling task dependencies

So far our example program has been fairly easy on a conceptual level. It consists of a single step, namely retrieving a bunch of database items. Now, imagine that once your data is loaded you want to run a follow-up processing task on it. To be more concrete, let's say we want to use the result of our first retrieve task to query other, related data (pretend that you're querying data from a different table in the same database). The following code sets up such a task and runs both our `retrieve_task` and `follow_up_task` consecutively.

Example 2-2. Running a follow-up task that depends on another Ray task.

```
@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]  ❹

result = [print(data) for data in ray.get(follow_up_refs)]
```

❶ Using the result of `retrieve_task` we compute another Ray task on top of it.

❷ Leveraging the `original_item` from the first task, we `retrieve` more data.

❸ Then we return both the original and the follow-up data.

❹ We pass the object references from the first task into the second task.

Running this code results in the following output:

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

If you don't have a lot of experience with asynchronous programming, you might not be impressed by Example 2-2. But I hope to convince you that it's at least a bit surprising that this code snippet runs at all.[9] So, what's the big deal? After all, the code reads like regular Python - a function definition and a few list comprehensions. The point is that the function body of `follow_up_task` expects a Python `tuple` for its input argument `retrieve_result`, which we unpack in the first line of the function definition.

But by invoking `[follow_up_task.remote(ref) for ref in retrieve_refs]` we do *not* pass in tuples to the follow-up task at all. Instead, we pass in Ray *object references* with `retrieve_refs`. What happens under the hood is that Ray knows that `follow_up_task` requires actual values, so internally in this task it will call `ray.get` to resolve the futures.[10] Ray builds a dependency graph for all tasks and executes them in an order that respects the dependencies. You do not have to tell Ray explicitly when to wait for a previous task to finish, it will infer that information for you. This is also shows you a powerful feature of the Ray object store: if intermediate values are large, you can avoid copying them back to the driver. You can just pass your object references to the next task and let Ray handle the rest.

The follow-up tasks will only be scheduled once the individual retrieve tasks have finished. If you ask me, that's an incredible feature. In fact, if we had called `retrieve_refs` something like `retrieve_result`, you may not even have noticed this important detail. That's by design. Ray wants you to focus on your work, not on

the details of cluster computing. In figure Figure 2-1 you can see the dependency graph for the two tasks visualized.

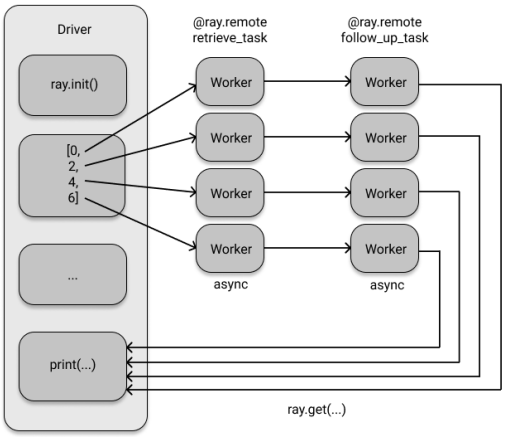


Figure 2-1. Running two dependent tasks asynchronously and in parallel with Ray.

If you feel like it, try to rewrite Example 2-2 so that it explicitly uses `get` on the first task before passing values into the follow-up task. That does not only introduce more boilerplate code, but it's also a bit less intuitive to write and understand.

From classes to actors

Before wrapping up this example, let's discuss one more important concept of Ray Core. Notice how in our example everything is essentially a function. We just used the `ray.remote` decorator to make some of them remote functions, and other than that simply used plain Python.

Let's say we wanted to track how often our `database` has been queried. Sure, we could simply count the results of our retrieve tasks, but is there a better way to do this? We want to track this in a "distributed" way that will scale. For that, Ray has the concept of *actors*. Actors allow you to run *stateful* computations on your cluster. They can also communicate between each other.[11] Much like Ray tasks were simply decorated functions, Ray actors are decorated

Python classes. Let's write a simple counter to track our database calls.

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

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

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

❶ We can make any Python class a Ray actor by using the same `ray.remote` decorator as before.

This `DataTracker` class is already an actor, since we equipped it with the `ray.remote` decorator. This actor can track state, here just a simple counter, and its methods are Ray tasks that get invoked precisely like we did with functions before, namely using `.remote()`. Let's see how we can modify our existing `retrieve_task` to incorporate this new actor.

```
@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 We pass in the `tracker` actor into this task.
❷ The `tracker` receives an `increment` for each call.
❸ We instantiate our `DataTracker` actor by calling `.remote()` on the class.
❹ The actor gets passed into the retrieve task.
❺ Afterwards we can get the `counts` state from our `tracker` from another remote invocation.

Unsurprisingly, the result of this computation is in fact `8`. We didn't need actors to compute this, but I hope you can see how useful it can be to have a mechanism to track state across the cluster, potentially spanning multiple tasks. In fact, we could pass our actor into any dependent task, or even pass it into the constructor of yet another actor. There is no limitation to what you can do, and it's this flexibility that makes the Ray API so powerful. It's also worth mentioning that it's not very common for distributed Python tools to allow for stateful computations like this. This feature can come in very handy, especially when running complex distributed algorithms, for instance when using reinforcement learning. This completes our extensive first Ray API example. Let's see if we can concisely summarize the Ray API next.

> **NOTE**
>
> In this introduction by example we focused a lot on Ray tasks and actors as distributed versions of Python functions and classes. But *objects* are also first-class citizens in Ray Core and should be seen as equal in status to tasks and actors. The object store is a central component of Ray, and we'll discuss it in more detail later.

An Overview of the Ray Core API

If you recall what exactly we did in the previous example, you'll notice that we used a total of just six API methods.[12] You used

`ray.init()` to start the cluster and `@ray.remote` to turn functions and classes into tasks and actors. Then we used `ray.put()` to pass values into Ray's object store and `ray.get()` to retrieve objects from the cluster. Finally, we used `.remote()` on actor methods or tasks to run code on our cluster, and `ray.wait` to avoid blocking calls.

While six API methods might not seem like much, those are the only ones you'll likely ever care about when using the Ray API.[13] Let's briefly summarize them in a table, so you can easily reference them in the future.

Table 2-1. The six major API methods of

RayCore.

API call	Description
`ray.init()`	Initializes your Ray cluster. Pass in an `address` to connect to an existing cluster.
`@ray.remote`	Turns functions into tasks and classes into actors.
`ray.put()`	Puts values into Ray's object store.
`ray.get()`	Gets valuyes from the object store. Returns the values you've `put` there or that were computed by a task or actor.
`.remote()`	Runs actor methods or tasks on your Ray cluster and is used to instantiate actors.
`ray.wait()`	Returns two lists of object references, one with finished tasks we're waiting for and one with unfinished tasks.

Now that you've seen the Ray API in action, let's spend some time on its system architecture.

Understanding Ray System Components

You've seen how the Ray API can be used and understand the design philosophy behind Ray. Now it's time to get a better

understanding of the underlying system components. In other words, how does Ray work and how does it achieve what it does?

Scheduling and Executing Work on a Node

You know that Ray clusters consist of nodes. We'll first look at what happens on individual nodes, before we zoom out and discuss how the whole cluster interoperates.

As we've already discussed, a worker node consists of several worker processes or simply workers. Each worker has a unique ID, an IP address and a port by which they can be referenced. Workers are called as they are for a reason, they're components that blindly execute the work you give them. But who tells them what to do and when? A worker might be busy already, it may not have the proper resources to run a task (e.g. access to a GPU), and it might not even have the values it needs to run a given task. On top of that, workers have no knowledge of what happens before or after they've executed their workload, there's no coordination.

To address these issues, each worker node has a component called *Raylet*. Think of Raylets as the smart components of a node, which manage the worker processes. Raylets are shared between jobs and consist of two components, namely a *task scheduler* and an *object store*.

Let's talk about object stores first. In the running example in this chapter we've already used the concept of an object store loosely, without explicitly specifying it. Each node of a Ray cluster is equipped with an object store, within that node's Raylet, and all object stored collectively form the distributed object store of a cluster. The object store manages a *shared pool of memory* across workers on the same node and ensures that workers can access objects that were created on a different node. The object store is implemented in Plasma, which now belongs to the Apache Arrow project. Functionally, the object store takes care of memory management

and ultimately makes sure workers have access to the objects they need.

The second component of a Raylet is its scheduler. The scheduler takes care of *resource management*, among other things. For instance, if a task requires access to four CPUs, the scheduler needs to make sure it can find a free worker process that it can grant access to said resources. By default, the scheduler knows about and acquires information about the number of CPUs and GPUs, as well as the amount of memory available on its node. If a scheduler can't provide the required resources, it simply can't schedule execution of a task right away and needs to queue it. The scheduler limits which tasks are running concurrently to make sure that you don't run out of physical resources.

Apart from resources, the other requirement the scheduler takes care of is *dependency resolution*. That means it needs to ensure that each worker has all the objects it needs to execute a task in the local object store. For that to work, the scheduler will first resolve local dependencies by looking up values in its object store. If the required value is not available on this node's object store, the scheduler will have to communicate with other nodes (we'll tell you how in a bit) and pull in remote dependencies. Once the scheduler has ensured enough resources for a task, resolved all needed dependencies, and found a worker for a task, it can schedule said task for execution.

Task scheduling is a very difficult topic, even if we're only talking about single nodes. We think you can easily imagine scenarios in which an incorrectly or naively planned task execution can "block" downstream tasks because there are not enough resources left. Especially in a distributed context, assigning work like this can be become tricky very quickly.

Fault Tolerance and Ownership

Now that you know about Raylets, let's briefly come back to worker processes and wrap up the discussion by briefly explaining how Ray

can recover from failures and the concepts needed to do so.

In short, workers store metadata for all tasks invoked by them and the object references returned by those tasks. This concept, called *ownership*, means that the process that generates an object reference is also responsible for its resolution. In other words, each worker process "owns" the tasks it submits, which includes proper execution and ensuring availability of results. Worker processes need to track what they own, for instance in case of failures, which is why they have a so-called *ownership table*. This way, if a task fails and needs to be recomputed, the worker already owns all the information it needs to do so.[14]

To give you a concrete example of an ownership relationship, as opposed to the concept of dependency discussed earlier, let's say we have a program that starts a simple task and internally calls another task:

```
@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)
```

Let's quickly analyse ownership and dependency for this example. We defined two tasks in `task` and `task_owned`, and we have three variables in total, namely `val`, `res` and `res_owned`. Our main program defines both `val` (which puts `"value"` into the object store) and `res`, and it also calls `task`. In other words, the driver *owns* `task`, `val` and `res` according to Ray's ownership definition. In

contrast, `res` depends on `task`, but there's no ownership relationship between the two. When `task` gets called, it takes `val` as a dependency. It then calls `task_owned` and assigns `res_owned`, and hence owns them both. Lastly, `task_owned` itself does not own anything, but certainly `rew_owned` depends on it.

To sum up this discussion about worker nodes, figure Figure 2-2 gives you an overview of all involved components.

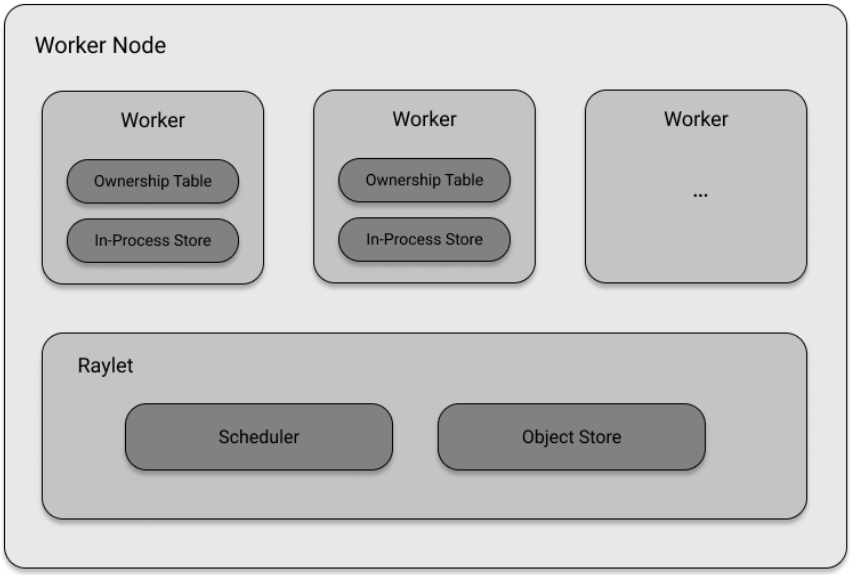


Figure 2-2. The system components comprising a Ray worker node.

The Head Node

We've already indicated in Chapter 1 that each Ray cluster has one special node called head node. So far you know that this node has a driver process.[15] Drivers can submit tasks themselves, but can't execute them. You also know that the head node can have some worker processes, which is important to be able to run local clusters consisting of a single node.

The head node is identical to other worker nodes, but it additionally runs processes responsible for cluster management such as the autoscaler (that we cover in Chapter 9) and a component called *Global Control Service* (GCS). This is an important component that

carries global information about the cluster. The GCS is a key-value store that stores information such as system-level metadata. For instance, it has a table with heart beat signals for each Raylet, to ensure they are still reachable. Raylets, in turn, send heart beat signals to the GCS to indicate that they are alive. The GCS also stores the locations of Ray actors. The ownership model just discussed tells us that all object information is stored at their owner worker process, which avoids making the GCS a bottleneck.

Distributed Scheduling and Execution

Let's briefly talk about cluster orchestration and how nodes manage, plan and execute tasks. When talking about worker nodes, we've indicated that there are several components to distributing workloads with Ray. Here's an overview of the steps and intricacies involved in this process.

- Distributed memory: The object stores of individual Raylets manages memory on a node. But sometimes objects needs to be transferred between nodes, which is called *distributed object transfer*. This is needed for remote dependency resolution, so that workers have the objects they need to run tasks.
- Communication: Most of the communication in a Ray cluster, such as object transfer, takes place via *gRPC*.
- Resource management and fulfillment: On a node, Raylets are responsible to grant resources and *lease* worker processes to task owners. All schedulers across nodes form the distributed scheduler, which effectively means that nodes can schedule tasks on other nodes. Through communication with the GCS, local schedulers know about other nodes' resources.
- Task execution: Once a task has been submitted for execution, all its dependencies (local and remote data) need to be

resolved, e.g. by retrieving large data from the object store, before execution can begin.

If the last few sections seem a bit involved technically, that's because they are. In my view it's important to understand the basic patterns and ideas of the software you're using, but I'll admit that the details of Ray's architecture can be a bit tough to wrap your head around in the beginning. In fact, it's one of Ray's design principles to trade-off usability for architectural complexity. If you want to delve deeper into Ray's architecture, a good place to start is their architecture white paper.

To wrap things up, let's summarize all we know in a concise architecture overview with figure Figure 2-3:

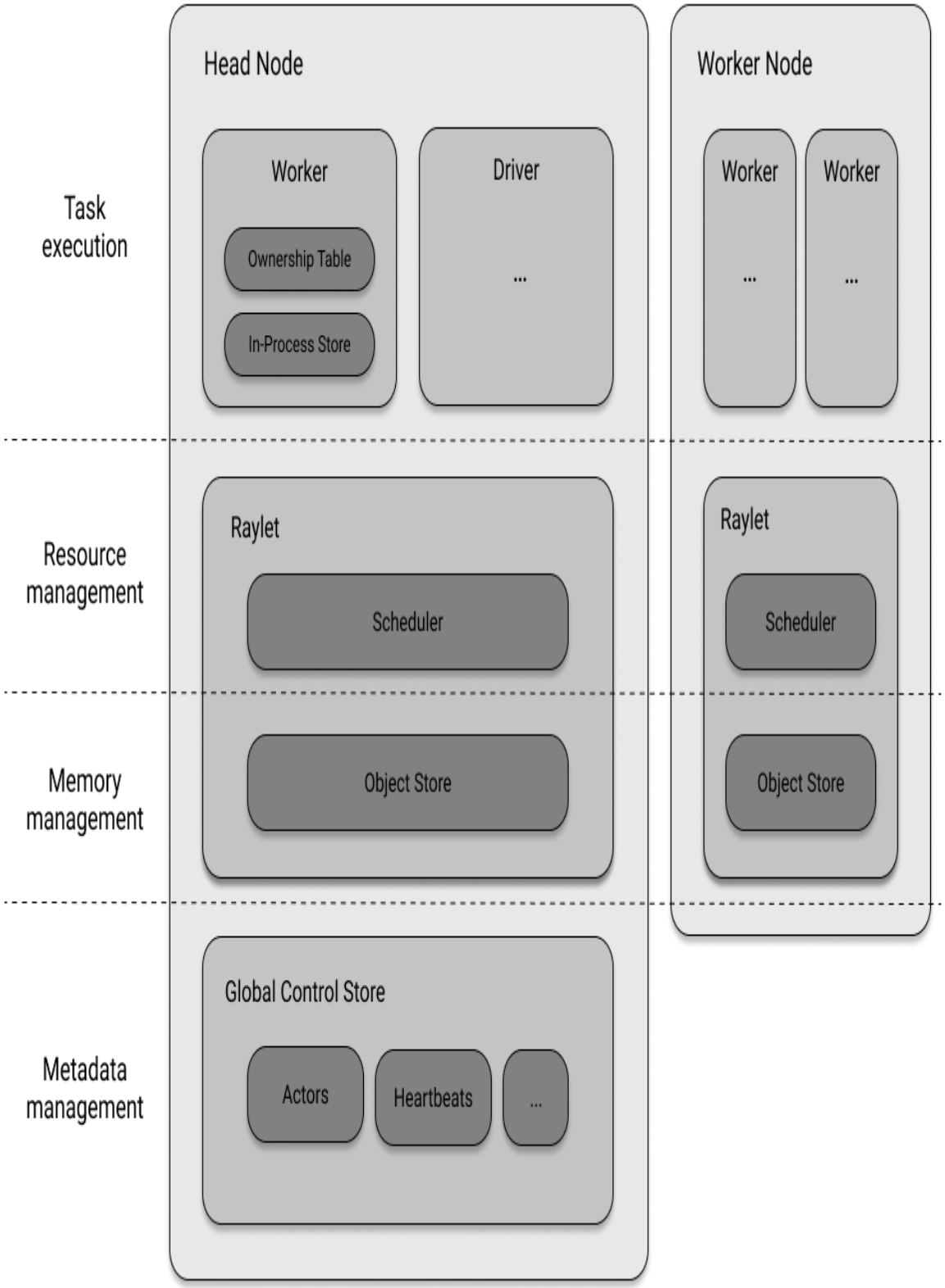


Figure 2-3. An overview of Ray's architectural components.

SYSTEMS RELATED TO RAY.

With the architecture and functionality of it in mind, how does Ray relate to other systems? We're not going into the details here, but just touch on the most important topics in broad strokes. Ray can be used as a parallelization framework for Python, and shares properties with tools like `celery` or `multiprocessing`. In fact, there's a drop-in replacement for the latter implemented in Ray. Ray is also related to data processing frameworks such as Spark, Dask, Flink or MARS. We'll explore this relationship in Chapter 11, when talking about Ray's ecosystem. As a distributed computing tool, Ray also has to deal with the problems of cluster management and orchestration, and we'll see how Ray does that in relation to tools like Kubernetes in Chapter 9. Since Ray is implementing the actor model of concurrency, it's also interesting to explore its relationship with frameworks like Akka. Lastly, since Ray banks on a performant, low-level API for communication, there's a certain relationship with high-performance computing (HPC) frameworks and communication protocols like the message passing interface (MPI).

Now that you've learned the basics of the Ray Core API and know the fundamentals of Ray's Cluster architecture, let's compute one more complex example together.

A Simple MapReduce Example with Ray

We can't let you go without discussing an example of one of the most important milestones in distributed computing in recent decades, namely *MapReduce*. Many successful big data technologies like Hadoop are based on this programming model, and it's worth revisiting it in the context of Ray. To keep things simple, we'll restrict our MapReduce implementation to a single use case, namely the task of counting word occurrences across several documents. This is an almost trivial task in single processing, but it becomes an interesting challenge once a massive corpus of

documents is involved, and you need multiple compute nodes to crunch the numbers.

Implementing a MapReduce word-count example might just be the most well-known example we have in distributed computing,[16] so it's worth knowing. If you don't know about this classical paradigm, it's based on three straightforward steps:

1. Take a set of documents and transform or "map" its elements (for instance the words contained in them) according to a function you provide. This *map phase* produces key-value pairs by design, in which a *key* represents a document element and a *value* is simply a metric you want to compute for that element. Since we're interested in counting words, whenever we encounter a `word` in a document, our *map function* will simply emit the pair `(word, 1)` to indicate that we found one occurrence of it.

2. The second step is to collect and group all the outputs of the map phase according to their key. Since we work in a distributed setup and the same key might be present on several compute nodes, this might require shuffling of data between nodes.[17] For that reason this step is often referred to as the *shuffle phase*. To give you an idea of what grouping might mean in our concrete use case, let's say we have a total of four `(word, 1)` occurrences produced in the map phase. The shuffle would then co-locate all occurrences of the same word on the same node.

3. The last step is to aggregate or "reduce" the elements from the shuffle step, which is why we refer to it as the *reduce phase*. Continuing with the example we laid out, we simply sum up all word occurrences on each node to get the final count. For instance, four occurrences of `(word, 1)` would be reduced to `word: 4`.

Evidently, MapReduce gets its name from the first and last of these three phases, but the second one is arguably just as important. While schematically these phases may look simple, their power lies in the fact that they can be massively parallelized across hundreds of machines. In figure Figure 2-4 we illustrate an example of applying the three MapReduce phases to a corpus of documents that has been distributed across three partitions:

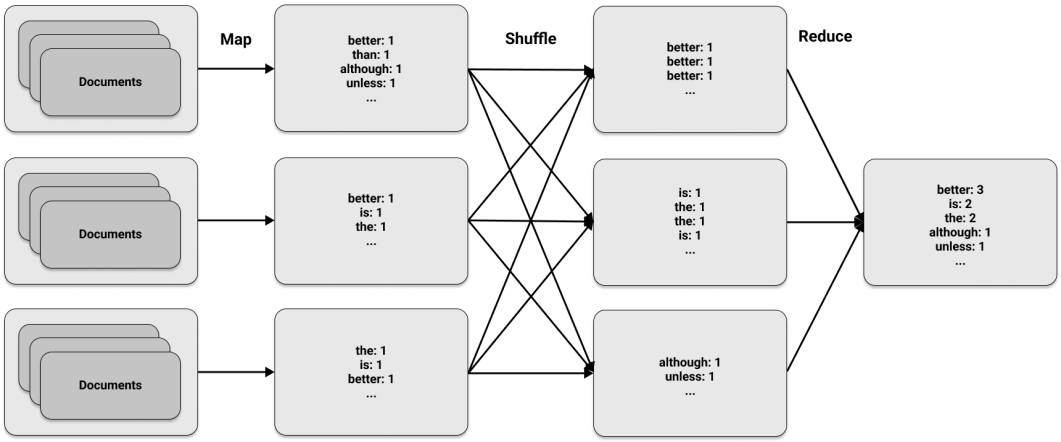


Figure 2-4. To run MapReduce on a distributed corpus of documents, we first map each document to a set of key-value pairs, then shuffle the results to ensure that all key-value pairs with the same key are on the same node, and finally reduce the key-value pairs to compute the final word counts.

Let's implement the MapReduce algorithm for our word-count use case in Python, and parallelize the computation using Ray. Let's first load example data, so that you get a better idea of what we're operating on:

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

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

```
    range(num_partitions)
]
```

❶ Our text corpus is the content of the Zen of Python.
❷ We split up the corpus into three partitions for demonstration purposes.

The data we're using is the so-called Zen of Python, a small set of guidelines by the Python community. The Zen is hidden in an "Easter egg" and gets printed when you type `import this` in a Python session. It's worth reading these guidelines as a Python programmer, but for this exercise we're only interested in counting the words they contain. Put simply, we load the Zen of Python, treat each line of it as a separate "document", and split it into three partitions.

To start off our implementation of MapReduce, we'll first cover the map phase and discuss how Ray can help us take care of shuffling the results of it.

Mapping and Shuffling Document Data

To define the map phase, we need a map function that we apply to each document. In our case we want to emit the pair `(word, 1)` for each `word` we find in a `document`. For simple text documents loaded as Python strings this looks as follows.[18]

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

Next, we want to apply this map function to a whole corpus of documents. We do this by making the following `apply_map` function a Ray task via the `@ray.remote` decorator. When we later call `apply_map`, we will apply it to three partitions (`num_partitions=3`) of document data, just like we indicated in

figure Figure 2-4. Note that `apply_map` will return three lists, one for each partition. As you will see in a moment, we do this so that Ray can automatically shuffle the results of the map phase to the right nodes for us.

```
import ray

@ray.remote
def apply_map(corpus, num_partitions=3):
    map_results = [list() for _ in range(num_partitions)]
❶
    for document in corpus:
        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 The Ray task `apply_map` returns one result for each data partition.

❷ We assign each `(word, 1)` pair to a partition by using the `ord` function to generate a `word_index`. This ensures that each occurrence of a `word` gets shuffled to the same partition.

❸ The pairs are then successively appended to the correct list.

For a text corpus that can be loaded on a single machine, this is of course overkill, and we could count the words straight-up. But in a distributed setting, in which we have to partition the data across several nodes, this map phase makes perfect sense.

To apply the map phase to our corpus of documents in parallel, we use a `remote` call on `apply_map` as we've done many times before in this chapter. The notable difference is that now we also instruct Ray to return three results (one for each partition) via the `num_returns` argument:

```
map_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"Mapper {i}, return value {j}:
    {result[:2]}")
```

❶ We use `options` to tell Ray to return `num_partitions` values.
❷ We then execute `apply_map` remotely.
❸ And we do so by iterating over each of the `partitions` we defined.
❹ We inspect the results for illustration purposes only. Normally you would not call `ray.get` yet.

If you run this code, you will see that each map phase result consists of three lists, of which we print the first two elements each.

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

As you will see, we can make it so that all pairs from the `j`-th return value end up on the same node for the reduce phase.[19] Let's discuss this phase next.

Reducing Word Counts

In the reduce phase we can now simply create a dictionary that sums up all word occurrences on each partition:

```
@ray.remote
def apply_reduce(*results):  ❶
    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  ❷

    return reduce_results
```

❶ We reduce the list of shuffled map-results.
❷ We iterate over each `result` obtained from the map phase and increase word counts by one for each occurrence of a word.

We can now collect the `j`-th return value from each mapper and pass it to the `j`-th reducer as follows. Note that we use a toy dataset here, but this code would scale to datasets that don't fit on a single machine. That's because we're passing Ray object references to the reducers, not the actual data. The map and reduce phases are Ray tasks that can be executed on any Ray cluster, and the shuffling of the data is handled by Ray as well.

```
outputs = []
for i in range(num_partitions):
    outputs.append(  ❶
        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()}  ❷

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

❶ First, we gather one output from each map task and supply it to `apply_reduce`.

❷ We can then collect all reduce-phase results in a single Python `count` dictionary.
❸ Lastly, we can print the sorted word count over the full corpus.

Running this example will yield the following output:

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

If you want a deep dive into making MapReduce tasks scale to multiple nodes with Ray, including detailed memory considerations, we recommend studying the excellent blog post on this topic.

The important part about this MapReduce example is to realize how *flexible Ray's programming model really is*. Surely, a production-grade MapReduce implementation takes a bit more effort. But being able to reproduce common algorithms like this one quickly goes a long way. Keep in mind that in the earlier phases of MapReduce, say around 2010, this paradigm was often the only thing you had to express your workloads. With Ray, a whole range of interesting distributed computing patterns become accessible to any intermediate Python programmer.[20]

Summary

You've seen the basics of the Ray API in action in this chapter. You know how to `put` values into the object store, and how to `get` them back. Also, you're familiar with declaring Python functions as Ray tasks with the `@ray.remote` decorator, and you know how to run them on a Ray cluster with the `.remote()` call. In much the same way, you understand how to declare a Ray actor from a Python

class, how to instantiate it and leverage it for stateful, distributed computations.

On top of that, you also know the basics of Ray clusters. After starting them with `ray.init(...)` you know that you can submit jobs consisting of tasks to your cluster. The driver process, sitting on the head node, will then distribute the tasks to the worker nodes. Raylets on each node will schedule the tasks and worker processes will execute them. You've also seen a quick implementation of the MapReduce paradigm with Ray as an example of a common pattern of building Ray applications.

This quick tour through Ray Core should get you started with writing your own distributed programs, and in the next chapter we'll test your knowledge by implementing a basic machine learning application together.

---

1 Note that Ray comes with a drop-in replacement for `multiprocessing` that you might find useful for certain workloads.

2 This represents a trade-off in terms of generality versus specialisation. By also providing specialised, yet interoperable libraries on top of the Core API, Ray provides tooling at various levels of abstraction.

3 The dashboard gets redesigned as we write these lines. As much as we'd like to show you screenshots of it and walk you through it, you'll have to discover it for yourself for now.

4 In case you fall into these categories, it might be comforting to hear that many data scientists rarely use Ray Core directly. Instead, they work directly with Ray's higher level libraries like Datasets, Train or Tune.

5 I still don't know how to pronounce this acronym, but I get the feeling that the same people who pronounce GIF like "giraffe" also say GIL like "guitar". Just pick one, or spell it out.

6 Strictly speaking, this first example is a bit of an anti-pattern, as you should not normally share mutable state across Ray tasks through global variables. Having said that, for this toy data example you shouldn't overthink this part. Rest assured that we'll show you a better way of doing things in the next section.

7 This book is geared towards data science practitioners, so we can't discuss the conceptual details of Ray's architecture here. If you're curious and want to learn more about how Ray tasks are executed, check out the Ray architecture whitepaper.

8 This example has been adapted from Dean Wampler's fantastic report "What is Ray?".

9 According to Clarke's third law any sufficiently advanced technology is indistinguishable from magic. For me, this example has a bit of magic to it.

10 The exact same thing already happened earlier, when we passed an object reference to the remote call of `retrieve_task` and then directly accessed the respective items of the database `db` there. We didn't want to distract you too much from the main point of the example earlier.

11 The actor model is an established concept in computer science, which you can find implemented e.g. in Akka or Erlang. However, the history and specifics of actors are not relevant to our discussion.

12 To paraphrase Alan Kay, to get simplicity, you need to find slightly more sophisticated building blocks. In my eyes, the Ray API does just that for distributed Python.

13 You can check out the API reference to see that there are in fact quite a bit more methods available. At some point you should invest in understanding the arguments of `init`, but all other methods likely won't be of interest to you, if you're not an administrator of your Ray cluster.

14 This is an extremely limited description of how Ray handles failures in general. After all, just having all the information to recover does not tell you how to do so. We can yet again only refer you to the architecture whitepaper for an in-depth discussion on this topic.

15 In fact, it could have multiple drivers, but this is inessential for our discussion. Also, starting a single driver on the head node is the most common, but driver processes can also be started on any node in the cluster and there can be multiple drivers on a single cluster.

16 It's a *drosophila melanogaster* of sorts, not unlike computing a classifier on the ubiquitous MNIST data set.

17 In general, a shuffle is any operation that requires redistributing data across its partitions. Shuffles can be quite costly. If your map phase operates on `N` partitions, it will produce `N * N` results that need to be shuffled.

18 Note the usage of `yield` in the map function. This is the quickest way of building a generator with the data we need in Python. You could also build and `return` a list of pairs, if that's clearer to you.

19 By construction, all same-key pairs will end up on the same node this way. For instance, note how in the sample output we printed the word `is` appears in the `0`-th return value of two of the mappers. All occurrences of `is` will end up on the same partition for the reduce phase.

20 You're encouraged to check out Ray's in-depth patterns and anti-patterns for both tasks and actors.

Chapter 3. Building Your First Distributed Application

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Now that you've seen the basics of the Ray API in action, let's build something more realistic with it. By the end of this chapter, you will have built a reinforcement learning (RL) problem from scratch, implemented your first algorithm to tackle it, and used Ray tasks and actors to parallelize this solution to a local cluster — all in less than 250 lines of code.

This chapter is designed to work for readers who don't have any experience with reinforcement learning. We'll work on a straightforward problem and develop the necessary skills to tackle it hands-on. Since chapter Chapter 4 is devoted entirely to this topic, we'll skip all advanced RL topics and language and just focus on the problem at hand. But even if you're a quite advanced RL user, you'll likely benefit from implementing a classical algorithm in a distributed setting.

This is the last chapter working *only* with Ray Core. I hope you learn to appreciate how powerful and flexible it is, and how quickly you can implement distributed experiments, that would otherwise take considerable efforts to scale.

Before we jump into any implementation, let's quickly talk about the paradigm of Reinforcement Learning in a bit more detail. Feel free to skip this section if you've worked with RL before.

Introducing Reinforcement Learning

One of my favorite apps on my phone can automatically classify or "label" individual plants in our garden. It works by simply showing it a picture of the plant in question. That's immensely helpful, as I'm terrible at distinguishing them all. (I'm not bragging about the size of my garden, I'm just bad at it.) In the last couple of years we've seen a surge of impressive applications like that.

Ultimately, the promise of AI is to build intelligent agents that go far beyond classifying objects. Imagine an AI application that not only knows your plants, but can take care of to them, too. Such an application would have to

- operate in dynamic environments (like the change of seasons),
- react to changes in the environment (like a heavy storm or pests attacking your plants),
- take sequences of actions (like watering and fertilizing plants),
- and accomplish long-term goals (like prioritizing plant health).

By observing its environment such an AI would also learn to explore the possible actions it could take and come up with better solutions over time. If you feel like this example is artificial or too far out, it's not difficult to come up with examples on your own that share all the above requirements. Think of managing and optimizing a supply chain, strategically restocking a warehouse considering fluctuating demands, or orchestrating the processing steps in an assembly line. Another famous example of what you could expect from an AI would be Stephen Wozniak's famous "Coffee Test". If you're invited to a friend's house, you can navigate to the kitchen, spot the coffee

machine and all necessary ingredients, figure out how to brew a cup of coffee, and sit down to enjoy it. A machine should be able to do the same, except the last part might be a bit of a stretch. What other examples can you think of?

You can frame all the above requirements naturally in a subfield of machine learning called reinforcement learning (RL).[1] For now, it's enough to understand that it's about agents interacting with their environment by observing it and emitting actions. In RL, agents evaluate their environments by attributing a reward (e.g., how healthy is my plant on a linear scale). The term "reinforcement" comes from the fact that agents will hopefully learn to seek out behaviour that leads to good outcomes (high reward), and shy away from punishing situations (low or negative reward).

The interaction of agents with their environment is usually modeled by creating a computer simulation of it (although sometimes that's not feasible). So, let's build an example of such a simulation with agents acting in their environments together to give you an idea of what this looks like in practice.

Setting Up a Simple Maze Problem

As with the chapters before, I encourage you to code this chapter with me and build this application together as we go. In case you don't want to do that, you can also simply follow the notebook for this chapter.

To give you an idea, the app we're building is structured as follows:

- You implement a simple 2D-maze game in which a single player can move around in the four major directions.
- You initialize the maze as a `5x5` grid to which the player is confined.

- One of the `25` grid cells is the “goal” that a player called the “seeker” must reach.
- Instead of hard-coding a solution, you will employ a reinforcement learning algorithm, so that the seeker learns to find the goal.
- This is done by repeatedly running simulations of the maze, rewarding the seeker for finding the goal and smartly keeping track of which decisions of the seeker worked and which didn’t.
- As running simulations can be parallelized and our RL algorithm can also be trained in parallel, we utilize the Ray API to parallelize the whole process.

We’re not quite ready to deploy this application on an actual Ray cluster comprised of multiple nodes just yet, so for now we’ll continue to work with local clusters. If you’re interested in infrastructure topics and want to learn how to set up Ray clusters, jump ahead to Chapter 9. In any case, make sure you have Ray installed with `pip install ray`.

Let’s start by implementing the 2D maze we just sketched. The idea is to implement a simple grid in Python that spans a 5x5 grid starting at `(0, 0)` and ending at `(4, 4)` and properly define how a player can move around the grid. To do this, we first need an abstraction for moving in the four cardinal directions. These four actions, namely moving up, down, left, and right, can be encoded in Python as a class we call `Discrete`. The abstraction of moving in several discrete actions is so useful that we’ll generalize it to `n` directions, instead of just four. In case you’re worried, this is not premature - we’ll actually need a general `Discrete` class in a moment.

```
import random


class Discrete:
```

```
    def __init__(self, num_actions: int):
        """ Discrete action space for num_actions.
        Discrete(4) can be used as encoding moving in
        one of the cardinal directions.
        """
        self.n = num_actions

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


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

❶ A discrete action can be uniformly sampled between `0` and `n-1`. For instance, a `Discrete(4)` sample will give you `0`, `1`, `2`, or `3`.

❷

Sampling from a `Discrete(4)` like in this example will randomly return `0`, `1`, `2`, or `3`. How we interpret these numbers is up to us, so let's say we go for "down", "left", "right", and "up" in that order.

Now that we know how to encode moving around the maze, let's code the maze itself, including the `goal` cell and the position of the `seeker` player that tries to find the goal. To this end we're going to implement a Python class called `Environment`. It's called that, because the maze is the environment in which the player "lives". To make matters easy, we'll always put the `seeker` at `(0, 0)` and the `goal` at `(4, 4)`. To make the `seeker` move and find its goal, we initialize the `Environment` with an `action_space` of `Discrete(4)`.

There is one last bit of information we need to set up for our maze environment, namely an encoding of the `seeker` position. The reason for that is that we're going to implement an algorithm later that keeps track of which actions led to good results for which seeker positions. By encoding the seeker position as a `Discrete(5*5)`, it becomes a single number that's much easier to work with. In RL lingo it is common to call the information of the game that is

accessible to the player an *observation*. So, in analogy to the actions we can carry out for our `seeker`, we can also define an `observation_space` for it. Here's the implementation of what we've just discussed:

```
import os


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

        self.action_space = Discrete(4)  ❷
        self.observation_space = Discrete(5*5)  ❸
```

❶ The `seeker` gets initialized in the top left, the `goal` in the bottom right of the maze.
❷ Our `seeker` can move down, left, up and right.
❸ And it can be in a total of `25` states, one for each position on the grid.

Note that we defined an `info` variable as well, which can be used to print information about the current state of the maze, for instance for debugging purposes. To play an actual game of find-the-goal from the perspective of the seeker, we have to define a few helper methods. Clearly, the game should be considered "done" when the seeker finds the goal. Also, we should reward the seeker for finding the goal. And when the game is over, we should be able to reset it to its initial state, to play again. To round things off, we also define a `get_observation` method that returns the encoded `seeker` position. Continuing our implementation of the `Environment` class, this translates into the following four methods.

```
    def reset(self):  ❶
        """Reset seeker position and return
observations."""
```

```
        self.seeker = (0, 0)

        return self.get_observation()

    def get_observation(self):
        """Encode the seeker position as integer"""
        return 5 * self.seeker[0] + self.seeker[1]  ❷

    def get_reward(self):
        """Reward finding the goal"""
        return 1 if self.seeker == self.goal else 0  ❸

    def is_done(self):
        """We're done if we found the goal"""
        return self.seeker == self.goal  ❹
```

❶ To play a new game, we'll have to `reset` the grid to its original state.

❷ Converting the seeker tuple to a value from the environment's `observation_space`.

❸ The seeker is only rewarded when reaching the goal.

❹ If the seeker is at the goal, the game is over.

The last essential method to implement is the `step` method. Imagine you're playing our maze game and decide to go right as your next move. The `step` method will take this action (namely `3`, the encoding of "right") and apply it to the internal state of the game. To reflect what changed, the `step` method will then return the seeker's observations, its reward, whether the game is over, and the `info` value of the game. Here's how the `step` method works:

```
    def step(self, action):
        """Take a step in a direction and return all
available information."""
        if action == 0:  # move down
            self.seeker = (min(self.seeker[0] + 1, 4),
self.seeker[1])
        elif action == 1:  # move left
            self.seeker = (self.seeker[0],
max(self.seeker[1] - 1, 0))
```

```
        elif action == 2:  # move up
            self.seeker = (max(self.seeker[0] - 1, 0),
self.seeker[1])
        elif action == 3:  # move right
            self.seeker = (self.seeker[0],
min(self.seeker[1] + 1, 4))
        else:
            raise ValueError("Invalid action")

        obs = self.get_observation()
        rew = self.get_reward()
        done = self.is_done()
        return obs, rew, done, self.info  ❶
```

❶ After taking a step in the specified direction, we return observation, reward, whether we're done, and any additional information we might find useful.

I said the `step` method was the last essential method, but we actually want to define one more helper method that's extremely helpful to visualize the game and help us understand it. This `render` method will print the current state of the game to the command line.

```
    def render(self, *args, **kwargs):
        """Render the environment, e.g. by printing its
representation."""
        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
```

❶ First we clear the screen.

❷ Then we draw the grid and mark the goal as `G` and the seeker as `S` on it.

❸ The grid then gets rendered by printing it to your screen.

Great, now we have completed the implementation of our `Environment` class that's defining our 2D-maze game. We can `step` through this game, know when it's `done` and `reset` it again. The player of the game, the `seeker`, can also observe its environment and gets rewarded for finding the goal.

Let's use this implementation to play a game of find-the-goal for a seeker that simply takes random actions. This can be done by creating a new `Environment`, sampling and applying actions to it, and rendering the environment until the game is over:

```
import time

environment = Environment()

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

❶ We can test our environment by applying sampled actions until we're done.

❷ To visualize the environment we render it after waiting for a tenth of a second (otherwise the code runs too fast to follow).

If you run this on your computer, eventually you'll see that the game is over and the seeker has found the goal. It might take a while if you're unlucky.

In case you're objecting that this is an extremely simple problem, and to solve it all you have to do is take at total of 8 steps, namely going right and down four times each in arbitrary order, I'm not arguing with you. The point is that we want to tackle this problem using machine learning, so that we can take on much harder problems later. Specifically, we want to implement an algorithm that figures out on its own how to play the game, merely by playing the

game repeatedly: observing what's happening, deciding what to do next, and getting rewarded for your actions.

If you want to, now is a good time to make the game more complex yourself. As long as you do not change the interface we defined for the `Environment` class, you could modify this game in many ways. Here are a few suggestions:

- Make it a 10x10 grid or randomize the initial position of the seeker.
- Make the outer walls of the grid dangerous. Whenever you try to touch them, you'll incur a reward of -100, i.e a steep penalty.
- Introduce obstacles in the grid that the seeker cannot pass through.

If you're feeling really adventurous, you could also randomize the goal position. This requires extra care, as currently the seeker has no information about the goal position in terms of the `get_observation` method. Maybe come back to tackling this last exercise after you've finished reading this chapter.

Building a Simulation

With the `Environment` class implemented, what does it take to tackle the problem of "teaching" the seeker to play the game well? How can it find the goal consistently in the minimum number of 8 steps necessary? We've equipped the maze environment with reward information, so that the seeker can use this signal to learn to play the game. In reinforcement learning, you play games repeatedly and learn from the experience you made in the process. The player of the game is often referred to as *agent* that takes *actions* in the environment, observes its *state* and receives a *reward*.[2] The better an agent learns, the better it becomes at interpreting the current

game state (observations) and finding actions that lead to more rewarding outcomes.

Regardless of the RL algorithm you want to use (in case you know any), you need to have a way of simulating the game repeatedly, to collect experience data. For this reason we're going to implement a simple `Simulation` class in just a bit.

The other useful abstraction we need to proceed is that of a `Policy`, a way of specifying actions. Right now the only thing we can do to play the game is sampling random actions for our seeker. What a `Policy` allows us to do is to get better actions for the current state of the game. In fact, we define a `Policy` to be a class with a `get_action` method that takes a game state and returns an action.

Remember that in our game the seeker has a total of `25` possible states on the grid, and can carry out `4` actions. A simple idea would be to look at pairs of states and actions and assign a high value to a pair if carrying out this action in this state will lead to a high reward, and a low value otherwise. For instance, from your intuition of the game it should be clear that going down or right is always a good idea, whereas going left or up is not. Then, create a `25x4` lookup table of all possible state-action pairs and store it in our `Policy`. Then we could simply ask our policy to return the highest value of any action given a state. Of course, implementing an algorithm that finds good values for these state-action pairs is the challenging part. Let's implement this idea of a `Policy` in first and worry about a suitable algorithm later.

```
import numpy as np

class Policy:

    def __init__(self, env):
        """A Policy suggests actions based on the current
state.
        We do this by tracking the value of each state-
action pair.
```

```
        """
        self.state_action_table = [
            [0 for _ in range(env.action_space.n)]
            for _ in range(env.observation_space.n)   ❶
        ]
        self.action_space = env.action_space

    def get_action(self, state, explore=True, epsilon=0.1):
❷
        """Explore randomly or exploit the best value
currently available."""
        if explore and random.uniform(0, 1) < epsilon:   ❸
            return self.action_space.sample()
        return np.argmax(self.state_action_table[state])
❹
```

❶ We define a nested list of values for each state-action pair, initialized to zero.

❷ On demand, we can `explore` random actions so that we don't get stuck in suboptimal behavior.

❸ Sometimes we might want to randomly explore actions in the game, which is why we introduce an `explore` parameter to the `get_action` method. By default, this happens 10% of the time.

❹ We return the action with the highest value in the lookup table, given the current state.

I've snuck in a little implementation detail into the `Policy` definition that might be a bit confusing. The `get_action` method has an `explore` parameter. The reason for this is that if you learn an extremely poor policy, e.g. one that always wants you to move left, you have no chance of ever finding better solutions. In other words, sometimes you need to explore new ways, and not "exploit" your current understanding of the game. As indicated before, we haven't discussed how to learn to improve the values in the `state_action_table` of our policy. For now, just keep in mind that the policy gives us the actions we want to follow when simulating the maze game.

Moving on to the `Simulation` class we spoke about earlier, a simulation should take an `Environment` and compute actions of a given `Policy` until the goal is reached and the game ends. The data we observe when "rolling out" a full game like this is what we call the *experience* we gained. Accordingly, our `Simulation` class has a `rollout` method that computes `experiences` for a full game and returns them. Here's what the implementation of the `Simulation` class looks like:

```
class Simulation(object):
    def __init__(self, env):
        """Simulates rollouts of an environment, given a
policy to follow."""
        self.env = env

    def rollout(self, policy, render=False, explore=True,
epsilon=0.1):  ❶
        """Returns experiences for a policy rollout."""
        experiences = []
        state = self.env.reset()  ❷
        done = False
        while not done:
            action = policy.get_action(state, explore,
epsilon)  ❸
            next_state, reward, done, info =
self.env.step(action)  ❹
            experiences.append([state, action, reward,
next_state])  ❺
            state = next_state
            if render:  ❻
                time.sleep(0.05)
                self.env.render()

        return experiences
```

❶ We compute a game "roll-out" by following the actions of a `policy`, and we can optionally render the simulation.

❷ To be sure, we reset the environment before each rollout.

❸ The passed in `policy` drives the actions we take. The `explore` and `epsilon` parameters are passed through.

❹ We step through the environment by applying the policy's `action`.
❺ We define an experience as a `(state, action, reward, next_state)` quadruple.
❻ Optionally render the environment at each step.

Note that each entry of the `experiences` we collect in a `rollout` consists of four values: the current state, the action taken, the reward received, and the next state. The algorithm we're going to implement in a moment will use these experiences to learn from them. Other algorithms might use other experience values, but those are the ones we need to proceed.

Now we have a policy that hasn't learned anything just yet, but we can already test its interface to see if it works. Let's try it out by initializing a `Simulation` object, calling its `rollout` method on a not-so-smart `Policy`, and then printing the `state_action_table` of it:

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

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

❶ We roll-out one full game with an "untrained" policy that we render.
❷ The state-action values are currently all zero.

If you feel like we haven't made much progress since the last section, I can promise you that things will come together in the next one. The prep work of setting up a `Simulation` and a `Policy` were necessary to frame the problem correctly. Now the only thing that's left is to devise a smart way to update the internal state of the

`Policy` based on the experiences we've collected, so that it actually learns to play the maze game.

Training a Reinforcement Learning Model

Imagine we have a set of experiences that we've collected from a couple of games. What would be a smart way to update the values in the `state_action_table` of our `Policy`? Here's one idea. Let's say you're sitting at position `(3,5)`, and you've decided to go right, which puts you at `(4,5)`, just one step away from the goal. Clearly you could then just go right and collect a reward of `1` in this scenario. That must mean the current state you're in combined with an action of going "right" should have a high value. In other words, the value of this particular state-action pair should be high. In contrast, moving left in the same situation does not lead to anything, and the corresponding state-action pair should have a low value.

More generally, let's say you were in a given `state`, you've decided to take an `action`, leading to a `reward`, and you're then in `next_state`. Remember that this is how we defined an experience. With our `policy.state_action_table` we can peek a little ahead and see if we can expect to gain anything from actions taken from `next_state`. That is, we can compute

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

How should we compare the knowledge of this value to the current state-action value, which is `value = policy.state_action_table[state][action]`? There are many ways to go about this, but we clearly can't completely discard the current `value` and put too much trust in `next_max`. After all, this is just a single piece of experience we're using here. So as a first approximation, why don't we simply compute a weighted sum of the old and the expected value and go with `new_value = 0.9 *`

`value + 0.1 * next_max`? Here, the values `0.9` and `0.1` have been chosen somewhat arbitrarily, the only important piece is that the first value is high enough to reflect our preference to keep the old value, and that both weights sum to `1`. That formula is a good starting point, but the problem is that we're not at all factoring in the crucial information that we're getting from the `reward`. In fact, we should put more trust in the current `reward` value than in the projected `next_max` value, so it's a good idea to discount the latter a little, let's say by 10%. Updating the state-action value would then look like this:

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

Depending on your level of experience with this kind of reasoning, the last few paragraphs might be a lot to digest. The good thing is that, if you've understood the explanations up to this point, the remainder of this chapter will likely come easy to you. Mathematically, this was the last (and only) hard part of this example. If you've worked with RL before, you will have noticed by now that this is an implementation of the so-called Q-Learning algorithm. It's called that, because the state-action table can be described as a function `Q(state, action)` that returns values for these pairs.

We're almost there, so let's formalize this procedure by implementing an `update_policy` function for a policy and collected experiences:

```
def update_policy(policy, experiences, weight=0.1,
discount_factor=0.9):
    """Updates a given policy with a list of (state,
action, reward, state)
    experiences."""
    for state, action, reward, next_state in experiences:
❶
        next_max =
np.max(policy.state_action_table[next_state])  ❷
        value = policy.state_action_table[state][action]
```

```
❸
        new_value = (1 - weight) * value + weight * \
                    (reward + discount_factor * next_max)
❹
        policy.state_action_table[state][action] =
new_value  ❺
```

❶ We loop through all experiences in order.
❷ Then we choose the maximum value among all possible actions in the next state.
❸ We then extract the current state-action value.
❹ The new value is the weighted sum of the old value and the expected value, which is the sum of the current reward and the discounted `next_max`.
❺ After updating, we set the new `state_action_table` value.

Having this function in place now makes it really simple to train a policy to make better decisions. We can use the following procedure:

- Initialize a policy and a simulation.
- Run the simulation many times, let's say for a total of `10000` runs.
- For each game, first collect the experiences by running a `rollout`.
- Then update the policy by calling `update_policy` on the collected experiences.

That's it! The following `train_policy` function implements the above procedure straight up.

```
def train_policy(env, num_episodes=10000, weight=0.1,
discount_factor=0.9):
    """Training a policy by updating it with rollout
experiences."""
    policy = Policy(env)
    sim = Simulation(env)
    for _ in range(num_episodes):
```

```
        experiences = sim.rollout(policy)  ❶
        update_policy(policy, experiences, weight,
discount_factor)  ❷

    return policy


trained_policy = train_policy(environment)  ❸
```

❶ Collect experiences for each game.
❷ Update our policy with those experiences.
❸ Finally, train and return a policy for our `enviroment` from before.

Note that the high-brow way of speaking of a full play-through of the maze game is an *episode* in the RL literature. That's why we call the argument `num_episodes` in the `train_policy` function, rather than `num_games`.

Q-LEARNING

The Q-Learning algorithm we just implemented is often the first algorithm taught in RL classes, mostly because it is relatively easy to reason about. You collect and tabulate experience data that shows you how well state-action pairs work, and update the table according to the Q-learning update rule.

For RL problems that either have a huge number of states or actions, the Q-table can become excessively large. The algorithm then becomes inefficient, because it would take too much time to collect enough experience data for all (relevant) state-action pairs.

One way to address this issue is to use a neural network to approximate the Q-table. By this we mean that you can employ a deep neural network to learn a function that maps states to actions. This approach is called Deep Q-Learning and the networks used for learning are called Deep Q-Networks (DQN). From Chapter 4 on out we will exclusively use deep learning to tackle RL problems in this book.

Now that we have a trained policy, let's see how well it performs. We've run random policies twice before in this chapter, just to get an idea of how well they work for the maze problem. But let's now properly evaluate our trained policy on several games and see how it does on average. Specifically, we'll run our simulation for a couple of episodes and count how many steps it took per episode to reach the goal. So, let's implement an `evaluate_policy` function that does precisely that:

```
def evaluate_policy(env, policy, num_episodes=10):
    """Evaluate a trained policy through rollouts."""
    simulation = Simulation(env)
    steps = 0
```

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

    print(f"{steps / num_episodes} steps on average "
          f"for a total of {num_episodes} episodes.")

    return steps / num_episodes


evaluate_policy(environment, trained_policy)
```

❶ This time we set `explore` to `False` to fully exploit the learnings of the trained policy.

❷ The length of the `experiences` is the number of steps we took to finish the game.

Apart from seeing the trained policy crush the maze problem ten times in a row, as we hoped it would, you should also see the following prompt:

```
8.0 steps on average for a total of 10 episodes.
```

In other words, the trained policy is able to find optimal solutions for the maze game. That means you've successfully implemented your first RL algorithm from scratch!

With the understanding you've built up by now, do you think placing the `seeker` into randomized starting positions and then running this evaluation function would still work? Why don't you go ahead and make the changes necessary for that?

Another interesting question to ask yourself is what assumptions went into the algorithm we used. For instance, it's clearly a prerequisite for the algorithm that all state-action pairs can be tabulated. Do you think this would still work well if we had millions of states and thousands of actions?

Building a Distributed Ray App

You hopefully enjoyed the example so far, but might be wondering how what we've done until now relates to Ray (which is a great question). As you'll see shortly, all we need to make the above RL experiment a distributed Ray app is writing three short code snippets. This is what we're going to do:

- We make the `Simulation` a Ray actor in just a few lines of code.
- Then we define a parallel version of `train_policy` that's structurally similar to its original. For simplicity, we will only parallelize the rollouts, not the policy updates.
- Finally, we simply train and evaluate the policy as before, but using `train_policy_parallel`.

Let's tackle the first step of this plan by implementing a Ray actor called `SimulationActor`:

```
import ray

ray.init()

@ray.remote
class SimulationActor(Simulation):  ❶
    """Ray actor for a Simulation."""
    def __init__(self):
        env = Environment()
        super().__init__(env)
```

❶ This Ray actor wraps our `Simulation` class in a straightforward way.

With the foundations on Ray Core you've developed in chapter Chapter 2 you should have no problems reading this code. It might take some getting used to writing it yourself, but conceptually you should be on top of this example.

Moving on, let's define a `train_policy_parallel` function that distributes this RL workload on your local Ray cluster. To do so, we create a `policy` on the driver and a total of four `SimulationActor` instances that we can use for distributed rollouts. We then put the `policy` into the object store with `ray.put` and pass it to the remote `rollout` calls as an argument to collect experiences for a given number of training episodes. We then use `ray.wait` to get the finished rollouts (and account for the fact that some rollouts might finish earlier than others) and update our policy with the collected experiences. Finally, we return the trained policy.

```
def train_policy_parallel(env, num_episodes=1000,
num_simulations=4):
    """Parallel policy training function."""
    policy = Policy(env)  ❶
    simulations = [SimulationActor.remote() for _ in
range(num_simulations)]  ❷

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

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

    return policy
```

❶ We initialize a policy for the given environment.
❷ Instead of one simulation, we create four simulation actors.
❸ We then put the policy into the object store.
❹ For each of the `1000` episodes, we collect experience data in parallel using our simulation actors.
❺ Finished rollouts can be retrieved from the object store and used to update the policy.

This allows us to take the last step and run the training procedure in parallel and then evaluate the resulting as before.

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

The result of those two lines is the same as before, when we ran the serial version of the RL training for the maze. I hope you appreciate how `train_policy_parallel` has the exact same high-level structure as `train_policy`. It's a good exercise to compare the two line-by-line.

Essentially, all it took to parallelize the training process was to use the `ray.remote` decorator on a class in a suitable way and then use the right `remote` calls. Of course, you need some experience to get this right. But notice how little time we spent on thinking about distributed computing, and how much time we could spend on the actual application code. We didn't need to adopt an entirely new programming paradigm and could simply approach the problem in the most natural way. Ultimately, that's what you want — and Ray is great at giving you this kind of flexibility.

To wrap things up, let's have a quick look at the execution graph of the Ray application that we've just built. Figure Figure 3-1 summarizes this task graph in a compact way:

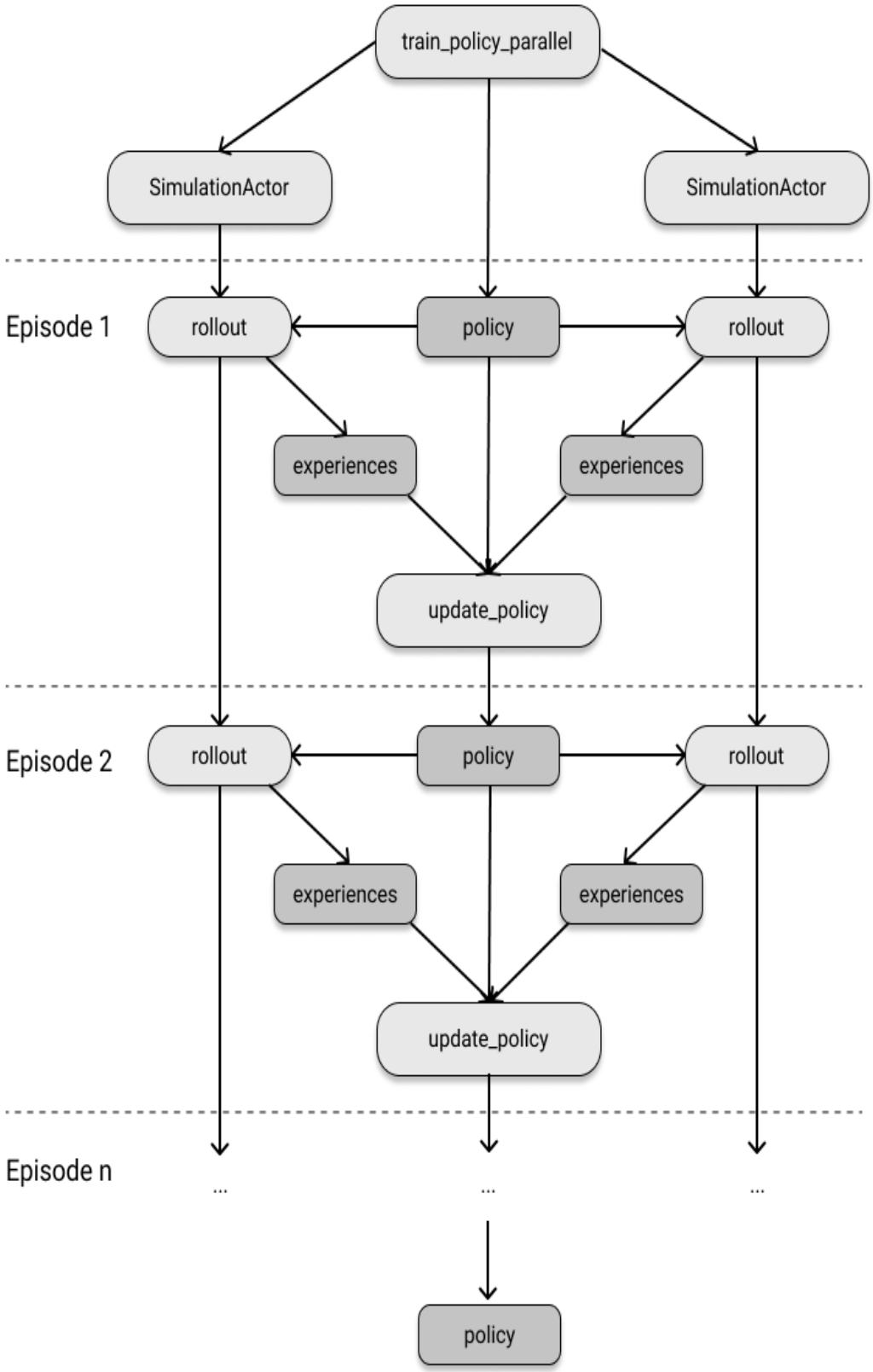

train_policy_parallel
SimulationActor
SimulationActor
Episode 1
rollout
policy
rollout
experiences
experiences
update_policy
Episode 2
rollout
policy
rollout
experiences
experiences
update_policy
Episode n
...
...
...
policy

Figure 3-1. Parallel training of a reinforcement learning policy with Ray.

An interesting side note about the running example of this chapter is that it's an implementation of the pseudo-code example used to illustrate the flexibility of Ray in the initial paper by its creators. That paper has a figure similar to Figure 3-1 and is worth reading for context.

Recapping RL Terminology

Before we wrap up this chapter, let's discuss the concepts we've encountered in the maze example in a broader context. Doing so will prepare you for more complex RL settings in the next chapter and show you where we simplified things a little for the running example of this chapter. If you know RL well enough, you can skip this section.

Every RL problem starts with the formulation of an *environment*, which describes the dynamics of the "game" you want to play. The environment hosts a player or *agent* which interacts with its environment through a simple interface. The agent can request information from the environment, namely its current *state* within the environment, the *reward* it has received in this state, and whether the game is *done* or not. In observing states and rewards, the agent can learn to make decisions based on the information it receives. Specifically, the agent will emit an *action* that can be executed by the environment by taking the next `step`.

The mechanism used by an agent to produce actions for a given state is called a *policy*, and we will sometimes say that the agent follows a given policy. Given a policy, we can simulate or *roll-out* a few steps or an entire game using said policy. During a roll-out we can collect *experiences*, which we collect information about the current state and reward, the next action and the resulting state. An entire sequence of steps from start to finish is referred to as an

episode, and the environment can be `reset` to its initial state to start a new episode.

The policy we used in this chapter was based on the simple idea of tabulating *state-action values* (also called *Q-values*), and the algorithm used to update the policy from the experiences collected during roll-outs is called *Q-learning*. More generally, you can consider the state-action table we implemented as the *model* used by the policy. In the next chapter you will see examples of more complex models, such as a neural network to learn state-action values. The policy can decide to *exploit* what it has learnt about the environment by choosing the best available value of its model, or *explore* the environment by choosing a random action.

Many of the basic concepts introduced here hold for any RL problem, but we've made a few simplifying assumptions. For instance, there could be *multiple agents* acting in the environment (imagine having multiple seekers competing for reaching the goal first), and we'll look into so-called multi-agent environments and multi-agent RL and in the next chapter. Also, we assumed that the *action space* of an agent was *discrete*, meaning that the agent could only take a fixed set of actions. You can, of course, also have *continuous* action spaces, and the cartpole example from Chapter 1 is one example of this. Especially when you have multiple agents, action spaces can be more complicated, and you might need tuples of actions or even nest them accordingly. The *observation space* we've considered for the maze game was also quite simple, and was modeled as a discrete set of states. You can easily imagine that complex agents like robots interacting with their environments might work with image or video data as observations, which would require a more complex observation space, too.

Another crucial assumption we made is that the environment is *deterministic*, meaning that when our agent chose to take an action, the resulting state would always reflect that choice. In general environments this is not the case, and there can be elements of

randomness at play in the environment. For instance, we could have implemented a coin flip in the maze game and whenever tails came up, the agent would get pushed in a random direction. In that scenario, we couldn't have planned ahead like we did in this chapter, as actions would not deterministically lead to the same next state every time. To reflect this probabilistic behavior, in general we have to account for *state transition probabilities* in our RL experiments.

And the last simplifying assumption I'd like to talk about here is that we've been treating the environment and its dynamics as a game that can be perfectly simulated. But the fact is that there are physical systems that can't be faithfully simulated. In that case you might still interact with this physical environment through an interface like the one we defined in our `Environment` class, but there would be some communication overhead involved. In practice, I find that *reasoning* about RL problems as if they were games takes very little away from the experience.

Summary

To recap, we've implemented a simple maze problem in plain Python and then solved the task of finding the goal in that maze using a straightforward reinforcement learning algorithm. We then took this solution and ported it to a distributed Ray application in roughly 25 lines of code. We did so without having to plan upfront how to work with Ray — we simply used the Ray API to parallelize our Python code. This example shows how Ray gets out of your way and lets you focus on your application code. It also demonstrates how custom workloads that use advanced techniques like RL can be efficiently implemented and distributed with Ray.

In the next chapter, you'll build on what you've learned here and see how easy it is to solve our maze problem directly with the higher-level Ray RLlib library.

1 We don’t have gardening robots like the one I’ve sketched yet. And we don’t know which AI paradigm will get us there. For the experts among you, I don’t claim that RL is the answer. RL is just a paradigm that naturally fits into this specific discussion of AI goals.

2 As we’ll see in chapter Chapter 4, you can run RL on multi-player games, too. Making the maze environment a so-called multi-agent environment, in which multiple seekers compete for the goal, is an interesting exercise.

Chapter 4. Reinforcement Learning with Ray RLlib

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

In the last chapter you've built a Reinforcement Learning (RL) environment, a simulation to play out some games, an RL algorithm, and the code to parallelize the training of the algorithm - all completely from scratch. It's good to know how to do all that, but in practice the only thing you really want to do when training RL algorithms is the first part, namely specifying your custom environment, the "game" you want to play.[1] Most of your efforts will then go into selecting the right algorithm, setting it up, finding the best parameters for the problem, and generally focusing on training a well-performing policy.

Ray RLlib is an industry-grade library for building RL algorithms at scale. You've seen a first example of RLlib in Chapter 1 already, but in this chapter we'll go into much more depth. The great thing about RLlib is that it's a mature library for developers that comes with good abstractions to work with. As you will see, many of these abstractions you already know from the last chapter.

We start out this chapter by first giving you an overview of RLlib's capabilities. Then we quickly revisit the maze game from Chapter 3 and show you how to tackle it both with the RLlib command line interface (CLI) and the RLlib Python API in a few lines of code. You'll

see how easy RLlib is to get started with before learning about its key concepts, such as RLlib environments and algorithms.

We'll also take a closer look at some advanced RL topics that are extremely useful in practice, but are not often properly supported in other RL libraries. For instance, you will learn how to create a curriculum for your RL agents so that they can learn simple scenarios first, before moving on to more complex ones. You will also see how RLlib deals with having multiple agents in a single environment, and how to leverage experience data that you've collected outside your current application to improve your agent's performance.

An Overview of RLlib

Before we dive into any examples, let's quickly discuss what RLlib is and what it can do. As part of the Ray ecosystem, RLlib inherits all the performance and scalability benefits of Ray. In particular, RLlib is distributed by default, so you can scale out your RL training to as many nodes as you want.

Another benefit of being built on top of Ray is that RLlib integrates tightly with other Ray libraries. For instance, the hyperparameters of any RLlib algorithm can be tuned with Ray Tune, as we will see in Chapter 5. You can also seamlessly deploy your RLlib models with Ray Serve.[2]

What's extremely useful is that RLlib works with both of the predominant deep learning frameworks at the time of this writing, namely PyTorch and TensorFlow. You can use either one of them as your backend and can easily switch between them, often by just changing one line of code. That's a huge benefit, as companies are often locked into their underlying deep learning framework and can't afford to switch to another system and rewrite their code.

RLlib also has a track record of solving real-world problems and is a mature library used by many companies to bring their RL workloads to production. The RLlib API appeals to many engineers, as it offers the right level of abstraction for many applications, while still being flexible enough to be extended.

Apart from these more general benefits, RLlib has a lot of RL specific features that we will cover in this chapter. In fact, RLlib is so feature rich that it would deserve a book on its own, which means we can only touch on some aspects of it here.

For instance, RLlib has a rich library of advanced RL algorithms to choose from. In this chapter we will only focus on a few select ones, but you can track the growing list of options on the RLlib algorithms page. RLlib also has many options for specifying RL environments and is very flexible in handling them during training, see for an overview of RLlib environments.

Getting Started with RLlib

To use RLlib, make sure you have installed it on your computer:

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

As with every chapter in this book, if you don't feel like following along by typing the code yourself, you can check out the accompanying notebook for this chapter.

Every RL problem starts with having an interesting environment to investigate. In Chapter 1 we already looked at the classical cartpole balancing problem. Recall that we didn't implement this cartpole environment, it came out of the box with RLlib.

In contrast, in Chapter 3 we implemented a simple maze game on our own. The problem with this implementation is that we can't directly use it with RLlib, or any other RL library for that matter. The reason is that in RL you have ubiquitous standards, and our

environments need to implement certain interfaces. The best known and most widely used library for RL environments is `gym`, an open-source Python project from OpenAI.

Let's have a look at what `gym` is and how to convert our maze `Environment` from the last chapter to a `gym` environment compatible with RLlib.

Building a Gym Environment

If you look at the well-documented and easy to read `gym.Env` environment interface on GitHub, you'll notice that an implementation of this interface has two mandatory class variables and three methods that subclasses need to implement. You don't have to check the source code, but I do encourage you to have a look. You might just be surprised by how much you already know about these environments.

In short, the interface of a gym environment looks like the following pseudocode:

```
import gym


class Env:

    action_space: gym.spaces.Space
    observation_space: gym.spaces.Space  ❶

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

    def reset(self):  ❸
        ...

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

❶ The `gym.Env` interface has an action and an observation space.

❷ The `Env` can run a `step` and returns a tuple of observations, reward, done condition, and further info.
❸ An `Env` can `reset` itself, which will return the initial observations of a new episode
❹ We can `render` an `Env` for different purposes, like for human display or as string representation.

If you've read Chapter 3 carefully, you'll notice that this is very similar to the interface of the maze `Environment` we built there. In fact, `gym` has a so-called `Discrete` space implemented in `gym.spaces`, which means that we can make our maze `Environment` a `gym.Env` as follows. We assume that you store this code in a file called `maze_gym_env.py` and that the code for the `Environment` from Chapter 3 is either located at the top of that file (or is imported there).

```
# maze_gym_env.py  | Original definition of `Environment`
goes at the top.

import gym
from gym.spaces import Discrete  ❶


class GymEnvironment(Environment, gym.Env):  ❷
    def __init__(self, *args, **kwargs):
        """Make our original `Environment` a gym `Env`."""
        super().__init__(*args, **kwargs)


gym_env = GymEnvironment()
```

❶ We replace our own `Discrete` implementation with that of `gym`.
❷ We then simply make our `GymEnvironment` implement a `gym.Env`. The interface is essentially the same as before.

Of course, we could have made our original `Environment` implement `gym.Env` by simply inheriting from it in the first place. But

the point is that the `gym.Env` interface comes up so naturally in the context of RL that it is a good exercise to implement it without having to resort to external libraries.[3]

The `gym.Env` interface also comes with helpful utility functionality and many interesting example implementations. For instance, the `CartPole-v1` environment we used in Chapter 1 is an example from `gym`,[4] and there are many other environments available to test your RL algorithms.

Running the RLlib CLI

Now that we have our `GymEnvironment` implemented as a `gym.Env`, here's how you can use it with RLlib. You've seen the RLlib CLI in action in Chapter 1 before, but this time the situation is a bit different. In the first chapter we simply ran a tuned example using the `rllib example` command.

This time around we want to bring our own `gym` environment class, namely the class `GymEnvironment` that we defined in `maze_gym_env.py`. To specify this class in Ray RLlib, you use the full qualifying name of the class from where you're referencing it, so in our case that's `maze_gym_env.GymEnvironment`. If you had a more complicated Python project and your environment is stored in another module, you'd simply add the module name accordingly.

The following Python file specifies the minimal configuration needed to train an RLlib algorithm on the `GymEnvironment` class. To align as closely as possible with our experiment from Chapter 3, in which we used Q-learning, we use a `DQNConfig` to define a `DQN` algorithm and store it in a file called `maze.py`. This already gives you a quick preview of RLlib's Python API, which we'll cover in the next section.

```
# maze.py

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

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

To run this with RLlib, we're using the `rllib train` command. We do this by specifying the `file` we want to run, namely `maze.py`. To make sure we can control the time of training, we tell our algorithm to `stop` after running for a total of `10000` time steps (`timesteps_total`):

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

This single line of code basically takes care of everything we did in Chapter 3, but better:

- It runs a more sophisticated version of Q-Learning for us[5] (`DQN`)
- It takes care of scaling out to multiple workers under the hood (in this case two)
- It even creates checkpoints of the algorithm automatically for us.

From the output of that training script you should see that Ray will write training results to a directory located at `~/ray_results/maze_env`. And if the training run finishes successfully,[6] you'll get a checkpoint and a copiable `rllib evaluate` command in the output, just as in the example from Chapter 1. Using this reported `<checkpoint>`, you can now evaluate the trained policy on our custom environment by running the following command:

```
rllib evaluate ~/ray_results/maze_env/<checkpoint>\
  --algo DQN\
  --env maze_gym_env.Environment\
  --steps 100
```

The algorithm used in `--algo` and the environment specified with `--env` have to match the ones used in the training run, and we evaluate the trained algorithm for a total of 100 `steps`. This should lead to output of the following form:

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

It should not come as a big surprise that the `DQN` algorithm from RLlib gets the maximum reward of `1` for the simple maze environment we tasked it with every single time.

Before moving on to the Python API, we should mention that the RLlib CLI uses Ray Tune under the hood, for instance for creating the checkpoints of your algorithms. You will learn more about this integration in Chapter 5.

Using the RLlib Python API

In the end, the RLlib CLI is merely a wrapper around its underlying Python library. As you will likely spend most of your time coding your reinforcement learning experiments in Python, we'll focus the rest of this chapter on aspects of this API.

To run RL workloads with RLlib from Python, your main entrypoint is the `Algorithm` class. To define an algorithm, you always start with a corresponding `AlgorithmConfig` class. For instance, in the previous section we used a `DQNConfig` as a starting point, and the `rllib train` command took care of instantiating the `DQN` algorithm for us. All other RLlib algorithms follow the same pattern.

Training RLlib algorithms

Every RLlib `Algorithm` comes with reasonable default parameters, meaning that you *can* initialize them without having to tweak any configuration parameters for these algorithms.[7]

That said, it's worth noting that RLlib algorithms are highly configurable, as you will see in the following example. We start by creating a `DQNConfig` object. Then we specify its `environment` and set the number of rollout workers to two by using the `rollouts` method. What that means is that the `DQN` algorithm will spawn two Ray actors, each using a CPU by default, to run the algorithm in parallel. Also, for later evaluation purposes, we set `create_env_on_local_worker` to `True`:

```
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 First, we set the `environment` to our custom `GymEnvironment` class. We also configure the number of rollout workers and ensure that an environment instance is created on the local worker.

❷ We use the `DQNConfig` from RLlib to `build` a `DQN` algorithm for training. This time we use two rollout workers.

❸ We can then simply call the `train` method to train the algorithm for ten iterations.

❹

With the `pretty_print` utility we can generate human-readable output of the training results.

Note that the number `10` training iterations has no special meaning, but it should be enough for the algorithm to learn to solve the maze problem adequately. The example just goes to show you that you have full control over the training process.

From printing the `config` dictionary, you can verify that the `num_rollout_workers` parameter is set to two.[8] The `result` contains detailed information about the state of the `DQN` algorithm and the training results, which is too verbose to show here. The part that's most relevant for us right now is information about the reward of the algorithm, which hopefully indicates that the algorithm learned to solve the maze problem. You should see output of the following form (we're only showing the most relevant information for clarity):

```
...
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
...
```

In particular, this output shows that the minimum reward attained on average per episode is `1.0`, which in turn means that the agent always reached the goal and collected the maximum reward (`1.0`).

Saving, loading and evaluating RLlib models

Reaching the goal for this simple example isn't too hard, but let's see if evaluating the trained algorithm confirms that the agent can also do so in an optimal way, namely by only taking the minimum number of eight steps to reach the goal.

To do so, we utilize another mechanism that you've already seen from the RLlib CLI, namely *checkpointing*. Creating algorithm checkpoints is very useful to ensure you can recover your work in case of a crash, or simply to track training progress persistently. You can simply create a checkpoint of an RLlib algorithm at any point in the training process by calling `algo.save()`. Once you have a checkpoint, you can easily restore your `Algorithm` with it. And evaluating a model is as simple as calling `algo.evaluate(checkpoint)` with the checkpoint you created. Here's how that looks like if you put it all together:

```
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 You can `save` algorithms to create checkpoints.
❷ RLlib algorithms can be evaluated at any point in time by calling `evaluate`.
❸ We can `stop` an `algo` free all claimed resources.
❹ And you can also restore any `Algorithm` from a given checkpoint with `from_checkpoint`.

Looking at the output of this example, we can now confirm that the trained RLlib algorithm did indeed converge to a good solution for the maze problem, as indicated by episodes of length `8` in evaluation:

```
~/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
  ...
```

Computing actions

RLlib algorithms have much more functionality than just the `train`, `evaluate`, `save`, and `from_checkpoint` methods we've seen so far. For example, you can directly compute actions given the current state of an environment. In Chapter 3 we implemented episode rollouts by stepping through an environment and collecting rewards. We can easily do the same with RLlib for our `GymEnvironment` as follows:

```
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 To compute actions for given `observations` use `compute_single_action`.

In case you should need to compute many actions at once, not just a single one, you can use the `compute_actions` method instead, which takes dictionaries of observations as input and produces dictionaries of actions with the same dictionary keys as output.

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

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

❶ For multiple actions, you can use `compute_actions`.

Accessing Policy and Model States

Remember that each reinforcement learning algorithm is based on a *policy* that chooses next actions given the current observations the agent has of the environment. Each policy is in turn based on an underlying *model*.

In the case of vanilla Q-Learning that we discussed in Chapter 3 the model was a simple look-up table of state-action values, also called Q-values. And that policy used this model for predicting next actions in case it decided to *exploit* what the model had learned so far, or to *explore* the environment with random actions otherwise.

When using Deep Q-Learning, the underlying model of the policy is a neural network that, loosely speaking, maps observations to actions. Note that for choosing next actions in an environment, we're ultimately not interested in the concrete values of the approximated Q-values, but rather in the *probabilities* of taking each action. The probability distribution over all possible actions is called an *action distribution*. In the maze example we're using here as a running examples we can move up, right, down or left. So, in our case an action distribution is a vector of fourfour probabilities, one for each action. In the case of Q-Learning, the algorithm will always *greedily* choose the action with the highest probability of this distribution, while other algorithms will sample from it.

To make things concrete, let's have a look at how you access policiesfootenote:[We should note that the `Policy` class as it exists in RLlib today will be replaced in one of the next releases. The new `Policy` class will likely be a drop-in replacement for the most part, will exhibit some minor differences. The idea of the class remains the same, though. a *policy* is a class that encapsulates the logic of

choosing actions given observations, and it gives you access to the underlying models used.] and models in RLlib:

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

model = policy.model
```

Both `policy` and `model` have many useful methods to explore. In this example we use `get_weights` to inspect the parameters of the model underlying the policy (which are called “weights” by standard convention).

To convince you that there is in fact not just one model at play here, but in fact a collection of models,[9] we can access all the workers we used in training - and then ask each worker’s policy for their weights using `foreach_worker`: ch the updated weights after each training step from the local worker to those 4 worker models.

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

In this way, you can access every method available on an `Algorithm` instance on each of your workers. In principle, you can use this to *set* model parameters as well, or otherwise configure your workers. RLlib workers are ultimately Ray actors, so you can alter and manipulate them in almost any way you like.

We haven’t talked about the specific implementation of Deep Q-Learning used in `DQN`, but the `model` used is in fact a bit more complex than what I’ve described so far. Every RLlib `model` obtained from a policy has a `base_model` that has a neat `summary` method.[10] to describe itself:

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

As you can see from the output below, this model takes in our `observations`. The shape of these `observations` is a bit strangely annotated as `[(None, 25)]`, but essentially this just means we have the expected `5*5` maze grid values correctly encoded. The model follows up with two so-called `Dense` layers and predicts a single value at the end.[11]

```
Model: "model"
__________________________________________________________________________
_____________________
Layer (type)                 Output Shape              Param #
Connected to
==========================================================================
=====================
observations (InputLayer)    [(None, 25)]              0
__________________________________________________________________________
_____________________
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]
==========================================================================
=====================
Total params: 72,705
Trainable params: 72,705
Non-trainable params: 0
__________________________________________________________________________
_____________________
```

Note that it's perfectly possible to customize this model for your RLlib experiments. If your environment is quite complex and has a big observation space, for instance, you might need a bigger model to capture that complexity. However, doing so requires in-depth

knowledge of the underlying neural network framework (in this case TensorFlow), which we don't assume you have.[12]

STATE-ACTION VALUES AND STATE-VALUE FUNCTIONS

So far we've been mostly concerned with the concept of state-action values, since this concept takes center stage in the formulation of Q-Learning that we used extensively so far.

The `model` we've just had a look at has a dedicated output, in deep learning terms called a *head*, for predicting Q-values. You can access and summarize this part of the model through `model.q_value_head.summary()`.

In contrast to that, it's also possible to ask of how valuable a particular *state* is, without specifying an action that pairs with it. This leads to the concept of state-value functions, or simply value functions, that are very important in the RL literature. We can't go into more detail in this RLlib introduction, but note that you have access to a *value function head* as well through `model.state_value_head.summary()`.

Next, let's see if we can take some observations from our environment and pass them to the `model` we just extracted from our `policy`. This part is a bit technically involved, because models are a bit more difficult to access directly in RLlib. The reason is that normally you would only interface with a `model` through your `policy`, which takes care of preprocessing the observations, among other things.

Luckily, we can simply access the preprocessor used by the policy, `transform` the observations from our environment, and then pass them to the model:

```
from ray.rllib.models.preprocessors import get_preprocessor


env = GymEnvironment()
obs_space = env.observation_space
preprocessor = get_preprocessor(obs_space)(obs_space)  ❶

observations = env.reset()
transformed =
preprocessor.transform(observations).reshape(1, -1)  ❷

model_output, _ = model({"obs": transformed})  ❸
```

❶ You can use `get_preprocessor` to access the preprocessor used by the policy.

❷ For any `observations` obtained from your `env` you can use `transform` on them to the format expected by the model. Note that we need to reshape the observations, too.

❸ You get the model output by calling the model on a preprocessed observation dictionary.

Having computed our `model_output`, we can now both access the Q-values, and the action distribution of the model for this output like this:

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

action_distribution = policy.dist_class(model_output,
model)  ❷
sample = action_distribution.sample()  ❸
print(sample)
```

❶ The `get_q_value_distributions` method is specific to `DQN` models only.

❷ By accessing `dist_class` we get the policy's action distribution class.

❸ Action distributions can be sampled from.

Configuring RLlib Experiments

Now that you've seen the basic Python training API of RLlib in an example, let's take a step back and discuss in more depth how to configure and run RLlib experiments. By now you know that to define an `Algorithm`, you start with the respective `AlgorithmConfig` and then `build` your algorithm from it. So far we've only used the `rollout` method of an `AlgorithmConfig` to set the number of rollout workers to two, and set our `environment` accordingly.

If you want to alter the behaviour of your RLlib training run, you chain more utility methods onto the `AlgorithmConfig` instance and then call `build` on it at the end. As RLlib algorithms are fairly complex, they come with many configuration options. To make things easier, the common properties of algorithms are naturally grouped into useful categories.[13] Each such category comes with its own respective `AlgorithmConfig` method:

- `training()`: Takes care of all training-related configuration options of your algorithm. The `training` method is the one place that RLlib algorithms differ in their configuration. All other methods below are algorithm-agnostic.
- `environment()`: With this method you configure all aspects of your environment.
- `rollouts()`: This configuration method is used to modify the setup and behaviour of your rollout workers.
- `exploration()`: You can use this method to alter the behaviour of your exploration strategy.
- `resources`: This method is used to configure the compute resources used by your algorithm.
- `offline_data`: Defines options for training with so-called offline data, a topic we cover later in this chapter.

- `multi_agent`: Specifies options for training algorithms using *multiple agents*. We discuss an explicit example of this in the next section.

The algorithm-specific configuration in `training()` only becomes more relevant once you've settled on an algorithm and want to squeeze it for performance. In practice, RLlib provides you with good defaults to get started.

You can look up configuration arguments in the API reference for RLlib algorithms for more details on configuring RLlib experiments. But before we move on to discussing examples, at the very least you should know about the most common configuration options in practice first.

Resource Configuration

Whether you use Ray RLlib locally or on a cluster, you can specify the resources used for the training process. Here are the most important options to consider. We continue using the `DQN` algorithm as example, but this would apply to any other RLlib algorithm as well.

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

config = DQNConfig().resources(
    num_gpus=1,  ❶
    num_cpus_per_worker=2, ❷
    num_gpus_per_worker=0, ❸
)
```

❶ Specify the number of GPUs to use for training. It's important to check whether your algorithm of choice supports GPUs first. This value can also be fractional. For example, if using four rollout workers in `DQN` (`num_rollout_workers` = 4), you can set `num_gpus=0.25` to pack all four workers on the same GPU, so

that all rollout workers benefit from the potential speedup. This only affects the local learner process, not the rollout workers.

❷ Set the number of CPUs to use for each rollout worker.

❸ Set the number of GPUs used per worker.

Rollout Worker Configuration

RLlib also lets you configure how your rollouts are computed and how to distribute them:

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

config = DQNConfig().rollouts(
    num_rollout_workers=4,  ❶
    num_envs_per_worker=1, ❷
    create_env_on_local_worker=True, ❸
)
```

❶ You've seen this one already. It's used to specify the number of Ray workers to use.

❷ Specify the number of environments to evaluate per worker. This setting allows you to "batch" evaluation of environments. In particular, if your models take a long time to evaluate, grouping environments like this can speed up training.

❸ When `num_rollout_workers` > 0, the driver ("local worker") does not need an environment. That's because sampling and evaluation is done by the rollout workers. If you still want an environment on the driver, you can set this option to `True`.

Environment Configuration

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

config = DQNConfig().environment(
    env="CartPole-v1",  ❶
    env_config={"my_config": "value"}, ❷
    observation_space=None,
    action_space=None, ❸
```

```
        render_env=True, ❹
    )
```

❶ Specify the environment you want to use for training. This can either be a string of an environment known to Ray RLlib, such as any `gym` environment, or the class name of a custom environment you've implemented.[14]

❷ You can optionally specify a dictionary of configuration options for your environment that will be passed to the environment constructor.

❸ You can specify the observation and action spaces of your environment, too. If you don't specify them, they will be inferred from the environment.

❹ `False` by default, this property allows you to turn on rendering of the environment, which requires you to implement the `render` method of your environment.

Note that we left out many available configuration options for each of the types we listed. On top of that, we can't touch on aspects here that alter the behavior of the RL training procedure in this introduction (like modifying the underlying model to use). But the good news is that you'll find all the information you need in the RLlib Training API documentation.

Working with RLlib Environments

So far we've only introduced you to `gym` environments, but RLlib supports a wide variety of environments. After giving you a quick overview of all available options, we'll show you two concrete examples of advanced RLlib environments in action.

An Overview of RLlib Environments

All available RLlib Environments extend a common `BaseEnv` class. If you want to work with several copies of the same `gym.Env`

environment, you can use RLlib's `VectorEnv` wrapper. Vectorized environments are useful, but also straightforward generalizations of what you've seen already. The two other types of environments available in RLlib are more interesting and deserve more attention.

The first is called `MultiAgentEnv`, which allows you to train a model with *multiple agents*. Working with multiple agents can be tricky. That's because you have to take care of defining your agents within your environment with a suitable interface, and account for the fact that each agent might have a completely different way of interacting with its environment.

What's more is that agents might interact with each other, and have to respect each other's actions. In more advanced setting there might even be a *hierarchy* of agents, which explicitly depend on each other. In short, running multi-agent RL experiments is difficult, and we'll see how RLlib handles this in the next example.

The other type of environment we will look at is called `ExternalEnv`, which can be used to connect external simulators to RLlib. For instance, imagine our simple maze problem from earlier was a simulation of an actual robot navigating a maze. It might not be suitable in such scenarios to co-locate the robot (or its simulation, implemented in a different software stack) with RLlib's learning agents. To account for that, RLlib provides you with a simple client-server architecture for communicating with external simulators, which allows communication over a REST API. In case you want to work both in a multi-agent and external environment setting, RLlib offers a `MultiAgentExternalEnv` environment that combines both.

In Figure 4-1 we summarize all available RLlib environments for you:

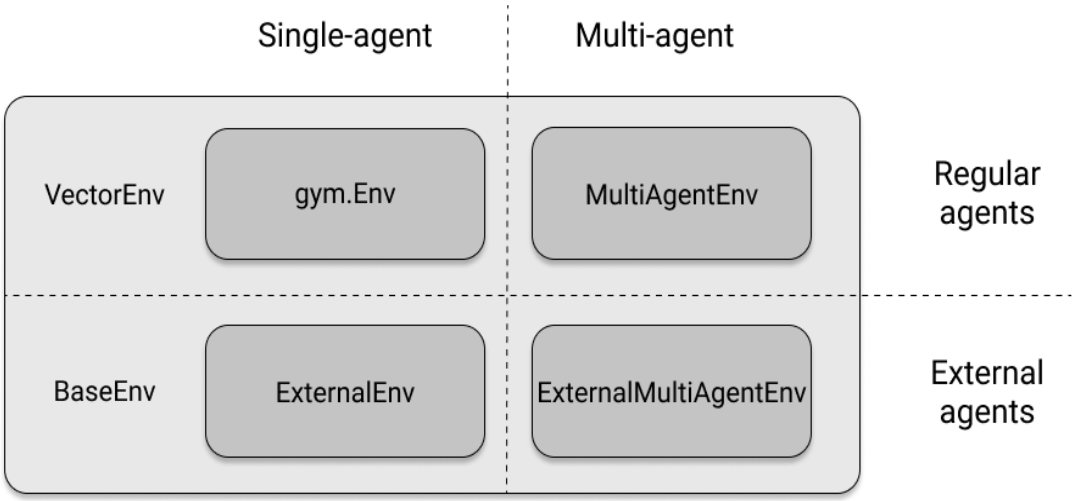


Figure 4-1. An overview of all available RLlib environments.

Working with Multiple Agents

The basic idea of defining multi-agent environments in RLlib is simple. Whatever you define as a single value in a gym environment, you now assign an agent ID to each agent and use it as key in a dictionary, and define values per-agent. Of course, the details are a little more complicated than that in practice. But once you have defined an environment hosting several agents, you have to define how these agents should learn.

In a single-agent environment there's one agent and one policy to learn. In a multi-agent environment there are multiple agents that might map to one or several policies. For instance, if you have a group of homogenous agents in your environment, then you could define a single policy for all of them. If they all *act* the same way, then their behavior can be learnt the same way. In contrast, you might have situations with heterogeneous agents in which each of them has to learn a separate policy. Between these two extremes, there's a spectrum of possibilities displayed in Figure 4-2:

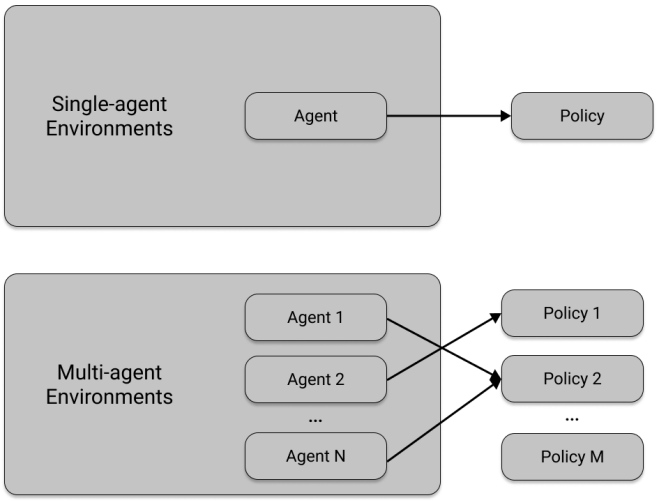


Figure 4-2. Mapping agents to policies in multi-agent reinforcement learning problems.

We continue to use our maze game as a running example for this chapter. This way you can check for yourself how the interfaces differ in practice. So, to put the ideas we just outlined into code, let's define a multi-agent version of the `GymEnvironment` class. Our `MultiAgentEnv` class will have precisely two agents, which we encode in a Python dictionary called `agents`, but in principle this works with any number of agents. We start by initializing and resetting our new environment:

```
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 Action and observation spaces stay exactly the same as before.
❷ We now have two seekers with `(0, 4)` and `(4, 0)` starting positions in an `agents` dictionary.
❸ For the `info` object we're using agent IDs as keys.
❹ Observations are now per-agent dictionaries.

Notice that we didn't touch the action and observation spaces at all. That's because we're using two essentially identical agents here that can re-use the same spaces. In more complex situations you'd have to account for the fact that the actions and observations might look different for some agents.[15]

To continue, let's generalize our helper methods `get_observation`, `get_reward`, and `is_done` to work with multiple agents. We do this by passing in an `action_id` to their signatures and handling each agent the same way as before.

```
    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
```

Next, to port the `step` method to our multi-agent setup, you have to know that `MultiAgentEnv` now expects the `action` passed to a `step` to be a dictionary with keys corresponding to the agent IDs,

too. We define a step by looping through all available agents and acting on their behalf.[16]

```
    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:  # move down
                seeker = (min(seeker[0] + 1, 4), seeker[1])
            elif action[agent_id] == 1:  # move left
                seeker = (seeker[0], max(seeker[1] - 1, 0))
            elif action[agent_id] == 2:  # move up
                seeker = (max(seeker[0] - 1, 0), seeker[1])
            elif action[agent_id] == 3:  # move right
                seeker = (seeker[0], min(seeker[1] + 1, 4))
            else:
                raise ValueError("Invalid action")
            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 Actions in a `step` are now per-agent dictionaries.
❷ After applying the correct action for each seeker, we set the correct states of all `agents`.
❸ `observations`, `rewards`, and `dones` are also dictionaries with agent IDs as keys.
❹ Additionally, RLlib needs to know when all agents are done.

The last step is to modify rendering the environment, which we do by denoting each agent by its ID when printing the maze to the screen.

```
    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]))
```

Randomly rolling out an episode until *one* of the agents reaches the goal can for instance be done by the following code:[17]

```
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
```

Note how we have to make sure to pass two random samples by means of a Python dictionary into the `step` method, and how we check if any of the agents are `done` yet. We use this `break` condition for simplicity, as it's highly unlikely that both seekers find their way to the goal at the same time by chance. But of course we'd like both agents to complete the maze eventually.

In any case, equipped with our `MultiAgentMaze`, training an RLlib `Algorithm` works *exactly* the same way as before:

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

simple_trainer =
```

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

This covers the most simple case of training a multi-agent reinforcement learning (MARL) problem. But if you remember what we said earlier, when using multiple agents there's always a mapping between agents and policies. By not specifying such a mapping, both of our seekers were implicitly assigned to the same policy. This can be changed by calling the `.multi_agent` method on our `DQNConfig` and setting the `policies` and `policy_mapping_fn` arguments accordingly:

```
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
    ).build()

print(algo.train())
```

❶ We first define multiple `policies` for our agents, each with a different `"gamma"` value.

❷ Each agent can then be mapped to a policy with a custom `policy_mapping_fn`.

As you can see, running multi-agent RL experiments is a first-class citizen of RLlib, and there's a lot more that could be said about it.

The support of MARL problems is probably one of RLlib's strongest features.

Working with Policy Servers and Clients

For the last example in this section on environments, let's assume our original `GymEnvironment` can only be simulated on a machine that can't run RLlib, for instance because it doesn't have enough resources available. We can run the environment on a `PolicyClient` that can ask a respective *server* for suitable next actions to apply to the environment. The server, in turn, does not know about the environment. It only knows how to ingest input data from a `PolicyClient`, and it is responsible for running all RL related code, in particular it defines an RLlib `AlgorithmConfig` object and trains an `Algorithm`.

Typically, you want to run the server that trains your algorithm on a powerful Ray cluster, and then the respective client runs outside that cluster. Figure Figure 4-3 schematically illustrates this setup.

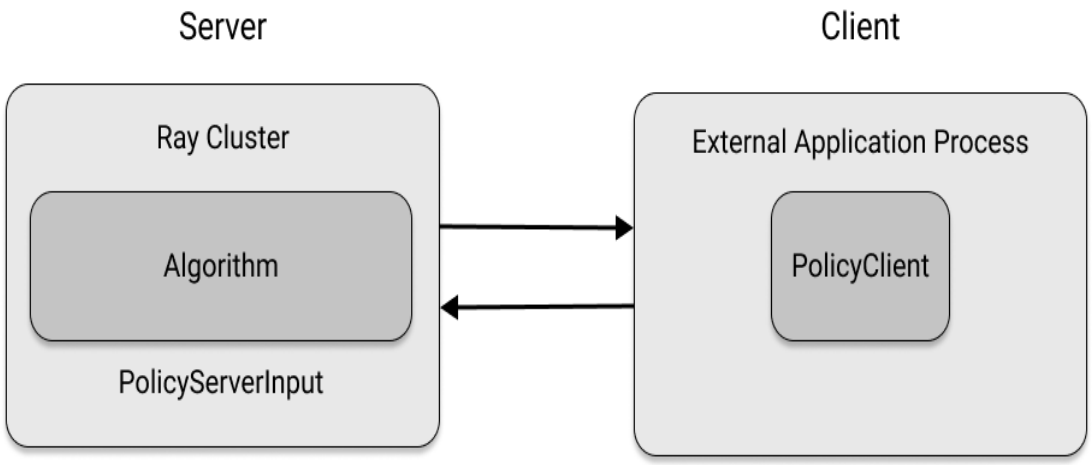


Figure 4-3. Working with policy servers and clients in RLlib.

Defining a Server

Let's start by defining the server-side of such an application first. We define a so-called `PolicyServerInput` that runs on `localhost` on port `9900`. This policy input is what the client will provide later on. With this `policy_input` defined as `input` to our algorithm configuration, we can define yet another `DQN` to run on the server:

```
# 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)
❶


config = DQNConfig()\
    .environment(
        env=None,  ❷
        action_space=gym.spaces.Discrete(4),  ❸
        observation_space=gym.spaces.Discrete(5*5))\
    .debugging(log_level="INFO")\
    .rollouts(num_rollout_workers=0)\
    .offline_data(  ❹
        input=policy_input,
        input_evaluation=[])\


algo = config.build()
```

❶ The `policy_input` function returns a `PolicyServerInput` object running on localhost on port 9900.

❷ We explicitly set the `env` to `None` because this server does not need one.

❸

We therefore need to define both an `observation_space` and an `action_space`, as the server is not able to infer them from the environment.

❹ To make this work, we need to feed our `policy_input` into the experiment's `input`.

With this `algo` defined,[18] we can now start a training session on the server like so:

```
# policy_server.py
if __name__ == "__main__":

    time_steps = 0
    for _ in range(100):
        results = algo.train()
        checkpoint = algo.save()  ❶
        if time_steps >= 1000:  ❷
            break
        time_steps += results["timesteps_total"]
```

❶ We train for a maximum of 100 iterations and store checkpoints after each iteration.

❷ If training surpasses 1000 time steps, we stop the training.

In what follows we assume that you store the last two code snippets in a file called `policy_server.py`. If you want to, you can now start this policy server on your local machine by running `python policy_server.py` in a terminal.

Defining a Client

Next, to define the corresponding client-side of the application, we define a `PolicyClient` that connects to the server we just started. Since we can't assume that you have several computers at home (or available in the cloud), contrary to what we said prior, we will start this client on the same machine. In other words, the client will connect to `http://localhost:9900`, but if you can run the server

on different machine, you could replace `localhost` with the IP address of that machine, provided it's available in the network.

Policy clients have a fairly lean interface. They can trigger the server to start or end an episode, get next actions from it, and log reward information to it (that it would otherwise not have). With that said, here's how you define such a client.

```
# 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
```

❶ We start a policy client on the server address with `remote` inference mode.

❷ Then we tell the server to start an episode.

❸ For given environment observations, we can get the next action from the server.

❹ It's mandatory for the `client` to log reward information to the server.

❺ If a certain condition is reached, we can stop the client process.
❻ If the environment is `done`, we have to inform the server about episode completion.

Assuming you store this code under `policy_client.py` and start it by running `python policy_client.py`, then the server that we started earlier will start learning with environment information solely obtained from the client.

Advanced Concepts

So far we've been working with simple environments that were easy enough to tackle with the most basic RL algorithm settings in RLlib. Of course, in practice you're not always that lucky and might have to come up with other ideas to tackle harder environments. In this section we're going to introduce a slightly harder version of the maze environment and discuss some advanced concepts that help you solve this environment.

Building an Advanced Environment

Let's make our maze `GymEnvironment` a bit more challenging. First, we increase its size from a `5x5` to a `11x11` grid. Then we introduce obstacles in the maze that the agent can pass through, but only by incurring a penalty, a negative reward of `-1`. This way our seeker agent will have to learn to avoid obstacles, while still finding the goal. Also, we randomize the starting position of the agent. All of this makes the RL problem harder to solve. Let's have a look at the initialization of this new `AdvancedEnv` first:

```
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:  ❶
            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 = [  ❷
            (i, j) for i in range(self.maze_len) for j in
range(self.maze_len)
            if i % 2 == 1 and j % 2 == 0
        ]
```

❶ We can now set the `seeker` position upon initialization.
❷ We introduce `punish_states` as obstacles for the agent.

Next, when resetting the environment, we want to make sure to reset the agent's position to a random state.[19] We also increase the positive reward for reaching the goal to `5`, to offset the negative reward for passing through an obstacle (which will happen a lot before the RL algorithm picks up on the obstacle locations). Balancing rewards like this is a crucial task in calibrating your RL experiments.

```
    def reset(self):
        """Reset seeker position randomly, return
observations."""
        self.seeker = (
            random.randint(0, self.maze_len - 1),
```

```
                random.randint(0, self.maze_len - 1)
            )
            return self.get_observation()

        def get_observation(self):
            """Encode the seeker position as integer"""
            return self.maze_len * self.seeker[0] +
    self.seeker[1]

        def get_reward(self):
            """Reward finding the goal and punish forbidden
    states"""
            reward = -1 if self.seeker in self.punish_states
    else 0
            reward += 5 if self.seeker == self.goal else 0
            return reward

        def render(self, *args, **kwargs):
            """Render the environment, e.g. by printing its
    representation."""
            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]))
```

There are many other ways you could make this environment harder, like making it much bigger, introducing a negative reward for every step the agent takes in a certain direction, or punishing the agent for trying to walk off the grid. By now you should understand the problem setting well enough to customize the maze yourself further.

While you might have success training this environment, this is a good opportunity to introduce some advanced concepts that you can apply to other RL problems.

Applying Curriculum Learning

One of the most interesting features of RLlib is to provide an `Algorithm` with a *curriculum* to learn from. What that means is that, instead of letting the algorithm learn from arbitrary environment setups, we cherry-pick states that are much easier to learn from and then slowly but surely introduce more difficult states. Building a learning curriculum this way is a great way to make your experiments converge to solutions quicker. The only thing you need to apply curriculum learning is a view on which starting states are easier than others. For many environments that can actually be a challenge, but it's easy to come up with a simple curriculum for our advanced maze. Namely, the distance of the seeker from the goal can be used as a measure of difficulty. The distance measure we'll use for simplicity is the sum of the absolute distance of both seeker coordinates from the goal to define a `difficulty`.

To run curriculum learning with RLlib, we define a `CurriculumEnv` that extends both our `AdvancedEnv` and a so-called `TaskSettableEnv` from RLLib. The interface of `TaskSettableEnv` is very simple, in that you only have to define how to get the current difficulty (`get_task`) and how to set a required difficulty (`set_task`). Here's the full definition of this `CurriculumEnv`:

```
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 We define the `difficulty` of the current state as the sum of the absolute distance of both seeker coordinates from the goal.

❷ To define `get_task` we can then simply return the current `difficulty`.

❸ To set a task difficulty, we `reset` the environment until its `difficulty` is *at most* the specified `task_difficulty`.

To use this environment for curriculum learning, we need to define a curriculum function that tells the algorithm when and how to set the task difficulty. We have many options here, but we use a schedule that simply increases the difficulty by one every `1000` time steps trained:

```
def curriculum_fn(train_results, task_settable_env,
env_ctx):
    time_steps = train_results.get("timesteps_total")
    difficulty = time_steps // 1000
    print(f"Current difficulty: {difficulty}")
    return difficulty
```

To test this curriculum function, we need to add it to our RLlib algorithm `config`, namely by setting the `env_task_fn` property to our `curriculum_fn`. Note that before training a `DQN` for `15` iterations, we also set an `output` folder in our config. This will store experience data of our training run to the specified `temp` folder.[20]

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


temp = tempfile.mkdtemp()  ❶

trainer = (
    DQNConfig()
```

```
    .environment(env=CurriculumEnv,
env_task_fn=curriculum_fn)   ❷
    .offline_data(output=temp)   ❸
    .build()
)

for i in range(15):
    trainer.train()
```

❶ First, we create a `temp` file to store our training data for later use.

❷ In the `environment` part of our config we set the `CurriculumEnv` as our environment and assign our `curriculum_fn` to the `env_task_fn` property.

❸ We then use the `offline_data` method to store `output` in our `temp` folder.

Running this algorithm, you should see how the task difficulty increases over time, thereby giving the algorithm easy examples to start with so that in can learn from them and progress to more difficult tasks as it progresses.

Curriculum learning is a great technique to be aware of and RLlib allows you to easily incorporate it into your experiments through the curriculum API we just discussed.

Working with Offline Data

In our previous curriculum learning example we stored training data to a temporary folder. What's interesting is that you already know from Chapter 3 that in Q-learning you can collect experience data first, and decide when to use it in a training step later. This separation of data collection and training opens up many possibilities. For instance, maybe you have a good heuristic that can solve your problem in an imperfect, yet reasonable manner. Or you have records of human interaction with your environment, demonstrating how to solve the problem by example.

The topic of collecting experience data for later training is often discussed as working with *offline data*. It's called offline, as it's not directly generated by a policy interacting online with the environment. Algorithms that don't rely on training on their own policy output are called off-policy algorithms, and Q-Learning, respectively `DQN`, is just one such example. Algorithms that don't share this property are accordingly called on-policy algorithms. In other words, off-policy algorithms can be used to train on offline data.[21]

To use the data we stored in the `temp` folder before, we can create a new `DQNConfig` that takes this folder as `input`. We will also set `explore` to `False` , since we simply want to exploit the data previously collected for training - the algorithm will not explore according to its own policy.

Using the resulting RLlib algorithm works exactly as before, which we demonstrate by training it for `10` iterations and then evaluating it:

```
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()
```

Note that we called the algorithm `imitation_algo`. That's because this training procedure intends to *imitate* the behavior reflected in the data we collected before. This type of learning by demonstration in RL is therefore often called *imitation learning* or *behavior cloning*.

Other Advanced Topics

Before concluding this chapter, let's have a look at a few other advanced topics that RLlib has to offer. You've already seen how flexible RLlib is, from working with a range of different environments, to configuring your experiments, training on a curriculum, or running imitation learning. To give you a taste of what else is possible, you can also do the following things with RLlib:

- You can completely customize the models and policies used under the hood. If you've worked with deep learning before, you know how important it can be to have a good model architecture in place. In RL this is often not as crucial as in supervised learning, but still a vital part of running advanced experiments successfully.
- You can change the way your observations are preprocessed by providing custom preprocessors. For our simple maze examples there was nothing to preprocess, but when working with image or video data, preprocessing is often a crucial step.
- In our `AdvancedEnv` we introduced states to avoid. Our agents had to learn to do this, but RLlib has a feature to automatically avoid them through so-called *parametric action spaces*. Loosely speaking, what you can do is to "mask out" all undesired actions from the action space for each point in time.
- In some cases it can also be necessary to have variable observation spaces, which is also fully supported by RLlib.
- We only briefly touched on the topic of offline data. Rllib has a full-fledged Python API for reading and writing experience data that can be used in various situation.
- Lastly, I want to stress again that we have solely worked with `DQN` here for simplicity, but RLlib has an impressive range of training algorithms. To name just one, the MARWIL algorithm is a complex hybrid algorithm with which you can run imitation

learning from offline data, while also mixing in regular training on data generated “online”.

Summary

To summarize, you’ve seen a selection of interesting features of RLlib in this chapter. We covered training multi-agent environments, working with offline data generated by another agent, setting up a client-server architecture to split simulations from RL training, and using curriculum learning to specify increasingly difficult tasks.

We’ve also given you a quick overview of the main concepts underlying RLlib, and how to use its CLI and Python API. In particular, we’ve shown how to configure your RLlib algorithms and environments to your needs. As we’ve only covered a small part of the possibilities of RLlib, we encourage you to read its documentation and explore its API.

In the next chapter you’re going to learn how to tune the hyperparameters of your RLlib models and policies with Ray Tune.

---

1 We’re merely using a simple game to illustrate the process of RL. There is a multitude of interesting industry applications of RL that are not games.

2 We don’t cover this integration in this book, but you can learn more about deploying RLlib models in the “Serving RLlib Models” tutorial on the Ray documentation.

3 From Ray 2.3.0 onwards, RLlib will be using the `gymnasium` library as drop-in replacement for `gym`. This will likely introduce some breaking changes, so it’s best to stick with Ray 2.2.0 to follow this chapter.

4 `gym` comes with a variety of interesting environments that are worth exploring. For instance, you can find many of the Atari environments that were used in the famous “Playing Atari with Deep Reinforcement Learning” paper from DeepMind, or advanced physics simulations using the MuJoCo engine.

5 To be precise, RLlib uses a double- and dueling DQN.

6 In the GitHub repo for this book we've also included an equivalent `maze.yml` file that you could use via `rllib train file maze.yml` (no `--type` needed).

7 Of course, configuring your models is a crucial part of RL experiments. We will discuss configuration of RLlib algorithms in more detail in the next section.

8 If you set `num_rollout_workers` to `0`, only the local worker on the head node will be created, and all sampling from the env is done there. This is particularly useful for debugging, as no additional Ray actor processes are spawned.

9 Technically speaking, only the *local* model is used for actual training. The two worker models are used for action computation and data collection (rollouts). After each training step, the local model sends its current weights to the workers for synchronization. Fully distributed training, as opposed to distributed sampling, will be available across all RLlib algorithms in a future Ray versions.

10 This is true by default, since we're using TensorFlow and Keras under the hood. Should you opt to change the `framework` specification of your algorithm to work with PyTorch directly do `print(model)`, in which case `model` is-a `torch.nn.Module`. In a future release the access to the underlying model will be unified across all frameworks.

11 In case you're not familiar with these types of networks, the "value" output of this network represents the Q-value of state-action pairs.

12 If you want to learn more about customizing your RLlib models, check out the guide to custom models on the Ray documentation.

13 We only list the ones we introduce in this chapter. Apart from those we mention, you also find options for `evaluation` of your algorithms, `reporting`, `debugging`, `checkpointing`, adding `callbacks`, altering your deep learning `framework`, requesting `resources` and accessing `experimental` features.

14 There's also a way to *register* your environments so that you can refer to them by name, but this requires using Ray Tune. You will learn about this feature in Chapter 5.

15 You can find a good example that defines different observation and actions spaces for multiple agents on the RLlib documentation.

16 Note how this can lead to issues like deciding which agent gets to act first. In our simple maze problem the order of actions is irrelevant, but in more complex scenarios this becomes a crucial part of modeling the RL problem correctly.

17 Deciding when an episode is done is a crucial part of multi-agent RL, and it depends entirely on the problem at hand and what you want to achieve.

18 For technical reasons we have to specify observation and action spaces here, which might not be necessary in future releases of RLlib, as it leaks environment information. Also note that we need to set `input_evaluation` to an empty list to make this server work.

19 In the definition of `reset` below, we allow the seeker to reset on top of the goal to keep the definition simpler. Allowing this trivial edge case does not affect learning.

20 Note that if you run the notebook for this chapter on the cloud, the training process could take a while to finish.

21 Note that RLlib has a wide range of on-policy algorithms like `PPO` as well.

Chapter 5. Hyperparameter Optimization with Ray Tune

In the last chapter we've seen how to build and run various reinforcement learning experiments. Running such experiments can be expensive, both in terms of compute resources and the time it takes to run them. This expense only gets amplified as you move on to more challenging tasks, since it is unlikely that you can just pick an algorithm out of the box and run it to get a good result. In other words, at some point you'll need to tune the hyperparameters of your algorithms to get the best results. As we'll see in this chapter, tuning machine learning models is hard, but Ray Tune is an excellent choice to help you tackle this task.

Ray Tune is a powerful tool for hyperparameter optimization. Not only does it work in a distributed manner by default (and works in any other Ray library discussed in this book), but it's also one of the most feature-rich hyperparameter optimization (HPO) libraries available. To top this off, Tune integrates with some of the most prominent HPO libraries out there, such as HyperOpt, Optuna, and many more. This makes Tune an ideal candidate for distributed HPO experiments, practically no matter what other libraries you're coming from or if you start from scratch.

In this chapter we'll first revisit in a bit more depth why HPO is hard to do, and how you could naively implement it yourself with Ray. We then teach you the core concepts of Ray Tune and how you can use it to tune the RLlib models we've built in the previous chapter. To wrap things up, we'll also have a look at how to use Tune for supervised learning tasks, using frameworks like Keras. Along the way, we demonstrate how Tune integrates with other HPO libraries and introduce you to some of its more advanced features.

Tuning Hyperparameters

Let's recap the basics of hyperparameter optimization briefly. If you're familiar with this topic, you can skip this section, but since we're also discussing aspects of distributed HPO, you might still benefit from following along. As always, the notebook for this chapter can be found in the GitHub repository of this book.

If you recall our first RL experiment introduced in Chapter 3, we defined a very basic Q-learning algorithm whose internal *state-action values* were updated according to an explicit update rule. After initialization, we never touched these *model parameters* directly, they were learnt by the algorithm. By contrast, in setting up the algorithm, we explicitly chose a `weight` and a `discount_factor` parameter prior to training. I didn't tell you how we chose to set these parameters back then, we simply accepted the fact that they were good enough to crack the problem at hand.

In the same way, in Chapter 4 we initialized an RLlib algorithm with a `config` that used a total of four rollout workers for our DQN algorithm by setting `num_workers=4`. Parameters like these are called *hyperparameters*, and finding good choices for them can be crucial for successful experiments. The field of hyperparameter optimization entirely is devoted to efficiently finding such good choices.

Building a Random Search Example with Ray

Hyperparameters like the `weight` or the `discount_factor` of our Q-learning algorithm are *continuous* parameters, so we can't possibly test all combinations of them. What's more, these parameter choices may not be independent of each other. If we want them to be selected for us, we also need to specify a *value range* for each of them (both hyperparameters need to be chosen between 0 and 1 in this case). So, how do we determine good or even optimal hyperparameters?

Let's take a look at a quick example that implements a naive, yet effective approach to tuning hyperparameters. This example will also allow us to introduce some terminology that we'll use later on. The core idea is that we can attempt to *randomly sample* hyperparameters, run the algorithm for each sample, and then select the best run based on the results we got. But to do the theme of this book justice, we don't just want to run this in a sequential loop, we want to compute our runs in parallel using Ray.

To keep things simple we'll revisit our simple Q-learning algorithm from Chapter 3 again. If you don't recall the signature of the main training function, we defined it as `train_policy(env, num_episodes=10000, weight=0.1, discount_factor=0.9)`. That means we can tune the `weight` and `discount_factor` parameters of our algorithm by passing in different values to the `train_policy` function and see how the algorithm performs. To do that, let's define a so-called *search space* for our hyperparameters. For both parameters in question we simply uniformly sample values between 0 and 1, for a total of 10 choices. Here's what that looks like:

```
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)
```

Next, we define an *objective function*, or simply *objective*. The role of an objective function is to evaluate the performance of a given set of hyperparameters for the task we're interested in. In our case, we want to train our RL algorithm and evaluate the trained policy. Recall that in Chapter 3 we also defined an `evaluate_policy` function for precisely this purpose. The `evaluate_policy` function was defined to return the average number of steps it took for an agent to reach the goal in the underlying maze environment. In other words, we want to find a set of hyperparameters that minimizes the result of our objective function. To parallelize the objective function, we'll use the `ray.remote` decorator to make our `objective` a Ray task.

```
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 We pass in a dictionary with a hyperparameter sample into our objective.
❷ Then we train our RL policy using the chosen hyperparameters.
❸ Afterwards we can evaluate the policy to retrieve the score we want to minimize.
❹ We return both score and hyperparameter choice together for later analysis.

Finally, we can run the objective function in parallel using Ray, by iterating over the search space and collecting the results:

```
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])
```

The actual results of this hyperparameter run are not very interesting, as the problem is so easy to solve (most runs will return the optimum of 8 steps, regardless of the hyperparameters chosen). What's more interesting here is how easy it is to parallelize the objective function with Ray. In fact, I'd like to encourage you to rewrite the above example to simply loop through the search space and call the objective function for each sample, just to confirm how painfully slow such a serial loop can be.

Conceptually, the three steps we took to run the above example are representative of how hyperparameter tuning works in general. First, you define a search space, then you define an objective function, and finally you run an analysis to find the best hyperparameters. In HPO it is common to speak of one evaluation of the objective function per hyperparameter sample as a *trial*, and all trials form the basis for your analysis. How exactly parameters are sampled from the search space (in our case, randomly) is up to a *search algorithm* to decide. In practice, finding good hyperparameters is easier said than done, so let's have a closer look at why this problem is so hard.

Why Is HPO hard?

If you zoom out from the above example just enough, you can see that there are several intricacies in making the process of hyperparameter tuning work well. Here's a quick overview of the most important ones:

- Your search space can be composed of a large number of hyperparameters. These parameters might have different data types and ranges. Some parameters might be correlated, or even depend on others. Sampling good candidates from complex, high-dimensional spaces is a difficult task.
- Picking parameters at random can work surprisingly well, but it's not always the best option. In general, you need to test more complex search algorithms to find the best parameters.
- In particular, even if you parallelize your hyperparameter search like we just did, a single run of your objective function can take a long time to complete. That means you can't afford to run too many searches overall. For instance, training neural networks can take hours to complete, so your hyperparameter search better be efficient.
- When distributing search, you need to make sure to have enough compute resources available to run searches over the objective function effectively. For instance, you might need a GPU to compute your objective function fast enough, so all your search runs need to have access to a GPU. Allocating the necessary resources for each trial is critical to speeding up your search.
- You want to have convenient tooling for your HPO experiments, like stopping bad runs early, saving intermediate results, restarting from previous trials, or pausing and resuming runs, etc.

As a mature, distributed HPO framework, Ray Tune takes addresses all these topics and provides you with a simple interface for running hyperparameter tuning experiments. Before we look into how Tune works, let's rewrite our above example to use Tune.

An Introduction to Tune

To get your first taste of Tune, porting over our naive Ray Core implementation of random search to Tune is straightforward and follows the same three steps as before. First, we define a search space, but this time using `tune.uniform`, instead of the `random` library:

```
from ray import tune


search_space = {
    "weight": tune.uniform(0, 1),
    "discount_factor": tune.uniform(0, 1),
}
```

Next, we can define an objective function that almost looks the same as before. We designed it like that. The only differences are that this time we return the score as a dictionary, and that we don't need a `ray.remote` decorator, as Tune will take care of distributing this objective function for us internally.

```
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}
```

With this `tune_objective` function defined, we can pass is into a `tune.run` call, together with the search space we defined. By default,[1] Tune will run random search for you, but you can also specify other search algorithms, as you will see soon. Calling `tune.run` generates random search trials for your objective and returns an `analysis` object that contains information about the hyperparameter search. We can get the best hyperparameters found

by calling `get_best_config` and specifying the `metric` and `mode` arguments (we want to minimize our score):

```
analysis = tune.run(tune_objective, config=search_space)
print(analysis.get_best_config(metric="score", mode="min"))
```

This quick example covers the very basics of Tune, but there's a lot more to unpack. The `tune.run` function is quite powerful and takes a lot of arguments for you to configure your runs. To understand these different configuration options, we first need to introduce you to the key concepts of Tune.

How Does Tune Work?

To effectively work with Tune, you have to understand a total of six key concepts, four of which you have already used in the last example. Here's an informal overview of Ray Tune's components and how you should think about them:

- *Search spaces*: These spaces determine which parameters to select. Search spaces define the range of values for each parameter, and how they should be sampled. They are defined as dictionaries and use Tune's sampling functions to specify valid hyperparameter values. You have already seen `tune.uniform`, but there are many more options to choose from.
- *Trainables*: A `Trainable` is Tune's formal representation of an objective you want to "tune". Tune has class-based API as well, but we will only use the function-based API in this book. For us, a `Trainable` is a function with a single argument, a search space, which reports scores to Tune. The easiest way to do so is by returning a dictionary with the score you're interested in.
- *Trials*: By triggering `tune.run(...)`, Tune will make sure to set up trials and schedule them for execution on your cluster. A

trial contains all necessary information about a single run of your objective, given a set of hyperparameters.

- *Analyses*: Completing a `tune.run` call returns an `ExperimentAnalysis` object, with the results of all trials. You can use this object to drill down into the results of your trials.
- *Search Algorithms*: Tune supports a large variety of search algorithms, which are at the core of how to tune your hyperparameters. So far you've only implicitly encountered Tune's default search algorithm, which randomly selects hyperparameters from the search space.
- *Schedulers*: The last, crucial component of a Tune experiment is that of a *scheduler*. Schedulers plan and execute what the search algorithm selects. By default, Tune schedules trials selected by your search algorithm on a first-in-first-out (FIFO) basis. In practice, you can think of schedulers as a way to speed up your experiments, for instance by stopping unsuccessful trials early.

Figure Figure 5-1 sums up these major components of Tune, and their relationship in one diagram:

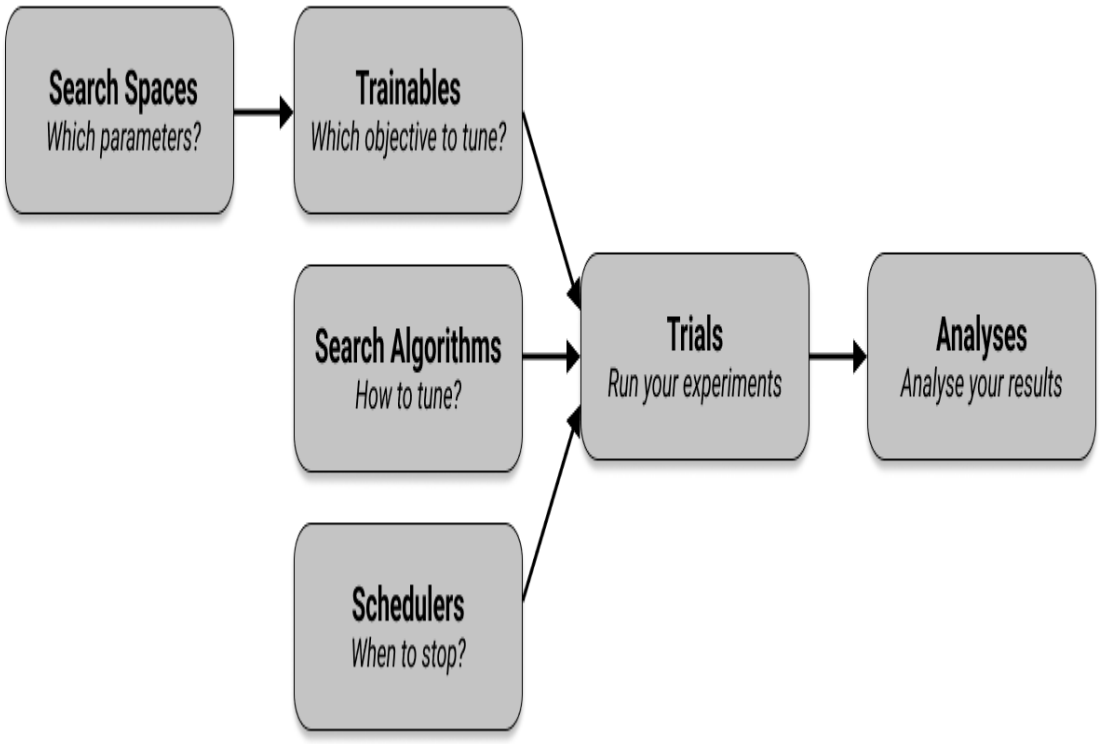


Figure 5-1. The core components of Ray Tune.

Note that internally Tune runs are started on the driver process of your Ray cluster, which spawns several worker processes (using Ray actors) that execute individual trials of your HPO experiment. Your trainables, defined on the driver, have to be sent to the workers, and trial results need to be communicated to the driver running `tune.run(...)`.

Search spaces, Trainables, trials, and analyses don't need much additional explanation, and we'll see more examples of each of those components in the rest of this chapter. But search algorithms, or simply *searchers* for short, and schedulers need a bit more elaboration.

NOTE

Throughout this chapter we exclusively use `tune.run` to illustrate Tune's functionality. Tune also has a newer API called `Tuner` that has been added in Ray 2.0 as part of the AI Runtime (Ray AIR). You will learn more about the `Tuner` API in Chapter 7 on distributed training, and how to use it within Ray AIR in Chapter 10.

At the time of this writing, `tune.run` is still the more mature API. For instance, `tune.run(...)` experiments return an `ExperimentAnalysis` object, which is a powerful tool for analyzing your results. Analogous calls using the `Tuner` API return a so-called `ResultGrid` instead. In the long run `ResultGrid` will be the successor of `ExperimentAnalysis`, but is not yet at feature parity.

If you want to learn more about the similarities and differences, please refer to the Tune API documentation on this topic.

Search Algorithms

All advanced search algorithms provided by Tune, and the many third-party HPO libraries it integrates with, fall under the umbrella of *Bayesian Optimization*. Unfortunately, going into the details of specific Bayesian search algorithms is far beyond the scope of this book. The basic idea is that you update your beliefs about which hyperparameter ranges are worth exploring based on the results of your previous trials. Techniques using this principle make more informed decisions and hence tend to be more efficient than independently sampling parameters (e.g. at random).

Apart from the basic random search we've seen already, and *grid search*, which picks hyperparameters from a predefined "grid" of choices, Tune integrates with a wide range of Bayesian optimization searchers. For instance, Tune integrates with the popular HyperOpt and Optuna libraries,[2] and you can use the popular TPE (Tree-structured Parzen Estimator) searcher with Tune through both of these libraries. Not only that, Tune also integrates with tools such as Ax, BlendSearch, FLAML, Dragonfly, Scikit-Optimize,

BayesianOptimization, HpBandSter, Nevergrad, ZOOpt, SigOpt, and HEBO. If you need to run HPO experiments with any of these tools on a cluster, or want to easily switch between them, Tune is the way to go.

To make things more concrete, let's rewrite our basic random search Tune example from earlier to use the `bayesian-optimization` library. To do so, make sure you install this library in your Python environment first, e.g. with `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},
)
```

Note that we "warm start" our Bayesian optimization with four random steps at the beginning, and we explicitly `stop` the trial runs after 10 training iterations.

Note that since we're not just randomly selecting parameters with `BayesOptSearch`, the `search_alg` we use in our Tune run needs to know which `metric` to optimize for and whether to minimize or optimize this metric. As we've argued before, we want to achieve a `"min"` `"score"`.

Schedulers

Let's next discuss how to use *trial schedulers* in Tune to make your runs more efficient. We also use this section to introduce a slightly

different way to report your metrics to Tune within an objective function.

So, let's say that instead of computing a score straight-up, like we did in all examples in this chapter so far, we compute an *intermediate score* in a loop. This is a situation that often occurs in supervised machine learning scenarios, when training a model for several iterations (we'll see concrete applications of this later in this chapter). With good hyperparameter choices selected, this immediate score might stagnate way before the loop in which it is computed. In other words, if we don't see enough incremental changes anymore, why not stop the trial early? This is exactly one of the cases Tune's schedulers are built for.

Here's a quick example of such an objective function. This is a toy example, but it will help us reason about the optimal hyperparameters we want Tune to find much better than if we were discussing a black-box scenario.

```
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 Often you may want to compute intermediate scores, e.g. in a "training loop".

❷ You can use `tune.report` to let Tune know about these intermediate scores.

The score we want to minimize here is the square root of a positive number times a `weight`, plus adding a `bias` term. It's clear that both of these hyperparameters need to be as small as possible to minimize the `score` for any positive `x`. Given that the square root

function “flattens out”, we might not have to compute all `30` passes through the loop to find sufficiently good values for our two hyperparameters. If you imagine that each `score` computation took an hour, stopping early can be a huge boost to make your experiments run quicker.

Let’s illustrate this idea by using the popular Hyperband algorithm as our trial scheduler. This scheduler needs to be passed a metric and mode (again, we `min`-imize our `score`). We also make sure to run for 10 samples so as not to stop too prematurely:

```
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"))
```

Note that in this case we did not specify a search algorithm, which means that Hyperband will run on parameters selected by random search. We could also have *combined* this scheduler with another search algorithm instead. This would have allowed us to pick better trial hyperparameters and stop bad trials early. However, note that not every scheduler can be combined with search algorithms. You’re advised to check Tune’s scheduler compatibility matrix for more information.

To wrap this discussion up, apart from Hyperband Tune includes distributed implementations of early stopping algorithms such as the Median Stopping Rule, ASHA, Population Based Training (PBT) and Population Based Bandits (PB2).

Configuring and Running Tune

Before looking into more concrete machine learning examples using Ray Tune, let's dive into some useful topics that help you get more out of your Tune experiments, such as properly utilizing resources, stopping and resuming trials, adding callbacks to your Tune runs, or defining custom and conditional search spaces.

Specifying resources

By default, each Tune trial will run on one CPU and leverage as many CPUs as available for concurrent trials. For instance, if you run Tune on a laptop with 8 CPUs, any of the experiments we computed so far in this chapter will spawn 8 concurrent trials and allocate one CPU each for each trial. Changing this behaviour can be controlled using the `resources_per_trial` argument of a Tune run.

What's interesting is that this does not stop with CPUs, you can also determine the number of GPUs used per trial. Plus, Tune allows you to use *fractional resources*, i.e., you can share resources between trials. So, let's say that you have a machine with 12 CPUs and two GPUs and you request the following resources for your `objective`:

```
from ray import tune

tune.run(
    objective,
    config=search_space,
    num_samples=10,
    resources_per_trial={"cpu": 2, "gpu": 0.5}
)
```

That means Tune can schedule and execute up to four concurrent trials on your machine, as this would max out GPU utilization on this machine (while you'd still have 4 idle CPUs for other tasks). If you want, you can also specify the amount of `"memory"` used by a trial by passing the number of bytes into `resources_per_trial`. Also note that should you have the need to explicitly *restrict* the number of

concurrent trials, you can do so by passing in the `max_concurrent_trials` parameter to your `tune.run(...)`. In the above example, let's say you want to always keep one GPU available for other tasks, you can limit the number of concurrent trials to two by setting `max_concurrent_trials = 2`.

Note that everything we just exemplified for resources on a single machine naturally extends to any Ray cluster and its available resources. In any case, Ray will always try to schedule the next trials, but will wait and ensure enough resources are available before executing them.

Callbacks and Metrics

If you've spent some time investigating the outputs of the Tune runs we've started in this chapter so far, you'll have noticed that each trial comes equipped with a lot of information by default, such as the trial ID, its execution date, and much more. What's interesting is that Tune not only allows you to customize the metrics you want to report, you can also hook into a `tune.run` by providing *callbacks*. Let's compute a quick, representative example that does both.

Slightly modifying a previous example, let's say we want to log a specific message whenever a trial returns a result. To do so, all you need to do is implement the `on_trial_result` method on a `Callback` object from the `ray.tune` package.[3] Here's how that would look like for an objective function that reports a `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"Trial {trial} in iteration {iteration}, "
              f"got result: {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=
{})
```

Note that, apart from the score, we also report `step` and `more_metrics` to Tune. In fact, you could expose any other metric you'd like to track there, and Tune would add it to its trial metrics. Here's how you'd run a Tune experiment with our custom callback, and print the custom metrics we just defined:

```
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))
```

Running this code will result in the following outputs (additional to what you'll see in any other Tune run). Note that we need to specify `mode` and `metric` explicitly here, so that Tune knows what we mean by `best_result`. First, you should see the output of our callback, while the trials are running:

```
...
Trial objective_85955_00000 in iteration 57, got result:
1.5379782083952644
Trial objective_85955_00000 in iteration 58, got result:
1.5539087627537493
Trial objective_85955_00000 in iteration 59, got result:
1.569535794562848
```

```
Trial objective_85955_00000 in iteration 60, got result:
1.5848760187255326
Trial objective_85955_00000 in iteration 61, got result:
1.5999446700996236
...
```

Then, at the very end of the program, we print the metrics of the best available trial, which includes the three custom metrics we defined. The following output omits some default metrics to make it more readable. We recommend that you run an example like this on your own, in particular to get used to reading the outputs of Tune trials (which can be a bit overwhelming due to their concurrent nature).

```
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'
...
```

Note that we used `on_trial_result` as an example of a method to implement a custom Tune `Callback`, but you have many other useful options that are all relatively self-explanatory. It's not very helpful to list them all here, but some callback methods I find particularly useful are `on_trial_start`, `on_trial_error`, `on_experiment_end` and `on_checkpoint`. The latter hints at an important aspect of Tune runs that we'll discuss next.

Checkpoints, Stopping, and Resuming

The more Tune trials you kick off and the longer they each run individually, especially in a distributed setting, the more you need a mechanism to protect you against failures, to stop a run, or pick a run up again from previous results. Tune makes this possible by periodically creating *checkpoints* for you. The checkpoint cadence is dynamically adjusted by Tune to ensure at least 95% of the time is spent on running trials, and not too many resources are devoted to storing checkpoints.

In the example we just computed, the checkpoint directory, or `logdir`, used by default is of the form `~/ray_results/<your-objective>_<date>_<time>`. If you know this checkpoint directory of your experiment, you can easily `resume` it like so:

```
analysis = tune.run(
    objective,
    name="<your-logdir>",
    resume=True,
    config=search_space)
```

Similarly, you can *stop* your trials by defining stopping conditions and explicitly passing them to your `tune.run`. The easiest option to do that, and we've already seen this option before, is by providing a dictionary with a stopping condition. Here's how you stop running our `objective` analysis after reaching a `training_iteration` count of 10, a built-in metric of all Tune runs:

```
tune.run(
    objective,
    config=search_space,
    stop={"training_iteration": 10})
```

One of the drawbacks of this way of specifying a stopping condition is that it assumes the metric in question is *increasing*. For instance, the `score` we compute starts high and is something we want to minimize. To formulate a flexible stopping condition for our `score`, the best way is to provide a stopping function as follows.

```
def stopper(trial_id, result):
    return result["score"] < 2


tune.run(
    objective,
    config=search_space,
    stop=stopper)
```

In situations that require a stopping condition with more context or explicit state, you can also define a custom `Stopper` class to pass into the `stop` argument of your Tune run, but we won't cover this case here.

Custom and conditional search spaces

The last more advanced topic we're going to cover here is that of complex search spaces. So far, we've only looked at hyperparameters that were independent of each other, but in practice it happens quite often that some depend on others. Also, while Tune's built-in search spaces have quite a lot to offer, sometimes you might want to sample parameters from a more exotic distribution or your own modules.

Here's how you can handle both situations in Tune. Continuing with our simple `objective` example, let's say that instead of Tune's `tune.uniform` you want to use the `random.uniform` sampler from the `numpy` package for your `weight` parameter. And then your `bias` parameter should be `weight` times a standard normal variable. Using `tune.sample_from` you can tackle this situation (or more complex and nested ones) as follows:

```
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)
```

There are many more interesting features to explore in Ray Tune, but let's switch gears here and look into some machine learning applications using Tune.

Machine Learning with Tune

As we've seen, Tune is versatile and allows you to tune hyperparameters for any objective you give it. In particular, you can use it with any machine learning framework you're interested in. In this section we're going to give you two examples to illustrate this.

First, we're going to use Tune to optimize parameters of an RLlib reinforcement learning experiment, and then we're tuning a Keras model using Optuna through Tune.

Using RLlib with Tune

RLlib and Tune have been designed to work together, so you can quite easily set up an HPO experiment for your existing RLlib code. In fact, RLlib Trainers can be passed into the first argument of `tune.run`, as `Trainable`. You can choose between the actual Trainer class, like `DQNTrainer`, or its string representation, like `"DQN"`. As Tune `metric` you can pass any metric tracked by your RLlib experiment, for instance `"episode_reward_mean"`. And the `config` argument to `tune.run` is just your RLlib Trainer configuration, but you can use the full power of Tune's search space API to sample hyperparameters like the learning rate or training

batch size.[4] Here's a full example of what we just described, running a tuned RLlib experiment on the `CartPole-v0` gym environment:

```
from ray import tune

analysis = tune.run(
    "DQN",
    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]),
    },
)
```

Tuning Keras Models

To wrap up this chapter, let's look at a slightly more involved example. As we mentioned before, this is not primarily a machine learning book, but rather an introduction to Ray and its libraries. That means that we can neither introduce you to the basics of ML, nor can we spend much time on introducing ML frameworks in detail. So, in this section we assume familiarity with Keras and its API, and some basic knowledge about supervised learning. If you do not have these prerequisites, you should still be able to follow along and focus on the Ray Tune specific parts. You can view the following example as a more realistic scenario of applying Tune to machine learning workloads.

From a bird's eye view, we're going to

- load a common data set,
- prepare it for an ML task,
- define a Tune objective by creating a deep learning model with Keras that reports an accuracy metric to Tune,

- and use Tune's HyperOpt integration to define a search algorithm that tunes a set of hyperparameters of our Keras model.

The Tune workflow remains the same as before - we define an objective, a search space, and then use `tune.run` with the configuration we want. On a high level, the process of using Tune with any ML framework works as follows.

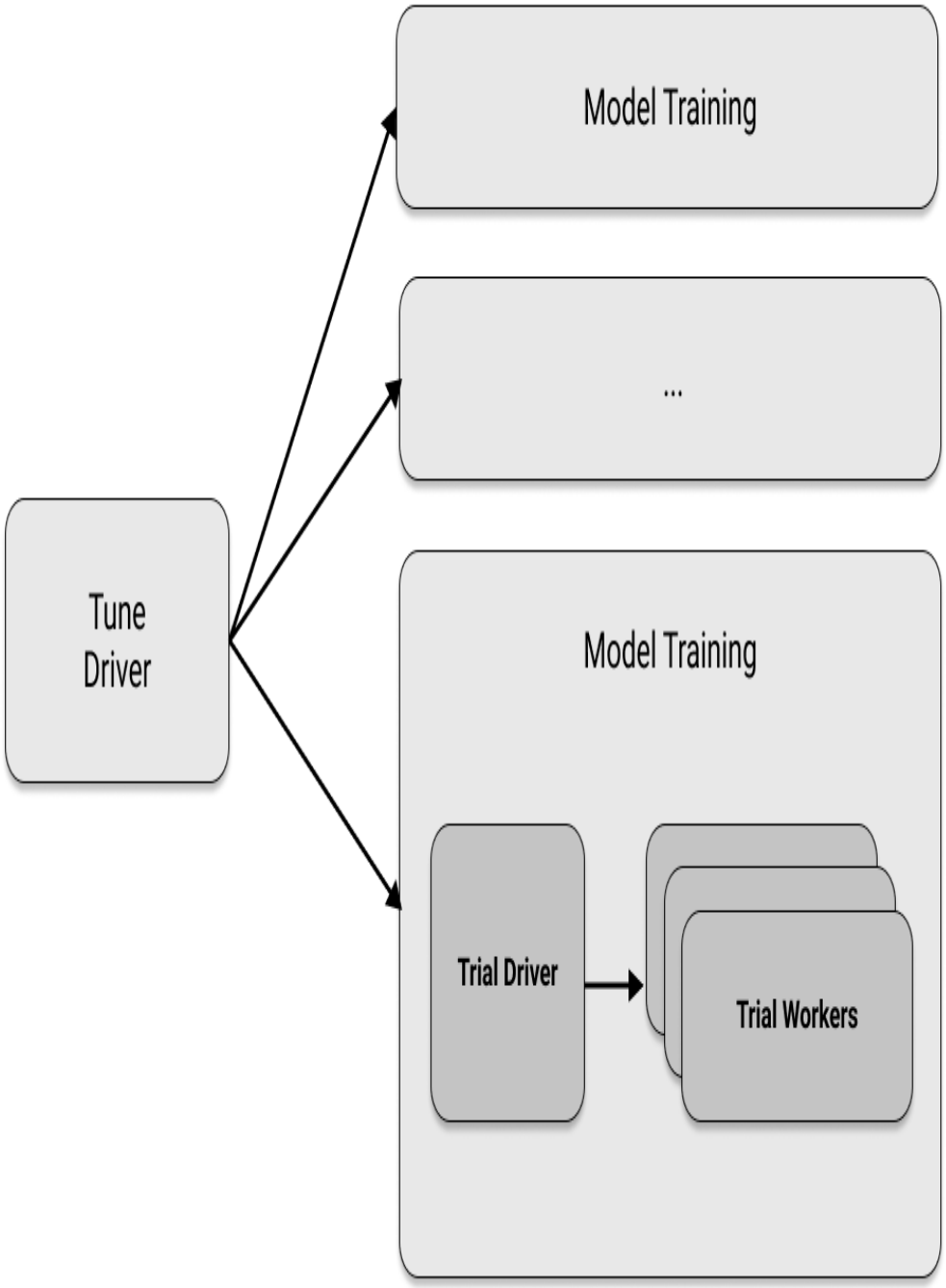


Figure 5-2. Tune sets up distributed HPO for your ML models by executing trials on Ray workers in your cluster and reporting metrics back to the driver.

To define a data set to train on, let's write a simple `load_data` utility function that loads the famous MNIST data that ships with Keras. MNIST consists of 28 times 28 pixel images of handwritten digits. We normalize the pixel values to be between 0 and 1, and make the labels for those ten digits *categorical variables*. Here's how you can do this purely with Keras' built-in functionality (make sure to `pip install tensorflow` before running this):

```
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()
```

Note that after defining `load_data`, we call it once so that the data gets downloaded locally. That's because when you call `mnist.load_data()`, it first looks for a locally cached copy. If that's present it tries to open the local copy, and will otherwise download it first. If we didn't load the data first, several Tune workers would try to download the data in parallel, which can lead to problems.[5]

Next, we define a Tune `objective` function, or Trainable, by loading the data we just defined, setting up a sequential Keras model with hyperparameters selected from the `config` we pass into our `objective`, and then compile and fit the model. To define our deep learning model, we first flatten the MNIST input images to vectors

and then add two fully-connected layers (called `Dense` in Keras) and a `Dropout` layer in between.

The hyperparameters we want to tune are the activation function of the first `Dense` layer, the `Dropout` rate, and the number of "hidden" output units of the first layer. We could tune any other hyperparameter of this model the same way, this selection is just an example.

We could manually report a metric of interest in the same way we did in other examples in this chapter (e.g. by returning a dictionary in our `objective` or using `tune.report(...)`). But since Tune comes with a proper Keras integration, we can use the so-called `TuneReportCallback` as a custom Keras callback that we pass into our model's `fit` method. This is what our Keras `objective` function looks like:

```
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"})])
```

Next, let's use a custom search algorithm to tune this objective. Specifically, we're using the `HyperOptSearch` algorithm, which

gives us access to HyperOpt's TPE algorithm through Tune. To use this integration, make sure to install HyperOpt on your machine (for instance with `pip install hyperopt==0.2.7`). `HyperOptSearch` allows us to define a list of promising, initial hyperparameter choices to investigate.

This is entirely optional, but sometimes you might have good guesses to start from. In our case, we go with a dropout `"rate"` of 0.2, 128 `"hidden"` units, and a rectified linear unit (ReLU) `"activation"` function initially. Other than that, we can define a search space with `tune` utility just as we did before. Finally, we can get an `analysis` object to determine the best hyperparameters found by passing everything into a `tune.run` call.

```
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("Best hyperparameters found were: ",
analysis.best_config)
```

Note that we're using the full power of HyperOpt here, without having to learn any specifics of it. HyperOpt itself is not distributed (by default). By using HyperOpt through the Tune API, we can leverage it for distributed HPO on a Ray cluster.

We chose the combination of Keras and HyperOpt as example of using Tune with an advanced ML framework and a third-party HPO library here. But we could have chosen any other machine learning library and practically any other HPO library supported by Tune. If you're interested in diving deeper into any of the many other integrations Tune has to offer, check out the Ray Tune documentation examples.

Summary

Tune is arguably one of the most versatile HPO tools you can choose today. It's very feature-rich, offering many search algorithms, advanced schedulers, complex search spaces, custom stoppers and many other features that we couldn't cover in this chapter. Also, it seamlessly integrates with most notable HPO tools, such as Optuna or HyperOpt, making it easy to either migrate from these tools, or simply leverage their features through Tune. You can view Ray Tune as a flexible, distributed HPO framework that *extends* others that might only work on single machines.

---

1 Tune uses the same resource model as Ray Core. Each `tune_objective` run will be executed on a different CPU core by default. If you want to, you can also specify a (fractional) GPU to be used for each trial. We will discuss thin in more detail later.

2 An important question to address in open source software is who is responsible for maintaining an integration. We will discuss this more in Chapter 11 that covers Ray's ecosystem as a whole. In the case of Tune, among the integrations listed here, the HyperOpt and Optuna integrations are maintained by the Ray Tune team, the rest are community-sponsored.

3 If you want to learn more about how to use callbacks in Tune, or create your own callbacks, please check out the user guide on callbacks and metrics in Tune.

4 In case you were wondering why the “config” argument in `tune.run` was not called `search_space`, the historical reason lies in this interoperability with RLlib `config` objects.

5 This can even lead to a race condition in which one worker has just started downloading the data and another one checks for and sees a local copy. But since the download is not complete yet, the second worker will try to open a potentially corrupted file. This goes to show you that, while Ray takes care of a lot of things in the background for you, you still need to be mindful about how you write your code.

Chapter 6. Data Processing with Ray

Edward Oakes

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

In the previous chapter, you learned how to tune hyperparameters for your machine learning experiments. Of course, the key component to applying machine learning in practice is data. In this chapter, we'll explore the core set of data processing capabilities on Ray: Ray Data.

While not meant to be a replacement for more general data processing systems such as Apache Spark or Apache Hadoop, Ray Data offers basic data processing capabilities and a standard way to load, transform, and pass data to different parts of a Ray application. This enables an ecosystem of libraries on Ray to all speak the same language so users can mix and match functionality in a framework-agnostic way to meet their needs.

The central component of the Ray Data ecosystem is the aptly-named "Datasets," which offers the core abstractions for loading, transforming, and passing references to data in a Ray cluster. Datasets are the "glue" that enable different libraries to interoperate on top of Ray. You'll see this in action later in the chapter, as we show how you can do dataframe processing using the full expressiveness of the Dask API using Dask on Ray and transform

the result into a Dataset. You'll also see how a Dataset can be used to efficiently do distributed deep learning training at scale using Ray Train and Ray Tune.

The main benefits of Datasets are:

- **Flexibility**: Datasets support a wide range of data formats, work seamlessly with library integrations like Dask on Ray, and can be passed between Ray tasks and actors without copying data.
- **Performance for ML workloads**: Datasets offers important features like accelerator support, pipelining, and global random shuffles that accelerate ML training and inference workloads.

This chapter is intended to get you familiar with the core concepts for doing data processing on Ray and to help you understand how to accomplish common patterns and why you would choose to use different pieces to accomplish a task. The chapter assumes a basic familiarity with data processing concepts such as map, filter, groupby, and partitions, but it's not to be a tutorial on data science in general nor a deep dive into the internals of how these operations are implemented. Readers with a limited data science background should not have a problem following along.

We'll start by giving a basic introduction to the core building block: Ray Datasets. This will cover the architecture, basics of the API, and an example of how Datasets can enable you to build complex data-intensive applications. Then, we'll briefly cover external library integrations on Ray, focusing on Dask on Ray. Finally, we'll bring it all together by building a scalable end-to-end machine learning pipeline in a single Python script.

The notebook for this chapter is available online along with the data used in the end-to-end example.

Ray Datasets

The main goal of Datasets is to support a scalable, flexible abstraction for data processing on Ray. Datasets are intended to be the standard way to read, write, and transfer data across the full ecosystem of Ray libraries. One of the most powerful uses of Datasets is acting as the data ingest and preprocessing layer for machine learning workloads, allowing you to efficiently scale up training using Ray Train and Ray Tune. This will be explored in more detail in the last section of the chapter.

If you've worked with other distributed data processing APIs such as Apache Spark's RDDs in the past, you will find the Datasets API very familiar. The core of the API leans on functional programming and offers standard functionality such as reading and writing different data sources, performing basic transformations like map, filter, and sort, as well as some simple aggregations such as groupby.

Under the hood, Datasets implements Distributed Apache Arrow. Apache Arrow is a unified columnar data format for data processing libraries and applications, so integrating with it means that Datasets get interoperability with many of the most popular processing libraries such as NumPy and Pandas out of the box.

A Dataset consists of a list of Ray object references, each of which points at a "block" of data. These blocks are either Arrow tables or Python lists (for data that isn't supported by the Arrow format) in Ray's shared memory object store, and compute over the data such as for map or filter operations happens in Ray tasks (and sometimes actors).

Because Datasets relies on the core Ray primitives of tasks and objects in the shared memory object store, it inherits key benefits of Ray: scalability to hundreds of nodes, efficient memory usage due to sharing memory across processes on the same node, as well as object spilling and recovery to gracefully handle failures. Additionally, because Datasets are just lists of object references, they can also be passed between tasks and actors efficiently without needing to make

a copy of the data, which is crucial for making data-intensive applications and libraries scalable.

Datasets Basics

This section will give a basic overview of Ray Datasets, covering how to get started reading, writing, and transforming datasets. This is not meant to be a comprehensive reference, but rather to introduce you to the basic concepts, so we can build up to some interesting examples of what makes Ray Datasets powerful in later sections. For up-to-date information on what's supported and exact syntax, see the Datasets documentation.

To follow along with the examples in this section, please make sure you have Ray Datasets installed locally:

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

Creating a Dataset

First, let's create a simple Dataset and perform some basic operations on it:

```
import ray

# Create a dataset containing integers in the range [0,
10000).
ds = ray.data.range(10000)

# Basic operations: show the size of the dataset, get a few
samples, print the schema.
print(ds.count())  # -> 10000
print(ds.take(5))  # -> [0, 1, 2, 3, 4]
print(ds.schema())  # -> <class 'int'>
```

Here we created a Dataset containing the numbers from 0 to 10000, then printed some basic information about it: the total number of

records, a few samples, and the schema (we will discuss this in more detail later).

Reading from and writing to storage

Of course, for real workloads you'll often want to read from and write to persistent storage to load your data and write the results. Writing and reading Datasets is simple; for example, to write a Dataset to a CSV file and then load it back into memory, we just need to use the builtin write_csv and read_csv utilities:

```
# Save the dataset to a local file and load it back.
ray.data.range(10000).write_csv("local_dir")
ds = ray.data.read_csv("local_dir")
print(ds.count())
```

Datasets supports a number of common serialization formats such as CSV, JSON, and Parquet and can read from or write to local disk as well as remote storage like HDFS or AWS S3.

In the example above, we provided just a local file path (`"local_dir"`) so the Dataset was written to a directory on the local machine. If we wanted to write to and read from S3 instead, we would instead provide a path like `"s3://my_bucket/"` and Datasets would automatically handle efficiently reading and writing remote storage,[1] parallelizing the requests across many tasks to improve throughput.

Note that Datasets also supports custom data sources that you can use to write to any external data storage system that isn't supported out-of-the-box.

Built-in transformations

Now that we understand the basic APIs around how to create and inspect Datasets, let's take a look at some of the builtin operations we can do on them. The code sample below shows three basic operations that Datasets support:

```
ds1 = ray.data.range(10000)
ds2 = ray.data.range(10000)
ds3 = ds1.union(ds2)  ❶
print(ds3.count())  # -> 20000

# Filter the combined dataset to only the even elements.
ds3 = ds3.filter(lambda x: x % 2 == 0)  ❷
print(ds3.count())  # -> 10000
print(ds3.take(5))  # -> [0, 2, 4, 6, 8]

# Sort the filtered dataset.
ds3 = ds3.sort()  ❸
print(ds3.take(5))  # -> [0, 0, 2, 2, 4]
```

❶ First, we `union` two Datasets together. The result is a new Dataset that contains all the records of both.
❷ Then, we `filter` the elements of a Dataset to only include even integers by providing a custom filter function.
❸ Finally, we `sort` the Dataset.

In addition to the operations above, Datasets also support common aggregations you might expect such as `groupby`, `sum`, `min`, etc. You can also pass a user-defined function for custom aggregations.

Blocks and repartitioning

One of the important things to keep in mind when using Datasets is the concept of *blocks* discussed earlier in the section. Blocks are the underlying chunks of data that make up a Dataset; operations are applied to the underlying data one block at a time. If the number of blocks in a Dataset is too high, each block will be small and there will be a lot of overhead for each operation. If the number of blocks is too small, operations won't be able to be parallelized as efficiently.

If we take a peek under the hood from the example above, we can see that the initial datasets we created each had 200 blocks by default. When we combined them, the resulting Dataset had 400 blocks. Given that the number of blocks is important for efficiency, we may want to reshuffle the data to match our original 200 blocks

and retain the same parallelism. This process of changing the number of blocks is called *repartitioning* and Ray Datasets offer a simple `.repartition(num_blocks)` API to achieve it. Let's use the API to repartition our resulting dataset back into 200 blocks:

```
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
```

Blocks also control the number of files that are created when we write a Dataset to storage (so if you want all of the data to be coalesced into a single output file, you should call `.repartition(1)` before writing it).

Schemas and Data Formats

Up until this point, we've been operating on simple Datasets made up only of integers. However, for more complex data processing we often want to have a schema, allowing us to more easily comprehend the data and enforce types on each column.

Given that datasets are meant to be the point of interoperation for applications and libraries on Ray, they are designed to be agnostic to a specific datatype and offer flexibility to read, write, and convert between many popular data formats. Datasets by supporting Arrow's columnar format, which enables converting between different types of structured data such as Python dictionaries, DataFrames, and serialized Parquet files.

The simplest way to create a Dataset with a schema is to create it from a list of Python dictionaries:

```
ds = ray.data.from_items([{"id": "abc", "value": 1}, {"id":
"def", "value": 2}])
print(ds.schema())  # -> id: string, value: int64
```

In this case, the schema was inferred from the keys in the dictionaries we passed in. We can also convert to/from data types from popular libraries such as Pandas:

```
pandas_df = ds.to_pandas()  # pandas_df will inherit the
schema from our Dataset.
```

Here we went from a Dataset to a Pandas DataFrame, but this also works in reverse: if you create a Dataset from a dataframe, it will automatically inherit the schema from the DataFrame.

Computing Over Datasets

In the section above, we introduced some of the functionality that comes built in with Ray Datasets such as filtering, sorting, and creating unions. However, one of the most powerful parts of Datasets is that they allow you to harness the flexible compute model of Ray and perform computations efficiently over large amounts of data.

The primary way to perform a custom transformation on a Dataset is using `.map()`. This allows you to pass a custom function that will be applied to the records of a Dataset. A basic example might be to square the records of a Dataset:

```
ds = ray.data.range(10000).map(lambda x: x ** 2)
ds.take(5)  # -> [0, 1, 4, 9, 16]
```

In this example, we passed a simple lambda function and the data we operated on was integers, but we could pass any function here and operate on structured data that supports the Arrow format.

We can also choose to map batches of data instead of individual records using `.map_batches()`. There are some types of computations that are much more efficient when they're *vectorized*, meaning that they use an algorithm or implementation that is more efficient operating on a set of items instead of one at a time.

Revisiting our simple example of squaring the values in the Dataset, we can rewrite it to be performed in batches and use the `numpy.square` optimized implementation instead of the naive Python implementation:

```
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]
```

Vectorized computations are especially useful on GPUs when performing deep learning training or inference. However, generally performing computations on GPUs also has significant fixed cost due to needing to load model weights or other data into the GPU RAM. For this purpose, Datasets supports mapping data using Ray actors. Ray actors are long-lived and can hold state, as opposed to stateless Ray tasks, so we can cache expensive operations costs by running them in the actor's constructor (such as loading a model onto a GPU).

For example, to perform batch inference using Datasets, we need to pass a class instead of a function, specify that this computation should run using actors, and use `.map_batches()` so we can perform vectorized inference. Datasets will automatically auto-scale a group of actors to perform the map operation.

```
def load_model():
    # Returns a dummy model for this example.
    # In reality, this would likely load some model weights
onto a GPU.
```

```
        class DummyModel:
            def __call__(self, batch):
                return batch

        return DummyModel()


    class MLModel:
        def __init__(self):
            # load_model() will only run once per actor that's
    started.
            self._model = load_model()

        def __call__(self, batch):
            return self._model(batch)


    ds.map_batches(MLModel, compute="actors")
```

To run the inference on a GPU, we would pass `num_gpus=1` to the `map_batches` call to specify that the actors running the map function each require a GPU.

Dataset Pipelines

By default, Dataset operations are blocking, meaning they run synchronously from start to finish and there is only a single operation happening at a time. This pattern can be very inefficient for some workloads, however. For example, consider the following set of Dataset transformations on Parquet data that might be used to do batch inference for a machine learning model.[2]

```
ds = (ray.data.read_parquet("s3://my_bucket/input_data")
❶
      .map(cpu_intensive_preprocessing)  ❷
      .map_batches(gpu_intensive_inference,
compute="actors", num_gpus=1)  ❸
      .repartition(10))  ❹

ds.write_parquet("s3://my_bucket/output_predictions")
```

There are five stages to this process, and each of them stresses different parts of the system:

1. Reading from remote storage requires ingress bandwidth to the cluster and may be limited by the throughput of the storage system. In this stage, a group of Ray tasks is spawned that will read from remote storage in parallel and the resulting blocks of data are stored in the Ray object store.
2. Preprocessing the inputs requires CPU resources. The objects from the first phase are passed into a group of tasks that will execute the `cpu_intensive_preprocessing` function on each block.
3. Vectorized inference on the model requires GPU resources. The same process as in the second stage is repeated for `gpu_intensive_inference`, except this time the function is run on actors that are each allocated a GPU and multiple blocks are passed into each function call (in batches). Actors are used for this step to avoid repeatedly reloading the model used for inference onto the GPU.
4. Repartitioning requires network bandwidth within the cluster. After completing stage three, more tasks are spawned to repartition the data into 10 blocks and write each of those 10 blocks to remote storage.
5. Writing to remote storage requires egress bandwidth from the cluster and may be limited by the throughput of storage once again.

Figure Figure 6-1 depicts a basic implementation where each stage runs in sequence. This naive implementations to idle resources because each stage is blocking and run in sequence. For example, because GPU resources are only used in the final stage, they will be idle waiting for all of the data to be loaded and preprocessed.

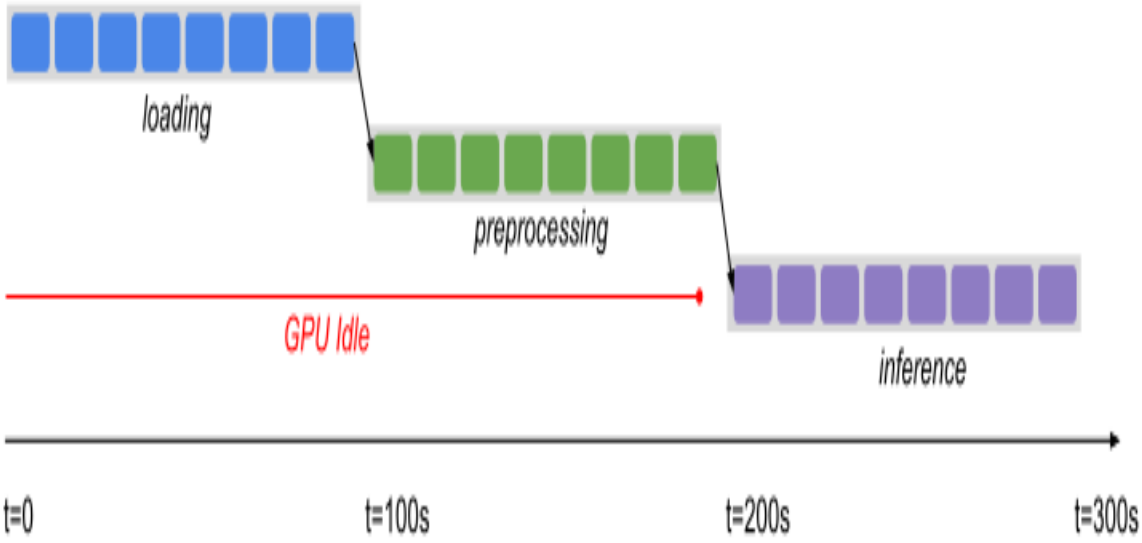


Figure 6-1. A naive Dataset computation, leading to idle resources between stages.

In this scenario, it would be more efficient to instead *pipeline* the stages and allow them to overlap as shown in figure Figure 6-2. This means that as soon as some data has been read from storage, it is fed into the preprocessing stage, then to the inference stage, and so on.

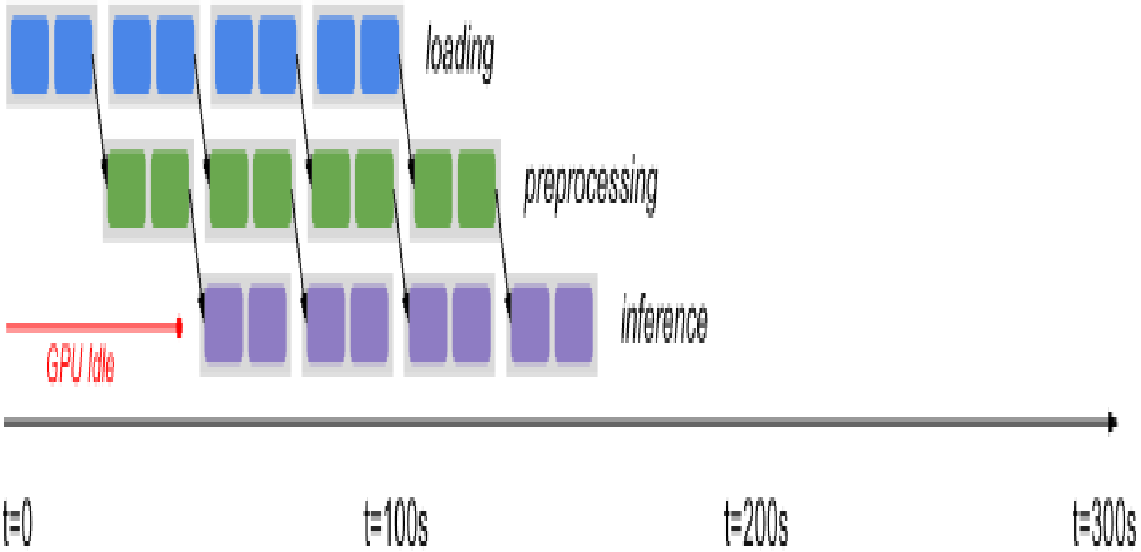


Figure 6-2. An optimized DatasetPipeline that enables overlapping compute between stages and reduces idle resources.

This pipelining will improve the overall resource usage of the end-to-end workload, improving throughput and therefore decreasing the cost it takes to run the computation (fewer idle resources is better!).

Datasets can be converted to DatasetPipelines using `ds.window()`, enabling the pipelining behavior that we want in this scenario. A window specifies the number of blocks that will be passed through a stage in the pipeline before being passed to the next stage in the pipeline. This behavior can be tuned using the `blocks_per_window` parameter, which defaults to 10.

Let's rewrite the inefficient pseudocode above to use a DatasetPipeline instead:

```
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")
```

The only modification made was the addition of a `.window()` call after `read_parquet` and before the preprocessing stage. Now the Dataset has been converted to a DatasetPipeline and its stages will proceed in parallel in 5-block windows, decreasing idle resources and improving efficiency.

DatasetPipelines can also be created using `ds.repeat()` to repeat stages in a pipeline a finite or infinite number of times. This will be explored further in the next section, where we'll use it for a training workload. Of course, pipelining can be equally beneficial for performance for training in addition to inference.

Example: Training Copies of a Classifier in Parallel

One of the key benefits of Datasets is that they can be passed between tasks and actors. In this section, we'll explore how this functionality can be used to write efficient implementations of complex distributed workloads like distributed hyperparameter tuning and machine learning training. We'll implement an example of *distributed* training of ML models in this section, a topic that we'll cover in much more detail in Chapter 7 when we introduce you to *Ray Train*.

As discussed in Chapter 5, a common pattern in machine learning training is to explore a range of hyperparameters to find the ones that result in the best model. We may want to run across a wide range of hyperparameters, and doing this naively could be very expensive. Ray Data allows us to easily share the same in-memory data across a range of parallel training runs in a single Python script: we can load and preprocess the data once, then pass a reference to it to many downstream actors who can read the data from shared memory.

Additionally, sometimes when working with very large data sets it can be infeasible to load the full training data into memory in a single process or on a single machine. In this case, it's common to *shard* the data, which means to give each worker its own subset of the data that can fit into memory. This local subset of the data is referred to as a data shard. After each worker trains on its shard of data in parallel, the results are combined either synchronously or asynchronously using a parameter server. There are some important considerations that can make this difficult to get right:

1. Many distributed training algorithms take a *synchronous* approach, requiring the workers to synchronize their weights after each training epoch. This means there needs to be some coordination between the workers to maintain consistency between which batch of data they are operating on.

2. It's important that each worker gets a random sample of the data during each epoch. A global random shuffle has been shown to perform better than local shuffle or no shuffle.

Figure Figure 6-3 illustrates how the `ray.data` package is used to create shards of data forming a Ray Dataset from a given input dataset.

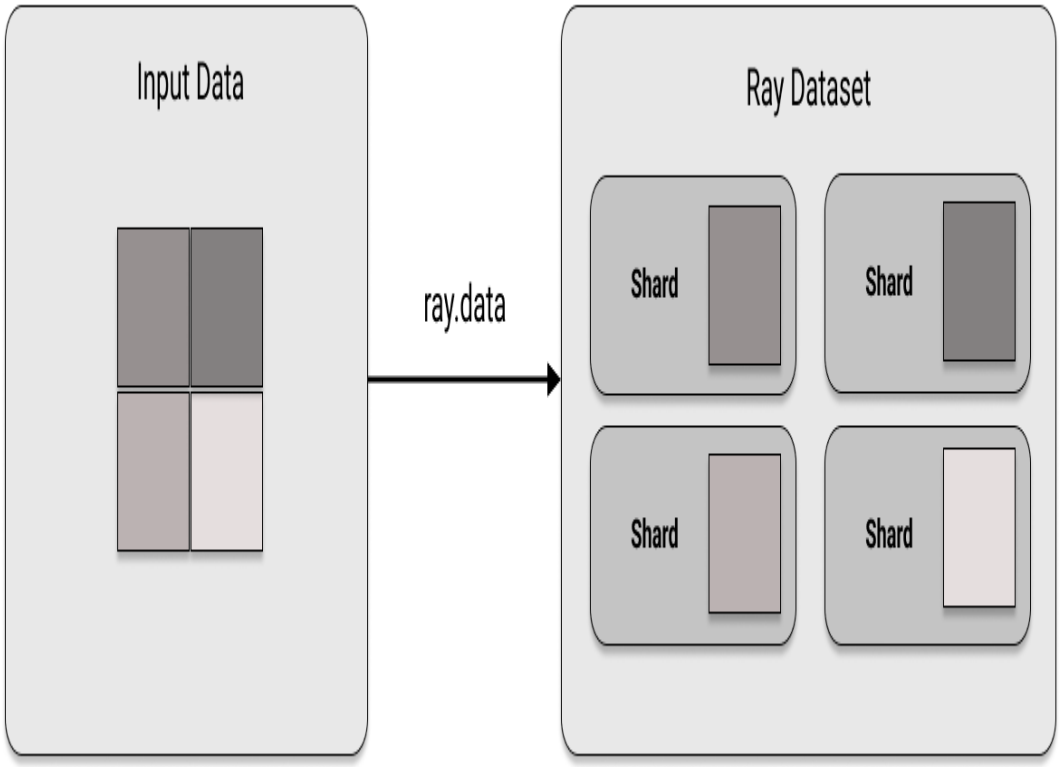


Figure 6-3. Creating Ray Datasets from input data with the `ray.data` package.

Let's walk through an example of how we can implement this type of pattern using Ray Datasets. In the example, we will train multiple copies of a machine learning model using different hyperparameters across different workers in parallel.

We'll be training a scikit-learn `SGDClassifier` algorithm on a generated binary classification dataset and the hyperparameter we'll

tune is the regularization term of this classifier.[3] The actual details of the ML task and model aren't too important to this example, you could replace the model and data with any number of examples. The main thing to focus on here is how we orchestrate the data loading and computation using Datasets.

To follow along with the examples in this section, please make sure you have Ray Datasets and `scikit-learn` installed locally:[4]

```
pip install "ray[data]==2.2.0" "scikit-learn==1.0.2"
```

First, let's define our `TrainingWorker` that will train a copy of the classifier on the data:

```
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)
```

There are a few important things to note about the `TrainingWorker`: - It's a simple wrapper around the `SGDClassifier` and instantiates it with a given alpha value. - The main training function happens in the `train` method. For each epoch, it trains the classifier on the data available. - We also have a

`test` method that can be used to run the trained model against a testing set.

Now, let's instantiate a number of `TrainingWorker` instances with different hyperparameters (`alpha` values):

```
ALPHA_VALS = [0.00008, 0.00009, 0.0001, 0.00011, 0.00012]

print(f"Starting {len(ALPHA_VALS)} training workers.")
workers = [TrainingWorker.remote(alpha) for alpha in
ALPHA_VALS]
```

Next, we generate training and validation data and convert the training data to a Dataset. Here, we're using `.repeat()` to create a DatasetPipeline. This defines the number of epochs that our training will run for. In each epoch, the subsequent operations will be applied to the Dataset and the workers will be able to iterate over the resulting data. We also shuffle the data randomly and shard it to be passed to the training workers, each getting an equal chunk.

```
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(10)  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 We generate training and validation data for a classification problem.

❷ We convert the training data to a Dataset by using `from_items`.

❸ Then, we define a DatasetPipeline using `.repeat`. This is similar to using `.window` as we showed earlier, but allows us to iterate over the same dataset multiple times (in this case, 10).

❹ We shuffle the data randomly each time it is repeated.
❺ We want each worker to have its own local shard of the data, so we `split` the `DatasetPipeline` into multiple smaller ones that can be passed to each worker.
❻ Finally, we wait for training to complete on all the workers.

To run the training on the workers, we invoke their `train` method and pass in one shard of the `DatasetPipeline` to each. We then block waiting for training to complete across all the workers. To summarize what happens during this phase:

- Each epoch, each worker gets a random shard of the data.
- The worker trains its local model on the shard of data assigned to it.
- Once a worker has finished training on the current shard, it blocks until the other workers have finished.
- The above repeats for the remaining epochs (in this case, 10 total).

Finally, we can test out the trained models from each worker on some test data to determine which alpha value produced the most accurate model.

```
# Get validation results from each worker.
print(ray.get([worker.test.remote(X_test, Y_test) for
worker in workers]))
```

In reality for this type of workload you should likely reach for Ray Tune or Ray Train, which we'll cover in the next chapter, but this example conveys the power of Ray Datasets for machine learning workloads. In just a few tens of lines of Python code, we implemented a complex distributed hyperparameter tuning and training workflow that could easily be scaled up to hundreds of machines and is agnostic to any framework or specific ML task.

External Library Integrations

While Ray Datasets supports a number of common data processing functionalities out of the box, as mentioned above it's not a replacement for full data processing systems. Instead, as shown in Figure 6-4, it's more focused on performing "last mile" processing such as basic data loading, cleaning, and featurization before ML training or inference.

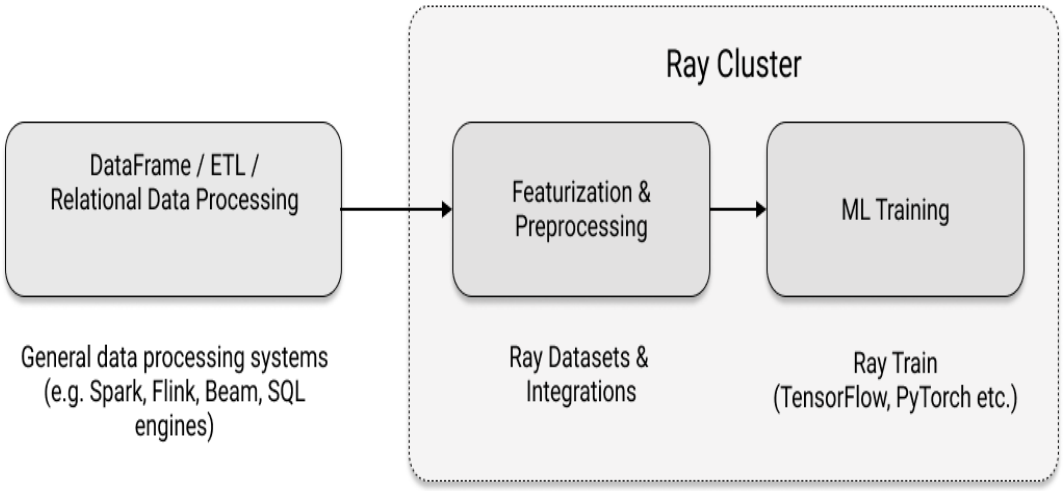


Figure 6-4. A typical workflow using Ray for machine learning: use external systems for primary data processing and ETL, use Ray Datasets for last-mile preprocessing.

However, there are a number of other more fully-featured DataFrame and relational data processing systems that integrate with Ray:

- Dask on Ray
- RayDP (Spark on Ray)
- Modin (Pandas on Ray)
- Mars on Ray

These are all standalone data processing libraries that you may be familiar with outside the context of Ray. Each of these tools has an

integration with Ray core that enables more expressive data processing than comes with the built-in Datasets while still using Ray's deployment tooling, scalable scheduling, and shared memory object store for exchanging data. As shown in Figure 6-5, this complements Ray Datasets and enables end-to-end data processing on Ray.

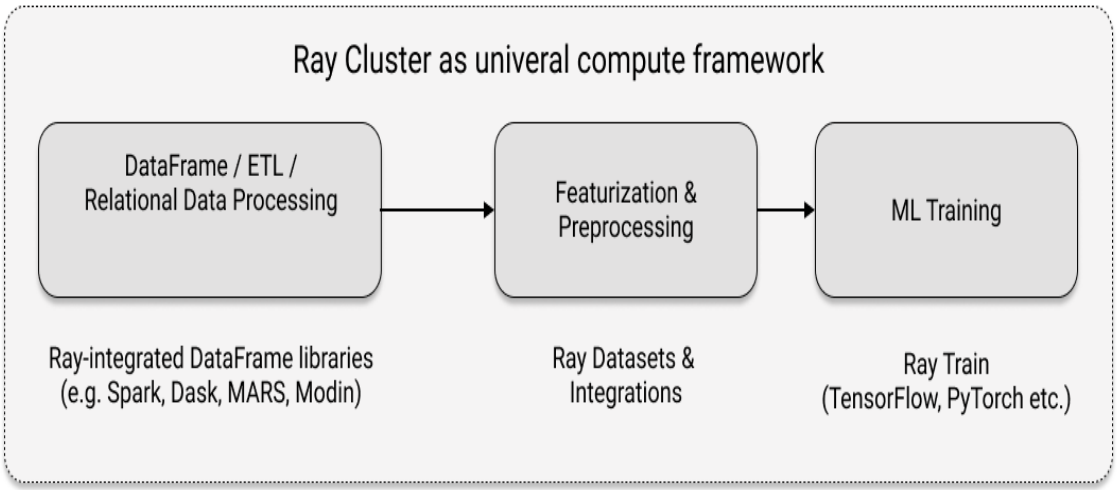


Figure 6-5. The benefit of Ray data ecosystem integrations, enabling more expressive data processing on Ray. These libraries integrate with Ray Datasets to feed into downstream libraries such as Ray Train.

For the purposes of this book, we'll explore Dask on Ray in slightly more depth to give you a feel for what these integrations look like. If you're interested in the details of a specific integration, please see the latest Ray documentation for up-to-date information.

Example: Dask on Ray

To follow along with the examples in this section, please install Ray and Dask:

```
pip install "ray[data]==2.2.0" "dask==2022.2.0"
```

Dask is a Python library for parallel computing that is specifically target at scaling analytics and scientific computing workloads to a cluster. One of the most popular features of Dask is Dask DataFrames, which offers a subset of the Pandas DataFrame API that can be scaled to a cluster of machines in cases where processing in memory on a single node is not feasible. DataFrames work by creating a *task graph* that is submitted to a scheduler for execution. The most typical way to execute Dask DataFrames operations is using the Dask Distributed scheduler, but there is also a pluggable API that allows other schedulers to execute these task graphs as well.

Ray comes packaged with a Dask scheduler backend, allowing Dask DataFrame task graphs to be executed as Ray tasks and therefore make use of the Ray scheduler and shared memory object store. This doesn't require modifying the core DataFrames code at all; instead, in order to run using Ray all you need to do is first connect to a running Ray cluster (or run Ray locally) and then enable the Ray scheduler backend:

```
import ray
from ray.util.dask import enable_dask_on_ray

ray.init()  # Start or connect to Ray.
enable_dask_on_ray()  # Enable the Ray scheduler backend
for Dask.
```

Now we can run regular Dask DataFrames code and have it scaled across the Ray cluster. For example, we might want to do some time series analysis using standard DataFrame operations like filter, groupby, and computing the standard deviation (example taken from Dask documentation).

```
import dask

df = dask.datasets.timeseries()
```

```
df = df[df.y > 0].groupby("name").x.std()
df.compute()  # Trigger the task graph to be evaluated.
```

If you're used to Pandas or other DataFrame libraries, you might wonder why we need to call `df.compute()`. This is because Dask is *lazy* by default, and will only compute results on demand, allowing it to optimize the task graph that will be executed across the cluster.

One of the most powerful aspects of Dask on Ray is that it integrates very nicely with Ray Datasets. We can convert a Ray Dataset to a Dask DataFrame and vice versa using built-in utilities:

```
import ray
ds = ray.data.range(10000)

# Convert the Dataset to a Dask DataFrame.
df = ds.to_dask()
print(df.std().compute())  # -> 2886.89568

# Convert the Dask DataFrame back to a Dataset.
ds = ray.data.from_dask(df)
print(ds.std())  # -> 2886.89568
```

This simple example might not look too impressive because we're able to compute the standard deviation using either Dask DataFrames or Ray Datasets. However, as you'll see in the next section when we build an end-to-end ML pipeline, this enables really powerful workflows. For example, we can use the full expressiveness of DataFrames to do our featurization and preprocessing, then pass the data directly into downstream operations such as distributed training or inference while keeping everything in memory. This highlights how Datasets enables a wide range of use cases on top of Ray, and how integrations like Dask on Ray make the ecosystem even more powerful.

Building an ML Pipeline

Although we were able to we build a simple distributed training application from scratch in the previous section, there were many edge cases, opportunities for performance optimization, and usability features that we would want to address to build a real-world application. As you've learned in the previous chapters about Ray RLlib, and Ray Tune, Ray has an ecosystem of libraries that enable us to build production-ready ML applications. In this section, we'll explore how to use Datasets as the "glue layer" to build an ML pipeline end-to-end.

To successfully productionize a machine learning model, one first needs to collect and catalog data using standard ETL processes. However, that's not the end of the story: in order to train a model, we also often need to do featurization of the data before feeding into our training process, and how we feed the data into training can make a big impact on cost and performance. After training a model, we'll also want to run inference across many different datasets — that's the whole point of training the model after all!

Though this might seem like just a chain of steps, in practice the data processing workflow for machine learning is an iterative process of experimentation to define the right set of features and train a high-performing model on them. Efficiently loading, transforming, and feeding the data into training and inference is also crucial for performance, which translates directly to cost for compute-intensive models. Often times, implementing such ML pipelines means stitching together multiple different systems and materializing intermediate results to remote storage between the stages. This has two major downsides:

1. First, it requires orchestrating many different systems and programs for a single workflow. This can be a lot to handle for any ML practitioner, so many people reach to workflow orchestration systems like Apache Airflow. While Airflow has some great benefits, it's also a lot of complexity to introduce (especially in development).

2. Second, running our ML workflow across multiple different systems means we need to read from and write to storage between each stage.[5] This incurs significant overhead and cost due to data transfer and serialization.

In contrast, using Ray we are able to build a complete machine learning pipeline as a single application that can be run as a single Python script as depicted in Figure 6-6. The ecosystem of built-in and third party libraries make it possible to mix-and-match the right functionality for a given use case and build scalable, production-ready pipelines. Importantly, Ray Datasets acts as the glue layer that enables efficient loading, preprocessing, and computing over the data while avoiding expensive serialization costs and keeping intermediate data in shared memory.

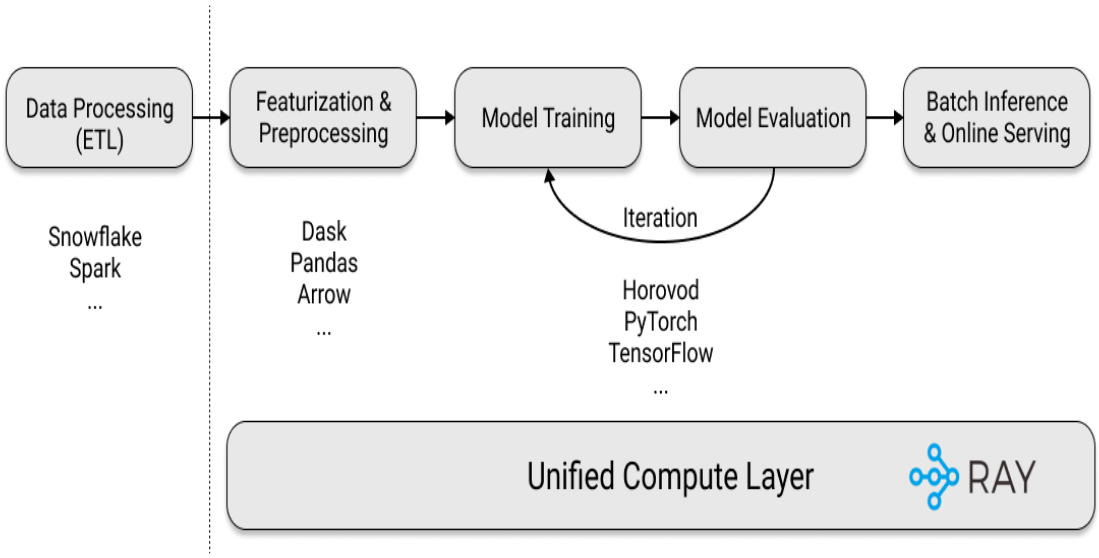


Figure 6-6. A simplified depiction of the typical machine learning workflow and where Ray fits in. ML consists of a number of steps and very often requires a lot of iteration. Without Ray, this means stitching together many independent systems for one end-to-end process. Ray acts as a unified compute layer, enabling nearly the entire workflow to be run as a single application.

In the next chapter we're going to show you a concrete example of building an end-to-end ML pipeline using Ray Datasets and other libraries in the Ray ecosystem in practice.

Summary

This chapter introduced Ray Datasets, a core building block in Ray. Ray Datasets offers built-in functionality for distributed data processing, but its true power lies in its integrations with both first- and third-party libraries. We've only covered a small portion of its functionality. For more details, API references, and examples, please see the documentation.

We've also shown you a simple example of distributed training of a scikit-learn classifier using Ray Datasets, and discussed external library integrations such as Dask on Ray. Lastly, we've indicated the value of building end-to-end ML pipelines using the Ray ecosystem, which allows you to run your entire workflow in a single Python script. For data scientists and machine learning engineers, this means faster iteration time, better ML models, and ultimately more business value.

---

1 If you're interested in following an example that reads actual data from an S3 bucket, please have a look at the Batch Inference example in the Ray documentation.

2 Parquet is a structured, column-oriented format that enables efficient compression, data storage and retrieval. Many example and real-world datasets use Parquet, so it is used in the example below. However, the code could easily be modified to use a different format just by changing the `read_parquet` and `write_parquet` calls.

3 SGD stands for *stochastic gradient descent*, which is a common optimization algorithm used in machine learning, and specifically deep learning.

4 In this and the following two chapters we're discussing more advanced ML examples that need additional dependencies. We pin the versions of these dependencies here and in the notebooks for the book to ensure the examples work as expected. Having said that, the examples very likely work with a relatively wide range of versions, as long as you make sure they're not too old.

5 Another challenge of having relying on many tools is cultural. Knowledge transfer of best practices can be costly, especially in larger companies.

Chapter 7. Distributed Training with Ray Train

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

In the previous chapter we've discussed how to train copies of a simple model on shards of data using Ray Datasets — but there's much more to distributed training than that. As we indicated in Chapter 1, Ray has a dedicated library for distributed training called Ray Train. Train comes with an extensive suite of machine learning training integrations and allows you to scale your experiments seamlessly on Ray clusters.

We will start this chapter by showing you why you might need to scale out your machine learning training, and introduce you to the different ways of doing so. After that, we'll then introduce Ray Train and walk through an extensive end-to-end example. We also cover some key concepts you need to know in order to use Ray Train, such as preprocessors, trainers and checkpoints. Finally, we'll cover some of the more advanced functionality that Ray Train provides.

As always, you can follow along the code examples using the notebook for this chapter.

The Basics of Distributed Model Training

Machine learning often requires a lot of heavy computation. Depending on the type of model that you're training, whether it be a gradient boosted tree or a neural network, you may face some common problems with training machine learning models:

- The time it takes to finish training is too long.
- The size of data is too large to fit into one machine.
- The model itself is too large to fit into a single machine.

For the first case, training can be accelerated by processing data with increased throughput. Some machine learning algorithms, such as neural networks, can parallelize parts of the computation to speed up training.[1]

In the second case, your choice of algorithm may require you to fit all the available data from a dataset into memory, but the given single node memory may not be sufficient. In this case, you will need to split the data across multiple nodes and train in a distributed manner. On the other hand, sometimes your algorithm may not require data to be distributed, but if you're using a distributed database system to begin with, you still want a training framework that can leverage your distributed data.

In the third case, when your model doesn't fit into a single machine, you may need to split up the model into multiple parts, spread across multiple machines. The approach of splitting models across multiple machines is called model parallelism. To run into this issue, you first need a model that is large enough to not fit into a single machine anymore. Usually, large companies like Google or Facebook tend to have the need for model-parallelism, and also rely on in-house solutions to handle the distributed training.

In contrast to the third case, the first two problems often arise much earlier in the machine learning development process. The solutions we just sketched for these problems fall under the umbrella of data-

parallel training. Instead of splitting up the model across multiple machines, you instead rely on distributed data to speed up training.

Specifically for the first problem, if you can speed up your training process, hopefully with minimal or no loss in accuracy, and you can do so cost-efficiently, why not go for it? And if you have distributed data, whether by necessity of your algorithm or the way you store your data, you need a training solution to deal with it. As you will see, Ray Train is built for efficient, data-parallel training.

To wrap up this section, let's have a look at figure Figure 7-1, which gives you a brief summary of the two basic types of distributed training we just discussed:

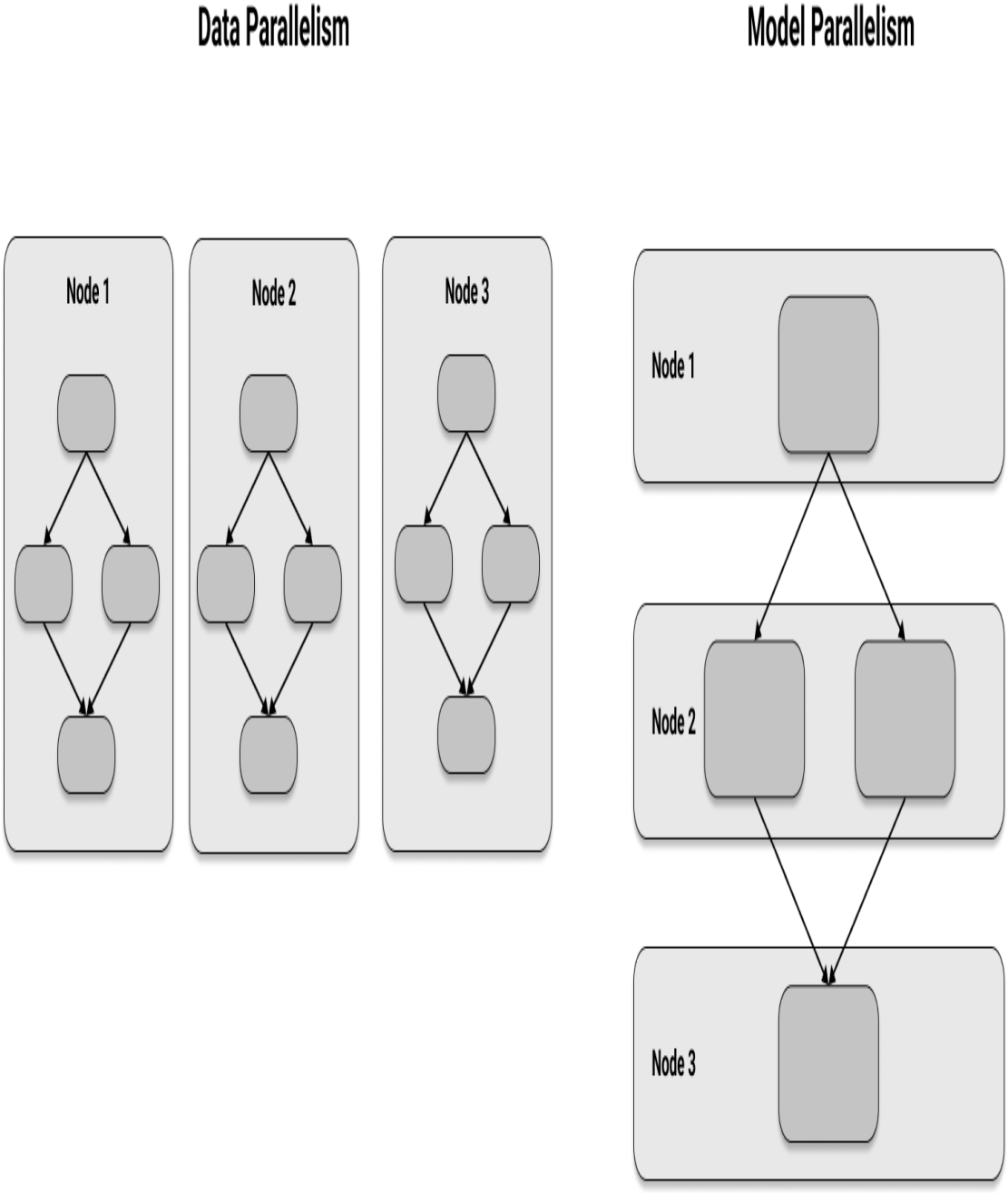


Figure 7-1. Data versus Model Parallelism in Distributed Training

Introduction to Ray Train by Example

Ray Train is a library for distributed data-parallel training on Ray. It offers key tools for different parts of the training workflow, from feature processing, to scalable training, to integrations with ML tracking tools, to export mechanisms for models.

In a basic ML training pipeline you will use the following key components of Ray Train.

- *Trainers*: Ray Train has several Trainer classes that make it possible to do distributed training. Trainers are wrapper classes around third-party training frameworks like XGBoost, Pytorch, TensorFlow providing integration with core Ray actors (for distribution), Ray Tune, and Ray Datasets.
- *Predictors*: Once you have a trained model, Batch Predictors are used to make predictions on batches of data. They are used to evaluate the performance of a model on a validation set.

Additionally, Ray Train provides several common *Preprocessor* objects and utilities to process dataset objects into consumable features for Trainers. Finally, Ray Train provides a *Checkpoint* class that allows you to save and restore the state of a training run. In our first walk-through we will not use any preprocessors, but we will cover them in more detail later on.

Ray Train is built with first-class support for training on large datasets. Along the same philosophy that you should not have to think about *how* to parallelize your code, you can simply "connect" your large dataset to Ray Train without thinking about how to ingest and feed your data into different parallel workers.

Let's see how to put these components in practice by walking through your first Ray Train example. To load the training data we're going to leverage our knowledge from Chapter 6 and make heavy use of Ray Datasets.

Predicting Big Tips in NYC Taxi Rides

This section walks through a practical, end-to-end example of building a deep learning pipeline using Ray. We will build a binary classification model to predict if a taxi ride will result in a big tip (>20% of the fare) using the public New York City Taxi and Limousine Commission (TLC) Trip Record Data. Our workflow will closely resemble that of a typical ML practitioner:

- First, we will load the data, do some basic preprocessing, and compute features we'll use in our model.
- Then, we will define a neural network and train it using distributed data-parallel training.
- Finally, we will apply the trained neural network to a fresh batch of data.

The example will use Dask on Ray and train a PyTorch neural network, but note that nothing here is specific to either of those libraries, Ray Datasets and Ray Train can be used with a wide range of popular machine learning tools. To follow along with the example code in this section, please install Ray, PyTorch, and 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"
```

In the examples below, we'll be loading the data from local disk to make it easy to run the examples on your machine. The data is available in the same GitHub repository as the notebook for this chapter. The file paths in the examples below assume you've cloned the repository and are running from within its top-level directory.

Loading, Preprocessing, and Featurization

The first step in training our model is to load and preprocess it. To do this, we'll be using Dask on Ray, for which you've already seen a first example in Chapter 6. Dask on Ray gives us a convenient DataFrames API and the ability to scale up the preprocessing across a cluster and efficiently pass it into our training and inference operations. Below is our code for preprocessing data and building features for our model, defined in a single `load_dataset` function:

```
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
```

❶ First, you drop unused columns of the initial Dask DataFrame loaded from Parquet files.
❷ Then you do basic cleaning, and drop null values and outliers.
❸ You then add three new features: trip duration, hour the trip started, and day of the week.
❹ Next, you calculate the label column: if the tip was more or less than 20% of the fare.
❺ You then drop all unused columns.
❻ Finally, you return a repartitioned Ray Dataset created *from Dask*.

This involves basic data loading and cleaning as well as transforming some columns into a format that can be used as features in our machine learning model. For instance, we transform the pickup and drop-off date-times, which are provided as a string, into three numerical features: `trip_duration`, `hour`, and `day_of_week`. This is made easy by Dask's built-in support for Python datetime utilites. If this data is going to be used for training, we also need to compute the label column.

Finally, once we've computed our preprocessed Dask DataFrame, we transform it into a Ray Dataset, so we can pass it into our training and inference processes later.

Defining a Deep Learning Model

Now that we've cleaned and prepared the data, we need to define a model architecture that we'll use for the model. In practice, this would likely be an iterative process and involve researching the state of the art for similar problems. For the sake of our example, we'll keep things simple and use a basic PyTorch neural network that we call `FarePredictor`. The neural network has three linear transformations starting with the dimension of our feature vector and then outputs a value between 0 and 1 using a Sigmoid activation function. We also use *batch normalization* layers in the network to improve training. This output value will be rounded to produce the binary prediction of if the ride will result in a big tip or not.

```
import torch
import torch.nn as nn
import torch.nn.functional as F


class FarePredictor(nn.Module):
    def __init__(self):
        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
```

Distributed Training with Ray Train

Now that we've defined the neural network architecture, we need a way to efficiently train this on our data. This data set is very large, so our best bet is to perform *data-parallel training*.

This means that we train the model on multiple machines in parallel, each of which has a copy of the model and a subset of the data. We will use Ray Train to define a scalable training process with Ray Train that will use PyTorch DataParallel under the hood. We won't go into conceptual details of the training process here, but rather discuss them in the sections following this end-to-end example.

> **NOTE**
>
> The code example you're about to see uses imports from the `ray.air` module. We've mentioned the Ray AI Runtime (AIR) in the first chapter of this book, but will only introduce it formally in Chapter 10. For now, treat this module as a useful utility for defining and running your distributed training processes.
>
> In particular, we're making use of a so-called AIR `session` that can for instance be used to `report` metrics that get collected during the training process. This follows a similar usage pattern as the `tune.report` API that we've discussed in Chapter 5.

The first thing we need to do is define the core logic that will happen to train on a batch of data on each worker in each epoch. This will take in a local shard of the full dataset, run it through the local copy of the model, and perform backpropagation to update the model weights. After each epoch, the worker will use Ray Train utilities to report the result and save the current model weights for use later.

```
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 We pass a `config` dictionary into our training loop to specify some parameters at runtime.

❷

We then retrieve the data shard for the current worker, using Ray Train's `get_data_shard` utility.

❸ Next, we prepare the PyTorch model for distributed training by simply applying `prepare_model` to it.

❹ We then define a standard PyTorch training loop, iterating over batches of data and performing backpropagation.

❺ The only non-standard part is the usage of `iter_torch_batches` to iterate over the data shard.

❻ After each epoch we `report` the `loss` computed and a *model checkpoint* using a Ray `session`.

In case you're not familiar with PyTorch, note that the code between the definition of the `loss_function` and aggregating the `batch_loss` to our `loss` is a standard training loop for a PyTorch model (except for iterating over batches of the dataset shard, which is specific to Ray).

Now that the training process has been defined, we need to load the training and validation data to feed into our training workers. Here, we call the `load_dataset` function we defined earlier that will do preprocessing and featurization.[2]

This dataset is passed into a `TorchTrainer` along with some configuration parameters such as the batch size, number of epochs, and number of workers to use. Each worker will have access to a shard of the data locally and can iterate over it. After training has completed, we can fetch the final trained checkpoint from the returned result object.

```
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 Each `TorchTrainer` requires you to specify a `train_loop_per_worker`.
❷ Optionally, if your train loop takes in a `config` dictionary, you can specify it as `train_loop_config`.
❸ Every Ray Train `Trainer` needs a so-called `ScalingConfig` to know how to scale out training on your Ray cluster.
❹ Another required argument of each `Trainer` is a `datasets` dictionary. We define a `"train"` Dataset here, is that is what we use in our training loop.
❺ You can simply `.fit()` a `TorchTrainer` to start training.

The last line exports our trained model as a *Checkpoint* for later use in downstream applications like serving and inference. Ray Train generates these checkpoints to serialize intermediate state for training. Checkpoints can include both models and other training artifacts, such as preprocessors.

Distributed Batch Inference

Once we've trained a model and gotten the best accuracy that we can, the next step is to actually apply it in practice. Sometimes this means powering a low-latency service, which we'll explore in Chapter 8, but often the task is to apply the model across batches of data as they come in.

Let's use the trained model weights from our `trained_model` apply them across a new batch of data (in this case, it'll just be another chunk of the same public data set). To do this, first we need to load, preprocess, and featurize the data in the same way we did for

training. Then we will load our model and `map` it across the whole data set. Ray Datasets allows us to do this efficiently with Ray Actors, even using GPUs just by changing one parameter. We simply load the trained model checkpoint and call `.predict_pipelined()` on it. This will use Ray datasets to perform distributed batch inference across the data.

```
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)
```

This example showed how Ray Train and Datasets can be used to implement an end-to-end machine learning workflow as a single application. We were able to featurize the dataset, train and validate a machine learning model, and then apply that model across a different dataset in a single Python script. Ray Datasets acted as the glue layer, connecting the different stages and avoiding expensive serialization costs between them. We also used checkpoints to store the model and ran a Ray Train batch prediction job to apply the model to a new dataset.

Now, that you've seen your first example of Ray Train, let's take a closer look at its primary abstraction, the `Trainer`.

More on Trainers in Ray Train

As you've seen in the above example of using a `TorchTrainer`, Trainers are framework-specific classes that run model training in a distributed fashion. All Ray `Trainer` classes share a common

interface. At this point, it's enough to know about two aspects of this interface, namely:

- the `.fit()` method, which fits a given `Trainer` with the given datasets, configuration and desired scaling properties, and
- the `.checkpoint` property, which returns the Ray `Checkpoint` object for this trainer.

Ray Train's trainers integrate with common machine learning frameworks such as Pytorch, HuggingFace, TensorFlow, Horovod, Scikit-learn, and more. There's even a `Trainer` specifically for RLlib models, which we can't cover here. Let's discuss another PyTorch example to point out specific aspects of the `Trainer` API, and with an emphasis on how to migrate an existing PyTorch model to Ray Train.

> **NOTE**
>
> Ray Train also offers support for different gradient boosted decision tree frameworks.
>
> XGBoost is an optimized distributed gradient boosting library designed to be highly efficient, flexible and portable. It implements machine learning algorithms under the Gradient Boosting framework. XGBoost provides a parallel tree boosting that solve many data science problems in a fast and accurate way.
>
> LightGBM is a gradient boosting framework based on tree-based learning algorithms. Compared to XGBoost, it is a relatively new framework, but one that is quickly becoming popular in both academic and production use cases.
>
> To use these frameworks, you can use the `XGBoostTrainer` and `LightGBMTrainer` classes, respectively.

In this example, we want to focus on the details of Ray Train itself, so we'll use a much simpler training dataset and a small neural

network that takes random noise as input. We define a three-layer `NeuralNetwork`, an explicit `training_loop`, similar to the one we saw in the previous section, that you can use straight-up to train the model. For clarity, we extract the training code run for each epoch in a helper function called `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)  ❶


input_data = train_data()
label_data = torch.randn(num_samples, output_size)
train_dataset = from_torch(input_data)  ❷


def train_one_epoch(model, loss_fn, optimizer):  ❸
    output = model(input_data)
    loss = loss_fn(output, label_data)
```

```
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()


    def training_loop():  ❹
        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 In this example we use a randomly generated dataset.
❷ We use `from_torch` to create a Ray Dataset from this data.
❸ We extract the PyTorch code to train one epoch into a helper function.
❹ This training loop can be run as-is to train your PyTorch model on a single machine.

Typically, if you wanted to distribute your training without Ray Train, you would need to do the following two things:

1. Establish a backend that coordinates interprocess communication.
2. Instantiate multiple parallel processes on each node that you want to distribute your training across.

In contrast, let's see how easy it is to use Ray Train to distribute your training process.

Migrating to Ray Train with Minimal Code Changes

With Ray Train, you can make a one-line change to your code to take care of both above tasks (interprocess communication and process instantiation) for you under the hood.

The code change is made by calling `prepare_model` on the Pytorch model that you plan to train. This change is literally the only difference between the `training_loop` defined before and the following `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 Prepare the model for distributed training by calling `prepare_model`.

Then, we can instantiate a TorchTrainer model, which has three required arguments:

- `train_loop_per_worker`: A function that trains your model for each worker. This function has access to the `datasets` provided and can be fed an optional `config` dictionary that can be passed to your trainer as `train_loop_config`. This function will typically report metrics, e.g. via a `session`.
- `datasets`: This dictionary can contain several keys with Ray Datasets as values. It's kept flexible, so that you can have training data, validation data, or any other type of data needed for your training loop.
- `scaling_config`: This is a `ScalingConfig` object that specifies how your training should scale. For instance, you can specify the number of training workers with `num_workers` and the usage of GPUs with the `use_gpu` flag. We'll elaborate on this more in the next section.

Here's how you set up your trainer in our example:

```
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()
```

With an initialized trainer, you can call `fit()`, which will execute the training across your Ray cluster. Here's a quick visual summary of working with a `TorchTrainer`:

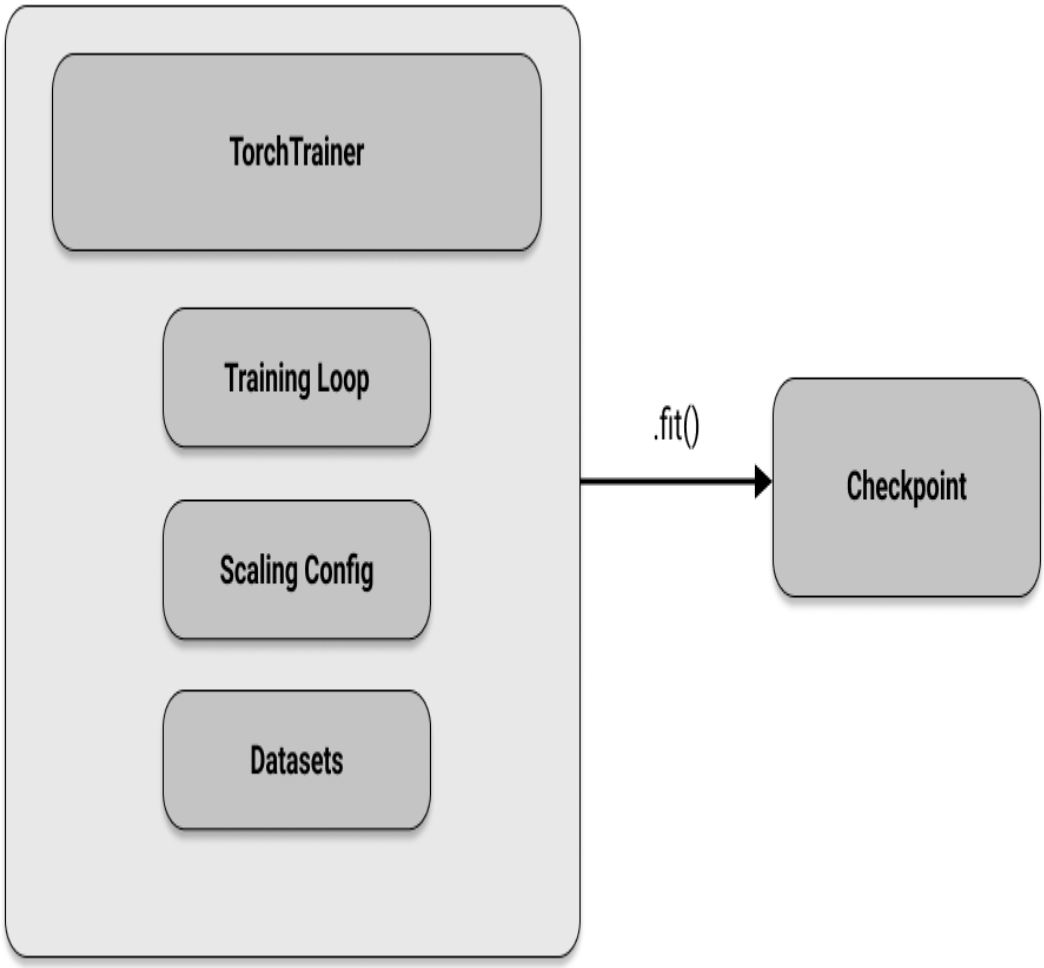


Figure 7-2. Working with a `TorchTrainer` requires you to specify a training loop, datasets, and a scaling configuration.

Scaling Out Trainers

The general philosophy of Ray Train is that the user should not need to think about how to parallelize their code. Specifying a `scaling_config` allows you to scale out your training without writing distributed logic and to declaratively specify the *compute resources* used by a Trainer. The nice thing about this specification is that you don't need to think about the underlying hardware. In particular, you can use hundreds of workers by specifying the

parameters of your cluster nodes in your `ScalingConfig` accordingly:

```
import ray
from ray.air.config import ScalingConfig
from ray.train.xgboost import XGBoostTrainer

ray.init(address="auto")  ❶

scaling_config = ScalingConfig(num_workers=200,
use_gpu=True)  ❷

trainer = XGBoostTrainer(  ❸
    scaling_config=scaling_config,
    # ...
)
```

❶ We connect to an existing, large Ray cluster here.
❷ Then we define a `ScalingConfig` according to the cluster's available resources.
❸ We can then, for instance, use Ray Train's XGBoost integration to train a model on this cluster.

Preprocessing with Ray Train

Data preprocessing is a common technique for transforming raw data into features for a machine learning model, and we've seen many examples of this in this and the previous chapter. So far we've "manually" preprocessed our data by writing custom functions to transform our data into the right format. But Ray Train has several built-in preprocessors for common use cases and also provides interfaces to define your own custom logic.

The `Preprocessor` is the core class offered by Ray Train for handling data preprocessing. Each preprocessor has the following APIs:

`.transform()`	Used to process and apply a processing transformation to a Dataset.
`.fit()`	Used to calculate and store aggregate state about the Dataset on Preprocessor. Returns self for chaining.
`.fit_transform()`	Syntactic sugar for performing transformations that require aggregate state. May be optimized at the implementation level for specific Preprocessors.
`.transform_batch()`	Used to apply the same transformation on batches for prediction.

You will often want to make sure you can use the same data preprocessing operations at training time and at serving time. Training-serving skew is a major problem in deploying machine learning, and it describes a situation where there is a difference between performance during training and performance during serving. This skew is often caused by a discrepancy between how you handle data in the training and serving pipelines. Thus, you want to make sure you have consistent data handling for training and serving.

You can use the above preprocessors by passing them to the constructor of a `trainer`. That means once you've created a `preprocessor`, you don't have to apply it to your Datasets manually. Instead, you can pass it to your `trainer` and Ray Train will take care of applying it in a distributed fashion. Here is how this works schematically (we'll discuss a concrete example in the next section):

```
from ray.data.preprocessors import StandardScaler
from ray.train.xgboost import XGBoostTrainer


trainer = XGBoostTrainer(
    preprocessor=StandardScaler(...),
    # ...
)
result = trainer.fit()
```

Now, some preprocessing operators such as one-hot encoders are easy to run in training and transfer to serving. However, other operators such as those that do standardization are a bit trickier, since you don't want to do large data crunching (to find the mean of a particular column) during serving time.

The nice thing about the Ray Train preprocessors is that they're serializable. This makes it so that you can easily get consistency from training to serving just by serializing these operators. For instance, you can simply pickle a `preprocessor` like this:

```
import pickle
from ray.data.preprocessors import StandardScaler

preprocessor=StandardScaler(...)
pickle.dumps(preprocessor)
```

Let's next discuss a concrete example of a training procedure using preprocessors, by also showing you how to tune your Trainer's hyperparameters.

Integrating Trainers with Ray Tune

Ray Train provides an integration with Ray Tune that allows you to perform hyperparameter optimization in just a few lines of code. Tune will create one Trial per hyperparameter configuration. In each Trial, a new Trainer will be initialized and run the training function with its generated configuration.

In the below code, we create an `XGBoostTrainer` and specify hyperparameter ranges for common hyperparameters. Specifically, we're going to choose between two different *preprocessors* in our training scenario.

To be precise, we will use a `StandardScaler`, which translates and scales each specified column by its mean and standard deviation (the resulting columns will thus follow a standard normal distribution) and `MinMaxScaler`, which simply scales each column to the range `[0, 1]`.

Here's corresponding the parameter space that we will search over next:

```
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),
    },
}
```

We can now create a `Trainer` as before, this time going with an `XGBoostTrainer`, and then pass it to an instance of a `Tuner` from Ray Tune, which we can `.fit()` just like the `trainer` itself:

```
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()
```

Note that we're using another component of a Ray `Trainer` here that you haven't seen before, the `RunConfig`. This configuration is used for all runtime options of a trainer, in our case the log verbosity of the experiment (`0` would be silent, `1` gives only status updates, `2`, the default, gives status updates and brief results, and `3` gives detailed results).

Compared to other distributed hyperparameter tuning solutions, Ray Tune and Ray Train have some unique features. Ray's solution is

fault-tolerant, and it has the ability to specify the dataset and preprocessor as a parameter, as well as to adjust the number of workers during training time.

Using Callbacks to Monitor Training

To discuss one more feature of Ray Train, you may want to plug in your training code with your favorite experiment management framework. Ray Train provides an interface to fetch intermediate results and callbacks to process or log your intermediate results. It comes with built-in callbacks for popular tracking frameworks, but you can implement your own callback via Tune's so-called `LoggerCallback` interface.

For instance, you can log results in JSON format using the `JsonLoggerCallback`, to TensorBoard using the `TBXLoggerCallback`, to MLFlow using the `MLFLowLoggerCallback`.[3] The following example shows how you can use all three in one single training run by specifying a list of `callbacks`:

```
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()
])
```

Summary

In this chapter, we've discussed the basics of distributed model training and showed you how to run data-parallel training with Ray Train. We walked you through an extensive example that used both Ray Data and Ray Train on an interesting data set. Specifically, we demonstrated how to use Dask on Ray to load, preprocess, and featurize your datasets, and then use Ray Train to run a PyTorch training loop in a distributed fashion. We then discussed Ray Trainers in more detailed and showed you how they integrate with Ray Tune via Tuners, how you can use them with preprocessors, and how you can use callbacks to monitor training.

---

1 This applies specifically to the gradient computation in neural networks.

2 The code only loads a subset of the data for testing, to run at scale we would use all partitions of the data when calling `load_dataset` and increase `num_workers` when training the model.

3 MLflow and Tensorboard are open source projects for machine learning experiment tracking and visualization. They are very useful for monitoring the progress of your machine learning model.

Chapter 8. Online Inference with Ray Serve

Edward Oakes

In previous chapters you've learned how to use Ray to process data, train machine learning (ML) models, and apply them in a batch inference setting. However, many of the most exciting use cases for machine learning involve "online inference".

Online inference is the process of using ML models to enhance API endpoints that users interact with directly or indirectly. This is important in situations where latency matters: you can't simply apply models to data behind the scenes and serve the results. There are many real-world examples of use cases where online inference can provide a lot of value:

Recommendation systems

> Providing recommendations for products (e.g., online shopping) or content (e.g., social media) is a bread-and-butter use case for machine learning. While it's possible to do this in an offline manner, recommendation systems often benefit from reacting to users' preferences in real time. This requires performing online inference using recent behavior as a key feature.

Chat bots

Online services often have real-time chat windows to provide support to customers from the comfort of their keyboard. Traditionally, these chat windows were staffed by customer support staff but there has been a recent trend to reduce labor costs and improve time-to-resolution by replacing them with ML-powered chat bots that can be online 24/7. These chat bots require a sophisticaed mix of multiple machine learning techniques and must be able to respond to customer input in real time.

Estimating arrival times

Ride sharing, navigation, and food delivery services all rely on being able to provide an accurate estimation for arrival times (e.g., for your driver, yourself, or your dinner). Providing accurate estimates is very difficult because it requires accounting for real-world factors such as traffic patterns, weather, and accidents. Estimates are also often refreshed many times over the course of one trip.

These are just a few examples where applying machine learning in an online setting can provide a lot of value in application domains that are traditionally very difficult (imagine writing logic to estimate arrival time by hand!). The list of applications goes on: a number of nascent domains such as self-driving cars, robotics, video-processing pipelines are also being redefined by machine learning.

All these applications share one crucial requirement: latency. In the case of online services, low latency is paramount in order to provide a good user experience. For applications that interact with the real world (such as robotics or self-driving cars), higher latency can have even stronger implications in safety or correctness.

This chapter will provide a gentle introduction to Ray Serve, a Ray-native library that enables building online inference applications on top of Ray. First, we will discuss the challenges of online inference

that Ray Serve addresses. Then, we'll cover the architecture of Ray Serve and introduce its core set of functionality. Finally, we will use Ray Serve to build an end-to-end online inference API consisting of multiple natural language processing models. You can follow along with the code in the notebook for this chapter.

Key Characteristics of Online Inference

In the previous section we discussed that the main goal of online inference is to interact with machine learning models with low latency. However, this has long been a key requirement for API backends and web servers, so a natural question is: what's different about serving machine learning models?

ML Models Are Compute Intensive

Many of the challenges in online inference are a result of one key characteristic: machine learning models are very compute intensive. Compared to traditional web serving where requests are primarily handled by I/O intensive database queries or other API calls, most machine learning models boil down to performing many linear algebra computations, be it to provide a recommendation, estimate an arrival time, or detect an object in an image. This is especially true for the recent trend of "deep learning," which is an arm of machine learning characterized by neural networks that are growing larger and larger over time. Often, deep learning models can also benefit significantly from using specialized hardware such as GPUs or TPUs, which have special-purpose instructions optimized for machine learning computations and enable vectorized computations across multiple inputs in parallel.

Many online inference applications are required to be run 24/7 by nature. When combined with the fact that machine learning models are compute intensive, operating online inference services can be very expensive, requiring allocating many CPUs and GPUs at all

times. The primary challenges of online inference boil down to serving models in a way that minimizes end-to-end latency while also reducing cost. There are a few key properties that online inference systems provide in order to satisfy these requirements:

- Support for specialized hardware such as GPUs and TPUs.
- The ability to scale up and down the resources used for a model in response to request load.
- Support for request batching to take advantage of vectorized computations.

ML Models Aren’t Useful in Isolation

Often when machine learning is discussed in the academic or research setting, the focus is on an individual, isolated task such as object recognition or classification. However, in the real world applications are not usually so clear cut and well-defined. Instead, a combination of multiple ML models and business logic is required to solve a problem end-to-end. For example, consider a product recommendation use case. While there are a multitude of known ML techniques we could apply to the core problem of making a recommendation, there are also a lot of equally important challenges around the edges, many of which will be specific to each use case:

- Validating inputs and outputs to ensure the result returned to the user makes sense semantically. Often, we may have some manual rules such as avoiding returning the same recommendation to a user multiple times in succession.
- Fetching up-to-date information about the user and available products and converting it into features for the model (in some cases, this may be performed by an online feature store).
- Combining the results of multiple models using manual rules such as filtering to the top results or selecting the model with

highest confidence.

Implementing an online inference API requires the ability to integrate all of these pieces together into one unified service. Therefore, it's important to have the flexibility to compose multiple models together along with custom business logic. These pieces can't really be viewed completely in isolation: the "glue" logic often needs to evolve alongside the models themselves.

An Introduction to Ray Serve

Ray Serve is a scalable compute layer for serving machine learning models on top of Ray. Serve is framework-agnostic, meaning that it isn't tied to a specific machine learning library, but rather treats models as ordinary Python code. Additionally, it allows you to flexibly combine normal Python business logic alongside machine learning models. This makes it possible to build online inference services completely end-to-end: a Serve application could validate user input, query a database, perform scalable inference across multiple ML models, and combine, filter, and validate the output all in the process of handling a single inference request. Indeed, combining the results of multiple machine learning models is one of the key strengths of Ray Serve, as you'll see later in the chapter as we explore common multi-model serving patterns.

While being flexible in nature, Serve has purpose-built features for compute-heavy machine learning models, enabling dynamic scaling and resource allocation to ensure that request load can be handled efficiently across many CPUs and/or GPUs. Here, Serve inherits a lot of benefits from being built on top of Ray: it's scalable to hundreds of machines, offers flexible scheduling policies, and offers low-overhead communication across processes using Ray's core APIs.

This section will incrementally introduce core functionality from Ray Serve with a focus on how it helps to address the challenges of online inference as outlined above. To follow along with the code samples in this section, you'll need the following Python packages installed locally:

```
pip install "ray[serve]==2.2.0" "transformers==4.21.2"
"requests==2.28.1"
```

Running the examples assumes that you have the code saved locally in a file named `app.py` in the current working directory.

Architectural Overview

Ray Serve is built on top of Ray, so it inherits a lot of benefits such as scalability, low overhead communication, an API well-suited to parallelism, and the ability to leverage shared memory via the object store. The core primitive in Ray Serve is a **deployment**, which you can think of as a managed group of Ray actors that can be addressed together and will handle requests load-balanced across them. Each actor in a deployment is called a **replica** in Ray Serve. Often, a deployment will map one-to-one with a machine learning model, but deployments can contain arbitrary Python code so they might also house business logic.

Ray Serve enables exposing deployments over HTTP and defining the input parsing and output logic. However, one of the most important features of Ray Serve is that deployments can also call into each other directly using a native Python API, which will translate to direct actor calls between the replicas. This enables flexible, high performance composition of models and business logic; you'll see this in action later in the section.

Figure 8-1 provides a basic view of how a Ray Serve application runs on top of a Ray cluster: multiple deployments are placed across a Ray cluster. Each deployment is made up of one more "replicas",

each of which is a Ray actor. Incoming traffic is routed through an HTTP proxy that will load balance requests across the replicas.[1]

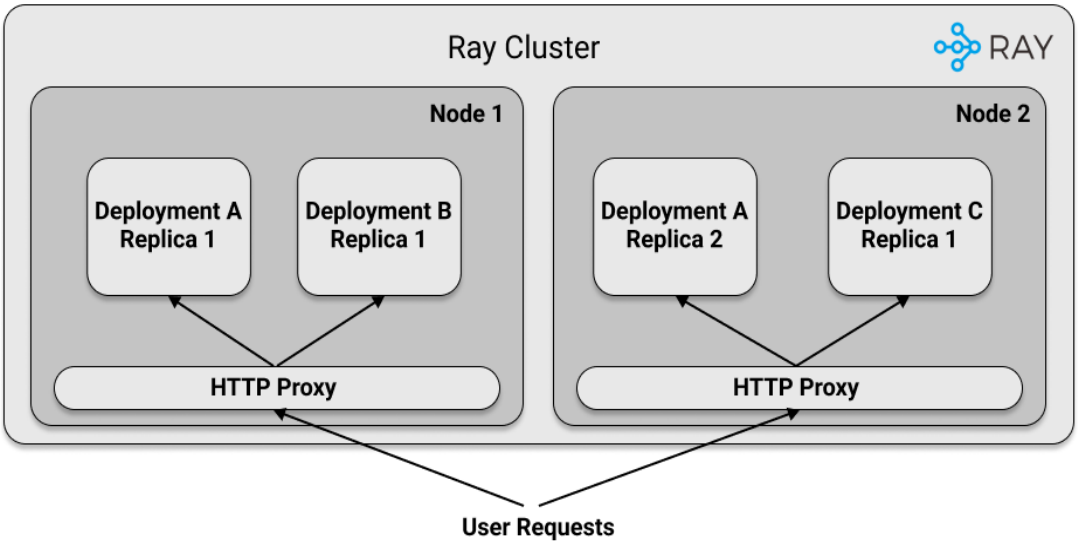


Figure 8-1. The architecture of a Ray Serve application consisting of multiple deployments. Each deployment is made up by a group of Ray actors and the deployments are spread across multiple nodes in a Ray cluster. The HTTP proxy accepts requests and load balances them to the downstream actors.

Under the hood, the deployments making up a Ray Serve application are managed by a centralized **controller** actor. This is a detached actor that is managed by Ray and will be restarted upon failure. The controller is in charge of creating and updating replica actors, broadcasting updates to other actors in the system, and performing health checking and failure recovery. If a replica or an entire Ray node crashes for any reason, the controller will detect the failures and ensure that the actors are recovered and can continue serving traffic.

Defining a Basic HTTP Endpoint

This section will introduce Ray Serve by defining a simple HTTP endpoint wrapping a single ML model. The model we'll deploy is a sentiment classifier: given a text input, it will predict if the output had a positive or negative sentiment. We'll be using a pretrained

sentiment classifier from the Hugging Face `transformers` library, which provides a simple Python API for pretrained models that will abstract away the details of the model and allow us to focus on the serving logic.

To deploy this model using Ray Serve, we need to define a Python class and turn it into a Serve **deployment** using the `@serve.deployment` decorator. The decorator allows us to pass a number of useful options to configure the deployment; we will explore some of those options later in the chapter.

```
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"]
```

There are a few important points to note here. First, we instantiate our model in the constructor of the class. This model may be very large, so downloading it and loading it into memory can be slow (10s of seconds or longer). In Ray Serve, the code in the constructor will only be run once in each replica on startup and any properties can be cached for future use. Second, we define the logic to handle a request in the `__call__` method. This takes a `Starlette` HTTP request as input and can return any JSON-serializable output. In this case, we'll return a single string from the output of our model: `"POSITIVE"` or `"NEGATIVE"`.

Once a deployment is defined, we use the `.bind()` API to instantiate a copy of it. This is where we can pass optional

arguments to the constructor to configure the deployment (such as a remote path to download model weights from). Note that this doesn't actually run the deployment, but just packages it up with its arguments (this will be more important later when we combine multiple models together).

```
basic_deployment = SentimentAnalysis.bind()
```

We can run the bound deployment using the `serve.run` Python API or corresponding `serve run` CLI command. Assuming you save the above code in a file called `app.py`, you can run it locally with the following command:

```
serve run app:basic_deployment
```

This will instantiate a single replica of our deployment and host it behind a local HTTP server. To test it, we can use the Python `requests` package:

```
import requests

print(requests.get(
    "http://localhost:8000/", params={"input_text": "Hello
friend!"}
).json())
```

Testing the sentiment classifier on a sample input text of `"Hello friend!"`, it correctly classifies the text as positive!

This example is effectively the "hello world" of Ray Serve: we deployed a single model behind a basic HTTP endpoint. Note, however, that we had to manually parse the input HTTP request and feed it into our model. For this basic example it was just a single line of code, but real world applications often take a more complex schema as input and hand-writing HTTP logic can be tedious and

error-prone. To enable writing more expressive HTTP APIs, Serve integrates with the FastAPI Python framework.[2]

A Serve deployment can wrap a `FastAPI` app, making use of its expressive APIs for parsing inputs and configuring HTTP behavior. In the following example, we rely on `FastAPI` to handle parsing the `input_text` query parameter for us, allowing to remove the boilerplate parsing code.

```
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()
```

The modified deployment should have exactly the same behavior on the example above (try it out using `serve run`!), but will gracefully handle invalid inputs. These may look like small benefits for this simple example, but for more complex APIs this can make a world of difference. We won't delve deeper into the details of FastAPI here, but for more details on its features and syntax check out their excellent documentation.

Scaling and Resource Allocation

As mentioned above, machine learning models are often compute hungry. Therefore, it's important to be able to allocate the correct

amount of resources to your ML application to handle request load while minimizing cost. Ray Serve allows you to adjust the resources dedicated to a deployment in two ways: by tuning the number of **replicas** of the deployment and tuning the resources allocated to each replica. By default, a deployment consists of a single replica that uses a single CPU but these parameters can be adjusted in the `@serve.deployment` decorator (or using the corresponding `deployment.options` API).

Let's modify the `SentimentClassifier` example from above to scale out to multiple replicas and adjust the resource allocation so that each replica uses 2 CPUs instead of 1 (in practice, you would want to profile and understand your model in order to set this paramter correctly). We'll also add a print statement to log the process ID of the process handling each request to show that the requests are now load balanced across two replicas.

```
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()
```

Running this new version of our classifier with `serve run app:scaled_deployment` and querying it using `requests` as we did above, you should see that there are now two copies of the

model handling requests! We could easily scale up to tens or hundreds of replicas just by tweaking `num_replicas` in the same way: Ray enables scaling to hundreds of machines and thousands of processes in a single cluster.

In this example we scaled out to a static number of replicas with each replica consuming two full CPUs, but Serve also supports more expressive resource allocation policies:

- Enabling a deployment to use GPUs simpliy requires setting `num_gpus` instead of `num_cpus`. Serve supports the same resource types as Ray core, so deployments can also use TPUs or other custom resources.
- Resources can be **fractional**, allowing replicas to be efficiently bin-packed. For example, if a single replica doesn't saturate a full GPU you can allocate `num_gpus=0.5` to it and multiplex with another model.
- For applications with varying request load, a deployment can be configured to dynamically autoscale the number of replicas based on the number of requests currently in flight.

For more details about resource allocation options, refer to the latest Ray Serve documentation.

Request Batching

Many machine learning models can be efficiently **vectorized**, meaning that multiple computations can be run in parallel more efficiently than running them sequentially. This is especially beneficial when running models on GPUs which are purpose-built for efficiently performing many computations in parallel. In the context of online inference this offers a path for optimization: serving multiple requests (possibly from different sources) in parallel can drastically improve the throughput of the system (and therefore save cost).

There are two high-level strategies to take advantage of request batching: **client-side** batching and **server-side** batching. In client-side batching, the server accepts multiple inputs in a single request and clients include logic to send them in batches instead of one at a time. This is mostly useful in situations where a single client is frequently sending many inference requests. Server-side batching, in contrast, enables the server to batch multiple requests without requiring any modification on the client. This can also be used to batch requests across multiple clients, which enables efficient batching even in situations with many clients that each sends relatively few requests.

Ray Serve offers a built-in utility for server-side batching, the `@serve.batch` decorator, that requires just a few code changes. This batching support uses Python's `asyncio` capabilities to enqueue multiple requests into a single function call. The function should take in a list of inputs and return the corresponding list of outputs.

Once again, let's revisit the sentiment classifier from earlier and this time modify it to perform server-side batching. The underlying Hugging Face `pipeline` supports vectorized inference, all we need to do is pass a list of inputs, and it will return the corresponding list of outputs. We'll split out the call to the classifier into a new method, `classify_batched`, that will take a list of input texts as input, perform inference across them, and return the outputs in a formatted list. `classify_batched` will use the `@serve.batch` decorator to automatically perform batching. The behavior can be configured using the `max_batch_size` and `batch_timeout_wait_s` parameters, here we'll set the max batch size to 10 and wait for up to 100ms.

```
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()
```

Notice that both the `classify` and `classify_batched` methods now use Python's `async` and `await` syntax, meaning that many of these calls can run concurrently in the same process.

To test this behavior, we'll use the `serve.run` Python API to send requests using the Python-native handle to our deployment.

```
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 Get a handle to the deployment, so we can send requests in parallel.

The handle returned by `serve.run` can be used to send multiple requests in parallel: here, we send 10 requests in parallel and wait for them all to return. Without batching, each request would be handled sequentially, but because we enabled batching we should

see the requests handled all at once (evidenced by batch size printed in the `classify_batched` method). Running on a CPU, this might be marginally faster than running sequentially, but running the same handler on a GPU we would observe a significant speedup for the batched version.

Multi-Model Inference Graphs

Up until now, we've been deploying and querying a single Serve deployment wrapping one ML model. As described earlier, machine learning models are not often useful in isolation: many applications require multiple models to be composed together and for business logic to be intertwined with machine learning. The real power of Ray Serve is in its ability to compose multiple models along with regular Python logic into a single application. This is possible by instantiating many different deployments and passing reference between them. Each of these deployments can use all of the features we've discussed up until this point: they can be independently scaled, perform request batching, and use flexible resource allocations.

This section provides illustrative examples of common multi-model serving patterns but doesn't actually contain any ML models in order to focus on the core capabilities that Serve provides. Later in the chapter we will explore an end-to-end multi-model inference graph containing ML models.

Core feature: binding multiple deployments

All types of multi-model inference graphs in Ray Serve center around the ability to pass a reference to one deployment into the constructor of another. In order to do this, we use another feature of the `.bind()` API: a bound deployment can be passed to another call to `.bind()` and this will resolve to a "handle" to the deployment at runtime. This enables deployments to be deployed and instantiated independently and then call each other at runtime. Below is the most basic example of a multi-deployment Serve application.

```python
@serve.deployment
class DownstreamModel:
    def __call__(self, inp: str):
        return "Hi from downstream model!"


@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)
```

In this example, the downstream model is passed into the “driver” deployment. Then at runtime the driver deployment calls into the downstream model. The driver could take any number of models passed in, and the downstream model could even take other downstream models of its own.

Pattern 1: Pipelining

The first common multi-model pattern among machine learning applications is “pipelining:” calling multiple models in sequence, where the input of one model depends on the output of the previous. Image processing, for example, often consists of a pipeline consistenting of multiple stages of transformations such as cropping, segmentation, and object recognition or optical character recognition (OCR). Each of these models may have different properties with some of them being lightweight transformations that can run on a CPU and others being heavyweight deep learning models that run on a CPU.

Such pipelines can easily be expressed using Serve’s API. Each stage of the pipeline is defined as an independent deployment and each deployment is passed into a top-level “pipeline driver.” In the

example below, we pass two deployments into a top-level driver and the driver calls them in sequence. Note that there could be many requests to the driver happening concurrently, therefore it is possible to efficiently saturate all stages of the pipeline.

```
@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)
```

To test this example, you can once again use the `serve run` API. Sending a test request to the pipeline returns `"'input|val1|val2'"` as output: each downstream “model” appended its own value to construct the final result. In practice, each of these deployments could be wrapping its own ML model and a single request may flow across many physical nodes in a cluster.

Pattern 2: Broadcasting

In addition to sequentially chaining models together, it’s often useful to broadcast an input or intermediate result to multiple models in

parallel. This could be to perform “ensembling,” or combining the results of multiple independent models into a single result, or in a situation where different models may perform better on different inputs. Often the results of the models need to be combined in some way into a final result: either simply concatenated together or maybe a single result chosen from the lot.

This is expressed very similarly to the pipelining example: a number of downstream models are passed into a top-level “driver.” In this case, it’s important that we call the models in parallel: waiting for the result of each before calling the next would dramatically increase the overall latency of the system.

```
@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)
```

Testing this endpoint after running it once again with `serve run` returns `'["val1", "val2"]'`, the combined output of the two

models called in parallel.

Pattern 3: Conditional logic

Finally, while many machine learning applications fit roughly into one of the above patterns, often having static control flow can be very limiting. Take, for instance, the example of building a service to extract license plate numbers from user-uploaded images. In this case, we'll likely need to build an image processing pipeline as discussed above, but we also don't simply want to feed any image into the pipeline blindly. If the user uploads something other than a car or with an image that is low quality, we likely want to short circuit, avoid calling into the heavy-weight and expensive pipeline, and provide a useful error message. Similarly, in a product recommendation use case we may want to select a downstream model based on user input or the result of an intermediate model. Each of these examples requires embedding custom logic alongside our ML models.

We can accomplish this trivially using Serve's multi-model API because our computation graph is defined as ordinary Python logic rather than as a statically-defined graph. For instance, in the example below, we use a simple random number generator (RNG) to decide which of two downstream models to call into. In a real-world example, the RNG could be replaced with business logic, a database query, or the result of an intermediate model.

```
@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)
```

Each call to this endpoint returns either `"val1"` or `"val2"` with 50/50 probability.

End-to-End Example: Building an NLP-Powered API

In this section, we'll use Ray Serve to build an end-to-end natural language processing (NLP) pipeline hosted for online inference. Our goal will be to provide a Wikipedia summarization endpoint that will leverage multiple NLP models and some custom logic to provide a succinct summary of the most relevant Wikipedia page for a given search term.

This task will bring together many of the concepts and features discussed above:

- We'll be combining custom business logic along with multiple machine learning models.
- The inference graph will consist of all three multi-model patterns discussed above: pipelining, broadcasting, and conditional logic.

- Each model will be hosted as a separate Serve deployment, so they can be independently scaled and given their own resource allocation.
- One of the models will leverage vectorized computation via batching.
- The API will be defined using Ray Serve's `FastAPI` for input parsing and defining our output schema.

Our online inference pipeline will be structured as shown in Figure 8-2:

- The user will provide a keyword search term.
- We'll fetch the content for the most relevant Wikipedia article for the search term.
- A sentiment analysis model will be applied to the article. Anything with a "negative" sentiment will be rejected and we'll return early.
- The article content will be broadcast to summarizer and named entity recognition models.
- We'll return a composed result based on the summarizer and named entity recognition outputs.

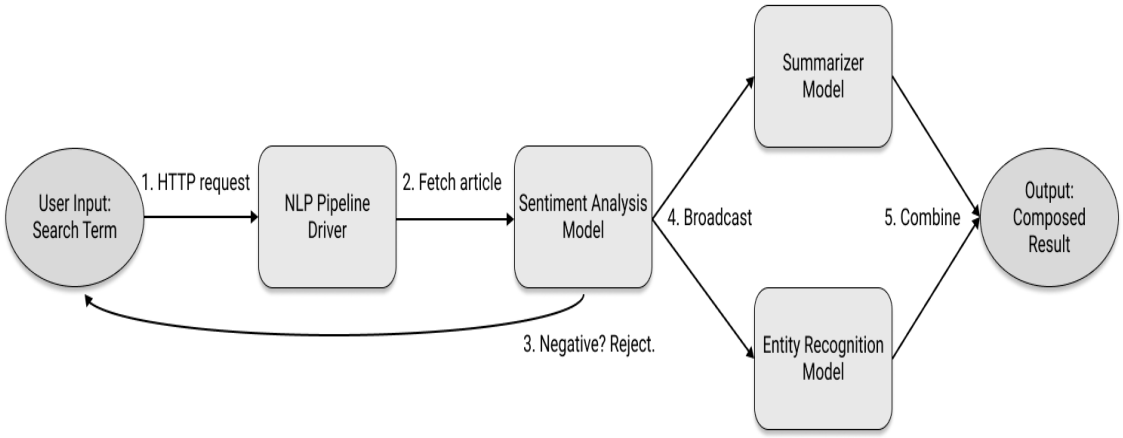


Figure 8-2. The architecture for a natural language processing pipeline to summarize Wikipedia articles. The pipeline consists of multiple stages and models: if an article has negative sentiment, it is rejected early. If not, it is broadcast to multiple downstream models and the results are combined into an aggregated result.

This pipeline will be exposed over HTTP and return the results in a structured format. By the end of this section, we'll have the pipeline running end-to-end locally and ready to scale up on a cluster. Let's get started!

Before we dive into the code, you'll need the following Python packages installed locally in order to follow along:

```
pip install "ray[serve]==2.2.0" "transformers==4.21.2"
pip install "requests==2.28.1" "wikipedia==1.4.0"
```

Additionally, in this section we'll assume that all the code samples are available locally in a file called `app.py` so we can run the deployments using `serve run` from the same directory.[3]

Fetching Content and Preprocessing

The first step is to fetch the most relevant page from Wikipedia given a user-provided search term. For this, we will leverage the `wikipedia` package on PyPI to do the heavy lifting. We'll first search for the term, then select the top result and return its page content. If no results are found, we'll return `None` — this edge case will be handled later when we define the API.

```
from typing import Optional

import wikipedia


def fetch_wikipedia_page(search_term: str) ->
Optional[str]:
    results = wikipedia.search(search_term)
    # If no results, return to caller.
    if len(results) == 0:
        return None

    # Get the page for the top result.
    return wikipedia.page(results[0]).content
```

NLP Models

Next, we need to define the ML models that will do the heavy lifting of our API. As we did in the introduction section, we'll be using the Hugging Face `transformers` library as it provides convenient APIs to pretrained state-of-the-art ML models, so we can focus on the serving logic.

The first model we'll use is a sentiment classifier, the same one we used in the examples above. The deployment for this model will take advantage of vectorized computations using Serve's batching API.

```
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_wait_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)
```

We'll also use a text summarization model to provide a succinct summary for the selected article. This model takes an optional "max_length" argument to cap the length of the summary. Because we know this is the most computationally expensive of the models, we set `num_replicas=2` — that way, if we have many requests coming in at the same time, it can keep up with the throughput of the other models. In practice, we may need more replicas to keep up with the input load, but we could only know that from profiling and monitoring.

```
@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"]
```

The final model in our pipeline will be a named entity recognition model: this will attempt extract named entities from the text. Each result will have a confidence score, so we can set a threshold to only

accept results above a certain threshold. We may also want to cap the total number of entities returned. The request handler for this deployment calls the model, then uses some basic business logic to enforce the provided confidence threshold and limit on the number of entities.

```
@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
```

HTTP Handling and Driver Logic

With the input preprocessing and ML models defined, we're ready to define the HTTP API and driver logic. First, we define the schema of the response that we'll return from the API using Pydantic. The response includes whether the request was successful and a status message in addition to our summary and named entities. This will allow us to return a helpful response in error conditions such as when no result is found or the sentiment analysis comes back as negative.

```
from pydantic import BaseModel


class Response(BaseModel):
    success: bool
```

```
    message: str = ""
    summary: str = ""
    named_entities: List[str] = []
```

Next, we need to define the actual control flow logic that will run in the driver deployment. The driver will not do any of the actual heavy lifting itself, but instead call into our three downstream model deployments and interpret their results. It will also house the FastAPI app definition, parsing the input and returning the correct `Response` model based on the results of the pipeline.

```
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:
        # Fetch the top page content for the search term if
found.
        page_content = fetch_wikipedia_page(search_term)
        if page_content is None:
            return Response(success=False, message="No
pages found.")

        # Conditionally continue based on the sentiment
analysis.
        is_positive = await
self._sentiment_analysis.remote(page_content)
        if not is_positive:
            return Response(success=False, message="Only
positivitiy allowed!")
```

```
        # Query the summarizer and named entity recognition
models in parallel.
        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
        )
```

In the body of the main handler, we first fetch the page content using our `fetch_wikipedia_page` logic (if no result is found, an error is returned). Then, we call into the sentiment analysis model. If this returns negative, we terminate early and return an error to avoid calling the other expensive ML models. Finally, we broadcast the article contents to both the summary and named entity recognition models in parallel. The results of the two models are stitched together into the final response, and we return success. Remember that we may have many calls to this handler running concurrently: the calls to the downstream models don't block the driver, and it could coordinate calls to many replicas of the heavyweight models.

Putting It All Together

At this point, we have defined all the core logic. All that's left is to bind the graph of deployments together and run it!

```
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)
```

First, we need to instantiate each of the deployments with any relevant input arguments. For example, here we pass a threshold

and limit for the entity recognition model. The most important piece is we pass a reference to each of the three models into the driver, so it can coordinate the computation. Now that we've defined the full NLP pipeline, we can run it using `serve run`:[4]

```
serve run app:nlp_pipeline_driver
```

This will deploy each of the four deployments locally and make the driver available at `http://localhost:8000/`. We can query the pipeline using the `requests` to see it in action. First, let's try querying for an entry on Ray Serve.

```
import requests


print(requests.get(
    "http://localhost:8000/", params={"search_term":
"rayserve"}
).text)

'{"success":false,"message":"No pages
found.","summary":"","named_entities":[]}'
```

Unfortunately, this page doesn't exist yet! The first chunk of validation business logic kicks in and returns a "No pages found" message. Let's try finding looking for something more common:

```
print(requests.get(
    "http://localhost:8000/", params={"search_term": "war"}
).text)

'{"success":false,"message":"Only positivitiy
allowed!,"summary":"","named_entities":[]}'
```

Maybe we were just interested in learning about history, but this article was a bit too negative for our sentiment classifier. Let's try something more neutral this time — what about science?

```
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"]}'
```

This example successfully ran through the full pipeline: the API responded with a cogent summary of the article and a list of relevant named entities.

To recap, in this section we were able to build an online natural language processing API using Ray Serve. This inference graph consisted of multiple machine learning models in addition to custom business logic and dynamic control flow. Each model can be independently scaled and have its own resource allocation, and we can exploit vectorized computations using server-side batching. Now that we were able to test the API locally, the next step would be to deploy to production. Ray Serve makes it easy to deploy on Kubernetes or other cloud provider offerings using the Ray cluster launcher, and we could easily scale up to handle many users by tweaking the resource allocations for our deployments.

Summary

This chapter introduced Ray Serve, a Ray-native library for building online inference APIs. Ray Serve is focused on solving the unique

challenges of serving machine learning models in production, offering functionality to efficiently scale models and allocate resources as well as compose multiple models together along with business logic. Additionally, like all of Ray, Serve is designed to be a general-purpose solution that avoids vendor lock in.

Although we walked through an end-to-end example of a multi-model pipeline, this chapter has only covered a small portion of Ray Serve's functionality and best practices for real-world applications. For more detail and examples, you're encouraged to read the Ray Serve documentation.

---

1 Deployments can also send traffic to each other directly. This enables building more complex applications involving model composition or mixing ML models with business logic. This will be explored in a later section.

2 Under the hood, Ray Serve serializes the user-provided `FastAPI` app object. Then, when each deployment replica runs, Ray Serve deserializes and runs the `FastAPI` app the same way it would be run in a typical web server. At runtime, there will be an independent `FastAPI` server running in each Ray Serve replica.

3 We've created this `app.py` file for you in the GitHub repository for this book. After cloning the repo and installing all dependencies, you should be able to directly run `serve run app:<deployment_name>` from the `notebook` directory for each deployment we're referencing in a `serve run` call in this chapter.

4 The models we use in this example are quite large, so please be warned that it will likely take several minutes to download them the first time you run this example.

Chapter 9. Ray Clusters

Richard Liaw

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

In the last eight chapters we've been focused on teaching you the basics of Ray for building machine learning applications. You know how to parallelize your Python code with Ray Core and run reinforcement learning experiments with RLlib. You've also seen how to preprocess data with Ray Datasets, tune hyperparameters with Ray Tune, and train models with Ray Train. But one of the key features that Ray brings is the ability to scale out seamlessly onto multiple machines. Outside a lab environment or a big tech company, it may be difficult set up multiple machines and join them into a single Ray cluster. This chapter is all about how to do that.[1]

Cloud technology has commoditized access to cheap machines for anyone. But it is often quite difficult to figure out the right APIs to handle the cloud provider tools. The Ray team has provided a couple tools that abstract the complexity away. There are three primary ways of launching or deploying a Ray cluster. You can do so manually, via a Kubernetes Operator, or via the cluster launcher CLI tool.

In the first part of this chapter we're going to cover these three methods in detail.[2] We only briefly explain manual cluster creation and the cluster launcher CLI, and spend most of our time on explaining how to use the Kubernetes Operator. After that, we'll

cover how run Ray Clusters on the cloud and how to auto-scale them up and down.

Manually Creating a Ray Cluster

Let's start with the most basic way of creating a Ray cluster. To build clusters manually we assume that you have a list of machines that can communicate with each other and have Ray installed on them.[3]

To start with, you can choose any machine to be the head node. On this node, run the following command.

```
ray start --head --port=6379
```

This command will print out the IP address of the Ray GCS server that was started, namely the local node IP address plus the port number you specified:[4]

```
...
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.
```

You need this `<head-address>` to connect your other nodes to the cluster, so make sure to copy it. If you omit the `--port` argument, Ray will use a random port.

Next, we can connect every other node in your cluster to the head node by running a single command on each node:

```
ray start --address=<head-address>
```

Make sure to pass the correct `<head-address>`, which should look something like `123.45.67.89:6379`. Running this command, you

should see output of the following form:

```
--------------------
Ray runtime started.
--------------------

To terminate the Ray runtime, run
  ray stop
```

If you wish to specify that a machine has 10 CPUs and 1 GPU, you can do this with the flags `--num-cpus=10` and `--num-gpus=1`. If you see `Ray runtime started.`, then the node successfully connected to the head node at the `--address`. You should now be able to connect to the cluster with `ray.init(address='auto')`.

> **NOTE**
>
> If the head node and the new node you want to connect are on a separate subnetwork with Network Address Translation (NAT), you can't use the `<head-address>` printed by the command starting the head node as `--address`. In this case, you need to find the address that will reach the head node from the new node. If the head node has a domain address like `compute04.berkeley.edu`, you can simply use that in place of an IP address and rely on the DNS.
>
> If you see `Unable to connect to GCS at ...`, this means the head node is inaccessible at the given `--address`. There could be several reasons for this, such as the head node is not actually running, a different version of Ray is running at the specified address, the specified address is wrong, or there are firewall settings preventing access.

If the connection fails, to check whether each port can be reached from a node, you can use a tool such as `nmap` or `nc`. Here's an example of how to run a check with both tools in a successful case:[5]

```
$ 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/ .
$ nc -vv -z $HEAD_ADDRESS $PORT
Connection to compute04.berkeley.edu 6379 port [tcp/...]
succeeded!
```

If your node cannot access port and IP address specified, you might see:

```
$ 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
```

Now, if you want to stop the Ray processes on any node, simply run `ray stop`.

This wraps up the manual way of creating a Ray cluster. Let's move on to discussing deployment of your Ray clusters with the popular Kubernetes orchestration framework.

Deployment on Kubernetes

Kubernetes is an industry-standard platform for cluster resource management. It allows software teams to seamlessly deploy, manage and scale their business applications in a wide variety of production environments. It was initially developed by Google, but

many organizations have now adopted Kubernetes as their cluster resource management solution.

The community-maintained KubeRay project is the standard way of deploying and managing Ray clusters on Kubernetes. The KubeRay operator helps deploy and manage Ray clusters on top of Kubernetes. Clusters are defined as a custom `RayCluster` resource and managed by a fault-tolerant Ray controller. The operator automates provisioning, management, autoscaling and operations of Ray clusters deployed to Kubernetes. The main features of this operator are:

- Management of first-class `RayCluster` via a custom resource.
- Support for heterogeneous worker types in a single Ray cluster.
- Built-in monitoring via Prometheus.
- Use of `PodTemplate` to create Ray pods.
- Updated status based on the running pods.
- Automatically populate environment variables in the containers.
- Automatically prefix your container command with the ray start command.
- Automatically adding the volumeMount at /dev/shm for shared memory.
- Use of `ScaleStrategy` to remove specific nodes in specific groups.

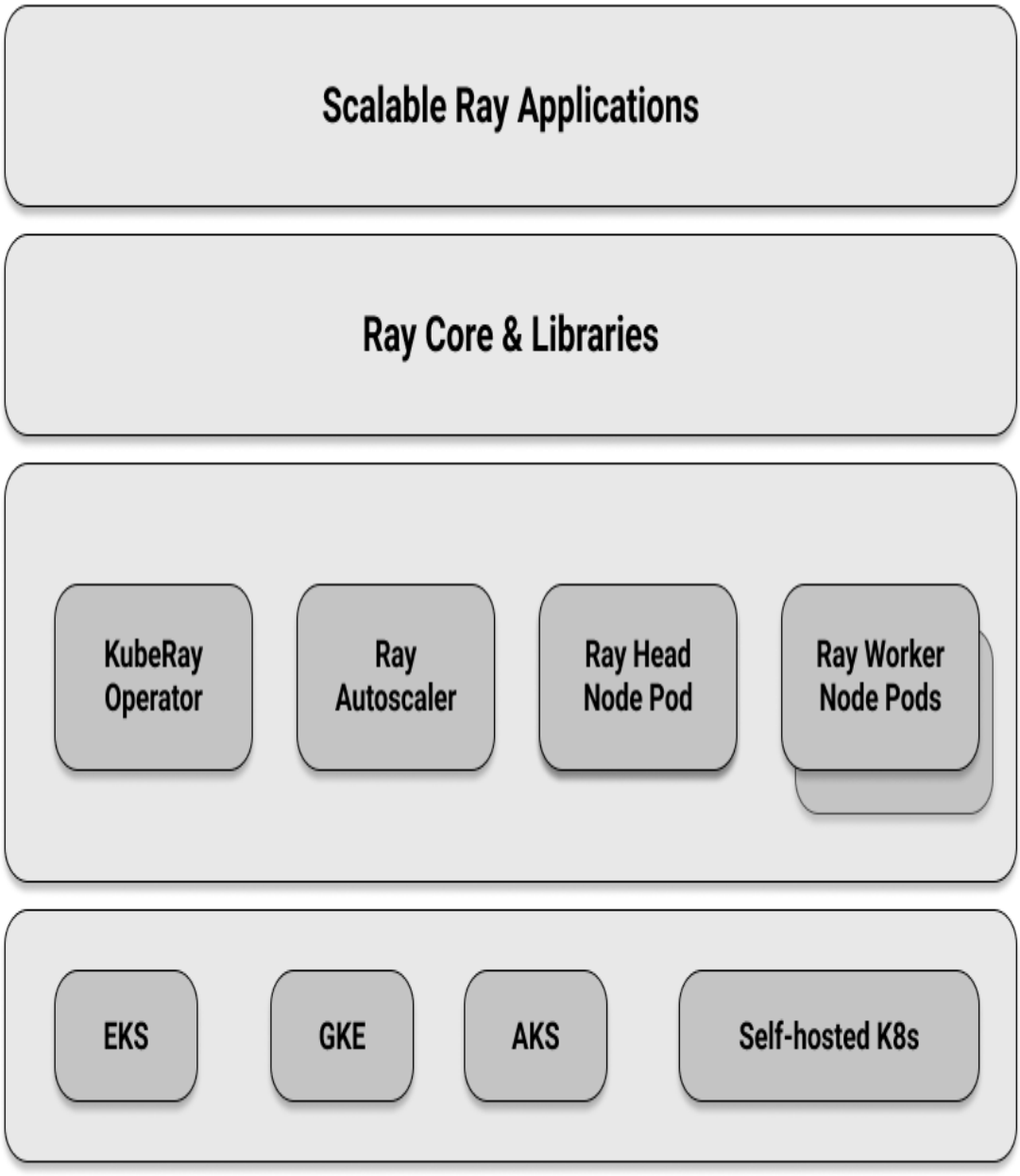


Figure 9-1. An Overview of KubeRay

Setting up Your First KubeRay Cluster

You can deploy the operator by cloning the KubeRay repository and calling the following command:

```
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"
```

You can verify that the operator has been deployed using

```
kubectl -n ray-system get pods
```

When deployed, the operator will watch for K8s events (create/delete/update) for the `raycluster` resource updates. Upon these events, the operator can create a cluster consisting of a head pod and multiple worker pods, delete a cluster, or update the cluster by adding or removing worker pods.

Now, let's deploy a new Ray cluster using a provided default cluster configuration. We'll cover how to configure this YAML file in a later section:

```
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
```

The KubeRay operator configures a Kubernetes service targeting the Ray head pod. To identify the service, run

```
kubectl get service --selector=ray.io/cluster=raycluster-
complete
```

The output of this command should resemble the following structure:

```
NAME                                TYPE        CLUSTER-IP
EXTERNAL-IP
raycluster-complete-head-svc   ClusterIP   xx.xx.xxx.xx
<none>
   PORT(S)                           AGE
   6379/TCP,8265/TCP,10001/TCP   6m10s
```

The three ports indicated in the output correspond to the following services of the Ray head pod:

- 6379: The Ray head's GCS service. Ray worker pods connect to this service when joining the cluster.
- 8265: Exposes the Ray Dashboard and the Ray Job Submission service.
- 10001: Exposes the Ray Client server.

You should note that the docker images we're using are quite large and can take a while to download. Also,even if you see the expected output from `kubectl get service`, that doesn't mean your cluster is ready to use yet. You should look at the pod status and make sure they are all actually in "Running" state.

Interacting with the KubeRay cluster

You might be wondering why we spend so much time on Kubernetes, since you're most likely just interested in learning how to run Ray scripts on it. That's understandable, and we'll get to that in a moment.

First, let's use this below Python script as the desired script to run on the cluster. We'll name it `script.py` (for simplicity), and the script will connect to the Ray Cluster and run a couple standard Ray commands:

```python
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)])
```

To run this script, there are three primary ways we will cover in the following sections, namely using `kubectl exec`, Ray Job Submission, or Ray Client.

Running Ray programs with `kubectl`

To start with, you can directly interact with the head pod via `kubectl exec`. Use the command below to get a Python interpreter on the head pod:

```
kubectl exec `kubectl get pods -o custom-
columns=POD:metadata.name |\
  grep raycluster-complete-head` -it -c ray-head -- python
```

With this python terminal, you can connect and run your own Ray application:

```
import ray
ray.init(address="auto")
...
```

There are other ways of interacting with these services without `kubectl`, but these will require some networking setup. The easiest route, and the one we'll take in what follows, is to use port-forwarding.

Using the Ray Job Submission Server

You can run scripts on the cluster by using the Ray Job Submission server. You can use the server to send a script or a bundle of

dependencies, and run custom scripts with that set of dependencies. To start, you'll need to port-forward the job submission server port:

```
kubectl port-forward service/raycluster-complete-head-svc
8265:8265
```

Now, submit the script by setting the `RAY_ADDRESS` variable to the job server s ubmission endpoint and using the Ray Job Submission CLI.

```
export RAY_ADDRESS="http://localhost:8265"

ray job submit --working-dir=. -- python script.py
```

You'll see an output that looks like:

```
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_U5hfr1rqJZWwJmLP' submitted successfully
-------------------------------------------------------

Next steps
  Query the logs of the job:
    ray job logs raysubmit_U5hfr1rqJZWwJmLP
  Query the status of the job:
    ray job status raysubmit_U5hfr1rqJZWwJmLP
  Request the job to be stopped:
    ray job stop raysubmit_U5hfr1rqJZWwJmLP

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_U5hfr1rqJZWwJmLP' succeeded
----------------------------------------
```

You can use `--no-wait` to run the job in the background.

Ray Client

To connect to the cluster via Ray Client from your local machine, first make sure the local Ray installation and Python minor version match the Ray and Python versions running in the Ray cluster. To do that, you can run `ray --version` and python --version on both instances. In practice, you will be using a container, in which case you can simply make sure you run everything in the same container. Also, if the versions don't match, you will see a warning message informing you of the issue.

Next, run the following command:

```
kubectl port-forward service/raycluster-complete-head-svc
10001:10001
```

This command will block. The local port `10001` will now be forwarded to the Ray head's Ray Client server.

To run a Ray workload on your remote Ray cluster, open a local Python shell and start a Ray Client connection:

```
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)])
```

With this method, you can just run the Ray program directly on your laptop (instead of needing to ship the code over via `kubectl` or job submission).

Exposing KubeRay

In the above examples, we used port-forwarding as a simple way access the Ray head's services. For production use cases, you may want to consider other means of exposing these services. The following notes are generic to services running on Kubernetes. Refer to the Kubernetes documentation for more detailed info.

By default, the Ray service is accessible from anywhere *within* the Kubernetes cluster where the Ray Operator is running. For example to use the Ray Client from a pod in the same Kubernetes namespace as the Ray Cluster, use `ray.init("ray://raycluster-complete-head-svc:10001")`.

To connect from another Kubernetes namespace, use `ray.init("ray://raycluster-complete-head-svc.default.svc.cluster.local:10001")`. (If the Ray cluster is a non-default namespace, use the namespace in place of `default`.)

If you are trying to access the service from outside the cluster, use an ingress controller. Any standard ingress controller should work with Ray Client and Ray Dashboard. Pick a solution compatible with your networking and security requirements — further guidance is out of scope for this book.

Configuring KubeRay

Let's take a closer look at the configuration for a Ray cluster running on K8s. The example file `kuberay/ray-operator/config/samples/ray-cluster.complete.yaml` is

a good reference. Here is a condensed view of a RayCluster config's most salient features:

```
apiVersion: ray.io/v1alpha1
kind: RayCluster
metadata:
  name: raycluster-complete
spec:
  headGroupSpec:
    rayStartParams:
      port: '6379'
      num-cpus: '1'
      ...
    template: # Pod template
        metadata: # Pod metadata
        spec: # Pod spec
            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: # Pod template
        ...
  - groupName: medium-group
    ...
```

Below, we cover some of the primary configuration values that you may use.

headGroupSpec and workerGroupSpecs: A Ray cluster consists of a head pod and a number of worker pods. The head pod's configuration is specified under `headGroupSpec`. Configuration for worker pods is specified under `workerGroupSpecs`. There may be multiple worker groups, each group with its own configuration `template`. The `replicas` field of a `workerGroup` specifies the number of worker pods of each group to keep in the cluster.

rayStartParams: This is a string-string map of arguments to the Ray pod's `ray start` entrypoint. For the full list of arguments, refer to the documentation for `ray start`. We make special note of the `num-cpus` and `num-gpus` field arguments:

- `num-cpus`: This field tells the Ray scheduler how many CPUs are available to the Ray pod. The CPU count can be auto-detected from the Kubernetes resource limits specified in the group spec's pod `template`. It is sometimes useful to override this auto-detected value. For example, setting `num-cpus:"0"` will prevent Ray workloads with non-zero CPU requirements from being scheduled on the head node.
- `num-gpus`: This specifies the number of GPUs available to the Ray pod. At the time of writing, this field is *not* detected from the group spec's pod `template`. Thus, `num-gpus` must be set explicitly for GPU workloads.

template: This is where the bulk of the `headGroup` or `workerGroup`'s configuration goes. The `template` is a Kubernetes Pod template which determines the configuration for the pods in the group. Here are some of the fields to pay attention to: - **resources**; It's important to specify container CPU and memory requests and limits for each group spec. For GPU workloads, you may also wish to specify GPU limits e.g. `nvidia.com/gpu: 1` if using an nvidia GPU device plugin.

> **WARNING**
>
> It is ideal, when possible, to size each Ray pod such that it takes up the entire Kubernetes node on which it is scheduled. In other words, it's best to run one large Ray pod per Kubernetes node; running multiple Ray pods on one Kubernetes node introduces unnecessary overhead. However, there are situations in which running multiple Ray pods on one K8s node makes sense:
>
> - Many users are running Ray clusters on a Kubernetes cluster with limited compute resources.
> - You or your organization are not directly managing Kubernetes nodes (e.g. when deploying on GKE Autopilot).

nodeSelector and tolerations: You can control the scheduling of worker group's Ray pods by setting the `nodeSelector` and `tolerations` fields of the pod spec. Specifically, these fields determine on which Kubernetes nodes the pods may be scheduled. Note that the KubeRay operator operates at the level of pods — KubeRay is agnostic to the setup of the underlying Kubernetes nodes. Kubernetes node configuration is left to your Kubernetes cluster's admins.

Ray container images: It's important to specify the images used by your cluster's Ray containers. The head and workers of the clusters should all use the same Ray version. In most cases, it makes sense

to the exact same container image for the head and all workers of a given Ray cluster. To specify custom dependencies for your cluster, it's recommended to build an image based on one of the official `rayproject/ray` images.

Volume mounts: Volume mounts can be used to preserve logs or other application data originating in your Ray containers. (See the logging section below.)

Container environment variables: Container environment variables may be used to modify Ray's behavior. For example, `RAY_LOG_TO_STDERR` will redirect logs to STDERR rather than writing them to container's file system. (See logging section below.)

Configuring Logging for KubeRay

Ray cluster processes typically write logs to the directory `/tmp/ray/session_latest/logs` in the pod. These logs are also viewable in the Ray Dashboard.

To persist Ray logs beyond the lifetime of a pod, you may use one of the following techniques.

Aggregate logs from the container's file system: For this strategy, mount an empty-dir volume with mountPath `/tmp/ray/` in the Ray container (see the example configuration above.) You can mount the log volume into a sidecar container running a log aggregation tool such as Promtail.

Container STDERR logging: An alternative is to redirect logging to STDERR. To do this set the environment variable `RAY_LOG_TO_STDERR=1` on all Ray containers. In terms of K8s configuration, that means adding an entry to the `env` field of the Ray container in each Ray `groupSpec`.

```
env:
    ...
    - name: "RAY_LOG_TO_STDERR"
```

```
      value: "1"
...
```

You may then use a Kubernetes logging tool geared towards aggregation from the `STDERR` and `STDOUT` streams.

Using the Ray Cluster Launcher

The goal of the cluster launcher is to make it easy to deploy a Ray cluster on any cloud. Here's what it will do for you:

- The cluster launcher provisions a new instance/machine using the cloud provider's SDK.
- It executes shell commands to set up Ray with the provided options.
- It optionally runs any custom, user defined setup commands. This can be useful for setting environment variables and installing packages.[6]
- It initializes the Ray cluster for you.
- And it deploys an autoscaler process.

We will walk you through the details of autoscaling in a later section. For now, let's focus on using the cluster launcher to deploy a Ray cluster. To do so, you need to provide a cluster configuration file.

Configuring Your Ray Cluster

To run your Ray cluster, you must specify the resource requirements in a cluster configuration file.

Here's our "scaffold" cluster specification. This is just about the minimum that you'll need to specify to launch your cluster. For more information about cluster yaml files, see a large example here (let's call it `cluster.yaml`).

```
# An unique identifier for the head node and workers of
this cluster.
cluster_name: minimal

# The maximum number of workers nodes to launch in addition
to the head
# node. min_workers default to 0.
max_workers: 1

# Cloud-provider specific configuration.
provider:
    type: aws
    region: us-west-2
    availability_zone: us-west-2a

# How Ray will authenticate with newly launched nodes.
auth:
    ssh_user: ubuntu
```

Using the Cluster Launcher CLI

Now that you have a cluster configuration file, you can use the Cluster Launcher CLI to deploy the specified cluster:

```
ray up cluster.yaml
```

This single line of code will take care of everything done via the manual cluster setup. It will interact with the cloud provider to provision the head node, and start the appropriate ray services or processes on that node.

This single line of code will not automatically start all specified nodes. In fact, it will only start a single “head” node, and run `ray start --head ...` on that head node. A Ray autoscaling process will then use the provided cluster configuration to start the worker nodes as a background thread after the head node has started.

Interacting with a Ray Cluster

After you start a cluster, you'll often want to interact with it through a variety of actions: * Run a script on the cluster * Move files, logs, and artifacts off the cluster * SSH onto the nodes to inspect machine details

There is a CLI for interacting with clusters launched by the Ray Cluster Launcher. If you have an existing script (like a `script.py`, from above), you can run the script on the cluster via `ray job submit` after you've port-forwarded the Job Submission endpoint:

```
# Run in one terminal:
ray attach cluster.yaml -p 8265

# Run in a separate terminal:
export RAY_ADDRESS=http://localhost:8265
ray job submit --working-dir=. -- python script.py
```

`--working-dir` will move your local files onto the cluster, and `python train.py` will be run on a shell on the cluster.

Let's say after you run this script, you generate a couple artifacts, like a `results.log` file that you want to inspect after. Use `ray rsync-down` to move the file back:

```
ray rsync-down cluster.yaml /path/on/cluster/results.log
./results.log
```

Working with Cloud Clusters

In this section, we will show you how to deploy Ray clusters on AWS, and other cloud providers.

AWS

First, install boto (`pip install boto3`) and configure your AWS credentials in `$HOME/.aws/credentials`, as described in the boto docs.

Once boto is configured to manage resources on your AWS account, you should be ready to launch your cluster. The provided ray/python/ray/autoscaler/aws/example-full.yaml cluster config file will create a small cluster with an m5.large head node (on-demand) configured to auto-scale up to two m5.large spot instance workers.

Test that it works by running the following commands from your local machine:

```
# Create or update the cluster. When the command finishes,
it will print
# out the command that can be used to SSH into the cluster
head node.
$ ray up ray/python/ray/autoscaler/aws/example-full.yaml

# Get a remote screen on the head node.
$ ray attach ray/python/ray/autoscaler/aws/example-
full.yaml
$ # Try running a Ray program with
'ray.init(address="auto")'.

# Tear down the cluster.
$ ray down ray/python/ray/autoscaler/aws/example-full.yaml
```

Using Other Cloud Providers

Ray Clusters can be deployed on most major clouds, including GCP, and Azure. Here are is a template to get started for Google Cloud:

```
# A unique identifier for the head node and workers of this
cluster.
cluster_name: minimal

# The maximum number of worker nodes to launch in addition
to the head
# node. min_workers default to 0.
max_workers: 1

# Cloud-provider specific configuration.
provider:
    type: gcp
```

```
    region: us-west1
    availability_zone: us-west1-a
    project_id: null # Globally unique project id

# How Ray will authenticate with newly launched nodes.
auth:
    ssh_user: ubuntu
```

Here is a template to get started for Azure:

```
# An unique identifier for the head node and workers of
this cluster.
cluster_name: minimal

# The maximum number of workers nodes to launch in addition
to the head
# node. min_workers default to 0.
max_workers: 1

# Cloud-provider specific configuration.
provider:
    type: azure
    location: westus2
    resource_group: ray-cluster

# How Ray will authenticate with newly launched nodes.
auth:
    ssh_user: ubuntu
    # you must specify paths to matching private and public
key pair files
    # use `ssh-keygen -t rsa -b 4096` to generate a new ssh
key pair
    ssh_private_key: ~/.ssh/id_rsa
    # changes to this should match what is specified in
file_mounts
    ssh_public_key: ~/.ssh/id_rsa.pub
```

You can read more about this on the Ray documentation.

Autoscaling

Ray is designed to support highly elastic workloads which are most efficient on an autoscaling cluster. At a high level, the autoscaler attempts to launch and terminate nodes in order to ensure that workloads have sufficient resources to run, while minimizing the idle resources. It does this by taking into consideration:

- User specified hard limits (min/max workers).
- User specified node types (nodes in a Ray cluster do *not* have to be homogenous).
- Information from the Ray core's scheduling layer about the current resource usage/demands of the cluster.
- Programmatic autoscaling hints.

The autoscaler resource demand scheduler will look at the pending tasks, actors, and placement groups resource demands from the cluster. It will then try to add the minimum list of nodes that can fulfill these demands.

When worker nodes are idle for more than `idle_timeout_minutes`, they will be removed. The head node is never removed unless the cluster is torn down.

The autoscaler uses a simple *binpacking algorithm* to pack the user demands into the available cluster resources. The remaining unfulfilled demands are placed on the smallest list of nodes that satisfies the demand while maximizing utilization (starting from the smallest node). You can learn more about the autoscaling algorithm in the Autoscaling section of the Ray architecture whitepaper.

Ray also provides documentation and tooling for other cluster managers such as YARN, SLURM and LFS. You can read more about this on the Ray documentation.

Summary

In this chapter you learned how to spin up your own Ray Clusters, so that you can deploy your Ray applications on them. Besides manually setting up and shutting down clusters, we had a closer look at deploying Ray Clusters on Kubernetes using KubeRay. We also looked at the Ray Cluster Launcher in detail and discussed how to work with Cloud Clusters on clouds such as AWS, GCP, or Azure. Finally, we discussed how to use the Ray Autoscaler to scale your Ray Clusters.

Now that you know more about scaling Ray Clusters, in the next chapter we'll come back to the application side of things, and discuss how all previously discussed Ray ML libraries effectively come together to form the Ray AI Runtime.

---

1 Ray itself is not opinionated about how you set up your cluster. In fact, you have many options, many of which are described in this chapter. Apart from the open source solutions we're describing here, there are also fully managed commercial solutions available, such as offered by Anyscale or Domino Data Lab.

2 While this chapter technically has an accompanying notebook as well, the material presented in this chapter is not well-suited for development in an interactive Python session. We recommend that you work through the examples of this chapter from the command line. No matter where you decide to work, please make sure to have Ray install, e.g. with `pip install "ray==2.2.0".

3 Depending on your setup or work situation, this might not be a realistic assumption for you. Don't worry, we're going to cover other ways of creating Ray clusters later in this chapter that don't require you to have any machines running. In any case, knowing the steps involved for manual Ray cluster creation is useful to know and fall back on.

4 If you already have remote Redis instances, you can use them by specifying the environment variable `RAY_REDIS_ADDRESS=ip1:port1,ip2:port2...`. Ray will use the first address as primary and the rest as shards.

5 If you want to learn more about the specific flags used in the examples below, we suggest to have a look at the official reference guide.

6 To dynamically set up environments after the cluster has been deployed, you can use a runtime environment.

Chapter 10. Getting Started with the Ray AI Runtime

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

We've come a long way since you've heard about the Ray AI Runtime (AIR) for the first time in Chapter 1. Besides the fundamentals of Ray Clusters and the basics of the Ray Core API, you've picked up a good understanding of all higher level libraries of Ray that can be leveraged in AI workloads, namely Ray RLlib, Tune, Train, Datasets and Serve in the chapters leading up to this one. The main reason we deferred to a deeper discussion of Ray AIR until now is that it's so much easier to reason about its concepts and compute complex examples if you know its building blocks.

In this chapter we're going to introduce you to the core concepts of Ray AIR and how you can use it to build and deploy common workflows. We'll build an AIR application together that leverages many of Ray's data science libraries that you already know about. We will also tell you when and why to use AIR, and give you a brief overview of the technical underpinnings of it. An in-depth discussion of the relationship of AIR with other systems, such as integrations and key differences, will only be tackled in Chapter 11 when we talk about Ray's ecosystem as it relates to AIR.

Why Use AIR?

Running ML workloads with Ray has been a constant evolution over the last couple of years. Ray RLlib and Tune were the first libraries that were built on top of Ray Core. Components like Ray Train, Serve or more recently Datasets, followed shortly after. The addition of Ray AIR as an umbrella for all other Ray ML libraries is the result of active discussions with and feedback from the ML community. Ray, as a Python-native tool with good GPU support and stateful primitives (Ray Actors) for complex ML workloads, is a natural candidate for building a runtime like AIR.

Ray AIR is a unified toolkit for your ML workloads that offers many third party integrations for model training or accessing custom data sources. In the spirit of the other ML libraries built on top of Ray Core, AIR hides lower-level abstractions and provides an intuitive API that was inspired by common patterns from tools such as scikit-learn.

At its core, Ray AIR was built for both data scientists and ML engineers alike. As a data scientist, you can use it to build and scale your end-to-end experiments, or individual subtasks such as preprocessing, training, tuning, scoring or serving of ML models. As ML engineer, you can go as far as to build a custom ML platform on top of AIR, or simply leverage its unified API to integrate it with other libraries from your ecosystem. And Ray always gives you the flexibility to drop down and delve into the lower-level Ray Core API, in case you need to for your use case.

As part of the Ray ecosystem, AIR can leverage all its benefits, which includes a seamless transition from experimentation on a laptop to production workflows on a cluster. You often see data science teams “hand over” their ML code to teams responsible for production systems. In practice this can be expensive and time-consuming, as this process often involves modifying or even rewriting parts of the code. As we will see, Ray AIR helps you with

this transition, as concerns such as scalability, reliability, or robustness are taken care of for you by AIR.

Ray AIR already has a respectable number of integrations today, but it's also fully extensible. And as we will show you in the next section, its unified API provides a smooth workflow that allows you to drop-in-replace many of its components. For instance, you can use the same interface to define an XGBoost or PyTorch Trainer with AIR, which makes experimentation with various ML models convenient.

At the same time, by choosing AIR you can avoid the problem of working with several (distributed) systems and writing glue code for them that's difficult to deal with. Teams working with many moving parts often experience rapid deprecation of integrations and a high maintenance burden. These issues can lead to *migration fatigue*, a reluctance to adopt new ideas due to the anticipated complexity of system changes.

As with every chapter, you can follow the code examples in the accompanying Jupyter notebook.

Key AIR Concepts by Example

One key to AIR's design philosophy is to provide you with the ability to *tackle your ML workloads in a single script, run by a single system*. Let's introduce AIR and its critical concepts by walking through an extended usage example. Here's what we're going to do:

- We load the breast cancer data set that you've already seen in Chapter 7 as a Ray Dataset and use AIR to preprocess it.
- We define an XGBoost model for training a classifier on this data.
- We set up a so-called Tuner for our training procedure to tune its hyperparameters.
- We store checkpoints of trained models.

- We run batch prediction using AIR.
- Finally, we deploy our predictor as a service with AIR.

You tackle these steps by building scalable pipelines with the AIR API. To follow along this example, make sure to install the following requirements:

```
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"
```

Here’s a summary of the steps we’re going to take in the following example, alongside the AIR components we’re going to use:

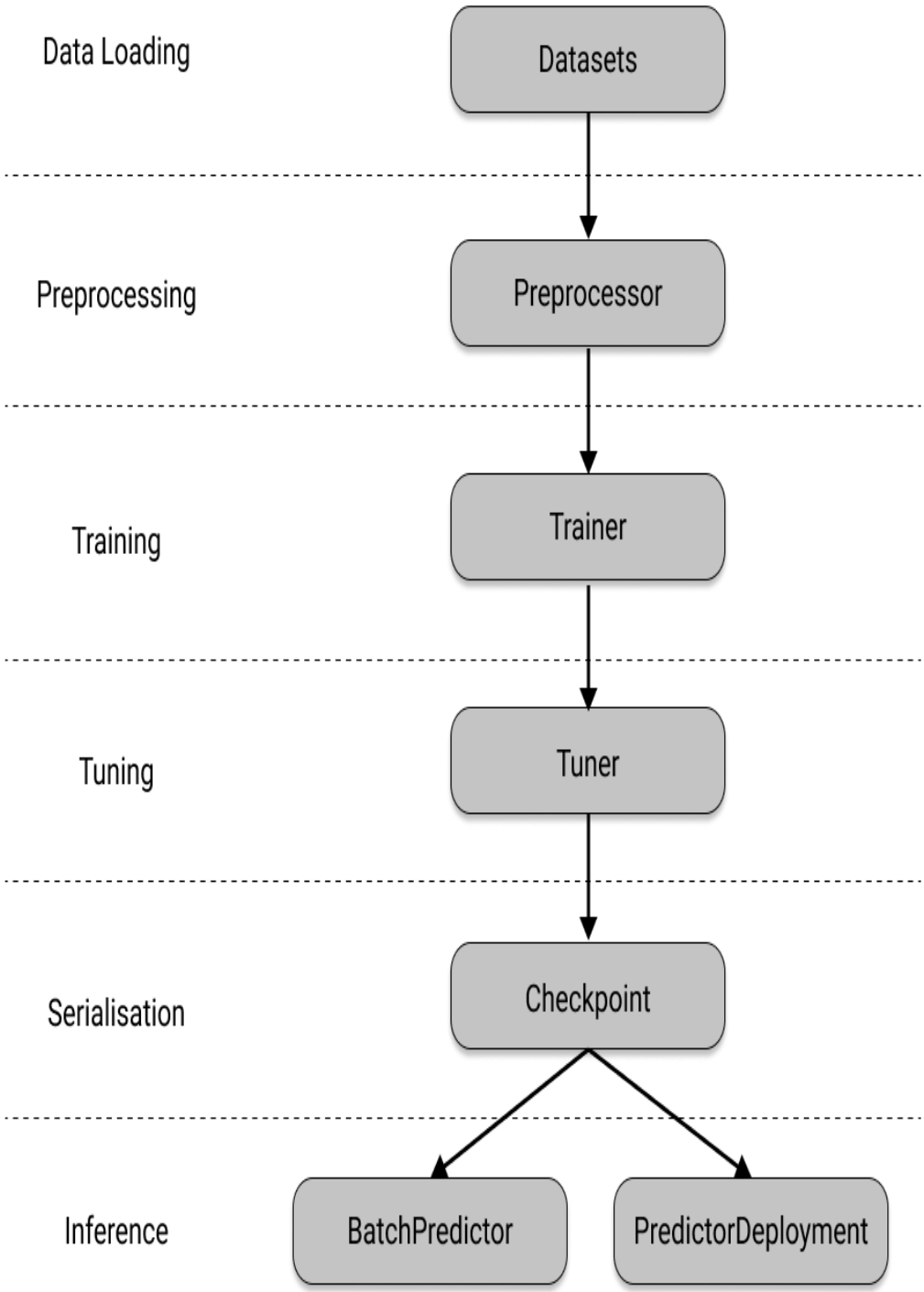

Data Loading
Datasets
Preprocessing
Preprocessor
Training
Trainer
Tuning
Tuner
Serialisation
Checkpoint
Inference
BatchPredictor
PredictorDeployment

Figure 10-1. Going rom data loading to inference with AIR as the single distributed system.

Datasets and Preprocessors

The standard way for you to load data in Ray AIR is with Ray Datasets. AIR *Preprocessors* are used to transform input data into features for ML experiments. We've already briefly touched on preprocessors in Chapter 7, but have not discussed them in the context of AIR yet.

Since these Preprocessors operate on Datasets and leverage the Ray ecosystem, they allow you to scale your preprocessing steps efficiently. During training an AIR Preprocessor is *fitted* to the specified training data and can then later be used for both training and serving.[1] AIR comes packaged with many common Preprocessors that cover many use cases. If you don't find the one you need, you can easily define a custom Preprocessor on your own.

In our example, we want to read a CSV file from an S3 bucket into a columnar Ray Dataset first, using the `read_csv` utility. Then we split our dataset into a training and a test dataset and define an AIR Preprocessor, namely `StandardScaler`, which normalizes all specified columns of our dataset to have mean of `0` and a variance of `1`. Note that just specifying a Preprocessor does not transform the data just yet. Here is how you implement what we just discussed:

```
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 Loading the breast cancer CSV file from S3 using Ray Datasets.
❷ After defining a training and a test dataset we drop the `target` column on the test data.
❸ Defining an AIR preprocessor to scale two variables of the `dataset` to be normally distributed.

Note that for simplicity we're using the test dataset for as a validation dataset in future training as well, hence the naming convention.

Before moving on to the training step in AIR workflows, let's have a look at the different types of AIR Preprocessors that are available to you. If you want to know more about all available Preprocessors, you can consult the user guide on this topic. In this book we're only using Preprocessors for feature scaling, but the other types of Preprocessors in AIR can be very useful as well:

Table 10-1. RayAIR Preprocessors

Preprocessor Type	Examples

Feature Scalers	MaxAbsScaler, MinMaxScaler, Normalizer, PowerTransformer, StandardScaler
Generic Preprocessors	BatchMapper, Chain, Concatenator, SimpleImputer
Categorical Encoders	Categorizer, LabelEncoder, OneHotEncoder
Text Encoders	Tokenizer, FeatureHasher

Trainers

Once you have your training and test Datasets ready and your Preprocessors defined, you can move on to specifying a *Trainer* that runs an ML algorithm on your data. Trainers from the Ray Train package have been introduced in Chapter 7 and provide a consistent wrapper for training frameworks such as TensorFlow, PyTorch, or XGBoost. In this example we'll focus on the latter, but it's important that all other framework integrations work exactly the same way in terms of the Ray AIR API.

Let's define a so-called `XGBoostTrainer`, one of the many specific `Trainer` implementations that come with Ray AIR. Defining such a trainer requires you to specify a couple of arguments, namely:

- an AIR `ScalingConfig` that describes how you want to scale out training on your Ray Cluster,
- a `label_column` that specifies which column of your dataset is used as *label* in supervised learning with XGBoost.
- a `datasets` argument with at least a `train` key and an optional `valid` key to specify the training and validation datasets, respectively,

- an AIR `preprocessor` to compute the features of your ML model,
- and framework specific parameters (e.g. the number of boosting rounds in XGBoost), as wells as a common set of parameters called `params`.

```
# NOTE: Colab does not have enough resources to run this
example.
# try using num_workers=1, resources_per_worker={"CPU": 1,
"GPU": 0} in your
# ScalingConfig below.
# In any case, this training loop will take considerable
time to run.
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 Every `Trainer` comes with a scaling configuration. Here we're using two Ray workers and no GPU.

❷ `XGBoostTrainers` need specific configuration, as well as a training `objective` and evaluation metrics to track.

❸ The `Trainer` specifies the Datasets it's supposed to operate on.[2]

❹ In the same way, you can provide AIR Preprocessors that the `Trainer` should use.

❺ After everything is defined, a simple `fit` call is enough to start the training procedure.

Trainers provide scalable ML training that operate on AIR Datasets and Preprocessors. On top of that, they're also built to integrate well with Ray Tune for hyperparameter optimization, as we'll see next.

To summarize this section, Figure 10-2 shows how AIR Trainers fit ML models on Ray Datasets given preprocessors and a scaling configuration.

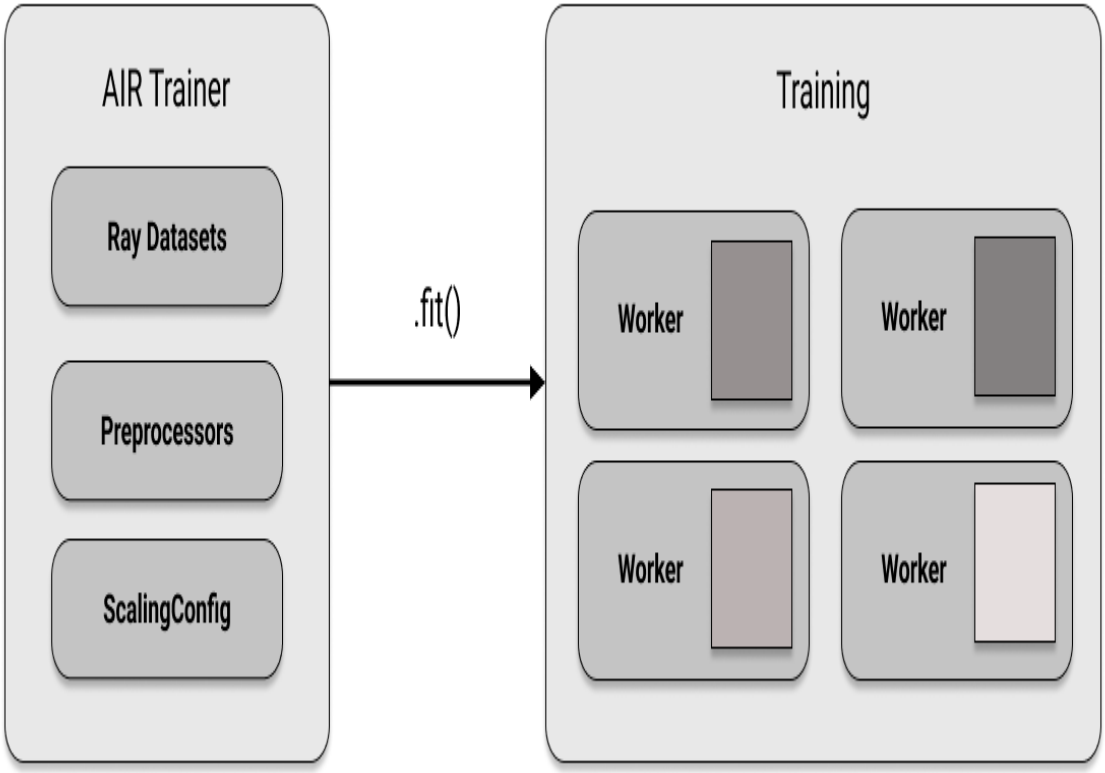


Figure 10-2. AIR Trainers operate on Ray Datasets and use AIR Preprocessors and scaling configurations.

Tuners and Checkpoints

Tuners, introduced with Ray 2.0 as part of AIR, offer scalable hyperparameter tuning through Ray Tune. Tuners work seamlessly with Trainers, but also support arbitrary training functions. In our example, instead of calling `fit()` on your `trainer` instance from the last section, you can also pass your `trainer` into a Tuner. To do so, a Tuner needs to be instantiated with a parameter space to search over, a so-called `TuneConfig`. This config has all Tune-specific configurations like the metric you want to optimize, and an optional `RunConfig` that lets you configure runtime-specific aspects such as the log verbosity of your Tune run.

Continuing with the `XGBoostTrainer` we defined earlier, here's how you wrap that `trainer` instance in a Tuner to calibrate the `max_depth` parameter of your XGBoost model.

```
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("Best Result:", best_result)
```

❶ You initialize your Tuner with a Trainer instance which in turn specifies the scaling configuration of the run.

❷ Your Tuner also takes a `param_space` to search over.

❸ It also needs a dedicated `TuneConfig` to tell Tune how to optimize your Trainer, given its parameter space.
❹ Tuner runs are started the same way as Trainers, namely by calling `.fit()`.

Whenever you run AIR Trainers or Tuners, they generate framework-specific *Checkpoints*. You can use these checkpoints to load models for usage across several AIR libraries, such as Tune, Train or Serve. You can get a Checkpoint by accessing the result of a `.fit()` call on either a Trainer or Tuner. In our example, that means you can simply access a `checkpoint` on the `best_result` object, or any other entry from the `result_grid` like this:

```
checkpoint = best_result.checkpoint
print(checkpoint)
```

The other main way to work with checkpoints is by creating one from an existing, framework-specific model. Every ML framework supported by AIR can be used to do this, but since it's easiest to define a simple model with it, we're going to show you how this looks like for a sequential TensorFlow Keras model:

```
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)
```

Having checkpoints is great because they're AIR's native model exchange format. You can also use them to pick up trained models at a later stage, without having to worry about custom ways to store

and load the models in question. Figure Figure 10-3 schematically shows how AIR Tuners work with AIR Trainers.

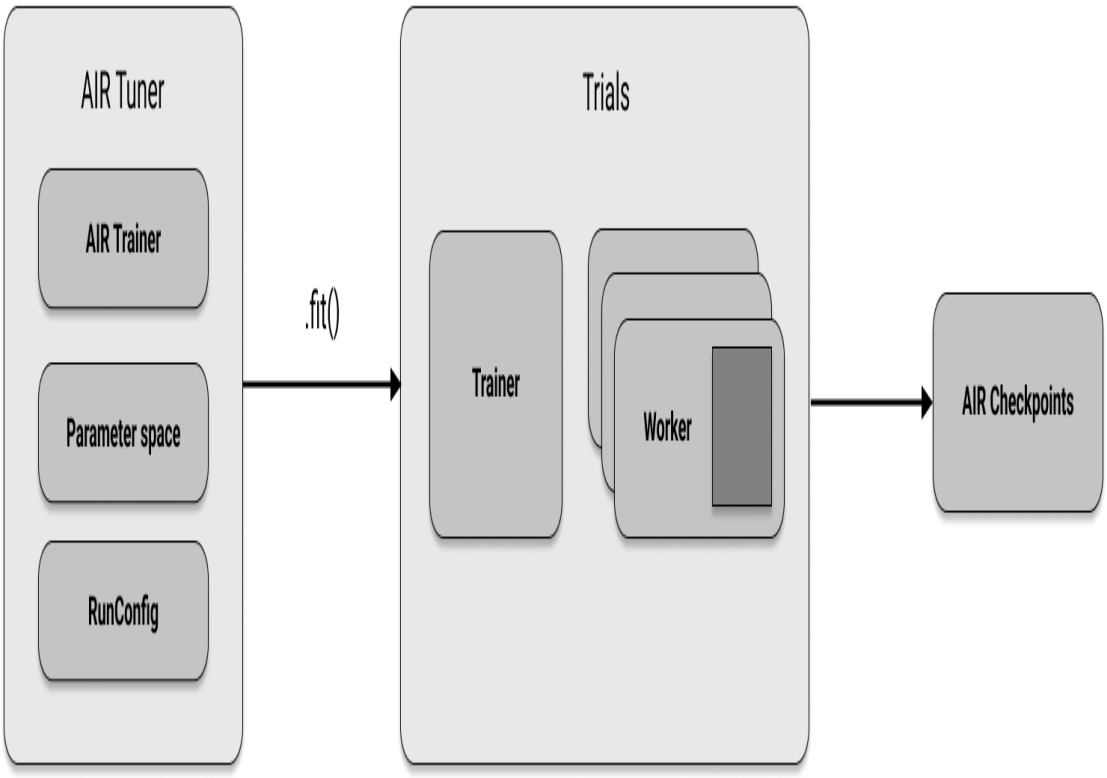


Figure 10-3. An AIR `Tuner` calibrates the hyperparameters of AIR Trainers. They need a parameter space to search over and an optional `RunConfig` to configure runtime-specific aspects.

Batch Predictors

Once you have a trained model through AIR, that is by fitting either a `Trainer` or a `Tuner`, you can use the resulting AIR Checkpoint for prediction on batches of data in Python. To do that you create a `BatchPredictor` from your Checkpoint and then use its `predict` method on your Dataset. In our case we need to use the framework specific class of the predictor, namely `XGBoostPredictor`, to tell AIR how to load the `checkpoint` correctly.

```
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()
```

❶ We first load an XGBoost model from a checkpoint into a BatchPredictor object.

❷ Then we run batch inference on our test dataset to get predicted probabilities.

This can be visualised in the following figure:

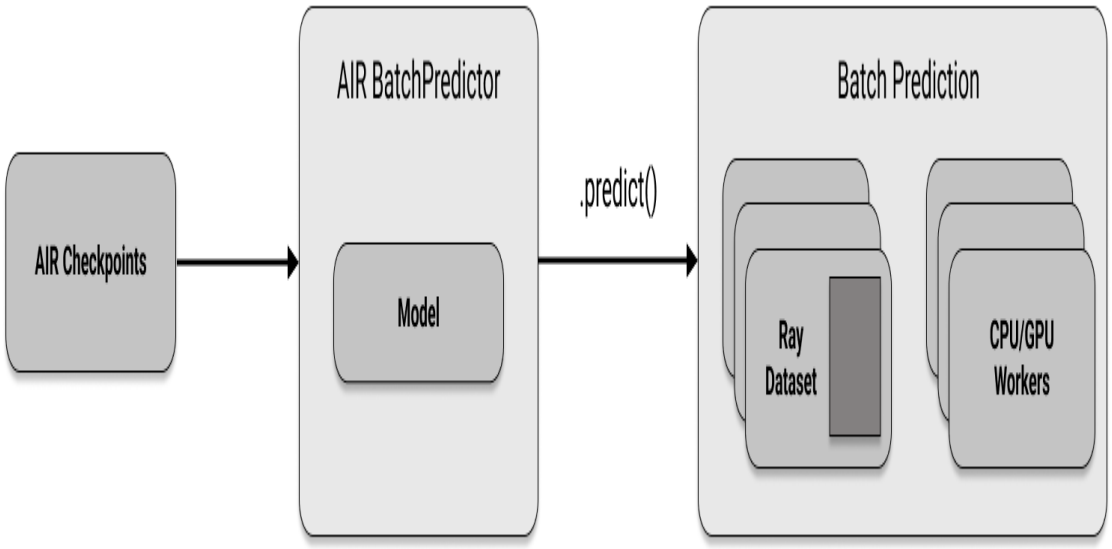


Figure 10-4. Using AIR BatchPredictors from AIR Checkpoints to run batch inference on AIR Datasets.

Deployments

Instead of using a `BatchPredictor` and interacting with the model in question directly, you can also leverage Ray Serve to deploy an inference service that you can query over HTTP. You do that by using the `PredictorDeployment` class and `deploy` it using our `checkpoint`. The only slightly tricky part about this is that our model operates on dataframes, which we can't directly send over HTTP. That means we need to explicitly tell our prediction service how to pick up and transform the *payload* we define and create a dataframe from it. We do this by specifying an *adapter* for our deployment.[3]

```
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(  ❸
    XGBoostPredictor,
    checkpoint,
    http_adapter=adapter
)

print(deployment.url)
```

❶ An adapter takes an HTTP request object and returns data in a format accepted by our model.

❷ After starting Serve we can create a deployment for our model

❸ To actually `deploy` the deployment object we need to pass in the model `checkpoint`, the `adapter` function, and the `XGBoostPredictor` class for correct model loading.

To test this deployment, let's create some sample input from our test data that we can throw at our service. For simplicity, we take the first item of our test dataset and convert it to a Python dictionary, so that the ubiquitous `requests` library can `post` a request to our deployment URL with it:

```
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 We can post our `sample_input` to the `deployment.url`, for instance by using `requests`.

❷ After you're done using the service, you can safely shut down Ray Serve.

Figure Figure 10-5 summarises how AIR deployments with the `PredictorDeployment` class work.

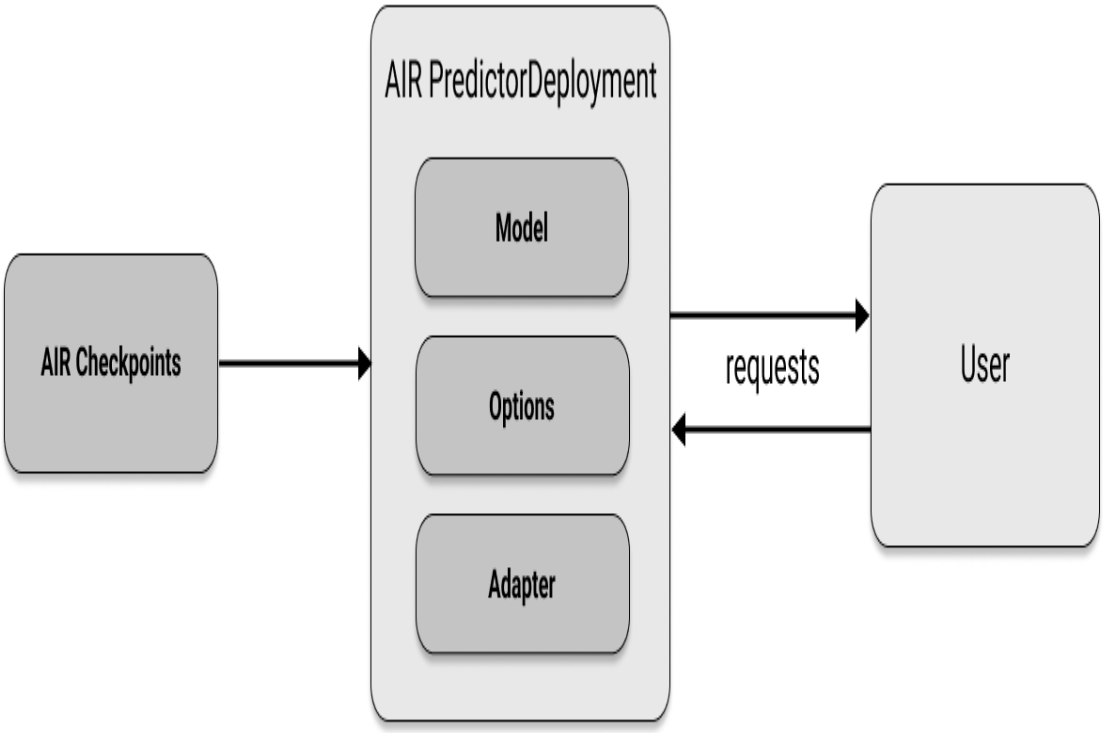


Figure 10-5. Creating PredictorDeployments from AIR Checkpoints to serve models that users can interact with.

In summary, figure Figure 10-6 gives you a quick visual overview of all components and concepts involved in Ray AIR, including pseudocode for all main AIR components we've covered in this chapter:

Key Ray AIR Concepts

Datasets & Preprocessors

```
dataset = ray.data.read_datasource(...)
train_data, test_data = ...
preprocessor = StandardScaler(...)
```

Trainers

```
trainer = ray.train.xgboost.XGBoostTrainer(
  scaling_config=ScalingConfig(...),
  ...,
  datasets={"train": train_data, ...},
  preprocessor=preprocessor)
```

Tuners & Checkpoints

```
tuner = ray.tune.Tuner(
  trainer,
  param_space=...,
  tune_config=TuneConfig(...),
  run_config=RunConfig(...))
result = tuner.fit()
checkpoint = result.best_result().checkpoint
```

Batch Predictors

```
predictor = ray.train.batch_predictor.BatchPredictor.from_checkpoint(
  checkpoint,
  XGBoostPredictor)
predictions = predictor.predict(test_data)
```

Deployments

```
deployment = ray.serve.PredictorDeployment.options(
  name="XGBoost")
deployment.deploy(XGBoostPredictor, checkpoint, ...)
result = requests.post(deployment_url, ...)
```

Figure 10-6. AIR combines many Ray libraries by providing a unified API for common data science workloads.

It's important to stress again that we've been using *a single Python script for this example*, and used a single distributed system in Ray AIR to do all the heavy lifting. In fact, you can use this example script and scale it out to a large cluster that uses CPUs for preprocessing, GPUs for training, and separately configure the deployment simply by modifying the parameters of the scaling configuration and similar options in that script. This isn't as easy or common as it may look like, and it is not unusual for data scientists to have to use multiple frameworks to achieve this (e.g., one for data loading and processing, one for training, and one for serving).

> **NOTE**
>
> You can also use Ray AIR with RLlib, but the integration is still in its early stages. For instance, to integrate RLlib with AIR Trainers, you'd use the so-called `RLTrainer` that allows you to pass in all arguments that you'd pass to a standard RLlib algorithm. After training, you can store the resulting RL model in an AIR Checkpoint, just as with any other AIR Trainer. To deploy your trained RL model, you can use Serve's `PredictorDeployment` class, by passing in your checkpoint along with the `RLPredictor` class.
>
> This API might be subject to change, but you can see an example of how this works in the AIR documentation.

What Workloads Are Suited for AIR

Now that we've seen examples of AIR and its fundamental concepts, let's zoom out a little and discuss which kinds of workloads you can run with it in principle. We've tackled all of these workloads already throughout the book, but it's good to quickly recap them in a systematic way. As the name suggests, AIR is built to capture

common tasks in AI projects. These tasks can be roughly classified in the following way:

- *Stateless Computation*: Tasks like preprocessing data or computing model predictions on a batch of data are stateless.[4] Stateless workloads can be computed independently in parallel. If you recall our treatment of Ray Tasks from Chapter 2, stateless computation is exactly what they were built for. AIR primarily uses Ray Tasks for stateless workloads.[5] Many *big data processing* tools fall into this category.
- *Stateful Computation*: In contrast, model training or hyperparameter tuning are stateful operations, as they update the model state during their respective training procedure. Updating stateful workers in such *distributed training* is a difficult topic that Ray handles for you. AIR uses Ray Actors for stateful computations.
- *Composite Workloads*: Combining stateless and stateful computation, for instance by first processing features and then training a model, is quite common in AI workloads. In fact, it's rare for end-to-end projects to exclusively use one or the other. Running such advanced composite workloads in a distributed fashion can be described as *big data training*, and AIR is built to handle both the stateless and stateful parts efficiently.
- *Online Serving*: Lastly, AIR is built to handle scalable online serving of (multiple) models. The transition from the previous three workloads to serving is frictionless by design, as you still operate within the same AIR ecosystem.

Figure Figure 10-7 visualises the typical stateless tasks of Ray AIR.

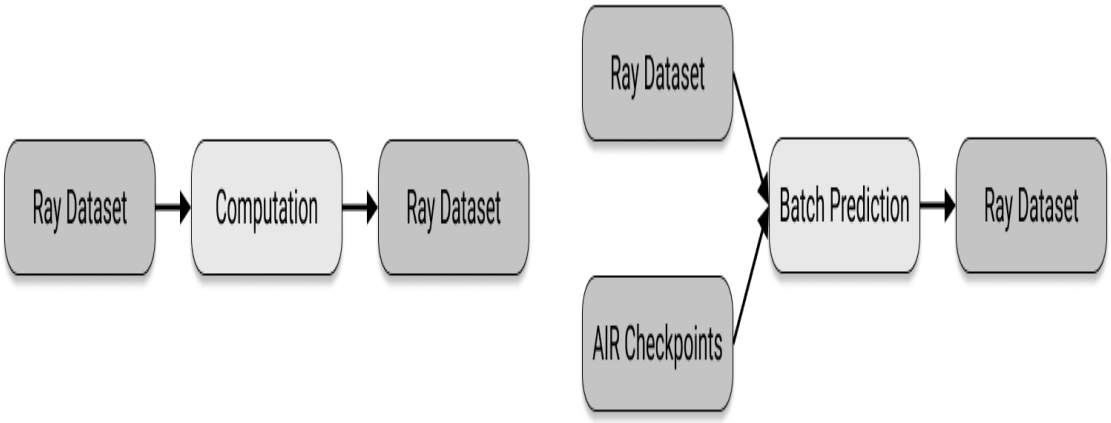


Figure 10-7. Stateless AIR tasks.

The above four tasks map to Ray's libraries in a straightforward way. For instance, in this chapter we've discussed several ways in which Ray Datasets are used for stateless computation. You could run batch inference on a given Dataset, by passing it into a `BatchPredictor` loaded from an AIR Checkpoint. Or you could preprocess a Dataset to produce features for later training.[6]

Likewise, there are three AIR libraries dedicated to stateful computations, namely Train, Tune, and RLlib. As we've seen, both Train and RLlib integrate seamlessly with Tune in AIR, namely by passing the respective `Trainer` objects into a `Tuner`.

When it comes to advanced composite workloads, Ray AIR and its combined usage of both Tasks and Actors can really shine. For instance, some ML training procedures need to perform complex data processing tasks during training. Others may require shuffling the training dataset before each epoch. Since Ray AIR's training libraries (based on Actors) can seamlessly leverage data processing operations (mostly based on Tasks), even the most complex use cases can be reflected in AIR.

Also, since you can use any AIR Checkpoint with Ray Serve, AIR makes it easy to switch from training to serving workloads, using the same infrastructure. We've seen in this chapter how you can use a `PredictorDeployment` to host a model behind an HTTP endpoint, which are optimized for low latency and high throughput. By deploying AIR, you can scale your prediction services to multiple replicas and use Ray's autoscaling features to adjust your cluster according to inbound traffic.[7]

You can use these types of workloads in different scenarios, too. For instance, you can use AIR to replace and scale out a single component of an existing pipeline. Or you can create your own end-to-end machine learning apps with AIR, as we've indicated in this chapter. Lastly, you can also use AIR to build your own AI platform, a topic that we'll have a look at in Chapter 11.

Figure Figure 10-8 summarizes how these four types of AI workloads are covered by AIR as part of the Ray ecosystem.

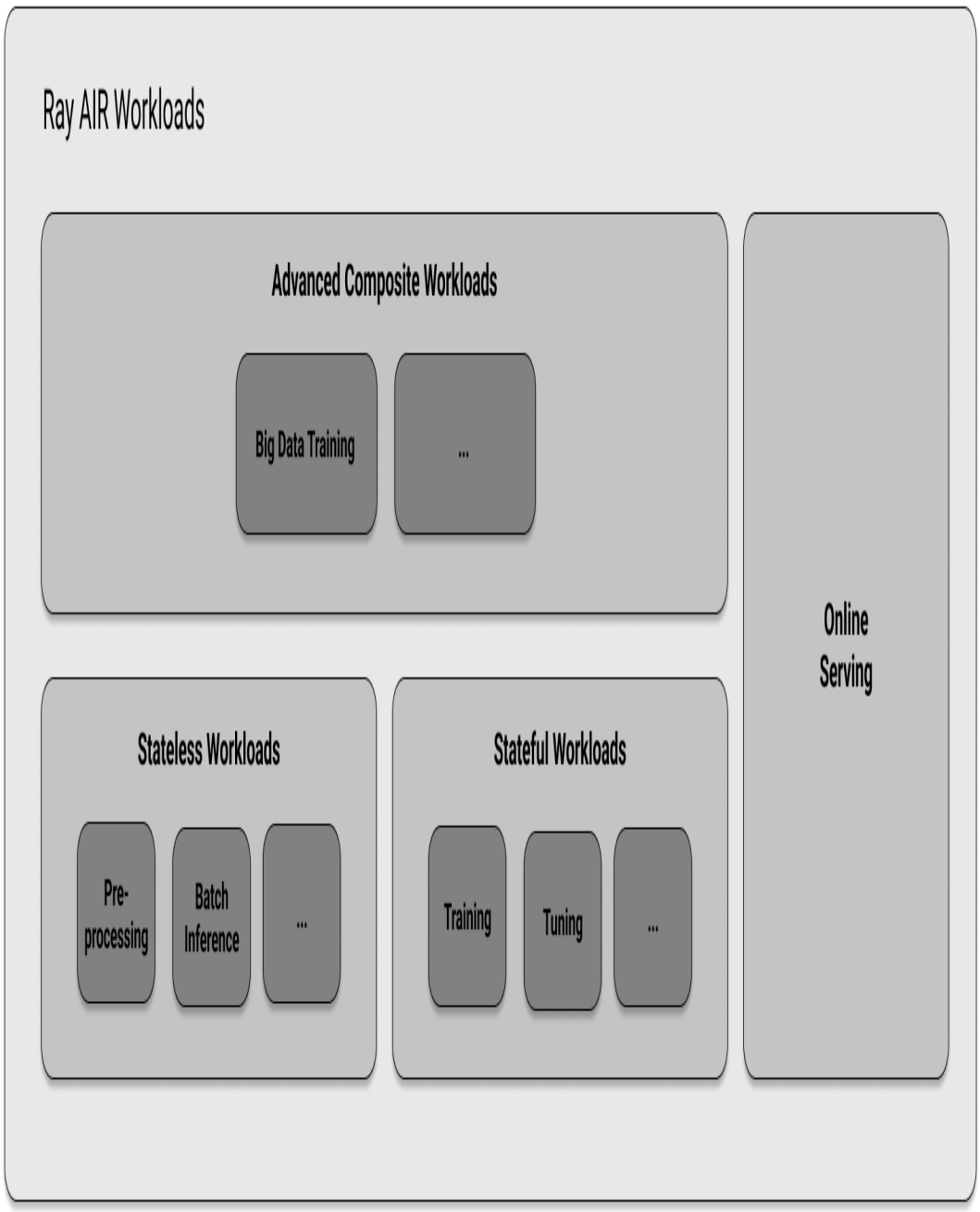


Figure 10-8. The four types of AI workloads AIR enables you to run on Ray Clusters.

Next, we're going to discuss several aspects of each of these four workload types of AIR in more detail. Specifically, we're going to investigate how Ray executes such workloads internally. Also, we're

going to dive a little deeper into technical aspects behind Ray AIR, such as how it manages memory or how it handles failures. We can only give you a brief overview of these topics, but will give you links to more advanced material as we go.

AIR Workload Execution

Let's have a closer look at the execution model of AIR first.

Stateless Execution

Ray Datasets use Ray Tasks or Actors to execute transformations. Tasks are preferred since they allow for easier and more flexible scheduling. The Datasets library uses a scheduling strategy that evenly balances Tasks and their output across your cluster.

Actors are used if a transformation has state or needs an expensive setup, like loading large model checkpoints. In this case, loading large models in an Actor once to reuse it for inference tasks can benefit overall performance. When Ray Datasets is using Actors, these actors are created first and the necessary data (in our example the loaded model) is transferred to them before the transformation in question gets executed.

In general, Datasets are stored in Ray object store memory, while large Datasets are spilled to disk. But for stateless transformations there is often no need to keep intermediate results in memory. Using *pipelining* on Datasets, as we've shown in Chapter 6, data can instead be streamed to and from storage to increase performance.[8] The idea is to only load the fraction of the data currently needed to execute a transformation. This can drastically reduce the memory footprint of the transformation and often speed up the overall execution.

Stateful Execution

Ray Train and RLlib spawn Actors for their distributed training workers. As demonstrated multiple times, both libraries also integrate seamlessly with Tune. In Chapter 5 we've discussed in detail how Tune launches Trials, which in essence are groups of actors executing a certain workload. If you use Train or RLlib with Tune, that in turn means that a tree of actors gets created, namely an actor for each Tune Trial, and sub-actors for the parallel training workers requested by Train or RLlib.

In terms of workload execution, this naturally creates an inner and an outer layer. Each sub-actor in a Tune Trial has full autonomy over its workload, as requested in the respective Train or RLlib training run you specified. This represents the inner execution layer. On the outer layer, Tune needs to monitor the status of individual trials, which it does this by periodically reporting metrics to the Trial driver.

Figure Figure 10-9 illustrates this nested creation and execution of Ray Actors for this scenario explicitly.

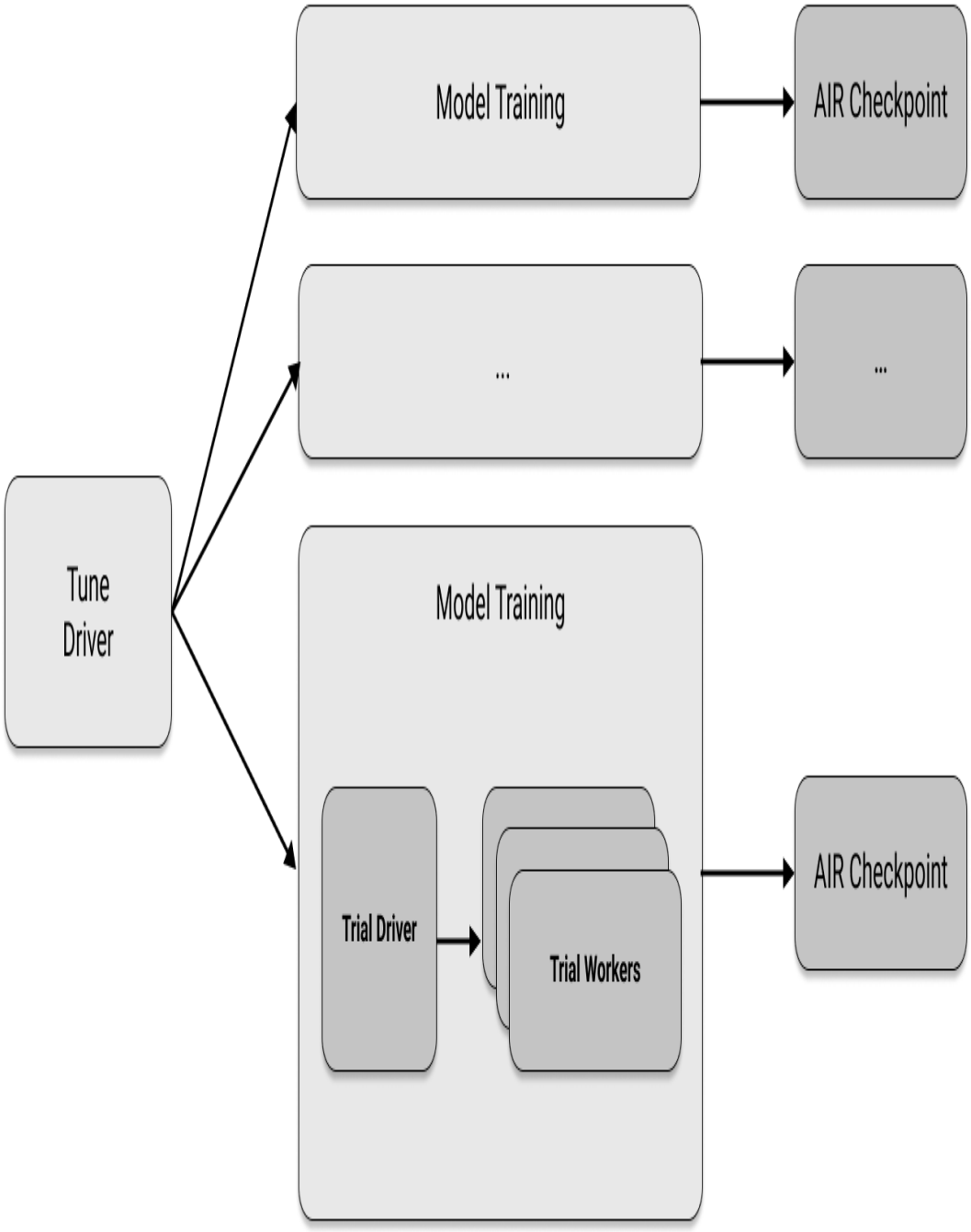


Figure 10-9. Running distributed training on AIR with Tune.

Composite Workload Execution

Composite workloads leverage both Task and Actor-based computation simultaneously. This can lead to interesting resource allocation challenges. Trial actors need to reserve their resources up-front, but stateless tasks don't. The problem you can run into is that all available resources might get reserved for your training Actors and none are left for data loading tasks.

AIR prevents this by allowing Tune to only reserve a maximum of 80% of a node's CPU. You can tune this parameter, but this is a sensible default that ensures basic resource availability for stateless computations. In the common scenario in which your training operations leverage GPUs and your data processing steps don't, this becomes a non-issue.

Online Serving Execution

For online serving Ray Serve manages a pool of stateless actors to serve your requests. Some actors listen for incoming requests and call other actors to perform the predictions. Requests are automatically load balanced using a round-robin algorithm to the pool of actors hosting the models. Load metrics are sent to the Serve component to perform autoscaling.

AIR Memory Management

In this section we're giving you a bit of deep dive into specific memory management techniques of AIR. We're going to discuss to what extent the Ray object store is used by AIR. You can skip this section if you perceive it as too technical. The upshot is that Ray employs smart techniques that will ensure your data and compute are properly distributed and scheduled.

When you load data using Ray Datasets, you already know that internally these Datasets are partitioned into blocks of data in your cluster. A block is simply a collection of Ray objects. Choosing the right block size is a difficult topic and presents a trade-off between

the overhead of having to manage too many small blocks and risking out-of-memory (OOM) exceptions due to blocks that are too large. AIR takes a pragmatic approach and tries to distribute in such a way that blocks don't exceed 512 MB. In case this can't be ensured, a warning will be issued. Should a block not fit into memory, AIR will spill your data to local disk.

Your stateful workloads will use the Ray object store to varying degree. For instance, RLlib uses Ray objects to broadcast model weights to individual rollout workers and for experience data collection. Tune uses it to set up Trials by sending and retrieving Checkpoints. For technical reasons actors risk running into OOM issues if too much memory is required relative to the allocated resources.[9] If you know your memory requirements in advance, you can adapt the `memory` in your `ScalingConfig` accordingly, or simply ask for additional `cpu` resources.

In composite workloads stateful actors (e.g. for training) have to access data created by stateless tasks (e.g. for preprocessing), which makes memory allocation more challenging. Let's have a look at two scenarios:

- If the Actors responsible for training have enough space (in the object store) to fit all training data in memory the situation is simple. First the preprocessing steps run, then all data blocks are downloaded to the respective nodes. Training Actors then simply iterate of data that's kept in memory.
- Otherwise, data processing needs *pipelined execution*, which means that data will be processed by Tasks on the fly and will afterwards be downloaded on demand by training Actors. If the respective training Actor is co-located on the node that did the processing, data will be retrieved from shared memory.

AIR Failure Model

AIR offers fault tolerance for most stateless computations through *lineage reconstruction*. This means Ray will reconstruct Dataset blocks if they are lost due to node failures by re-submitting the necessary tasks, enabling workloads to scale to large clusters. Note that fault tolerance does not apply to head node failures. And a crash of the Global Control Service (GCS) storing cluster metadata will kill all your jobs in the cluster.[10]

Jobs involving stateful computations primarily rely on checkpoint-based fault tolerance. Tune will restart distributed trials from their last checkpoint as configured in its failure configuration. With a configured checkpoint interval, this means that Tune can run trials effectively on clusters consisting of "spot instances". In addition, it is possible to resume entire Tune experiments from the experiment-wide checkpoint in case of whole-cluster failure.

Composite workloads inherit the fault tolerance strategies of both stateless and stateful workloads, retaining the best of both worlds. This means that lineage reconstruction applies to the stateless portion of the workload, and application-level checkpointing still applies to the overall computation.

Auto-Scaling AIR Workloads

AIR libraries can run on Ray autoscaling clusters that we introduced in Chapter 9. For stateless workloads, Ray will auto-scale automatically if there are queued tasks (or queued Dataset compute actors). For stateful workloads, Ray will auto-scale up if there are pending placement groups (i.e., Tune trials) not yet scheduled in the cluster. Ray will auto-scale down when nodes are idle. A node is considered idle when there is no resource usage on the node and also no Ray objects in-memory or spilled on-disk on the node. Since most AIR libraries leverage objects, this means that nodes may be kept if they are holding objects referenced by workers on other nodes (e.g., Dataset block used by another trial).

You should be aware that autoscaling may result in less than ideal data balancing in the cluster, because nodes that are started earlier naturally run more tasks over their lifetime. Consider limiting (e.g., starting with a certain min cluster size) or disabling autoscaling to optimize the efficiency of data-intensive workloads.

Summary

In this chapter you've seen how all Ray libraries introduced up to this point come together to form the Ray AI Runtime. You've learned about all the key concepts that allow you to build scalable ML projects, from experimentation to production. In particular, you've seen how Ray Datasets are used for stateless computations such as feature preprocessing, and how Ray Train, Tune and RLlib are used for stateful computations such as model training. Seamlessly combining these types of computations in complex AI workloads and scaling them out to large clusters is a key strength of AIR. Deploying your AIR projects comes essentially for free, as AIR fully integrates with Ray Serve as well.

In the next and final chapter we're going to show you how Ray, and particularly AIR, fits into the broader landscape of related tools. Knowing the rich set of integrations and extensions of Ray's ecosystem will help you understand better how to leverage Ray in your own projects.

---

1 This gives you parity between your training and serving pipelines, which makes working with AIR convenient, as you don't have to reimplement pipelines for different use cases.

2 Note that technically speaking, not every Trainer needs to specify a `datasets` argument. You can also use framework-specific data loaders, for which we can't show you any examples here.

3 Note that if you run the following example from a jupyter notebook, you do not have worry about it blocking the notebook — it will run just fine. Starting a

server like this is often implemented as a blocking call, but `PredictorDeployment` isn't.

4 Of course, the model used for inference has to be loaded first, but since it's parameters don't change during prediction, trained models can be considered static data in this case. We sometimes refer to this situation as *soft state*.

5 Sometimes AIR uses Ray Actors for stateless tasks for performance reasons, such as caching models in batch inference.

6 Note that preprocessors *can* be stateful, but we haven't discussed any examples of that scenario here.

7 On top of multiple replicas for single models, AIR also supports deployment of multiple models. This allows you to compose multiple models or run A/B testing.

8 All crucial AIR components such as `Trainer` or `BatchPredictor` support this pipelining feature.

9 Stateful workloads use Python's heap memory which is not managed by Ray.

10 GCS can be deployed in *high availability* (HA) mode to prevent this, but that is typically only beneficial for online serving workloads.

Chapter 11. Ray’s Ecosystem and Beyond

Over the course of this book you’ve seen many examples of Ray’s ecosystem already. Now it’s time to take a more systematic approach and show you the full extent of integrations currently available for Ray. We do so by discussing this ecosystem as seen from Ray AIR, so that we can discuss it in the context of a representative AIR workflow.

Clearly, we can’t give you concrete code examples for a majority of the libraries in Ray’s ecosystem. Instead, we have to be content with giving you another Ray AIR example showcasing some integrations and discussing what others are available and how to use them. Where appropriate, we’ll point you to more advanced resources to deepen your understanding.

Now that you know much more about Ray and its libraries, this chapter is also the right place to compare Ray with all it has to offer to *similar* systems. As you’ve seen, Ray’s ecosystem is quite complex, can be seen from different angles, and used for different purposes. That means there are many aspects of Ray that can be compared to other tools in the market.

We'll also comment on how to integrate Ray into more complex workflows in existing ML platforms. To wrap things up, we'll also give you an idea how to continue your journey of *learning Ray* after finishing this book. The notebook for this chapter can be found on GitHub.

A Growing Ecosystem

To give you a glimpse at Ray's ecosystem by means of a concrete example,[1] we're going to show you how to use Ray AIR with data and models from the PyTorch ecosystem, how to log your hyperparameter tuning runs to MLFlow, and how to deploy your trained models with Ray's Gradio integration. Along the way we give you an overview and discuss usage patterns of other noteworthy integrations at each respective stage.

To follow along the code examples in this chapter, make sure to install the following dependencies in your Python environment first:

```
pip install "ray[air, serve]==2.2.0" "gradio==3.11.0"
"requests==2.28.1"
pip install "mlflow==1.30.0" "torch==1.12.1"
"torchvision==0.13.1"
```

As always, the notebook for this chapter is available on GitHub. In the example for this chapter we're first going to load and transform a data set using utilities from the PyTorch framework, and then convert this data into a Ray Dataset to work with it in Ray AIR. We then define a standard PyTorch model and a simple training loop for it that we can leverage in Ray Train. Next, we wrap this `TorchTrainer` in a Tuner and log trial results to MLFlow using the `MLFlowLogger` that ships with Ray Tune. Finally, we're going to serve our trained model using Gradio running on Ray Serve.

In other words, the example we're discussing takes common Python data science libraries that you might already use and wraps them in

an AIR workflow by leveraging Ray's ecosystem integrations. The focus is more on these integrations and how they interface with AIR, and less on the concrete use case.

Data Loading and Processing

In Chapter 6 you've learned about the basics of Ray Datasets, how to create them from common Python datastructures, load Parquet files from storage systems like S3, or use Ray's *Dask on Ray* integration to interface with Dask.

To give you yet another example of Ray Dataset's capabilities, let's show how you can work with image data loaded through PyTorch's `torchvision` extension. The idea is simple. We're going to load the common `CIFAR-10` data set that is available in PyTorch through the `torchvision.datasets` package and then make it a Ray Dataset. Specifically, we're going to define a function called `load_cifar` that returns the `CIFAR-10` data. for training or testing purposes.

```
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 We use a PyTorch transform to return normalized tensor data.

❷

The `CIFAR-10` data set is loaded using the `datasets` module of the `torchvision` package.

❸ We make is so that the `loader` function either returns training or testing data.

Note that so far we haven't touched any Ray library, and you could have used any other dataset or transform from PyTorch in the same way. To get Ray Datasets for use with AIR, we're going to supply our `load_cifar` data loader functions to the `from_torch` utility from Ray AIR:

```
from ray.data import from_torch


train_dataset = from_torch(load_cifar(train=True))
test_dataset = from_torch(load_cifar(train=False))
```

The `CIFAR-10` dataset is used for image classification tasks and consists of `32` pixels wide square images, and comes with labels of a total of ten categories. So far we've only loaded this dataset in the form provided by PyTorch, but we still need to transform it to use it with an AIR Trainer. You do this by creating an `image` and a `label` column that we can then reference in a Trainer. The best way to do so is by *mapping batches* of train and test data to a dictionary of numpy arrays with precisely these two columns.

```
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)
```

❷

```
test_dataset = test_dataset.map_batches(to_labeled_image)
```

❶ We transform each `batch` of data by returning an `image` and a `label` numpy array.

❷ We apply `map_batches` to transform our initial datasets.

Before we move on to model training, let's take a brief look at the supported input formats of the Datasets library:[2]

Table 11-1. The RayDatasets Ecosyst

e
m

Integration	Type	Description
Text, Binary, Image Files, CSV, JSON	Basic data formats	Supporting such basic formats should not strictly speaking be considered an *integration*, but it's worth knowing that Ray Datasets can load and store these formats.
NumPy, Pandas, Arrow, Parquet, Python objects	Advanced data formats	Datasets supports working with common ML data libraries such as Numpy and Pandas, but can also read custom Python objects or read Parquet files.
Spark, Dask, Mars, Modin	Advanced third-party integrations	Ray interoperates with more complex data processing systems by means of community integrations, such as Spark on Ray (RayDP), Dask on Ray, Mars on Ray, or Pandas on Ray (Modin).

We'll talk about the relationship of Ray with systems such as Dask or Spark in more detail later in this chapter.

Model Training

Having properly shaped our `CIFAR-10` train and test data using Ray Datasets, we can now define a classifier to train our data on. As this is likely the most natural scenario, we're going to define a PyTorch model to define an AIR Trainer here. But it's worth reminding you from Chapter 7 that you could just as easily switch frameworks at this point and work with Keras, HuggingFace, scikit-learn or any other library supported by AIR.

We proceed in three steps: define a PyTorch model, specify the training loop that AIR should run using this model, and define an AIR Trainer that we can fit on our training data. To start with, let's define a simple convolutional neural network with max pooling and rectified linear (`relu`) activations with Pytorch that's built to operate on our `CIFAR-10` dataset. If you know PyTorch, the following definition of the neural network `Net` should be straightforward. If you don't, it suffices to know that to define a `torch.nn.Module` the only thing you need to provide is the definition of a `forward` pass of your neural net.

```
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
```

Now, to define an AIR `TorchTrainer` for our `Net`, you already know that you need a training dataset, a scaling and an optional run configuration.[3] You also need to tell AIR what each worker should do when calling `fit`, by defining an explicit training loop that grants you

maximal flexibility for your training process. The training function takes a `config` dictionary that we can use to specify properties at runtime.

The training loop we're using here is just what you would expect: we simply load the model and the data on each worker and then train on batches of data for a specified number of epochs (while reporting the training progress). That's a fairly standard training loop, but there are a couple of crucial spots in which you have to be careful:

- To load the model on a worker, you have to use the `prepare_model` utility from `ray.train.torch` on our `Net()`.
- To access the data shard available to the worker, you access `get_dataset_shard` on your current `ray.air.session`. For training, we use the `"train"` key of that shard and transform it to batches of the right size using `iter_torch_batches`.
- To pass information about the training metrics you're interested in, you use `session.report` from AIR.

Here's the full definition of our PyTorch training loop for each worker:

```
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())
            ),
        )
```

❶ The training loop defines the model, loss and optimizer used first. Note the usage of `prepare_model` here.

❷ We then load the train dataset *shard* on this worker and create an iterator containing batches of data from it.

❸ As per our earlier definition, our `data` is a pandas dataframe which has an `"image"` and a `"label"` column.

❹ In the training loop we compute the forward and backward pass as we would with any PyTorch model.

❺ We keep track of a running loss term for each 1000 training batches.

❻ Lastly, we `report` this loss using our AIR `session`, by passing a Checkpoint with the current model state.

The definition of this function might feel a bit long, given that we're doing a fairly standard training procedure. While it'd certainly be possible to create a simple wrapper for PyTorch models in such

cases, defining your own training loop gives you full customizability to tackle more complex scenarios. We pass the `training_loop` to the `train_loop_per_worker` argument of our AIR trainer and specify the configuration for this loop by passing a dictionary with the necessary keys to the `train_loop_config`.

To make things more interesting, and to showcase another Ray integration, we're going to be logging the results of our `TorchTrainer` training run to mlflow by passing a callback to our `RunConfig`, namely an `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()
```

You could also use other third party logging libraries, such as *Weights & Biases* or *CometML*, by passing similar callbacks to your AIR Trainer or Tuner.[4]

We summarize all ML training-related integrations of Ray, which span both Ray Train an RLlib, in the following table:

Table 11-2. The RayTrain & RLlib Ecosyst

e
m

Integration	Type
TensorFlow, PyTorch, XGBoost, LightGBM, Horovod	Train integrations maintained by the Ray Team
scikit-learn, HuggingFace, Lightning	Train integrations maintained by the community
TensorFlow, PyTorch, OpenAI gym	RLlib integrations maintained by the Ray Team
JAX, Unity	RLlib integrations maintained by the Ray Team

We distinguish between community-sponsored integrations and ones that are maintained by the Ray team itself. Most integrations we've talked about in this book were native integrations, but due to the collaborative nature of open source software, you often feel no difference in maturity between native and third-party integrations.

To wrap up the training-related integrations of AIR, let's have a quick bird's eye view at Tune's ecosystem as well:

Table 11-3. The RayTune Ecosystem

Integration	Type

Optuna, HyperOpt, Ax, BayesOpt, BOHB, Dragonfly, FLAML, HEBO, Nevergrad, SigOpt, skopt, ZOOpt	Hyperparameter Optimization library
TensorBoard, mlflow, Weights & Biases, CometML	Logging and Experimentation management.

Model Serving

Gradio is a popular way for practitioners to demo their ML models, and it provides you with many simple primitives to create graphical user interface elements simply by describing them through the `gradio` library in Python. As you will see, defining and deploying Gradio interfaces is straightforward, but it's even easier to then wrap it in a so-called `GradioServer` from Ray Serve, which allows you to scale out any Gradio app on a Ray cluster.

To showcase how to run a Gradio app with Ray Serve on the model we just trained, let's first store the `result` of our training procedure to disk. We do this by writing the respective AIR Checkpoint to a local folder of our choice, so that we can restore this model *from checkpoint* in another script.

```
CHECKPOINT_PATH = "torch_checkpoint"
result.checkpoint.to_directory(CHECKPOINT_PATH)
```

Next, let's create a file called `gradio_demo.py` right next to the `"torch_checkpoint"` path for simplicity. In this script, let's load our PyTorch model again by first restoring the `TorchCheckpoint` and then using this Checkpoint and our `Net()` definition to create a `TorchPredictor` that we can use for inference:

```
# 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()
)
```

Note that this requires you to import, or otherwise make available, the definition of `Net` in the `gradio_demo.py` script.[5]

Next, we have to define the Gradio `Interface`, which we define to take images as input and produce labels as output. Additionally, we have to specify how an input `Image` has to be transformed to produce a `Label`. By default, Gradio represents images as numpy arrays, so we can simply ensure that this array has the right shape and data type and then pass it to our `predictor`. As this `predictor`, namely our `TorchPredictor`, produces a probability distribution, we take the `argmax` of its prediction to get an integer result that we can use as label. Put the following code into your `gradio_demo.py` Python script, too.

```
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,
    inputs=gr.Image(),
    outputs=gr.Label(num_top_classes=10)
)
```

```
app = GradioServer.options(  ❸
    num_replicas=2,
    ray_actor_options={"num_cpus": 2}
).bind(demo)
```

❶ The `predict` function maps Gradio inputs to outputs by leveraging our `predictor`.

❷ The Gradio interface has one input (an image), one output (a label for one of ten categories of the CIFAR-10 dataset), and a function `fn` to connect the two.

❸ We `bind` the Gradio `demo` to a Ray Serve `GradioServer` object which gets deployed on two replicas with two CPUs as resources each.

To run this application you can now simply type the following command into a shell:[6]

```
serve run gradio_demo:app
```

This command spins up our Serve-backed Gradio demo that you can access on `localhost:8000`. You can upload or drag-and-drop images into the respective input field and request predictions that you see in the output field of the app.[7]

It's important to note that this example is really just a thin wrapper around Gradio and would work with any other Gradio app of your choice. In fact, if you would call `demo.launch()` instead of defining the Serve `app` in your script, you could simply launch this as a regular Gradio app with `python gradio_demo.py` as well.

The other noteworthy detail that's easy to glance over is that we fed our `predictor` a `numpy` array. If you check the definition of the data format we used for training, you'll recall that `predictor` is expected to work on Ray `Dataset` instances. The `predictor` instance is also smart enough infer that a single `numpy` input must

be the `"image"` portion of the full input (we don’t need a `"label"` to run inference).

To wrap up this section, let’s have a look at all of Serve’s current integrations in one table:

Table 11-4. The RayServe Ecosystem

Library	Description
Serving Frameworks & Applications	FastAPI, Flask, Streamlit, Gradio
Explainability & Observability	Arize, Seldon Alibi, Whylabs

Building Custom Integrations

Before explaining the relationship of Ray with other complex software frameworks in a bit more detail, let's talk about how to build your own integrations for Ray AIR. Since AIR is designed for extensibility, you find suitable interfaces for all the tasks you want to build custom integrations for.

To give you an example, let's say you want to read data from Snowflake, train a JAX model on it and log your tuning results to Neptune.[8] At the time of this writing, there are no such integrations available, but it's not unlikely that this will change in the future. We didn't pick these integrations (Snowflake, JAX, Neptune) to showcase any preference, they just happen to be interesting tools from the ecosystem. In any case, it's worth knowing how to build such integrations.

To load data from Snowflake into a Ray Dataset, you have to create a new `Datasource`. You define such a `Datasource` by specifying how to set it up (`create_reader`), how to write to the source (`do_write`), and what happens on successful and failed write attempts (`on_write_complete` and `on_write_failed`). Given a concrete `SnowflakeDatasource` implementation, you could then read your data into a Ray `Dataset` as follows:

```
from ray.data import read_datasource, datasource

class SnowflakeDatasource(datasource.Datasource):
```

```
    pass


dataset = read_datasource(SnowflakeDatasource(), ...)
```

Next, let's say you have an interesting JAX model that you want to scale out using Ray Train's capabilities. Specifically, let's assume you want to run data-parallel training of the model, that is to train this single model on several data shards in parallel. For this purpose, Ray comes with a so-called `DataParallelTrainer`. To define one, you have to create a `train_loop_per_worker` for your training framework and define how JAX should be handled by Train internally.[9] Having a `JaxTrainer` implementation, you can leverage the same `Trainer` interface that we've used in all AIR examples:

```
from ray.train.data_parallel_trainer import
DataParallelTrainer


class JaxTrainer(DataParallelTrainer):
    pass


trainer = JaxTrainer(
    ...,
    scaling_config=ScalingConfig(...),
    datasets=dict(train=dataset),
)
```

Finally, to use Neptune for logging and visualising your Tune Trials, you can define a `LoggerCallback` that gets passed into the run configuration of your Tuner. To define one, you need to specify how to create the logger (`setup`), what's supposed to happen at the beginning and end of a trials (`log_trial_start` and `log_trial_end`) and how to log your results (`log_trial_result`). If you've implemented such a class, say `NeptuneCallback`, you can the use it the same way as we used the `MLflowLoggerCallback` earlier in this chapter:

```
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()])
)
```

While building integrations is not always as hard as initially perceived, third party software is a moving target and maintaining integrations can be challenging. Still, you are now aware of three of the most common integration scenarios for new AIR components, and maybe you feel inclined to work on a community-sponsored integration of your favorite tool.

An Overview of Ray's Integrations

Let's summarize all the integrations mentioned in this chapter (and throughout the book) in one concise diagram. In figure Figure 11-1 we list all integrations available at the time of writing.

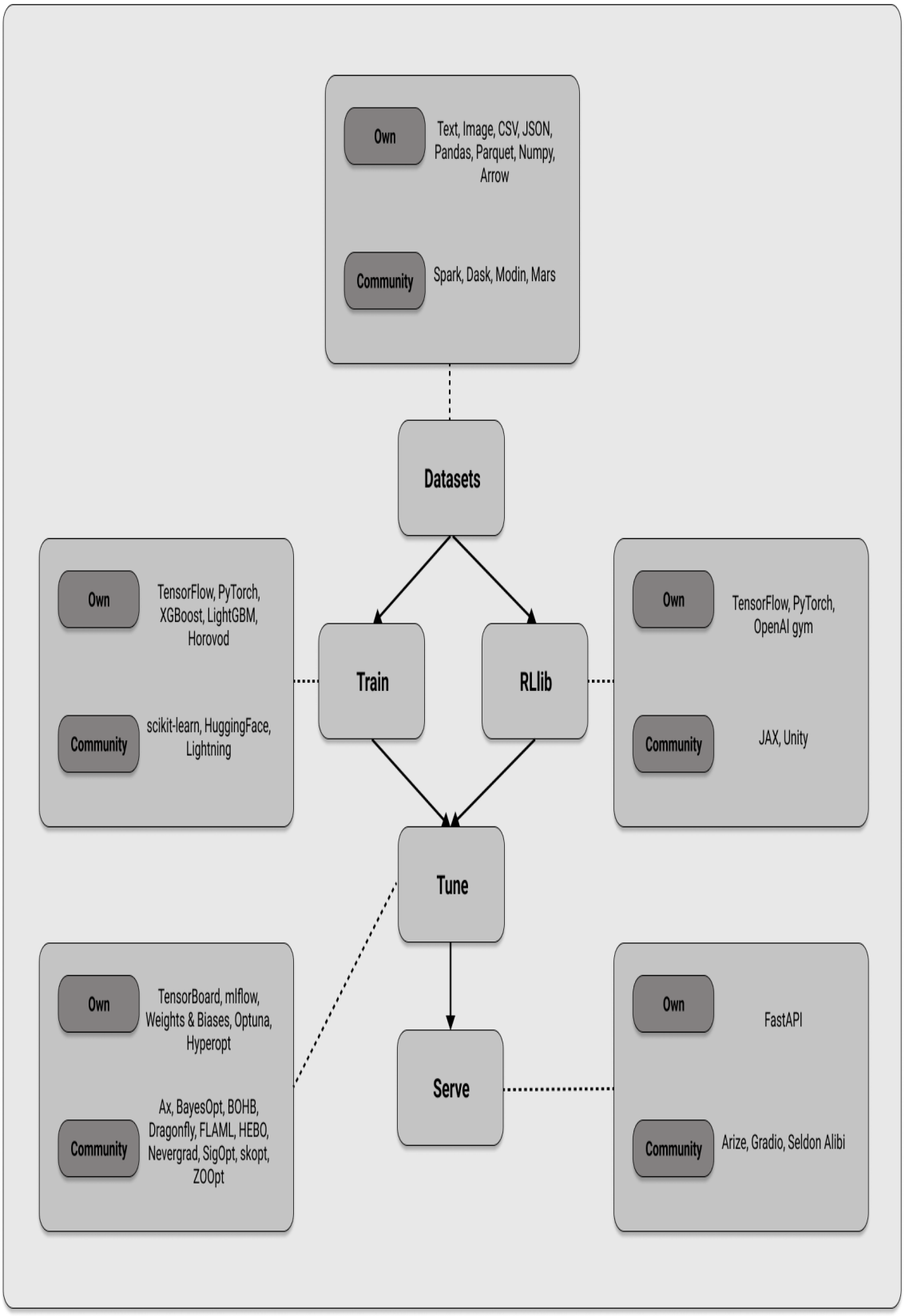

Own
Text, Image, CSV, JSON, Pandas, Parquet, Numpy, Arrow
Community
Spark, Dask, Modin, Mars
Datasets
Own
TensorFlow, PyTorch, XGBoost, LightGBM, Horovod
Community
scikit-learn, HuggingFace, Lightning
Train
RLlib
Own
TensorFlow, PyTorch, OpenAI gym
Community
JAX, Unity
Tune
Own
TensorBoard, mlflow, Weights & Biases, Optuna, Hyperopt
Community
Ax, BayesOpt, BOHB, Dragonfly, FLAML, HEBO, Nevergrad, SigOpt, skopt, ZOOpt
Serve
Own
FastAPI
Community
Arize, Gradio, Seldon Alibi

Figure 11-1. The Ray AIR Ecosystem summarized

Ray and Other Systems

We've not made any direct comparisons with other systems up to this point, for the simple reason that it makes little sense to compare Ray to something if you don't have a good grasp of what Ray is yet. As Ray is quite flexible and comes with a lot of components, it can be compared to different types of tools in the broader ML ecosystem.

Let's start with a comparison of the more obvious candidates, namely Python-based frameworks for cluster computing.

Distributed Python Frameworks

If you consider frameworks for distributed computing that offer full Python support and don't lock you into any cloud offering, the current "big three" are Dask, Spark, and Ray. While there are certain technical and context-dependent performance differences between these frameworks, it's best to compare the in terms of the workloads you want to run on them. The following table gives a brief overview for some of the most common workload types:

Table 11-5. Comparing workload type

support of Ray, Dask, and Spark

Workload Type	Dask	Spark	Ray
Structured Data Processing	First class support	First class support	Supported via Ray Datasets and integrations, but not first class

Low-level Parallelism	First class support via tasks	None	First class support via tasks and actors
Deep Learning Workloads	Supported, but not first class	Supported, but not first class	First class support via several ML libraries

Ray AIR and the Broader ML Ecosystem

Ray AIR focuses primarily on *AI compute*, for instance by providing any kind of distributed training via Ray Train, but it's not built to cover every aspect of an AI workload. For instance, AIR chooses to integrate with tracking and monitoring tools for ML experiments, as well as with data storage solutions, rather than providing native solutions. In terms of purely complementary components, we can identify the following categories:

Table 11-6. Complementary Ecosystem

Components

Category	Examples
ML Tracking and Observability	mlflow, Weights & Biases, Arize, etc.
Training Frameworks	PyTorch, TensorFlow, Lightning, JAX, etc.
ML Feature Stores	Feast, Tecton, etc.

On the other side of the spectrum, you can find categories of tools for which Ray AIR can be considered and alternative. For instance, there are many framework-specific toolkits such as TorchX or TFX that tie in tightly with their respective frameworks. In contrast, AIR is framework-agnostic, thereby preventing vendor lock-in, and offers similar tooling.[10]

It's also interesting to briefly touch on how Ray AIR compares to specific cloud offerings. Some major cloud services offer comprehensive toolkits to tackle ML workloads in Python. To just name one, AWS Sagemaker is a great all-in-one package that allows you to connect well with your AWS ecosystem. AIR does not aim to replace tools like Sagemaker. Instead, it aims to provide alternatives

for compute-intensive components like training, evaluation, and serving.[11]

AIR also represents a valid alternative to ML workflow frameworks such as KubeFlow or Flyte. In contrast to many container-based solutions, AIR offers an intuitive, high-level Python API, and offers native support for distributed data. We can summarize the situation as follows:

Table 11-7. Alternative Ecosystem C

o
m
p
o
n
e
n
t
s

Category	Examples
Framework-specific Toolkits	TorchX, TFX, etc.
ML Workflow Frameworks	KubeFlow, Flyte, FBLearner FLow

Sometimes the situation is not as clear-cut, and Ray AIR can be seen or used as both an alternative or a complementary component in the ML ecosystem.

For instance, as open source systems Ray, and AIR in particular, can be used within hosted ML platforms such as SageMaker, but you can also build your own ML Platforms with it.[12] Also, as mentioned before, AIR can't always compete with dedicated big data processing systems like Spark or Dask, but often Ray Datasets can be enough to suit your processing needs.

As we've mentioned in Chapter 10, it is central to AIR's design philosophy to have the ability to express your ML workloads in a single script and execute it on Ray as the single distributed system. Since Ray handles all the task placement and execution on your cluster for you under the hood, there's usually no need to explicitly orchestrate your workloads (or stitch together many complex distributed systems). Of course, this is philosophy should not be

taken too literally — sometimes you need multiple systems or split up tasks into several stages. On the other hand, dedicated workflow orchestration tools like Argo or AirFlow can be very useful when used in a complementary fashion.[13] For instance, you might want to run Ray as a step in the Lightning MLOps framework.

Table 11-8. Ecosystem Components that a

t
A
I
R

c
a
n

c
o
m
p
l
e
m
e
n
t
o
r
r
e
p
r
e
s
e
n
t
a
n

a
lt
e

r
n
a
ti
v
e

f
o
r
.

Category	Examples
ML Platforms	SageMaker, Azure ML, Vertex AI, Databricks, etc.
Data Processing Systems	Spark, Dask, etc.
Workflow Orchestrators	Argo, AirFlow, Metaflow
MLOps Frameworks	ZenML, Lightning

In case you already have an existing machine learning platform, such as Vertex or SageMaker, you can use any subset of Ray AIR to augment your system.[14] In other words, AIR can complement existing ML platforms by integrating with existing pipeline or workflow orchestrators, storage, and tracking services, without requiring a replacement of your entire ML platform.

How to Integrate AIR into Your ML Platform

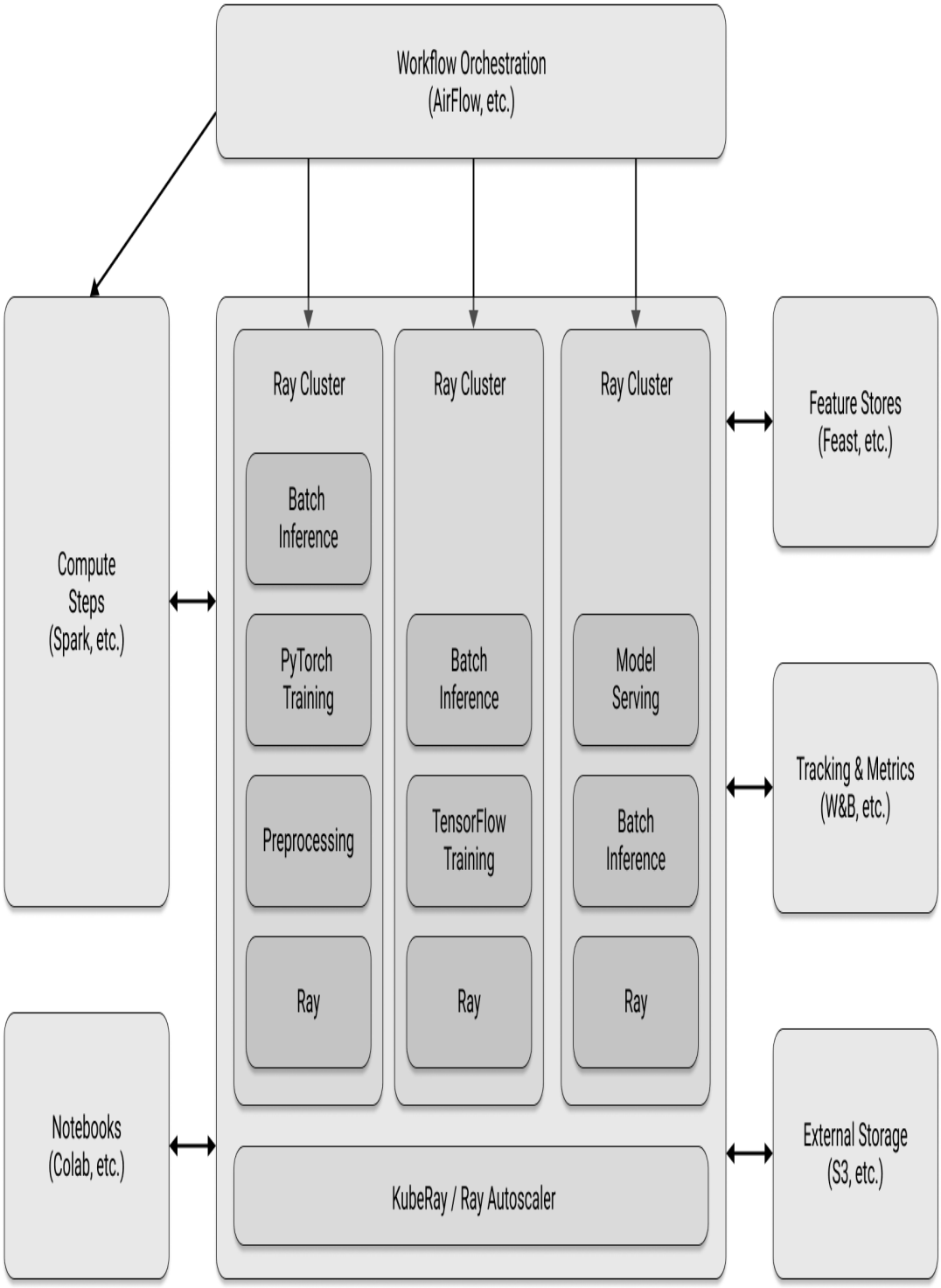
Now that you have a deeper understanding of the relationship of Ray, and AIR in particular, to other ecosystem components, let's try

to summarize what it takes to build your own ML platform, and how to integrate Ray with other ecosystem components.

The core of your ML system build with AIR consists of a set of Ray Clusters, each responsible for different jobs. For instance, one cluster might run preprocessing, train a PyTorch model and run inference, another one might simply pick up previously trained models for batch inference and model serving, and so on. You can leverage the Ray Autoscaler to fulfill your scaling needs, and could deploy the whole system on Kubernetes with KubeRay. You can then augment this core system with other components as you see fit, for instance:

1. You might want to add other compute steps to your setup, such as running data-intensive preprocessing tasks with Spark.
2. You can use a workflow orchestrator such as AirFlow, Oozie, or SageMaker Pipelines to schedule and create your Ray clusters and running Ray AIR apps and services. Each AIR app can be part of a larger orchestrated workflow, for instance by tying into a Spark ETL job from the last bullet point.[15]
3. You can also create your Ray AIR clusters for interactive use with Jupyter notebooks, for instance hosted by Google Colab or Databricks Notebooks.
4. If you need access to a feature store such as Feast or Tecton, Ray Train, Datasets, and Serve have an integration for such tools.[16]
5. For experiment tracking or metric stores, you've already seen that Ray Train and Tune provide integration with tools such as MLFlow and Weights & Biases.
6. You can also retrieve and store your data and models from external storage solutions like S3, as we've shown in many examples already.

Figure Figure 11-2 puts all those pieces together in one concise diagram:



Ray AI Runtime

Figure 11-2. Building your own ML Platform with Ray's AI Runtime and other components from the ML ecosystem.

Where to Go from Here?

We've come a long way from a first overview of Ray in the first chapter to the discussion of its ecosystem in this one. But as this is an introductory book, we've just touched the proverbial tip of the iceberg of Ray's capabilities. While you should now have a good grasp of the basics of Ray Core and how Ray Clusters work, know when to use AIR and its constituent libraries Datasets, Train, RLlib, Tune and Serve, there's still so much more to learn about each aspect.

To start with, the extensive user guides of Ray Core will give you a much deeper understanding about Ray Tasks, Actors and Objects, their placement on cluster nodes, and how to handle dependencies for your applications. In particular, you find interesting patterns and anti-patterns to design your Ray Core programs well and avoid common pitfalls. To learn more about the inner workings of Ray, we recommend reading some of the advanced papers on Ray, such as its architecture whitepaper.

An interesting topic we skipped entirely is Ray's tooling around *observability*. The official Ray Observability documentation is a good starting point for this topic. You can learn how to debug and profile your Ray applications there, how to log information from your Ray clusters, monitor their behaviour, and export important metrics there. You also find an introduction to the *Ray dashboard* there,[17] which can be quite helpful to understand your Ray programs.

The focus of this book was to introduce the core ideas of Ray to ML practitioners and give you practical starting points to tackle your workloads with Ray. However, a topic that deserves more attention than the one chapter we could include in this book, is how to build, scale and maintain Ray Clusters. Currently, the best introduction to

advanced Ray Cluster topics is the Ray documentation. There you can learn how to deploy clusters on all major cloud providers, how to scale out clusters on Kubernetes, and how to submit Ray jobs to a cluster in much more detail than we could cover here.

In this chapter, we could only mention the majority of Ray integrations in passing. If you want to learn more about Ray's third-party integrations, a good starting point is the Ecosystem page on Ray's documentation. Also, it's important to mention that there are more Ray libraries available that simply didn't make the cut for this book, like Ray Workflows or a distributed, drop-in replacement for Python's `multiprocessing` library.

Lastly, if you're interested in becoming part of the Ray community, there are many good ways of doing so. You can join the Ray Slack to get in touch with Ray developers and other community members, or join Ray's discussion forum to get your questions answered. If you want to help develop Ray, be it contributing to the documentation, adding a new use case, or helping the open source community with new features or bug fixes, you should check out the official contributor guide to Ray.

Summary

In this chapter you learned more about Ray's ecosystem, as seen from its AI Runtime. You've seen the full extent of available integrations of Ray AIR libraries and an example of training and serving an ML model using three different integrations, namely PyTorch for data loading and training, MLFlow for logging, and Gradio for serving your model. You should now be able to go out there and run your own AIR experiments, together with all the tools you're already using or intend to use in the future. We've also discussed Ray's limits, how it compares to related systems of various kind, and how you can use Ray with other tools to augment or build out your own ML platforms.

This wraps up this chapter, and the book. We hope we could peak your interest in Ray and help you start your journey with it. The introduction of AIR brought many new features to the Ray ecosystem, and there's certainly more on the roadmap that you can be excited about. There's no doubt that you can now dive into any advanced Ray material — and maybe you already have an idea to build your first own Ray app?

---

1 You can find the current list of integrations to the AIR ecosystems on the Ray documentation. More generally, you find a list of integrations for Ray on the Ray Ecosystem page. On the latter, you find many more integrations than the ones we discuss here, such as ClassyVision, Intel Analytics Zoo, John Snow Labs' NLU, Ludwig AI, PyCaret, and SpaCy.

2 You can see the continually updated list of supported formats on the Datasets documentation. The supported *output* formats for Datasets largely overlap with the input formats.

3 To convert any PyTorch model to Ray AIR, follow the user guide on this topic.

4 For instance, using the `WandbLoggerCallback` you can not only log training results to Weights & Biases, but also automatically upload your checkpoints, as demonstrated in this Ray tutorial.

5 For simplicity, we duplicated the definition of `Net` in `gradio_demo.py` on GitHub.

6 When using the Gradio integration, Ray Serve will automatically run a Gradio app under the hood. The Gradio app is instrumented to access the type hints on each Serve deployment. When the user submits a request, the Gradio app displays the output of each deployment using a Gradio Block that matches the output type.

7 The application expects images of the right size, as we didn't want to make preprocessing in `predict` more involved than necessary. You can use the image data provided in the repository of this book or simply search for CIFAR-10 images online to test the app yourself.

8 In this chapter we're going to assume that you know about these different tools from the ecosystem. If you don't, for this section it's enough to understand that Snowflake is a database solution that you might want to integrate with, JAX is an ML framework, and Neptune can be used for experiment tracking.

9 To be precise, you have to define a `Backend` for JAX, together with a `BackendConfig`. Your `DataParallelTrainer` then has to be initialized with this backend and your training loop.

10 This represents a clear trade-off, as framework-specific tools are highly customized and offer many benefits.

11 It should also be mentioned that Anyscale itself provides a managed service with enterprise features to support you building ML application on top of Ray.

12 We'll give you a rough sketch for how to do this in the next section.

13 We should also mention that Ray has a library called *Ray Workflows*, which is currently in alpha. Compared to tools such as AirFlow, Workflows is more low-level, but allows you to run durable application workflows natively on Ray. You can find more information about Workflows on the Ray documentation.

14 We use the term *ML platform* in the broadest sense possible, namely to signify any system that is responsible for running end-to-end ML workloads.

15 Orchestration of task graphs can be handled entirely within Ray AIR. External workflow orchestrators will integrate nicely but are only needed if running non-Ray steps.

16 The Ray team has build a demo of a reference architecture that showcases the Feast integration.

17 At the time of this writing, the dashboard gets overhauled. Look and functionality might change drastically, so we didn't include descriptions or screenshots here.

Index

About the Author

Max Pumperla is a data science professor and software engineer located in Hamburg, Germany. He's an active open source contributor, maintainer of several Python packages, author of machine learning books and speaker at international conferences. As head of product research at Pathmind Inc. he's developing reinforcement learning solutions for industrial applications at scale using Ray. Pathmind works closely with the AnyScale team and is a power user of Ray's RLlib, Tune and Serve libraries. Max has been a core developer of DL4J at Skymind, helped grow and extend the Keras ecosystem, and is a Hyperopt maintainer.

Edward Oakes

Richard Liaw

Colophon

The animal on the cover of *Learning Ray* is a marbled electric ray (*Torpedo marmorata*), also known as a torpedo ray. Marbled electric rays can be found in the eastern Atlantic Ocean from Africa to Norway, as well as in the Mediterranean Sea. They are bottom dwellers, preferring to live in shallow to moderately deep water in rocky reefs, seagrass beds, and muddy flats.

These rays have a mottled brown and black skin color, camouflage that makes them difficult to spot. Female rays can grow up to 2 feet long, while male rays are slightly smaller at 1.2 feet. Unlike other rays that have venomous scales or barbs, marbled electric rays have specialized electrogenic organs at the base of their pectoral fins. The organs can emit an electric charge of up to 200 volts.

Marbled electric rays bury themselves in muddy waters during the day and emerge at night to forage for food. They use their camouflage to hide from prey, wait for it to swim near, and then jump, using their electric charge to shock or kill. Rays like to eat small fish, such as gobies, mullet, mackerel, and damselfish. It is also possible for them to swallow fish larger than the width of their mouth because they are able to expand their jaws.

Because these rays are capable of electrocution, they have few natural predators in their habitats. They are occasionally collected when trolling for fish but are tossed back into the water because they are of little use to humans. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on an antique line engraving from Lydekker's*Royal Natural History*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.