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 Data Science

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.

Max Pumperla, Edward Oakes, and Richard Liaw

Beijing · Boston · Farnham · Sebastopol · Tokyo O'REILLY®

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

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

The views expressed in this work are those of the authors, and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

This work is part of a collaboration between O'Reilly and Anyscale. See our statement of editorial independence.

978-1-098-11716-0

Table of Contents

1. An Overview of Ray. 7
What is Ray? 8
What led to Ray? 8
Three Layers: Core, Libraries & Ecosystem 11
A Distributed Computing Framework 11
A Suite of Data Science Libraries 14
Machine Learning and the Data Science Workflow 14
Data Processing with Ray Data 16
Model Training 18
Hyperparameter Tuning 22
Model Serving 24
A Growing Ecosystem 26
How Ray Integrates and Extends 26
Ray as Distributed Interface 26
Summary 27

2. Getting Started With Ray Core. 29
An Introduction To Ray Core 29
A First Example Using the Ray API 30
An Overview of the Ray Core API 40
Design Principles 41
Understanding Ray System Components 42
Scheduling and Executing Work on a Node 42
The Head Node 45
Distributed Scheduling and Execution 46
Summary 48

3. **Building Your First Distributed Application. 49**
Setting Up A Simple Maze Problem 50
Building a Simulation 55
Training a Reinforcement Learning Model 58
Building a Distributed Ray App 62
Recapping RL terminology 67
Summary 68

4. **Reinforcement Learning with Ray RLlib. 69**
An Overview of RLlib 70
Getting Started With RLlib 71
Building A Gym Environment 71
Running the RLlib CLI 73
Using the RLlib Python API 74
Configuring RLlib Experiments 81
Resource Configuration 81
Debugging and Logging Configuration 82
Rollout Worker and Evaluation Configuration 82
Environment Configuration 83
Working With RLlib Environments 83
An Overview of RLlib Environments 83
Working with Multiple Agents 84
Working with Policy Servers and Clients 89
Advanced Concepts 92
Building an Advanced Environment 92
Applying Curriculum Learning 93
Working with Offline Data 95
Other Advanced Topics 96
Summary 97

5. **Distributed Training with Ray Train. 99**
The Basics of Distributed Model Training 99
Introduction to Ray Train 100
Creating an end-to-end example for Ray Train 101
Preprocessors in Ray Train 103
Usage of preprocessors 103
Serialization of preprocessors 103
Trainers in Ray Train 104
Distributed Training for Gradient Boosted Trees 104
Distributed Training for Deep Learning 105
Scaling out training with Ray Train Trainers 107
Connecting Data to Distributed Training 108

Ray Train Features 110
- Checkpoints 110
- Callbacks 110
- Integration with Ray Tune 111
- Exporting Models 112
- Some Caveats 113

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

—Leslie Lamport

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 you simply might *have to* scale them 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 I find the most useful for my 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 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, 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 data science*.

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, but we defer any in-depth treatment of Ray's APIs and components to later chapters. You can view this chapter as an overview of the whole book as well.

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. 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[1], 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?

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. None of the tools existing at the time met their needs. They were looking for easy ways to speed up their workloads by distributing them to compute clusters. The workloads they had in mind were quite flexible in nature and didn't fit into the analytics engines available. At the same time, RISELab 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. 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. Anyscale, the company behind Ray, is building a managed Ray Platform and offers hosted solutions for your Ray applications. Let's have a look at an example of what kinds of applications Ray was designed for.

Flexible Workloads in Python & 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

[1] Moore's Law held for a long time, but there might be signs that it's slowing down. We're not here to argue it, though. What's important is not that our computers generally keep getting faster, but the relation to the amount of compute we need.

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). We've dedicated all of Chapter 4 to RL. 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 scale from 1 to 10). 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. These simulations can become complicated quite quickly, as you might imagine from the examples we've given.

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.[2] What I do know is that the world is full of complex, dynamic and interesting examples that we need to tackle. For that we need computational frameworks that help us do that, and Ray was built to do exactly that.

[2] For the experts among you, I don't claim that RL is the answer. RL is just a paradigm that naturally fits into this discussion of AI goals.

RISELab created Ray to build and run complex AI applications at scale, and reinforcement learning has been an integral part of Ray from the start.

Three Layers: Core, Libraries & 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.

1. A low-level, distributed computing framework for Python with a concise core API.[3]
2. A set of high-level libraries for data science built and maintained by the creators of Ray.
3. A growing ecosystem of integrations and partnerships with other notable projects.

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 it. In practice, most data scientists will use these higher level libraries directly and won't often need to resort to the core API. 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.

[3] 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.

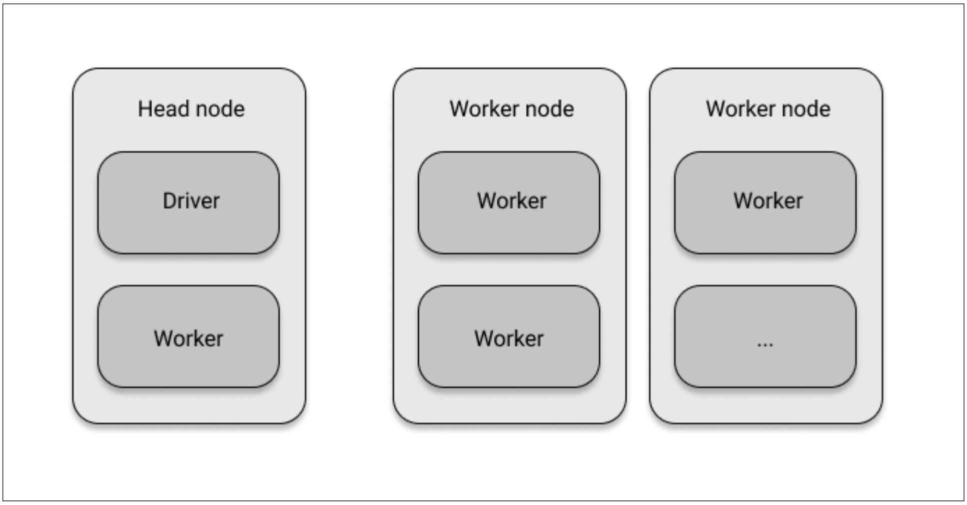

Figure 1-1. The basic components of a Ray cluster.

What's interesting is that a Ray cluster can also be a *local cluster*, i.e. 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[4] on any of the major operating systems should work seamlessly using `pip`:

```
pip install "ray[rllib, serve, tune]"==1.9.0
```

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. 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 use the `ipython` interpreter, which I find to be the most suitable environment 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. In your Python session you can now easily import and initialize Ray as follows:

[4] We're using Ray version `1.9.0` at this point, as it's the latest version available as of this writing.

Example 1-1.

```
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. In this case you didn't provide any arguments to the `init` function. If you wanted to run Ray on a "real" cluster, you'd have to pass more arguments to `init`. The rest of your code would stay the same.

After running this code you should see output of the following form (we use ellipses to remove the clutter):

```
... INFO services.py:1263 -- View the Ray dashboard at http://127.0.0.1:8265
{'node_ip_address': '192.168.1.41',
 'raylet_ip_address': '192.168.1.41',
 'redis_address': '192.168.1.41:6379',
 'object_store_address': '.../sockets/plasma_store',
 'raylet_socket_name': '.../sockets/raylet',
 'webui_url': '127.0.0.1:8265',
 'session_dir': '...',
 'metrics_export_port': 61794,
 'node_id': '...'}
```

This 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. In all likelihood 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. We'll come back to the dashboard in later chapters.

We're not quite ready to dive into all the details of a Ray cluster here. To jump ahead just a little, you might see the `raylet_ip_address`, which is a reference to a so-called *Raylet*, which is responsible for scheduling tasks on your worker nodes. Each Raylet has a store for distributed objects, which is hinted at by the `object_store_address` above. Once tasks are scheduled, they get executed by worker processes. In Chapter 2 you'll get a much better understanding of all these components and how they make up a Ray cluster.

Before moving on, we should also briefly mention that 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.

Right now your Ray cluster doesn't do much, but that's about to change. After giving you a quick introduction to the data science workflow in the following section, you'll run your first concrete Ray examples.

A Suite of Data Science Libraries

Moving on to the second layer of Ray, in this section we'll 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. If you have a good idea of the data science process, you can safely skip ahead to section "Data Processing with Ray Data" on page 16.

Machine Learning 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.[5] To me, it's *the practice of gaining insights and building real-world applications by leveraging data*. That's quite a broad definition, and you don't have to agree with me. My point is that data science is an inherently practical and applied field that centers around building and understanding things, which makes fairly little sense in a *purely* academic context. In that sense, describing practitioners of this field as "data scientists" is about as bad of a misnomer as describing hackers as "computer scientists".[6]

Since you are familiar with Python and hopefully bring a certain craftsmanship attitude with you, we can approach the Ray's data science libraries from a very pragmatic angle. Doing data science in practice is an iterative process that goes something like this:

- Requirements engineering: You talk to stakeholders to identify the problems you need to solve and clarify the requirements for this project.
- Data collection: Then you source, collect and inspect the data.
- Data processing: Afterwards you process the data such that you can tackle the problem.

[5] I 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*. It doesn't do a cook justice to tell them they sit at the intersection of agriculture, thermodynamics and human relations. It's not wrong, but also not very helpful.

[6] 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?

- Model building: You then move on to build a model (in the broadest sense) using the data. That could be a dashboard with important metrics, a visualisation, or a machine learning model, among many other things.
- Model evaluation: The next step is to evaluate your model against the requirements in the first step.
- Deployment: If all goes well (it likely doesn't), you deploy your solution in a production environment. You should understand this as an ongoing process that needs to be monitored, not as a one-off step.
- Otherwise you need to circle back and start from the top. The most likely outcome is that you need to improve your solution in various ways, even after initial deployment.

Machine learning is not necessarily part of this process, but you can see how building smart applications or gaining insights might benefit from ML. Building a face detection app into your social media platform, for better or worse, might be one example of that. When the data science process just described explicitly involves building machine learning models, 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 specialised software packages for ML model serving.

This list is by no means exhaustive. Don't worry if you've never gone through these steps or struggle with the terminology, we'll come back to this in much more detail in later chapters. 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.

Figure Figure 1-2 gives an overview of the steps we just discussed:

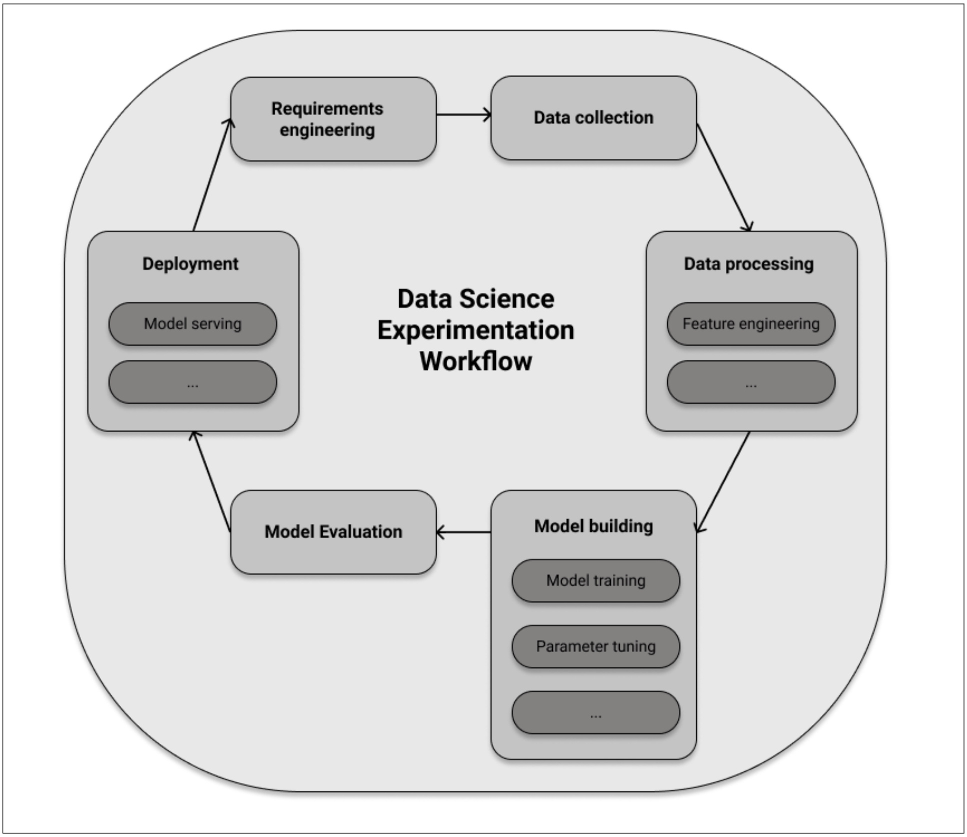

Figure 1-2. An overview of the data science experimentation workflow using machine learning.

At this point you might be wondering how any of this relates to Ray. The good news is that Ray has a dedicated library for each of the four ML-specific tasks above, covering data processing, model training, hyperparameter tuning and model serving. And the way Ray is designed, all these libraries are *distributed by construction*. Let's walk through each of them one-by-one.

Data Processing with Ray Data

The first high-level library of Ray we talk about is called "Ray Data". 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.

To use Ray Data, 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:

Example 1-2.

```
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 10 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 distributed rows, but what can you do with that data? The `Dataset` API bets heavily on functional programming, as it is very 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, let's see how easily you can transform your dataset `ds`:

Example 1-3. Transforming a `Dataset` with common functional programming routines.

```
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 example Example 1-3 this 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.

Example 1-4.

```
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 Data, especially its integration with notable data processing systems, but we'll have to defer an in-depth discussion until ???.

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 TensorFlow for consistency. Go ahead and install it with `pip install tensorflow` right now.

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 with `pip`. 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.

We'll look at a fairly classical control problem of balancing a pendulum. Imagine you have a pendulum like the one in figure Figure 1-3, fixed at as single point and subject to gravity. You can manipulate that pendulum by giving it a push from the left or the right. If you assert just the right amount of force, the pendulum might remain in an upright position. That's our goal - and the question is whether we can teach a reinforcement learning algorithm to do so for us.

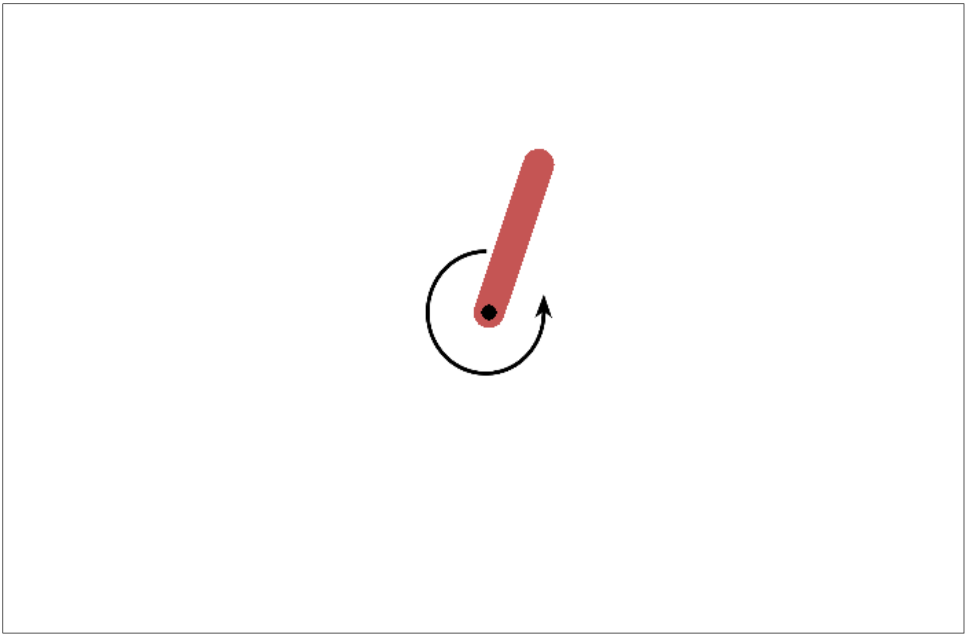

Figure 1-3. Controlling a simple pendulum by asserting force to the left or the right.

Specifically, we want to train a reinforcement learning agent that can push to the left or right, thereby acting on its environment (manipulating the pendulum) to reach the "upright position" goal for which it will be rewarded. To tackle this problem with Ray RLlib, store the following content in a file called `pendulum.yml`.

Example 1-5.

```
# pendulum.yml
pendulumppo:
    env: Pendulum-v1  ❶
    run: PPO  ❷
    checkpoint_freq: 5  ❸
    stop:
        episode_reward_mean: 800  ❹
    config:
        lambda: 0.1  ❺
        gamma: 0.95
        lr: 0.0003
        num_sgd_iter: 6
```

❶ The `Pendulum-v1` environment simulates the pendulum problem we just described.

❷ We use a powerful RL algorithm called Proximal Policy Optimization, or PPO.

❸ After every five "training iterations" we checkpoint a model.

❹ Once we reach a reward of `-800` , we stop the experiment.

❺ 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, don't get distracted by them. The important part is that you specify the built-in `Pendulum-v1` environment and sufficient RL-specific configuration to ensure the training procedure works. The configuration is a simplified version of one of Ray's tuned examples. We chose this one because it doesn't require any special hardware and finishes in a matter of minutes. If your computer is powerful enough, you can try to run the tuned example as well, which should yield much better results. To train this pendulum example you can now simply run:

```
rllib train -f pendulum.yml
```

If you want, you can check the output of this Ray program and see how the different metrics evolve during the training procedure. In case you don't want to create this file on your own, and want to run an experiment which gives you much better results, you can also run this:

```
curl https://raw.githubusercontent.com/maxpumperla/learning_ray/main/notebooks/pendulum.yml -o per
rllib train -f pendulum.yml
```

In any case, assuming the training program finished, we can now check how well it worked. To visualize the trained pendulum you need to install one more Python library with `pip install pyglet`. The only other thing you need to figure out is where Ray stored your training progress. When you run `rllib train` for an experiment, Ray will create a unique experiment ID for you and stores results in a subfolder of `~/ray-results` by default. For the training configuration we used, you should see a folder with results that looks like `~/ray_results/pendulum-ppo/PPO_Pendulum-v1_<experiment_id>`. During the training procedure intermediate model checkpoints get generated in the same folder. For instance, I have a folder on my machine called:

```
~/ray_results/pendulum-ppo/PPO_Pendulum-v1_20cbf_00000_0_2021-09-24_15-20-03/checkpoint_000029/ch
```

Once you figured out the experiment ID and chose a checkpoint ID (as a rule of thumb the larger the ID, the better the results), you can evaluate the training performance of your pendulum training run like this:

```
rllib evaluate \
  ~/ray_results/pendulum-ppo/PPO_Pendulum-v1_<experiment_id>/checkpoint_0000<cp-id>/checkpoint-<cp
  --run PPO --env Pendulum-v1 --steps 2000
```

You should see an animation of a pendulum controlled by an agent that looks like figure Figure 1-3. Since we opted for a quick training procedure instead of maximizing performance, you should see the agent struggle with the pendulum exercise. We could have done much better, and if you're interested to scan Ray's tuned examples for the `Pendulum-v1` environment, you'll find an abundance of solutions to this exercise. The point of this example was to show you how simple it can be to train and evaluate reinforcement learning tasks with RLlib, using just two command line calls to `rllib`.

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 concrete and informative example for Ray Train. We'll discuss all of that in Chapter 5, when it's time to. But we can at least roughly sketch what a distributed training "wrapper" for an ML model would look like, which is simple enough conceptually:

Example 1-6.

```
from ray.train import Trainer
```

```
def training_function():  ❶
    pass


trainer = Trainer(backend="tensorflow", num_workers=4)  ❷
trainer.start()

results = trainer.run(training_function)  ❸
trainer.shutdown()
```

❶ First, define your ML model training function. We simply pass here.

❷ Then initialize a `Trainer` instance with TensorFlow as the backend.

❸ Lastly, scale out your training function on a Ray cluster.

If you're interested in distributed training, you could jump ahead to Chapter 5.

Hyperparameter Tuning

Naming things is hard, but the Ray team hit the spot with *Ray Tune*, which you can use to tune all sorts of parameters. Specifically, it was built 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).

Example 1-7. Minimizing an objective for an expensive training function with Ray Tune.

```
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 5 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.

The Tune example in Example 1-7 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. 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 ???.

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 now 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. 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:

Example 1-8.

```
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
```

```python
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 ??? 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. 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, I also don't want to give off the impression that Ray is all you need from now on. In fact, I believe 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. There's usually no reason to re-invent the wheel.

How Ray Integrates and Extends

To give you an example for how Ray integrates with other tools, consider that Ray Data is a relatively new addition to its libraries. If you want to boil it down, and maybe oversimplify a little, Ray is a compute-first framework. In contrast, distributed frameworks like Apache Spark[7] or Dask can be considered data-first. Pretty much anything you do with Spark starts with the definition of a distributed dataset and transformations thereof. Dask bets on bringing common data structures like Pandas dataframes or Numpy arrays to a distributed setup. Both are immensely powerful in their own regard, and we'll give you a more detailed and fair comparison to Ray in ???. The gist of it is that Ray Data does not attempt to replace these tools. Instead, it integrates well with both. As you'll come to see, that's a common theme with Ray.

Ray as Distributed Interface

One aspect of Ray that's vastly understated in my eyes is that its libraries seamlessly integrate common tools as *backends*. Ray often creates common interfaces, instead of trying to create new standards[8]. 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. 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. Ray Serve can be used with frameworks such as FastAPI, and Ray Data is backed by Arrow and comes with many integrations to other frameworks,

[7] 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.

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

such as Spark and Dask. Overall this seems to be a robust design pattern that can be used to extend current Ray projects or integrate new backends in the future.

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. For practical data science workflows you can use Ray Data 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. On top of that, Ray's ecosystem has many extensions that we'll look more into later on. Maybe you can already spot a few tools you know and like in Figure 1-4[9]?

[9] Note that "Ray Train" has been called "raysgd" in older versions of Ray, and does not have a new logo yet.

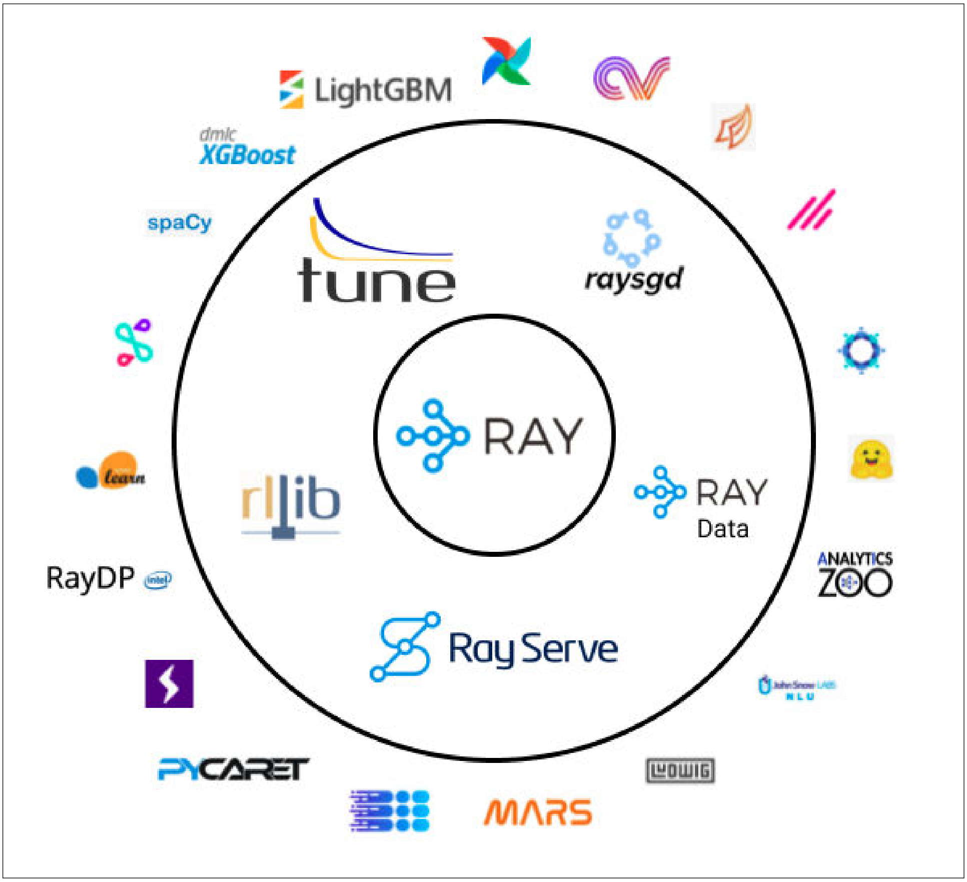

Figure 1-4. Ray in three layers. Its core API sits at the center, surrounded by the libraries RLlib, Tune, Ray Train, Ray Serve, Ray Data and the many third-party integrations we can't all list here.

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 which makes it difficult to, for example, 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. For instance, libraries like `multiprocessing` can help you distribute work on a single machine, but not beyond.

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 about all the fundamental concepts underlying Ray and what its architecture looks like. In other words, 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 ??? we've introduced you to the very basics of Ray clusters and showed you how start a local cluster simply by typing

Example 2-1.

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

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.

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`.

Example 2-2.

```
import time

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


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, from `database[0]` for “Learning” to `database[7]` for “Science”. 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:

Example 2-3.

```
def print_runtime(input_data, start_time, decimals=1):
    print(f'Runtime: {time.time() - start_time:.{decimals}f} 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: 2.8 seconds, data:
(0, 'Learning')
(1, 'Ray')
(2, 'Flexible')
(3, 'Distributed')
(4, 'Python')
(5, 'for')
(6, 'Data')
(7, 'Science')
```

We cut off the output of the program after one decimal number. There’s a little overhead that brings the total closer 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 have at least 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[1] is undoubtedly one of the most infamous features of the Python language. 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:

Example 2-4.

```
@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`.

[1] 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, if you feel insecure.

In this way, the function retrieve_task becomes a so-called Ray task. 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 data and measures performance? It turns out, not much. Let's have a look at how you'd do that:

Example 2-5. Measuring performance of your Ray task.

```
start = time.time()
data_references = [retrieve_task.remote(item) for item in range(8)]  ❶
data = ray.get(data_references)  ❷
print_runtime(data, start, 2)
```

❶ To run retrieve_task on your local Ray cluster, you use .remote() and pass in your data as before. You'll get a list of object references.

❷ To get back 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 the remote function. When tasks get executed remotely, even on your local cluster, Ray does so *asynchronously*. The list items in data_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(data_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.

We still want to work more on this example[2], 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

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

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 ??? 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. My laptop has a total of 8 CPU cores, so Ray will create 8 worker processes on my one-node local cluster. 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 data between the driver and workers (or between workers). You can simply use `put` to place your data into Ray's *distributed object store* and then use `get` on the workers to retrieve it as follows.

Example 2-6.

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


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

❶ `put` your `database` into the object store and receive a reference to it.

❷ This allows your workers to `get` the data, no matter where they are located in the cluster.

By using the object store this way, you can let Ray handle data access across the whole cluster. We'll talk about how exactly data is passed between nodes and within workers when talking about Ray's infrastructure. While the interaction with the object store requires some overhead, Ray is really smart about storing the data, which 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 2-5 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 2-5 we used `ray.get(data_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 data 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 data to be processed. One more question to keep in mind is what happens if one of the data 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, we should not wait longer than 10 times the longest data retrieval task before stopping the task. Here's how you can do that with Ray by using `wait`:

Example 2-7.

```
start = time.time()
data_references = [retrieve_task.remote(item) for item in range(8)]
all_data = []

while len(data_references) > 0:  ❶
    finished, data_references = ray.wait(data_references, num_returns=2, timeout=7.0)  ❷
    data = ray.get(finished)
    print_runtime(data, start, 3)  ❸
    all_data.extend(data)  ❹
```

❶ Instead of blocking, we loop through unfinished `data_references`.

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

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

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

As you can see `ray.wait` returns two arguments, namely finished data 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 data items is available. On my laptop this results in the following output:

```
Runtime: 0.108 seconds, data:
(0, 'Learning')
(1, 'Ray')
Runtime: 0.308 seconds, data:
```

```
(2, 'Flexible')
(3, 'Distributed')
Runtime: 0.508 seconds, data:
(4, 'Python')
(5, 'for')
Runtime: 0.709 seconds, data:
(6, 'Data')
(7, 'Science')
```

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 data 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 data. 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 2-8. 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) 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, 'Data'), (7, 'Science'))
```

If you don't have a lot of experience with asynchronous programming, you might not be impressed by Example 2-8. But I hope to convince you that it's at least a bit surprising[3] that this code snippet runs at all. 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. 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.

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 I 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.

[3] 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.

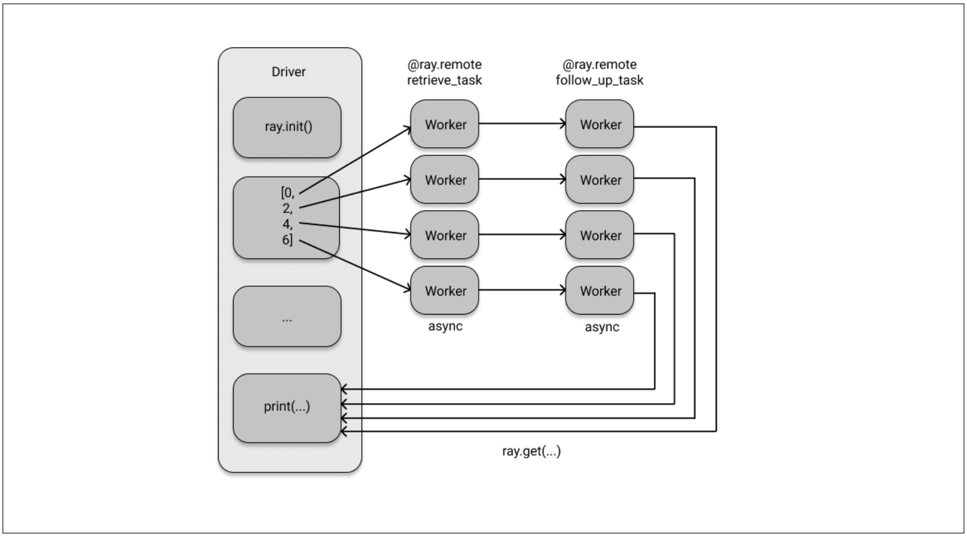

Figure 2-1. Running two dependent tasks asynchronously and in parallel with Ray.

If you feel like it, try to rewrite Example 2-8 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[4]. 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.

Example 2-9.

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

[4] 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.

```
        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.

Example 2-10.

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


tracker = DataTracker.remote()  ❸

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

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[5]. 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 data into Ray's object store and `ray.get()` to retrieve data 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[6]. 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 Ray Core.

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 data into Ray's object store.
`ray.get()`	Gets data from the object store. Returns data you've `put` there or that was 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 quickly discuss Ray's design philosophy, before moving on to discussing its system architecture.

[5] 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.

[6] 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.

Design Principles

Ray is built with several design principles in mind, most of which you've got a taste of already. Its API is designed for simplicity and generality. Its compute model banks on flexibility. And its system architecture is designed for performance and scalability. Let's look at each of these in more detail.

Simplicity & Abstraction

As you've seen, Ray's API does not only bank on simplicity, it's also intuitive to pick up. 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

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 *heterogenity* 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. In the example we discussed in this chapter you've seen that Ray's API is built for that.

You also need to ensure you are flexible in your resource usage. 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 & Scalability

Another of Ray's design principles is the speed at which Ray executes its heterogeneous tasks. It can handle millions of tasks per second. What's more is that you only incur very low latencies with Ray. It's build 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. 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.[7] In any case, Ray is built to recover quickly from failures, which contributes to its overall speed.

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 data 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.

[7] 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, I'm afraid Ray can't help you.

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 really 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. An object store has *shared memory* across the node, so that each worker process has easy access to it. 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 data 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 4 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 and the amount of memory available on its node, but you can register custom resources, if you want to. If it can't provide the required resources, it can't schedule execution of a task.

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 input data it needs to execute a task. For that to work, the scheduler will first resolve local dependencies by looking up data in its object store. If the required data 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. I 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 very tricky very quickly.

Now that you know about Raylets, let's briefly come back to worker processes, so that we can wrap up the discussion around worker nodes. An important concept that contributed to the performance or Ray overall is that of *ownership*.

Ownership means that a process that runs something is responsible for it. This makes for a decentralized overall design, since individual tasks have a unique owner. In concrete terms this means that each worker process owns the tasks it submits, which includes proper execution and availability of results (i.e., correct resolution of object references). Also, anything that gets registered through `ray.put()` is owned by the caller. You should understand ownership in contrast to dependency, which we've already covered by example when discussing task dependencies.

To give you a concrete example, let's say we have a program that starts a `task` which takes an input value `val` and internally calls another task. That could look as follows:

Example 2-11.

```
@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)
```

From this point on we won't mention it again, but this example assumes that you have a running Ray cluster started with `ray.init()`. 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 `rew_owned`. Our main program defines both `val` (which puts `"value"` into the object store) and `res`, the final result of th whole program, 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.

Ownership is important to know about, but it's not a concept you encounter all that often when working with Ray. The reason we brought it up in this context is that worker processes need to track what they own. In fact, they possess a so-called *ownership table* exactly for that reason. If a task fails and needs to be recomputed, the worker already owns all the information it needs to do so. On top of that, workers also have an in-process store for small objects, which has a default limit of 100KB. Workers have that store so that small data can be directly accessed and stored without incurring communication overhead with the Raylet object store, which is reserved for large objects.

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

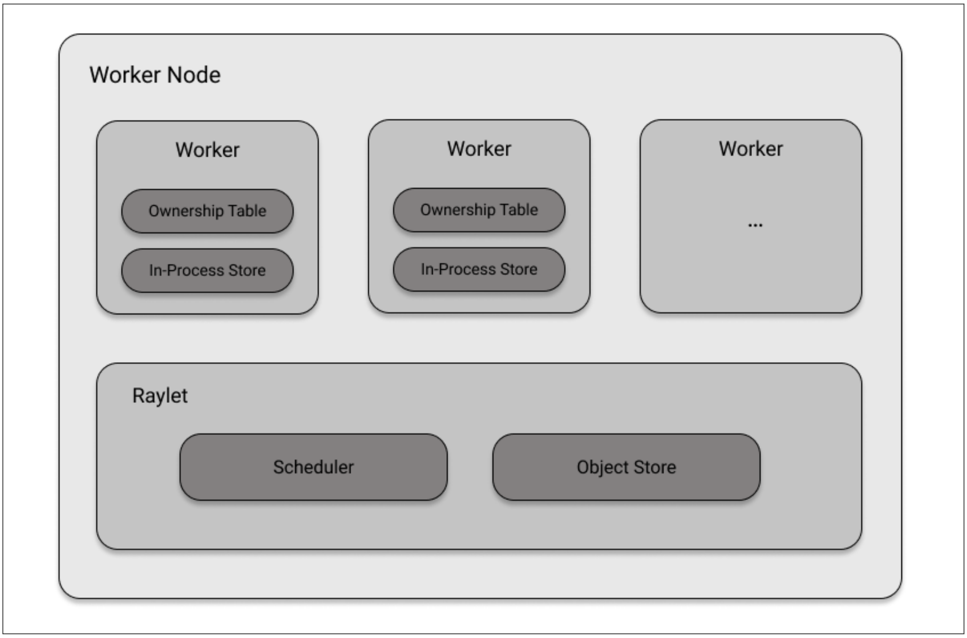

Figure 2-2. The system components comprising a Ray worker node.

The Head Node

We've already indicated in ??? that each Ray cluster has one special node called head node. So far you know that this is the node that has the driver process[8]. 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 constisting of a single node. In other words, the head node has everything a worker node has (including a Raylet), but it also has a driver process.

Additionally, the head node comes with a component called *Global Control Store* (GCS). The GCS is a key-value store currently implemented in Redis. It's an important component that carries global information about the cluster, 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 and large objects in respective tables, and knows about the dependencies between objects.

[8] In fact, it could have multiple drivers, but this is inessential for our discussion.

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 share their memory on a node. But sometimes data needs to be transferred between nodes, which is called *distributed object transfer*. This is needed for remote dependency resolution, so that workers have the data they need to run tasks.
- Communication: Most of the communication in a Ray cluster, such as object transfer, takes place via the *gRPC* protocol.
- 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. 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 tradeoff 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:

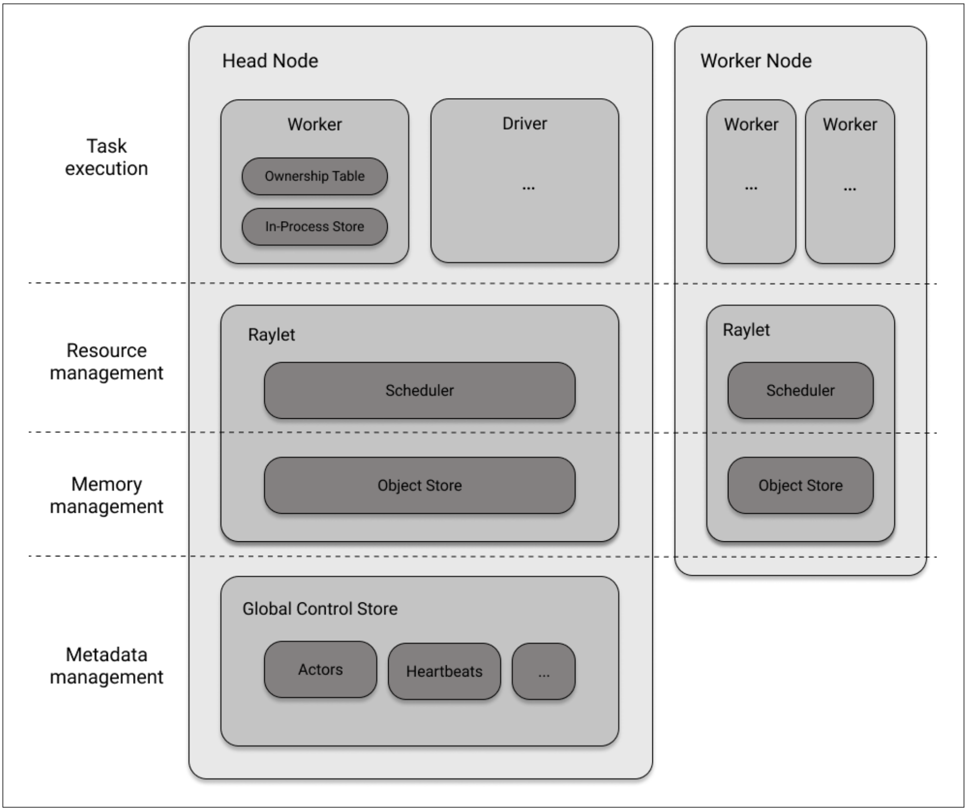

Figure 2-3. An overview of Ray's architectural components.

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 ???, 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 ???. 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).

Summary

You've seen the basics of the Ray API in action in this chapter. You know how to `put` data to the object store, and how to `get` it 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. 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.

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 comparatively short 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.

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 ???, and to see a fully deployed Ray application you can go to ???.

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.

Example 3-1.

```
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`.

❷ For instance, a `Discrete(4)` sample will give you `0`, `1`, `2`, or `3`.

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:

Example 3-2.

```
import os


class Environment:

    seeker, goal = (0, 0), (4, 4)  ❶
    info = {'seeker': seeker, 'goal': goal}

    def __init__(self,  *args, **kwargs):
        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.

Example 3-3.

```
    def reset(self):  ❶
        """Reset seeker and goal positions, return observations."""
        self.seeker = (0, 0)
        self.goal = (4, 4)

        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:

Example 3-4.

```
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")

    return self.get_observation(), self.get_reward(), self.is_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.

Example 3-5.

```
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:

Example 3-6.

```
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*.[1] 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

[1] 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.

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.

Example 3-7.

```
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])
```

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.

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:

Example 3-8.

```
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:

Example 3-9.

```
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:

Example 3-10.

```
import numpy as np


def update_policy(policy, experiences):
    """Updates a given policy with a list of (state, action, reward, state) experiences."""
    weight = 0.1
    discount_factor = 0.9
    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.

Example 3-11.

```
def train_policy(env, num_episodes=10000):
    """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)  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:

Example 3-12.

```
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.")


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

Let's take a step back here. If you're an RL expert, you'll know what we've been doing the whole time. If you're completely new to RL, you might just be a little overwhelmed. If you're somewhere in between, you hopefully like the example but might be wondering how what we've done so far relates to Ray. That's 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 create a Ray task that can initialize a `Policy` remotely.
- Then we make the `Simulation` a Ray actor in just a few lines of code.
- After that we wrap the `update_policy` function in a Ray task.
- Finally, we define a parallel version of `train_policy` that's structurally identical to its original version.

Let's tackle the first two steps of this plan by implementing a `create_policy` task and a Ray actor called `SimulationActor`:

Example 3-13.

```
import ray

ray.init()
environment = Environment()
env_ref = ray.put(environment)  ❶


@ray.remote
```

```
def create_policy():
    env = ray.get(env_ref)
    return Policy(env)  ❷


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

❶ After initializing it, we put our `environment` into the Ray object store.

❷ This remote task returns a new `Policy` object.

❸ 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 distributed `update_policy_task` Ray task and then wrap everything (two tasks and one actor) in a `train_policy_parallel` function that distributes this RL workload on your local Ray cluster:

Example 3-14.

```
@ray.remote
def update_policy_task(policy_ref, experiences_list):
    """Remote Ray task for updating a policy with experiences in parallel."""
    [update_policy(policy_ref, ray.get(xp)) for xp in experiences_list]  ❶
    return policy_ref


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

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

    return ray.get(policy)  ❻
```

❶ This task defers to the original `update_policy` function by passing a reference to a policy and experiences retrieved from the object store.

❷ To train in parallel, we first create a policy remotely, which returns a reference we call `policy`.

❸ Instead of one simulation, we create four simulation actors.

❹ Experiences now get collected from remote roll-outs on simulation actors.

❺ Then we can update our policy remotely. Note that `experiences` is a nested list of experiences.

❻ Finally, we return the trained policy by retrieving it from the object store again.

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

Example 3-15.

```
parallel_policy = train_policy_parallel()
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 three times in a suitable way. 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 task execution graph of the Ray application that we've just built. To recap, what we did was:

- The `train_policy_parallel` function creates several `SimulationActor` actors and a policy with `create_policy`
- The simulation actors create roll-outs with the policy and thereby collect experiences that `update_policy_task` uses to update the policy.
- This works, because of the way updating the policy is designed. It doesn't matter if the experiences were collected by one or multiple simulations.
- The rolling out and updating continues until we reached the number of episodes we wante to train for, then the final `trained_policy` is returned.

Figure Figure 3-1 summarizes this task graph in a compact way:

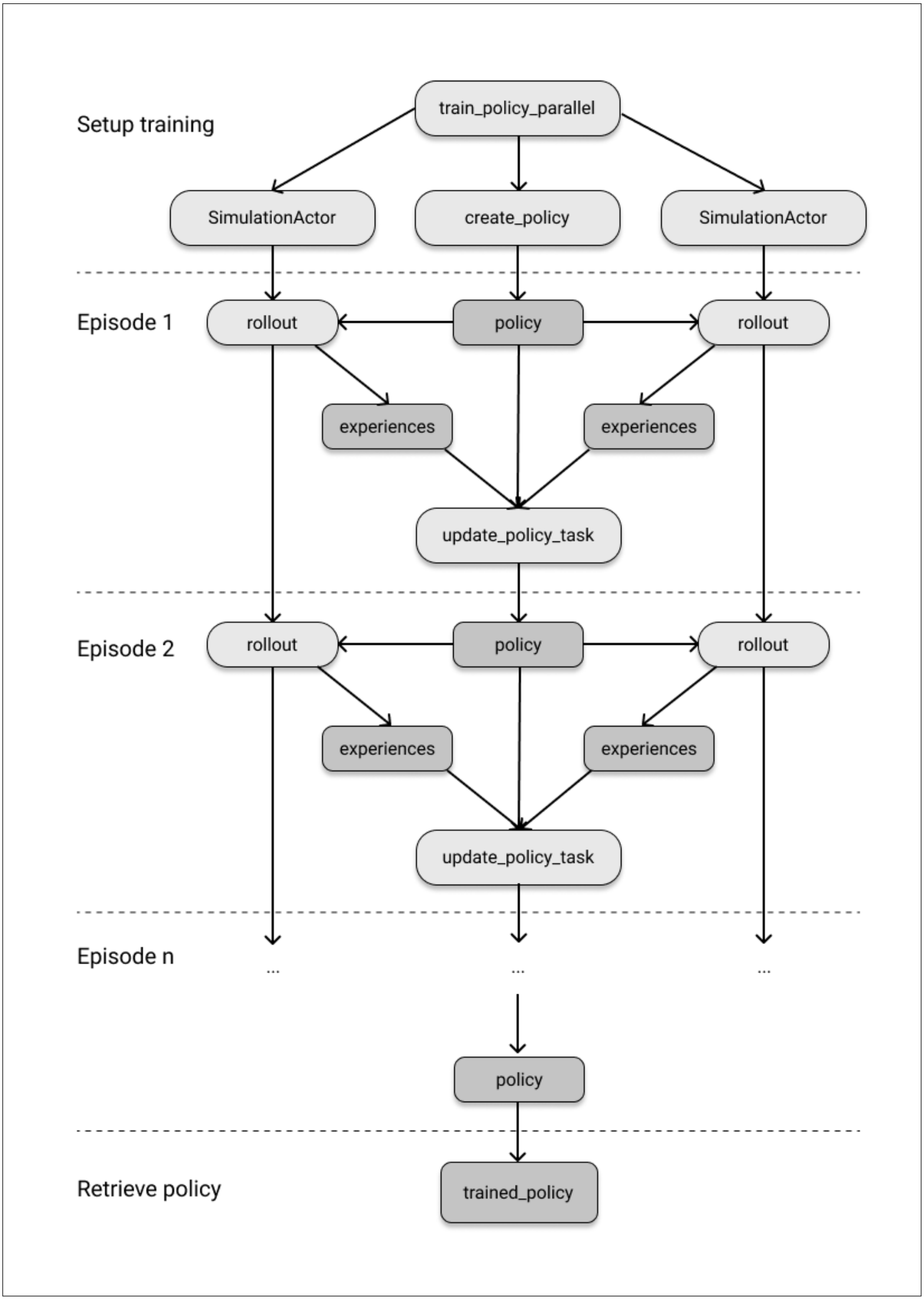

Figure 3-1. Parallel training of a reinforcement learning policy with Ray.

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.

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 pendulum example from ??? 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 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.

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"[1] you want to play. 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 algorithm.

Ray RLlib is an industry-grade library for building RL algorithms at scale. You've seen a first example of the RLlib in ??? 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 and 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

[1] We simply used a simple game to illustrate the process of RL. There is a multitude of interesting industry applications of RL that are not games.

of code. You'll see how easy RLlib is to get started with before learning about its key concepts, such as RLlib environments, algorithms, and trainers.

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 learning 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 some examples, let's take a quick overview of 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. Other RL libraries can potentially scale out experiments, but it's usually not straightforward to do so.

Another benefit of being built on top of Ray is that RLlib integrates tightly with other Ray libraries. For instance, all RLlib algorithms can be tuned with Ray Tune, as we will see in ???, and you can seamlessly deploy your RLlib models with Ray Serve, as we will discuss in ???.

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. I often recommend RLlib to engineers, because its API tends to appeal to them. One of the reasons for that is that the RLlib API offers the right level of abstraction for many applications, while still being flexible enough to be extended, if necessary.

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, so 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]"==1.9.0
```

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 ??? we already looked at the classical pendulum balancing problem. Recall that we didn't implement this pendulum 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 for environments. Your 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 make our maze `Environment` from the last chapter 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 `gym` environments.

In short, the interface of a gym environment looks like the following pseudo-code:

```
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 current observations

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 `Discrete` space and the `Environment` from Chapter 3 is either located at the top of that file (or is imported there).

```
# Original definition of `Environment` and `Discrete` go here.

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 override 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.

Notably, the `gym.Env` interface also comes with helpful utility functionality and many interesting example implementations. For instance, the `Pendulum-v1` environment we used in ??? is an example from `gym`, 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 ??? before, but this time the situation is a bit different. In the first chapter we simply referenced the Pendulum-v1 environment from by *name* in a YAML file, along with other RL training configuration. 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, i.e. in our case 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 YAML 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 DQN as the algorithm for our training run. Also, to make sure we can control the time of training, we set an explicit stop condition, namely by setting timesteps_total to 10000.

```
# maze.yml
maze_env:
    env: maze_gym_env.GymEnvironment  ❶
    run: DQN
    checkpoint_freq: 1 ❷
    stop:
        timesteps_total: 10000  ❸
```

❶ We specify the relative Python path to our environment class here.

❷ We store checkpoints of our model after each training iteration.

❸ We can also specify a stopping condition for training, here a maximum of 10000 steps.

Assuming you store this configuration in a file called maze.yml you can now kick off an RLlib training run by running the following train command:

```
rllib train -f maze.yml
```

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 (DQN), takes care of scaling out to multiple workers under the hood, and 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 logdir directory located at ~/ray_results/maze_env. Within that folder you'll find another directory that starts with DQN_maze_gym_env.GymEnvironment_ and contains both an identifier for this experiment (0ae8d in my case) and the current date and time. Within that directory you should find several other subdirectories starting with a checkpoint prefix. For the training run on my computer there are a total of 10 checkpoints available and we're using the last one (checkpoint_000010/checkpoint-10) to evaluate our trained RLlib algorithm with it. With the folders and checkpoints generated on my machine the rllib evaluate command you can use reads as follows (adapt the checkpoint path to what you see on your machine):

```
rllib evaluate ~/ray_results/maze_env/DQN_maze_gym_env.Environment_0ae8d_00000_
0_2022-02-08_13-52-59/checkpoint_000010/checkpoint-10\
  --run DQN\
  --env maze_gym_env.Environment\
  --steps 100
```

The algorithm used in --run 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 of RLlib, it should be noted that the train and evaluate CLI commands can come in handy even for more complex environments. The YAML configuration can take any parameter the Python API would, so in that sense there is no limit for training your experiments on the command line[2].

Using the RLlib Python API

Having said that, you will likely spend most of your time coding your reinforcement learning experiments in Python. In the end the RLlib CLI is merely a wrapper around the underlying Python library that we're going to look at now.

To run RL workloads with RLlib from Python, your main entrypoint is that of the Trainer class. Specifically, for the algorithm of your choice you want to use the corresponding Trainer of it. In our case, since we decided to use Deep Q-Learning (DQN) for demonstration purposes, we'll use the DQNTrainer class.

[2] We should mention that the RLlib CLI uses Ray Tune under the hood, among many other things for checkpointing models. You will learn more about this integration in ???

Training RLlib models

RLlib has good defaults for all its `Trainer` implementations, meaning that you can initialize them without having to tweak any configuration parameters for these trainers[3]. For instance, to generate a DQN trainer you can simply use `DQNTrainer(env=GymEnvironment)`. That said, it's worth noting that RLlib trainers are highly configurable, as you will see in the following example. Specifically, we pass a `config` dictionary to the `Trainer` constructor and tell it to use four workers in total. What that means is that the `DQNTrainer` will spawn four Ray actors, each using a CPU kernel, to train our DQN algorithm in parallel.

After you've initialized your trainer with the `env` you want to train on, and pass in the `config` you want, you can simply call the `train` method. Let's use this method to train the algorithm for ten iterations in total:

```
from ray.tune.logger import pretty_print
from maze_gym_env import GymEnvironment
from ray.rllib.agents.dqn import DQNTrainer

trainer = DQNTrainer(env=GymEnvironment, config={"num_workers": 4})  ❶

config = trainer.get_config()  ❷
print(pretty_print(config))

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

print(pretty_print(result))  ❹
```

❶ We use the `DQNTrainer` from RLlib to use Deep-Q-Networks (DQN) for training, using 4 parallel workers (Ray actors).

❷ Each `Trainer` has a complex default configuration.

❸ We can then simply call the `train` method to train the agent 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.

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

From printing the config dictionary, you can verify that the num_workers parameter is set to 4[4]. Similarly, If you run this training script, the result contains detailed information about the state of the Trainer and the training results that's too verbose to put 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:

```
...
episode_reward_max: 1.0
episode_reward_mean: 1.0
episode_reward_min: 1.0
episodes_this_iter: 15
episodes_total: 19
...
timesteps_total: 10000
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 model 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 your RLlib trainers at any point in the training process by calling trainer.save(). Once you have a checkpoint, you can easily restore your Trainer with it. And evaluating a model is as simple as calling trainer.evaluate(checkpoint) with the checkpoint you created. Here's how that looks like if you put it all together:

```
checkpoint = trainer.save()  ❶
print(checkpoint)

evaluation = trainer.evaluate(checkpoint)  ❷
print(pretty_print(evaluation))
```

[4] If you set num_workers to 0, only the local worker on the head node will be created, and all training is done there. This is particularly useful for debugging, as no additional Ray actor processes are spawned.

```
restored_trainer = DQNTrainer(env=GymEnvironment)
restored_trainer.restore(checkpoint)  ❸
```

❶ You can save trainers to create checkpoints.

❷ RLlib trainers can be evaluated at your checkpoints.

❸ And you can also restore any Trainer from a given checkpoint.

I should mention that you can also just call `trainer.evaluate()` without creating a checkpoint first, but it's usually good practice to use checkpoints anyway. Looking at the output, 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 trainers have much more functionality than just the `train`, `evaluate`, `save` and `restore` 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 = trainer.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 = trainer.compute_actions({"obs_1": observations, "obs_2": observations})
print(action)
# {'obs_1': 0, 'obs_2': 1}
```

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 we're 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 that case an action distribution is a vector of four probabilities, one for each action.

To make things concrete, let's have a look at how you access policies and models in RLlib:

```
policy = trainer.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 four models that we trained on separate Ray workers, we can access all the workers we used in training - and then ask each worker's policy for their weights like this:

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

In this way, you can access every method available on a `Trainer` 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 DQNTrainer, 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 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.

```
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[5].

State-action values and state-value functions

So far we've been concerned with the concept of state-action values a lot, since this concept takes center stage in the formulation of Q-Learning, which we've been using extensively in this and the last chapter.

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

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 and passing them to the model (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.from_batch({"obs": transformed})  ❸
```

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

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

❸ You get the model output by using the from_batch method of the model on a preprocessed observation dictionary.

Having computed our model_output, we can now both access the Q-values, as well as 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 your `Trainer` takes a `config` argument, which so far we've only used to set the number of Ray workers to 4.

If you want to alter the behaviour of your RLlib training run, the way to do this is to change the `config` argument of your `Trainer`. This is at the same time relatively simple, as you can add configuration properties quickly, and a bit tricky, as you have to know which key-words the `config` dictionary expects. Finding and tweaking the right configuration properties becomes easier once you have a good grasp of what's available and what to expect.

RLlib configuration splits in two parts, namely algorithm-specific and common configuration. We've used `DQN` as our algorithm in the examples so far, which has certain properties that are only available to this choice[6]. Algorithm-specific configuration only becomes more relevant once you've settled on an algorithm and want to squeeze it for performance, but in practice RLlib provides you with good defaults to get started. You can look up configuration arguments in the API reference for RLlib algorithms.

The common configuration of algorithms can be further split into the following types.

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:

- `num_gpus`: 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

[6] For the experts, our DQNs are dueling double Q-learning models via the `"dueling": True` and `"double_q": True` default arguments, for example.

fractional. For example, if using four rollout workers in DQN (num_workers = 4), you can set num_gpus=0.25 to pack all four workers on the same GPU, so that all trainers benefit from the potential speedup.

- num_cpus_per_worker: Set the number of CPUs to use for each worker.

Debugging and Logging Configuration

Debugging your applications is crucial for any project, and machine learning is no exception. RLlib allows you to configure the way it logs information and how you can access it.

- log_level: Set the level of logging to use. This can be either DEBUG, INFO, WARN, or ERROR and defaults to WARN. You should experiment with the different levels to see what suits your needs best in practice.
- callbacks: You can specify custom *callback functions* to be called at various points during training. We will take a closer look at this topic in ???.
- ignore_worker_failures: For testing it might be useful to ignore worker failures by setting this property to True (defaults to False).
- logger_config: You can specify a custom logger configuration, passed in as a nested dictionary.

Rollout Worker and Evaluation Configuration

Of course, you can also specify how many workers are used for rollouts during training and evaluation.

- num_workers: You've seen this one already. It's used to specify the number of Ray workers to use.
- num_envs_per_worker: 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.
- create_env_on_driver: If you've set num_workers at least to 1, then the driver process does not need to create an environment, since there are rollout workers for that. If you set this property to True you create an additional environment on the driver.
- explore: Set to True by default, this property allows you to turn off exploration, for instance during evaluation of your algorithms.
- evaluation_num_workers: Specify the number of parallel evaluation workers to use, which defaults to 0.

Environment Configuration

- env: 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. There's also a way to *register* your environments so that you can refer to them by name, but this requires using Ray Tune. We will learn about this feature in ???.
- observation_space and action_space: You can specify the observation and action spaces of your environment. If you don't specify them, they will be inferred from the environment.
- env_config: You can optionally specify a dictionary of configuration options for your environment that will be passed to the environment constructor.
- render_env: 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, there's a class of common configuration options to modify the behavior of the RL training procedure, like modifying the underlying model to use. These properties are the most important in a sense, while at the same time require the most specific knowledge of reinforcement learning. For this introduction to RLlib, we can't go into any more details. But the good news is that if you're a regular user of RL software, you will have no trouble identifying the relevant training configuration options.

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

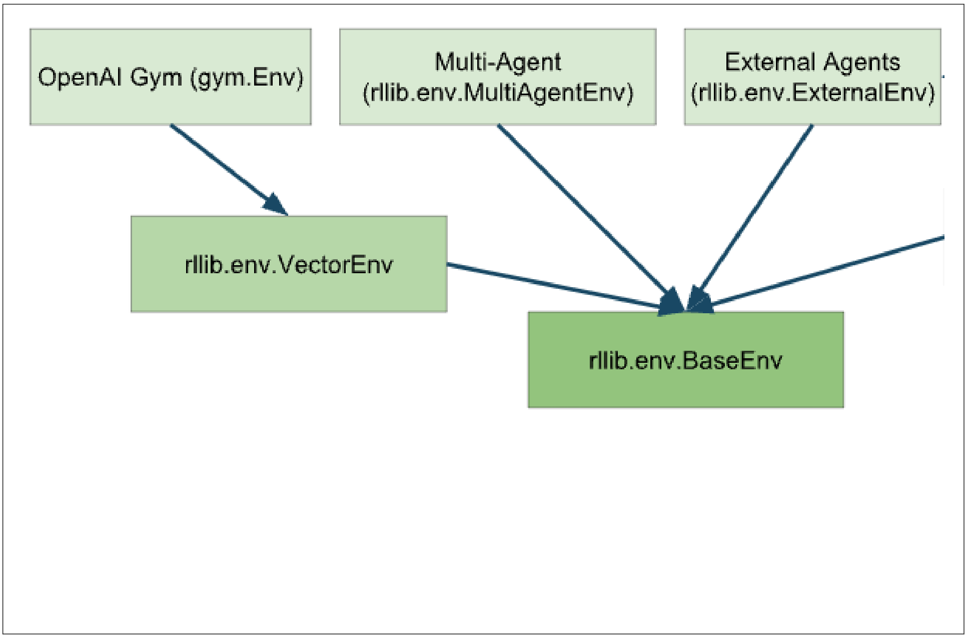
OpenAI Gym (gym.Env)
Multi-Agent (rllib.env.MultiAgentEnv)
External Agents (rllib.env.ExternalEnv)
rllib.env.VectorEnv
rllib.env.BaseEnv

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 define as a dictionary with values for each agent, and each agent has its unique key. Of course, the details are a little more complicated than that in practice. But once you have defined an environment hosting several agents, what's necessary is 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 Figure 4-2:

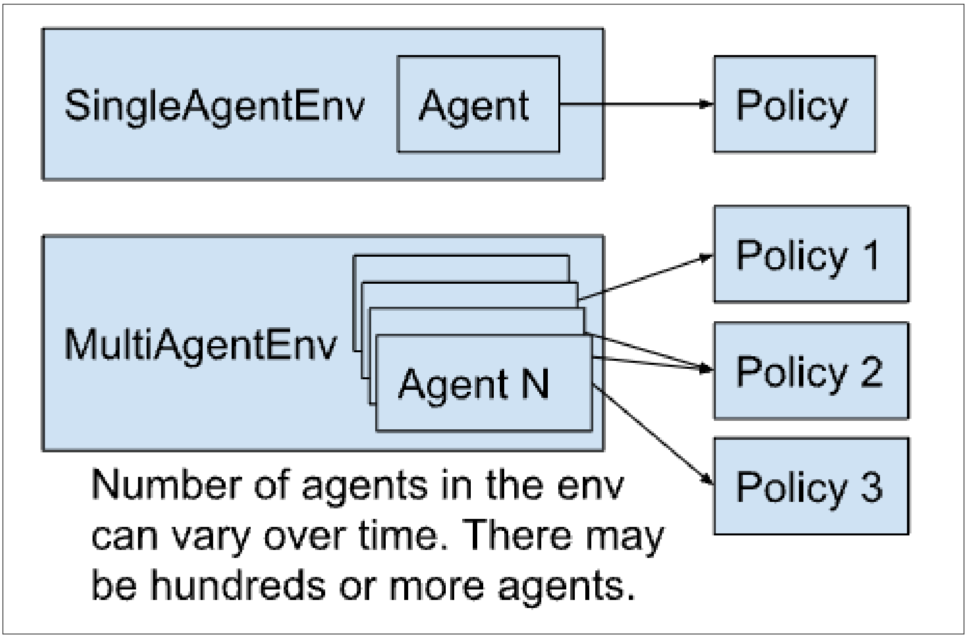
SingleAgentEnv
Agent
Policy
MultiAgentEnv
Agent N
Policy 1
Policy 2
Policy 3
Number of agents in the env can vary over time. There may be hundreds or more agents.

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 be 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):
```

```
agents = {1: (4, 0), 2: (0, 4)}  ❶
goal = (4, 4)
info = {1: {'obs': agents[1]}, 2: {'obs': agents[2]}}  ❷

def __init__(self,  *args, **kwargs):  ❸
    self.action_space = Discrete(4)
    self.observation_space = Discrete(5*5)

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

    return {1: self.get_observation(1), 2: self.get_observation(2)}  ❹
```

❶ We now have two seekers with `(0, 4)` and `(4, 0)` starting positions in an `agents` dictionary.

❷ For the `info` object we're using agent IDs as keys.

❸ Action and observation spaces stay exactly the same as before.

❹ Observations are now per-agent dictionaries.

Notice that compared to the single-agent situation we had to modify neither action nor observation spaces, since we're using two essentially identical agents here that can 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.

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):  ❶❷
    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
```

❶ Getting a specific agent from its ID.

❷ Redefining each helper method to work per-agent.

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[7].

```
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'
```

[7] 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.

```
        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:

```
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 `Trainer` works *exactly* the same way as before.

```
from ray.rllib.agents.dqn import DQNTrainer

simple_trainer = DQNTrainer(env=MultiAgentMaze)
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 modifying the `multiagent` dictionary in our trainer `config` as follows:

Example 4-1.

```
trainer = DQNTrainer(env=MultiAgentMaze, config={
    "multiagent": {
        "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
    },
})
```

```
print(trainer.train())
```

❶ We first define multiple policies for our agents.

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 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 `config` object and trains a `Trainer`.

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 trainer configuration, we can define yet another `DQNTrainer` to run on the server:

```
# policy_server.py
import ray
from ray.rllib.agents.dqn import DQNTrainer
from ray.rllib.env.policy_server_input import PolicyServerInput
import gym


ray.init()


def policy_input(context):
    return PolicyServerInput(context, "localhost", 9900)  ❶


config = {
    "env": None,  ❷
    "observation_space": gym.spaces.Discrete(5*5),
    "action_space": gym.spaces.Discrete(4),
    "input": policy_input,  ❸
    "num_workers": 0,
    "input_evaluation": [],
    "log_level": "INFO",
}

trainer = DQNTrainer(config=config)
```

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

With this `trainer` defined [8], 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 = trainer.train()
        checkpoint = trainer.save()  ❶
        if time_steps >= 10.000:  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 10.000 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

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

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
            obs = env.reset()

            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. 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 trainer 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 a `Trainer` with a *curriculum* to learn from. What that means is that, instead of letting the trainer 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 on 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 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):
        return abs(self.seeker[0] - self.goal[0]) + abs(self.seeker[1] - self.goal[1])

    def get_task(self):
        return self.difficulty()

    def set_task(self, task_difficulty):
        while not self.difficulty() <= task_difficulty:
            self.reset()
```

To use this environment for curriculum learning, we need to define a curriculum function that tells the trainer 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 trainer config, namely by setting the env_task_fn property to our curriculum_fn. Note that before training a DQNTrainer 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.

```
config = {
    "env": CurriculumEnv,
    "env_task_fn": curriculum_fn,
    "output": "/tmp/env-out",
}

from ray.rllib.agents.dqn import DQNTrainer

trainer = DQNTrainer(env=CurriculumEnv, config=config)

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

Running this trainer, you should see how the task difficulty increases over time, thereby giving the trainer 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[9].

To use the data we stored in the `/tmp/env-out` folder before, we can create a new training configuration that takes this folder as `input`. Note how we set `exploration` to `False` in the following configuration, since we simply want to exploit the data previously collected for training - the algorithm will not explore according to its own policy.

```
input_config = {
    "input": "/tmp/env-out",
    "input_evaluation": [],
    "explore": False
}
```

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

Using this `input_config` for training works exactly as before, which we demonstrate by training an agent for `10` iterations and evaluating it:

```
imitation_trainer = DQNTrainer(env=AdvancedEnv, config=input_config)
for i in range(10):
    imitation_trainer.train()

imitation_trainer.evaluate()
```

Note that we called the trainer `imitation_trainer`. 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 `DQNTrainer` 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 trainers 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.

CHAPTER 5

Distributed Training with Ray Train

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 previous chapters, you've learned how to build and scale reinforcement learning applications with Ray, and how to optimize hyperparameters for such applications. As we indicated in ???, Ray also comes with the Ray Train library, which provides an extensive suite of machine learning training integrations and allows them to scale seamlessly.

We will start this chapter by providing context about why you might need to scale out your machine learning training. Then we'll cover some key concepts you need to know in order to use Ray Train. Finally, we'll cover some of the more advanced functionality that Ray Train provides.

As always, you can follow along 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 a couple common problems with training machine learning models causing you to investigate distributed training solutions:

1. The time it takes to finish training is too long.

2. The size of data is too large to fit into one machine.
3. 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, the first two problems often arise much earlier in the journey to machine learning practitioners. 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.

Introduction to Ray Train

Ray Train is a library for distributed 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 typical ML training pipeline you will use the following key components of Ray Train:

[1] This applies specifically to the gradient computation in neural networks.

- **Preprocessors**: Ray Train provides several common preprocessor objects and utilities to process dataset objects into consumable features for Trainers.
- **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, providing integration with core Ray actors (for distribution), Tune, and Datasets.
- **Models**: Each trainer can produce a model. The model can be used in serving.

Let's see how to put these concepts in practice by computing a first example with Ray Train.

Creating an end-to-end example for Ray Train

In the below example, we demonstrate the ability to load, process, and train a machine learning model using Ray Train.

We'll be using a simple dataset for this example, using the `load_breast_cancer` function from the scikit-learn `datasets` package[2]. We load the data into a Pandas DataFrame first and then convert it into a so-called Ray `Dataset`. ??? is entirely devoted to the Ray Data library, we just use it here to illustrate the Ray Train API.

Example 5-1.

```
from ray.data import from_pandas
import sklearn.datasets


data_raw = sklearn.datasets.load_breast_cancer(as_frame=True)  ❶

dataset_df = data_raw["data"]
predict_ds = from_pandas(dataset_df)  ❷

dataset_df["target"] = data_raw["target"]
dataset = from_pandas(dataset_df)
```

❶ Load breast cancer data into a Pandas DataFrame.

❷ Create a Ray Dataset from the DataFrame.

Next, let's specify a preprocessing function. In this case, we'll be using three key preprocessors: a `Scaler`, a `Repartitioner`, and a `Chain` object to chain the first two.

[2] Ray can handle much larger datasets than that. In ??? we'll take a closer look at the Ray Data library to see how to handle huge datasets.

Example 5-2.

```
preprocessor = Chain(  ❶
    Scaler(["worst radius", "worst area"]),  ❷
    Repartitioner(num_partitions=2)  ❸
)
```

❶ Create a pre-processing `Chain`.

❷ Scale two specific data columns.

❸ Repartition the data into two partitions.

Our entrypoint for doing distributed training is the Trainer object. There are specific Trainers for different frameworks, and each are configured with some framework-specific parameters.

To give you an example, let's have a look at the `XGBoostTrainer` class, which implements distributed training for XGBoost.

Example 5-3.

```
trainer = XGBoostTrainer(
    scaling_config={  ❶
        "num_actors": 2,
        "gpus_per_actor": 0,
        "cpus_per_actor": 2,
    },
    label="target",  ❷
    params={  ❸
        "tree_method": "approx",
        "objective": "binary:logistic",
        "eval_metric": ["logloss", "error"],
    },
)

result = trainer.fit(dataset=dataset, preprocessor=preprocessor)  ❹

print(result)
```

❶ Specify the scaling configuration.

❷ Set the label column.

❸ Specify XGBoost-specific parameters.

❹ Train the model by calling `fit`.

Preprocessors in Ray Train

The Preprocessor is the core class 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.

Currently, Ray Train offers the following encoders

FunctionTransformer	Custom Transformers
Pipeline	Sequential Preprocessing
StandardScaler	Standardization
MinMaxScaler	Standardization
OrdinalEncoder	Encoding Categorical Features
OneHotEncoder	Encoding Categorical Features
SimpleImputer	Missing Value Imputation
LabelEncoder	Label Encoding

You will often want to make sure you can use the same data preprocessing operations at training time and at serving time.

Usage of preprocessors

You can use these preprocessors by passing them to a trainer. Ray Train will take care of applying the preprocessor to the dataset in a distributed fashion.

Example 5-4.

```
result = trainer.fit(dataset=dataset, preprocessor=preprocessor)
```

Serialization of preprocessors

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.

Example 5-5.

```
pickle.dumps(prep) ❶
```

❶ We can serialize and save preprocessors.

Trainers in Ray Train

Trainers are framework-specific classes that run model training in a distributed fashion. Trainers all share a common interface:

fit(self)	Fit this trainer with the given dataset.
get_checkpoints(self)	Return list of recent model checkpoints.
as_trainable(self)	Get a wrapper of this as a Tune trainable class.

Ray Train supports a variety of different trainers on a variety of frameworks, namely XGBoost, LightGBM, Pytorch, HuggingFace, Tensorflow, Horovod, Scikit-learn, RLlib, and more.

Next, we'll dive into two specific classes of Trainers: Gradient Boosted Tree Framework Trainers, and Deep Learning Framework Trainers.

Distributed Training for Gradient Boosted Trees

Ray Train offers Trainers for LightGBM and XGBoost.

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 (also known as GBDT, GBM) 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.

By leveraging Ray Train's XGBoost or LightGBM trainer, you can take a large dataset and train a XGBoost Booster across multiple machines.

Distributed Training for Deep Learning

Ray Train offers Deep Learning Trainers, for instance supporting frameworks such as Tensorflow, Horovod, and Pytorch.

Unlike Gradient Boosted Trees Trainers, these Deep Learning frameworks often give more control to the user. For example, Pytorch provides a set of primitives that the user can use to construct their training loop.

As such, the Deep Learning Trainer API allows the user to pass in a training function and provides callback functions for the user to report metrics and checkpoint. Let's take a look at an example Pytorch training script.

Below, we construct a standard training function:

Example 5-6.

```
import torch
import torch.nn as nn


num_samples = 20
input_size = 10
layer_size = 15
output_size = 5


class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.layer1 = nn.Linear(input_size, layer_size)
        self.relu = nn.ReLU()
        self.layer2 = nn.Linear(layer_size, output_size)

    def forward(self, input_data):
        return self.layer2(self.relu(self.layer1(input_data)))


input = torch.randn(num_samples, input_size)  ❶
labels = torch.randn(num_samples, output_size)


def train_func():  ❷
    num_epochs = 3
    model = NeuralNetwork()
    loss_fn = nn.MSELoss()
    optimizer = optim.SGD(model.parameters(), lr=0.1)
    for epoch in range(num_epochs):
        output = model(input)
        loss = loss_fn(output, labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

❶ In this example we use a randomly generated dataset.

❷ Define a training function.

We construct a training script for Pytorch, where we create a small neural network and use a Mean Squared Error (`MSELoss`) objective to optimize the model. The input to the model here is random noise, but you can imagine that to be generated from a Torch Dataset.

Now, there are two key things that Ray Train will take care of for you.

1. The establishment of a backend that coordinates interprocess communication.
2. Instantiation of multiple parallel processes.

So, in short, you just need to make a one-line change to your code:

Example 5-7.

```
import ray.train.torch


def train_func_distributed():
    num_epochs = 3
    model = NeuralNetwork()
    model = train.torch.prepare_model(model)  ❶
    loss_fn = nn.MSELoss()
    optimizer = optim.SGD(model.parameters(), lr=0.1)
    for epoch in range(num_epochs):
        output = model(input)
        loss = loss_fn(output, labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

❶ Prepare the model for distributed training.

Then, you can plug this into Ray Train:

Example 5-8.

```
from ray.train import Trainer


trainer = Trainer(backend="torch", num_workers=4, use_gpu=False)  ❶
trainer.start()
```

```
results = trainer.run(train_func_distributed)
trainer.shutdown()
```

❶ Create a trainer. For GPU Training, set `use_gpu` to True.

This code will work on both a single machine or a distributed cluster.

Scaling out training with Ray Train Trainers

The general philosphy of Ray Train is that the user should not need to think about **how** to parallelize their code.

With Ray Train Trainers, you can specify a `scaling_config` which allows you to scale out your training without writing distributed logic. The `scaling_config` allows you to declaratively specify the *compute resources* used by a Trainer.

In particular, you can specify the amount of parallelism that the Trainer should use by providing the number of workers, along with the type of device that each worker should use:

Example 5-9.

```
scaling_config = {"num_workers": 10, "use_gpu": True}

trainer = ray.train.integrations.XGBoostTrainer(
    scaling_config=scaling_config,
    # ...
)
```

Note that scaling configuration arguments depend on the Trainer type.

The nice thing about this specification is that you don't need to think about the underlying hardware. In particular, you can specify to use hundreds of workers and Ray Train will automatically leverage all the nodes within your Ray cluster:

Example 5-10.

```
# Connect to a large Ray cluster
ray.init(address="auto")

scaling_config = {"num_workers": 200, "use_gpu": True}

trainer = ray.train.integrations.XGBoostTrainer(
    scaling_config=scaling_config,
    # ...
)
```

Connecting Data to Distributed Training

Ray Train provides utilities to train on large datasets.

Along the same philosophy that the user should not need to think about **how** to parallelize their 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.

Here, we create a dataset from random data. However, you can use other data APIs to read a large amount of data (with `read_parquet`, which reads data from the Parquet format).

Example 5-11.

```
from typing import Dict
import torch
import torch.nn as nn

import ray
import ray.train as train
from ray.train import Trainer


def get_datasets(a=5, b=10, size=1000, split=0.8):

    def get_dataset(a, b, size):
        items = [i / size for i in range(size)]
        dataset = ray.data.from_items([{"x": x, "y": a * x + b} for x in items])
        return dataset

    dataset = get_dataset(a, b, size)
    split_index = int(dataset.count() * split)
    train_dataset, validation_dataset = dataset.random_shuffle().split_at_indices(
        [split_index]
    )
    train_dataset_pipeline = train_dataset.repeat().random_shuffle_each_window()
    validation_dataset_pipeline = validation_dataset.repeat()
    datasets = {
        "train": train_dataset_pipeline,
        "validation": validation_dataset_pipeline,
    }
    return datasets
```

You can then specify a training function that accesses these datapoints. Note that we use a specific `get_dataset_shard` function here. Under the hood, Ray Train will automatically shard the provided dataset so that individual workers can train on a different subset of the data at once. This avoids training on duplicate data within the same epoch. The `get_dataset_shard` function passes a subset of the data from the data source to each individual parallel training worker.

Next, we make a `iter_epochs` and `to_torch` call on each shard. `iter_epochs` will produce an iterator. This iterator will produce Dataset objects that will possess 1 shard of the entire epoch (named `train_dataset_iterator` and `validation_dataset_iterator`).

`to_torch` will convert the Dataset object into a Pytorch iterator. There is an equivalent `to_tf` function that converts it to a Tensorflow Data iterator.

When the epoch is finished, the Pytorch iterator will raise a `StopIteration`, and the `train_dataset_iterator` will be queried again for a new shard on a new epoch.

Example 5-12.

```
def train_func(config):
    batch_size = config["batch_size"]
    # hidden_size = config["hidden_size"]
    # lr = config.get("lr", 1e-2)
    epochs = config.get("epochs", 3)

    train_dataset_pipeline_shard = train.get_dataset_shard("train")
    validation_dataset_pipeline_shard = train.get_dataset_shard("validation")
    train_dataset_iterator = train_dataset_pipeline_shard.iter_epochs()
    validation_dataset_iterator = validation_dataset_pipeline_shard.iter_epochs()

    for _ in range(epochs):
        train_dataset = next(train_dataset_iterator)
        validation_dataset = next(validation_dataset_iterator)
        train_torch_dataset = train_dataset.to_torch(
            label_column="y",
            feature_columns=["x"],
            label_column_dtype=torch.float,
            feature_column_dtypes=torch.float,
            batch_size=batch_size,
        )
        validation_torch_dataset = validation_dataset.to_torch(
            label_column="y",
            feature_columns=["x"],
            label_column_dtype=torch.float,
            feature_column_dtypes=torch.float,
            batch_size=batch_size,
        )
        # ... training

    return results
```

You can put things together by using the Trainer in the following way:

Example 5-13.

```
num_workers = 4
use_gpu = False


datasets = get_datasets()
trainer = Trainer("torch", num_workers=num_workers, use_gpu=use_gpu)
config = {"lr": 1e-2, "hidden_size": 1, "batch_size": 4, "epochs": 3}
trainer.start()
results = trainer.run(
   train_func,
   config,
   dataset=datasets,
   callbacks=[JsonLoggerCallback(), TBXLoggerCallback()],
)
trainer.shutdown()
print(results)
```

Ray Train Features

Checkpoints

Ray Train will generate **model checkpoints** to checkpoint intermediate state for training. These model checkpoints provide the trained model and fitted preprocessor for usage in downstream applications like serving and inference.

Example 5-14.

```
result = trainer.fit()
model: ray.train.Model = result.checkpoint.load_model()
```

The goal of the model checkpoints is to abstract away the actual physical representation of the model and preprocessor. As a result, you should be able to generate a checkpoint from a cloud storage location, and convert it into an in-memory representation or on-disk representation, and vice versa.

Example 5-15.

```
chkpt = Checkpoint.from_directory(dir)
chkpt.to_bytes() -> bytes
```

Callbacks

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 (the values passed into `train.report(...)`).

Ray Train contains built-in callbacks for popular tracking frameworks, or you can implement your own callback via the TrainingCallback interface. Available callbacks include:

- Json Logging
- Tensorboard Logging
- MLflow Logging
- Torch Profiler

Example 5-16.

```
# Run the training function, logging all the intermediate results
# to MLflow and Tensorboard.

result = trainer.run(
    train_func,
    callbacks=[
        MLflowLoggerCallback(experiment_name="train_experiment"),
        TBXLoggerCallback(),
    ],
)
```

Integration 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.

Example 5-17.

```
from ray import tune

fail_after_finished = True
prep_v1 = preprocessor
prep_v2 = preprocessor

param_space = {
    "scaling_config": {
        "num_actors": tune.grid_search([2, 4]),
        "cpus_per_actor": 2,
        "gpus_per_actor": 0,
    },
    "preprocessor": tune.grid_search([prep_v1, prep_v2]),
    # "datasets": {
    #     "train_dataset": tune.grid_search([dataset_v1, dataset_v2]),
    # },
    "params": {
        "objective": "binary:logistic",
        "tree_method": "approx",
        "eval_metric": ["logloss", "error"],
        "eta": tune.loguniform(1e-4, 1e-1),
        "subsample": tune.uniform(0.5, 1.0),
        "max_depth": tune.randint(1, 9),
    },
}
if fail_after_finished > 0:
    callbacks = [StopperCallback(fail_after_finished=fail_after_finished)]
else:
    callbacks = None
tuner = tune.Tuner(
    XGBoostTrainer(
        run_config={"max_actor_restarts": 1},
        scaling_config=None,
        resume_from_checkpoint=None,
        label="target",
    ),
    run_config={},
    param_space=param_space,
    name="tuner_resume",
    callbacks=callbacks,
)
results = tuner.fit(datasets={"train_dataset": dataset})
print(results.results)
```

Compared to other distributed hyperparameter tuning solutions, Ray Tune and Ray Train has a couple unique features:

- Ability to specify the dataset and preprocessor as a parameter
- Fault tolerance
- Ability to adjust the number of workers during training time.

Exporting Models

You may want to export the model trained to Ray Serve or a model registry after you've trained it with Ray Train.

To do so, you can fetch the model using a load_model API:

Example 5-18.

```
result = trainer.fit(dataset=dataset, preprocessor=preprocessor)
print(result)

this_checkpoint = result.checkpoint
this_model = this_checkpoint.load_model()
predicted = this_model.predict(predict_ds)
print(predicted.to_pandas())
```

Some Caveats

In particular, recall that standard neural network training works by iterating through a dataset in separate batches of data (usually called minibatch gradient descent).

To speed this up, you can parallelize the gradient computation of every minibatch update. This means that the batch should be split across multiple machines.

One complication is that if you hold the size of the batch constant, the system utilization and efficiency reduces as you increase the number of workers.

To compensate, practitioners typically increase the amount of data per batch.

As a result, the time it takes to go through a single pass of the data (one epoch) should ideally reduce, since the number of total batches decreases.

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