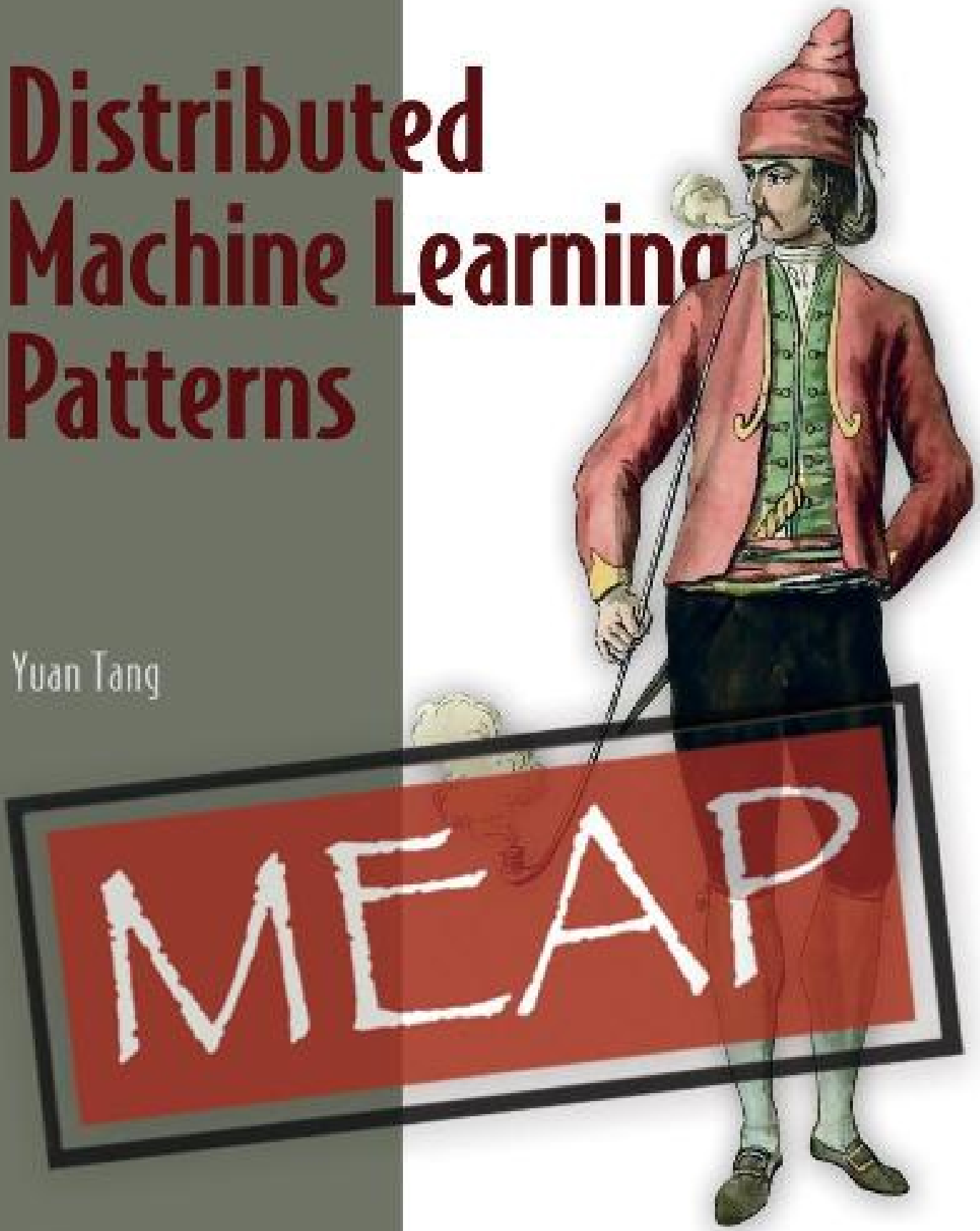
Distributed Machine Learning Patterns

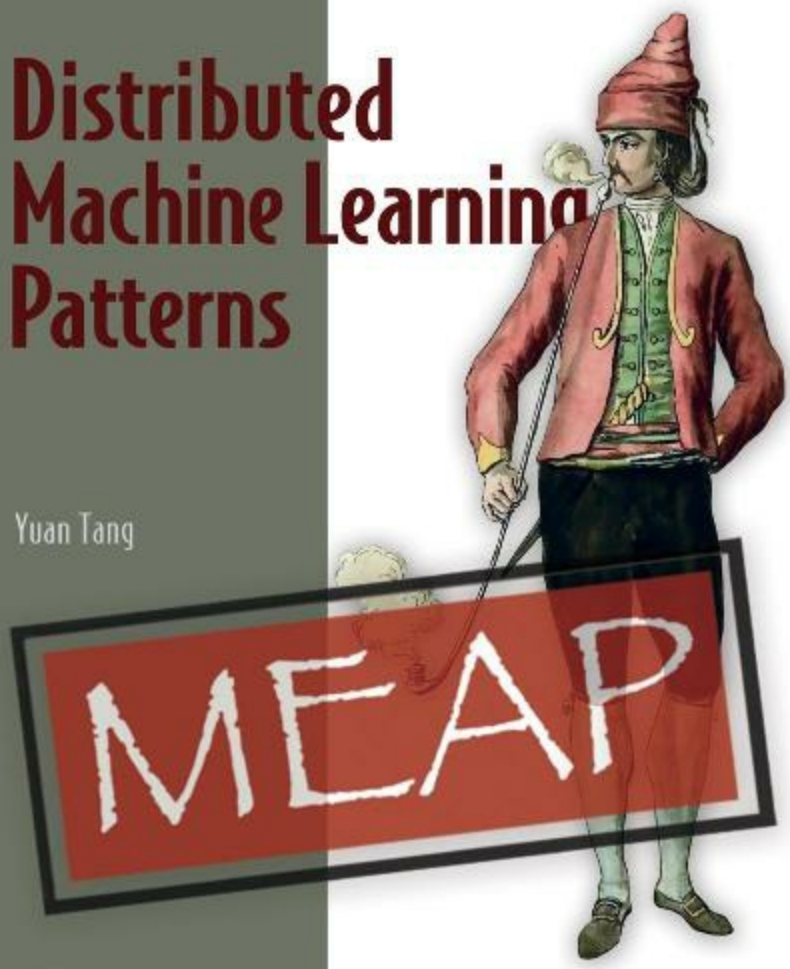
Yuan Tang



MANNING

Distributed Machine Learning Patterns

Yuan Tang



MANNING

Distributed Machine Learning Patterns MEAP V07

MEAP Edition

Manning Early Access Program

Distributed Machine Learning Patterns

Version 7

Copyright 2023 Manning Publications

https://livebook.manning.com/#!/book/distributed-machine-learning-patterns/discussion

For more information on this and other Manning titles go to

manning.com

welcome

Thank you for purchasing the MEAP of *Distributed Machine Learning Patterns*. I hope that what you'll get access to will be of immediate use to you and, with your help, the final book will be great!

In recent years, advances in machine learning have made tremendous progress yet large scale machine learning still remains challenging. With the variety of machine learning frameworks such as TensorFlow, PyTorch, and XGBoost, it's not easy to automate the process of training machine learning models on distributed clusters. Machine learning researchers and algorithm engineers with less or zero DevOps experience cannot easily launch, manage, and monitor distributed machine learning pipelines.

Being able to handle large scale problems and take what's developed on your personal laptop to large distributed clusters is exciting. This book introduces key concepts and practical patterns commonly seen in good distributed machine learning systems. These patterns would greatly speed up the development and deployment of machine learning models, leverage automations from different tools, and benefit from hardware accelerations.

After reading this book, you will be able to: choose and apply the correct patterns for building and deploying distributed machine learning systems; use common tooling, such as TensorFlow, Kubernetes, Kubeflow, and Argo Workflows appropriately within a machine learning pipeline; and gain practical experience for managing and automating machine learning tasks in Kubernetes.

There will be supplemental exercises at the end of each section in part two of the book to recap what we've learned. In addition, there will be one comprehensive hands-on project at the last part of the book that provides an opportunity to build a real-world distributed machine learning system that leverages many of the patterns we will learn in the second part of the book. Stay tuned for the final publication!

Please let me know your thoughts and suggestions in the liveBook Discussion Forum on what's been written so far and what you'd like to see in the rest of the book. Your feedback will be invaluable in and highly appreciated.

Thanks again for your interest and for purchasing the MEAP!

Yuan Tang

https://terrytangyuan.github.io/about/

In this book

1 Introduction to distributed machine learning systems

This chapter covers

- Handling the growing scale in large scale machine learning applications
- Establishing patterns to build scalable and reliable distributed systems
- Leveraging patterns in distributed systems and building reusable patterns that could accelerate distributed machine learning systems in a more scalable and reliable way

Machine learning systems are becoming more and more important nowadays: recommendation systems learn to generate recommendations of potential interest with the right context according to user feedback and interactions; anomalous event detection systems help monitor assets to avoid downtime due to extreme conditions; fraud detection systems protect financial institutions from security attacks and malicious fraud behaviors.

There are increasing demands on building large scale distributed machine learning systems. If a data analyst, data scientist, or software engineer has basic knowledge and hands-on experience in building machine learning models in Python and wants to take a step further to learn how to build something more robust, scalable, and reliable, then this is the right book to read. While experience in production environments or distributed systems is not a requirement, we expect readers in this position to have at least some exposure to machine learning applications running in production and should have written Python and Bash scripts for at least one year.

Being able to handle large scale problems and take what’s developed on your personal laptop to large distributed clusters is exciting. This book introduces best practices in various patterns that speed up the development and deployment of machine learning models, leverage automations from different tools, and benefit from hardware accelerations. After reading this book, the reader will be able to: choose and apply the correct patterns for building and

deploying distributed machine learning systems; use common tooling, such as TensorFlow, Kubernetes, Kubeflow, and Argo Workflows appropriately within a ML workflow; and gain practical experience for managing and automating machine learning tasks in Kubernetes. A comprehensive, hands-on project in the last chapter provides an opportunity to build a real-life distributed machine learning system that leverages many of the patterns we learn in the second part of the book. In addition, there are supplemental exercises at the end of some sections in each following chapter to recap what we've learned.

1.1 Large scale machine learning

The scale of machine learning applications has become unprecedentedly large. For example, users demand for faster responses to meet real-life requirements and machine learning pipelines and model architectures get more complex. Next we'll talk about the growing scale in more detail and what we can do to address the challenges.

1.1.1 The growing scale

As the demand for machine learning grows, the complexity involved in building machine learning systems are increasing as well. Machine learning researchers and data analysts are no longer satisfied to build simple machine learning models on their laptops on gigabytes of excel sheets.

Due to the growing demand and complexity, machine learning systems have to be built with the ability to handle the growing scale: the increasing volume of historical data, highly frequent batches of incoming data, complex machine learning architectures, heavy model-serving traffic, complicated end-to-end machine learning pipeline etc.

Let's consider two scenarios. First, imagine that you have a small machine learning model that has been trained on a small dataset (e.g. less than 1GB). This approach might work well for your analysis at hand since you have a laptop with sufficient computational resources. However, you realize that the size of the dataset grows by 1GB every hour and the original model is no longer useful and predictive in real-life. For example, you want to build a

time-series model that predicts whether a component of a train would fail in the next hour in order to prevent failures and avoid downtime. In this case, we have to build a machine learning model that has the prior knowledge gained from the original data as well as the most recent data that arrives every hour in order to generate more accurate predictions. Unfortunately, your laptop at hand has a fixed amount of computational resources that's no longer sufficient to build a new model using the entire dataset.

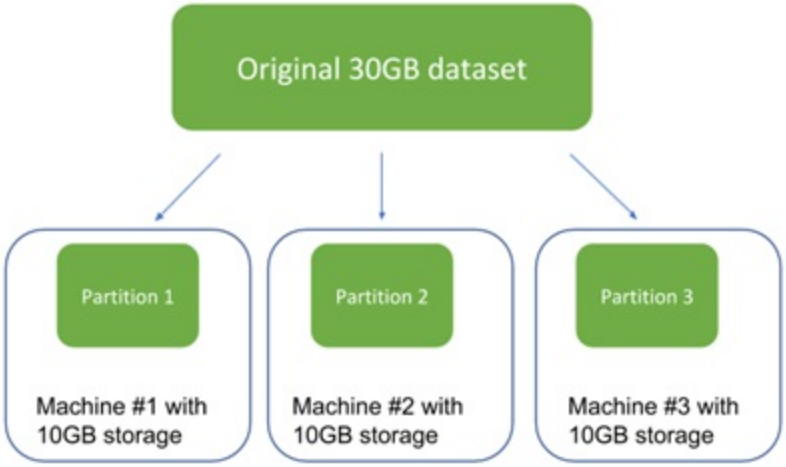
Second, suppose that now you have successfully trained a model and developed a simple web application that uses the trained model to make predictions based on the user's input. The web application may work well at the beginning. For example, it generated accurate predictions and the user was quite happy with the results. This user's friends heard about the good experience and decided to try it out as well so they sat in the same room and opened the website. Ironically, they started seeing longer and longer delays when they tried to see the prediction results. The reason behind the delays is that the single server that is used to run the web application can no longer handle the increasing number of user requests as this application gets more and more popular. This is a very common challenge that many machine learning applications will encounter as they grow from beta products to popular applications. These applications need to be built upon scalable machine learning system patterns in order to handle the growing scale of throughput.

1.1.2 What can we do?

In situations where the dataset is too large to fit in a single machine such as the first scenario in the previous section, how can we store the large dataset? Perhaps we can store different parts of the dataset on different machines and then we train the machine learning model by sequentially looping through the various parts of the dataset on different machines.

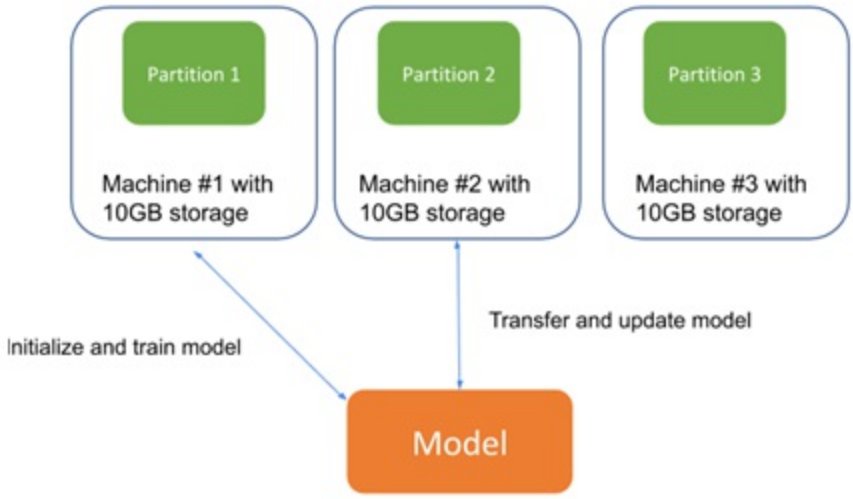
For example, if we have a 30GB dataset at hand like in Figure 1.1, we can divide it to three partitions of 10GB data where each partition sits in a separate machine with enough disk storage. We can then consume each of the partitions one by one without having to train the machine learning model using the entire dataset at once.

Figure 1.1 Example of dividing a large dataset to three partitions on three separate machines with sufficient disk storage.



Then we might ask: what if looping through different parts of the dataset is quite time-consuming? For example, let's assume that the dataset at hand has been divided into three partitions. As illustrated in Figure 1.2, we first initialize the machine learning model on the first machine and then train it using all the data in the first data partition. Then we transfer the trained model to the second machine and it continues training using the second data partition. If each partition is large and is quite time-consuming, then we'll spend a significant amount of time waiting.

Figure 1.2 Example of training the model sequentially on each data partitions.



In this case, we can think about adding additional workers where each worker is responsible for consuming each of the data partitions and all workers train the same model in parallel without waiting for each other. This is definitely a good approach to speed up the model training process. However, what if some workers finish consuming the data partitions that they are responsible for and want to update the model at the same time? Which of the worker's results (e.g. gradients) should we use to update the model first? Then here come the conflicts and trade-offs between performance and model quality. For example, in Figure 1.2, if the data partition that the first worker uses might have better quality due to a more rigorous data collection process than the second worker, then using the first worker's results first would produce a more accurate model. On the other hand, if the second worker has a smaller partition, it could finish training faster so we can then start using this worker's computational resources to train a new data partition. When more and more workers are being added, e.g. three workers as shown Figure 1.2, the conflicts between completion time for data consumption among different workers become even more obvious.

Similarly, if the application that makes predictions using the trained machine learning model observes a much heavier traffic, can we simply add additional servers where each server handles a certain percentage of the traffic? Unfortunately the answer is not that quite simple either and this naive solution would need to take other things into consideration, such as deciding the best load balancer strategy and avoid processing duplicate requests in different servers.

We will learn more about handling these types of problems in the second part of the book. For now, the main takeaway is that we have established patterns and best practices to deal with situations like those and we will leverage those patterns to make the best out of the limited computational resources.

1.2 Distributed system

In order to train a large machine learning model with a larger amount of data, a single machine or personal laptop can no longer satisfy the requirements. We need to write programs that can run on multiple machines and can be accessed by people from all over the world. In this section, we'll talk about

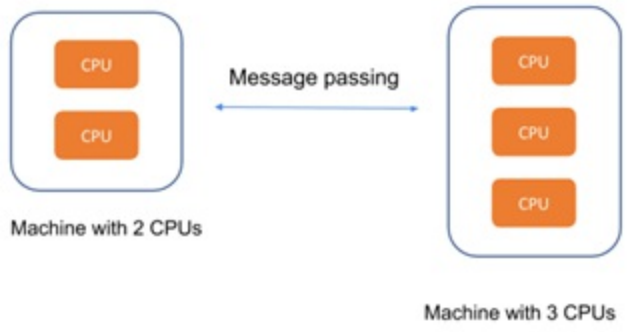
what a distributed system is and its complexity, and demonstrate with one concrete example pattern that's often seen in distributed systems.

1.2.1 What is it?

Computer programs have evolved from being able to run on only one machine to now working together with multiple machines. The increasing demand in computing and pursuit of higher efficiency, reliability, and scalability have boosted the advancement of large-scale datacenters that consist of hundreds or thousands of computers, communicating with each other via the shared network, which results in the development of *distributed systems*. A distributed system is a system in which components are located on different networked computers, which can communicate with one another to coordinate workloads and work together via message passing.

For example, Figure 1.3 illustrates a small distributed system consisting of two machines communicating with each other via message passing where one machine contains two CPUs and the other machine contains three CPUs. Note that obviously a machine contains other types of computational resources than just the CPUs but we only use CPUs here for illustration purposes. In real world distributed systems, the number of machines can be extremely large, e.g. tens of thousands depending on the use case. Machines with more computational resources can handle more workload and share the results with other machines.

Figure 1.3 Example of a small distributed system consisting of two machines with different amounts of computational resources communicating with each other via message passing.



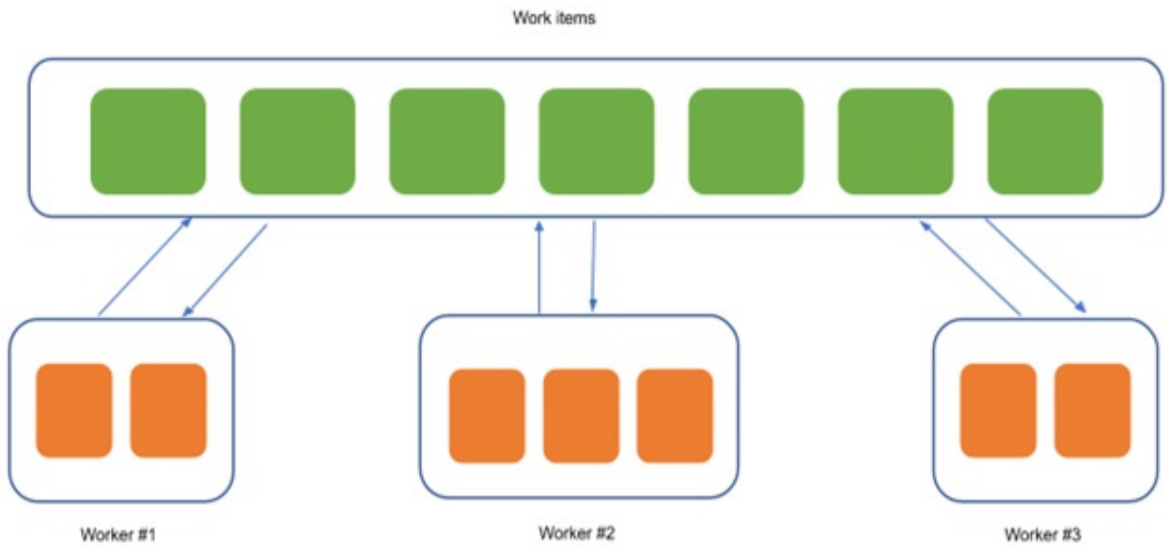
1.2.2 The complexity and patterns

These distributed systems can run on multiple machines and can be accessed by users from all over the world. They are often complex and need to be designed very carefully in order to be more reliable and scalable. Bad architectural considerations can lead to issues that are often at large scale and results in unnecessary cost.

There are a lot of good patterns and reusable components for distributed systems. For example, the *work-queue pattern* in a batch processing system that makes sure each piece of work is independent of the other and can be processed without any interventions within a certain amount of time. In addition, workers can be scaled up and down to ensure that the amount of workload can be handled properly.

For example, Figure 1.4 contains 7 work items where each work item might be an image that needs to be modified to grayscale by the system in the processing queue. Each of the 3 existing workers takes 2 to 3 work items from the processing queue. This makes sure that no worker is idled to avoid waste of computational resources and maximizes the performance by processing multiple images at the same time. This is possible since each of the work items is independent of each other.

Figure 1.4 Example of a batch processing system leveraging work-queue pattern to modify images to grayscale.



1.3 Distributed machine learning system

Distributed systems are not only useful in general computing tasks but also for machine learning applications. Imagine that we could utilize multiple machines with large amounts of computational resources in a distributed system to consume parts of the large dataset, store different partitions of a large machine learning model, etc., distributed systems can greatly speed up the machine learning application, with scalability and reliability in mind. In this section, we'll introduce what distributed machine learning systems are and similar patterns that are often seen in those systems, and talk about some real-life scenarios. At the end of this section, we'll also take a glance at what we'll be learning in this book.

1.3.1 What is it?

A *distributed machine learning system* is a distributed system that consists of a pipeline of steps and components responsible for different steps in machine learning applications, such as data ingestion, model training, model serving, etc. It leverages similar patterns and best practices in distributed systems but also patterns are designed specifically to benefit machine learning applications. Through careful design of a distributed machine learning system, the system is more scalable and reliable when handling large scale problems, e.g. large datasets, large models, heavy model-serving traffic, and

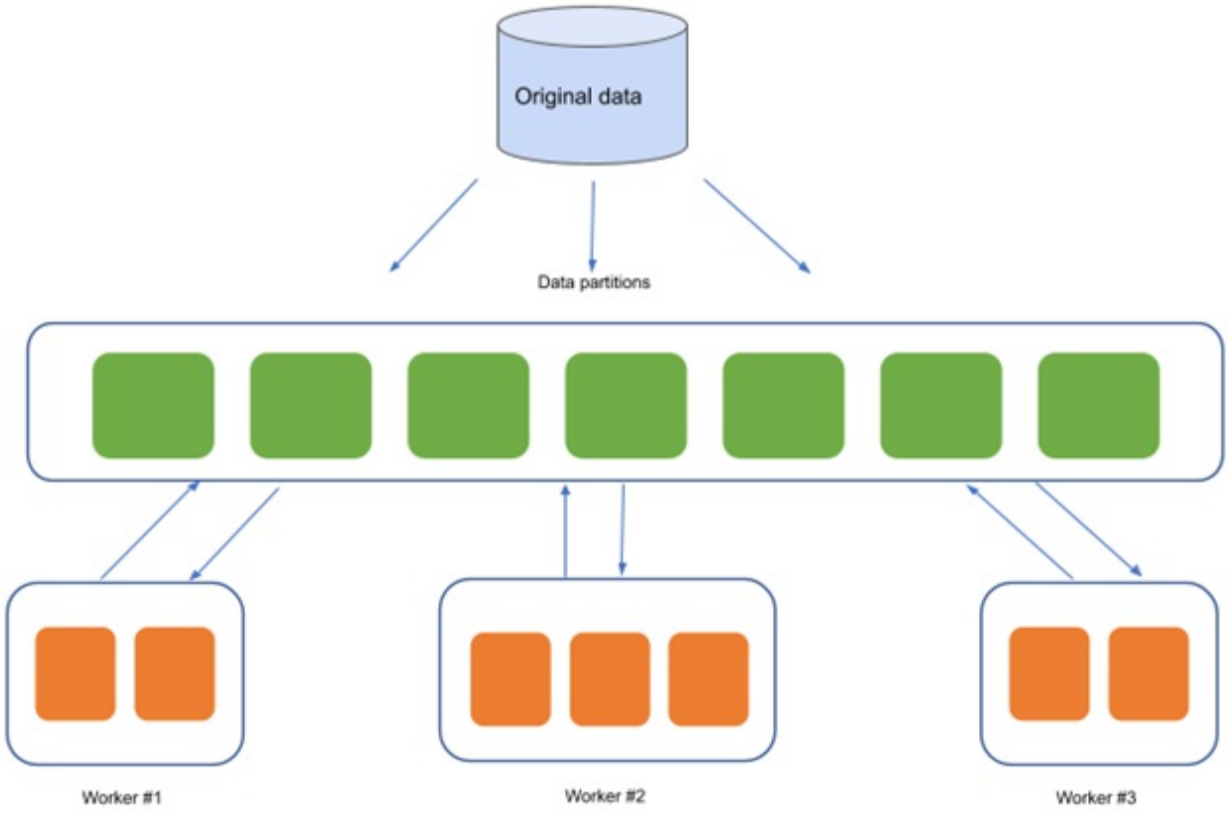
complicated model selection or architecture optimization.

1.3.2 Are there similar patterns?

In order to handle the increasing demand and scale of machine learning systems that will be deployed in real-life applications. We need to design the different components in a distributed machine learning pipeline very carefully. It’s often non-trivial but there are good patterns and best practices to follow that would speed up the development and deployment of machine learning models, leverage automations from different tools, and benefit from hardware accelerations.

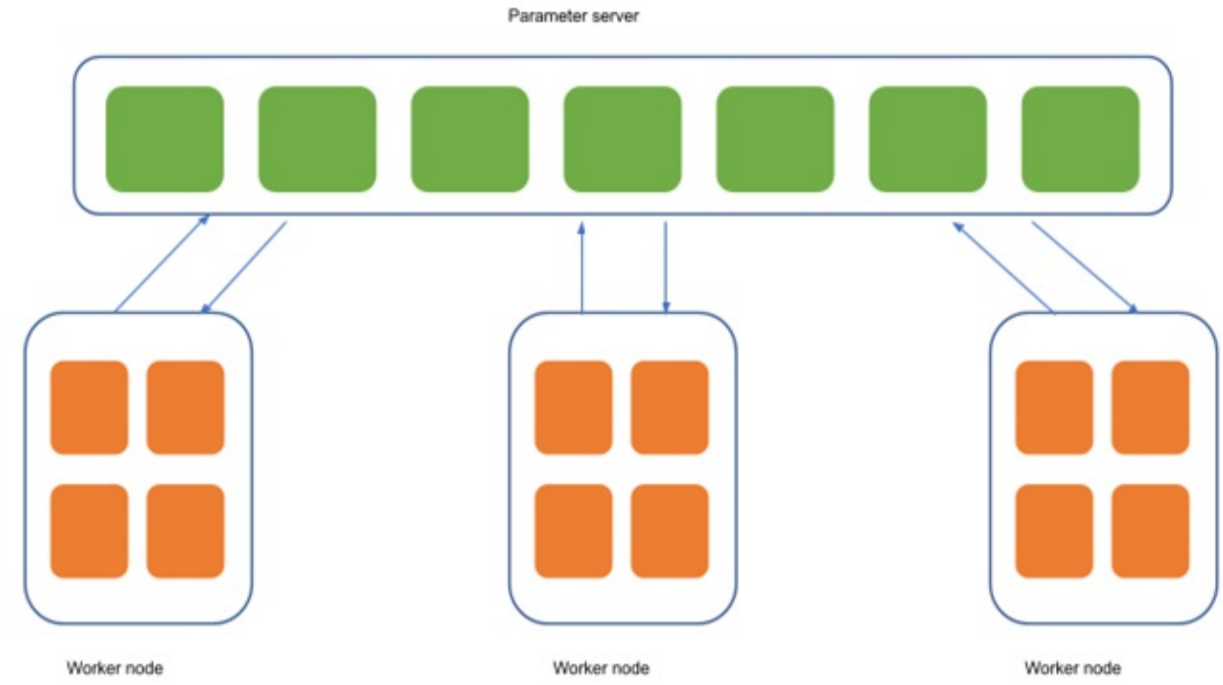
There are similar patterns in distributed machine learning systems. For example, multiple workers can be used to train the machine learning model asynchronously where each worker is responsible for consuming certain partitions of the dataset, which is similar to the work-queue pattern that we see in distributed systems and can speed up the model training process significantly. Figure 1.5 illustrates how we can apply this pattern to distributed machine learning systems by simply replacing the work items with data partitions. Each work takes some data partitions from the original data stored in a database and then to train a centralized machine learning model.

Figure 1.5 Example of applying the work-queue pattern in distributed machine learning systems.



Another example pattern usually seen in machine learning systems instead of general distributed systems is *parameter-server* pattern for distributed model training. As shown in Figure 1.6, the parameter servers are responsible for storing and updating a particular part of the trained model whereas each different worker node is responsible for taking a particular part of the dataset that will be used to update a certain part of the model parameters. This is useful in situations where the model is too large to fit in a single server and dedicated parameter servers for storing model partitions without allocating unnecessary computational resources.

Figure 1.6 Example of applying the parameter-server in a distributed machine learning system.



We will illustrate patterns like these in the second part of the book. For now, let's keep in mind that there are similar patterns in distributed machine learning systems that appear in general-purpose distributed systems as well as patterns specially designed for handling machine learning workloads at large scale.

1.3.3 When should we use it?

We should start thinking about designing a distributed machine learning system when any of the following scenarios occur. For example, if the dataset is large that cannot fit on our local laptops as illustrated previously in Figure 1.1 and 1.2, we can use patterns like data partitioning and or introduce additional workers to speed up model training.

There are also many other scenarios where we should apply distributed machine learning system patterns: model is large and consists of millions of parameters that a single machine cannot store and must be partitioned in different machines; the machine learning application needs to handle increasing amount of heavy traffic that a single server can no longer be capable of; the task at hand is not just building a simple model but consists many steps around the model lifecycle such as data ingestion, model serving,

data/model versioning, performance monitoring, etc.; there are a lot of computing resources at hand that we'd want to leverage for acceleration, e.g. tens of servers where each has many GPUs.

If any of these scenarios appears, it's usually a sign that a well-designed distributed machine learning system will be needed in the near future.

1.3.4 When should we not use it?

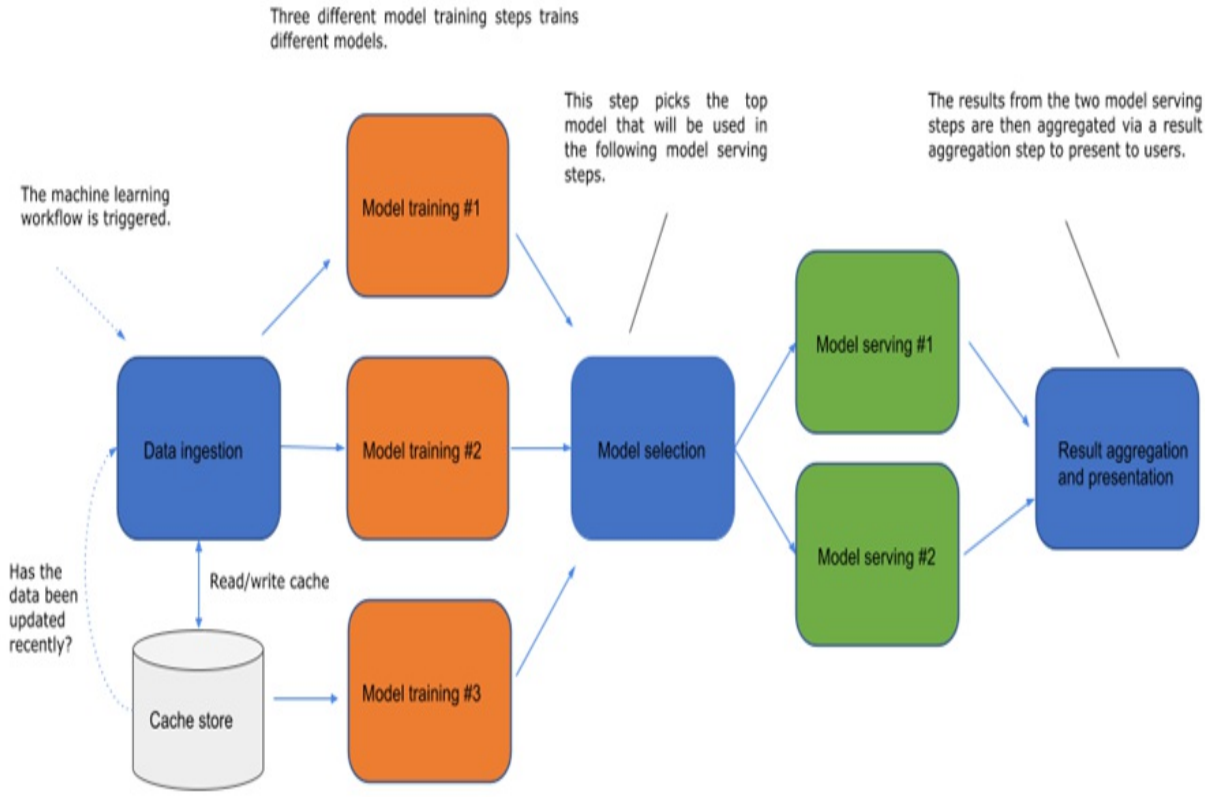
Though a distributed machine learning system is helpful in many situations, it is usually harder to design and requires experience to make the best out of it. Additional overhead and trade-off exist when developing and maintaining such complicated systems. If any of the following cases are encountered, we should not consider such systems and stay with the simple approach that already works well: the dataset is small, e.g. < 10GBs CSV file on your laptop; the model is simple and does not require heavy computations, e.g. linear regression; or there are only limited computing resources but sufficient for the tasks at hand.

1.3.5 What will we learn in this book?

In this book, we'll learn how to choose and apply the correct patterns for building and deploying distributed machine learning systems to handle large scale and gain practical experience for managing and automating machine learning tasks. Some popular frameworks and cutting-edge technologies, in particular TensorFlow, Kubernetes, Kubeflow, Dockerand Argo Workflows, will be used to build different components of a distributed machine learning workflow.

There will be a comprehensive hands-on project at the last part of the book that consists of an end-to-end distributed machine learning pipeline system. Figure 1.7 is the architecture diagram of the system that we will be building. We will gain hands-on experience implementing many of the patterns that we cover in the following chapters in this project. Being able to handle problems at larger scale and take what's developed on our personal laptop to large distributed clusters should be very exciting.

Figure 1.7 Architecture diagram of the end-to-end machine learning system that we will be building in the last part of the book.



We'll be using TensorFlow with Python to build machine learning and deep learning models for various tasks such as building useful features based on the real-life dataset, training predictive models, and making real-time predictions. We'll also Kubeflow to run distributed machine learning tasks on a Kubernetes cluster. Furthermore, Argo Workflows will be leveraged to build a machine learning pipeline that consists of many important components of a distributed machine learning system. The basics of these technologies will be introduced in the next chapter and we'll learn more in depth in the second part of the book to gain hands-on experience. Table 1.1 below shows the key technologies that will be used in this book and their example usages.

Table 1.1 The technologies that will be used in this book and their usages

Technology	Usage

TensorFlow	Build machine learning and deep learning models.
Kubernetes	Manage distributed environment and resources.
Kubeflow	Submit and manage distributed training jobs at ease on Kubernetes clusters.
Argo Workflows	Define, orchestrate, and manage workflows.
Docker	Build and manage images that will be used for starting up containerized environments.

Before we dive into more details in the next chapter, we recommend readers to have basic knowledge and hands-on experience in building machine learning models in Python. While experience in production environments or distributed systems is not a requirement, we expect readers in this position to have at least some exposure to machine learning applications running in production and should have written Python and Bash scripts for at least one year. In addition, understanding the basics of Docker and being able to manage images/containers using Docker CLI is required. Familiarity with basic YAML syntax is helpful but not required since it's pretty intuitive and should be pretty easy to pick up along the way. If most of these sound new to you, we suggest you pick them up from other resources before reading further.

1.4 References

1. TensorFlow: https://www.tensorflow.org/
2. Kubernetes: https://kubernetes.io/

3. Kubeflow: https://www.kubeflow.org/
4. Docker: https://www.docker.com/
5. Argo Workflows: https://argoproj.github.io/

1.5 Summary

- Machine learning systems deployed in real-life applications usually need to handle the growing scale of larger datasets and heavier model-serving traffic.
- It's non-trivial to design large scale distributed machine learning systems.
- A distributed machine learning system is usually a pipeline of many different components, such as data ingestion, model training, serving, monitoring, etc.
- There are good patterns for designing different components of a machine learning system to speed up the development and deployment of machine learning models, leverage automations from different tools, and benefit from hardware accelerations.

2 Data ingestion patterns

This chapter covers

- Understanding what's involved in data ingestion and what data ingestion is responsible for.
- Handling large datasets in memory by consuming datasets by small batches with the batching pattern.
- Preprocessing extremely large datasets as smaller chunks that are located in multiple machines with the sharding pattern.
- Fetching and re-accessing the same dataset more efficiently for multiple training rounds with the caching pattern.

In the previous chapter, we've discussed the growing scale of modern machine learning applications, e.g. larger datasets and heavier traffic for model serving. We've also talked about the complexity and challenges in building distributed systems and distributed systems for machine learning applications in particular. We've learned that a distributed machine learning system is usually a pipeline of many different components, such as data ingestion, model training, serving, monitoring, etc., where there are some established patterns for designing each individual component to handle the scale and complexity of real-world machine learning applications.

Data ingestion is the first step and an inevitable step in a machine learning pipeline. All data analysts and scientists should have some level of exposure to data ingestion. It could be either hands-on experience in building a data ingestion component or simply using a dataset from the engineering team or customer handed over to them.

Designing a good data ingestion component is non-trivial and requires understanding of the characteristics of the dataset we want to use for building a machine learning model. Fortunately there are established patterns that we can follow in order to build it on a reliable and efficient foundation.

In this chapter, we'll explore some of the challenges involved in the data

ingestion process and introduce a few established patterns adopted heavily in industries. For example, in Section 2.2, we will leverage the batching pattern in cases where we want to handle and prepare large datasets for model training, either when the machine learning framework we are using cannot handle large datasets or requires domain expertise in the underlying implementation of the framework. In Section 2.3, we will also learn how to apply the sharding pattern to split extremely large datasets into multiple data shards that spread among multiple worker machines and then speed up the training process as we add additional worker machines that are responsible for model training on each of the data shards independently. In the last section of this chapter, we will also introduce the caching pattern that could greatly speed up the data ingestion process where previously used dataset is re-accesses and processed for multi-epoch model training.

2.1 What is data ingestion?

Let's assume that we have a dataset at hand and we would like to build a machine learning system that would take the dataset and build a machine learning model from it. What is the first thing we should think about? The answer is actually quite intuitive: we should first get a better understanding of the dataset. For example, where did the dataset come from and how was it collected? Are the source and the size of the dataset changing over time? What are the infrastructure requirements for handling the dataset? These types of questions should be asked first and different perspectives that might affect the process of handling the dataset should also be considered before we can start building a distributed machine learning system. We will walk through these questions and considerations in the examples of the remaining sections of this chapter and learn how to address some of the potential issues we may encounter by leveraging different established patterns.

Data ingestion is the process that monitors the data source where the data comes from, consumes the data either all at once (non-streaming) or in a *streaming fashion* into memory, and performs preprocessing to prepare for the training process of machine learning models.

In short, streaming data ingestion often requires long-running processes to monitor the changes in data sources while non-streaming data ingestion

happens in the form of offline batch jobs that process datasets on demand. Additionally, the data is growing over time in streaming data ingestion while the size of the dataset is fixed in non-streaming data ingestion. A summary of the differences between them is shown in Table 2.1 below.

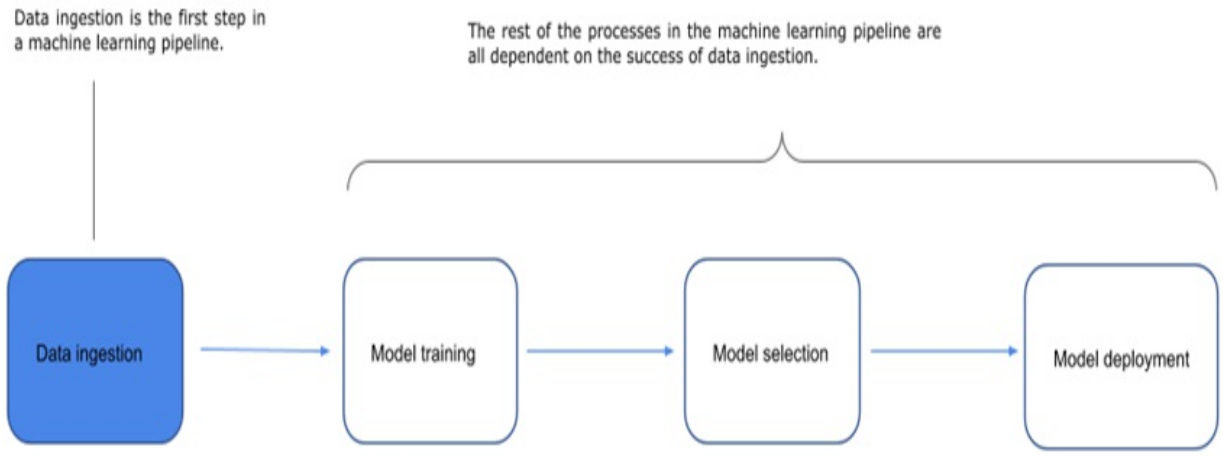
Table 2.1 Comparison between streaming and non-streaming data ingestion in machine learning applications.

	Streaming Data Ingestion	Non-streaming Data Ingestion
Dataset Size	Increasing over time	Fixed size
Infrastructure Requirements	Long-running processes to monitor the changes in data source	Offline batch jobs to process datasets on demand

In the remaining sections of this chapter, we will focus on data ingestion patterns from a non-streaming perspective but they can be applied to streaming data ingestion as well which we will also talk about in some of the discussion sections.

Data ingestion is the first step and an inevitable step in a machine learning pipeline, as shown in Figure 2.1 below. Without properly ingested dataset, the rest of the processes in a machine learning pipeline would not be able to proceed.

Figure 2.1 A flowchart that represents the machine learning pipeline. Note that data ingestion is the first step in the pipeline.



All data analysts and scientists should at least have some level of exposure to data ingestion. It could be either hands-on experience in building a data ingestion component on personal laptops or simply loading a dataset obtained from the engineering team or customer into memory to start training their machine learning models.

In the next section, we'll introduce the Fashion-MNIST dataset that will be used to illustrate the patterns in the remaining sections of this chapter. We'll focus on building patterns around data ingestion in distributed machine learning applications that are very distinct from data ingestion that happens on local machines or personal laptops. Data ingestion in distributed machine learning applications is often more complex and requires careful designs in order to handle datasets at very large scales or datasets that are growing at a rapid speed.

2.1.1 The Fashion-MNIST dataset

The MNIST dataset by Yann LeCun et al. is one of the widely used dataset for image classification. It contains 60,000 training images and 10,000 testing images extracted from images of handwritten digits and is used widely among the machine learning research community as a benchmark dataset to validate state-of-art algorithms or machine learning models.

Figure 2.2 below is a screenshot of some example images for handwritten digits from 0 to 9 where each row represents images for a particular

handwritten digit.

Figure 2.2 Screenshot of some example images for handwritten digits from 0 to 9 where each row represents images for a particular handwritten digit. Source: Josep Steffan, licensed under CC BY-SA 4.0.

Despite wide adoptions among the community, researchers found it unsuitable for distinguishing between stronger models and weaker ones since many simple models nowadays can achieve good classification accuracy over 95%. As a result, MNIST dataset now serves as more of a sanity check instead of a benchmark.

Note Brief history on Classification Accuracy for the MNIST dataset

The original creators of the MNIST dataset keep track of a list of some of the machine learning methods tested on the dataset. For example, in the original paper published on the MNIST dataset, they use a support-vector machine (SVM) model to get an error rate of 0.8%. Later, an extended dataset similar to MNIST called EMNIST was published in 2017, which contains 240,000 training images, and 40,000 testing images of handwritten digits and characters.

Instead of using MNIST, in several of our examples throughout this book, we will focus our discussions on a quantitatively similar but relatively more complex dataset, the Fashion-MNIST dataset that was released back in 2017.

Fashion-MNIST is a dataset of Zalando's article images — consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes. The Fashion-MNIST dataset is designed to serve as a direct drop-in replacement for the original MNIST dataset for benchmarking machine learning algorithms. It shares the same image size and structure of training and testing splits.

Figure 2.3 below is a screenshot of the collection of the images for all 10 classes (t-shirt/top, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag, and ankle boot) from Fashion-MNIST where each class takes three rows in the screenshot.

Figure 2.3 Screenshot of the collection of images from Fashion-MNIST dataset for all 10 classes (t-shirt/top, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag, and ankle boot). Source: Zalando SE, licensed under MIT License.

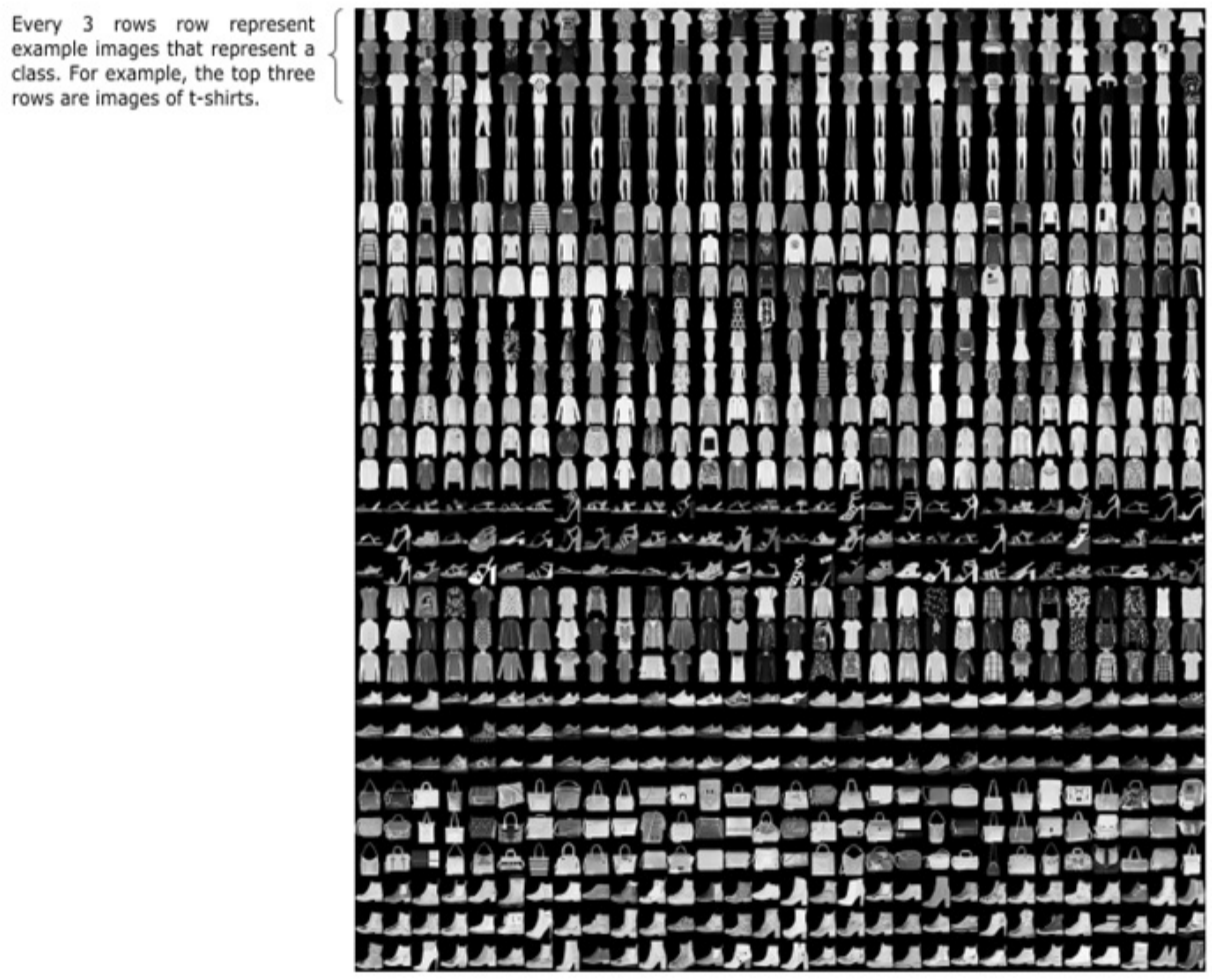


Figure 2.4 provides a closer look at the first few example images in the training set together with their corresponding labels in text. Next, let’s take a look at the scenario for the case study.

Figure 2.4 A closer look at the first few example images. In the training set with their corresponding labels in text.

Assume we have downloaded the Fashion-MNIST dataset and the compressed version of it should only take 30MBs on disk. Even how small the dataset is, it's very trivial to load all the downloaded dataset into memory at once using available implementations. For example, if we are using machine learning frameworks like TensorFlow, we can download and load the entire Fashion-MNIST dataset into memory with a couple of lines of Python code as shown below in Listing 2.1:

Listing 2.1 Loading the Fashion-MNIST dataset into memory with TensorFlow

```
> import tensorflow as tf  #A
>
> train, test = tf.keras.datasets.fashion_mnist.load_data()  #B

Downloading data from https://storage.googleapis.com/tensorflow/t
32768/29515 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/t
26427392/26421880 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/t
8192/5148 [==============================================] - 0s
Downloading data from https://storage.googleapis.com/tensorflow/t
4423680/4422102 [==============================] - 0s 0us/step
```

Alternatively, if the dataset is already in memory, for example, in the form of NumPy arrays, we can load the dataset from in-memory array representation into formats that are accepted by the machine learning framework, such as tf.Tensor objects that can be easily used for model training later. An example of this is shown in Listing 2.2.

Listing 2.2 Loading the Fashion-MNIST dataset from memory to TensorFlow

```
> import tensorflow as tf
>
> images, labels = train  #A
> images = images/255  #B
>
> dataset = tf.data.Dataset.from_tensor_slices((images, labels))
> dataset  #D
<TensorSliceDataset shapes: ((28, 28), ()), types: (tf.float64, t
```

2.2 Batching pattern: Performing expensive operations for Fashion-MNIST dataset with limited memory

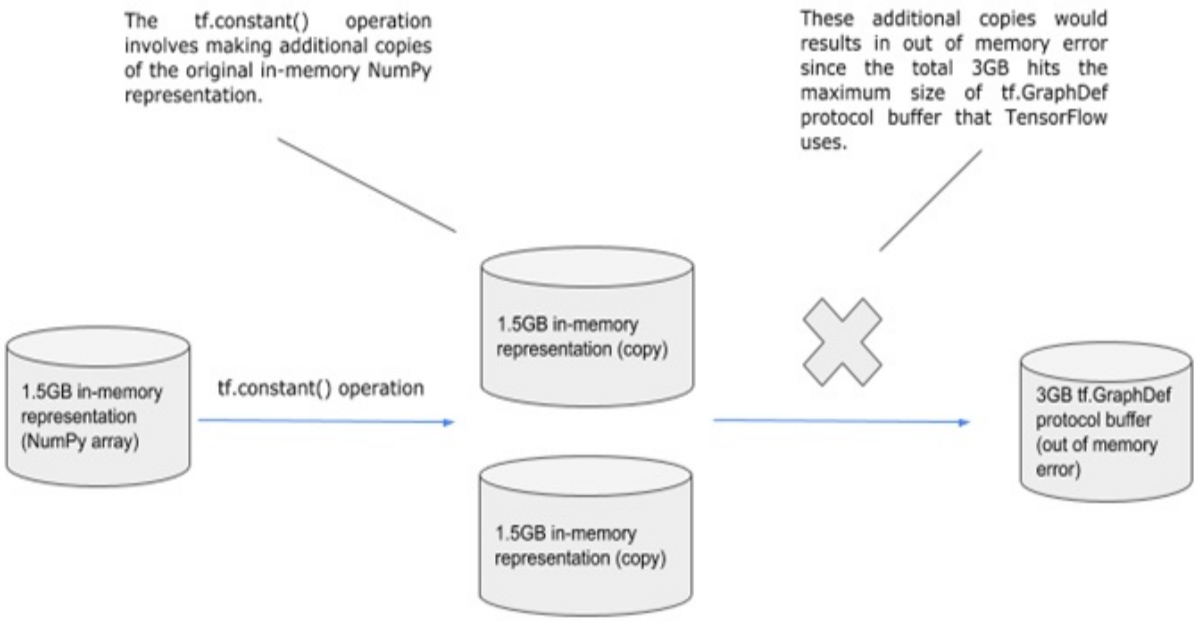
2.2.1 Problem

Even though it's very easy to load a small dataset like Fashion-MNIST into memory to prepare for model training, in real-world machine learning applications, however, this is often challenging.

For example, the code snippet in Listing 2.1 can be used to load the Fashion-MNIST into memory to prepare model training in TensorFlow, it will embed the features and labels arrays in your TensorFlow graph as tf.constant() operations. This works well for a small dataset, but wastes memory since the contents of the NumPy array will be copied multiple times and can run into the 2GB limit for the tf.GraphDef protocol buffer that TensorFlow uses underneath. In real-world applications, the datasets are much larger, especially in distributed machine learning systems where datasets accumulate and grow with time.

Figure 2.5 below is an example where there's a 1.5GB in-memory NumPy array representation that will be copied for two times with tf.constant() operation which would results in out of memory error since the total 3GB hits the maximum size of tf.GraphDef protocol buffer that TensorFlow uses.

Figure 2.5 An example where there's a 1.5GB in-memory NumPy array representation that hits out of memory error when being converted into tf.GraphDef protocol buffer.



Problems like the above happen often in different machine learning or data loading frameworks. Users may not be using the specific framework in an optimal way or the framework is simply not able to handle larger datasets.

In addition, even for small datasets like Fashion-MNIST, we may perform additional computations prior to feeding the dataset into the model, which is often common for tasks that often require additional transformations and cleaning. For example, for computer vision tasks, images often need to be resized, normalized, converted to gray-scale, or even more complex mathematical operations such as convolution operations. These operations may require a lot of additional memory space allocations while there aren't that much computational resources available after we loaded the entire dataset in memory.

2.2.2 Solution

Let's take a look at the first problem that we mentioned in Section 2.2.1. Recall that we'd like to use TensorFlow's from_tensor_slices() API to load the Fashion-MNIST dataset from in-memory NumPy array representation to a tf.Dataset object that can be used by TensorFlow's model training program. However, since the contents of the NumPy array will be copied multiple

times and we can run into the 2GB limit for the tf.GraphDef protocol buffer that TensorFlow uses underneath. As a result, we cannot load larger datasets that go beyond this limit.

It’s not uncommon to see issues like this for specific frameworks like TensorFlow. In this case the solution is actually very simple since we are not making the best out of TensorFlow. There are other APIs that allow us to load large datasets instead of loading the entire dataset into in-memory representation first.

For example, TensorFlow I/O library is a collection of file systems and file formats that are not available in TensorFlow's built-in support. We can load datasets like MNIST from a URL to access the dataset files that are passed directly to the tfio.IODataset.from_mnist() API call as shown in Listing 2.3. This is due to the inherent support that TensorFlow I/O library provides for the HTTP file system, and thus eliminating the need for downloading and saving datasets on a local directory.

Listing 2.3 Loading the MNIST dataset with TensorFlow I/O

```
> import tensorflow_io as tfio  #A
>
> d_train = tfio.IODataset.from_mnist(  #B
       'http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.
       'http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.
```

For larger datasets that might be stored in distributed file systems or databases, there are also APIs to load them without having to download everything at once that will likely hit memory or disk related issues. For demonstration purposes and without going into too much details here, below in Listing 2.4 is a glance at how we can load a dataset from PostgreSQL database (note that you’ll need to set up your own PostgreSQL database and provide the required environment variables to run this example):

Listing 2.4 Loading dataset from PostgreSQL database

```
> import os  #A
> import tensorflow_io as tfio  #B
>
> endpoint="postgresql://{}:{}@{}?port={}&dbname={}".format(  #C
```

```
        os.environ['TFIO_DEMO_DATABASE_USER'],
        os.environ['TFIO_DEMO_DATABASE_PASS'],
        os.environ['TFIO_DEMO_DATABASE_HOST'],
        os.environ['TFIO_DEMO_DATABASE_PORT'],
        os.environ['TFIO_DEMO_DATABASE_NAME'],
)
>
> dataset = tfio.experimental.IODataset.from_sql(  #D
        query="SELECT co, pt08s1 FROM AirQualityUCI;",
        endpoint=endpoint)
> print(dataset.element_spec)  #E
{'co': TensorSpec(shape=(), dtype=tf.float32, name=None), 'pt08s1
```

Now let’s go back to our scenario. In this case, assume that TensorFlow does not provide APIs like TensorFlow I/O that could deal with large datasets. Given that we don’t have too much free memory, we should not load the entire Fashion-MNIST dataset into memory directly. Let’s assume that the mathematical operations that we would like to perform on the dataset can be performed on subsets of the entire dataset. Then we can first divide the dataset into smaller subsets or *mini-batches*, load each individual mini-batch of example images, perform expensive mathematical operations on each batch, and then use only one mini-batch of images in each model training iteration.

For example, if the first mini-batch consists of the 19 example images in Figure 2.4, we can perform convolution or other heavy mathematical operations on those images first and then send the transformed images to the machine learning model for model training. We then repeat the same process for the remaining mini-batches while continuing performing model training in the meantime.

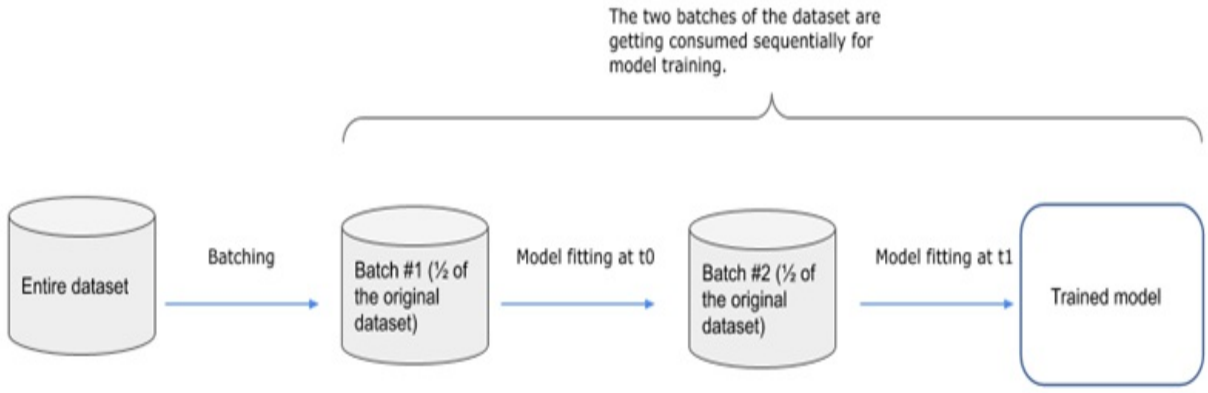
Since we’ve divided the dataset into many small subsets or mini-batches, we can avoid any potential issues with out-of-memory when performing various heavy mathematical operations on the entire dataset necessary for achieving an accurate classification model. We can then handle even larger datasets using this approach by reducing the size of the mini-batches.

This approach we just applied is called *batching*. Batching in data ingestion is the action to group a number of data records from the entire dataset into batches that will be used to train the machine learning model sequentially on

each batch.

For example, if we have a dataset with a total 100 records, we can first take 50 of the 100 records to form a batch and then train the model using this batch of records. Subsequently, we repeat this batching and model training process again for the remaining records. In other words, we make two batches in total where each batch consists of 50 records and the model we are training consumes the batches one by one. Figure 2.6 below is a diagram illustrating this process where the original dataset gets divided into two batches sequentially. The first batch gets consumed to train the model at time t0 and the second batch gets consumed at time t1. As a result, we don't have to load the entire dataset into memory at once and instead we are consuming the dataset batch by batch sequentially.

Figure 2.6 Dataset gets divided into two batches sequentially. The first batch gets consumed to train the model at time t0 and the second batch gets consumed at time t1.



This *batching pattern* can be summarized into the pseudo-code below in Listing 2.5 where we continuously try to read the next batch from the dataset and the train the model using the batch until there's no more batch left:

Listing 2.5 Pseudo-code for batching

```
batch = read_next_batch(dataset)  #A
while batch is not None:
       model.train(batch) #B
       batch = read_next_batch(dataset)  #C
```

We can apply the batching pattern in cases where we want to handle and prepare large datasets for model training. For example, when the framework we are using can only handle in-memory datasets, we can process small batches of the entire large datasets to ensure each batch can be handled within limited memory. In addition, if a dataset is divided into different batches, heavy computations can be done on each batch sequentially without requiring a huge amount of computational resources. We'll apply this pattern in Section 9.1.2.

2.2.3 Discussion

Other considerations need to be taken into account as well when performing batching. For example, this approach would only be feasible if the mathematical operations or algorithms we are performing can be done on subsets of the entire dataset in a streaming fashion.

For example, if an algorithm requires knowledge of the entire dataset, e.g. the sum of a particular feature over the entire dataset, then batching would no longer be a feasible approach as it's not possible to obtain this information over a subset of the entire dataset.

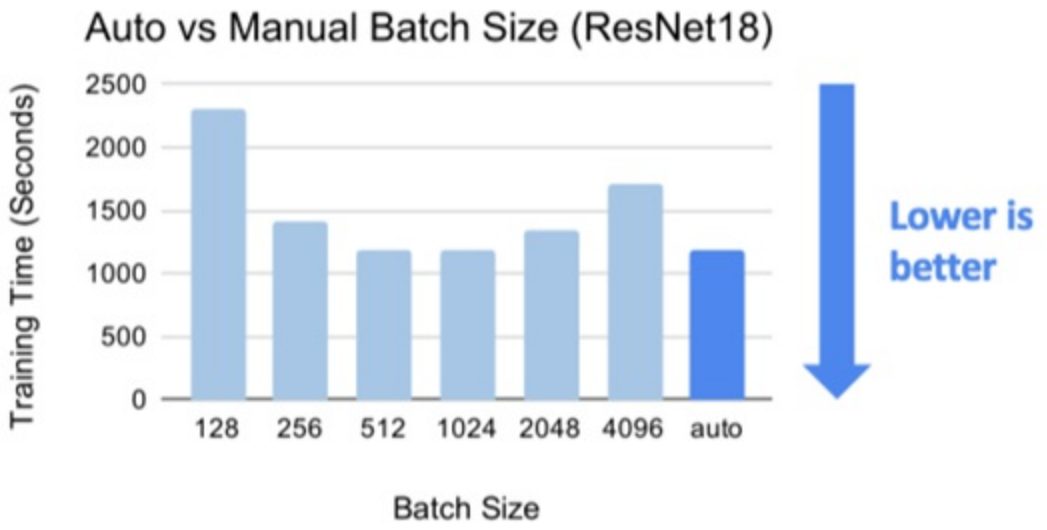
In addition, machine learning researchers and practitioners often would like to try out different machine learning models on the Fashion-MNIST dataset in order to get a more performing and accurate model. For example, if an algorithm would like to see at least 10 examples for each class in order to initialize some of its model parameters, then batching is not an appropriate approach here since there is no guarantee that every mini-batches contains at least 10 examples from each class, especially when batch size is small. An extreme case would be the batch size is 10 and it would be rare to see at least one image from each class in all batches.

Another thing to keep in mind is that the batch size of a machine learning model, especially for deep learning models, are strongly dependent on its allocation of resources, making them particularly difficult to decide in advance in shared-resource environments. Also, the allocation of resources that can be efficiently utilized by a machine learning job not only depends on the structure of the model being trained, but also on the batch size. This

codependency between the resources and the batch size creates a complex web of considerations that a machine learning practitioner must make in order to configure their job for efficient execution and resource utilization.

Fortunately there are algorithms and frameworks available that could avoid the manual tuning of the batch size. For example, AdaptDL offers automatic batch size scaling, which enables efficient distributed training without requiring any effort of tuning the batch size manually. It measures the system performance and gradient noise scale during training and adaptively selects the most efficient batch size. Figure 2.7 below shows a comparison between the effect of automatically vs. manually tuned batch size on the overall training time of ResNet18 model.

Figure 2.7 Comparison between the effect of automatically vs. manually tuned batch size on the overall training time of ResNet18 model. Source: Petuum, licensed under Apache License 2.0.



The batching pattern provides a great way to extract subsets of the entire dataset so that we can feed the batches sequentially for model training. However, for extremely large datasets that may not fit in a single machine, we'll need some other techniques. We will introduce a new pattern in the next section that addresses the challenges.

2.2.4 Exercises

1. Are we training the model using the batches in parallel or sequentially?
2. If the machine learning framework we are using does not handle large dataset, can we leverage the batching pattern?
3. If a machine learning model requires knowing the mean of a feature on the entire dataset, can we still use the batching pattern?

2.3 Sharding pattern: Splitting extremely large dataset among multiple machines

In Section 2.2, we introduced the Fashion-MNIST dataset and recall that the compressed version of it only takes 30MBs on disk. Even though it is very trivial to load all the dataset into memory at once, it's still challenging to load larger datasets for model training.

The batching pattern that we covered in Section 2.2 addresses the issue by grouping a number of data records from the entire dataset into batches that will be used to train the machine learning model sequentially on each batch. We can apply the batching pattern in cases where we want to handle and prepare large datasets for model training, either when the framework we are using cannot handle large datasets or requires domain expertise in the underlying implementation of the framework.

Let's now imagine that we have a much larger dataset at hand. This dataset is about 1000 times bigger than the Fashion-MNIST dataset. In other words, the compressed version of it takes 30MB x 1000 = 30GB on disk and it's about 50GB once it's decompressed. There are now 60,000 x 1000 = 60,000,000 training examples in this new dataset.

We now try to use this larger dataset with 60 millions training examples to train our machine learning model to classify images into different classes in the expanded Fashion-MNIST dataset, e.g. t-shirts, bags, etc. Let's neglect the detailed architecture of the machine learning model for now which will be discussed in depth in Chapter 3 and instead focus on its data ingestion component. In addition, here let's assume that we are allowed to use three machines for any potential speed-ups.

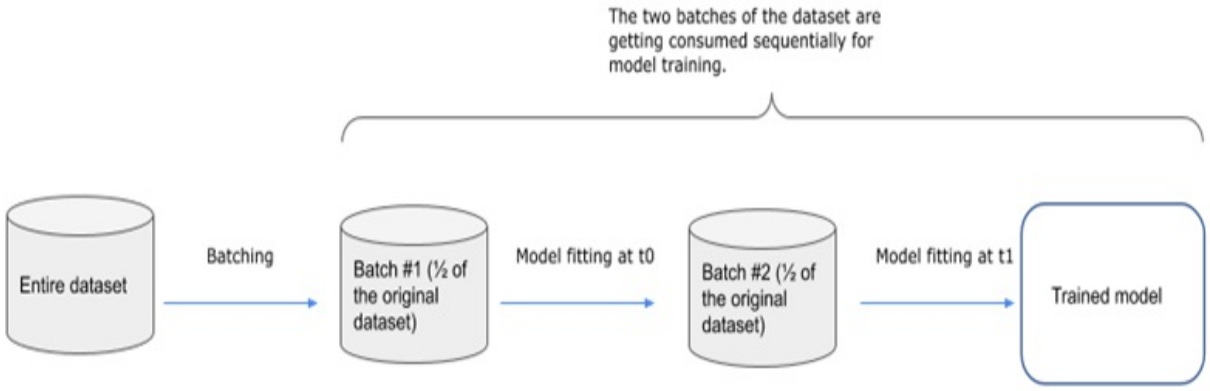
Given our previous experience, since the dataset is pretty large, we could first

try applying the batching pattern to divide the entire dataset into small batches so that they are small enough to load into memory for model training. Let's assume that our laptop has enough resources to store the entire 50GB decompressed dataset on disk and divide the dataset into 10 small batches where each batch is 5GB.

With this batching approach, we are able to handle large datasets like this as long as our laptop can store the large datasets and is able to divide them into batches. Next, we will start the model training process using the batches of data.

Recall that in section 2.2 we train the model batch by batch sequentially. In other words, one batch must be completely consumed by the machine learning model before the next batch gets consumed. In Figure 2.8 below, the second batch is consumed at time t1 by model fitting only after the first batch has been consumed by the model completely at time t0 where t0 and t1 represent two consecutive time points in this process.

Figure 2.8 Dataset gets divided into two batches sequentially. The first batch gets consumed to train the model at time t0 and the second batch gets consumed at time t1 where t0 and t1 represent two consecutive time points in this process.



2.3.1 Problem

Unfortunately, this sequential process of consuming data can be potentially slow. Assume that each batch of 5GB data takes about one hour to complete for this specific model we are training, it would take 10 hours to finish the model training process on the entire dataset.

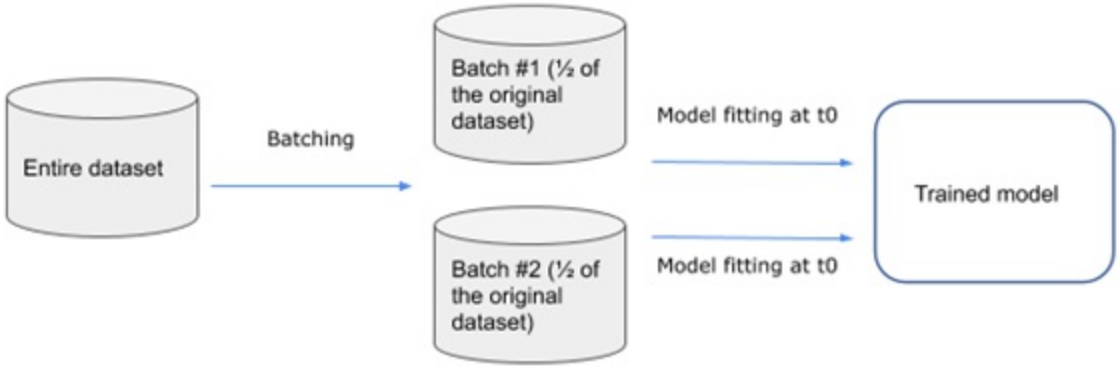
In other words, the batching approach may work well if we have enough time to train the model sequentially batch by batch. However, in real-world applications, there's always demand for more efficient model training, which will be affected by the time spent on ingesting the batches of data.

2.3.2 Solution

Now that we understand the slowness of training the model sequentially just by leveraging the batching pattern along. What can we do to speed up the data ingestion part that will greatly affect the model training process?

The major issue here is that we need to train the model sequentially batch by batch. Can we prepare multiple batches and then send them to the machine learning model for consumption at the same time? For example, Figure 2.9 below shows that the dataset gets divided into two batches where each batch is being consumed to train the model at the same time. This approach does not work yet as we recall that we cannot keep the entire dataset (two batches here) in memory at the same time but it is close to the solution that we will show soon here.

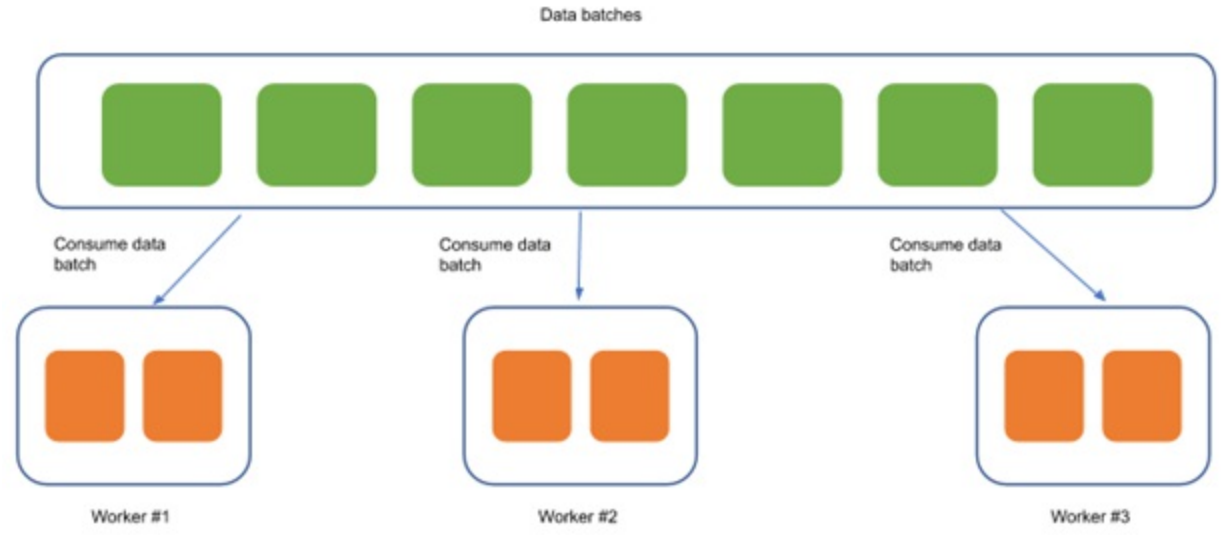
Figure 2.9 Dataset gets divided into two batches where each batch is being consumed to train the model at the same time.



Let's assume that we have multiple worker machines where each of the worker machines contains an exact copy of the machine learning model. Each of the model copies can then consume one batch of the original dataset and hence they can consume multiple batches independently.

Figure 2.10 below is the architecture diagram of multiple worker machines where each worker machine consumes batches independently to train the models copies located on them.

Figure 2.10 Architecture diagram of multiple worker machines where each worker machine consumes batches independently to train the models copies located on them.



You might be wondering -- how would multiple model copies work if they consume multiple different batches independently? Where would we obtain the final machine learning model from these model copies? These are great questions that we would like to hear and rest assured that we will go through the details of how the model training process works in this way in Chapter 3. For now, let's just assume that we have patterns available for multiple worker machines to consume multiple batches of dataset independently and this will greatly speed up the model training process that was originally slowed down due to the nature of sequential model training batch by batch.

Note Training models with multiple worker machines

We will be leveraging a pattern called collection communication pattern in Chapter 3 that would help us train models with multiple model copies located on multiple worker machines. For example, the collection communication pattern would be responsible for communicating updates of gradient calculations among worker machines and keeping each of the model copies in

sync.

Now we know that as long as we can produce batches to multiple worker machines, how would we produce those batches that will be used by those worker machines?

In our scenario, there are 60 millions training examples in the dataset and there are three worker machines available. Then it's simple to split the dataset into multiple non-overlapping subsets and then send each of them to the three worker machines as shown in the figure below.

The process of breaking up large datasets into smaller chunks that are spread across multiple machines is called *sharding* and those smaller data chunks are called *data shards*. Figure 2.11 below is the architecture diagram where the original dataset gets sharded into multiple non-overlapping data shards and then consumed by multiple worker machines.

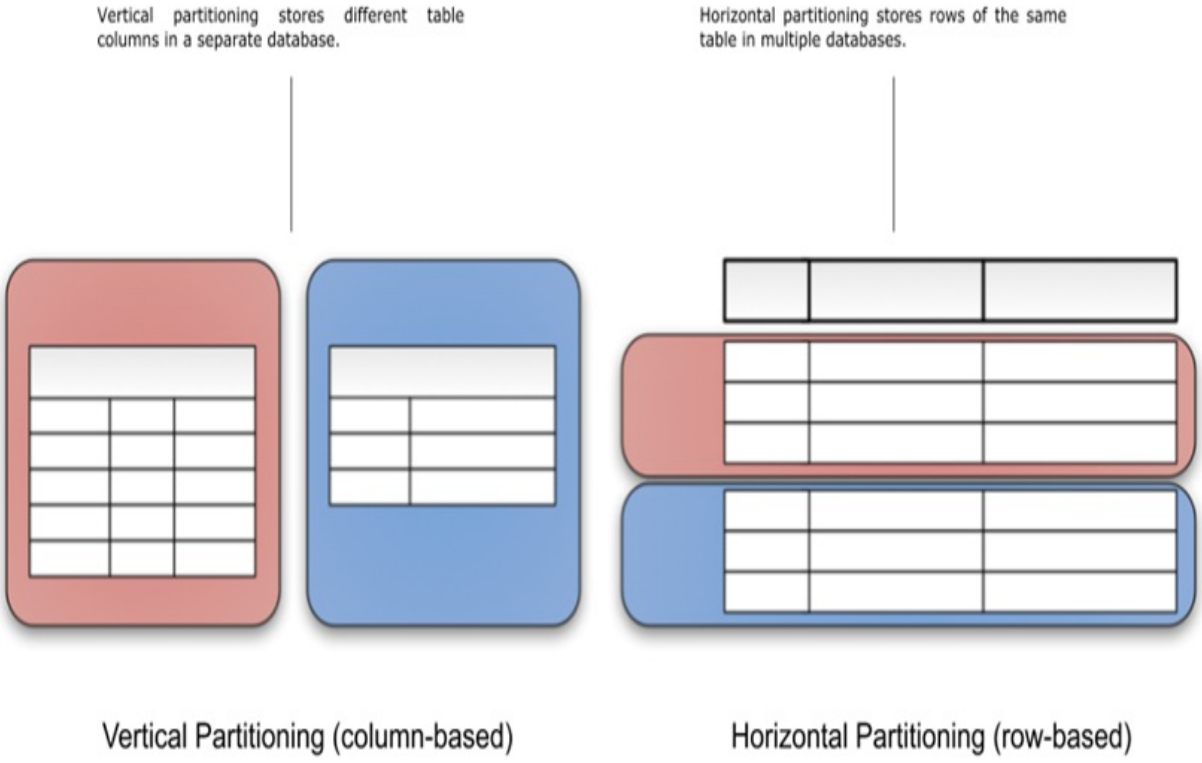
Note Sharding in distributed databases

Although we are introducing sharding here, it's not a new concept and is often seen in distributed databases. Sharding in distributed databases is extremely useful when solving scaling challenges such as providing high availability of the databases, increasing throughput, as well as reducing query response time.

Figure 2.11 Architecture diagram where the original dataset gets sharded into multiple non-overlapping data shards and then consumed by multiple worker machines.

A shard is essentially a *horizontal data partition* that contains a subset of the entire dataset and sharding is also referred to as *horizontal partitioning*. The distinction between horizontal and vertical comes from the traditional tabular view of a database. A database can be split vertically—storing different table columns in a separate database, or horizontally—storing rows of the same table in multiple databases. Figure 2.12 below is a diagram of Vertical partitioning vs. horizontal partitioning. Note that for vertical partitioning, we split the database into different columns where some of the columns may be empty, which is why we see only three out of the five rows appear in the partition on the right hand side in Figure 2.12.

Figure 2.12 Diagram of Vertical partitioning vs. horizontal partitioning. Source: YugabyteDB, licensed under Apache License 2.0.



This *sharding pattern* can be summarized into the pseudo-code below in Listing 2.6 where first create data shards from one of the worker machines (in this case worker machine with rank 0), send it to all other worker machines, and then on each worker machine we continuously try to read the next shard locally that will be used to train the model until there's no more shard left locally on this machine:

Listing 2.6 Pseudo-code for sharding

```
if get_worker_rank() == 0:  #A
create_and_send_shards(dataset)  #A
shard = read_next_shard_locally()  #B
while shard is not None:
       model.train(shard)  #C
shard = read_next_shard_locally()  #D
```

With the help of the sharding pattern, we can split extremely large datasets into multiple data shards that spread among multiple worker machines, and then each of the worker machines is responsible for consuming individual data shards independently. As a result, we have just avoided the slowness of

sequential model training due to the batching pattern. Sometimes it's also useful to shard large datasets into subsets of different sizes so that each shard can run different computational workloads depending on the amount of computational resource available in each worker machine. We'll apply this pattern in Section 9.1.2.

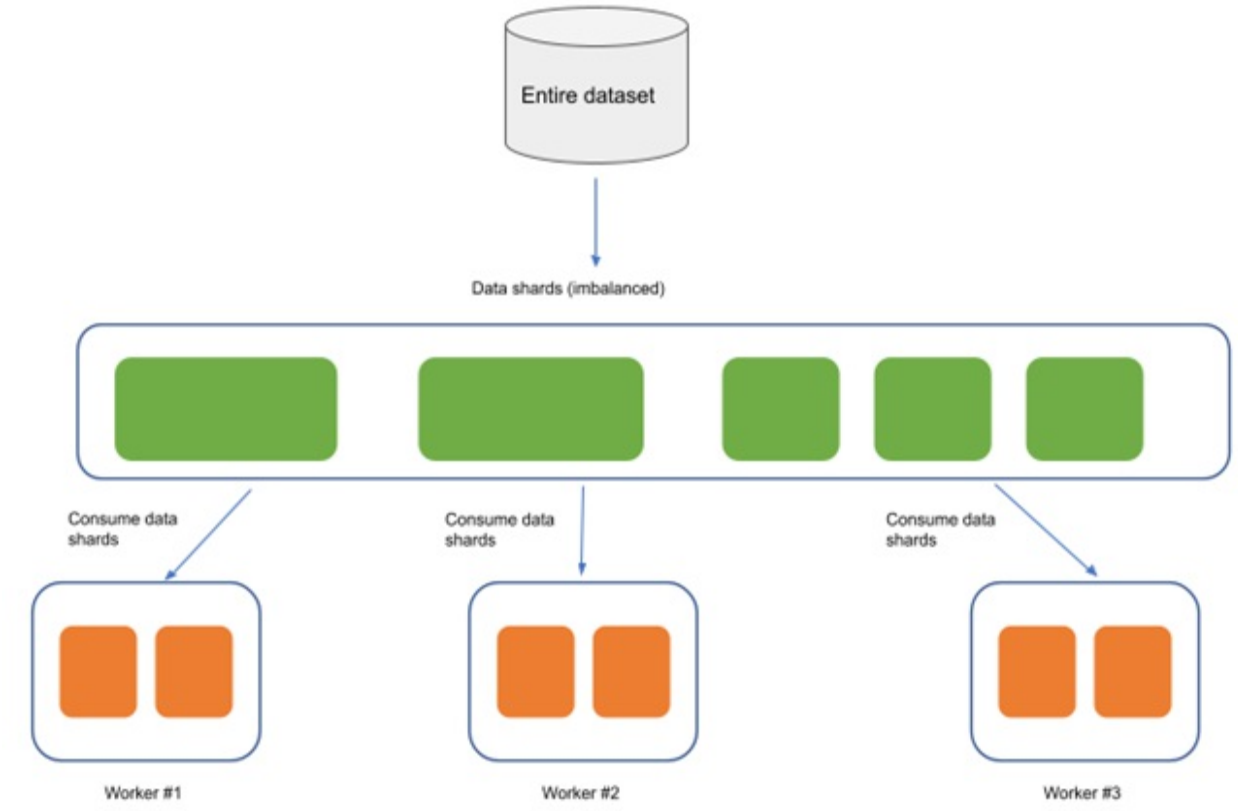
2.3.3 Discussion

We have just successfully leveraged the sharding pattern to split extremely large datasets into multiple data shards that spread among multiple worker machines and then speed up the training process as we add additional worker machines that are responsible for model training on each of the data shards independently. This is great and with this approach we can train machine learning models on extremely large datasets.

Now here comes the question: what if the dataset is growing continuously and we need to incorporate the new data that just arrived to the model training process? In this case, we'll have to re-shard every once a while if the dataset has been updated to re-balance each data shard to make sure they are splitted relatively evenly among different worker machines.

In Section 2.3.2, we simply just divided the dataset into two non-overlapping shards but unfortunately in real-world systems this manual approach is not ideal and may not work at all. One of the most significant challenges with manual sharding is uneven shard allocation. Disproportionate distribution of data could cause shards to become unbalanced, with some overloaded while others remain relatively empty. This imbalance could cause unexpected hanging of the model training process that involves multiple worker machines, which we'll talk about further in the next chapter. Figure 2.13 is an example where the original dataset gets sharded into multiple imbalanced data shards and then consumed by multiple worker machines.

Figure 2.13 Example where the original dataset gets sharded into multiple imbalanced data shards and then consumed by multiple worker machines.



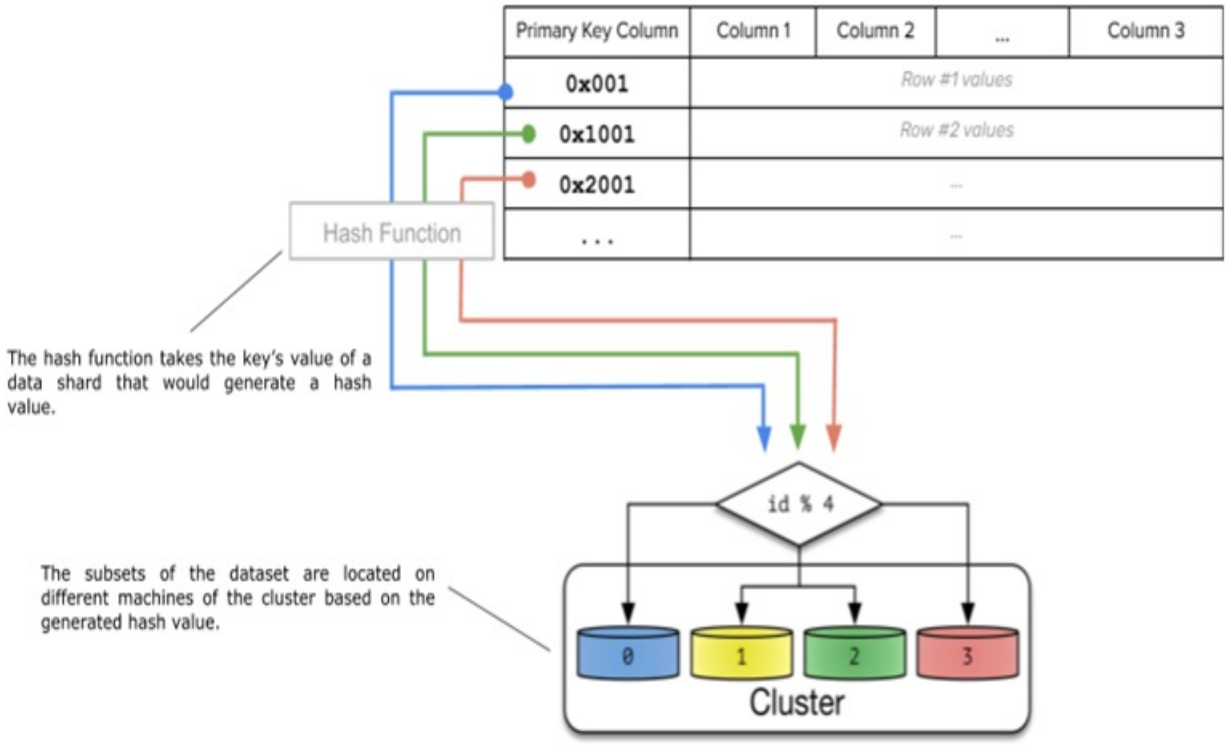
It’s best to avoid having too much data on one individual shard, which could lead to slowdowns and machine crashes. This problem could also happen when we force the dataset to be spread across too few shards. This is acceptable in development and testing environments, but definitely not ideal in production.

In addition, when manual sharding is used every time when we see an update in the growing dataset, the operational process is non-trivial. For example, backups will now have to be performed for multiple worker machines. Data migration and schema changes must be carefully coordinated to ensure all shards have the same schema copy.

To address that, we can actually apply auto sharding based on algorithms instead of sharding datasets manually. For example, hash sharding shown in Figure 2.14 below takes the key’s value of a data shard that would generate a hash value. The generated hash value is then used to determine where a subset of the dataset should be located. With a uniform hashing algorithm, the hash function can then evenly distribute data across different machines

and thus reducing the issues we mentioned above. In addition, data with shard keys that are closer to each other are unlikely to be placed on the same shard.

Figure 2.14 Diagram for hash harding where a hash value is generated to determine where a subset of the dataset should be located. Source: YugabyteDB, licensed under Apache License 2.0.



The sharding pattern works great by splitting extremely large datasets into multiple data shards that spread among multiple worker machines and then each of the worker machines is responsible for consuming individual data shards independently. With this approach, we can avoid the slowness of sequential model training due to the batching pattern. Both the batching and sharding patterns work great with the model training process and eventually, the dataset will be iterated thoroughly. However, there are machine learning algorithms that require multiple scans of the dataset which means that we might perform batching and sharding twice. We will introduce a pattern in the next section to speed up this process.

2.3.4 Exercises

1. Is the sharding pattern we introduced horizontal partitioning or vertical partitioning?
2. Where is the model reading each shard from?
3. Is there any alternative to manual sharding?

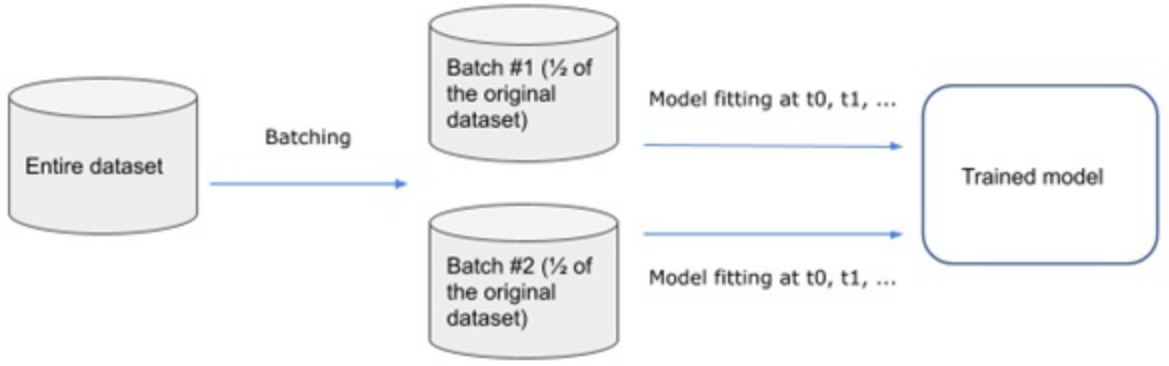
2.4 Caching pattern: Re-accessing previously used data for efficient multi-epoch model training

Let's recap what we patterns we have learned so far. In Section 2.2, We have successfully leveraged the batching pattern in cases where we want to handle and prepare large datasets for model training, either when the machine learning framework we are using cannot handle large datasets or requires domain expertise in the underlying implementation of the framework. With the help of batching, we can process large datasets and perform expensive operations under limited memory. We have also learned in Section 2.3 on how to apply the sharding pattern to split extremely large datasets into multiple data shards that spread among multiple worker machines and then speed up the training process as we add additional worker machines that are responsible for model training on each of the data shards independently.

Both of these patterns are great approaches that allow us to train machine learning models on extremely large datasets that otherwise either couldn't fit in a single machine or slows down the model training process.

One fact that we've been neglecting is that modern machine learning algorithms, such as tree-based algorithms and deep learning algorithms, often require training for multiple *epochs* where each epoch is a full pass through of all the data we are training on, where every sample has been seen once. For example, a single epoch means the single time the model sees all examples in the dataset. For example, a single epoch in the Fashion-MNIST dataset means that the model we are training has processed and consumed all the 60,000 examples once. Figure 2.15 below is a diagram of model training for multiple epochs at time t0, t1, etc.

Figure 2.15 Diagram of model training for multiple epochs at time t0, t1, etc.



Training these types of machine learning algorithms usually involves optimizing a large set of parameters that are heavily interdependent. As a matter of fact, it can take a lot of labelled training examples before the model even settles into an area of the solution space which is close to the optimal solution. This is exacerbated by the stochastic nature of batch gradient descent, in case of deep learning algorithms, where the underlying optimization algorithm is very data-hungry.

Unfortunately, these types of multidimensional data like the Fashion-MNIST dataset that those algorithms require is often expensive to label and takes up large amounts of storage space. As a result, even though we need to feed the model lots of data, the amount of samples available is generally much smaller than the number of samples that would allow the optimization algorithm to reach a good enough optimum. There may be enough information in these training samples, but the gradient descent algorithm takes time to extract it.

Fortunately, we can compensate for the limited number of samples by making multiple passes over the data. This will give the algorithm time to converge, without requiring an impractical amount of data. In other words, we can train a good enough model that consumes the training dataset for multiple epochs.

2.4.1 Problem

Now that we know that we would like to train a machine learning model for multiple epochs on the training dataset. Let’s assume that we want to do this on the Fashion-MNIST dataset.

If training one epoch on the entire training dataset takes 3 hours, then we would need to double the amount of time spent on model training if we would like to train two epochs, as shown in Figure 2.16. In real-world machine learning systems, an even larger number of epochs is often needed and this is not very efficient.

Figure 2.16 Diagram of model training for multiple epochs at time t0, t1, etc. where we spent 3 hours for each one of the epochs.

2.4.2 Solution

Given the unreasonable amount of time needed to train a machine learning model for multiple epochs, is there anything we can do to speed this up?

For the first epoch, there really isn't anything we can do to improve at this point since that is the first time the machine learning model has seen the entire set of training dataset. What about the second epoch then? Can we make use of the fact that the model has already seen the entire training dataset once?

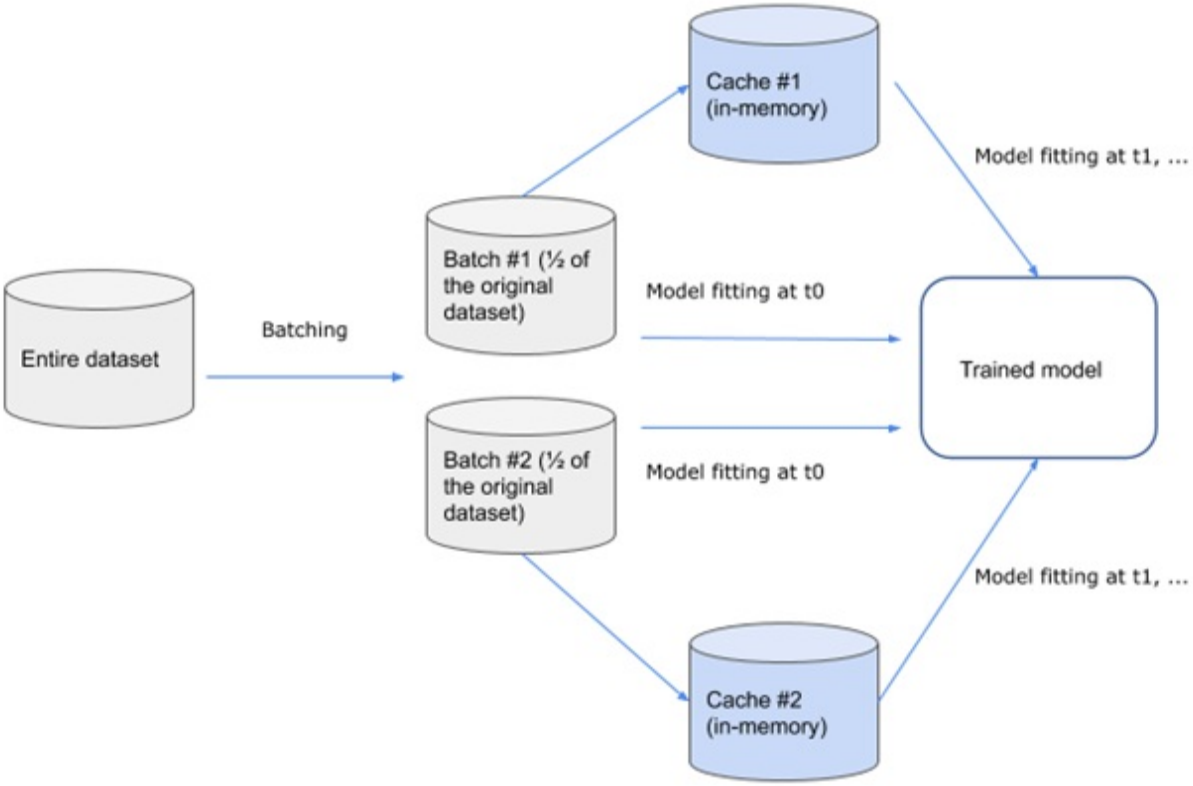
Let's assume that the single laptop we are using to train the model has sufficient computational resources such as memory and disk spaces. As soon as the machine learning model consumes each training example from the entire dataset, we can actually hold off recycling and instead we keep the consumed training examples in memory.

In other words, we are storing a *cache* of the training examples, in the form

of in-memory representation that could provide speed-ups when we re-access it again in the following training epochs.

For example, in Figure 2.17, after we have finished fitting the model for the first epoch, we store a cache for both of the two batches that we used for the first epoch of model training. Then we can start training the model for the second epoch by feeding the stored in-memory cache to the model directly without having to read from the data source over and over again for future epoches.

Figure 2.17 Diagram of model training for multiple epochs at time t0, t1, etc. with cache without having to read from the data source over and over again.



This *caching pattern* can be summarized into the pseudo-code below in Listing 2.7 where we first read the next batch to train the model and then append this batch to the initialized cache during the first epoch. For the remaining epochs, we read batches from the cache and then use those batches for model training:

Listing 2.7 Pseudo-code for caching

```
batch = read_next_batch(dataset)  #A
cache = initialize_cache(batch)  #B
while batch is not None:  #C
       model.train(batch)  #C
       cache.append(batch)  #C
       batch = read_next_batch(dataset)
while current_epoch() <= total_epochs:  #D        batch = cache.r
       model.train(batch)  #D
```

If we have performed expensive preprocessing steps on the original dataset, we could cache the processed dataset instead of the original dataset so that we can avoid wasting time on processing the dataset again. The pseudo-code for this is shown below in Listing 2.8. :

Listing 2.8 Pseudo-code for caching with preprocessing

```
batch = read_next_batch(dataset)
cache = initialize_cache(preprocess(batch))  #A
while batch is not None:
       batch = preprocess(batch)
       model.train(batch)
       cache.append(batch)
       batch = read_next_batch(dataset)
while current_epoch() <= total_epochs:
       processed_batch = cache.read_next_batch()  #B
       model.train(processed_batch)  #B
```

Note that Listing 2.8 is similar to what we have previously in Listing 2.7. One slight difference is that in #A we initialize the cache with the preprocessed batch instead of the raw batch as in Listing 2.7 and then in #B we read the processed batch from the batch directly without having to preprocess the batch again prior to model training.

With the help of the caching pattern, we can greatly speed up the re-access to the dataset for model training process that involves training on the same dataset for multiple epochs. Caching can also be useful to quickly recover from any failures so that a machine learning system can re-access the cached dataset easily to continue the rest of the processes in the pipeline. We’ll apply this pattern in Section 9.1.1.

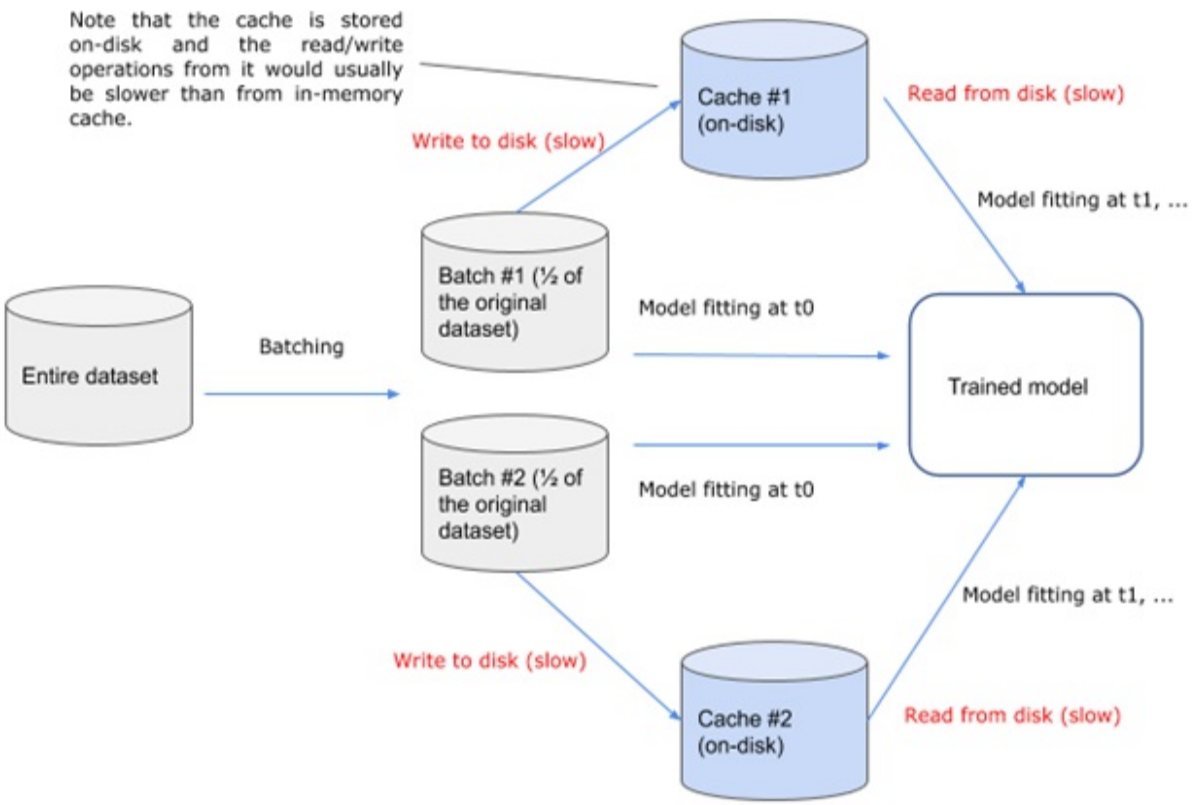
2.4.3 Discussion

Now that we have successfully used the caching pattern to store the cache in-memory on each worker machine in order to speed up the process to re-access previously used data for multiple epochs of model training. What if any failure happens in the worker machine? For example, if the training process gets killed due to out of memory, we would lose all the previously stored cache in memory.

To avoid losing the previously stored cache, we can write them to disk instead of storing them in memory and persist them as long as the model training process still needs it. This way we can easily recover the training process using previously stored cache of training data on disk. Details on how to recover the training process or make the training process more tolerant to failures will be discussed in more depth in Chapter 3.

Storing cache on disk is a good solution that could help us pick up from what's left. However, one thing to note is that reading from or writing to memory is about 6 times faster when we are doing sequential access but it's about 100,000 times faster when we are doing random access than accessing from disk. Random Access Memory (RAM) takes nanoseconds to read from or write to, while hard drive access speed is measured in milliseconds. In other words, there's a trade-off between storing cache in memory or on disk due to the difference in access speed as shown in Figure 2.18.

Figure 2.18 Diagram of model training for multiple epochs at time t0, t1, etc. with on-disk cache.



Generally speaking, storing cache on disk is preferred if we want to build a more reliable and fault-tolerant system while storing cache in memory is preferred when we want to have more efficient model training and data ingestion processes. On-disk cache can be extremely useful when the machine learning system requires reading from remote databases, where reading from on-disk cache is much faster than reading from remote databases, especially when network connection isn't fast and stable enough.

What if the dataset gets updated and accumulated over time like what we previously talked about in Section 2.3.3 where the data shard on each worker machine needs to be re-distributed and get balanced? In this case, we should take the freshness of the cache into account and update the cache on a schedule based on the specific application.

2.4.4 Exercises

1. Is caching useful for model training that requires training on the same or different dataset for multiple epochs?

2. What should we store in the cache if the dataset needs to be preprocessed?
3. Is on-disk cache faster to access than in-memory cache?

2.5 References

1. MNIST dataset: https://en.wikipedia.org/wiki/MNIST_database
2. Fashion-MNIST dataset: https://github.com/zalandoresearch/fashion-mnist
3. NumPy: https://numpy.org/
4. TensorFlow I/O: https://github.com/tensorflow/io
5. PostgreSQL: https://www.postgresql.org/
6. AdaptDL: https://github.com/petuum/adaptdl
7. ResNet18: https://arxiv.org/abs/1512.03385

2.6 Summary

- Data ingestion is usually the beginning process of a machine learning system and is responsible to monitor any incoming data and perform necessary processing steps to prepare for model training.
- The batching pattern helps handle large datasets in memory by consuming datasets by small batches.
- The sharding pattern prepares extremely large datasets as smaller chunks that are located in different machines.
- The caching pattern makes data fetching for multiple training rounds more efficient by caching previously accessed data that can be reused when accessed again for the additional rounds of model training on the same dataset.

Answers to exercises:

Section 2.2.4

1. Sequentially.
2. Yes, that’s actually one of the main use cases of batching.
3. No.

Section 2.3.4

1. Horizontal partitioning.
2. Locally on each worker machine.
3. Automatic sharding, e.g. hash sharding.

Section 2.4.4

1. Same dataset.
2. We should store the already preprocessed batches in the cache to avoid wasting time on preprocessing again in the following epochs.
3. No, generally in-memory cache is faster to access.

3 Distributed training patterns

This chapter covers

- Distinguishing the traditional model training process from the distributed training process that leverages multiple machines in a distributed cluster.
- Using parameter servers for building large and complex models that cannot fit in a single machine.
- Improving distributed model training performance for small or medium-sized models using the collective communication pattern and overcoming the potential communication overhead involved among parameter servers and workers.
- Handling unexpected failures due to corrupted datasets, unstable networks, and preempted machines during the distributed model training process.

In the previous chapter, we've introduced a couple of practical patterns that can be incorporated into the data ingestion process, which is usually the beginning process of a distributed machine learning system that's responsible for monitoring any incoming data and performing necessary preprocessing steps to prepare model training.

Distributed training is the next step after we've completed the data ingestion process and is what distinguishes distributed ML systems from other distributed systems. It's the most critical part in a distributed machine learning system.

The system design needs to be scalable and reliable to handle datasets and models of different sizes and various levels of complexity. There are large and complex models that cannot fit in a single machine as well as medium-sized models that are small enough to fit in single machines while struggling at improving the computational performance of distributed training.

It's also essential to know what to do when we see performance bottlenecks

and unexpected failures. For example, parts of the dataset is corrupted or cannot be used for training the model successfully or the distributed cluster that the distributed training is depending on may experience unstable or even disconnected network due to weather conditions or human errors.

In this chapter, we'll explore some of the challenges involved in the distributed training process and introduce a few established patterns adopted heavily in industries. For example, in Section 3.2, we'll see challenges when training very large machine learning models that tag main themes in new YouTube videos but cannot fit in a single machine and how we overcome the difficulty using the parameter server pattern. We'll also learn how to leverage the collective communication pattern to speed up distributed training for smaller models and avoid the unnecessary communication overhead among parameter servers and workers. In the last section, we will talk about some of the vulnerabilities that are often seen in distributed machine learning systems due to corrupted datasets, unstable networks, and preemptive worker machines and how we can address those issues.

3.1 What is distributed training?

Distributed Training is the process of taking the data that has already been processed by data ingestion that we discussed in the previous chapter, initialize the machine learning model, and then train the model with the processed data in a distributed environment such as multiple nodes.

Table 3.1 Comparison between traditional (non-distributed) training and distributed training for machine learning models

	Traditional Model Training	Distributed Model Training
Computational Resources	Personal laptop or single remote server	Cluster of machines

Dataset Location	Local disk on a single laptop or machine	Remote distributed database or partitioned on disks of multiple machines
Network infrastructure	Local hosts	InfiniBand or RDMA
Model size	Small enough fit on a single machine	Medium to large

It’s easy to get this confused with the traditional training process of machine learning models which only takes place in a single-node environment where the datasets and the machine learning model objects are in the same machine, such as a laptop. On the other hand, distributed model training usually happens in a cluster of machines that could work concurrently to greatly speed up the training process.

In addition, the dataset is often located in the local disk on a single laptop or machine in traditional model training while a remote distributed database is used to store the dataset or the dataset has to be partitioned on disks of multiple machines in distributed model training. If the model is not small enough to fit on a single machine, then it’s not possible to train the model in a traditional way with a single machine. From a network infrastructure perspective, an InfiniBand or RDMA network is often preferred for distributed training instead of relying on a single local host. A summary of the comparison can be found in Table 3.1.

Note InfiniBand and RDMA

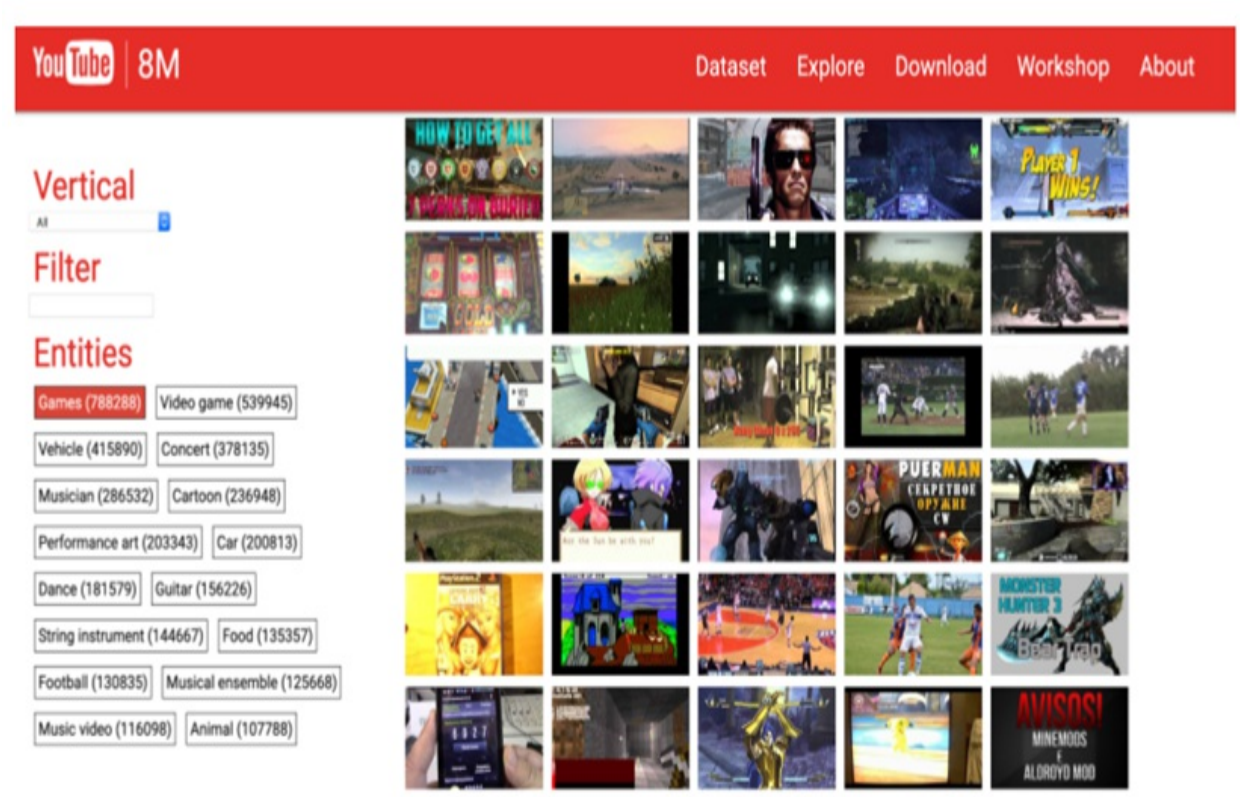
InfiniBand is a computer networking communications standard used in high-performance computing that features very high throughput and very low latency for data interconnect both among and within computers or storage systems, which is often required for distributed training.

Remote direct memory access (RDMA) is a direct memory access from the memory of multiple machines without involving either machine's operating system. This permits high-throughput, low-latency networking, which is especially useful in the distributed training process where the communications among different machines are very frequent.

3.2 Parameter server pattern: Tagging entities in 8 millions of YouTube videos

Assume that we have a dataset YouTube-8M at hand that consists of millions of YouTube video IDs, with high-quality machine-generated annotations from a diverse vocabulary of 3,800+ visual entities (such as food, car, music, etc.) as shown in Figure 3.1, and we'd like to train a machine learning model to tag the main themes of new YouTube videos that the model hasn't seen before.

Figure 3.1 The website that hosts the YouTube-8M dataset with millions of YouTube videos from a diverse vocabulary of 3,800 visual entities. Source: Sudheendra Vijayanarasimhan, et al., licensed under Non-exclusive License 1.0.

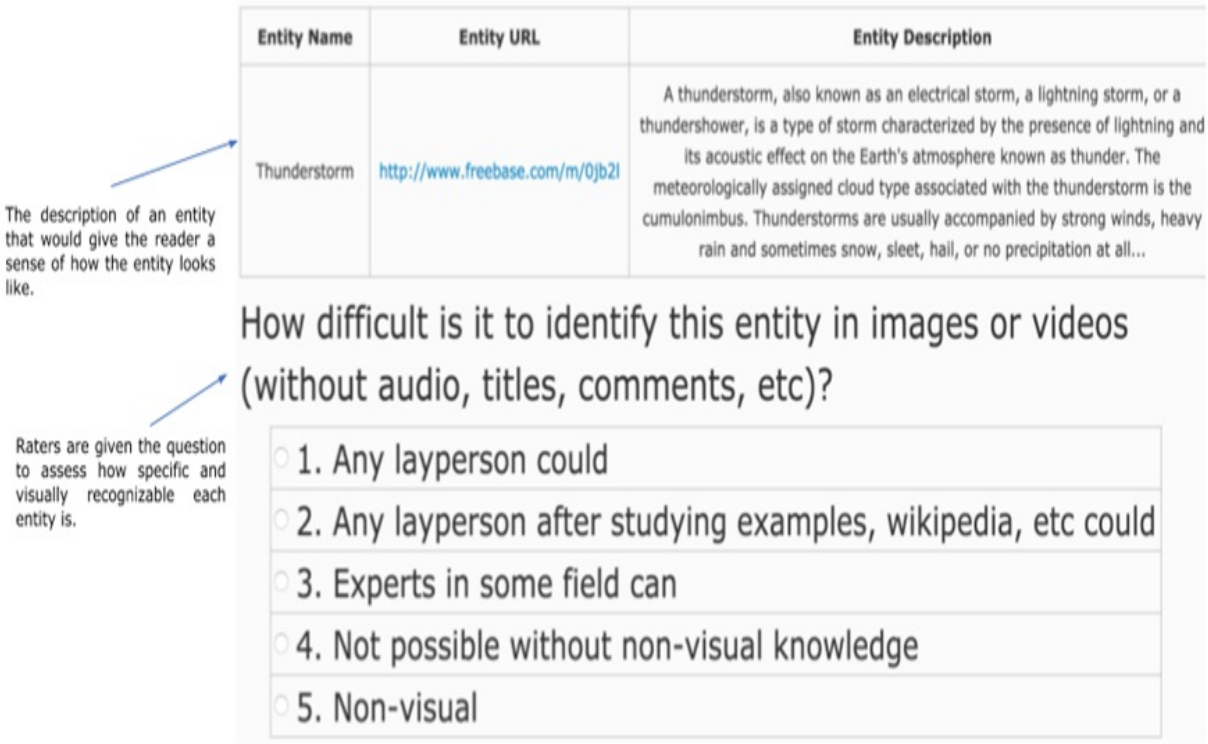


In this dataset, there are more than three thousands entities, including both *coarse* and *fine-grained* entities where coarse entities are the ones that can be recognized by non-domain experts after studying some existing examples and fine-grained entities are identified by domain experts who know how to differentiate between even extremely similar entities.

These entities have been semi-automatically curated and manually verified by 3 raters to be visually recognizable and each one of them has at least 200 corresponding video examples, with an average of 3552 training videos per entity.

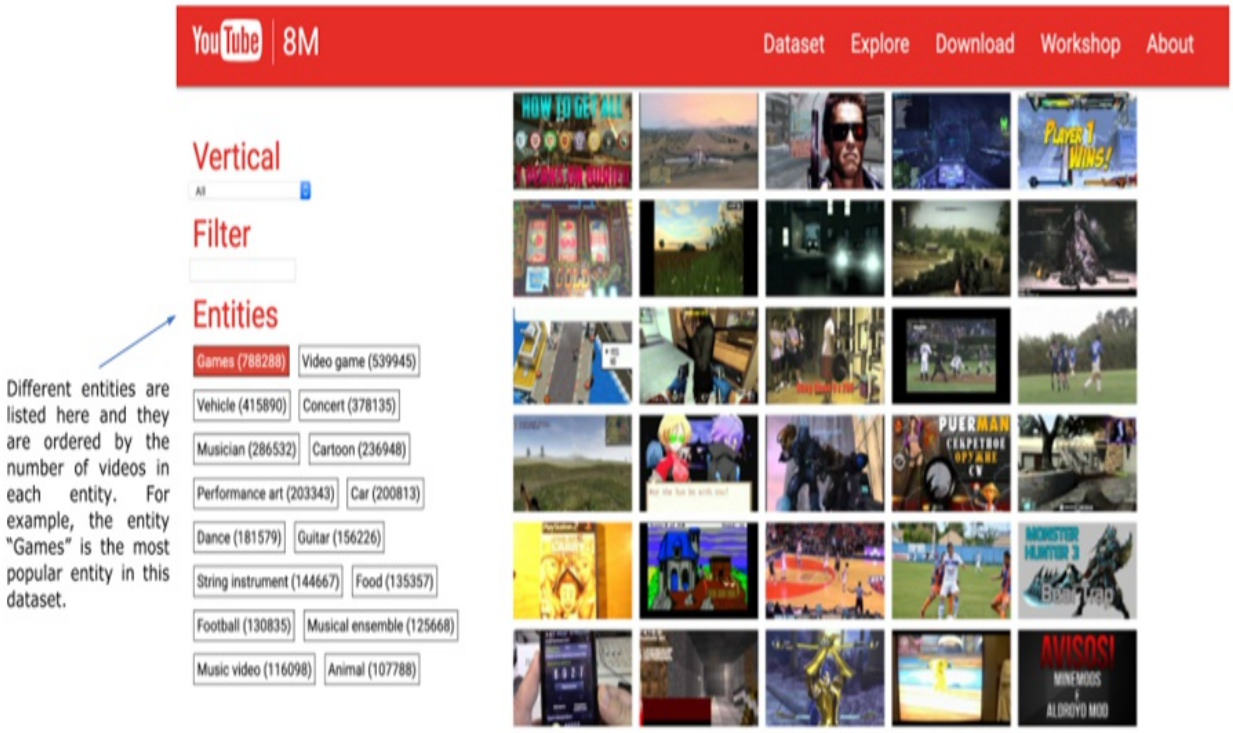
When the raters are identifying the entities in the videos, they are given a guideline to assess how specific and visually recognizable each entity is, on a discrete scale from 1 to 5 where 1 can be easily identified by a layperson, as shown in Figure 3.2.

Figure 3.2 A screenshot of the question and guideline displayed to human raters when identifying the entities in the YouTube videos to assess how visually recognizable each entity is. Source:



Using the online dataset explorer provided by YouTube-8M, we'll see the list of entities with the number of videos that belong to each entity on the right hand side of its entity name as seen in Figure 3.3.

Figure 3.3 A screenshot of the dataset explorer provided by YouTube-8M website where the entities are ordered by the number of videos in each entity.



Note that in the dataset explorer, the entities are ordered by the number of videos in each entity. We'll see that the three most popular entities are Game, Video Game, and Vehicle, respectively, ranging from 415890 to 788288 training examples. The least frequent are Cylinder and Mortar, with 123 and 127 training videos, respectively.

3.2.1 Problem

With this dataset, we'd like to train a machine learning model to tag the main themes of new YouTube videos that the model hasn't seen before. This may be a trivial task for a simpler dataset and machine learning model but it's certainly not the case for YouTube-8M dataset. This dataset comes with precomputed audio-visual features from billions of frames and audio segments so that we don't have to calculate and obtain those on our own which often takes a long time and requires a large amount of computational resources.

Even though it is possible to train a strong baseline model on this dataset in

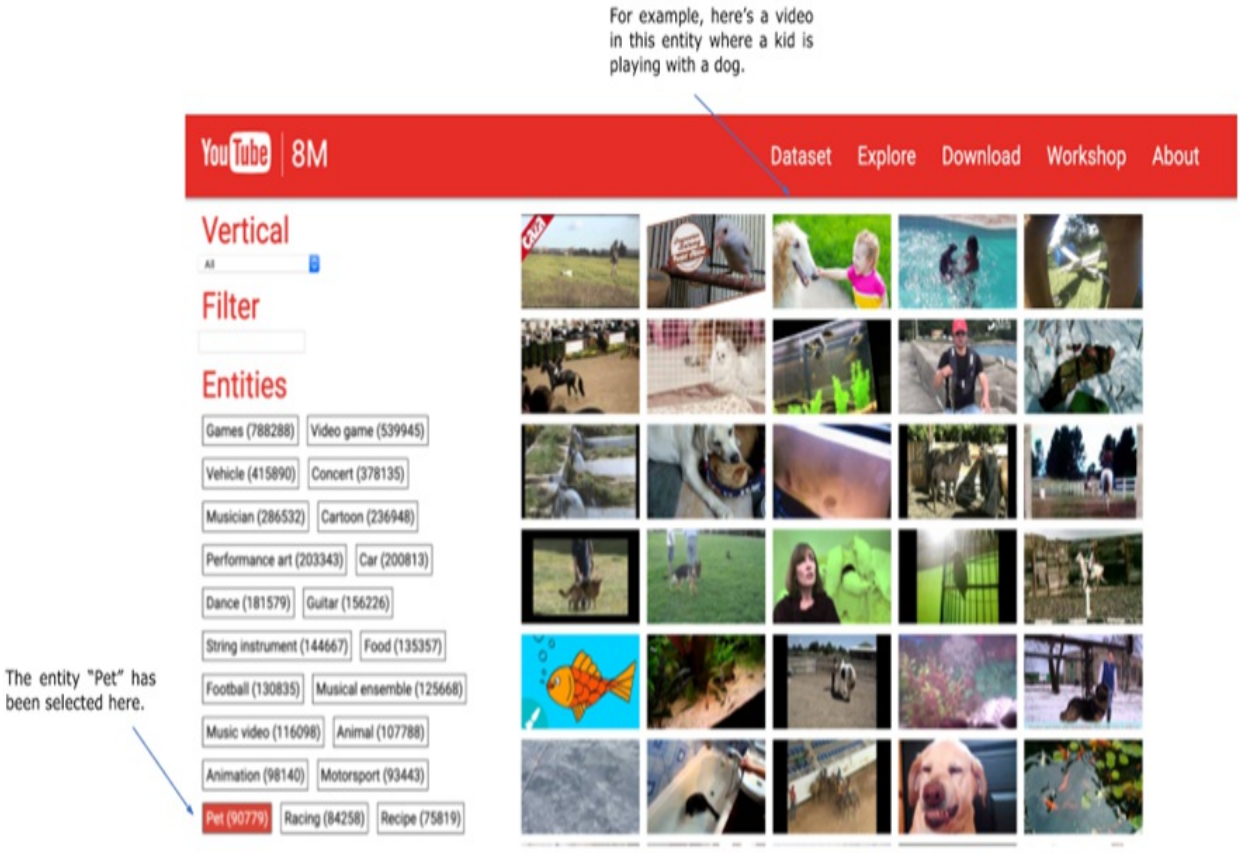
less than a day on a single GPU, the dataset's scale and diversity can enable deep exploration of complex audio-visual models that can take weeks to train. Is there any solution to train this potentially very large model efficiently?

3.2.2 Solution

Let’s first take a look at some of the entities using the data explorer on YouTube-8M website and see if there are any relationships among the entities. For example, are these entities unrelated to each other or do they have some level of overlap in the content? After some exploration, we will then make necessary adjustments to the model to take those relationships into account.

Figure 3.4 shows a list of YouTube videos that belong to the Pet entity. For example, there’s a kid playing with a dog in the third video of the first row.

Figure 3.4 Example videos that belong to the Pet entity.

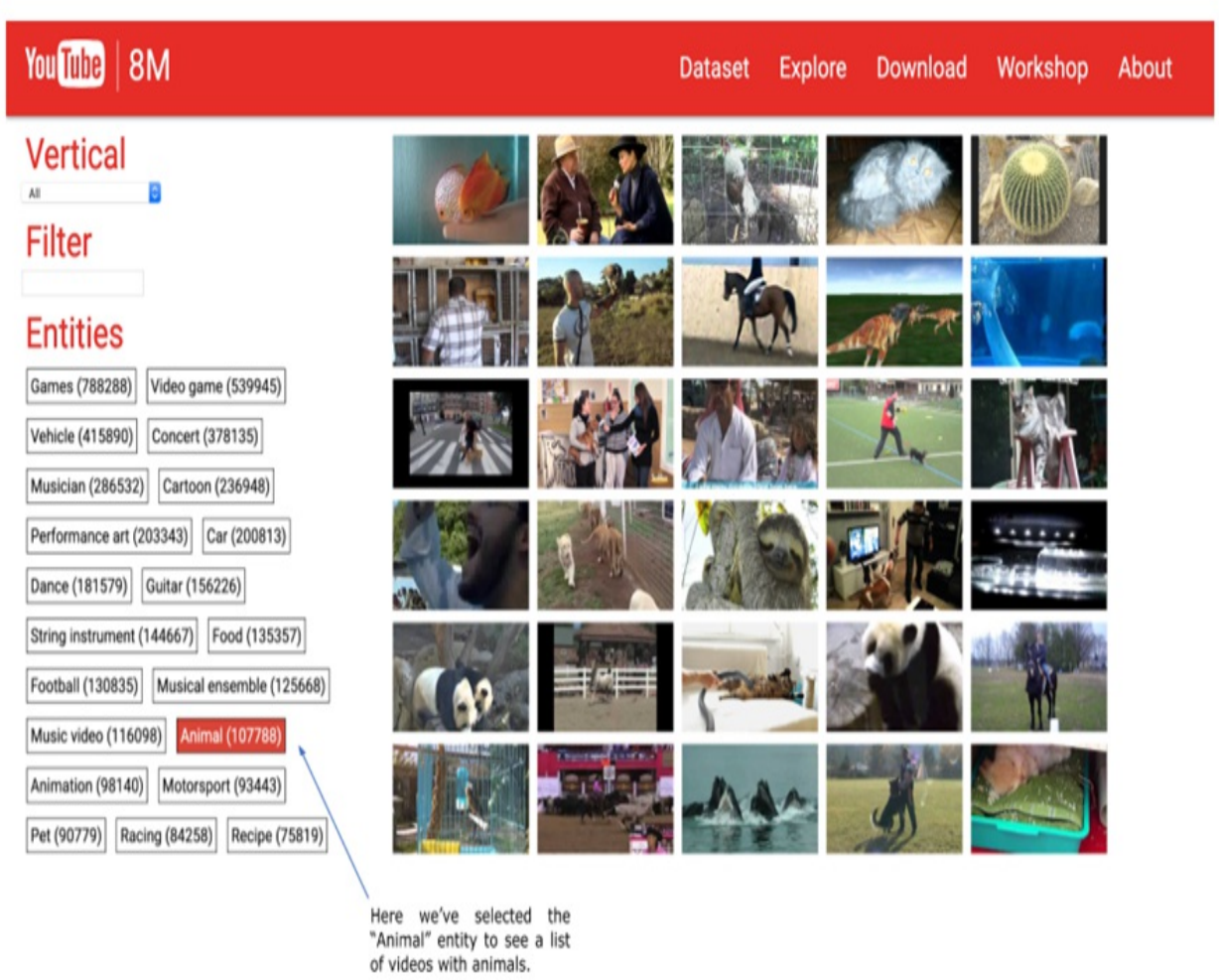


Let's take a look at a similar entity. Figure 3.5 shows a list of YouTube videos that belong to the Animal entity. We can see animals such as fishes, horses, and pandas in this entity. Interestingly, there's a cat getting cleaned by a vacuum in the third video of the fifth row. One might guess that this video is probably in the Pet entity as well since a cat can also be a pet if it's adopted by human-beings.

If we'd like to build machine learning models on this, we may do some additional feature engineering before fitting the model with the dataset directly. For example, we may combine the audio-visual features of these two entities into a derived feature since they provide similar information and have overlap, which can boost the model's performance depending on the specific machine learning model we selected. If we continue exploring the combinations of the existing audio-visual features in the entities or perform a huge number of feature engineering steps, we may no longer be able to train a

machine learning model on this dataset in less than a day on a single GPU.

Figure 3.5 Example videos that belong to the Animal entity.



If we are using a deep learning model instead of traditional machine learning models that require a lot of feature engineering and exploration of the dataset, the model itself would learn the underlying relationships among features, e.g. audio-visual features among similar entities. Each neural network layer in the model architecture consists of vectors of weights and biases that represent a trained neural network layer that will get updated over training iterations as it gathers more knowledge from the dataset.

For example, if we only use 10 out of the 3,862 entities and build a LeNet model such as shown in Figure 3.6 that could classify new YouTube videos into one of the 10 selected entities. At a high level, LeNet consists of a

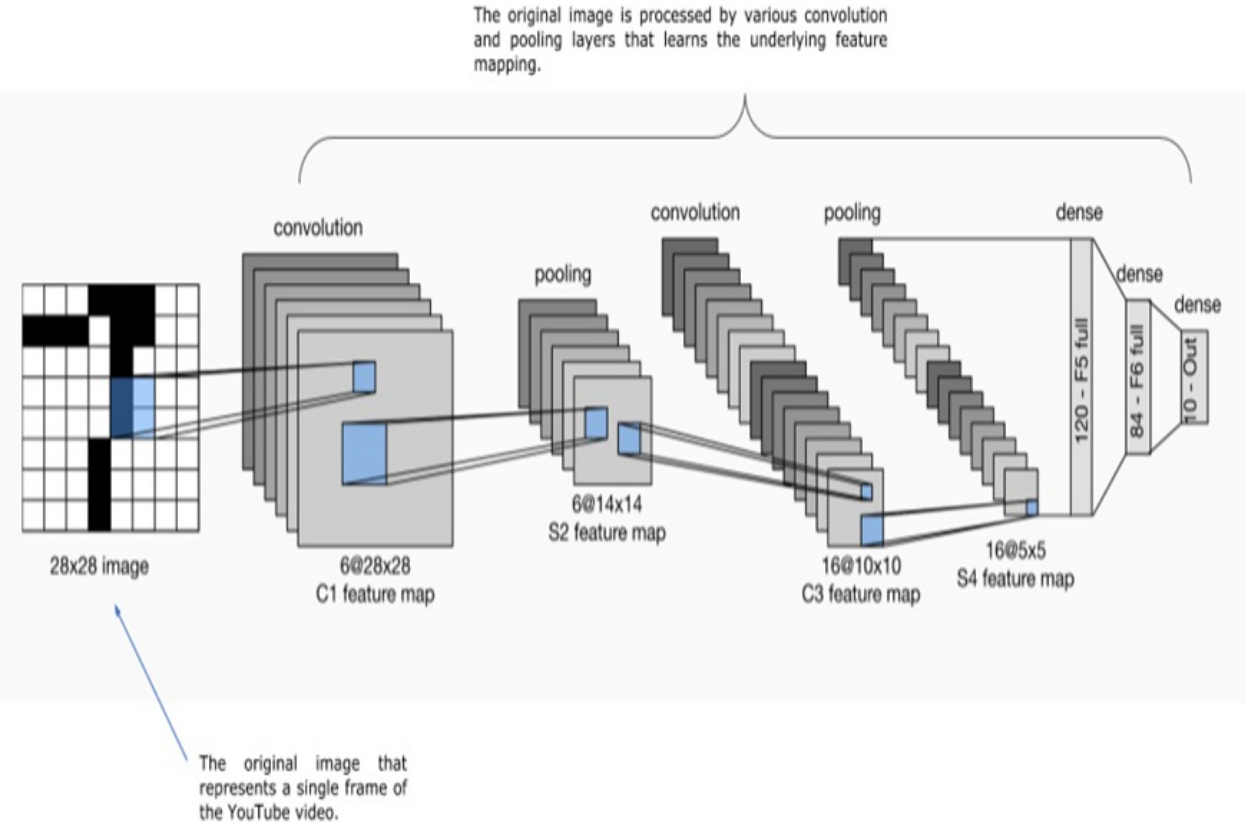
convolutional encoder consisting of two convolutional layers and a dense block consisting of three fully-connected layers. For simplicity, in this case we assume that each individual frame from the videos is a 28x28 image and it will be processed by various convolution and pooling layers that learns the underlying feature mapping between the audio-visual features and the entities.

Note Brief history on LeNet

LeNet is one of the first published Convolutional Neural Networks (CNNs) to capture wide attention for its performance on computer vision tasks. It was introduced by Yann LeCun who was a researcher at AT&T Bell Labs in order to recognize handwritten digits in images. In 1989, LeCun published the first study that successfully trained CNNs via backpropagation after a decade of research and development.

At that time LeNet achieved outstanding results matching the performance of Support Vector Machines (SVMs) which is the dominant approach in supervised machine learning algorithms.

Figure 3.6 LeNet model architecture that could be used to classify new YouTube videos into one of the 10 selected entities. Source: Aston Zhang, et al., licensed under Creative Commons Attribution-ShareAlike 4.0 International Public License.



In fact, those learned feature maps contain *parameters* that are related to the model. These parameters are in the forms of numeric vectors that are used as weights and biases for this layer of model representation. For each training iteration, the model will take every frame in the YouTube videos as features, calculate the loss, and then update those model parameters to optimize the model's objective so that the relationship between features and the entities can be modeled more closely.

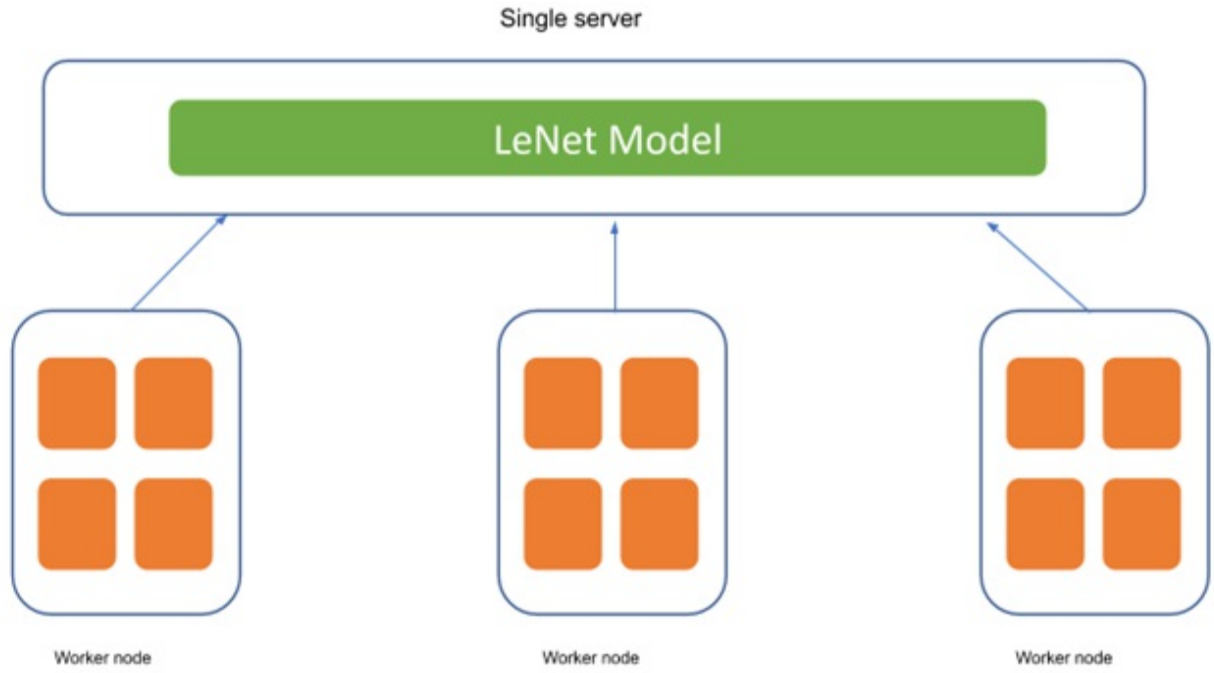
Unfortunately this training process is slow as it involves updating all the parameters in different layers. Here we have two potential solutions to speed up the training process.

Now let's take a look at the first approach. We do want to make an assumption here and we'll remove it later once we discuss a better approach. Let's assume that the model is not too large and we can fit the entire model using existing resources without any possibility of out of memory/disk.

In this case, we can use one dedicated server to store all the LeNet model

parameters and use multiple worker machines to split the computational workloads. An architecture diagram is shown in Figure 3.7.

Figure 3.7 A machine learning training component with a single parameter server.



Each worker node takes a particular part of the dataset to calculate the gradients and then sends the results to the dedicated server to update the LeNet model parameters. Since the worker nodes use isolated computational resources, they can perform the heavy computations asynchronous without having to communicate with each other. Therefore we've achieved around three times speed-up by simply introducing additional worker nodes if other costs like message passing among nodes are neglected.

This dedicated single server responsible for storing and updating the model parameters is called a *parameter-server* and we've actually just designed a more efficient distributed machine learning training system by incorporating the *parameter-server pattern*.

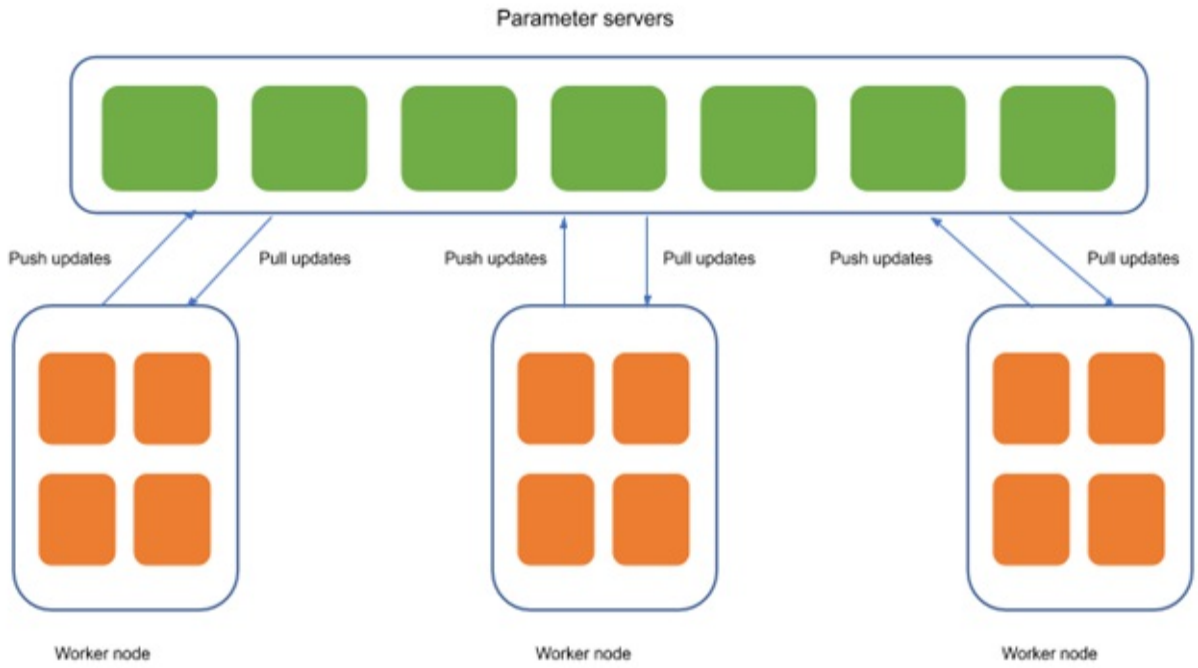
Next comes the real-world challenge. Deep learning models often get really complex and additional layers with custom structure can be added on top of a baseline model. Those complex models usually take a lot of disk space due to

the large number of model parameters in those additional layers and require a lot of computational resources in order to meet the memory footprint requirement that would end up training successfully. What if the model is very large and we cannot fit all of its parameters in a single parameter server?

Let's talk about the second solution that could address the challenges in this situation. We can introduce additional parameter servers where each of the parameter servers is responsible for storing and updating a particular *model partition* whereas each different worker node is responsible for taking a particular part of the dataset to update the model parameters in a model partition.

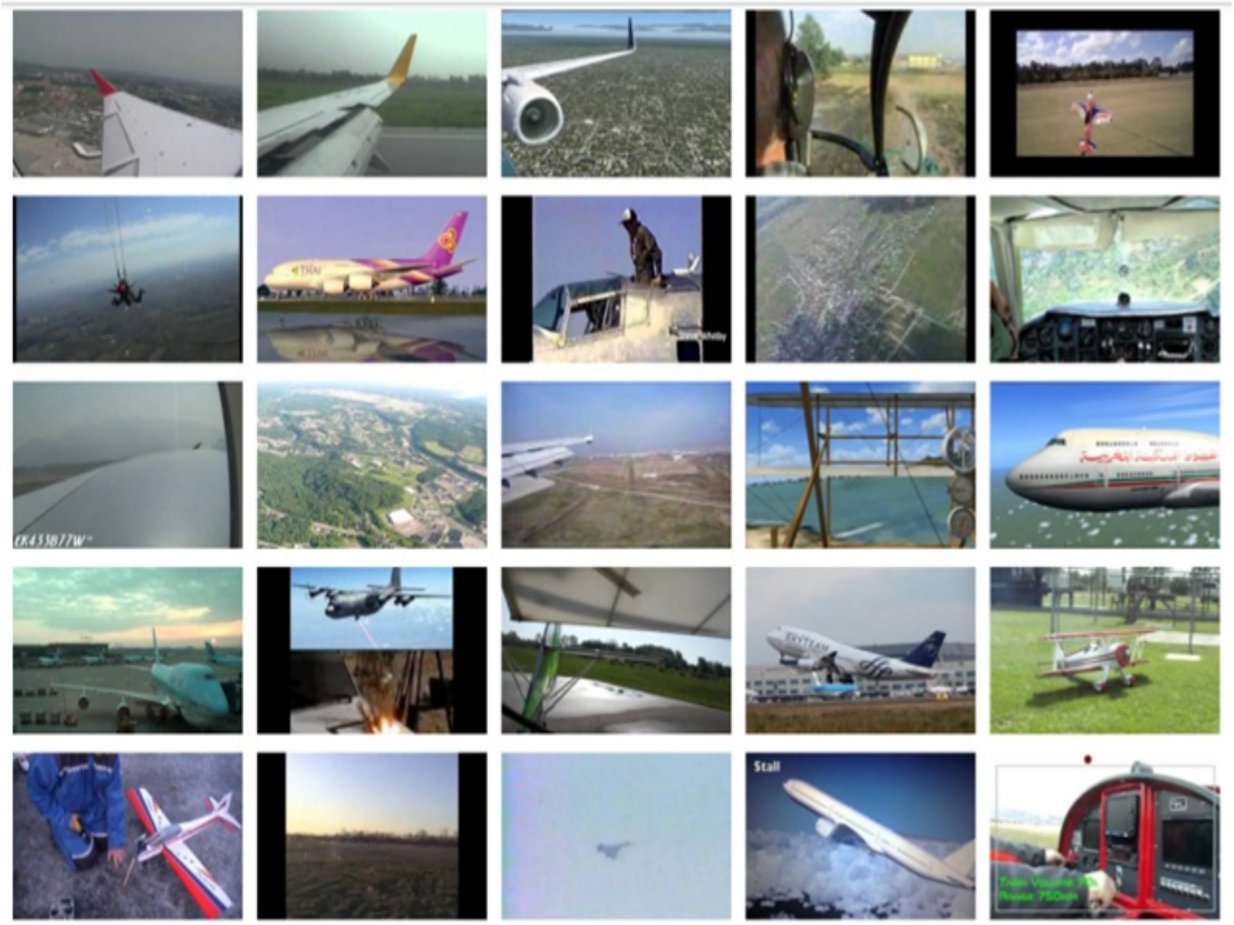
An architecture diagram of this pattern using multiple parameter servers can be found in Figure 3.8, which is different from Figure 3.7 where a single server is used to store all the LeNet model parameters and use multiple worker machines to split the computational workloads in Figure 3.7. Each worker node takes a subset of the dataset, performs calculations required in each of the neural network layers, and then sends the calculated gradients to update one model partition that's stored in one of the parameter servers. Note that since all workers perform calculations in an asynchronous fashion, the model partitions that each worker node uses to calculate the gradients may not be up-to-date. To guarantee that the model partitions each worker node is using or each parameter server is storing are the most recent, we will have to constantly *pull and push* the updates of the model among each worker nodes.

Figure 3.8 A machine learning training component with multiple parameter servers.



With the help of parameter servers, we could effectively resolve the challenges we have when building a machine learning model to tag the main themes of new YouTube videos that the model hasn't seen before. For example, Figure 3.9 shows a list of YouTube videos that are not used for model training and they have been tagged with the Aircraft theme successfully by the trained machine learning model. Even when the model might be too large to fit in a single machine, we could still successfully train the model efficiently. Note that although the parameter server pattern is very useful in this scenario, it is specially designed to make it possible to train models with a lot of parameters.

Figure 3.9 List of new YouTube videos not used for model training and tagged with Aircraft theme.



3.2.3 Discussion

In the previous section, we introduced the parameter server pattern and how it can be used to address the potential challenges for the YouTube-8M video identification application. Even though the parameter server pattern is useful in situations where the model is too large to fit in a single machine and this seems like a straightforward approach to address the challenge, however, in real-world applications, there are still decisions that have to be made in order to make the distributed training system efficient.

Machine learning researchers and DevOps engineers often find it hard to figure out a good ratio between the number of parameter servers and the number of workers for different machine learning applications. There are non-trivial communication costs to send the calculated gradients from workers to parameter servers, as well as costs to pull and push the updates of

the most recent model partitions. For example, if we find the model is getting larger and we add too many parameter servers to the system, the system will end up spending a lot of time communicating between nodes whereas we actually spent a very small amount of time on the computations among neural network layers.

In the next section, we will discuss these practical challenges in more detail and introduce a pattern that will address them so that engineers no longer need to spend time tuning the performance among workers and parameter servers for different types of models.

3.2.4 Exercises

1. If we'd like to train a model with multiple CPUs or GPUs on a single laptop, is this considered distributed training?
2. What's the result of increasing the number of workers or parameter servers?
3. What types of computational resources (e.g. CPUs/GPUs/memory/disk) should we allocate to parameter servers and how much of those different types of resources should we allocate?

3.3 Collective communication pattern: Improving performance when parameter servers become a bottleneck

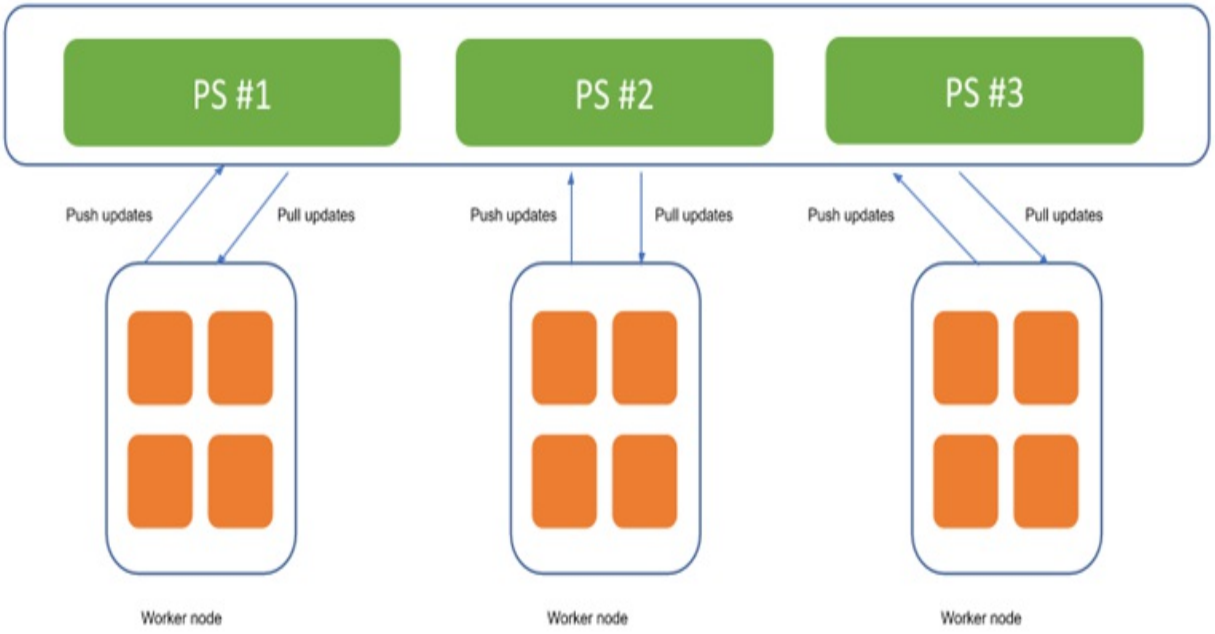
We introduced the parameter server pattern in the previous section that comes handy for situations where the model is too large to fit in a single machine such as the one we would have to build for tagging entities in the 8 millions of YouTube videos. Despite the fact that the parameter servers could be used to handle extremely large and complex models with a large number of parameters, it's non-trivial to incorporate the pattern and design an efficient distributed training system.

Recall that in Section 3.2.3 we discussed that DevOps engineers, who support the distributed machine learning infrastructure for data scientists or analysts, often have a hard time figuring out a good ratio between the number of

parameter servers and the number of workers for different machine learning applications.

For example, let's assume there are three parameter servers and three workers in the model training component of our machine learning system, as shown in Figure 3.10. All of the three workers perform intensive computations asynchronously and then send the calculated gradients to the parameter servers to update the different partitions of model parameters.

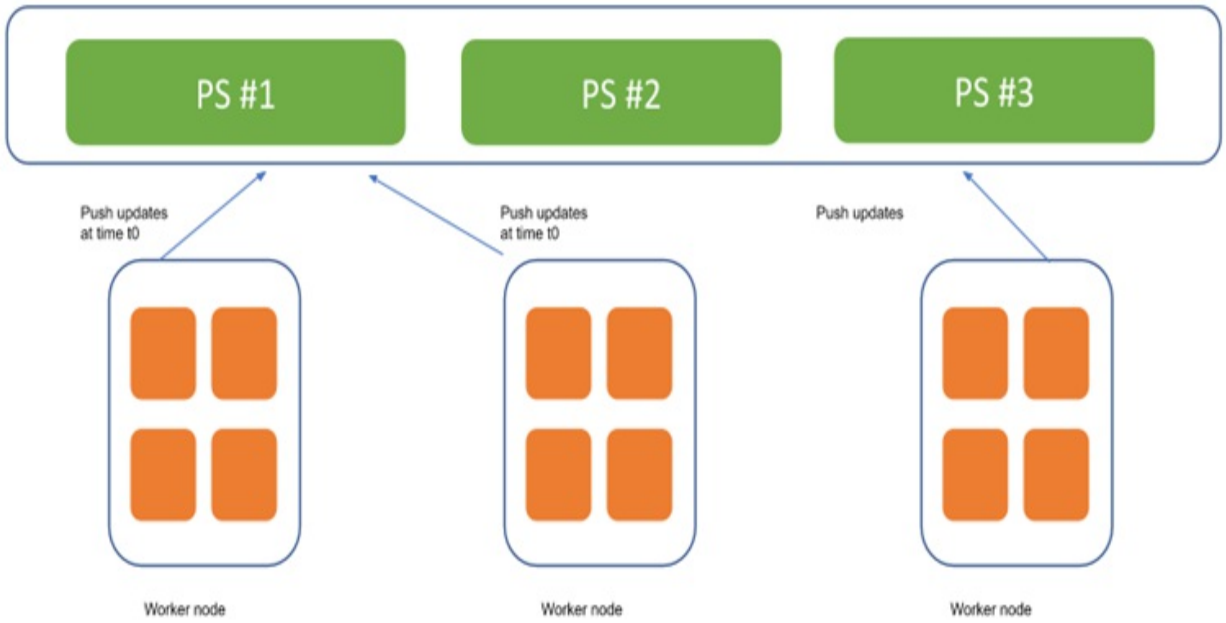
Figure 3.10 Distributed model training component consists of three parameter servers and three worker nodes.



In reality, worker nodes and parameter servers are not one-on-one mapping, in particular if the number of worker nodes is different from the number of parameter servers. In other words, multiple workers may send updates to the same subset of parameter servers.

Now imagine If two workers finished calculating the gradients at the same time and they both wanted to update the model parameters stored in the same parameter server as shown in Figure 3.11.

Figure 3.11 Two of the worker nodes finish calculating gradients and want to push updates to the first parameter server at the same time.



As a result, the two workers are *blocking* each other from sending the gradients to the parameter server. In other words, the gradients from both worker nodes cannot be accepted by the same parameter server at the same time.

3.3.1 Problem

In this case, only two workers are blocking each other when sending gradients to the same parameter server which makes it hard to gather the calculated gradients on time and requires a strategy to resolve the blocking issue. Unfortunately in real-world distributed training systems where parameter servers are incorporated, multiple workers may be sending the gradients at the same time and thus there are many blocking communications that need to be resolved.

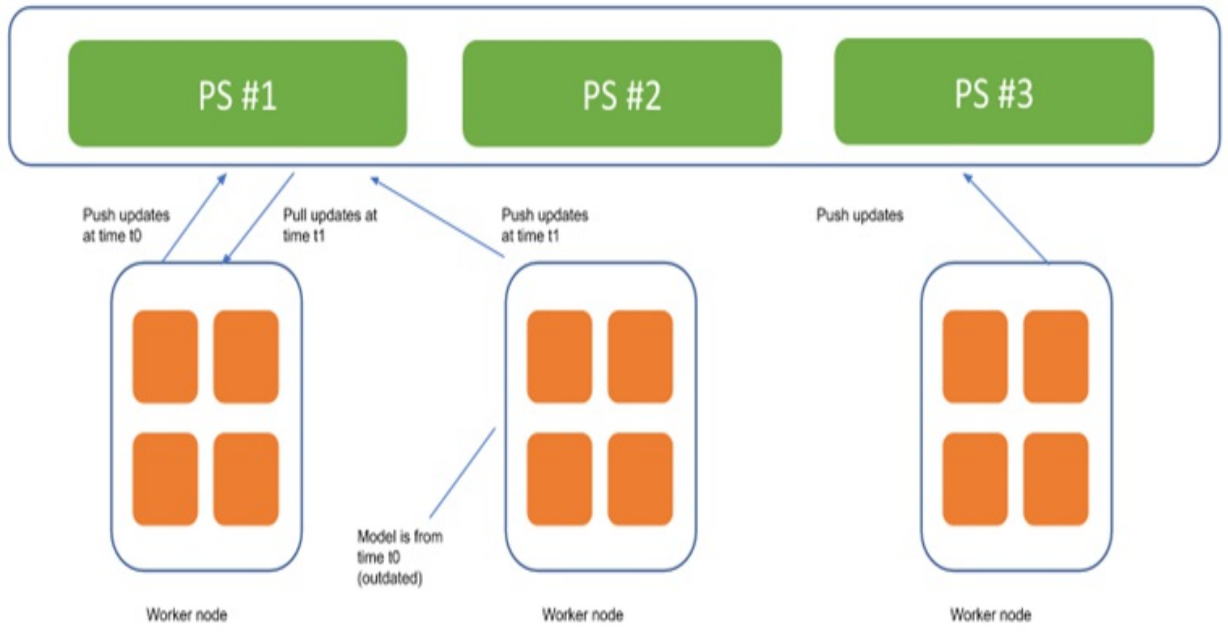
When the ratio between the number of workers and the number of parameter servers is not optimal, for example, many workers are sending gradients to the same parameter server at the same time, the problem gets even worse and eventually the blocking communications among different workers or parameter servers become a bottleneck. Is there a way to prevent this issue from appearing?

3.3.2 Solution

How would we approach this issue? In this situation, the two workers need to figure out an approach to continue. They would have to *reconcile*, decide which worker would take the next step first, and then take turns to send the calculated gradients to that particular parameter server one by one.

In addition, when one worker finishes sending gradients to update the model parameters in that parameter server, the parameter server then starts sending the updated model partition back to that worker so the worker has the most up-to-date model at hand to be fine tuned when feeding it with the incoming data. If at the same time another worker is also sending calculated gradients to that parameter server as shown in Figure 3.12, then another *blocking communication* occurs and they need to reconcile again.

Figure 3.12 One worker is pulling updates while another worker is pushing updates to the same parameter server.

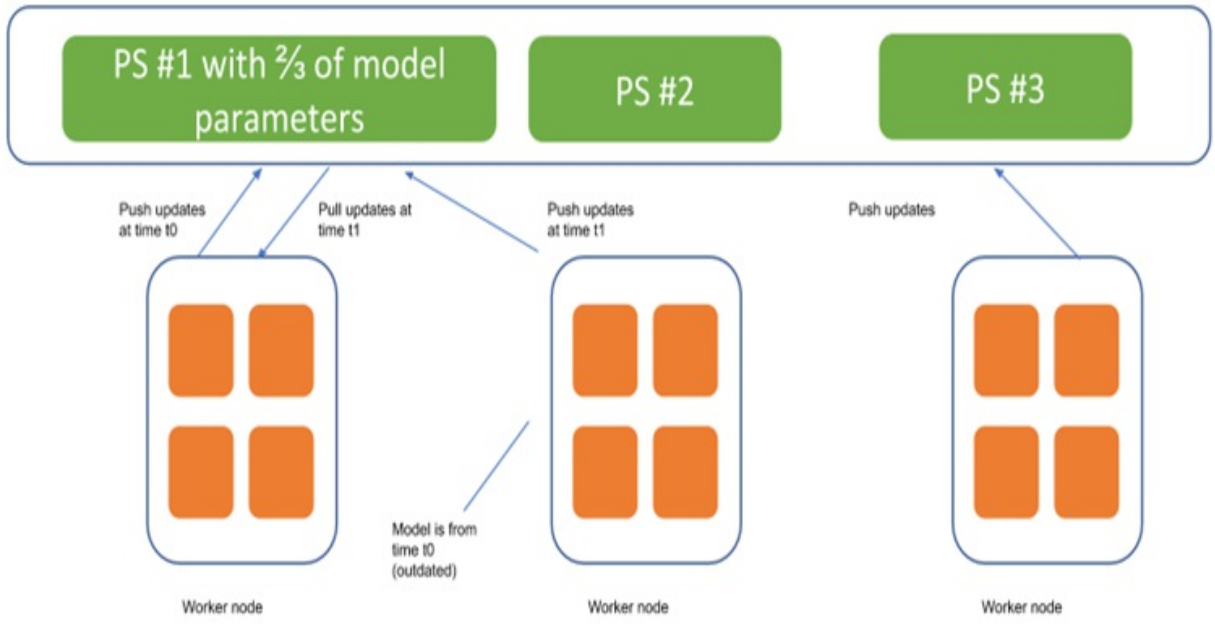


Unfortunately this time the reconciliation may not be so easy to resolve as the worker that is trying to send the calculated gradients may not have used the most updated model when calculating the gradients. This may be okay for cases where the differences between model versions are small but may cause

a huge difference in the statistical performance of the trained model eventually.

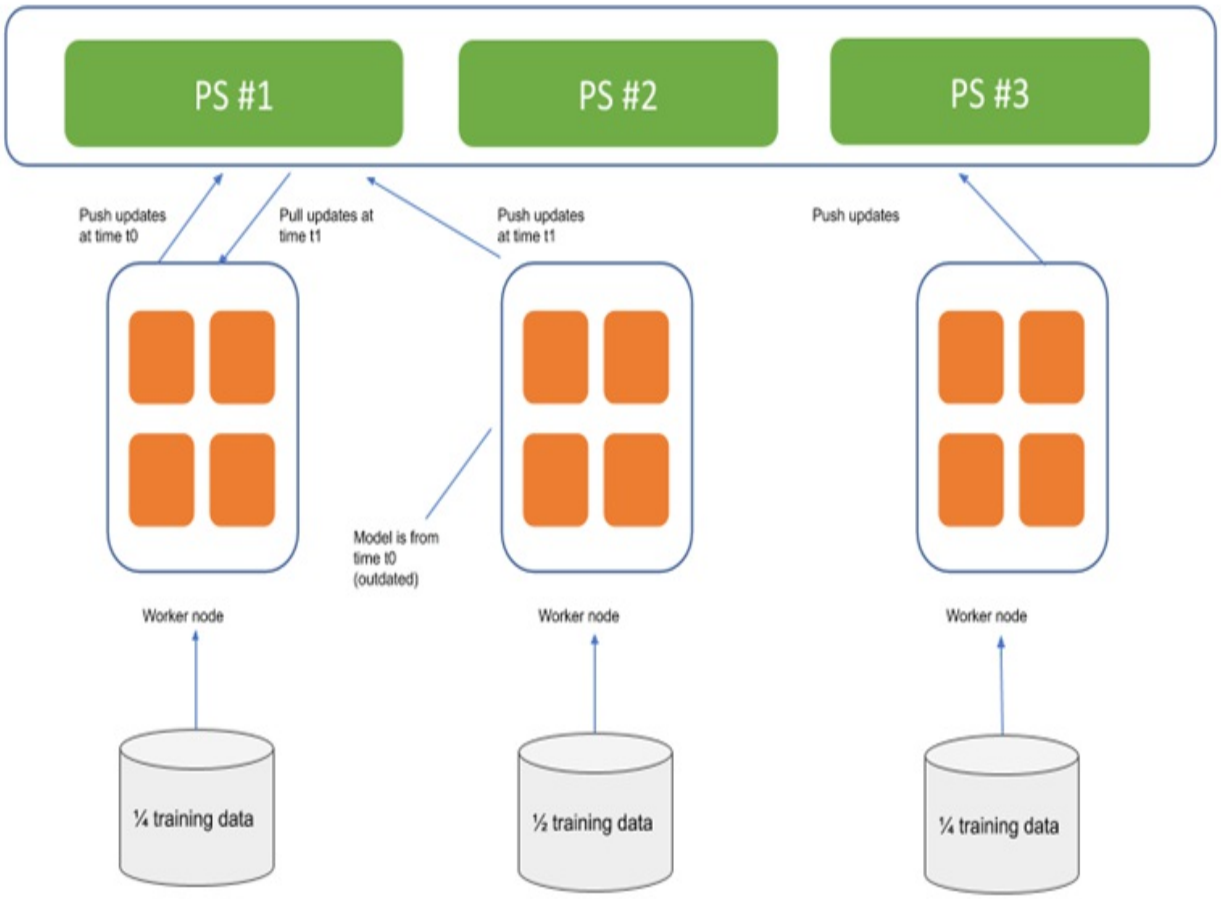
For instance, if each parameter server stores different model partitions unevenly, e.g. the first parameter server stores two third of the model parameters as shown in Figure 3.13, then calculated gradients using this outdated model partition will have a huge impact on the final trained model. In such cases, we may want to drop its calculated gradients and let the other worker send the more update-to-date gradients to the parameter servers.

Figure 3.13 Example of imbalance model partitions where the first parameter server contains two thirds of the entire set of model parameters.



Now comes another challenge - what if those dropped gradients that we consider as outdated are calculated from a larger portion of the entire training data as illustrated in Figure 3.14 and it could take a long time to be re-calculated using the latest model partition? In this case, we'll probably want to keep these gradients instead of dropping them so that we don't waste too much time on re-calculating the gradients.

Figure 3.14 Example of training data where the second worker is trying to push gradients calculated from half of the training data.



In real world distributed machine learning systems with parameter servers, we may encounter a lot of challenges and issues that cannot be resolved completely and when those happen we would have to reconcile and consider trade-offs among the approaches we take.

As the number of workers and parameter servers increases, the cost of reconciliation and communication required to pull and push model parameters among workers and parameter servers becomes non-trivial. The system will end up spending a lot of time communicating between nodes whereas we actually spent a very small amount of time on the computations among neural network layers.

Even though we may have gained a lot of experience understanding the trade-offs and performance differences when applying different ratios and computational resources for parameter servers and workers to our system, it still seems counter-intuitive and time-consuming to tune towards a perfect

system.

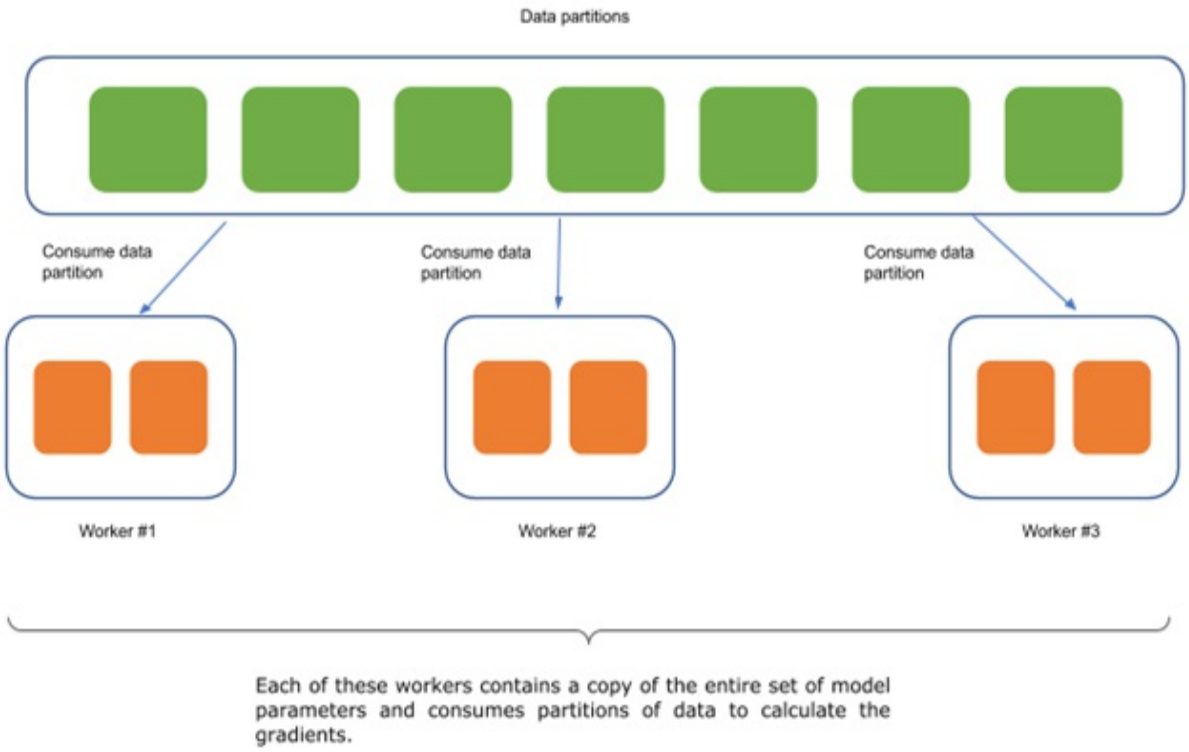
There are even circumstances where some of the workers or parameters fail during training or the network becomes unstable that causes issues when nodes are communicating with each other to push and pull updates. In other words, the parameter server pattern may not be suitable for our particular use case due to lack of expertise or time to work with the underlying distributed infrastructure.

Is there any alternative to this problem? The parameter server pattern may be one of the only good options for large models but here for simplicity and demonstration purposes, let's assume that the model size does not change and the whole model is small enough that could just fit in a single machine. In other words, each single machine has enough disk space to store the model.

With that assumption in mind, what would be an alternative to parameter servers if we just want to improve the performance of distributed training?

Now without parameter servers, there are only worker nodes where each worker node stores a copy of the entire set of model parameters, as shown in Figure 3.15.

Figure 3.15 Distributed model training component with only worker nodes where every worker stores a copy of the entire set of model parameters and consumes partitions of data to calculate the gradients.

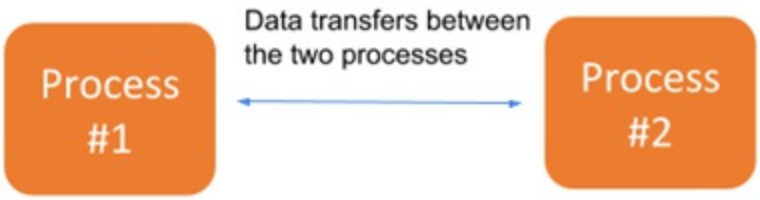


How do we perform model training in this case? Let's recall that every worker consumes some portions of data and calculates the gradients needed to update the model parameters stored locally on this worker node.

Once all of the worker nodes have successfully completed the calculation of gradients, we will need to aggregate all the gradients and make sure every worker's entire set of model parameters is updated based on the aggregated gradients. In other words, each worker should store a copy of the exact same updated model. How do we aggregate all the gradients?

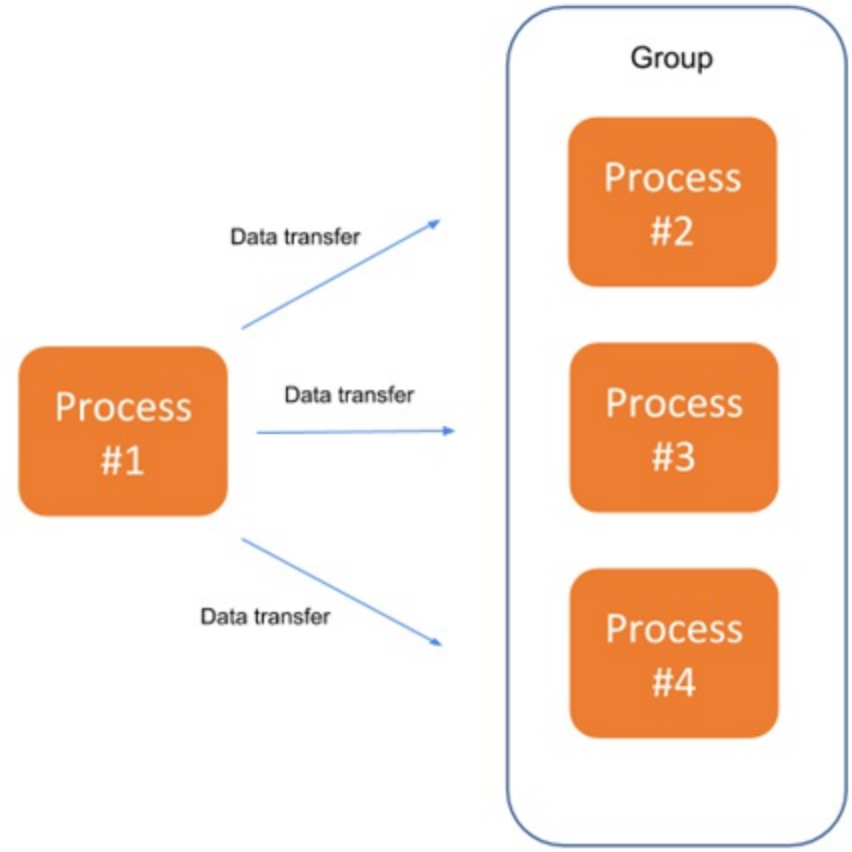
We are already familiar with the process to send gradients from one node to another, for instance, sending the calculated gradients from a worker node to a parameter server to update the model parameters in a particular model partition. In general, that process to transfer data from one process to another is called a *point-to-point communication*, as shown in Figure 3.16. Note that there's no other process involved other than these two processes.

Figure 3.16 Example of point-to-point communication where the data is being transferred between these two processes. Note that there's no other process involved other than these two.



Here in our situation point-to-point communication is not very efficient since there are only worker nodes involved and we need to perform some kind of aggregation on the results from all workers. Fortunately we can leverage another type of communication called *collective communication* that allows communication patterns across all processes in a *group*, which is composed of a subset of all processes. Figure 3.17 is an example of collective communication between one process and a group that consists of three other processes.

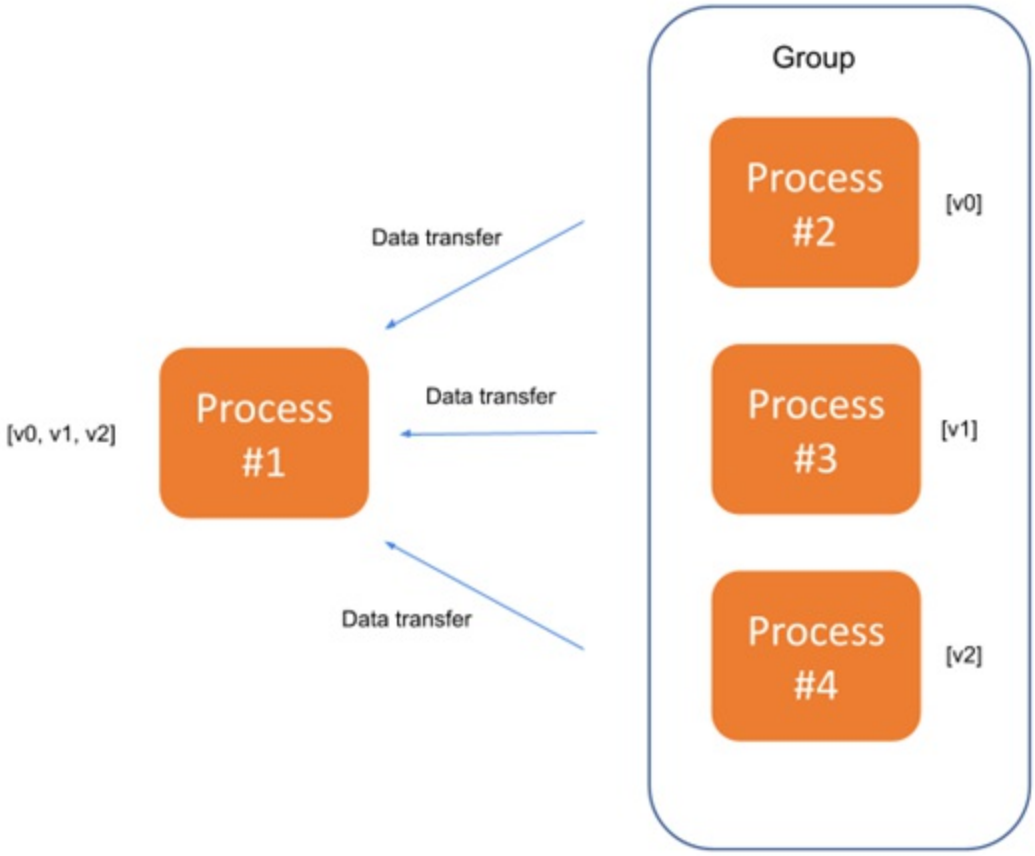
Figure 3.17 Example of collective communication between one process and a group that consists of three other processes.



In this case, each worker node carries the gradients and wants to send them to a group that includes the rest of the worker nodes so that all worker nodes will obtain the results from every worker.

For our machine learning models, we usually perform some kind of aggregate operation on all the received gradients before sending the aggregated result to all the workers. This type of aggregation is called *reduce*, which involves taking a set of numbers into a smaller set of numbers via a function. Examples of such reduce functions are finding the sum, maximum, minimum, or average of the set of numbers, in our case, the gradients we received from all the workers.

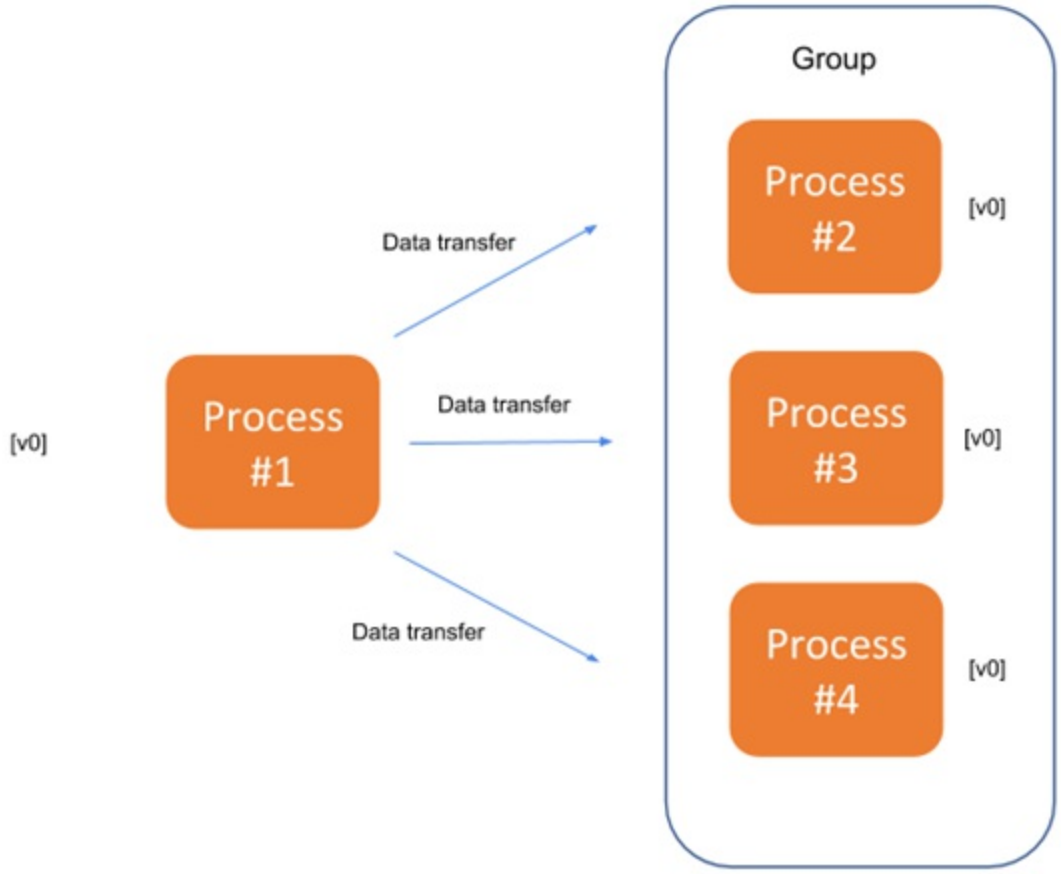
Figure 3.18 Example of an reduce operation with sum as the reduce function.



An example of an reduce operation is illustrated in Figure 3.18. Vectors v0, v1, and v3 in each of the processes in the process group which are then summed over to the first process via a reduce operation.

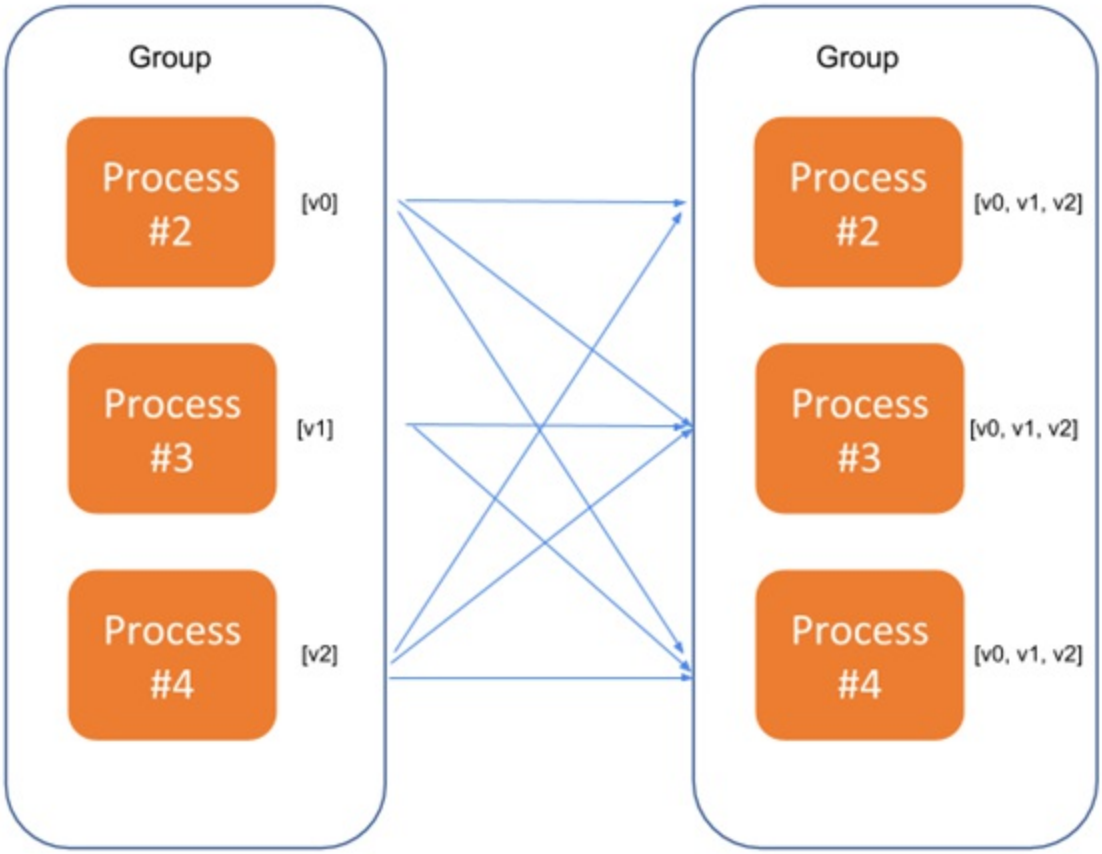
Once the gradients are reduced in a distributed fashion, we then send the reduced gradients to all the workers so that they are on the same page and can update the model parameters in the same way, which makes sure that they have the exact same models. This kind of operation is called *broadcast* operation that is used very often when performing collective communications. Figure 3.19 is an example of a broadcast operation to send a value to every process in the process group.

Figure 3.19 Example of a broadcast operation to send a value to every process in the process group.



Actually the combination of reduce and broadcast operations here is called *allreduce*, which reduces the results based on a specified reduce function and then distributes the reduced results to all processes, as shown in Figure 3.20, in our case, to all the workers so that the model stored in each worker is exactly the same and is the most up-to-date.

Figure 3.20 Example of an allreduce operation to reduce the results on each process in the group and then send the result to every process in the group.



Once we've finished a round of allreduce operation, we can then start the next round by feeding new data to the updated model, calculate gradients, and then perform allreduce operation again to gather all gradients from workers to update the model.

Let's take a break and see what we have just accomplished. We've successfully leveraged the collective communication pattern that takes advantage of the underlying network infrastructure to perform allreduce operations for communicating gradients among multiple workers and allows us to train a medium-sized machine learning model in a distributed fashion. As a result, we no longer need parameter servers and thus there is no communication overhead among parameter servers and workers as we previously discussed. The collective communication pattern is useful in machine learning systems but also in distributed and parallel computing systems where concurrency is applied to computations and communication primitives like broadcast and reduce are critical to communicating among different nodes. We'll apply this pattern in Section 9.2.2.

3.3.3 Discussion

The collective communication pattern is a great alternative to parameter servers when the machine learning model we are building is not too large. As a result, there is no communication overhead among parameter servers and workers and it's no longer necessary to spend a lot of efforts on tuning the ratio between the number of workers and parameter servers.

In other words, we can easily add additional workers to speed up the model training process without having to worry about the performance regression.

There's still one potential problem that's worth mentioning here though. After incorporating the collective communication pattern by applying the allreduce operation, each worker will need to communicate with all of its peer workers, which may slow down the entire training process if the number of workers becomes really large.

Actually since collective communications rely on we communication over the network infrastructure and we still haven't fully leveraged all the benefits of that yet in allreduce operation.

Fortunately, there are better collective communication algorithms that could be used to update the model more efficiently. For example, in *ring-allreduce* algorithm where the process is similar to allreduce operation but the data is transferred in ring-like fashion without the reduce operation, each of the N workers only needs to communicate with two of its peer workers $2 * (N - 1)$ times to update all the model parameters completely. In other words, this algorithm is bandwidth-optimal, meaning that if the aggregated gradients are large enough, it will optimally utilize the underlying network infrastructure.

Both the parameter server pattern and the collective communication pattern make distributed training scalable and efficient. In practice, however, any of the workers or parameter servers that we introduce may not start at all due to the lack of resources and may fail in the middle of distributed training. In the next section, we will introduce patterns that will help with those situations and make the entire distributed training process more reliable.

3.3.4 Exercises

1. Do those block communications only happen among the workers?
2. Do workers update the model parameters on them asynchronously or synchronously?
3. Can you represent an allreduce operation with a composition of other collective communication operations?

3.4 Elasticity and fault-tolerance pattern: Handling unexpected failures when training with limited computational resources

Both the parameter server pattern and collective communication pattern bring the possibility of scaling up the distributed model training process. Parameter servers can be used for handling very large models that don't fit in a single machine so that a large model can be partitioned and stored in multiple parameter servers while individual workers can perform heavy computations and update each individual partition of model parameters asynchronously. On the other hand, when too much communication overhead is observed when using parameter servers, we can leverage the collective communication pattern to speed up the training process for medium-sized models.

Let's assume that our distributed training component is well-designed and can train machine learning models very efficiently that could handle the requirements of different types of models, leveraging patterns like parameter servers or collective communications.

One thing worth mentioning here is that distributed model training is a long running task that usually persists for hours, days, or even weeks. However, like all other types of softwares and systems, this long running task is very vulnerable to unexpected interventions.

Since the model training is a long running process that would usually persist for hours, days, or even weeks, at any minute during this period, the training process may be affected by either internal or external interventions. Below are some examples that are often seen in a distributed model training system:

1. Parts of the dataset is corrupted or cannot be used for training the model successfully;
2. The distributed cluster that the distributed training is depending on may experience unstable or even disconnected network due to weather conditions or human errors;
3. Some of the parameter servers or worker nodes get *preempted,* for example, the computational resources they rely on are re-scheduled to other tasks and nodes that have higher priority.

3.4.1 Problem

When these types of unexpected interventions happen, if there are no actions taken to address them, the issues will start to accumulate. For example, in the first example above, since all workers use the same logic to consume the data to fit the model, when they see corrupted data that the training code is not able to handle, they will all fail eventually. When the network becomes unstable, the communications among parameter servers and workers will hang and until the network recovers. Furthermore, in the third example above when the parameter servers or worker nodes are preempted, the entire training will be forced to stop and lead to unrecoverable failure.

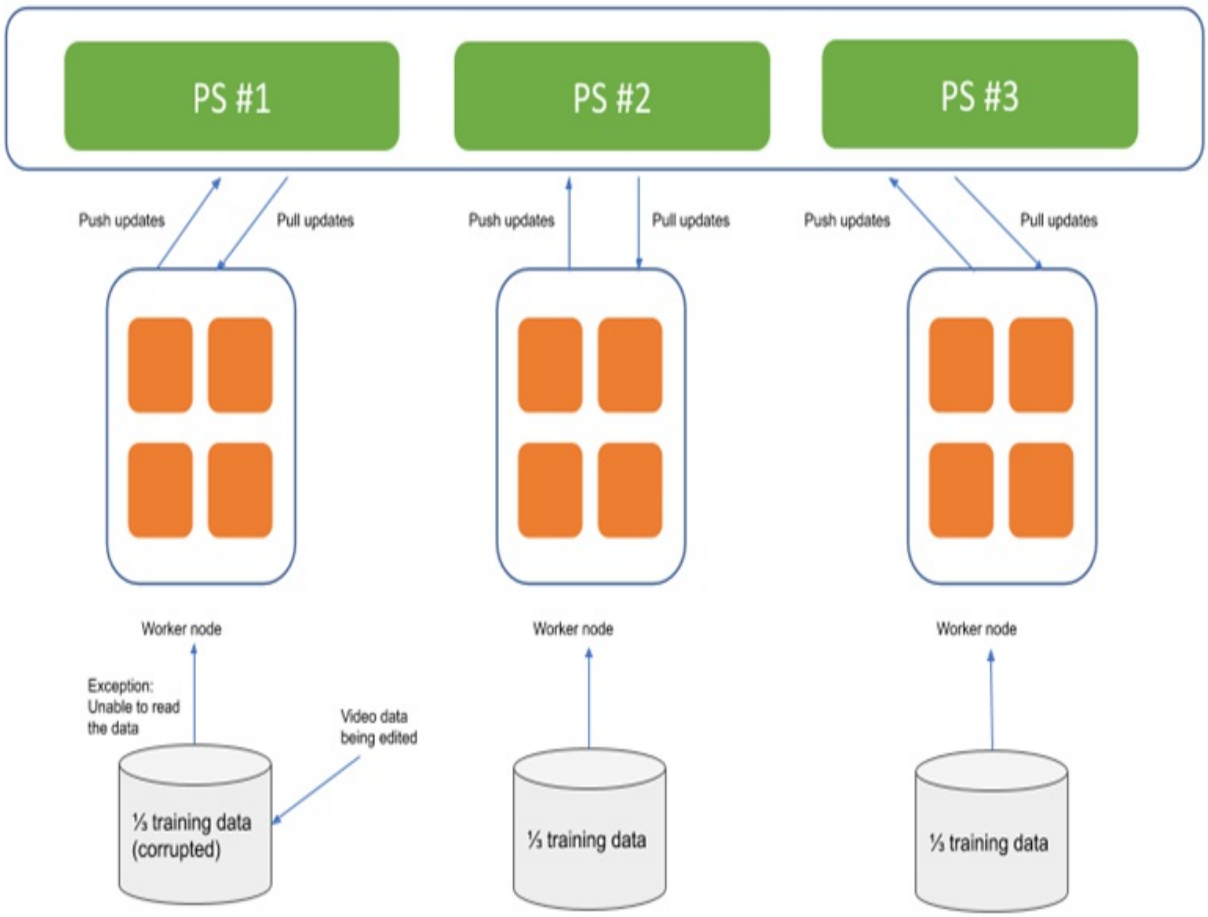
What should we do to help the distributed training system recover in those situations and is there a way to prevent those unexpected failures from happening?

3.4.2 Solution

First of all, let's take a look at the first situation. Assume the training process encounters a batch of data that's "corrupted", for example in Figure 3.21, some of the videos in YouTube-8M data are accidentally modified using some third-party video editing softwares after they were downloaded from the original source while the first worker node is trying to read those portions of the data to feed the model. The machine learning model object that has been initialized earlier cannot be fed using the edited and incompatible video data.

Figure 3.21 Example of a worker encounters new batches of training that's being edited and

cannot be consumed successfully.



When this happens, the training process encounters an *unexpected failure* as the existing code does not contain the logic to handle the format of edited or "corrupted" dataset. In other words, we need to modify the distributed model training logic to be able to handle this situation and then re-train the model from scratch.

Now let's start the distributed training process again and see if everything would work well. We can now skip the batches of data that we found "corrupted" and continue to train the machine learning model using the next batches of the remaining data.

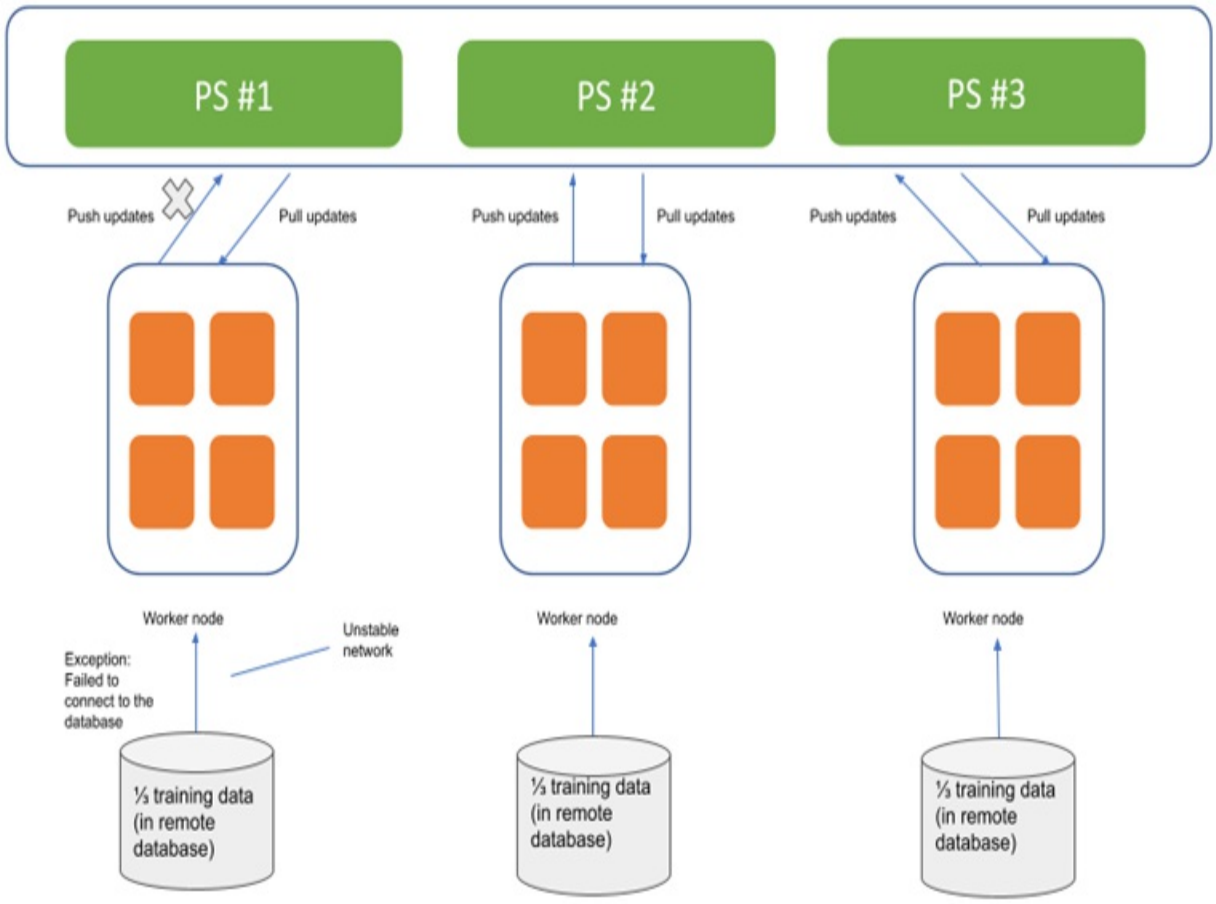
Unfortunately, after the model has been trained for hours using half of the data, we start to realize that the new batches of data are being consumed much slowly than earlier. After some digging and communicating with the

DevOps team, we observe that the network has become extremely unstable due to an incoming storm in one of our data centers! This is the second scenario that we mentioned earlier.

Let’s first take a look at the impact of this on the part where we feed the model with training data. If our dataset is residing somewhere remotely instead of having been downloaded to local machines earlier as shown in Figure 3.22, the training process would be stuck there waiting for the successful connection with the remote database.

One thing we should do while waiting is to *checkpoint or* store the current trained model parameters, and then pause the training process, which would allow us to easily resume the training process when the network becomes stable again.

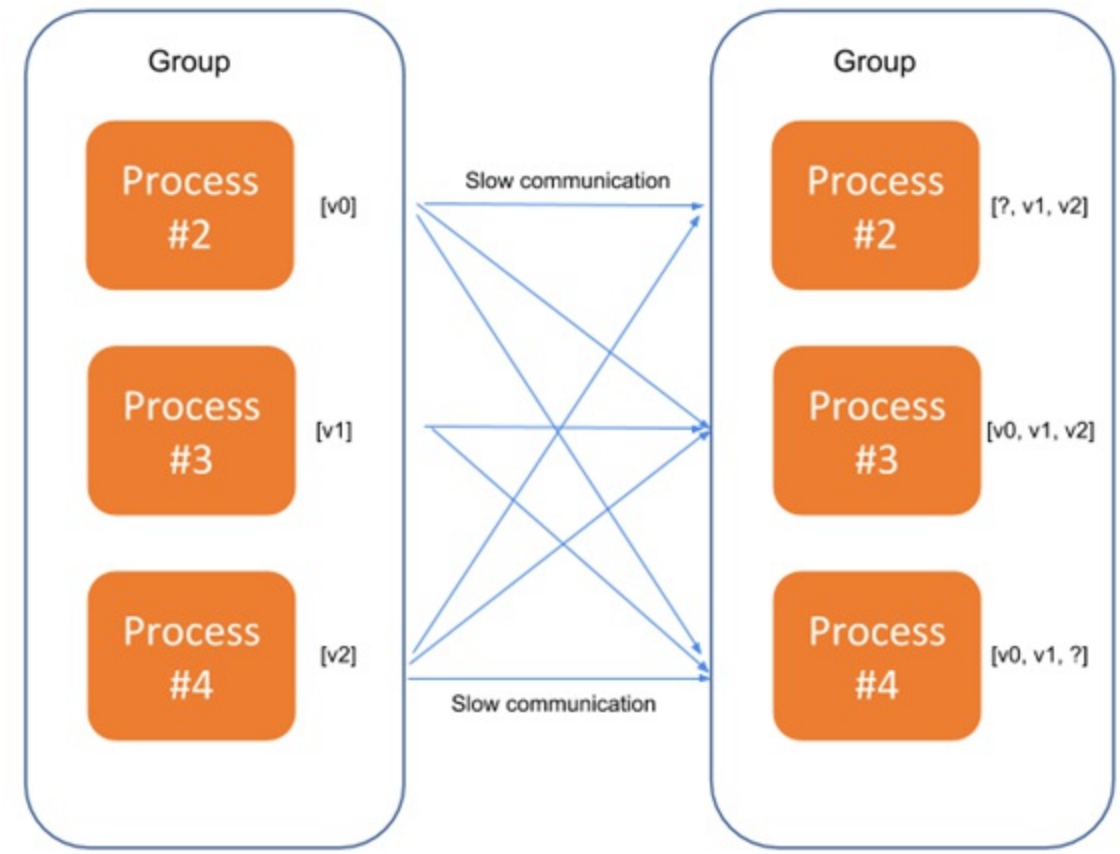
Figure 3.22 Example of a worker encounters an unstable network while fetching the data from a remote database.



Is there other impact from the unstable network? One thing we've neglected is that we also rely on network performance for worker and parameter server nodes to communicate with each other for sending the calculated gradients and updating the model parameters.

Recall that if the collective communication pattern is incorporated, the training process is synchronous, in other words, one worker's communication blocks other workers' communications since we would need to obtain all gradients from all workers in order to aggregate the results that can be used to update the model parameters. If one or more workers start to become slow in communicating with each other, the cascading effect would eventually lead to a stuck training process.

Figure 3.23 Example of allreduce process with slow communications due to the unstable network that blocks the entire training process.



For example, in Figure 3.23, three worker processes in the same process group are performing an allreduce operation. Two of the communications become slow due to the unstable network the underlying distributed cluster is experiencing. As a result, two of the processes that depend on those affected slow communications would not receive some of the values on time (denoted by question marks in the diagram) and the entire allreduce operation is stuck until everything is received.

Is there anything we can do to continue training without being affected by the degrading network performance from individual nodes? In this case, we can first abandon the two worker processes that are experiencing slow network connection and abandon the current allreduce operation. Given the nature of the collective communication pattern we incorporated, the remaining workers would still have the exact same copy of the model so that we can continue the training process by *reconstructing a new worker process group* that consists of the remaining workers and then performing allreduce operation again.

In fact, this would actually be the approach to deal with the case where some worker nodes get *preempted*, for example, the computational resources they rely on are re-scheduled to other tasks and nodes that have higher priority. When those workers get preempted, we reconstruct the worker process group and then perform the allreduce operation.

This approach would allow us to avoid wasting resources to train the model from scratch when unexpected failures happen. Instead, we can pick up the training process from where it's paused and leverage the existing workers that we already allocated computational resources to. If we have additional resources, we can also easily add additional workers and then reconstruct the worker process groups to train more efficiently. In other words, we can easily scale the distributed training system up and down so that the entire system is *elastic* to changes of the amount of available resources. Although we discussed the use cases of elasticity and fault-tolerance in machine learning systems, many other distributed systems also apply the same idea to make sure the systems in place are reliable and scalable.

3.4.3 Discussion

Now that we've successfully continued and recovered the distributed training process without wasting the resources we've used previously to calculate the gradients from each worker. What if our distributed training is leveraging parameter servers instead of collective communications with only workers?

Recall that when parameter servers are used, each of the parameter servers stores a model partition that contains a subset of the complete set of model parameters. If we need to abandon any of the workers or parameter servers, for example, when some of the communications failed or stuck due to an unstable network in one parameter server or the workers get pre-empted, we would need to checkpoint the model partition in the failed nodes and then *re-partition* the model partitions to the parameter servers that are still alive.

In reality, there are still many challenges involved. For example, how do we checkpoint the model partitions and where do we save them to? How often should we checkpoint them to make sure they are as recent as possible?

3.4.4 Exercises

1. What is the most important thing to save in a checkpoint in case any failures happen in the future?
2. When we abandon the workers that are stuck or unable to recover without having time to make model checkpoints, where should we obtain the latest model, assuming that we are using the collective communication pattern?

3.5 References

1. InfiniBand: https://en.wikipedia.org/wiki/InfiniBand
2. RDMA: https://en.wikipedia.org/wiki/Remote_direct_memory_access
3. YouTube-8M: http://research.google.com/youtube8m/
4. Dataset explorer provided by YouTube-8M: http://research.google.com/youtube8m/explore.html
5. LeNet: https://en.wikipedia.org/wiki/LeNet
6. Convolutional Neural Networks (CNNs): https://en.wikipedia.org/wiki/Convolutional_neural_network

3.6 Summary

- Distributed model training is very different from the traditional model training process given the size and location of the dataset, the size of the model, the computational resources, and the underlying network infrastructure.
- We've learned to use parameter servers for building large and complex models to store partitions of model parameters on each parameter server.
- If the communications between workers and parameter servers become a bottleneck, we can consider switching to incorporate the collective communication pattern to improve distributed model training performance for small or medium-sized models.
- Unexpected failures happen during distributed model training and there are various approaches we can take to avoid the waste of computational resources.

Answers to exercises:

Section 3.2.4

1. No, because the training happens on a single laptop in this case.
2. The system will end up spending a lot of time communicating between nodes whereas we actually spent a very small amount of time on the computations among neural network layers.
3. We need more disk spaces for parameter servers to store large model partitions and less CPUs/GPUs/memory on them since parameter servers do not perform heavy computations.

Section 3.3.4

1. No, they also appear between workers and parameter servers.
2. Asynchronous.
3. A reduce operation and then a broadcast operation.

Section 3.4.4

1. The most recent model parameters.
2. Under the collective communication pattern, the remaining workers would still have the exact same copy of the model that we can use to continue training.

4 Model serving patterns

This chapter covers

- Using model serving to generate predictions or make inferences on new data with previously trained machine learning models.
- Handling the growing number of model serving requests and achieving horizontal scaling with the help of replicated model serving services.
- Processing large model serving requests by leveraging the sharded services pattern.
- Assessing model serving systems and determining whether event-driven design would be beneficial for improving resource efficiency.

In the previous chapter, we explored some of the challenges involved in the distributed training component. We introduced a couple of practical patterns that can be incorporated into this component. Distributed training is the most critical part of a distributed machine learning system. For example, we've seen challenges when training very large machine learning models that tag main themes in new YouTube videos but cannot fit in a single machine and how we can overcome the difficulty using the parameter server pattern. We also learned how to leverage the collective communication pattern to speed up distributed training for smaller models and avoid the unnecessary communication overhead among parameter servers and workers. Last but not least, we've talked about some of the vulnerabilities that are often seen in distributed machine learning systems due to corrupted datasets, unstable networks, and preempted worker machines and how we can address those issues.

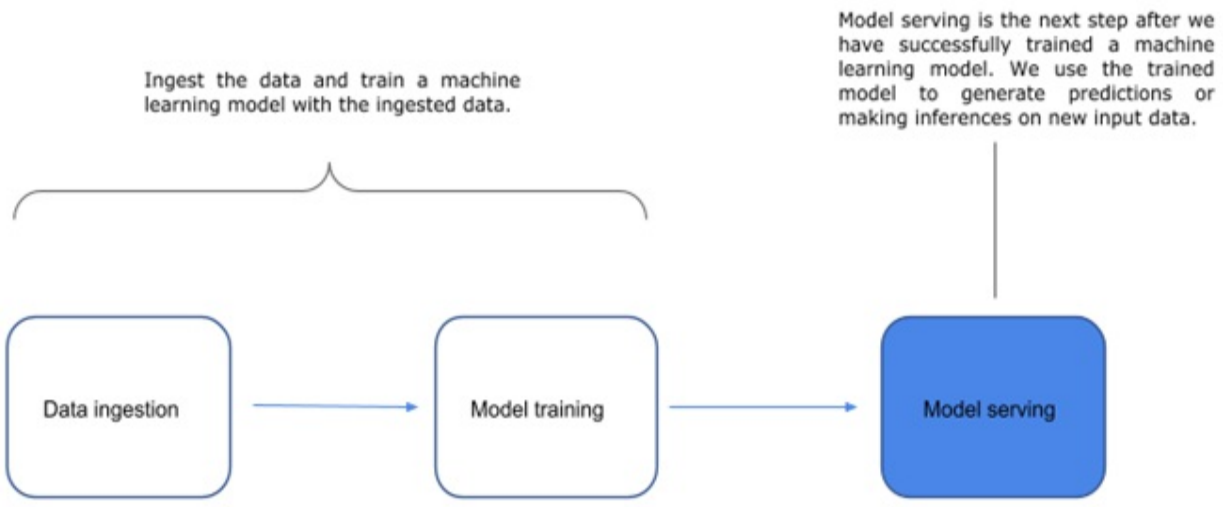
Model serving is the next step after we have successfully trained a machine learning model and is one of the essential steps in a distributed machine learning system. The model serving component needs to be scalable and reliable to handle the growing number of user requests and the growing size of individual requests. It's also essential to know what trade-offs we may see when making different design decisions in order to build a distributed model serving system. In this chapter, we'll explore some of the challenges involved

in distributed model serving systems and introduce a few established patterns adopted heavily in industries. For example, we'll see challenges when handling the growing number of model serving requests and how we can overcome the challenges to achieve horizontal scaling with the help of replicated services. We'll also discuss the sharded services pattern to help the system process large model serving requests. In addition, we'll learn how to assess model serving systems and determine whether event-driven design would be beneficial with real-world scenarios.

4.1 What is model serving?

Model Serving is the process of loading a previously trained machine learning model, generating predictions or making inferences on new input data. It's the step after we've successfully trained a machine learning model. Figure 4.1 below is a diagram showing where model serving fits in the machine learning pipeline.

Figure 4.1 A diagram showing where model serving fits in the machine learning pipeline.



Note that model serving is a general concept that appears in distributed and traditional machine learning applications. In traditional machine learning applications, model serving is usually a single program that runs on a local desktop or machine and generates predictions on new datasets that are not used for model training. Both the dataset and the machine learning model

used should be small enough to fit on a single machine for traditional model serving and they are stored in the local disk of a single machine.

In contrast, distributed model serving happens usually in a cluster of machines. Both the dataset and the trained machine learning model used for model serving can be very large and must be stored in a remote distributed database or partitioned on disks of multiple machines. The differences between traditional model serving and distributed model serving systems is summarized in Table 4.1 below.

Table 4.1 Comparison between traditional model serving and distributed model serving systems

	Traditional Model Serving	Distributed Model Serving System
Computational Resources	Personal laptop or single remote server	Cluster of machines
Dataset Location	Local disk on a single laptop or machine	Remote distributed database or partitioned on disks of multiple machines
Size of Model and Dataset	Small enough fit on a single machine	Medium to large

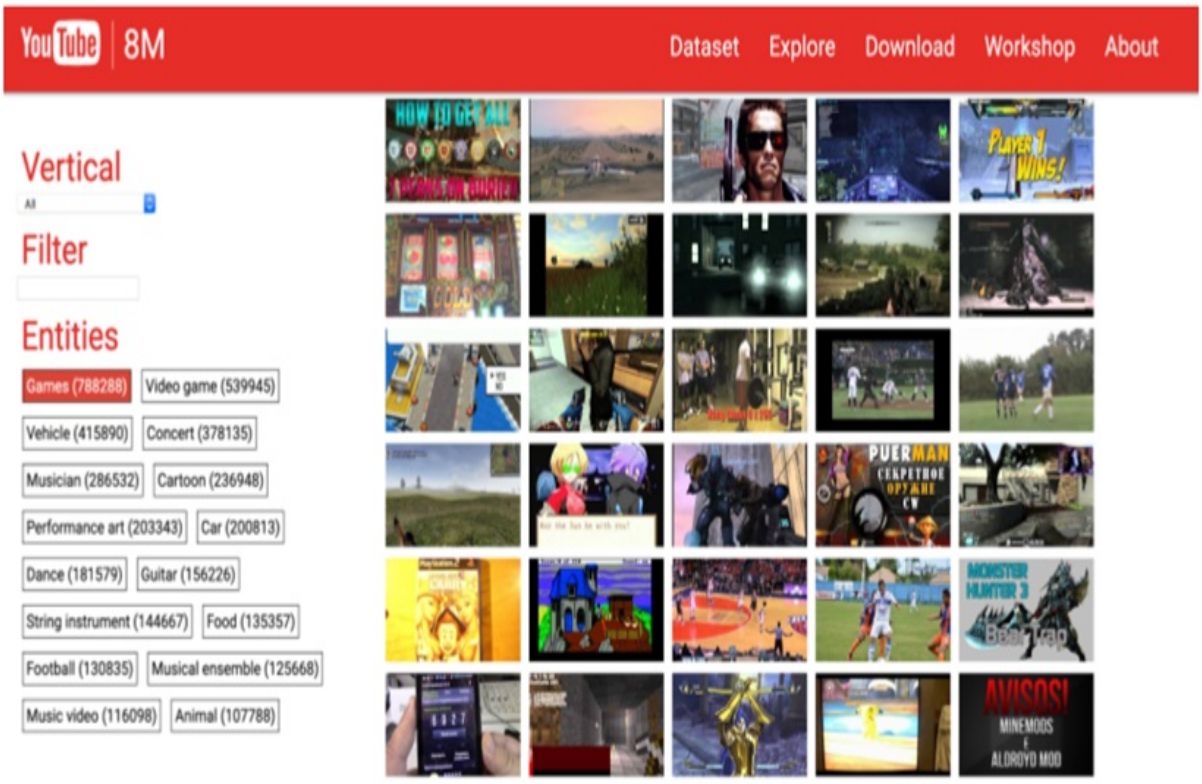
It’s non-trivial to build and manage a distributed model serving system that’s scalable, reliable, and efficient for different use cases. We will illustrate a couple of use cases as well as some established patterns that could address different challenges.

4.2 Replicated services pattern: Handling growing

number of serving requests

Recall that in the previous chapter we've built a machine learning model to tag the main themes of new videos that the model hasn't seen before using the YouTube-8M dataset, which consists of millions of YouTube video IDs, with high-quality machine-generated annotations from a diverse vocabulary of 3,800+ visual entities such as food, car, music, etc. A screenshot of what the videos in the YouTube-8M dataset look like is shown in Figure 4.2 below.

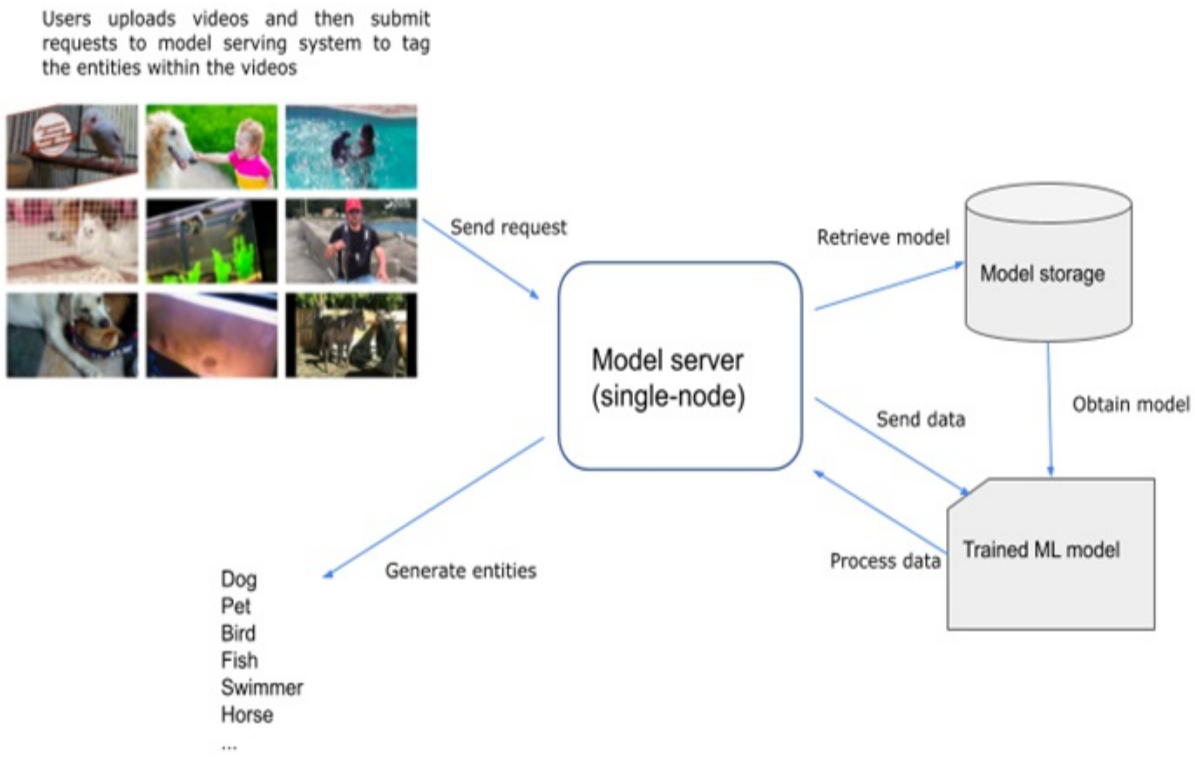
Figure 4.2 A screenshot of what the videos in the YouTube-8M dataset look like.



Now we would like to build a model serving system that allows users to upload new videos and then the system would load the previously trained machine learning model to tag entities/themes that appear in the uploaded videos. Note that the model serving system is stateless so users' requests won't affect the model serving results.

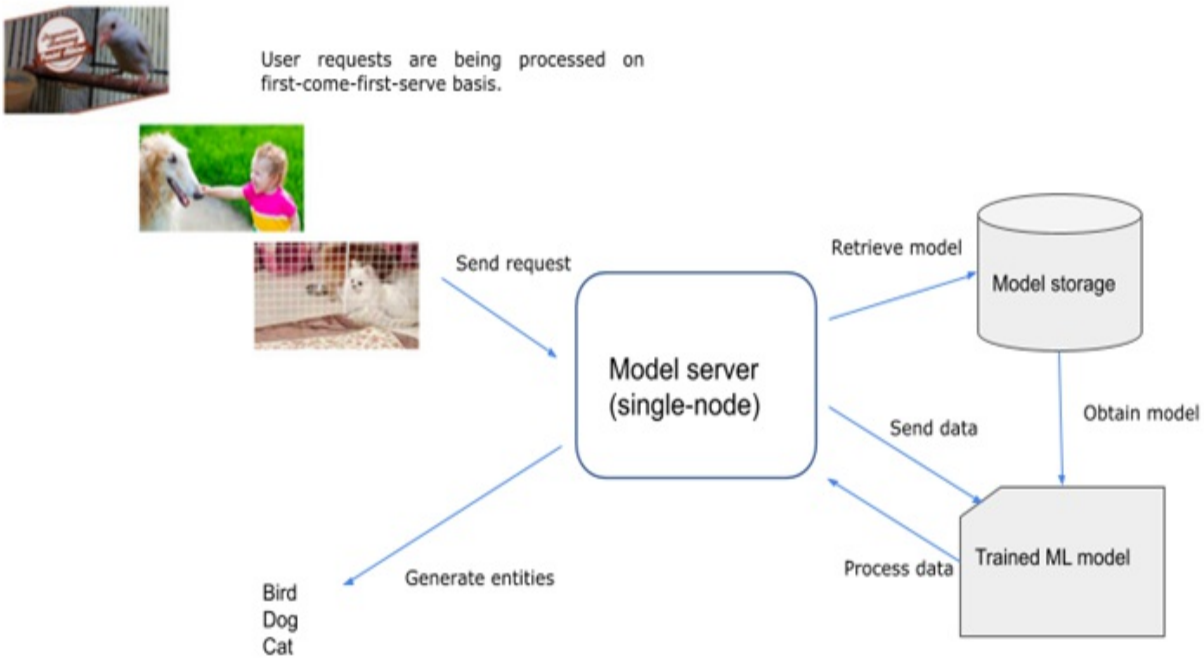
The system basically takes the videos uploaded by users and sends requests to the model server. The model server would then retrieve the previously trained entity tagging machine learning model from the model storage to process the videos and eventually generate possible entities that appear in the videos. A high-level overview of the system is shown in Figure 4.3 below.

Figure 4.3 A high-level architecture diagram of the single-node model serving system.



Note that this initial version of the model server only runs on a single machine and responds to model serving requests from users on a first-come-first-serve basis, as shown in Figure 4.4 below. This approach may work well already if there are only very few users testing the system. However, as the number of users or the number of model serving requests increase, users would experience huge delays while waiting for the system to finish processing any previous requests. In the real-world, this bad user experience would immediately lose our users' interest in continuing using this system.

Figure 4.4 The model server only runs on a single machine and responds to model serving requests from users on a first-come-first-serve basis.



4.2.1 Problem

The system is able to take the videos uploaded by users and then sends the requests to the model server. These model serving requests are being queued and waiting to be processed by the model server.

Unfortunately due to the nature of the single-node model server, it could only serve a limited number of model serving requests on a first-come-first-serve basis. As the number of requests grows in the real-world, the user experience gets bad when users have to wait for a long time to receive the model serving result. All requests are waiting to be processed by the model serving system but the computational resources are bound to this single node.

Is there a better way to handle the model serving requests instead of handling them sequentially?

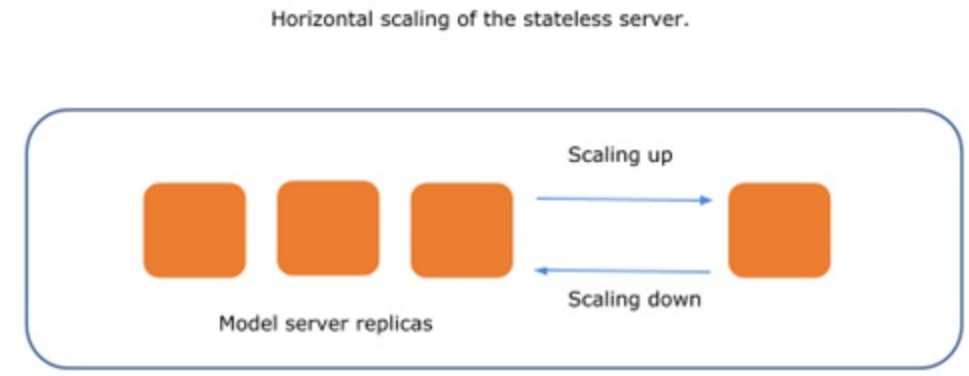
4.2.2 Solution

One fact we've neglected is that the existing model server is stateless, meaning that the model serving results for each request won't be affected by

other requests and the machine learning model can only process a single request. In other words, the model server doesn't require a saved state to operate correctly.

Since the model server is stateless, we can add more server instances to help handle additional user requests without interfering with each other's work as shown in Figure 4.5 below. These additional model server instances are exact copies of the original model server but with different server addresses and each handle different model serving requests. In other words, they are *replicated services* for model serving or *model server replicas* in short.

Figure 4.5 Additional server instances to help handle additional user requests without interfering with each other's work.



This way of adding additional resources into our system with more machines is called *horizontal scaling*. Horizontal scaling systems handle more and more users or traffic by adding more replicas. The opposite of horizontal scaling is *vertical scaling* which is usually implemented by adding computational resources to existing machines.

Note An analogy of horizontal scaling v.s. vertical scaling

One way to look at it is to think of vertical scaling like retiring your sports car and buying a race car when you need more horsepower. While race cars are fast and look amazing, they're not very practical, expensive, and at the end of the day, they can only take you so far before they're running out of gas. Not to mention that there's only one seat and has to be driven on a flat

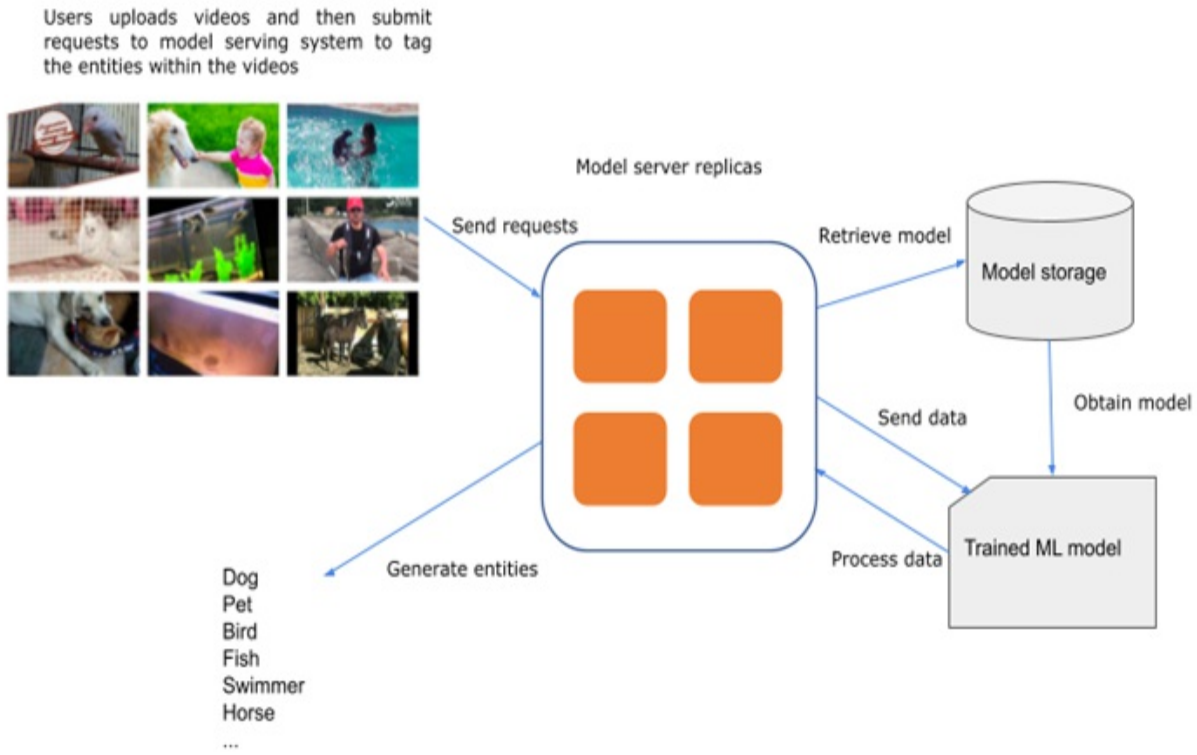
ground. They are really only suitable for racing.

Horizontal scaling gets you that added horsepower – not by favoring sports cars over race cars, but by adding another vehicle to the mix. In fact, you can think of horizontal scaling like several vehicles that could fit a lot of passengers at once. Maybe none of these machines is a race car, but none of them needs to be: across the fleet, you have all the horsepower you need.

Let's get back to our original model serving system, which takes the videos uploaded by users and sends requests to the model server. Unlike our previous design of the model serving system, the system now has multiple model server replicas to process the model serving requests asynchronously. Each of the model server replicas takes a single request, retrieves the previously trained entity tagging machine learning model from the model storage, and then processes the videos in the request to tag possible entities that appear in the videos.

As a result, we've successfully scaled up our model server by adding model server replicas to the existing model serving system. The new architecture is shown in Figure 4.6 below. The model server replicas are capable of handling many requests at a time since each replica can process individual model serving requests independently.

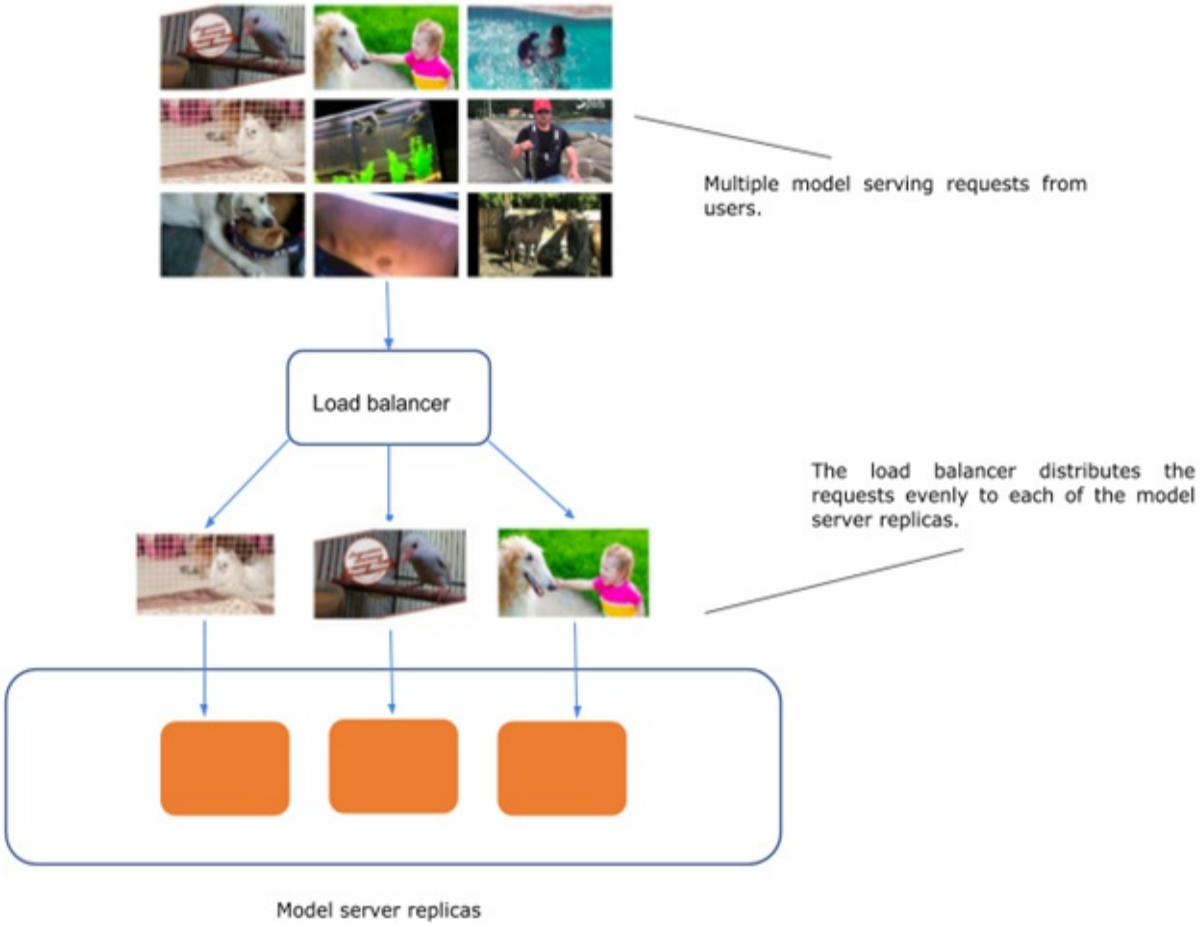
Figure 4.6 The system architecture after we've scaled up our model server by adding model server replicas to the system.



In the new architecture, it's worth noting that multiple model serving requests from users are sent to the model server replicas at the same time. However, we haven't discussed any details on how they are being distributed and processed? For example, which of the requests are being processed by which of the model server replicas? In other words, we haven't defined a clear mapping relationship among the requests and the model server replicas yet.

To do that, we can add an additional layer, namely a *load balancer*, which handles the distribution of model serving requests among the replicas. For example, the load balancer takes multiple model serving requests from our users and then distributes the requests evenly to each of the model server replicas which then are responsible for processing individual requests, including model retrieval and inference on the new data in the request. Figure 4.7 below illustrates this process.

Figure 4.7 A diagram showing how a loader balancer is used to distribute the requests evenly across model server replicas.



The load balancer uses different algorithms to decide which request goes to which model server replica. Example algorithms for load balancing include round robin, least connection method, hashing, etc.

Note Round robin for load balancing

Round robin is a simple technique for making sure that the load balancer forwards each request to a different server replicas based on a rotating list.

Even though it's easy to implement a load balancer with the round robin algorithm, one fact that's easily neglected is that the load is already on a load balancer server and it might be dangerous when the load balancer server itself receives a lot of requests that require expensive processing and becomes overloaded past the point it can effectively do its job.

The replicated services pattern provides a great way to scalable our model serving system horizontally. It can also be generalized for any systems that serve a large amount of traffic. Whenever a single instance cannot handle the traffic, introducing this pattern would ensure that all traffic can be handled equivalently and efficiently. We'll apply this pattern in Section 9.3.2.

4.2.3 Discussion

Now that we have load-balanced model server replicas in place. We should be able to support the growing number of user requests and the entire model serving system achieves horizontal scaling. Not only could we handle model serving requests in a scalable way, but the overall model serving system also becomes *highly available*t *(HA)*. High availability (HA) is a characteristic of a system which aims to ensure an agreed level of operational performance, usually uptime, for a higher than normal period. It's often expressed as a percentage of uptime in a given year.

For example, some organizations may require the services to reach a highly available service level agreement (SLA), which means making sure the service is up and running for 99.9% of the time (three-nines availability). In other words, the service can only get 1.4 minutes of downtime per day (24 hours x 60 minutes x 0.1%). With the help from replicated model services, if any of the model server replicas crashes or gets preempted on a spot instance, the remaining model server replicas would still be available and ready for processing any incoming model serving requests from users, which provides a good user-experience and makes the system reliable.

In addition, one thing worth mentioning is that since our model server replicas will need to retrieve previously trained machine learning models from a remote model storage, the model server replicas need to be *ready* in addition to being *alive*. It's important to build and deploy *readiness probes* to inform the load balancer that the replicas are all successfully established connections to the remote model storage and ready to serve model serving requests from users. A readiness probe helps the system determine whether a particular replica is ready to serve. With readiness probes, users would not experience unexpected behaviors when the system is not ready due to internal system issues.

The replicated services pattern addresses our horizontal scalability issue that prevents our model serving system from supporting large number of model serving requests. However, in real-world model serving systems, not only the number of serving requests increase, but the size of each request can get extremely large if the data or the payload we are receiving is large. In that case, replicated services may not be able to handle the large requests. We will talk about that scenario and introduce a pattern that would alleviate the problem in the next section.

4.2.4 Exercises

1. Are replicated model servers stateless or stateful?
2. What happens when we don't have a load balancer as part of the model serving system?
3. Can we achieve three-nines SLA with only one model server instance?

4.3 Sharded services pattern: Processing large model serving requests with high resolution videos

The replicated services pattern efficiently resolves our horizontal scalability issue so that our model serving system can support the growing number of user requests. There also comes an additional benefit to achieve high availability with the help of model server replicas and load balancer.

Note Computational resources on model server replicas

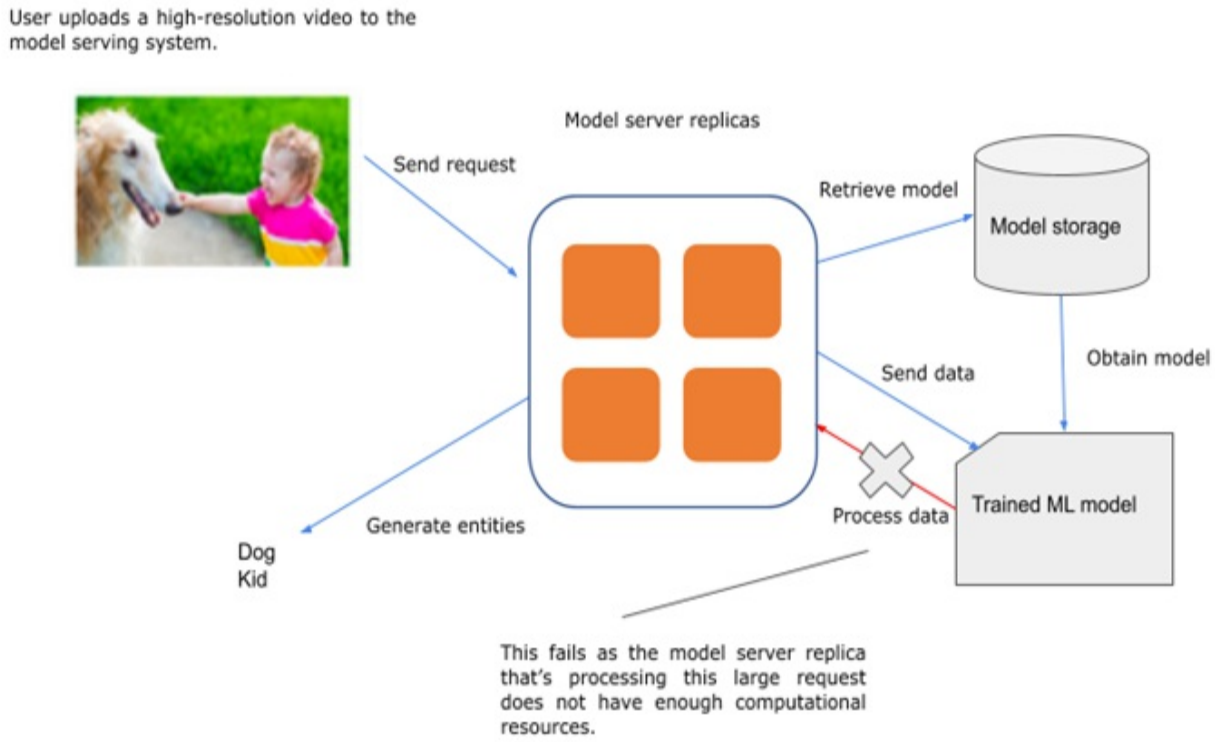
It's worth mentioning that each of the model server replicas has a limited and pre-allocated amount of computational resources. More importantly, the amount of computational resources for each replica must be identical for the load balancer to correctly and evenly distribute requests.

Next let's imagine that a user wants to upload a high-resolution YouTube video that needs to be tagged with an entity using the model server application. Even though the high-resolution video is too large, it may be uploaded successfully to the model server replica if it has sufficient disk storage. However, we could not process the request in any of the individual

model server replicas themselves since processing this single large request would require a larger memory allocated in the model server replica. This is often due to the complexity of the trained machine learning model as it may contain a lot of expensive matrix computations or mathematical operations as we've seen in the previous chapter.

For instance, a user uploads a high-resolution video to the model serving system through a large request. One of the model server replicas takes this request and successfully retrieves the previously trained machine learning model. Unfortunately the model then fails to process the large data in the request since the model server replica that's responsible for processing this request does not have sufficient memory. Eventually we may notify the user of this failure after the user has waited for a long time, which results in a bad user experience. A diagram for this situation is shown in Figure 4.8 below.

Figure 4.8 A diagram showing the situation when the model fails to process the large data in the request since the model server replica that's responsible for processing this request does not have sufficient memory.



4.3.1 Problem

The requests the system is serving are large since the videos users upload are of high-resolution. For cases where the previously trained machine learning model may contain expensive mathematical operations, these large requests of videos cannot be successfully processed and served by individual model server replicas within a limited amount of memory.

How do we design the model serving system so that it could successfully handle such large requests of high-resolution videos?

4.3.2 Solution

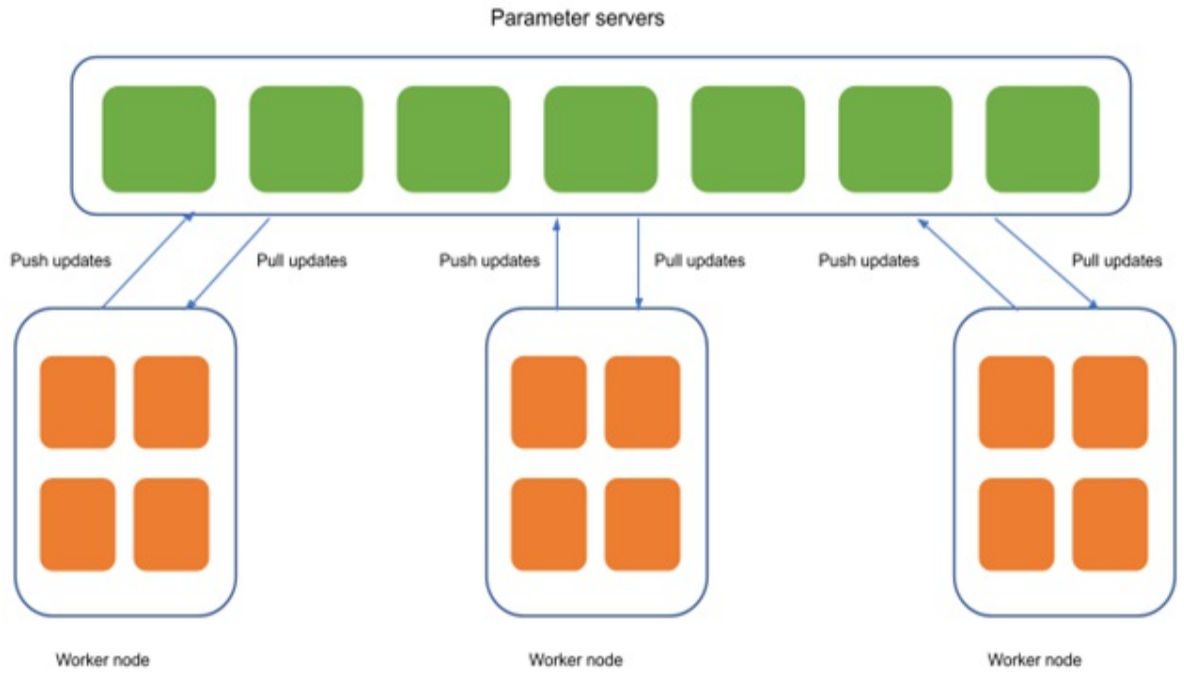
Given our requirement on the computational resources on each model server replicas, can we vertically scale and increase each replica's computational resources to make sure it can handle large requests like high-resolution videos? Note that since we are vertically scaling all the replicas by the same amount so this would not affect our load balancer's work.

Unfortunately we cannot simply vertically scale the model server replicas since we don't know how many such large requests there are. Imagine there are only a couple of users who have high-resolution videos at hand, e.g. they may be professional photographers who have professional cameras to allow them to capture high-resolution videos, and the remaining vast majority of the users only upload videos from their smartphones with much smaller resolution. As a result, most of the added computational resources on the model server replicas are idling and result in very low resource utilization on the replicas. We will discuss more from a resource utilization perspective in the next section but for now we know that this approach is not practical.

Remember we introduced the parameter server pattern in the previous chapter where we can partition a very large model? Below Figure 4.9 is the diagram we have discussed in the previous chapter showing distributed model training with multiple parameter servers where the large model has been partitioned and each partition is located on different parameter servers. Each worker node takes a subset of the dataset, performs calculations required in each of the neural network layers, and then sends the calculated gradients to update

one model partition that's stored in one of the parameter servers.

Figure 4.9 Distributed model training with multiple parameter servers where the large model has been sharded and each partition is located on different parameter servers.

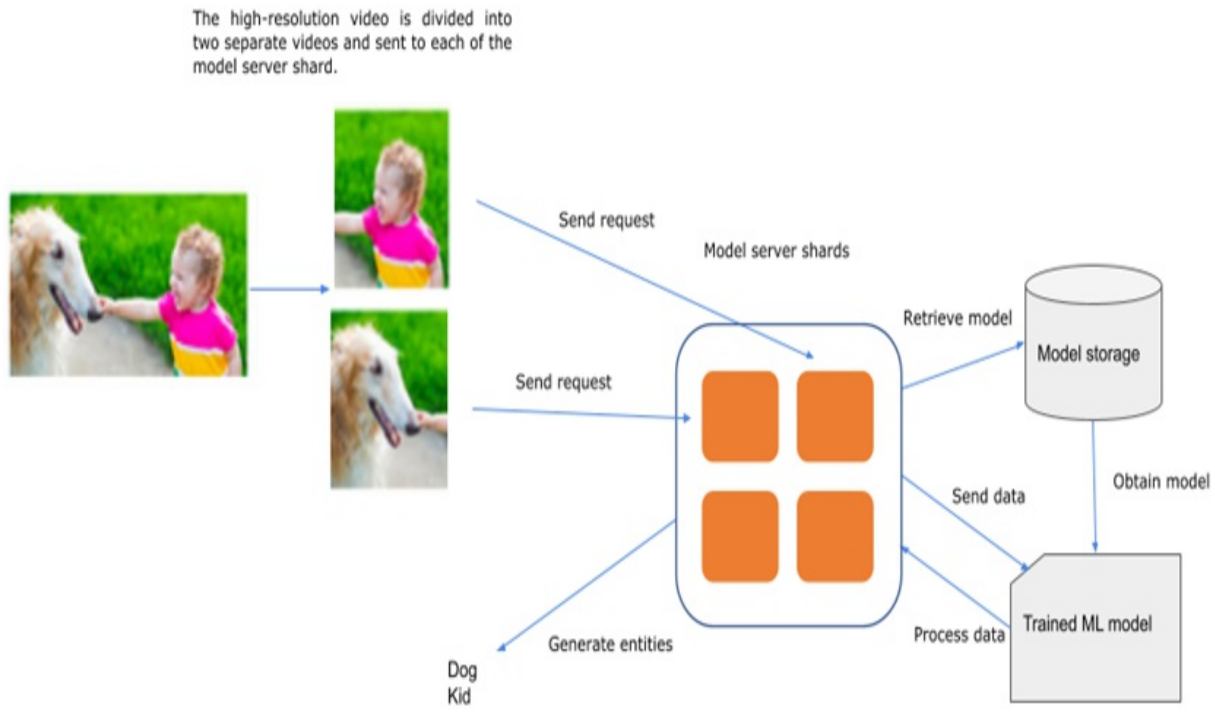


To deal with our problem with large model serving requests, we can borrow the same idea and apply it to our particular scenario.

We can first divide the original high-resolution video into multiple separate videos and then each video gets processed by multiple *model server shards* independently. The model server shards are partitions from a single model server instance and each model server shard is responsible for processing a subset of a large request.

The diagram below in Figure 4.10 is an example architecture of the *sharded services pattern*. In the diagram, a high-resolution video that contains a dog and a kid gets divided into two separate videos where each of the videos represents a subset of the original large request. One of the separated videos contains the part where the dog appears and the other video contains the part where the kid appears. These two separated videos become two separate requests and are processed by different model server shards independently.

Figure 4.10 An example architecture of the sharded services pattern where a high-resolution video gets divided into two separate videos where each of the videos represents a subset of the original large request and is processed by different model server shard independently.



After the model server shards receive the sub-requests where each contains part of the original large model serving request, each model server shard then retrieves the previously trained entity tagging machine learning model from the model storage, and then processes the videos in the request to tag possible entities that appear in the videos, similar to the previous model serving system we've designed.

Once all the sub-requests have been processed by each of the model server shards, we then merge the model inference result from two sub-requests, which consists of two entities, namely dog and kid, to present the result of the original large model serving request with the high-resolution video.

How do we distribute the two sub-requests to different model server shards? Similar to the algorithms we use to implement load balancer, we can use a *sharding function*, which is very similar to a hashing function, to determine which shard in the list of model server shards should be responsible for

processing each sub-requests.

Usually the sharding function is defined using a hashing function and the modulo (%) operator. For example, hash(request) % 10 would return 10 shards even when the outputs of the hash function are significantly larger than the number of shards in a sharded service.

Note Characteristics of hashing functions for sharding

The hashing function used to define the sharding function transforms an arbitrary object into an integer that represents a particular shard index. It has two important characteristics:

1. The output from hashing is always the same for a given input.

2. The distribution of outputs is always uniform within the output space.

These characteristics are important and can make sure that a particular request will always be processed by the same shard server and that the requests are evenly distributed among the shards.

The sharded services pattern solves the problem we encounter when building model serving systems at scale and provides a great way to handle large model serving requests. It's similar to the data sharding pattern we introduced in Chapter 2: instead of applying sharding to datasets, we apply sharding to model serving requests. When a distributed system has limited computational resources for a single machine, we can think about applying this pattern to offload the computational burden to multiple machines.

4.3.3 Discussion

The sharded services pattern we've introduced helps handle large requests and efficiently distributes the workload of processing large model serving requests to multiple model server shards. It's generally useful when considering any sort of service where there is more data than what can fit on a single machine.

However, unlike the replicated services pattern we discussed in the previous

section that’s useful when building stateless services, the sharded services pattern is generally used for building stateful services. In our case here, we need to maintain the state or the results from serving the sub-requests from the original large request using sharded services and then merge the results into the final response so it includes all entities from the original high-resolution video.

In some cases, this approach may not work well because it depends on how we divide the original large request into smaller requests. For example, if the original video has been divided into more than two sub-requests, some of the sub-requests may not be meaningful since they don’t contain any complete entities that are recognizable by the machine learning model we’ve trained. For situations like that, we need additional handling and cleaning of the merged result to remove meaningless entities that are not useful for our application.

Both the replicated services pattern and sharded services pattern solve the problem when building model serving system at scale to handle a great number of large model serving requests. However, in order to incorporate them into the model serving system, we need to know the required computational resources at hand, which may not be available if the traffic is rather dynamic. We will introduce another pattern that focus on model serving systems that handles dynamic traffic in the next section.

4.3.4 Exercises

1. Would vertical scaling be helpful when handling large requests?
2. Are the model server shards stateful or stateless?

4.4 Event-driven processing pattern: Responding model serving requests based on events

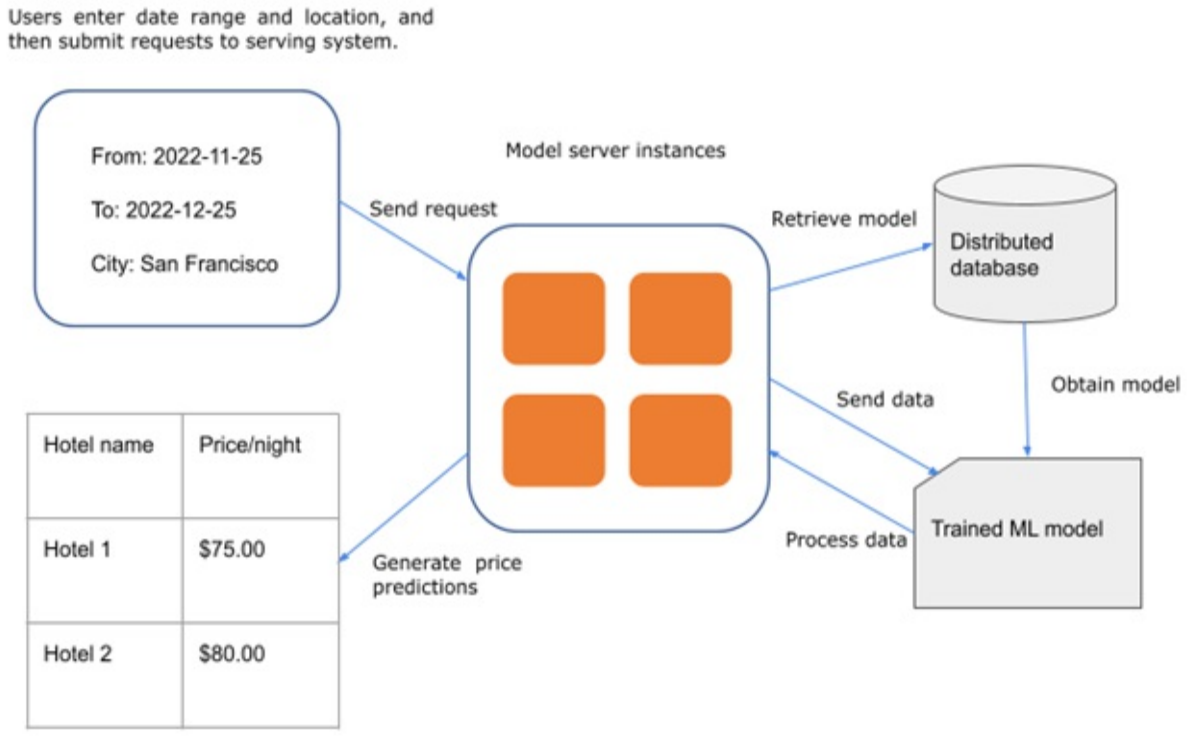
The replicated services pattern we’ve introduced in Section 4.2 helps handle a large number of model serving requests and the sharded services pattern in Section 4.3 solves the challenge when processing very large requests that may not fit in a single model server instance. While these patterns greatly

addressed the challenges when building model serving systems at scale, they are more suitable when the system knows how much computational resources, model server replicas, or model server shards to allocate before the system starts to handle the requests from users. However, for cases where we do not know how much model serving traffic the system will be receiving, it's hard to allocate and use resources efficiently.

Now imagine that we work for a company that provides holiday and event planning services to subscribed customers. We'd like to provide a new service that will use a trained machine learning model to predict hotel prices per night for the hotels located in resort areas, given a range of dates and a specific location where our customers would like to spend their holidays.

In order to provide that service, we can design a machine learning model serving system. This model serving system provides a user interface where users can enter the range of dates and locations that they are interested in staying for holidays. Once the requests are sent to the model server, the previously trained machine learning model will be retrieved from the distributed database and then process the data in the requests (dates and locations), and eventually the model server would return the predicted hotel prices for each location within the given date range. The complete process is shown in Figure 4.11 below.

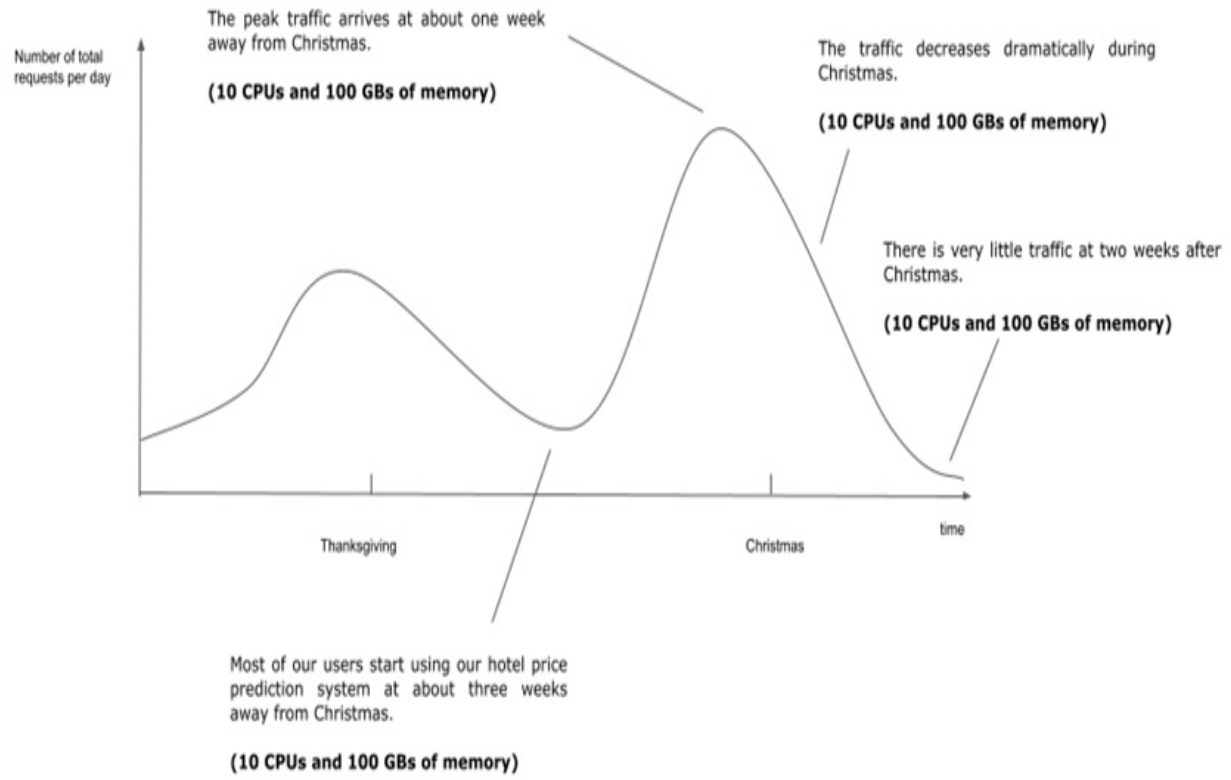
Figure 4.11 Diagram of the model serving system to predict hotel prices.



After we've tested this model serving system for one year on selected customers and we have collected sufficient data to plot the model serving traffic over time. As it turns out people prefer to book their holidays at the last moment and traffic increases abruptly shortly before holidays and then decreases again after the holiday periods. The problem with this traffic pattern is that it introduces a very low resource utilization rate.

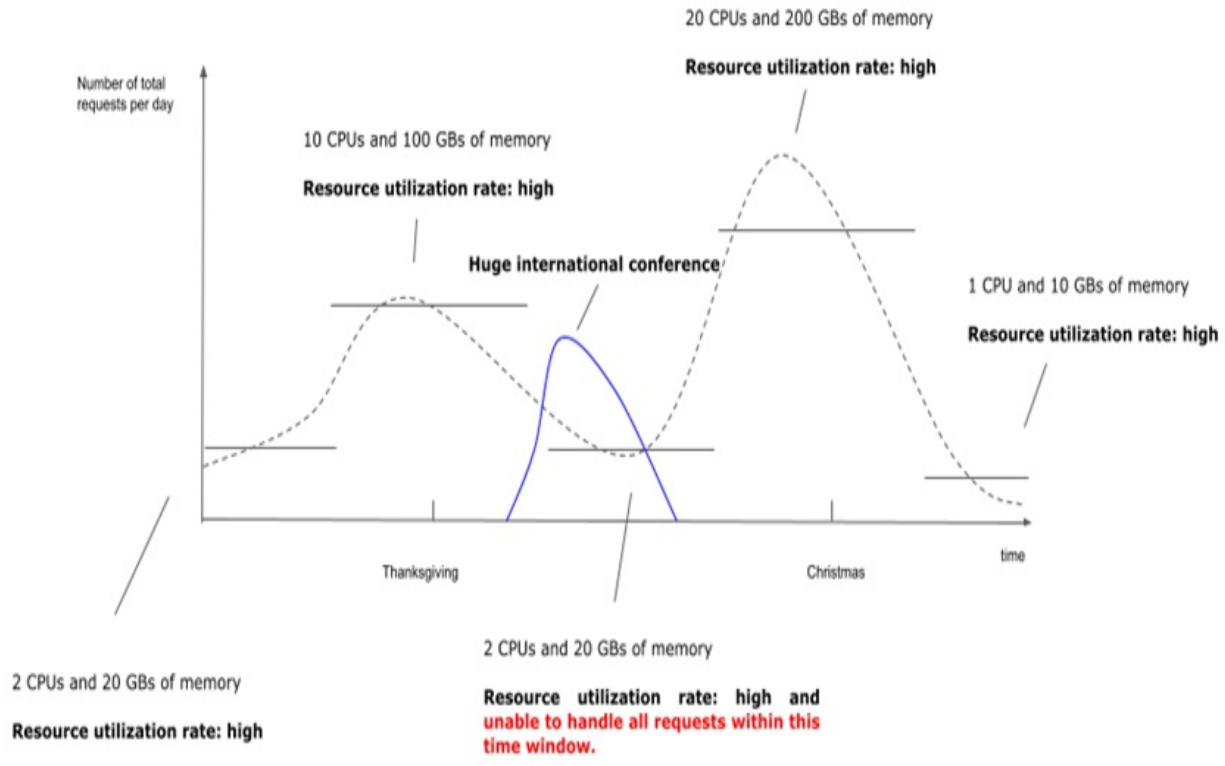
In our current architecture of model serving system, the underlying computational resources allocated to the model remain unchanged at all times and this strategy seems to be far from optimal: during periods of low traffic most of our resources are sitting there idling and wasted whereas during periods of high traffic our system struggles to respond in a timely fashion and more resources are required to operate normally. In other words, the system has to deal with either high or low traffic with the same amount of computational resources, e.g. 10 CPUs and 100 GBs of memory, as shown in Figure 4.12 below.

Figure 4.12 The traffic changes of the model serving system over time with an equal amount of computational resources allocated all the time.



Since we more or less know when those holiday periods are, why don't we plan accordingly? Unfortunately, it turns out that there can possibly be events that make it hard to predict surges of traffic. For example, there are days when a huge international conference is planned to be located near one of the resorts as shown in Figure 4.13 below. This unexpected event that happens before Christmas has suddenly added traffic at that particular time window (solid line). We then would miss a window that should be taken into account when allocating computational resources. Specifically in our scenario, 2 CPUs and 20 GBs of memory, though optimized for our use case, would no longer be sufficient to handle all resources within this time window. Otherwise, the user experience during that window would become really bad. Imagine all the conference attendants sitting in front of their laptops and waiting for a long time to get a hotel booked.

Figure 4.13 The traffic of our model serving system over time with an optimal amount of computational resources allocated for different time windows. In addition, there's an unexpected event happening before Christmas that suddenly added traffic at that particular time window (solid line).



In other words, this naive solution is still not very practical and effective since it's non-trivial to figure out the time windows to allocate different amounts of resources and how much additional resources we want to allocate for each of those time windows. Can we come up with any better approach?

In our scenario, we are dealing with the dynamic number of model-serving requests that varies from time to time and is highly correlated to the appearances of holidays. What if we can always make sure that we have enough resources and forget about the goal to increase resource utilization rate for now? If the computational resources are guaranteed to be more than sufficient all the time, we can make sure that the model serving system can handle heavy traffic during holiday seasons.

4.4.1 Problem

The naive approach, which is to estimate and to allocate computational resources accordingly before any possible time windows that the system might experience high volume of traffic, is infeasible as it's not easy to figure out the exact dates of the high traffic time windows and the exact amount of

computational resources to be allocated during each of the time windows.

In addition, simply increasing the computational resources to an amount that is sufficient at all times would not be practical either as the resource utilization rate that we were concerned about earlier still remains low. For example, if there are nearly no user requests being made during a particular time period, the computational resources we have allocated are unfortunately mostly idling and become wasted.

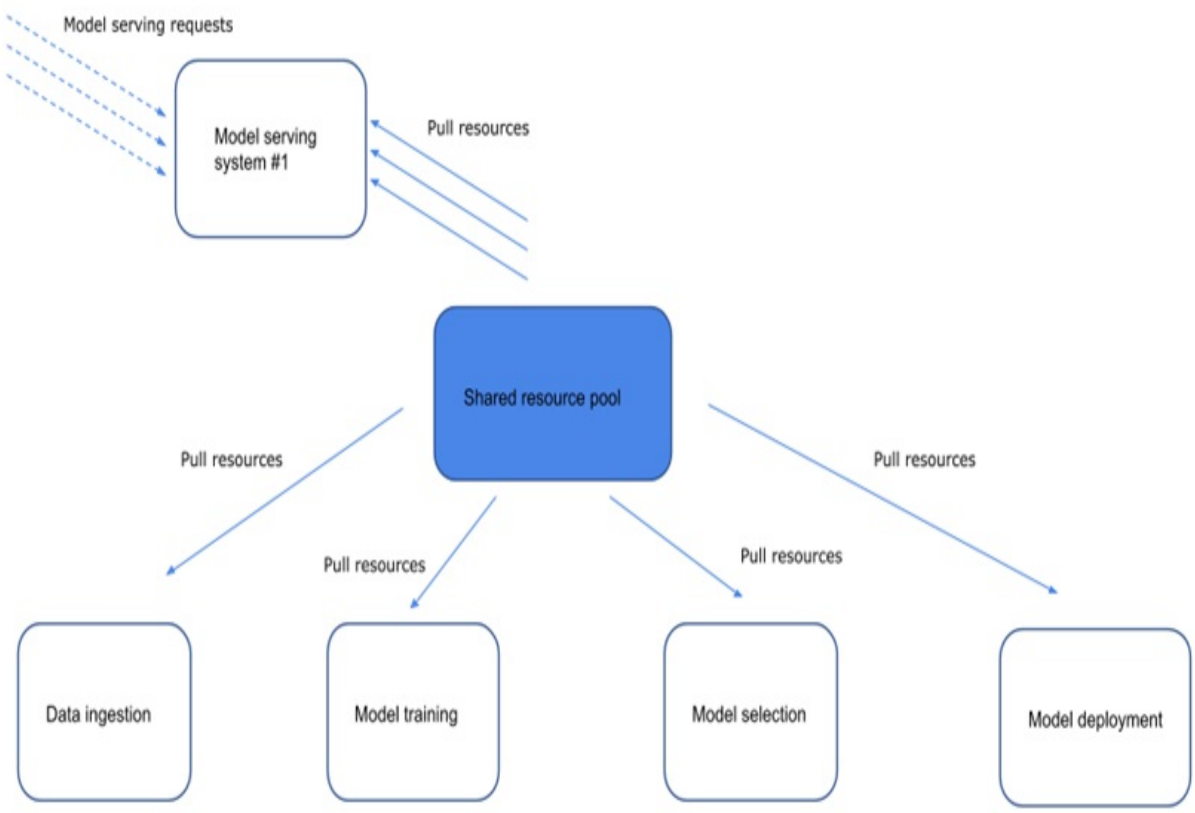
Is there any better approach to allocate and use computational resources more wisely?

4.4.2 Solution

The solution to this is that we can maintain a pool of computational resources (e.g. CPUs, memory, disk, etc.) allocated not just for this particular model serving system but also for model serving of other applications or other components of the distributed machine learning pipeline.

Figure 4.14 below is an example architecture diagram where a shared resource pool is being used by different systems, e.g. data ingestion, model training, model selection, and model deployment, and model serving, at the same time. This shared resource pool can make sure we always have enough resources to handle peak traffic for the model serving system by pre-allocating resources required during historical peak traffic and auto-scaling when the limit has reached. Therefore, we only use the resources when needed and only use the specific amount of resources required for each particular model serving request. That the arrows in solid lines indicate resources and the arrows in dashed lines indicate requests.

Figure 4.14 Architecture diagram where a shared resource pool is being used by different systems, e.g. data ingestion, model training, model selection, and model deployment, and two different model serving systems, at the same time.



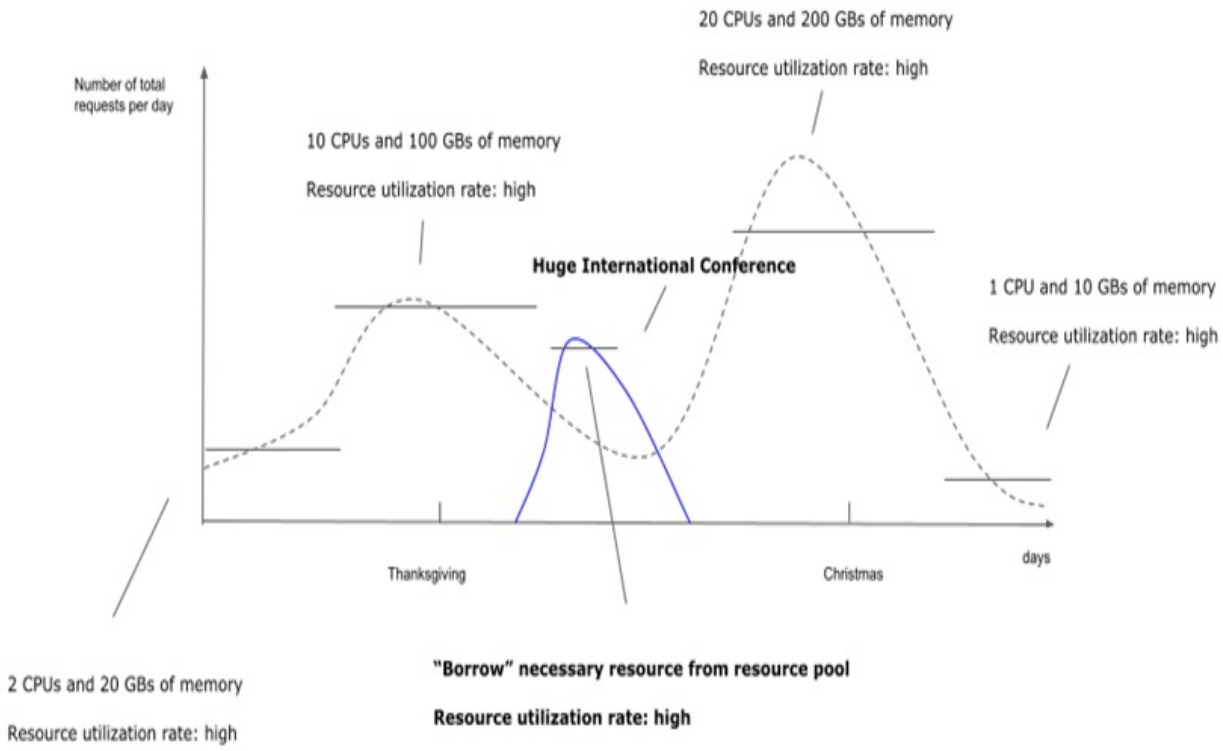
Note that for the purpose of our discussions, we only focus on the model serving system in the diagram and details for other systems are neglected here. In addition, here we assume that the model training component only utilizes similar types of resources such as CPUs. If the model training component requires GPUs or the mix of CPUs/GPUs then it may be better to use a separate resource pool depending on specific use cases.

When the users of our hotel price prediction application enter the range of dates and locations that they are interested in staying for holidays in the UI, the model serving requests are then sent to the model serving system. Upon receiving each request, the system notifies the shared resource pool that certain amounts of computational resources are being used by the system.

For example, Figure 4.15 below shows the traffic of our model serving system over time with an unexpected bump. The unexpected bump is due to a new huge international conference that happens before Christmas. This event suddenly added traffic but successfully handled via borrowing a necessary

amount of resources from the shared resource pool. With the help of the shared resource pool, the resource utilization rate remains high during this unexpected event. The shared resource pool would monitor the current amount of available resources and auto-scale when needed.

Figure 4.15 The traffic of our model serving system over time. The unexpected bump happening before Christmas that suddenly added traffic is handled successfully via borrowing a necessary amount of resources from the shared resource pool. The resource utilization rate remains high during this unexpected event.



This approach, in which the system listens to the user requests and only responds and utilizes the computational resources when the user request is being made, is called *event-driven processing*.

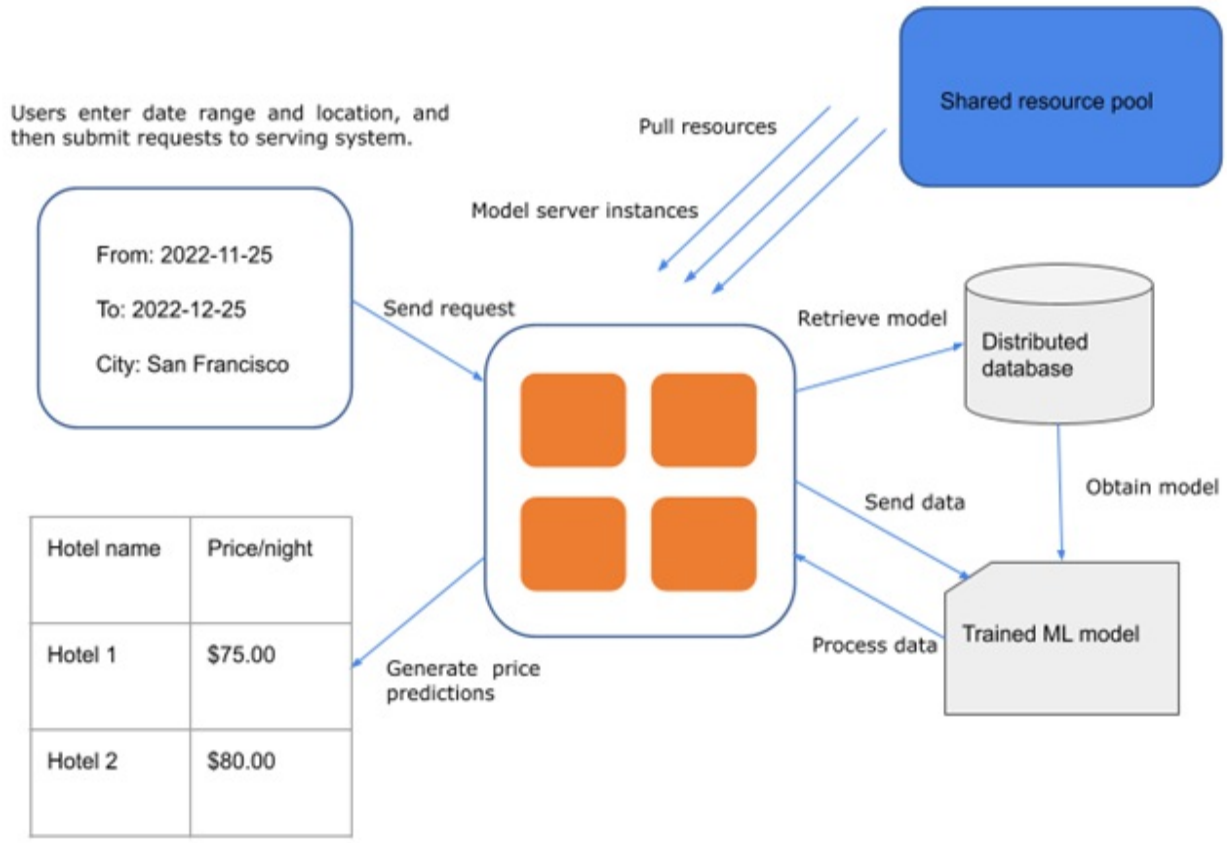
Note Event-driven processing v.s. long-running serving systems

Event-driven processing is different from the model serving systems that we've seen in previous sections, e.g. systems leveraging the replicated services in Section 4.2 and sharded services patterns in Section 4.3, where the servers that handle user requests are always up and running. Those long-

running serving systems work well for many applications that are under heavy load, keep a large amount of data in memory, or require some sort of background processing. However, for applications that only need to temporarily handle very few requests during non-peak periods or need to respond to specific events, such as our hotel price prediction system, the event-driven processing pattern is more suitable. This event-driven processing pattern has flourished in recent years as cloud providers have developed *function-as-a-service (FaaS)* products.

In our scenario, each model serving request made from our hotel price prediction system represents an *event* and our serving system listens this type of events, utilizes necessary resources from the shared resource pool, and then retrieves and loads the trained machine learning model from distributed database to estimate the hotel prices for the specified time-location query. Figure 4.16 below is a diagram of this event-driven model serving system.

Figure 4.16 Diagram of the event-driven model serving system to predict hotel prices.



With this event-driven processing pattern for our serving system, we can make sure that our system is only using the resources necessary to process every request without having to worry about the resource utilization and idling. As a result, the system would have sufficient resources to deal with peak traffic and return the predicted prices without experiencing noticeable delays or lags when users are using the system.

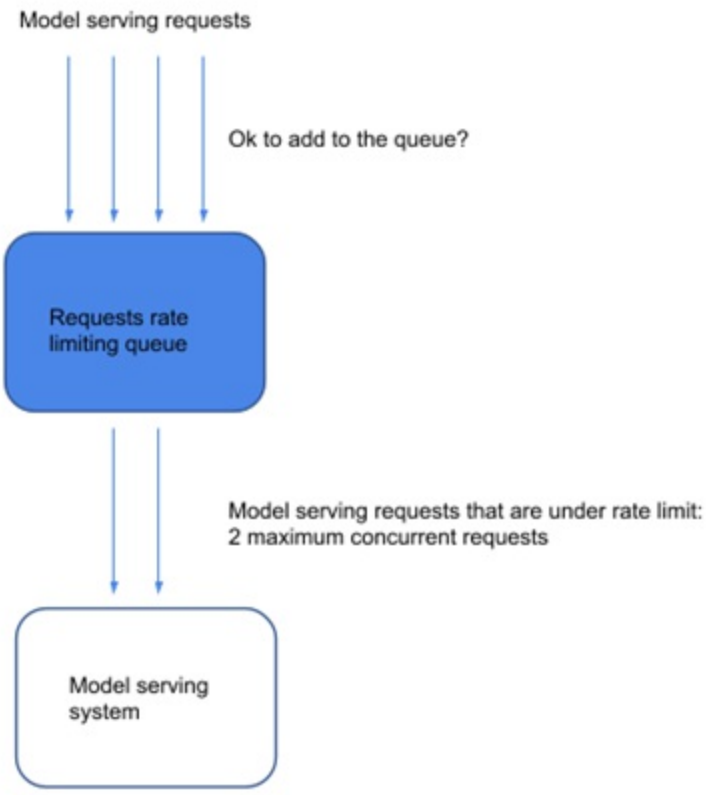
Note that even though we now have a shared pool of sufficient computational resources where we can “borrow” computational resources from to handle events of user requests on demand, we should also build a mechanism in our model serving system to defend *denial-of-service attacks*, which are interruptions in an authorized user's access to a computer network, typically one caused with malicious intent and often seen in model serving systems. These attacks could cause unexpected use of computational resources from the shared resource pool, which may eventually lead to resource scarcity of other services that rely on the shared resource pool.

Denial-of-service attacks may happen in various cases. For example, they may come from our users who accidentally sent a huge amount of model serving requests in a very short period of time. Developers may have misconfigured a client that uses our model serving APIs so it sends requests constantly or accidentally kicks off an unexpected load/stress test in a production environment.

To deal with those situations which often happen in real-world applications, it makes sense to introduce a defense mechanism for denial-of-service attacks. One approach to avoid this is via *rate limiting*, which adds the model serving requests to a queue and limits the rate the system is processing the requests in the queue.

Figure 4.17 below is a flowchart where there are 4 total model serving requests being sent to the model serving system but only 2 of them are under the current rate limit which only allows a maximum of 2 concurrent model serving requests. In this case, the rate limiting queue for model serving requests first checks whether the requests received are under the current rate limit. Once the system has finished processing those two requests, it will then proceed to the remaining 2 requests in the queue.

Figure 4.17 A flowchart of 4 total model serving requests being sent to the model serving system but only 2 of them are under the current rate limit which only allows a maximum of 2 concurrent model serving requests. Once the system has finished processing those two requests, it will then proceed to the remaining 2 requests in the queue.

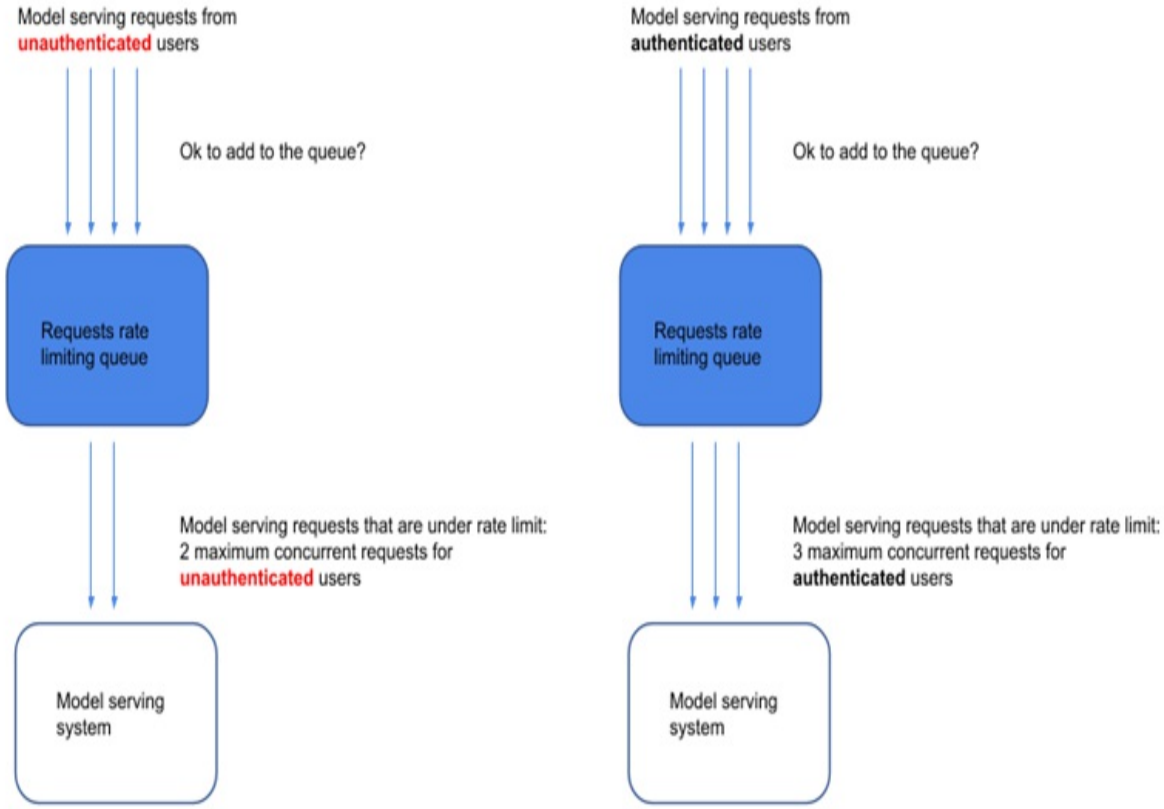


If we are deploying and exposing an API for a model serving service to our users, it's also generally a best practice to have a relatively small rate limit (e.g. only one request is allowed within one hour) for users with anonymous access and then asks users to log in to obtain a higher rate limit. This would allow the model serving system to better control and monitor the users' behavior and traffic so that we can take any necessary actions to address any potential issues or denial-of-service attacks. For example, requiring a login provides auditing to find out which users/events are responsible for the unexpected large number of model serving requests.

Figure 4.18 demonstrates the above strategy. In the diagram, the flowchart on the left hand side is the same as the previous figure where 4 total model serving requests from unauthenticated users are sent to the model serving system but only 2 of them can be served by the system due to the current rate

limit which only allows a maximum of 2 concurrent model serving requests for unauthenticated users. On the contrary, the model serving requests in the flowchart on the right hand side all come from authenticated users and 3 of them can be processed by the model serving system since the limit of maximum concurrent requests for authenticated users.

Figure 4.18 Comparison of behaviors from different rate limits applied to authenticated and unauthenticated users.



There are different rate limits depending on whether the user is authenticated and thus effectively controls the traffic of the model serving system and avoids the malicious denial-of-service attacks, which could cause unexpected use of computational resources from the shared resource pool and eventually lead to resource scarcity of other services that rely on the shared resource pool.

4.4.3 Discussion

Even though we've seen how the event-driven processing pattern benefits our particular serving system, it's worth noting that we should not attempt to use this pattern as a universal solution, as with many tools and patterns that help develop a distributed system to meet different real-world requirements.

For machine learning applications that have consistent traffic, for example, model predictions are calculated regularly based on a schedule, it's unnecessary to adopt this event-driven processing approach as the system already knows when to process the requests and there will be too much overhead trying to monitor this regular traffic. In addition, applications that can tolerate less accurate predictions can work well without being driven by events, e.g. they can also re-calculate and provide good enough predictions to a particular granularity level such as per day or per week.

On the contrary, event-driven processing is more suitable for applications that have different traffic patterns that are complicated for the system to prepare necessary computational resources beforehand. With event-driven processing, the model serving system only requests a necessary amount of computational resources on demand. The applications can also provide more accurate and real-time predictions since they obtain the predictions right after the users send requests instead of relying on pre-calculated prediction results based on a schedule.

From developers' perspective, one benefit of the event-driven processing pattern is that it's very intuitive to developers. For example, it greatly simplifies the process to deploy code to running services since there is no end artifact to create or push beyond the source code itself. The event-driven processing pattern makes it simple to deploy code from our laptops or web browser to running code in the cloud.

In our scenario, we only need to deploy the trained machine learning model that could be used as a *function* to be triggered based on user requests. Once deployed, this model serving function is then managed and scaled automatically without having to allocate resources manually by developers. In other words, as more traffic is loaded onto the service, more instances of the model serving function are created to handle the increase in traffic using the shared resource pool. If somehow the model serving function fails due to machine failures, it will be restarted automatically on other machines in the

shared resource pool.

On the other hand, given the nature of the event-driven processing pattern, each function that's used to process the model serving requests needs to be *stateless* and independent from other model serving requests. Each function instance cannot have local memory which requires all states to be stored in a storage service. For example, when our machine learning models depend heavily on the results from previous predictions, e.g. a time series model, then the event-driven processing pattern may not be suitable in this case.

4.4.4 Exercises

1. If we allocate the same amount of computational resources over the lifetime of the model serving system for hotel price prediction, what would the resource utilization rate look like over time?
2. Are the replicated services or sharded services long-running systems?
3. Is event-driven processing stateless or stateful?

4.5 References

1. YouTube-8M dataset: http://research.google.com/youtube8m/
2. High availability: https://en.wikipedia.org/wiki/High_availability

4.6 Summary

- Model serving is the process of loading a previously trained machine learning model, generating predictions or making inferences on new input data.
- Replicated services help handle the growing number of model serving requests and achieve horizontal scaling with the help of replicated services.
- The sharded services pattern allows the system to handle large requests and efficiently distributes the workload of processing large model serving requests to multiple model server shards.
- With the event-driven processing pattern, we can make sure that our system is only using the resources necessary to process every request

without having to worry about the resource utilization and idling.

Answers to exercises:

Section 4.2

1. Stateless.
2. The model server replicas would not know which requests from users to process and there will be potential conflicts or duplicate work when multiple model server replicas try to process the same requests.
3. Yes, only if the single server has no more than 1.4 minutes of downtime per day.

Section 4.3

1. Yes it helps but would decrease the overall resource utilization.
2. Stateful.

Section 4.4

1. It varies over time depending on the traffic.
2. Yes. Servers are required to keep them running there to accept user requests and computational resources need to be allocated and occupied all the time.
3. Stateless.

5 Workflow patterns

This chapter covers

- Using workflows to connect different machine learning system components (data ingestion, distributed model training, and model serving).
- Composing complex but maintainable structures within machine learning workflows with the fan-in and fan-out patterns.
- Accelerating machine learning workloads with concurrent steps that leverages the synchronous and asynchronous patterns.
- Improving performance and avoiding duplicate workloads in the workflows with the help of the step memoization pattern.

In the previous chapter, we've discussed some of the challenges involved in the model serving component and introduced a couple of practical patterns that can be incorporated into this component. Model serving is a critical step after we have successfully trained a machine learning model. It is the final artifact produced by the entire machine learning workflow and the results from model serving are presented to the users directly. Previously we've explored some of the challenges involved in distributed model serving systems and introduced a few established patterns adopted heavily in industries. For example, we've seen challenges when handling the growing number of model serving requests and how we can overcome the challenges to achieve horizontal scaling with the help of replicated services. We've also discussed the sharded services pattern to help the system process large model serving requests and learned how to assess model serving systems and determine whether event-driven design would be beneficial with real-world scenarios.

Workflow is an essential component in machine learning systems as it connects all other components in a machine learning system. A machine learning workflow can be as easy as chaining just data ingestion, model training, and model serving. On the other hand, it can be very complex to handle different real-world scenarios to allow additional steps and

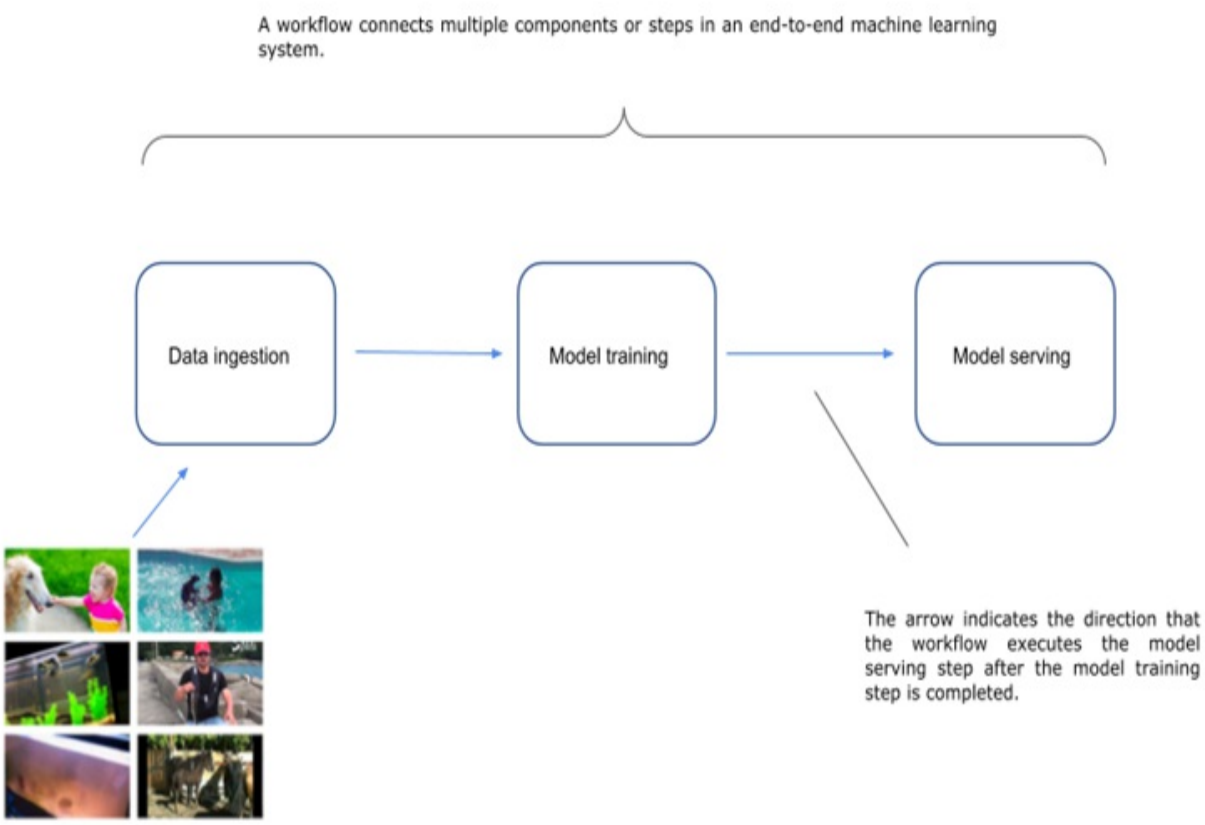
performance optimizations to be part of the entire workflow. It's essential to know what trade-offs we may see when making different design decisions in order to meet different business and performance requirements. In this chapter, we'll explore some of the challenges involved when building machine learning workflows in practice. Each of these established patterns can be reused to build simple to complex machine learning workflows that are efficient and scalable. For example, we'll see how to build a system to execute complex machine learning workflows to train multiple machine learning models and pick the most performant ones to provide good entity tagging results in the model serving system, with the fan-in and fan-out patterns. We'll also incorporate synchronous and asynchronous patterns to make machine learning workflows more efficient and avoid delays due to the long-running model training steps that block other consecutive steps.

5.1 What is workflow?

Workflow is the process of connecting multiple components or steps in an end-to-end machine learning system. A workflow consists of arbitrary combinations of the components that are commonly seen in real-world machine learning applications, such as data ingestion, distributed model training, and model serving that we have introduced in the previous chapters.

Figure 5.1 shows a simple machine learning workflow. This workflow connects multiple components or steps in an end-to-end machine learning system, including data ingestion step that consumes the Youtube-8M videos dataset, model training step that trains an entity tagging model, and model serving step that tags entities in unseen videos. The arrows indicate the directions. For example, the arrow on the right hand side denotes the order of the step execution, e.g. the workflow executes the model serving step after the model training step is completed.

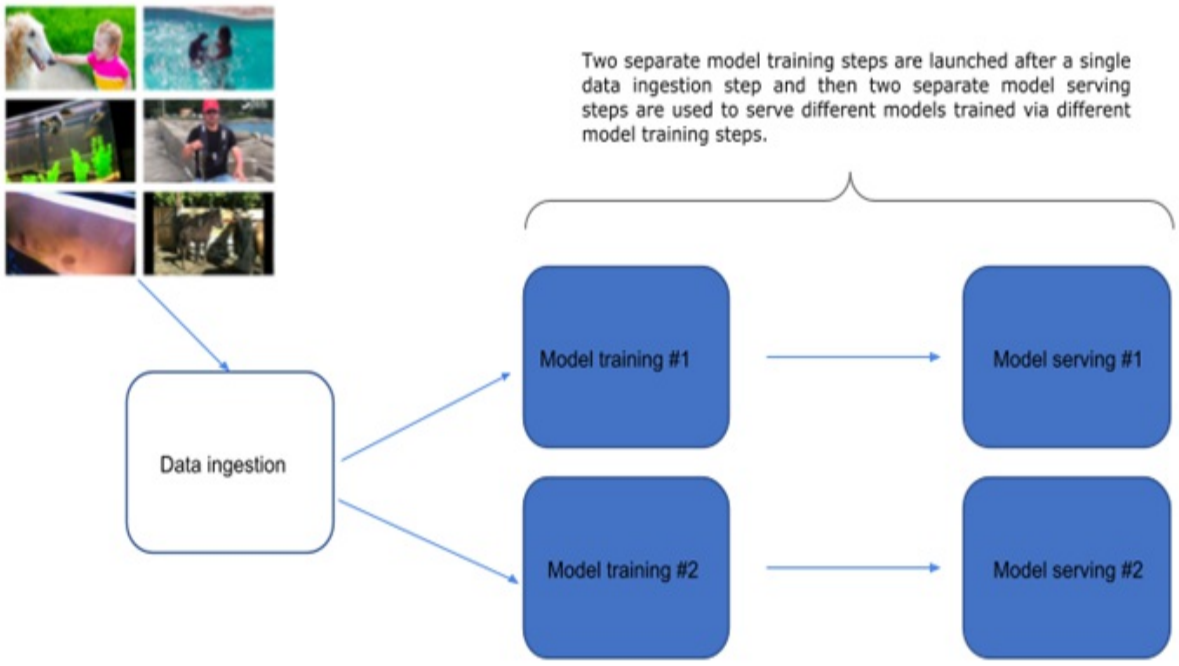
Figure 5.1 A diagram showing what a simple machine learning workflow looks like, including data ingestion, model training, and model serving.



Note that a machine learning workflow is also often referred to as a machine learning pipeline. We will be using these two terms interchangeably, e.g. using different technologies that use two terms differently, but please note that there should be no difference between the two terms in this book.

Since a machine learning workflow may consist of any combinations of the components, we often see machine learning workflows in different forms in different situations. Unlike the straightforward workflow shown in Figure 5.1, Figure 5.2 below illustrates a more complicated workflow where two separate model training steps are launched after a single data ingestion step and then two separate model serving steps are used to serve different models trained via different model training steps.

Figure 5.2 A more complicated workflow where two separate model training steps are launched after a single data ingestion step and then two separate model serving steps are used to serve different models trained via different model training steps.

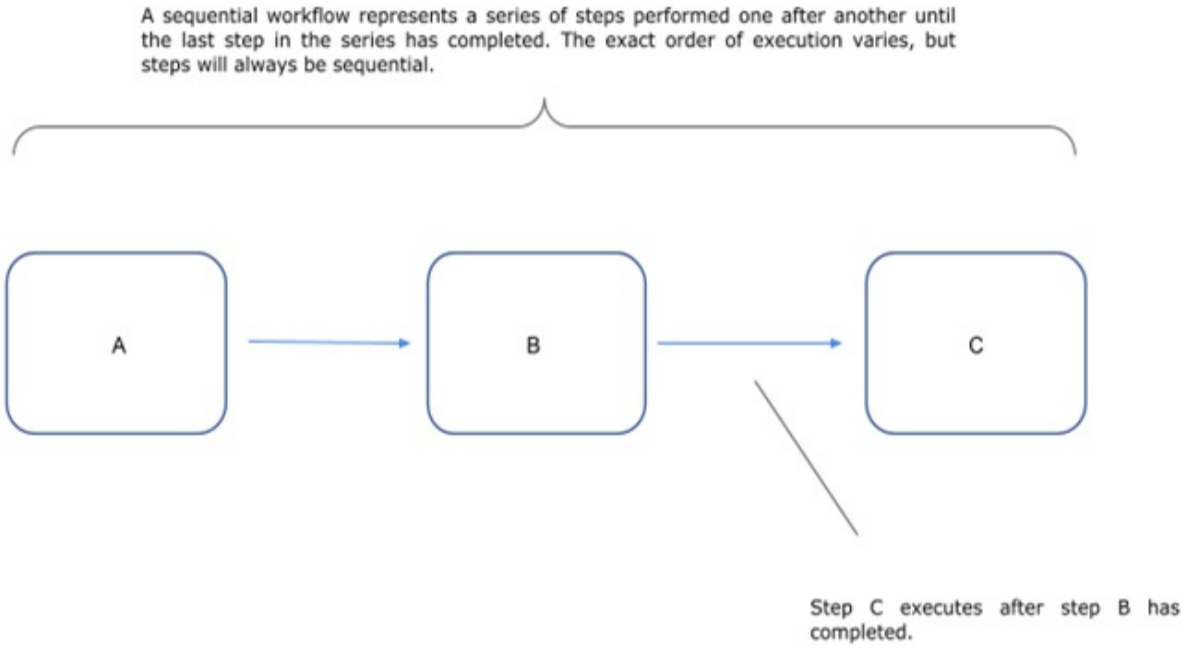


The previous ones are just some common examples. In practice, however, the complexity of different machine learning workflows varies, which increases the difficulty of building and maintaining scalable machine learning systems.

We will discuss some of the more complex machine learning workflows in the following sections in this chapter but for now it's good to introduce and distinguish the differences between the following two concepts: *sequential workflow* and *directed acyclic graph*.

A *sequential workflow* represents a series of steps performed one after another until the last step in the series has completed. The exact order of execution varies, but steps will always be sequential. Figure 5.3 below is an example sequential workflow with three steps executed sequentially in the following order: A, B, and C.

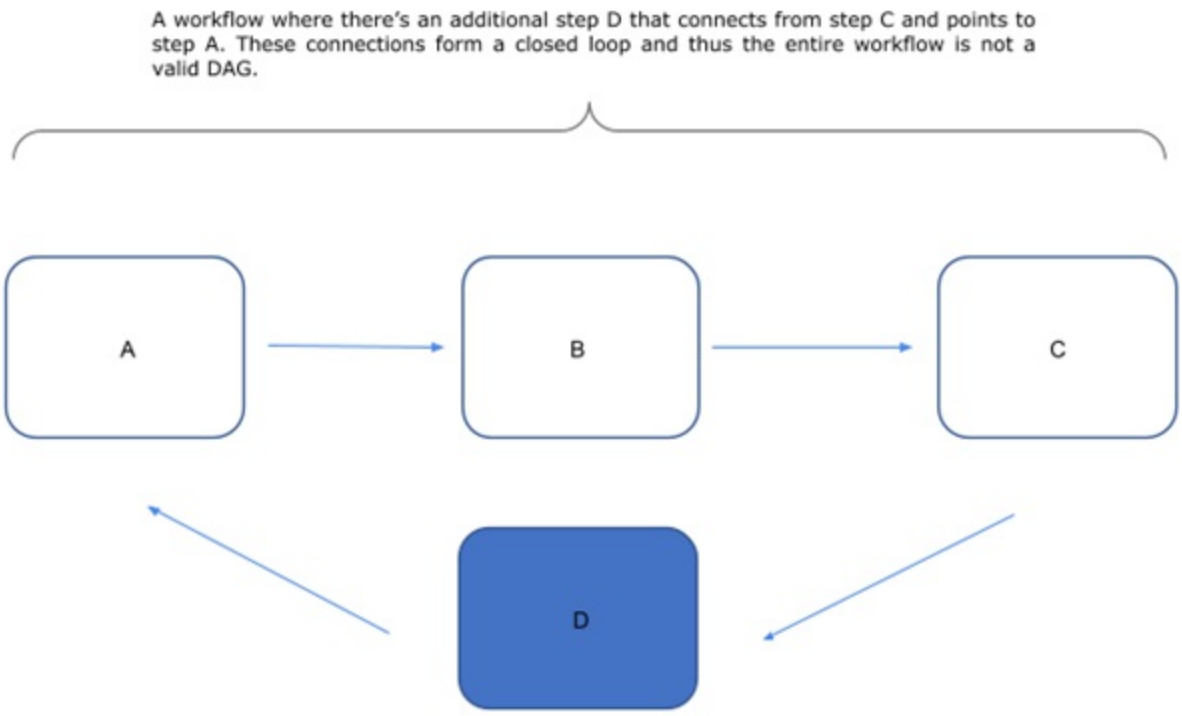
Figure 5.3 An example sequential workflow with three steps which executes sequentially in the following order: A, B, and C.



A workflow can be seen as a *directed acyclic graph (DAG)* if the workflow only consists of steps that are directed from one step to another but in the meantime the directions will never form a closed loop.

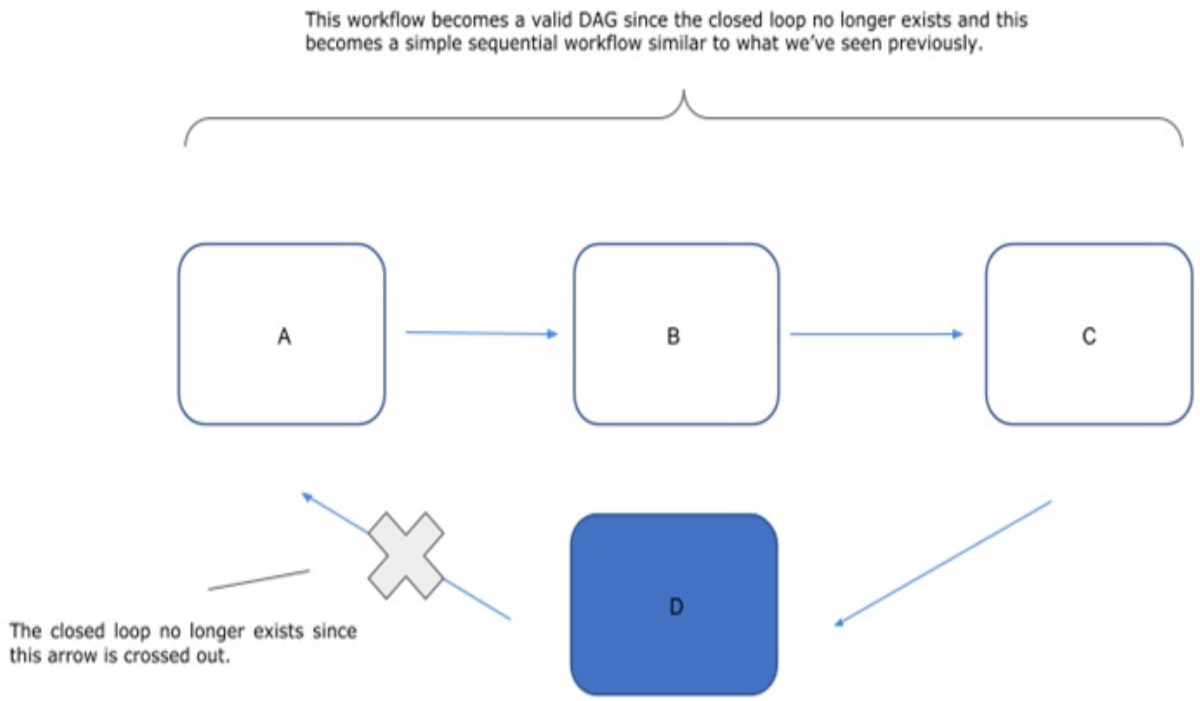
For example, the workflow in Figure 5.3 above is a valid DAG since the three steps are directed from step A to step B and then from step B to step C, where there is no closed loop. Another example workflow shown in Figure 5.4, however, is not a valid DAG since there's an additional step D that connects from step C and points to step A, which forms a closed loop.

Figure 5.4 An example workflow where there's an additional step D that connects from step C and points to step A. These connections form a closed loop and thus the entire workflow is not a valid DAG.



If the last step D does not point back to step A, as shown in Figure 5.5 below where the arrow is crossed out, then this workflow becomes a valid DAG since the closed loop no longer exists and this becomes a simple sequential workflow similar to what we've seen previously.

Figure 5.5 An example workflow where the last step D does not point back to step A. This workflow becomes a valid DAG since the closed loop no longer exists and this becomes a simple sequential workflow similar to what we've seen previously.



In real-world machine learning applications, the workflows can get really complex in order to meet the requirements of different use cases, such as batch retraining of the models, hyperparameter tuning experiments, etc. We will go through some of the more complex ones and abstract the structural patterns in the workflows that can be reused to compose workflows for various scenarios.

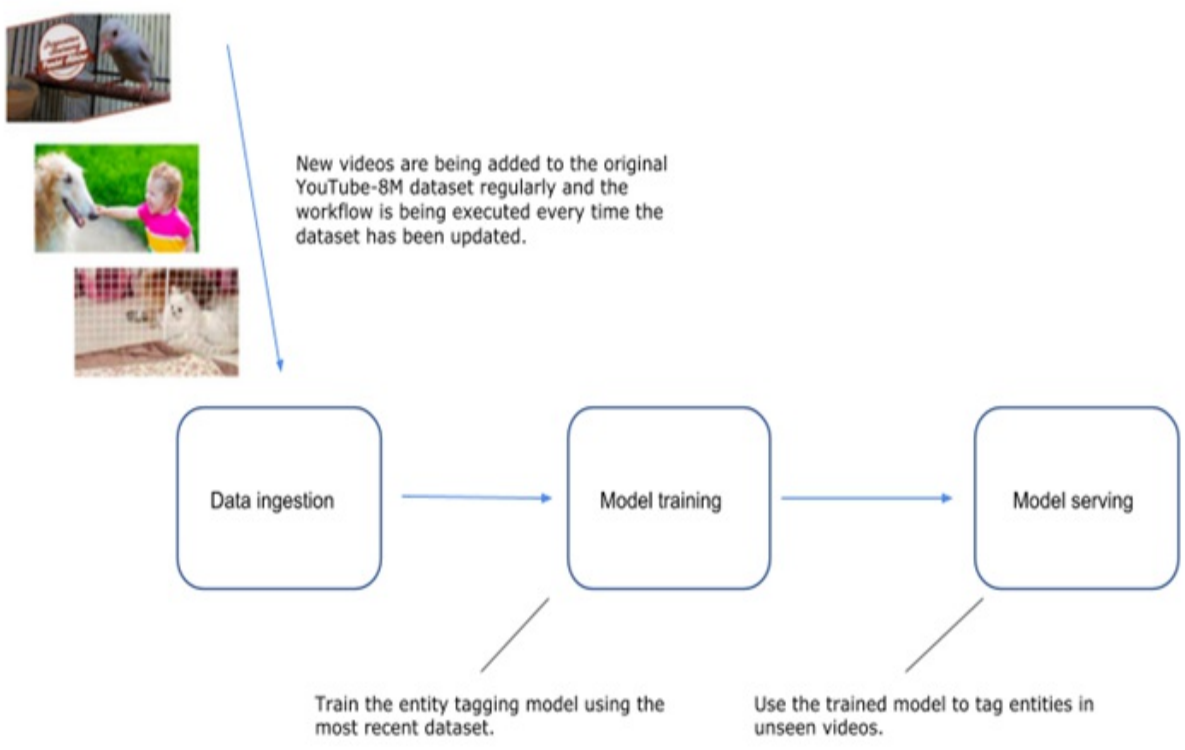
5.2 Fan-in and fan-out patterns: Composing complex machine learning workflows

Recall that in Chapter 3 we've built a machine learning model to tag the main themes of new videos that the model hasn't seen before using the YouTube-8M dataset, which consists of millions of YouTube video IDs, with high-quality machine-generated annotations from a diverse vocabulary of 3,800+ visual entities such as food, car, music, etc. In Chapter 4, we've also discussed patterns that are helpful to build scalable model serving systems where users can upload new videos and then the system would load the previously trained machine learning model to tag entities/themes that appear in the uploaded videos.

In real-world applications, we often want to chain these steps together and package them in a way that can be easily reused and distributed.

For example, what if the original YouTube-8M dataset has been updated and we'd like to train a new model from scratch using the same model architecture? In this case, it's actually pretty easy to containerize each of these components and chain them together in a machine learning workflow that can be reused by re-executing the end-to-end workflow again when the data gets updated. As shown in Figure 5.6, new videos are being added to the original YouTube-8M dataset regularly and the workflow is being executed every time the dataset has been updated. The next model training step trains the entity tagging model using the most recent dataset and then the last model serving step uses the trained model to tag entities in unseen videos.

Figure 5.6 New videos are being added to the original YouTube-8M dataset regularly and the workflow is being executed every time the dataset has been updated.



Now let's take a look at a more complex real-world scenario. Assuming that we know the implementation details for model training of any machine

learning model architecture, we would like to build a machine learning system to train different models and then use the top two models to both generate predictions so that the entire system is less likely to miss any entities in the videos since the two models may capture information from different perspectives in the videos.

5.2.1 Problem

We would like to build a machine learning workflow that would train different models after the system has ingested data from the data source, select the top two models, and then use these two models to provide model serving for users that generates predictions with the knowledge from both of the models.

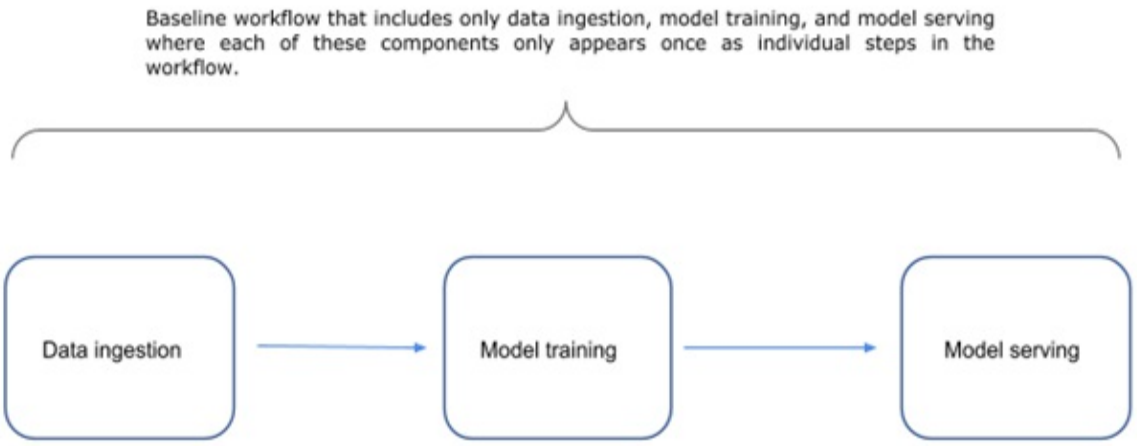
Though it is straightforward to build a workflow that includes the end-to-end normal process of a machine learning system with only data ingestion, model training, and model serving where each component only appears once as individual steps in the workflow. However, in our particular scenario, the workflow we are building is much more complex as we need to include multiple model training steps as well as multiple model serving steps.

How do we formalize and generalize the structure of this complex workflow so that it can be easily packaged, reused, and distributed?

5.2.2 Solution

Let’s start with the most basic machine learning workflow that includes only data ingestion, model training, and model serving where each of these components only appears once as individual steps in the workflow. We will build our system based on this workflow that serves as our baseline, as shown in Figure 5.7 below.

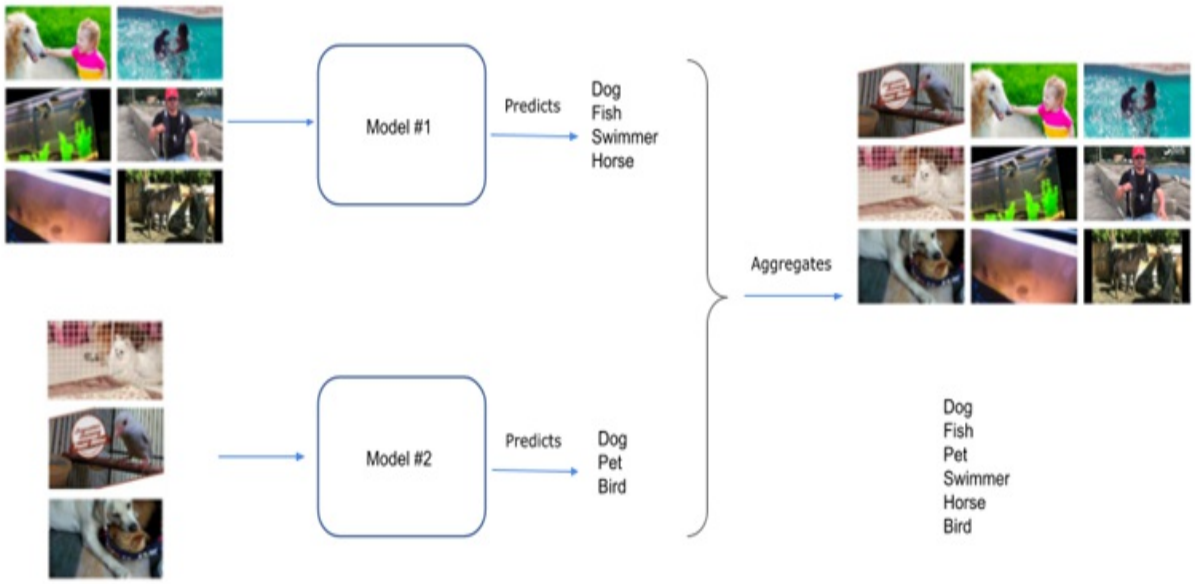
Figure 5.7 A baseline workflow that includes only data ingestion, model training, and model serving where each of these components only appears once as individual steps in the workflow.



Our goal is to have a way to represent the machine learning workflow that would build and select top two models that will both be used for model serving in order to give better inference results, leveraging the power of the two best performing models.

Let’s take a moment to understand why this approach might be used in practice. For example, in Figure 5.8 below, there are two models where the first model has knowledge of four entities and the second model has knowledge of three entities so each of them is able to tag the entities they know from the videos. If we use both of the models to try tag entities at the same time and then aggregate their results, then the aggregated result is obviously more knowledgeable and is able to cover more entities. In other words, two models can be more effective and produce more comprehensive entity tagging results.

Figure 5.8 Diagram of models where the first model has knowledge of four entities and the second model has knowledge of three entities so each of them is able to tag the entities they know from the videos. When both of the models are used to tag entities at the same time and then aggregate their results, then the aggregated result is obviously more knowledgeable and is able to cover more entities.

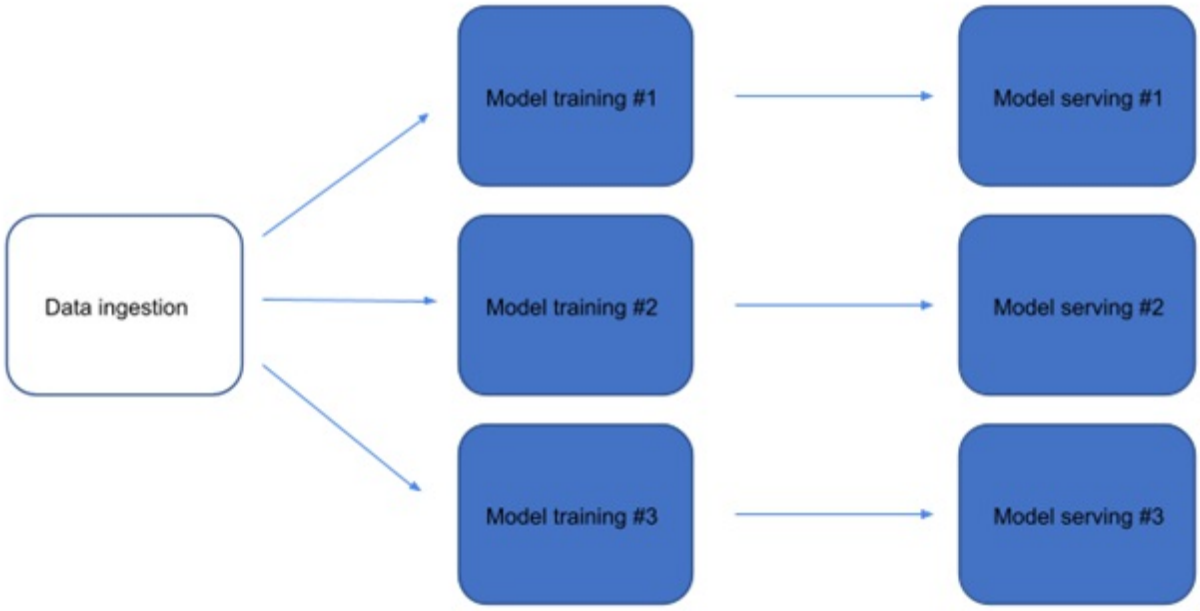


Now that we understand the motivation behind building this complex workflow, let's overview what the entire end-to-end workflow process looks like. We would like to build a machine learning workflow that:

1. Ingests data from the same data source;
2. Trains multiple different models, either different sets of hyperparameters of the same model architecture or various model architectures;
3. Picks the top two performing models to be used for model serving for each of the trained models;
4. Aggregates their results to be presented to the users for each of the two model serving systems.

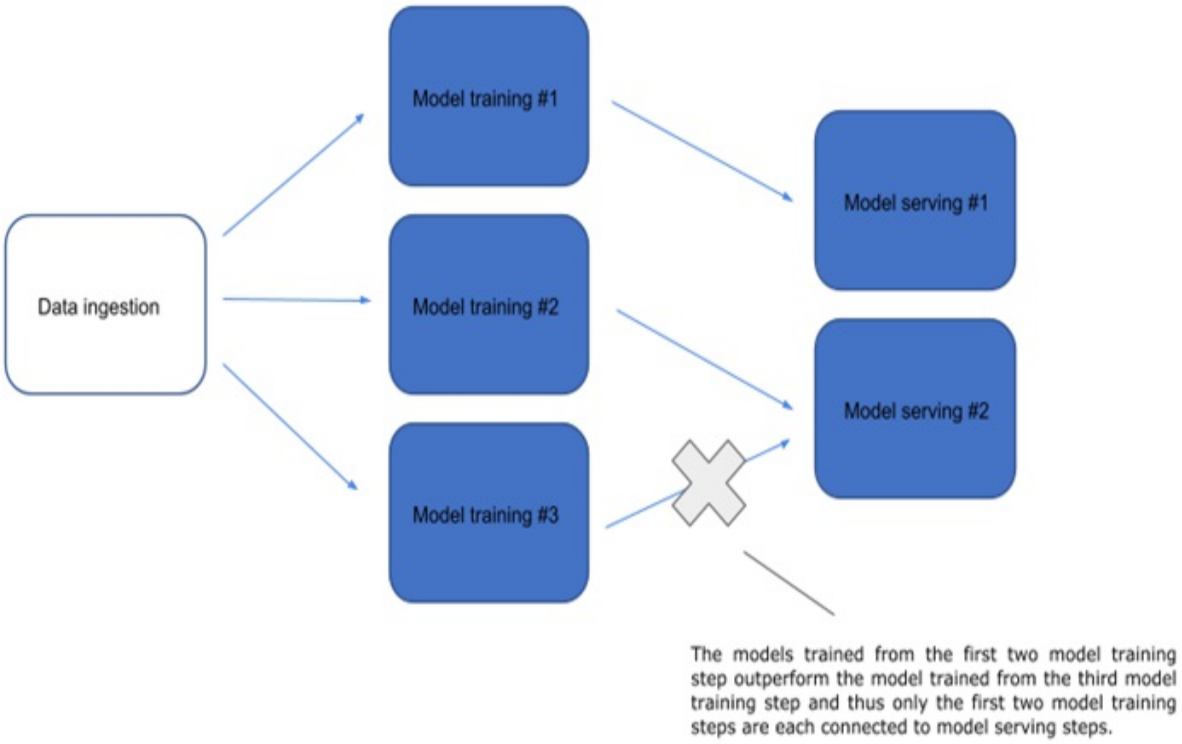
Let's first add some placeholders to the baseline workflow for multiple model training steps after data ingestion and then add multiple model serving steps once the multiple model training steps finish. A diagram of the enhanced baseline workflow is shown in Figure 5.9 below.

Figure 5.9 Diagram of the enhanced baseline workflow where there are multiple model training steps after data ingestion and multiple model serving steps after the multiple model training steps.



The key difference from what we've dealt with before in the baseline is that there are multiple model training components as well as multiple model serving components. It's also worth noting that the relationships among the steps are not one-to-one direct relationships. For example, each model training step may be connected to a single model serving step or not connected to any steps at all. Figure 5.10 below illustrates that the models trained from the first two model training steps outperform the model trained from the third model training step and thus only the first two model training steps are each connected to model serving steps.

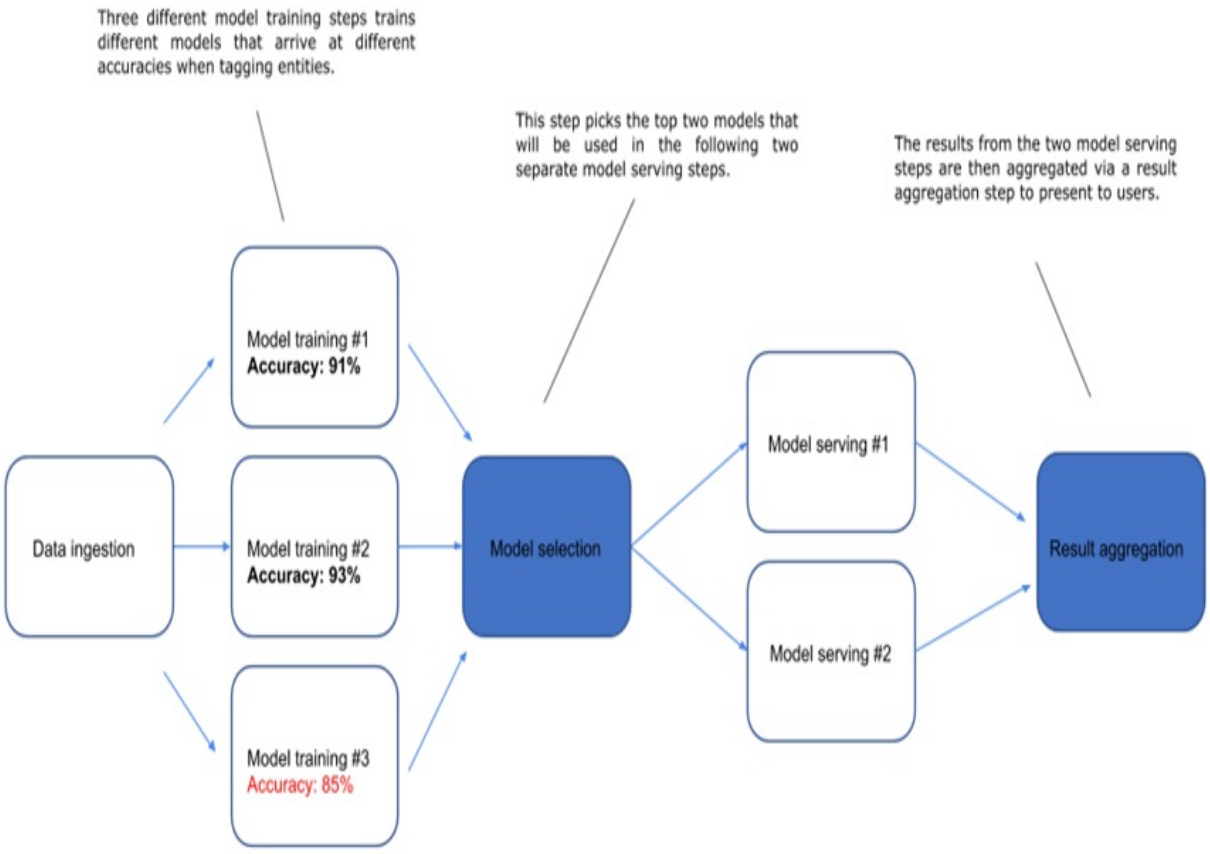
Figure 5.10 The models trained from the first two model training steps outperform the model trained from the third model training step and thus only the first two model training steps are each connected to model serving steps.



We can compose this workflow as follows. Upon successful data ingestion, there are multiple model training steps connecting to the data ingestion step so that they can use the shared data that's ingested and cleaned from the original data source. Next, a single step is connected to the model training steps to select the top two performing models and then produces two model serving steps that use those selected models to serve model serving requests from users. A final step at the end of this machine learning workflow is connected to the two model serving steps to aggregate the model inference results that will be presented to the users. A diagram for this workflow is shown below.

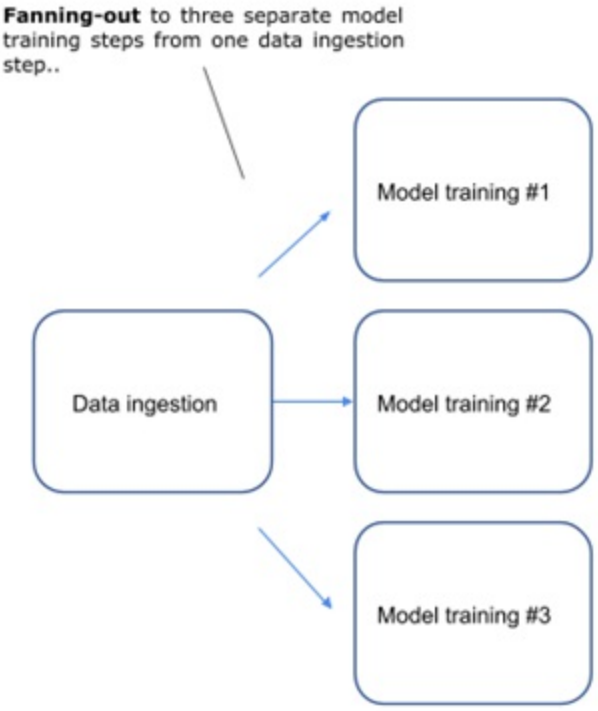
A diagram of the complete workflow is shown in Figure 5.11 below. This workflow trains different models via three different model training steps that arrive at different accuracies when tagging entities, and then a model selection step picks the top two models with at least 90% accuracy trained from the first two model training steps that will be used in the following two separate model serving steps. The results from the two model serving steps are then aggregated via a result aggregation step to present to users.

Figure 5.11 A machine learning workflow that trains different models that arrive at different accuracies when tagging entities and then selects the top two models with at least 90% accuracy to be used for model serving. The results from the two model serving steps are then aggregated to present to users.



There are actually two patterns we can abstract from this complex workflow. The first one we observe is the *fan-out* pattern. Fan-out is a term to describe the process of starting multiple separate steps to handle input from the workflow. In our workflow, the fan-out pattern appears when there are multiple separate model training steps connecting to the data ingestion step, as shown in Figure 5.12 below.

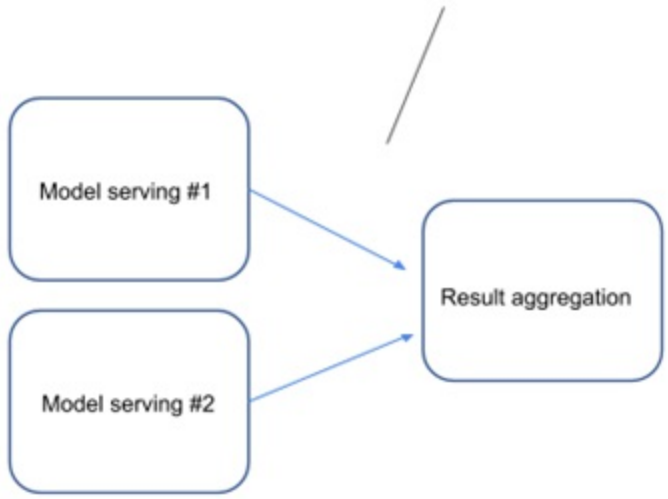
Figure 5.12 Diagram of the fan-out pattern that appears when there are multiple separate model training steps connecting to the data ingestion step.



In addition, there's also the *fan-in* pattern in our workflow where we have one single aggregation step that combines the results from the two model serving steps, as shown in Figure 5.13 below. Fan-in is a term to describe the process of combining results from multiple steps into one step.

Figure 5.13 Diagram of the fan-in pattern where we have one single aggregation step that combines the results from the two model serving steps.

Fanning-in from two model serving steps to one result aggregation step.



Formalizing these patterns would help us build and organize more complex workflows by composing different patterns to workflows based on real-world requirements.

We have successfully built the system in the form of a complex workflow that trains different models and then uses the top two models to both generate predictions so that the entire system is less likely to miss any entities in the videos.

These patterns are powerful when constructing complex workflows to meet real-world requirements. We can construct a workflow from a single data processing step to multiple model training steps to train different models with the same dataset. We can also start more than one model serving steps from each of these model training steps if the predictions from different models are useful in real-world applications. We'll apply this pattern in Section 9.4.1.

5.2.3 Discussion

By leveraging the fan-in and fan-out patterns in the system, the system is now able to execute complex workflows that train multiple machine learning models and pick the most performant ones to provide good entity tagging

results in the model serving system.

These patterns are great abstractions that can be incorporated in very complex workflows to meet the increasing demand of complex distributed machine learning workflows in the real-world. But what kind of workflows are suitable to utilize the fan-in and fan-out patterns?

In general, if both of the following applies, we can consider incorporating these patterns:

1. The multiple steps that we are fanning-in or fanning-out are independent of each other;
2. It takes a long time for these steps to run sequentially.

The multiple steps need to be order-independent because we have no guarantee in what order concurrent copies of those steps will run, nor in what order they will return. For example, if the workflow also contains a step that trains an ensemble of other models (also known as ensemble learning) in order to provide a better aggregated model, then this ensemble model depends on the completion of other model training steps. Consequently, we cannot use the fan-in pattern because the ensemble model training step will need to wait for other model training to complete before it can start running, which would require some extra waiting and delay the entire workflow.

Note Ensemble models

An ensemble model uses multiple machine learning models to obtain better predictive performance than could be obtained from any of the constituent models alone. It often consists of a number of alternative models that can learn the relationships in the dataset from different perspectives.

Ensemble models tend to yield better results when there is a significant diversity among the constituent models. Therefore many ensemble approaches try to increase the diversity among the models they combine.

The fan-in and fan-out patterns can compose very complex workflows that meet most of the requirements of machine learning systems. However, in order to achieve good performance on those complex workflows, we need to

spend time on what parts of the workflows to run first and what parts of the workflows can be executed in parallel. As a result of the optimization, data science teams would spend less time waiting for workflows to complete and in the meantime, infrastructure costs can be reduced as well. We will introduce some patterns that would help us organize the steps in the workflow from a computational perspective in the next section.

5.2.4 Exercises

1. If the steps are not independent of each other, can we use the fan-in or fan-out patterns?
2. What's the main issue when trying to build ensemble models with the fan-in pattern?

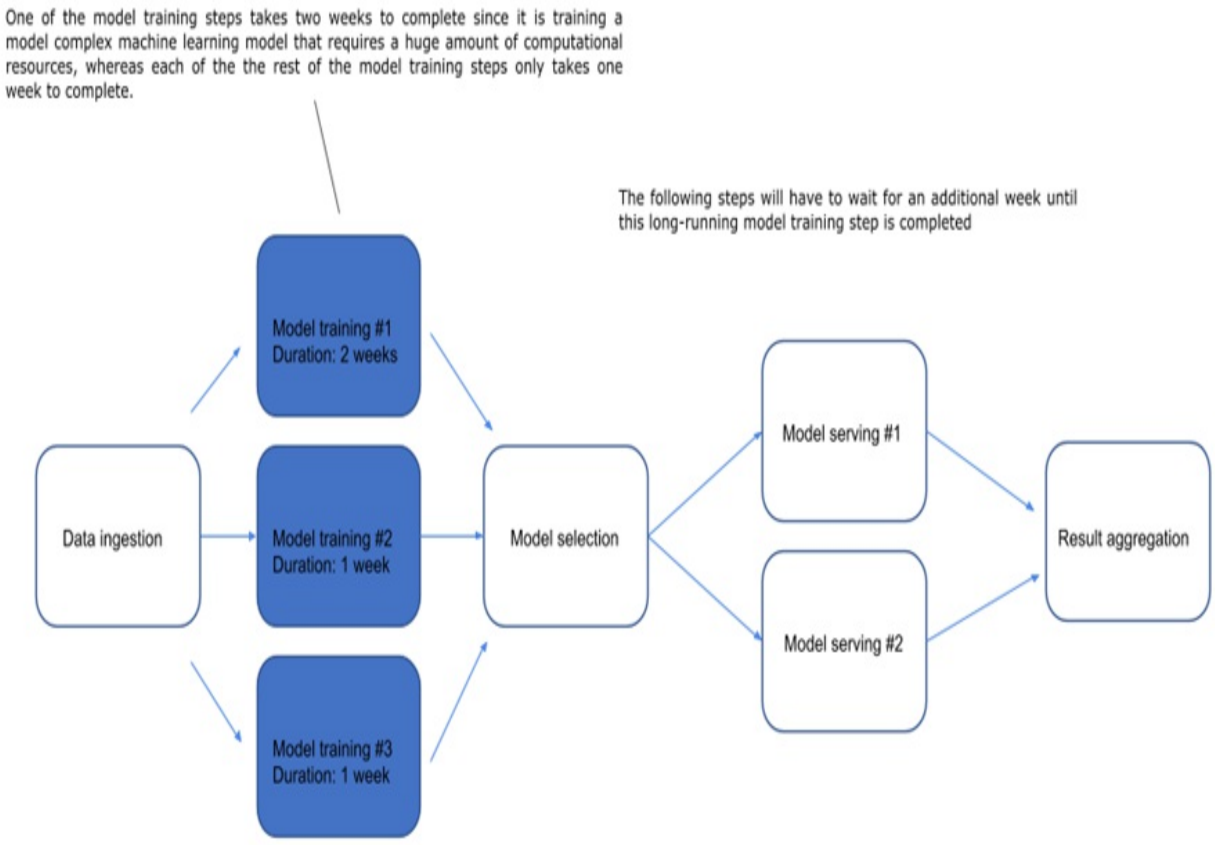
5.3 Synchronous and asynchronous patterns: Accelerating workflows with concurrency

We have leveraged the fan-in and fan-out patterns and successfully built the machine learning system with a complex workflow that trains different models and then uses the top two models to both generate predictions that will be aggregated and presented to the users.

It's worth mentioning that each of the model training steps in the system would take a long time to reach completion. In the meantime, their durations may vary across different model architectures or model parameters.

Imagine an extreme case where one of the model training steps takes two weeks to complete since it is training a model complex machine learning model that requires a huge amount of computational resources, whereas each of the the rest of the model training steps only takes one week to complete. Many of the steps, such as the model selection step and model serving steps, in the machine learning workflow we built earlier that leverages the fan-in and fan-out patterns will have to wait for an additional week until this long-running model training step is completed. A diagram that illustrates the duration differences among the three model training steps is shown in Figure 5.14 below.

Figure 5.14 A workflow that illustrates the duration differences among the three model training steps.



In this case, since the model selection step and the steps following it would require all of the model training steps to finish, the model training step that takes two weeks to complete will slow down the entire workflow by an entire week. We could have used that one additional week to finish executing all of the model training steps again that take one week to complete instead of wasting time waiting for one step!

5.3.1 Problem

We would like to build a machine learning workflow that would train different models and then select the top two models to be used in model serving for users that generates predictions with the knowledge from both of the models. Due to the variance of completion time for each of the model training steps in the existing machine learning workflow, the start of each of

the following steps, such as the model selection step and the model serving steps, depend on the completion of its previous steps.

However, when at least one of the model training steps takes much longer to complete than the rest of the steps, the model selection step that follows can only start to execute until this long model training step has completed. As a result, the entire workflow is being delayed by this particular long-running step.

Is there a way to accelerate this workflow so it will not be affected by the durations of individual steps?

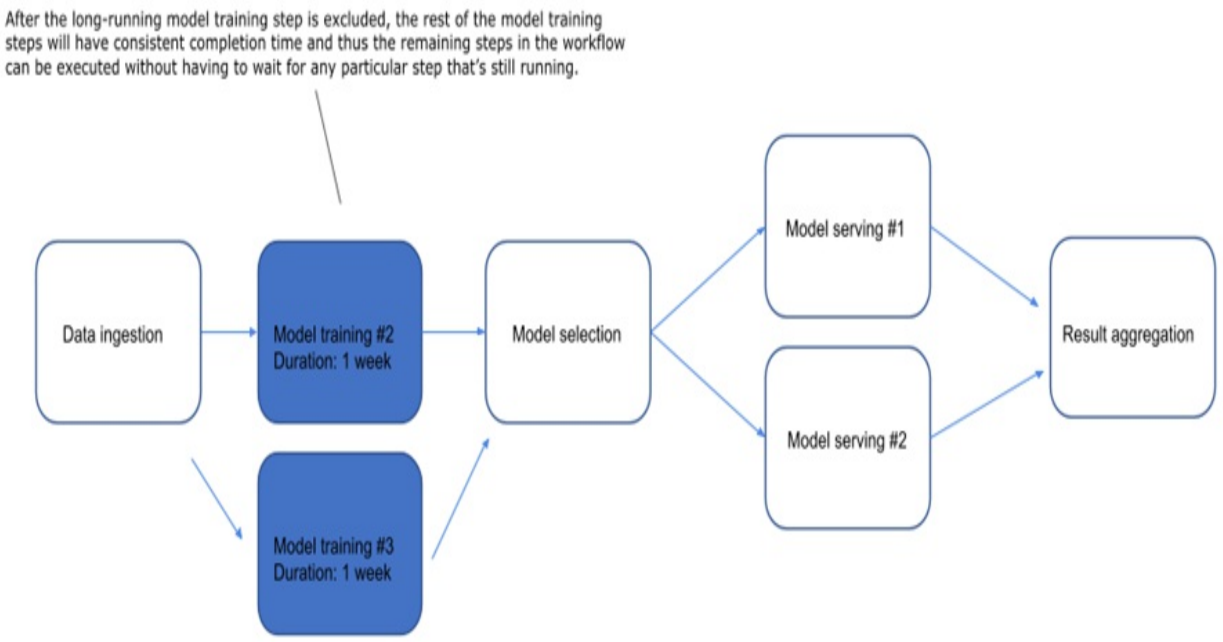
5.3.2 Solution

We would like to build the same machine learning workflow as before, which would train different models after the system has ingested data from the data source, select the top two models, and then use these two models to provide model serving for users that generates predictions with the knowledge from both of the models.

However, this time we noticed some performance bottleneck. Since the start of each of the following steps, such as the model selection step and the model serving steps, depend on the completion of its previous steps. In the meantime, there's one long-running model training step that must be completed before we proceed to the next step. The entire workflow is being delayed by this particular long-running step.

What if we can exclude the long-running model training step completely? Once we do that, the rest of the model training steps will have consistent completion time and thus the remaining steps in the workflow can be executed without having to wait for any particular step that's still running. A diagram of the updated workflow is shown in Figure 5.15 below.

Figure 5.15 The new workflow after the long-running model training step has been removed.



Though this naive approach may resolve our issue of extra waiting time for long-running steps. However, our original goal of having this type of complex workflow is to experiment with different machine learning model architectures and different sets of hyperparameters of those models in order to select the best performing models to be used for model serving. If we are simply excluding the long-running model training steps, we are essentially throwing away the opportunity to experiment with advanced models that may better capture the entities in the videos.

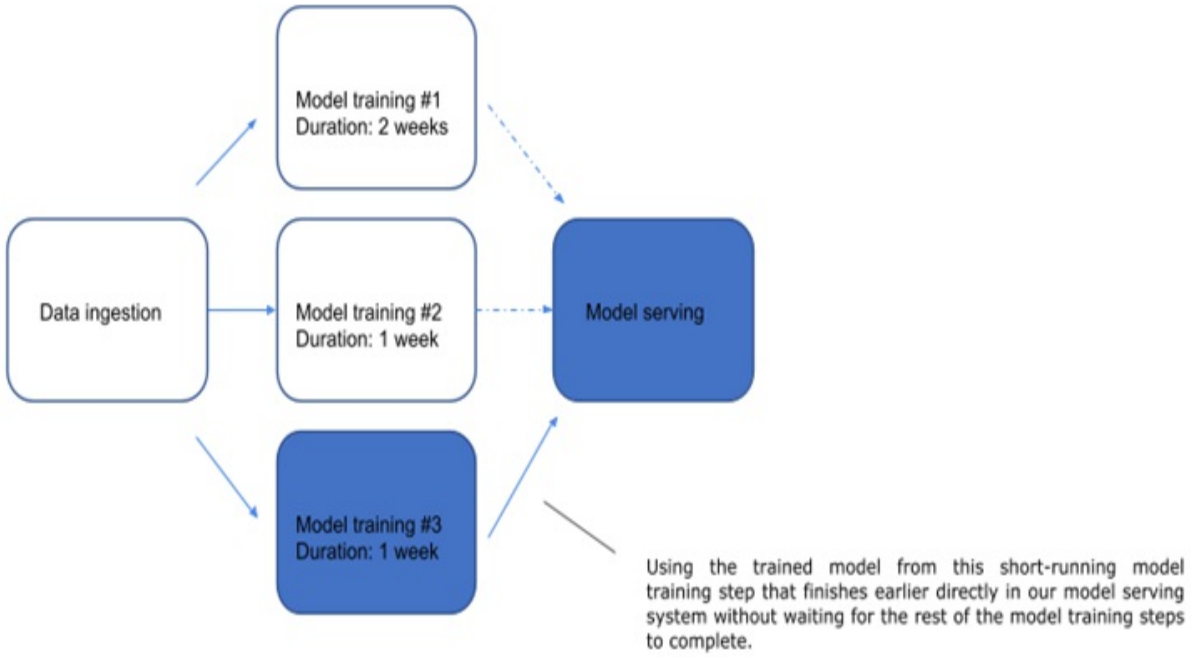
Is there a better way to speed up the workflow so that it will not be affected by the durations of individual steps?

Let's now focus our attention on the model training steps that only take one week to complete. What can we do when those short-running model training steps finish?

When one of those short-running model training steps finishes, we successfully obtain a trained machine learning model. In fact, we can already leverage this already trained model and use it directly in our model serving system without waiting for the rest of the model training steps to complete. As a result, the users would be able to see results of tagged entities from their model serving requests that contain videos as soon as we have trained one

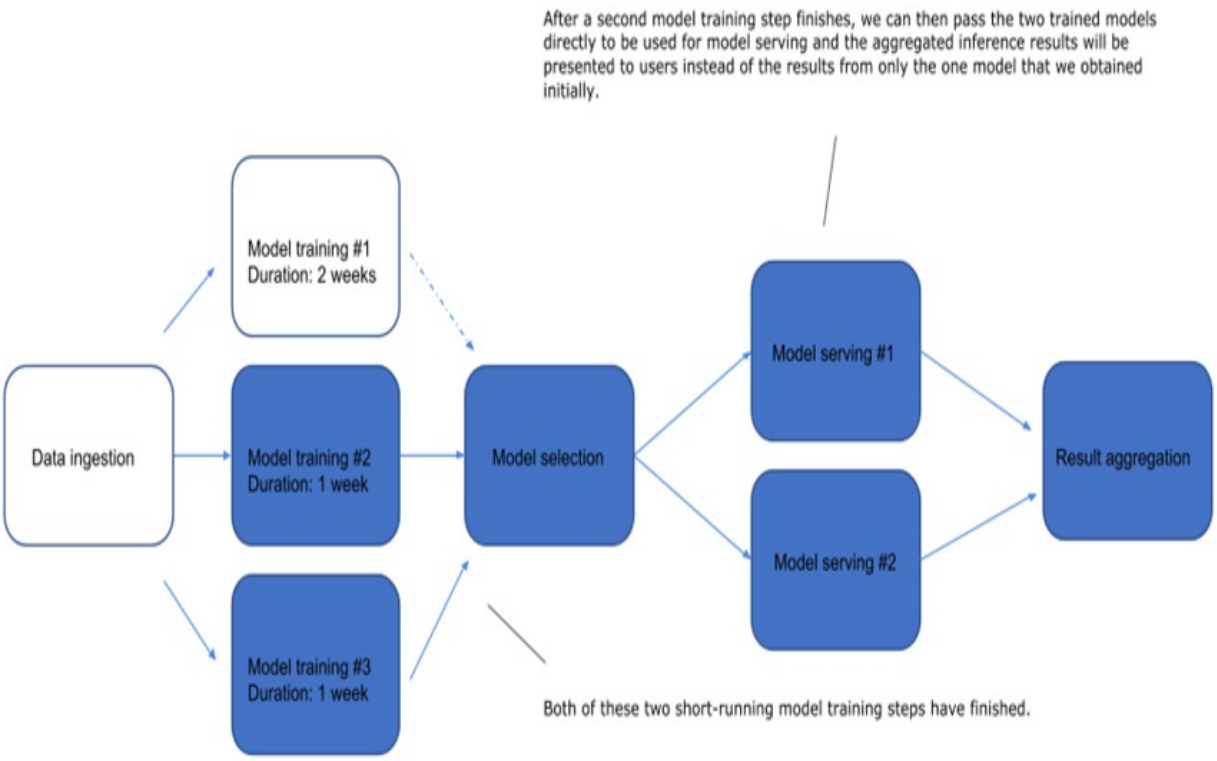
model from one of the steps in the workflow. A diagram of this workflow is shown in Figure 5.16 below.

Figure 5.16 The workflow where we use the trained model from this short-running model training step that finishes earlier directly in our model serving system without waiting for the rest of the model training steps to complete.



After a second model training step finishes, we can then pass the two trained models directly to be used for model serving and the aggregated inference results will be presented to users instead of the results from only the one model that we obtained initially, as shown in Figure 5.17 below.

Figure 5.17 After a second model training step finishes, we can then pass the two trained models directly to be used for model serving and the aggregated inference results will be presented to users instead of the results from only the one model that we obtained initially.



Note that while we can continue to use the trained models for model selection and model serving, the long-running model training steps are still running. In other words, they are executing *asynchronously* without depending on each other's completion. The workflow can proceed and start executing the next step before the previous step finishes.

On the other hand, *synchronous* steps are performed one at a time and only when one is completed, the following step becomes unblocked. In other words, you need to wait for a step to finish to move to the next one. For example, the data ingestion step must be completed before we start any of the model training steps.

By incorporating these patterns, the entire workflow will no longer be blocked by the long-running model training step. Instead, it can continue to use the already trained models from the short-running model training step in the model serving system and can start handling users' model serving requests.

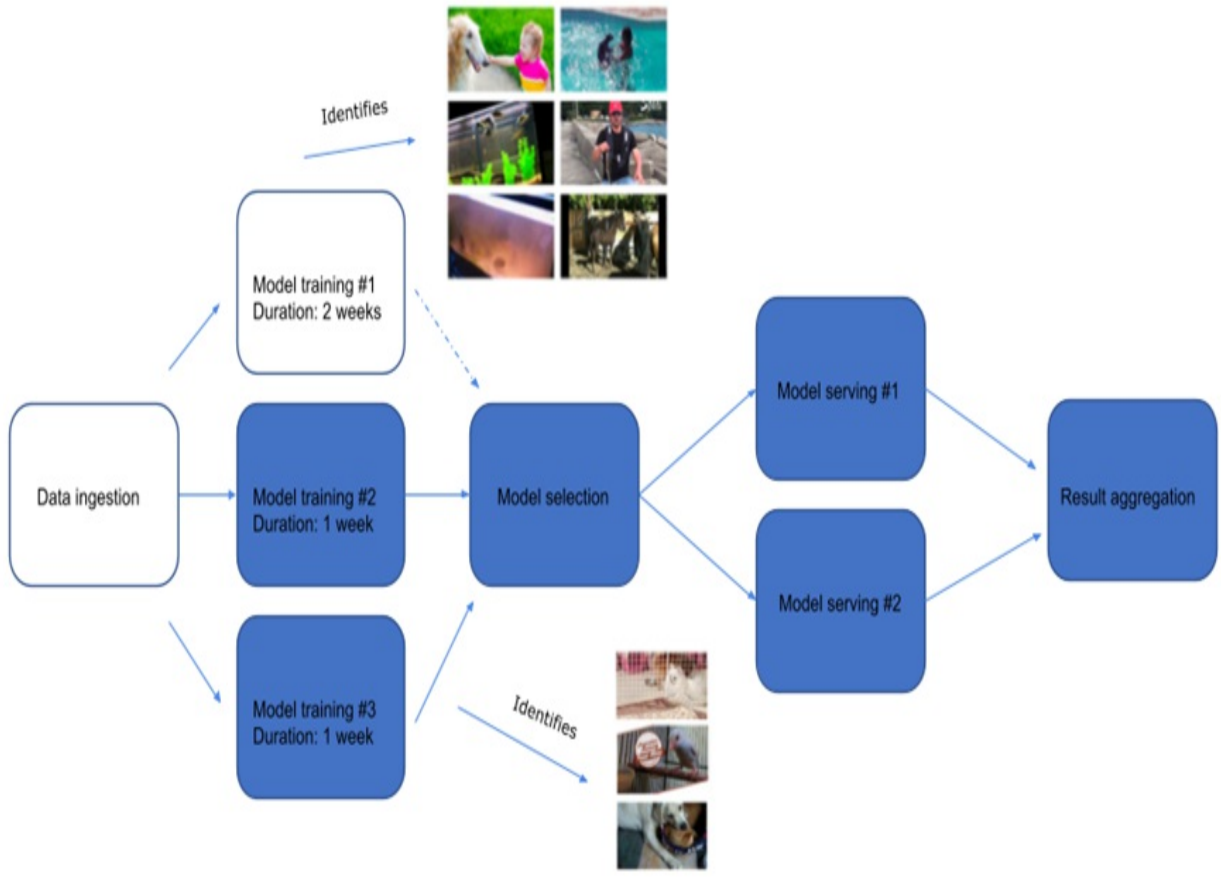
The synchronous and asynchronous patterns are also extremely useful in other distributed systems to optimize system performance and maximally leverage existing computational resources. This is especially useful when there's a limited amount of computational resources with heavy workloads. We'll apply this pattern in Section 9.4.1.

5.3.3 Discussion

With the help of the mix of synchronous and asynchronous patterns, we can create more efficient machine learning workflows and avoid any delays due to steps that prevent others from executing, such as the long-running model training steps.

The models trained from the short-running model training steps may not be accurate enough. In other words, they may not discover all the entities in the videos very well due to the simplicity of their model architectures. For example, the model trained from two finished short-running model training steps may be very simple models that serve as a baseline that is only able to identify a small number of entities whereas the model trained from the most time-consuming step can identify more entities. A diagram that illustrates this is shown in Figure 5.18 below.

Figure 5.18 The model trained from two finished short-running model training steps may be very simple models that serve as a baseline that is only able to identify a small number of entities whereas the model trained from the most time-consuming step can identify more entities.



As a result, we should keep in mind that the best models that we get early on may not be the best models and may only be able to tag a small number of entities that may not be satisfactory to our users.

When we deploy this end-to-end workflow to real-world applications, we should think about whether it’s more important for users to see inference results faster or see better results. If the goal is to allow users to see the inference results instantly as soon as a new model is available, then they may not see good enough results that they might be expecting. Alternatively, if certain delays can be acceptable to the users, it’s better to wait for more model training steps to finish and be selective to the models we've trained in order to pick the best performing models that will present very good entity tagging results. However, whether the delays are acceptable or insignificant will be subject to the requirements of the real-world applications.

By leveraging the synchronous and asynchronous patterns, we can organize the steps in machine learning workflows from both structural and

computational perspectives. As a result, data science teams would spend less time waiting for workflows to complete due to maximizing performance and in the meantime, infrastructure costs can be reduced as well since we have fewer idling computational resources. In the next section, we'll introduce another pattern that's used very often in real-world systems that would save more computational resources and make workflows run even faster.

5.3.4 Exercises

1. What is the root cause for the start of the following steps of the model training steps?
2. Are the steps blocking each other if they are running asynchronously?
3. What do we need to consider when deciding whether we want to use any available trained model as early as possible?

5.4 Step memoization pattern: Skipping redundant workloads via memoized steps

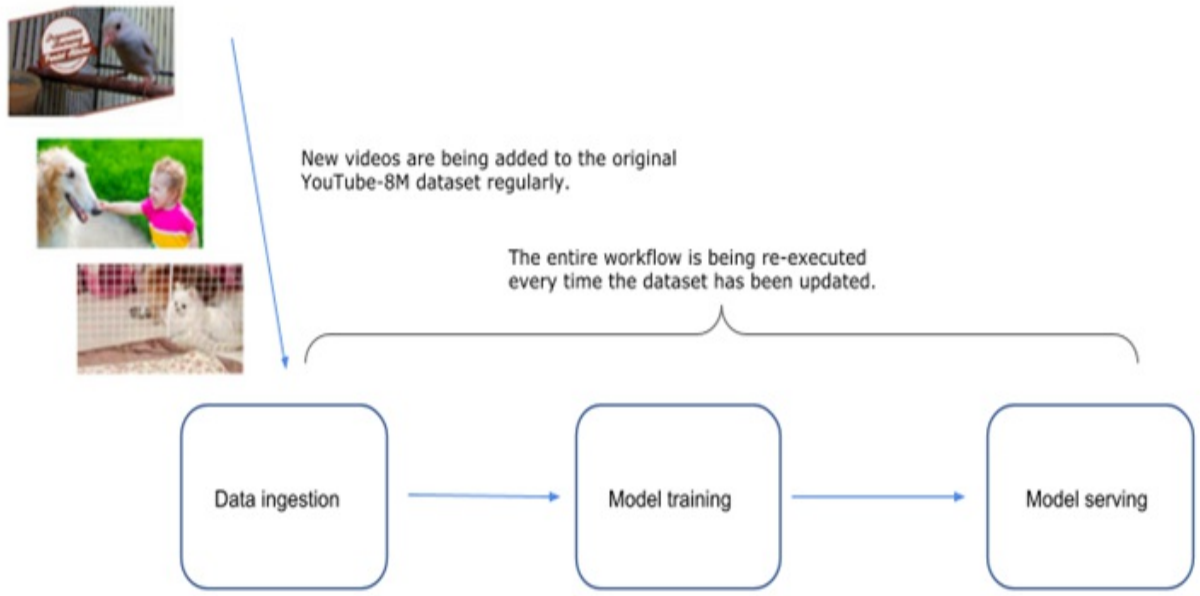
With the fan-in and fan-out patterns in the workflow, the system is able to execute complex workflows that train multiple machine learning models and pick the most performant ones to provide good entity tagging results in the model serving system. Additionally, the use of synchronous and asynchronous patterns makes machine learning workflows more efficient and delays due to the long-running model training steps that block other consecutive steps can be avoided.

It's worth mentioning that the workflows we've seen in this chapter only contain a single data ingestion step. In other words, the data ingestion step in the workflows always executes first before the rest of the steps, such as model training and model serving, can begin to process.

Unfortunately, in real-world machine learning applications, the dataset does not always remain unchanged. Now imagine that there are new YouTube videos available that are being added to the YouTube-8M dataset every week. Following our existing workflow architecture, if we would like to re-train the model so that it accounts for the additional videos that arrive on a regular

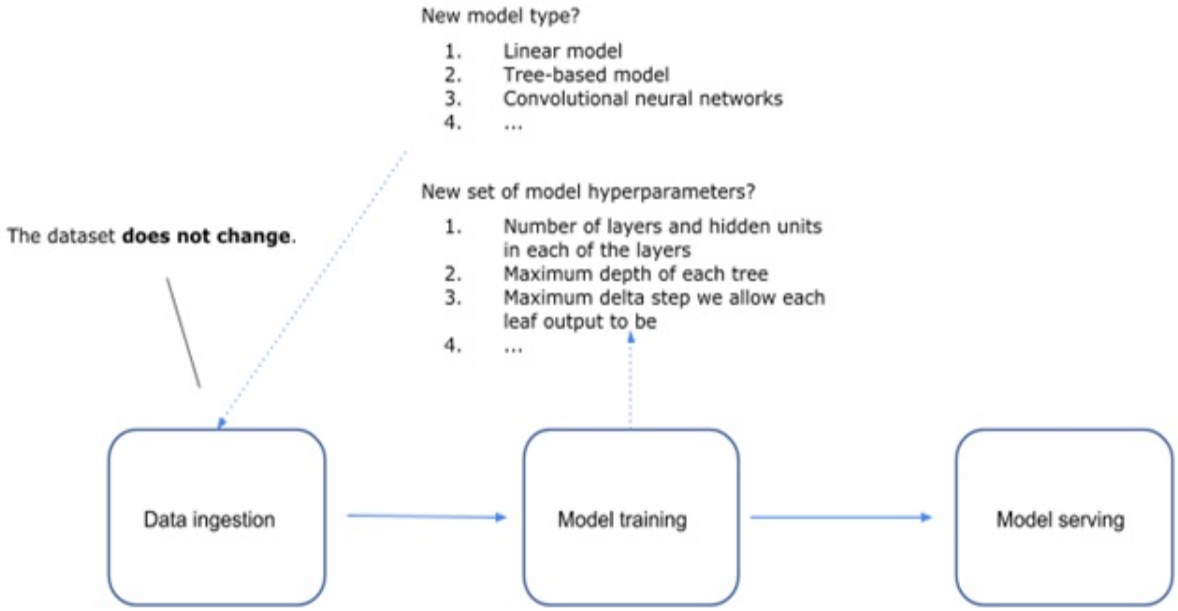
basis, then we need to run the entire workflow regularly from scratch, from the data ingestion step to the model serving step, as shown in Figure 5.19 below.

Figure 5.19 Diagram where the entire workflow is being re-executed every time the dataset has been updated.



Similarly as shown in Figure 5.20, when the dataset has not changed but we would like to experiment new model architectures or new sets of hyperparameters, which is very common for machine learning practitioners. For example, we may change the model architecture from simple linear models to more complex models such as tree-based models or convolutional neural networks. We can also stick with the particular model architecture we've used and only change the set of model hyperparameters such as the number of layers and hidden units in each of those layers for neural network models, or the maximum depth of each tree for tree-based models. For cases like these, we still need to run the end-to-end workflow that includes the data ingestion step to re-ingest the data from the original data source from scratch. Performing data ingestion again would be very time-consuming.

Figure 5.20 Diagram where the entire workflow is being re-executed every time a new model type or hyperparameters are being experimented while the dataset has not changed.



5.4.1 Problem

Machine learning workflows usually start with a data ingestion step. If the dataset is being updated regularly, we may want to re-run the entire workflow in order to train a fresh machine learning model that takes the new data into account, which means that we need to execute the data ingestion step every time. On the other hand, if the dataset is not updated but we would still like to experiment new models, we still need to execute the entire workflow that includes the data ingestion step.

However, the data ingestion step usually takes a long time to complete depending on the size of the dataset. Is there a way to make this workflow more efficient?

5.4.2 Solution

Given how time-consuming data ingestion steps usually are, we probably don’t want to re-execute it every time when the workflow runs to re-train or update our entity tagging models.

Let’s first think about the root cause of this problem. The dataset of YouTube

videos is being updated regularly and the new data is persisted to the data source on a regular basis as well, e.g. once a month.

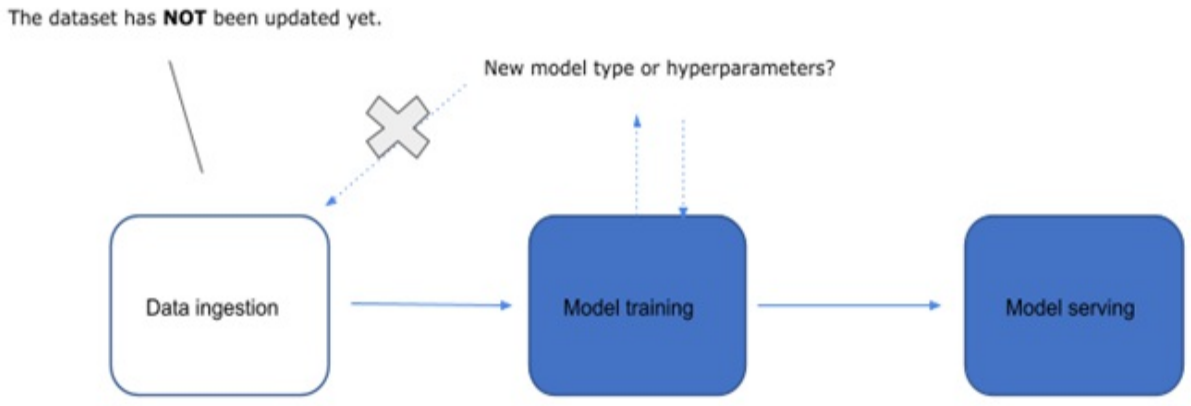
We have two use cases where we need to re-execute the entire machine learning workflow:

1. After the dataset has been updated, re-run the workflow to train a new model that uses the updated dataset;
2. We want to experiment with a new model architecture using that dataset that's already ingested, which may not have been updated yet.

The fundamental issue is around the time-consuming data ingestion step. With the current workflow architecture, the data ingestion step will need to be executed regardless whether the dataset has been updated or not.

Ideally, we don't want to re-ingest the data that's already collected again if the new data has not been updated yet. In other words, we would like to only execute the data ingestion step when we know that the dataset has already been updated, as shown in Figure 5.21.

Figure 5.21 Diagram where the data ingestion step is skipped when the dataset has not been updated yet.

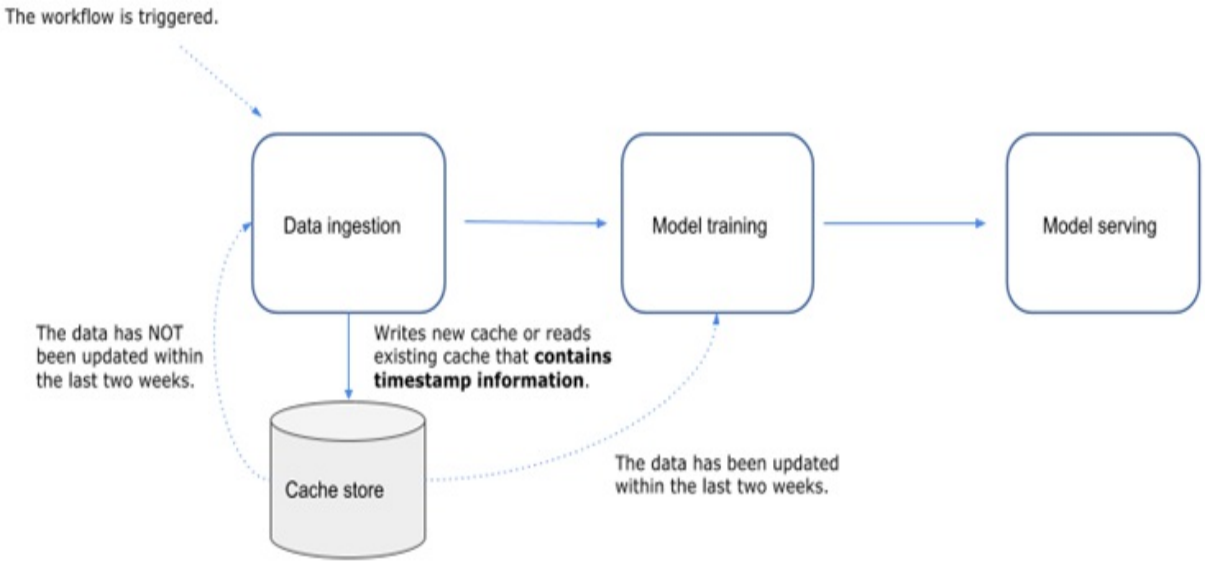


Now the challenge comes down to a question on how to tell whether the dataset has already been updated. Once we have a way to identify that, we can conditionally re-construct the machine learning workflow and control whether we would like to include a data ingestion step to be re-executed as

shown in the diagram above.

One way to identify whether the dataset has been updated is through the use of cache. Since our dataset is being updated regularly on a fixed schedule, e.g. once a month, we can create a *time-based cache* that stores the location of the ingested and cleaned dataset (assuming the dataset is located in a remote database) and the timestamp of its last updated time. The data ingestion step in the workflow will then be constructed and executed dynamically based on whether the last updated timestamp is within a particular window. For example, if the time window is set to be two weeks, then we consider the ingested data as fresh if it has been updated within the past two weeks. The data ingestion step will be skipped and the following model training steps will use the already ingested dataset from the location that's stored in the cache. Figure 5.22 illustrates the case where a workflow has been triggered and we check whether the data has been updated within the last two weeks by accessing the cache. If the data is fresh, we skip the execution of the unnecessary data ingestion step and execute the model training step directly.

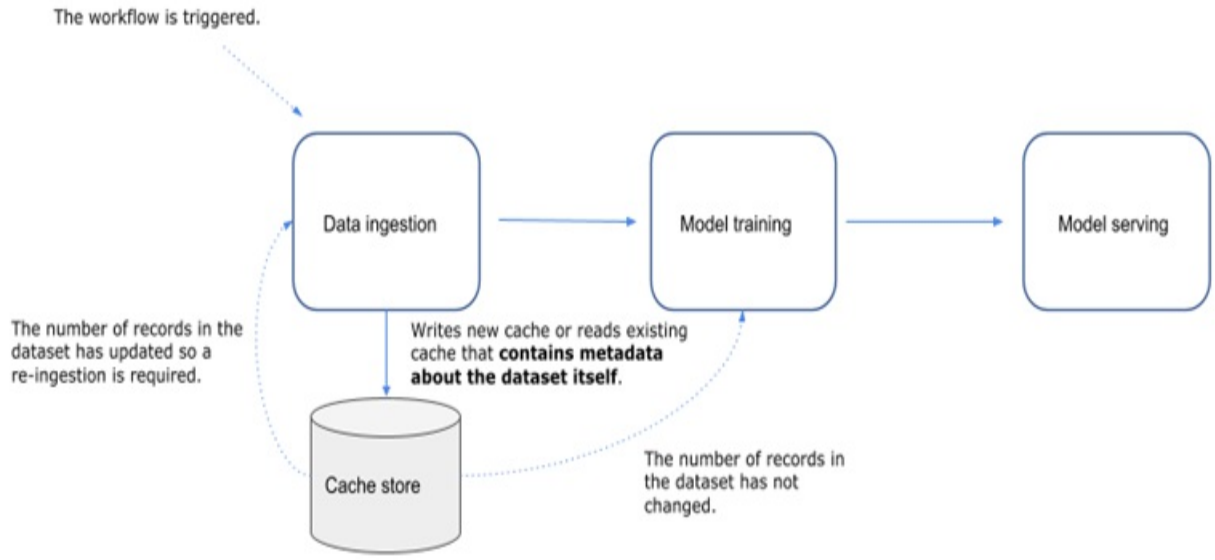
Figure 5.22 The workflow has been triggered and we check whether the data has been updated within the last two weeks by accessing the cache. If the data is fresh, we skip the execution of the unnecessary data ingestion step and execute the model training step directly.



The time window can be used to control how old a cache can be before we consider the dataset as fresh enough that it can be used directly for model training, instead of re-ingesting the data again from scratch.

Alternatively, we can also store some of the important metadata about the data source in the cache, such as the number of records in the original data source currently available. This type of cache is called *content-based cache* since they store information extracted from the particular step such as the input and output information. With this type of cache, we can identify whether the data source has significant changes, e.g. the number of original records have doubled in the data source. If there's significant change, it's usually a signal to re-execute the data ingestion step since the current dataset is very old and outdated. A workflow that illustrates this approach is shown in Figure 5.23 below.

Figure 5.23 The workflow has been triggered and we check whether the metadata collected from the dataset itself, such as the number of records in the dataset, is significant. If it's not significant, we then skip the execution of the unnecessary data ingestion step and execute the model training step directly.



This pattern, which leverages the cache to decide whether a step should be executed or skipped, is called *step memoization*. With the help of step memoization, a workflow can identify the steps with redundant workloads that can be skipped without being re-executed and thus greatly accelerate the

execution of the end-to-end workflow. We'll apply this pattern in Section 9.4.2.

5.4.3 Discussion

In real-world machine learning applications, there are many workloads that are computationally heavy and time-consuming besides data ingestion. For example, the model training step takes up a lot of computational resources in order to achieve high-performance model training which sometimes takes weeks to complete. If we are only experimenting with other components that do not require updating the trained model, it might make sense to avoid executing the expensive model training step again. The step memoization pattern comes handy in those situations to skip the heavy and redundant steps when appropriate.

It may not be trivial sometimes to decide on what type of information to be extracted and stored in the cache if we are creating content-based caches. For example, if we are trying to cache the results from a model training step, we may want to consider using the trained model artifact that includes information such as the type of machine learning model and the set of hyperparameters of the model. When the workflow is executed again, it will decide whether to re-execute the model training step based on whether we are trying the same model or not. Alternatively, we may also store information like the performance statistics (e.g. accuracy, mean-squared error, etc.) to identify whether it's beyond a threshold and worth training a more performant model.

Furthermore, when applying the step memoization pattern in practice, be aware that it requires a certain level of maintenance efforts to manage the lifecycle of the created cache. For example, if there are a thousand machine learning workflows everyday with an average of a hundred steps for each workflow that are being memoized, then there will be 100,000 caches created everyday. These caches take up a certain amount of space depending on the type of information they store but can accumulate rather quickly.

In order to apply this pattern at scale, there must be a garbage collection mechanism in place to delete any caches automatically to prevent the

accumulation of caches overtime that takes up a huge amount of disk space. For example, one simple strategy is to record the timestamp when the cache is last hit and used by a step in a workflow and then scanning the existing caches periodically to clean up the caches that are not used or hit after a long time.

5.4.4 Exercises

1. What type of steps can be most benefited from step memoization?
2. How do we tell if a step's execution can be skipped if its workflow has been triggered to run again?
3. What do we need to manage and maintain once we've leveraged the pattern in order to apply the pattern at scale?

5.5 References

1. Ensemble learning: https://en.wikipedia.org/wiki/Ensemble_learning

5.6 Summary

- Workflow is an essential component in machine learning systems as it connects all other components in a machine learning system. A machine learning workflow can be as easy as chaining just data ingestion, model training, and model serving.
- The fan-in and fan-out patterns can be incorporated into complex workflows so that they are maintainable and composable.
- The synchronous and asynchronous patterns accelerate the machine learning workloads with the help of concurrency.
- With the step memoization pattern, the performance of the workflows can be improved via skipping duplicate workloads.

Answers to Exercises:

Section 5.2

1. No, because we have no guarantee in what order concurrent copies of

those steps will run.
2. Training an ensemble model depends on the completion of other model training steps for the sub-models and we cannot use the fan-in pattern because the ensemble model training step will need to wait for other model training to complete before it can start running, which would require some extra waiting and delay the entire workflow.

Section 5.3

1. Due to the variance of completion time for each of the model training steps in the existing machine learning workflow, the start of each of the following steps, such as the model selection step and the model serving steps, depend on the completion of its previous steps.
2. No, asynchronous steps won’t block each other.
3. We need to consider it from the user's perspective. We should think about whether it’s more important for users to see inference results faster or see better results. If the goal is to allow users to see the inference results instantly as soon as a new model is available, then they may not see good enough results that they might be expecting. Alternatively, if certain delays can be acceptable to the users, it’s better to wait for more model training steps to finish and be selective to the models we've trained in order to pick the best performing models that will present very good entity tagging results.

Section 5.4

1. Steps that are time-consuming or require a huge amount of computational resources.
2. We can use the information stored in the cache, such as when the cache is initially created or metadata collected from the step, to decide whether we should skip an execution of a particular step.
3. We need to set up a garbage collection mechanism to automatically recycle and delete the created caches.

6 Operation patterns

This chapter covers

- Recognizing different areas of improvements related to operations in machine learning systems, such as job scheduling and metadata.
- Preventing resource starvation and avoiding deadlocks via different scheduling techniques, such as fair-share scheduling, priority scheduling, and gang scheduling.
- Gaining insights from machine learning workflows and handling failures more appropriately to reduce negative impact on users via the metadata pattern.

In Chapter 5, we've focused our discussion on machine learning workflows and the challenges involved when building them in practice. Workflow is an essential component in machine learning systems as it connects all other components in a machine learning system. A machine learning workflow can be as easy as chaining just data ingestion, model training, and model serving. On the other hand, it can be very complex to handle different real-world scenarios to allow additional steps and performance optimizations to be part of the entire workflow. It's essential to know what trade-offs we may see when making different design decisions in order to meet different business and performance requirements. Previously we've introduced a few established patterns adopted heavily in industries. Each of these established patterns can be reused to build simple to complex machine learning workflows that are efficient and scalable. For example, in Section 5.2, we've seen how to build a system to execute complex machine learning workflows to train multiple machine learning models and pick the most performant ones to provide good entity tagging results in the model serving system, with the fan-in and fan-out patterns. In Section 5.3, we've also leveraged the synchronous and asynchronous patterns to make machine learning workflows more efficient and avoid delays due to the long-running model training steps that block other consecutive steps.

Since real-world distributed machine learning workflows can be extremely

complex, as seen in Chapter 5, there's a huge amount of *operational* work involved to help maintain and manage the various components of the systems, such as improvements to system efficiency, observability, monitoring, deployment, etc. These operational work efforts usually require a lot of communication and collaboration between the DevOps and data science teams. For instance, the DevOps team may not have enough domain knowledge in machine learning algorithms that the data science team uses to help debug any encountered issues or optimize the underlying infrastructure to accelerate the machine learning workflows. For a data science team, the type of computational workload varies, depending on the team structure and the way team members collaborate. As a result, there's no universal way for the DevOps team to handle the requests of different workloads from the data science team.

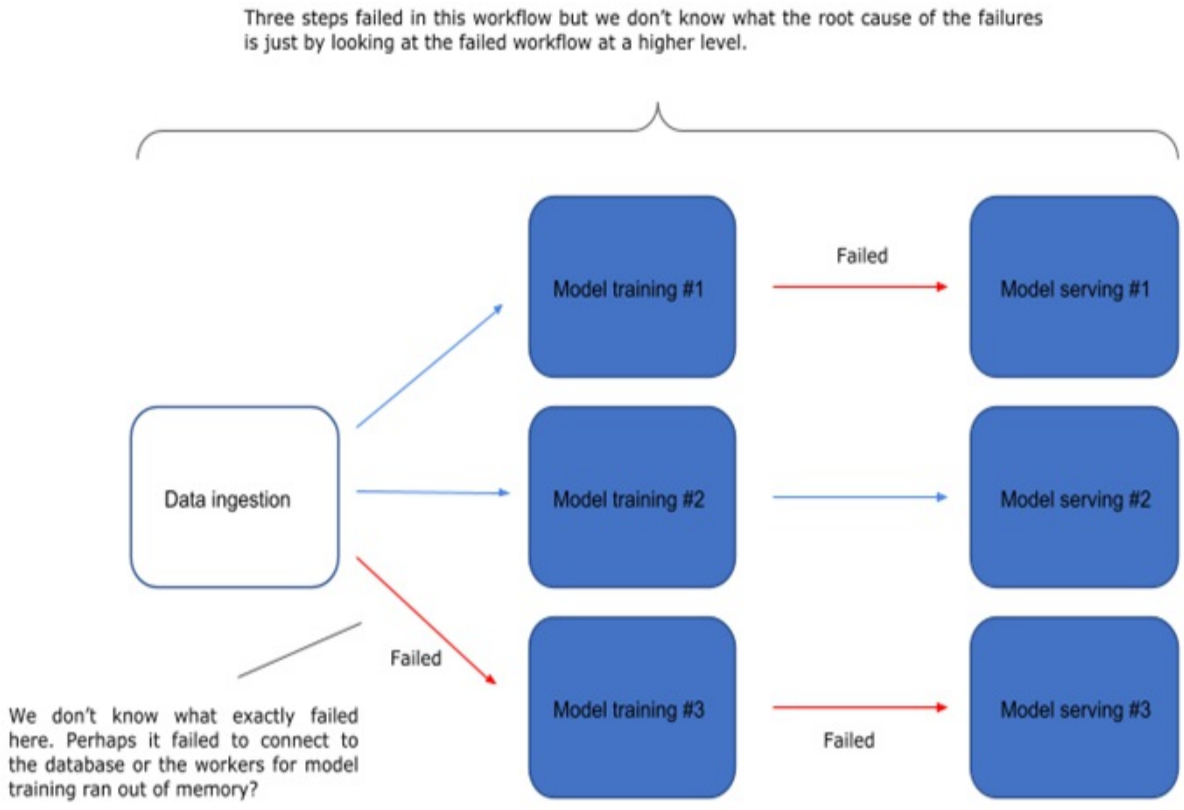
Fortunately, there are operational efforts and patterns that can be leveraged to greatly accelerate the end-to-end workflow and reduce maintenance and communication efforts when engineering teams are collaborating with teams of data scientists or machine learning practitioners before the systems become production ready. In this chapter, we'll explore some of the challenges involved when performing operations on machine learning systems in practice and introduce a few established patterns adopted heavily in industries. For example, we'll leverage some scheduling techniques to prevent resource starvation and avoid deadlocks when there are many team members collaboratively working in the same cluster with limited computational resources. We will also discuss the benefits of the metadata pattern that could be used to gain insights from the individual steps in machine learning workflows and help us handle failures more appropriately to reduce negative impact on users. Note that since the scope of MLOps can be extremely large according to different contexts, for this chapter, we will only focus on a selected set of mature patterns at the time of writing. Please also expect some updates to any future versions of this chapter as this field evolves.

6.1 What are operations in machine learning systems?

For the scope of this chapter, we will focus on operational techniques and patterns that are commonly seen in more than one components or steps in a machine learning workflow, instead of patterns that are specific to each individual component.

For example, the workflow shown in Figure 6.1 consists of three failed steps where there are multiple model training steps after data ingestion and multiple model serving steps after the multiple model training steps. Unfortunately each of these steps are like blackboxes and we don't know much details about them yet since at this point we only know whether they fail and whether the failures have affected the following steps. As a result, it's really hard to debug.

Figure 6.1 An example workflow where there are multiple model training steps after data ingestion and multiple model serving steps after the multiple model training steps. Note that there are three failed steps.



Operation patterns that we are going to introduce in this chapter would

increase the visibility of the entire workflow to help us understand the root cause of the failures and give us some ideas on how to handle the failures properly. In addition, the increased observability could help us come up with improvements to system efficiency that are beneficial for future executions of similar workflows.

Note What about MLOps?

We often hear the term MLOps nowadays, which is a term derived from machine learning or ML and operations or Ops. It usually means a collection of practices for managing machine learning lifecycles in production, which may include practices from machine learning and DevOps to efficiently and reliably deploy and manage machine learning models in production.

MLOps usually require communication and collaboration between DevOps and data science teams. It focuses on improving the quality of production machine learning and embracing automation while maintaining business requirements.

The scope of MLOps can be extremely large and varies depending on the context.

Given how large the scope of MLOps can be according to different contexts, for this chapter, we will only focus on a selected set of mature patterns at the time of writing. Please also expect some updates to any future versions of this chapter as this field evolves.

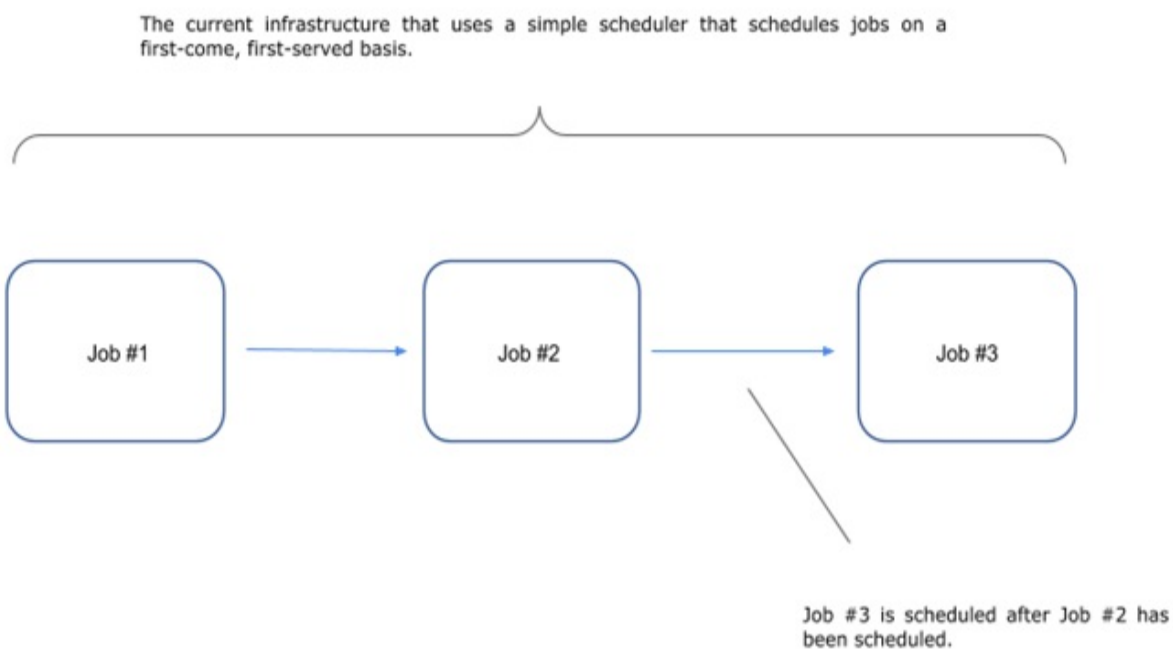
6.2 Scheduling patterns: Assigning resources effectively in a shared cluster

Let’s assume we have successfully set up the distributed infrastructure for users to submit distributed model training jobs that are scheduled to run on multiple CPUs by a default *scheduler*. A scheduler is responsible for assigning computational resources to perform tasks requested by the system. It is designed to keep computational resources busy and allow multiple users to collaborate with shared resources more easily. Different users are trying to build models using the shared computational resources in the cluster for

different scenarios. For example, one user is working on a fraud-detection model that tries to identify financial fraudulent behaviors such as international money laundry. Another user is working on a condition monitoring model that could generate a health score to represent the current condition for industrial assets such as components on trains, airplanes, wind turbines, etc.

The current infrastructure only provides a simple scheduler that schedules jobs on a first-come, first-served basis, as shown in Figure 6.2. For example, the third job is scheduled after the second job has been scheduled and each job's computational resources are allocated upon scheduling.

Figure 6.2 A diagram of the current infrastructure only provides a simple scheduler that schedules jobs on a first-come, first-served basis.



In other words, the users that schedule jobs later would have to wait for all previously submitted jobs to finish before their model training jobs can start executing. Unfortunately, in reality, users often want to submit multiple model training jobs to experiment different sets of models or hyperparameters, which would often block other users' model training jobs from executing since those previously submitted experiments are already

utilizing all the available computational resources.

When this happens, users would have to compete for resources, e.g. waking up in the middle of the night to submit model training jobs when there are less users using the system. As a result, the collaboration among team members may not be pleasant. Some jobs include training very large machine learning models which usually take up a lot of computational resources and thus increases the time others users have to wait if they submit jobs later.

In addition, for distributed model training jobs, if we have only scheduled some of the requested workers, the model training cannot begin to execute until all of the requested workers are ready due to the nature of the distribution strategy such as distributed training with the collective communication pattern. When there is a lack of available computational resources to be used by this job, it would never start and the already allocated computational resources for the existing workers would be wasted.

6.2.1 Problem

We have set up a distributed infrastructure for users to submit distributed model training jobs that are scheduled to run by a default scheduler that is responsible for assigning computational resources to perform various tasks requested by the users.

However, the default scheduler only provides a simple scheduler that schedules jobs on a first-come, first-served basis. As a result, when this cluster is used by different users, the users often need to wait for a long time for available computational resources until the previously submitted jobs are completed. In addition, distributed model training jobs cannot begin to execute until all of the requested workers are ready due to the nature of the distributed training strategy such as collective communication strategy.

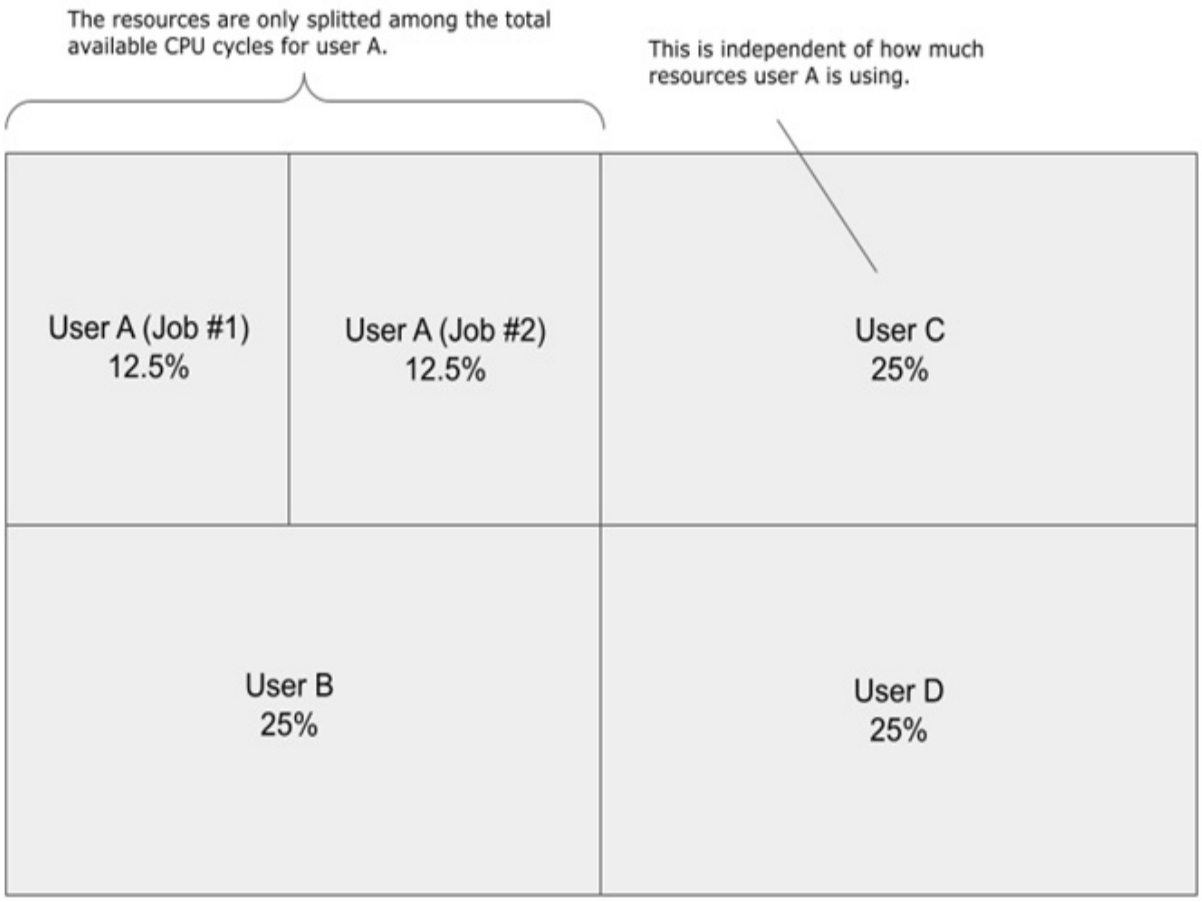
Are there any alternatives to the existing default scheduler so that we could assign the computational resources more effectively in a shared cluster?

6.2.2 Solution

In our scenario, the problem starts to occur when multiple users are trying to use the system to submit distributed model training jobs at the same time. Since the jobs are being executed on a first-come, first-served basis, there will be long waiting time for jobs that are submitted later, even when those jobs might be submitted from different users.

It’s easy to identify different users so an intuitive solution would be to limit how much of the total computational resources each user is able to use. For example, if there are four users (A, B, C, and D), once user A submits a job that uses 25% of the total available CPU cycles, this user would no longer be able to submit another job unless those allocated resources have been released and ready to be allocated to new jobs. Other users would be able to submit jobs independent of how much resources user A is using. For example, if user B starts a second process that uses the same amount of resources as the first process, the other users will still each receive 25% of the total cycles, but each of user B's processes will now be attributed 12.5% of the total CPU cycles each, totalling user B's share of 25%. Figure 6.3 illustrates the resource allocations among these four users.

Figure 6.3 The resource allocations among the four users (A, B, C, and D).



On the other hand, if a new user E starts a process on the system, the scheduler will reapportion the available CPU cycles such that each user gets 20% of the whole (100% / 5 = 20%).

The way we schedule our workloads to execute in our cluster is called *fair-share scheduling*. It is a scheduling algorithm for computer operating systems in which the CPU usage is equally distributed among system users or groups, as opposed to equal distribution among processes.

Note that so far we only discussed partitioning resources among the users. When there are multiple teams that are using the system to train their machine learning models where each team has multiple members, we can also partition users into different groups and then apply the fair-share scheduling algorithm to the groups in addition to users. Specifically, we first divide the available CPU cycles among the groups and then divide further among the users within each of the groups. For example, if there are three

groups containing three, two, and four users respectively, then each group will be able to use 33.3% (100% / 3) of the total available CPU cycles. We can then calculate the available CPU cycles for each users in each group as follows:

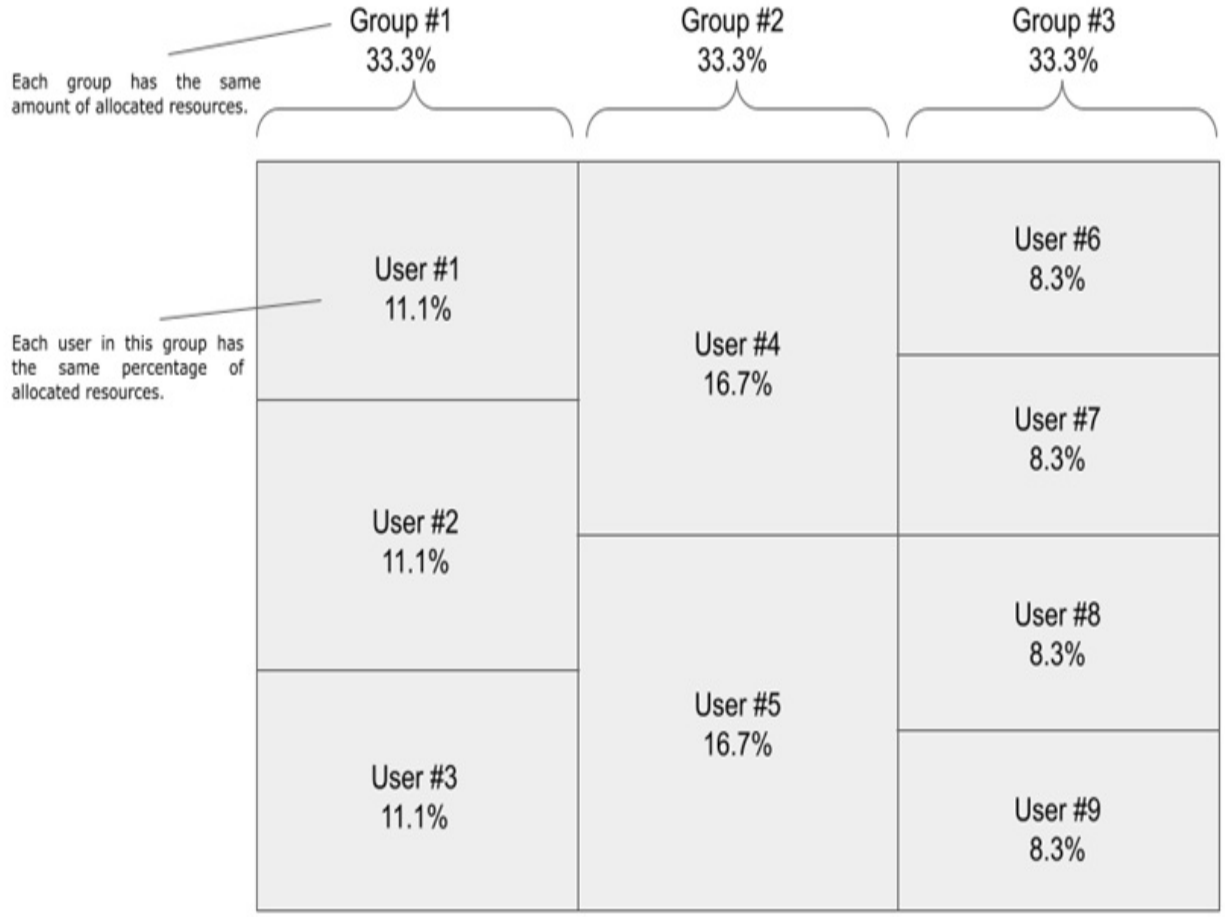
Group #1: (33.3% / 3 users) = 11.1% per user

Group #2: (33.3% / 2 users) = 16.7% per user

Group #3: (33.3% / 4 users) = 8.3% per user

Figure 6.4 also summarizes the resources allocation for each individual user in these three groups that we calculated above.

Figure 6.4 Summary of the resources allocation for each individual user in these three groups.



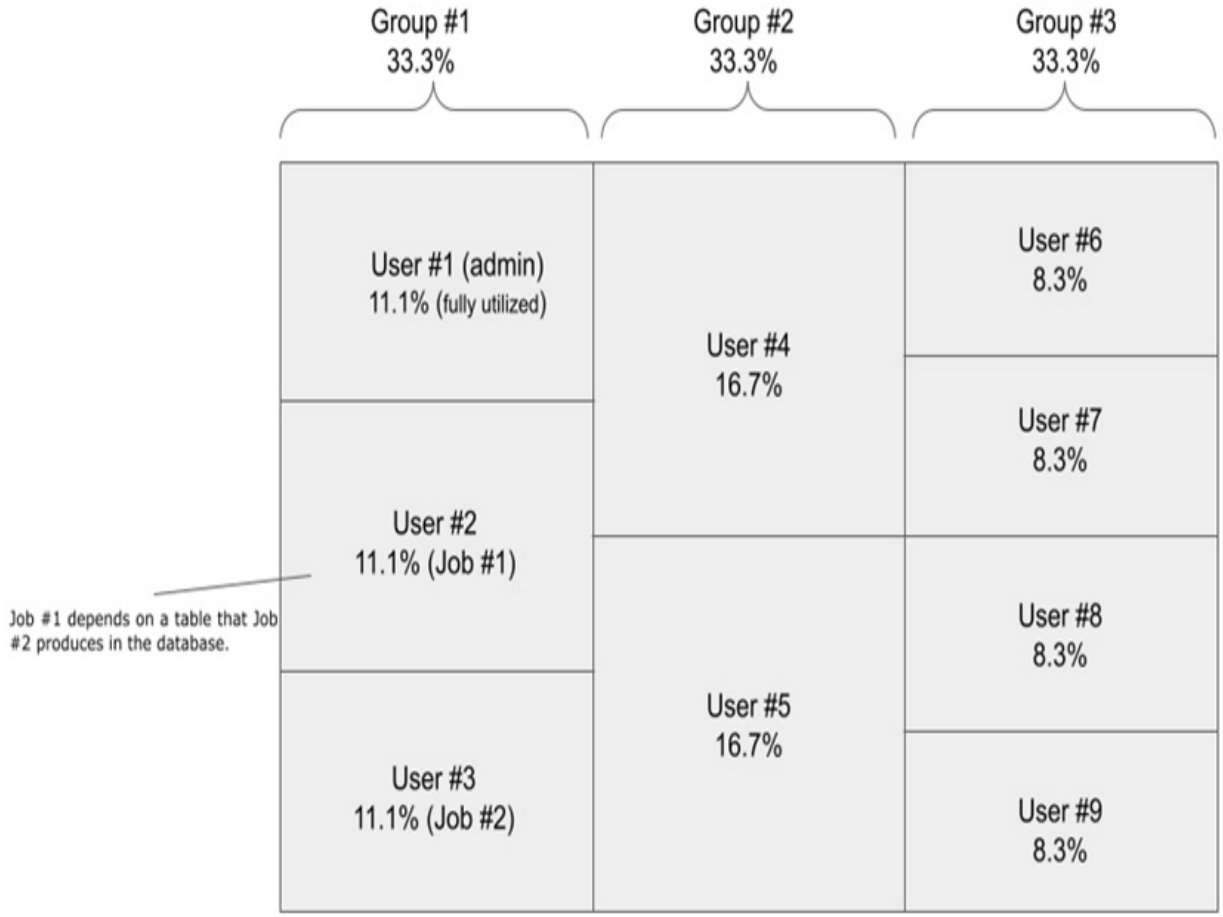
The fair-share scheduling would help us resolve the issue when multiple users are running distributed training jobs concurrently and it allows us to apply this scheduling strategy at each level of abstraction such as processes, users, groups, etc. All users have their own pool of available resources without interfering with each other.

However, there are situations when certain jobs should be executed earlier, e.g. cluster administrator who would like to submit jobs for cluster maintenance such as deleting jobs that have been stuck and taking up resources for a long time. Executing these cluster maintenance jobs earlier would help make more computational resources available and thus unblock others from submitting new jobs.

Let’s assume the cluster administrator is in Group #1 as User #1 and there are two other non-admin users in Group #1 as in the previous example. One of the non-admin users User #2 is running one huge Job #1 that is using all of the 11.1% of the CPU cycles allocated previously to user #2 based on the fair-share scheduling algorithm.

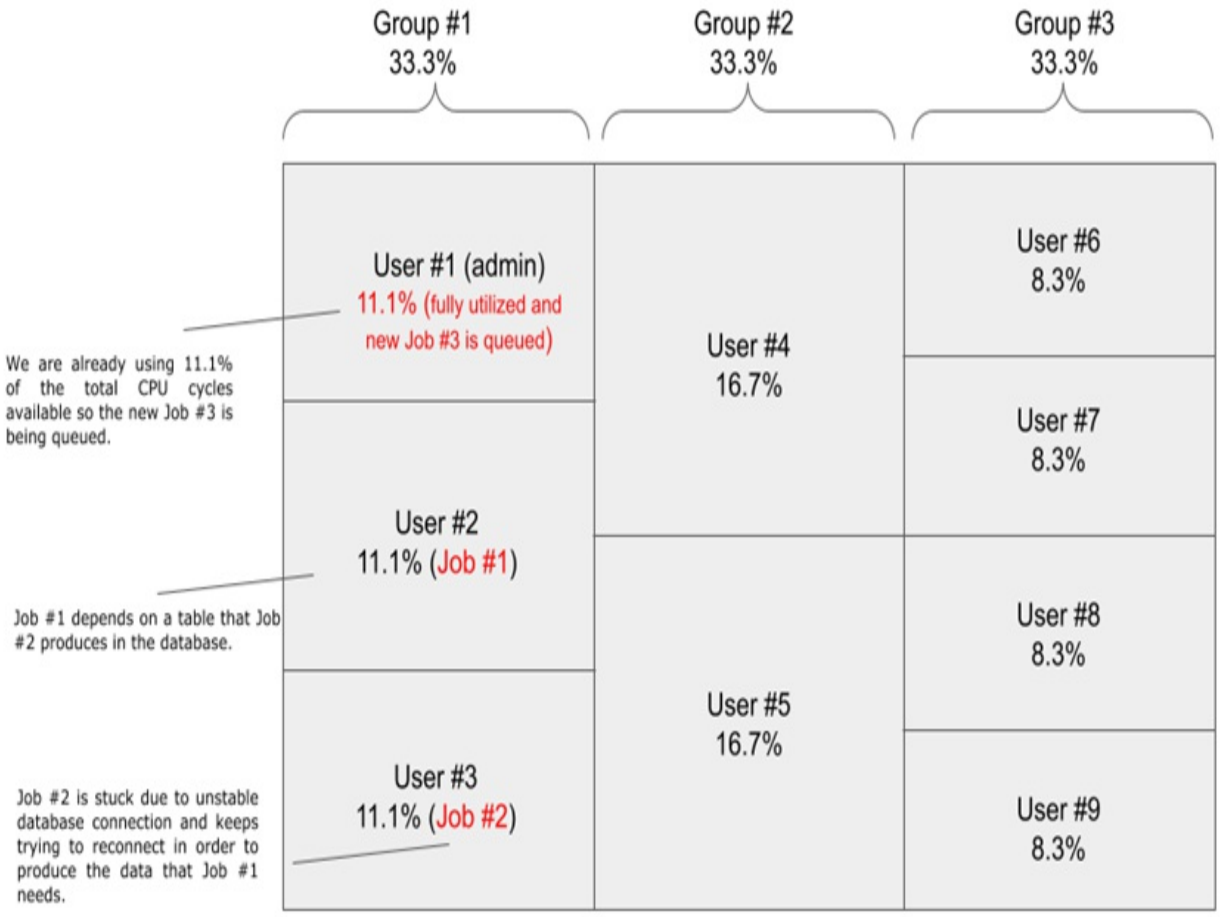
Even though User #2 has enough computational power to perform Job #1 at hand, Job #1 depends on the success of another Job #2 from User #3. For example, Job #2 from User #3 produces a table in the database that Job #1 needs in order to perform a distributed model training task. Figure 6.5 summarizes the resource allocations and usages for each user in the first group.

Figure 6.5 Summary of resource allocations and usages for each user in the first group.



Unfortunately, Job #2 is stuck due to unstable database connection and keeps trying to reconnect in order to produce the data that Job #1 needs. In order to fix the issue, the administrator needs to submit a Job #3 that kills the stuck Job #2 and restart Job #2. Now assume that the admin User #1 is already using 11.1% of the total CPU cycles available. As a result, since the maintenance Job #3 is submitted later than all previous jobs, it is added to the job queue and waits to be executed when resources are released, based on the first-come, first-served nature of our fair-share scheduling algorithm. As a result, we have just encountered a *deadlock* where no job can proceed, as illustrated in Figure 6.6.

Figure 6.6 Admin user is trying to schedule a job to restart the stuck job but encounters a deadlock where no job can proceed.

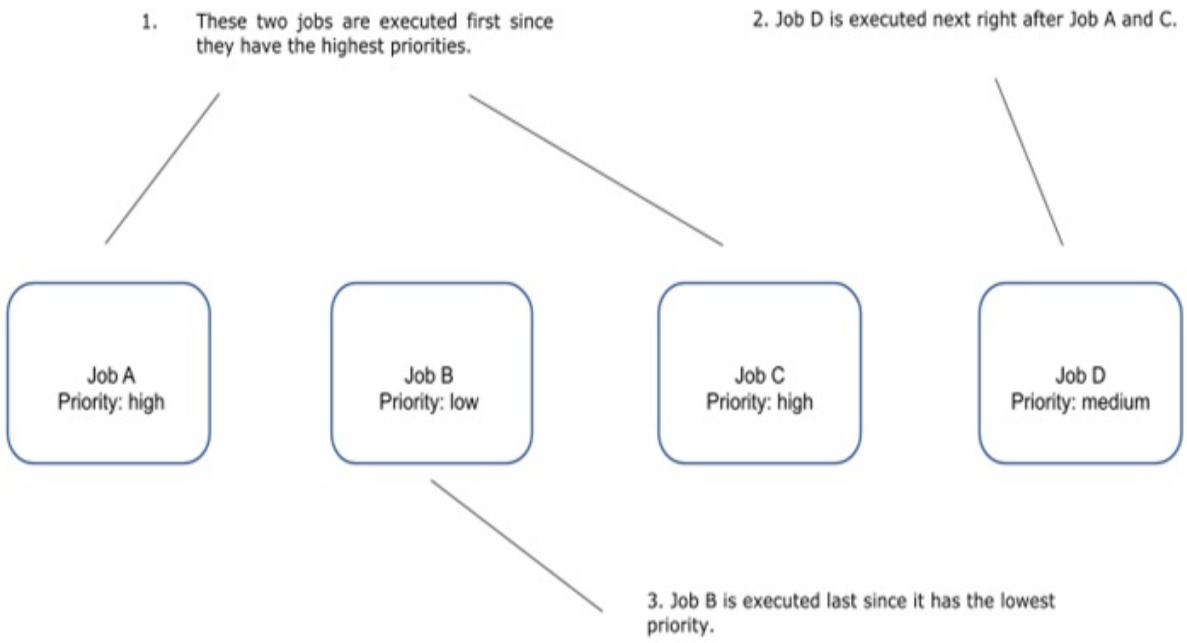


To fix this, we can allow users to assign *priorities* to each of the jobs so that jobs with higher priority are executed earlier, in contrast to the first-come first serve nature of the fair-share scheduling algorithm. In addition, the jobs that are already running can be *pre-empted* or *evicted* to make room for jobs with higher priorities if there are not enough computational resources available. This way of scheduling jobs based on priorities is called *priority scheduling*.

For example, there are four jobs A, B, C, and D that have been submitted concurrently. Each of these jobs have been marked with priorities by the users. Jobs A and C are both at high priority level whereas job B is at low priority and job D is at medium priority. With priority scheduling, jobs A and C will be executed first since they have the highest priorities, followed by the execution of Job D with medium priority and eventually Job B which is at low priority. Figure 6.7 illustrates the order of execution for the four jobs (A,

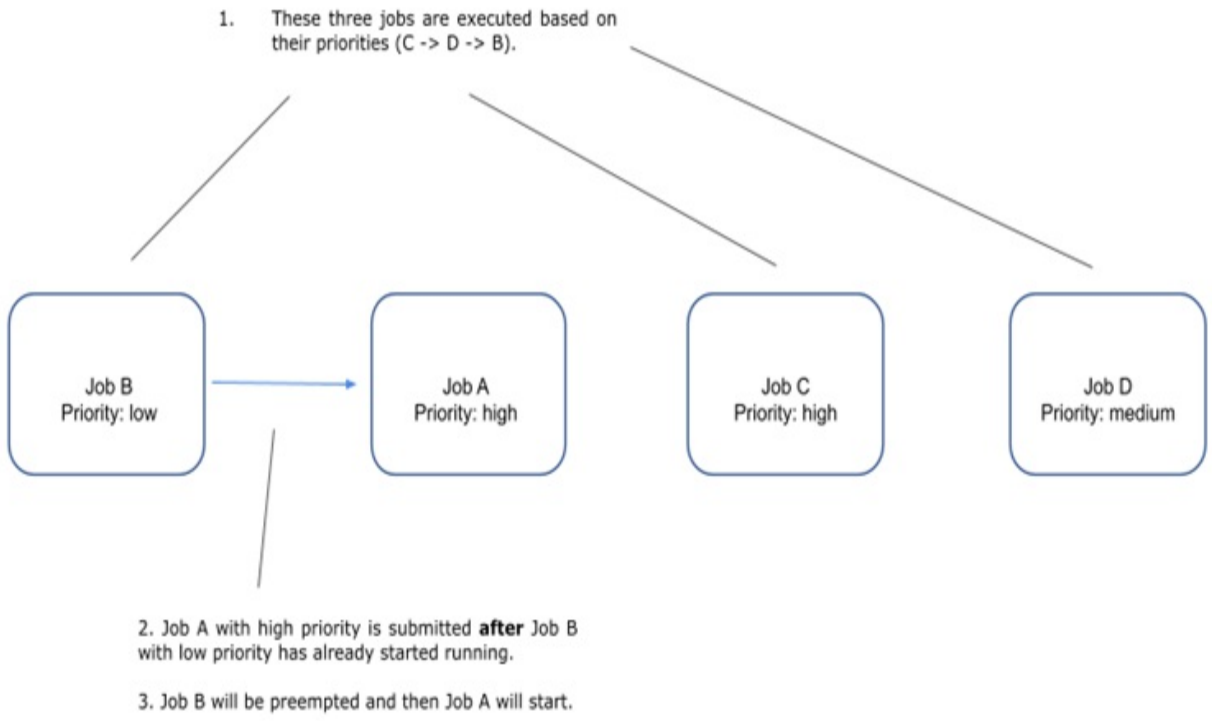
B, C, and D) when priority scheduling is used.

Figure 6.7 The order of execution for the four concurrently submitted jobs (A, B, C, and D) when priority scheduling is used.



Now let's consider another example. Assume there jobs (B, C, and D) with different priorities are submitted concurrently and they are executed based on their priorities similar to the previous example. Then if another Job A with high priority is submitted after Job B with low priority has already started running, Job B will be preempted and then Job A will start. The computational resources previously allocated for Job B will be released and taken over by Job A. Figure 6.8 summarizes the order of execution for the four jobs (A, B, C, and D) where the low priority Job B that's already running is preempted by a new job Job A with higher priority.

Figure 6.8 The order of execution for the four jobs (A, B, C, and D) where the running low priority job is preempted by a new job with higher priority.

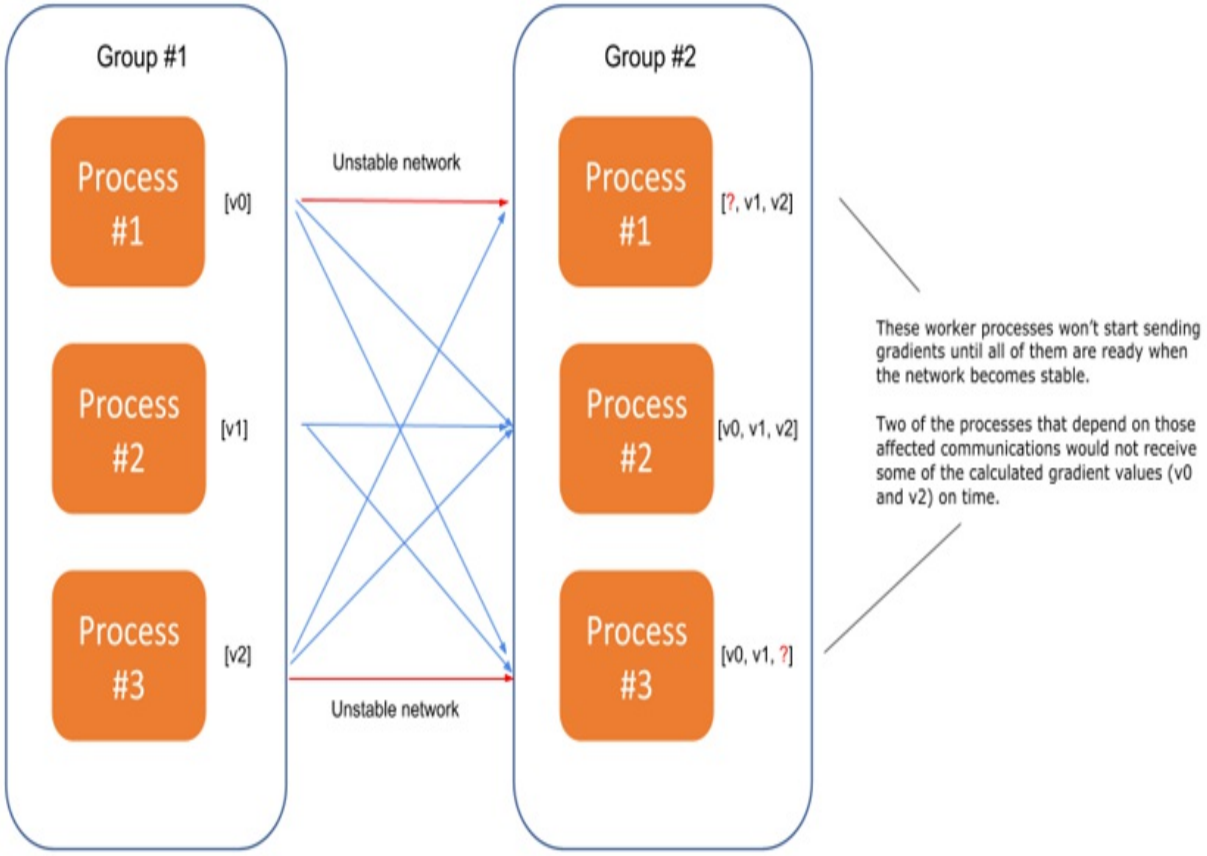


With priority scheduling, we can effectively eliminate the issue we encountered previously where jobs can only be executed sequentially on a first-come first serve basis and jobs can now be preempted in favor of tasks with high priorities.

However, for distributed machine learning tasks, model training tasks specifically, we would like to ensure that all workers are ready before starting distributed training. Otherwise, the ones that are ready would be waiting for the remaining workers before the training can proceed, which wastes resources.

For example, in Figure 6.9, we expect that there are three worker processes in the same process group performing an allreduce operation. However, two of the workers are not ready due to the unstable network the underlying distributed cluster is experiencing. As a result, two of the processes (Process #1 and Process #3) that depend on those affected communications would not receive some of the calculated gradient values (v0 and v2) on time (denoted by question marks in the diagram) and the entire allreduce operation is stuck until everything is received.

Figure 6.9 Example of allreduce process with unstable network among the worker processes that blocks the entire model training process.



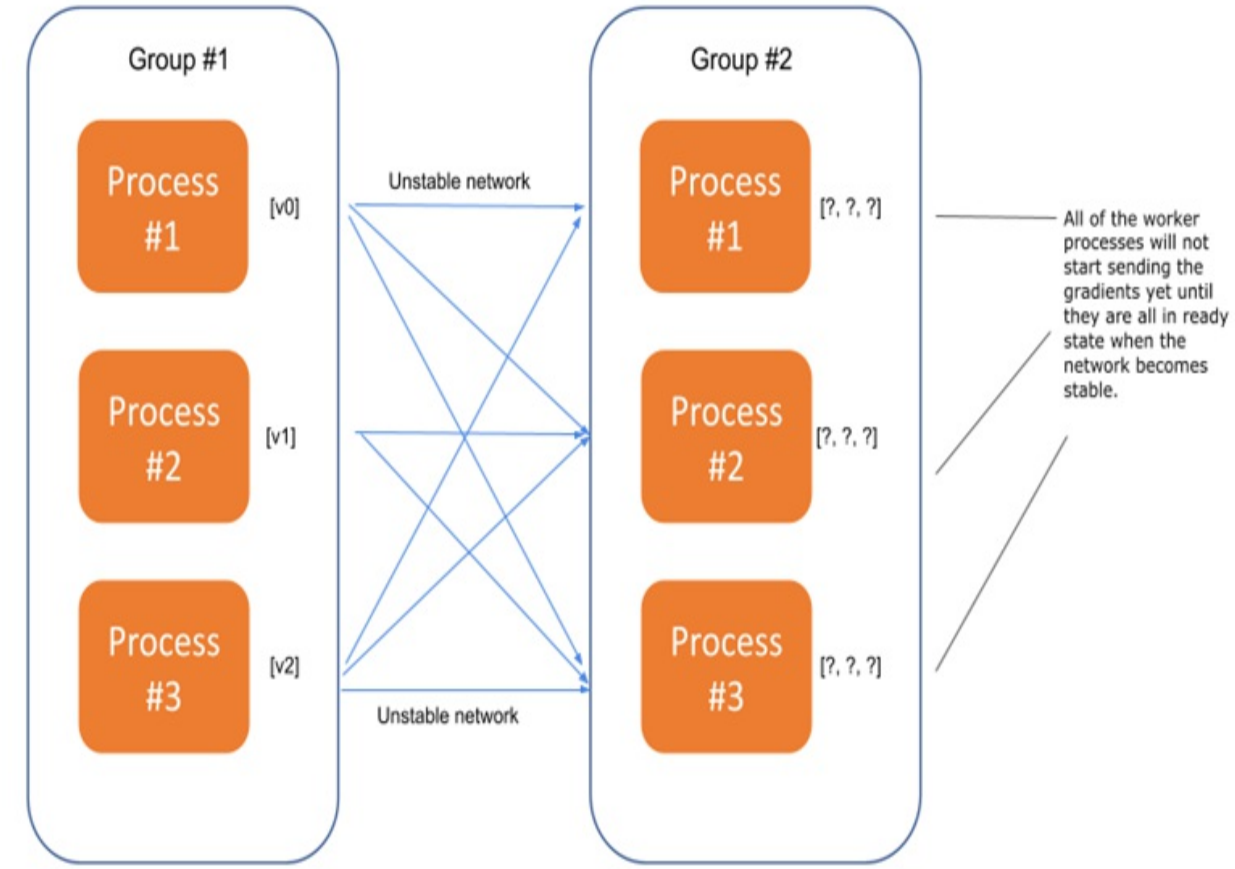
For running distributed model training tasks, *gang scheduling* is usually used to ensure that if two or more workers communicate with each other, they will all be ready to communicate at the same time. In other words, gang scheduling would only schedule workers when there are enough workers available that are ready to communicate.

If they were not gang-scheduled, then one could wait to send or receive a message to another while it is sleeping, and vice versa. When the workers are waiting for other workers to be ready for communication, we are essentially wasting the allocated resources on the workers that are ready and the entire distributed model training task is stuck.

For example, for collective communication based distributed model training tasks, all workers need to be ready to communicate the calculated gradients

and update the models on each worker in order to complete an allreduce operation. Note that here we assume that the machine learning framework does not support elastic scheduling yet which we will also discuss in the next section. As shown in Figure 6.10, the gradients are all denoted by question marks since they have not arrived in any of those worker processes in the second worker group yet. All of the worker processes have not started sending the gradients yet until they are all in ready state when the network becomes stable.

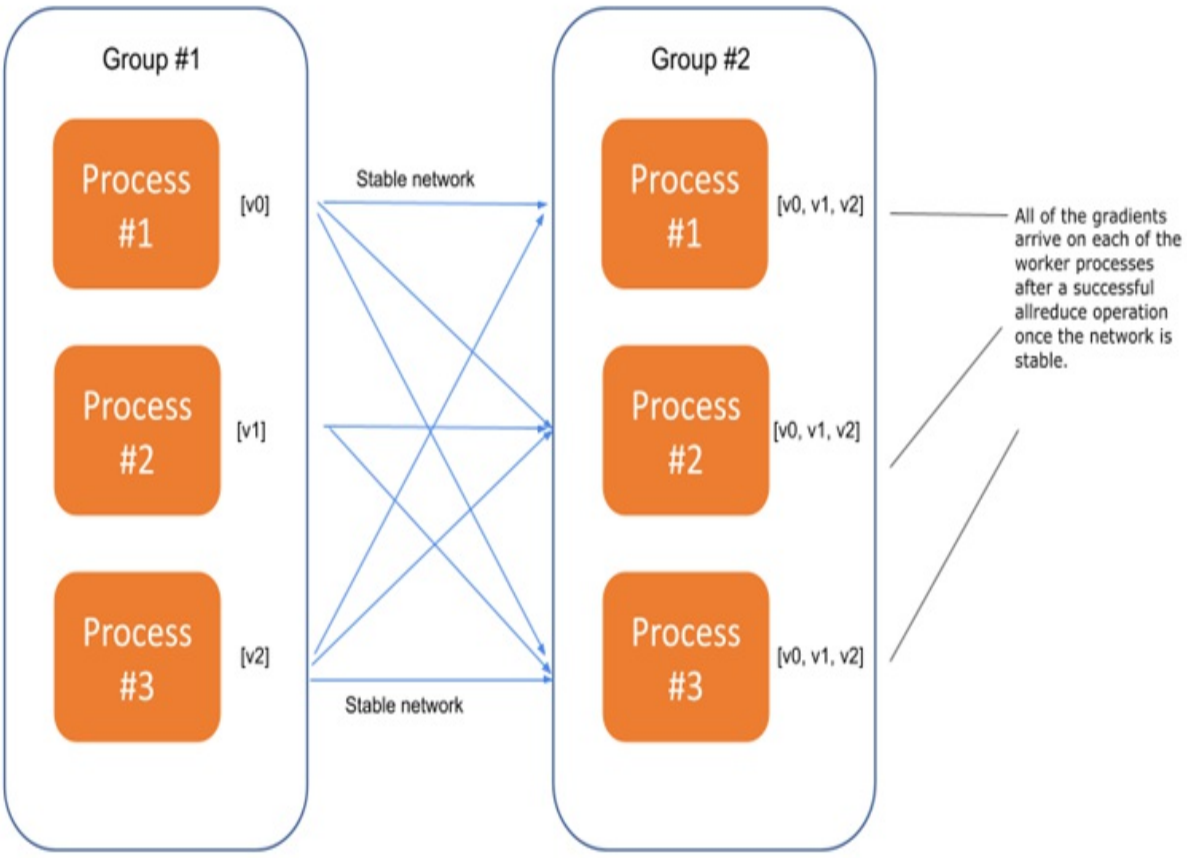
Figure 6.10 With gang scheduling, all of the worker processes will not start sending the gradients yet until they are all in ready state when the network becomes stable.



With gang scheduling, we can make sure not to start any of the worker processes yet until all of them are ready so none of them would be waiting for the remaining worker processes. As a result, we can avoid wasting computational resources. Once the network becomes stable, all of the gradients (v0, v1, and v2) would arrive on each of the worker processes after

a successful allreduce operation, as shown in Figure 6.11.

Figure 6.11 All of the gradients arrive on each of the worker processes after a successful allreduce operation once the network is stable.



Note that the details of different types of gang scheduling and their algorithms are out of scope of this book and will not be discussed here. However, we will be using an existing open source framework to integrate gang scheduling into distributed training in the last part of the book.

By incorporating different scheduling patterns, we are able to address various issues that arise when multiple users are using the infrastructure to schedule different types of jobs. Even though we mentioned the specific use cases of these scheduling patterns, they can be found in many systems that require careful management of computational resources, especially when there's resource scarcity. Many scheduling techniques are applied to even lower-level operating systems to make sure the applications are running efficiently

and share resources reasonably in the operation system.

6.2.3 Discussion

We've seen how the fair-share scheduling would help us resolve the issue when multiple users are running distributed training jobs concurrently and how it allows us to apply this scheduling strategy at each level of abstraction such as processes, users, groups, etc. We've also introduced priority scheduling that can be used to effectively eliminate the issue where jobs can only be executed sequentially on a first-come first serve basis by allowing jobs executing based on their priority levels and preempting low priority jobs to make room for jobs with high priorities.

It's worth noting that with priority scheduling, if a cluster is used by all kinds of users that may include users that are not trusted, any malicious user could create jobs at the highest possible priorities, causing other jobs to be evicted or not get scheduled at all. To deal with this, administrators of real-world clusters usually enforce certain rules and limits to prevent users from creating a huge number of jobs at high priorities.

We also discussed gang scheduling that ensures if two or more workers communicate with each other, they will all be ready to communicate at the same time. This is especially helpful for collective communication based distributed model training jobs where all workers need to be ready for communication of the calculated gradients to avoid wasting computational resources. Note that some machine learning frameworks do support elastic scheduling that we've introduced in Chapter 3, which allows distributed model training jobs to start with any number of workers available without waiting for all the requested workers to be ready for communication. In that case, gang scheduling is not suitable as we could've made significant progress towards model training with elastic scheduling instead of waiting for all workers to be ready without making any progress.

Although since the number of workers may change during model training, the batch size (sum of the size of mini-batches on each worker) will affect the model training accuracy. In that case, additional modifications on the model training strategy are needed. For example, we can support a customized

learning rate scheduler which will take epoch or batch size into account or adjust the batch size dynamically based on the number of workers being used. Together with these algorithmic improvements, we can allocate and utilize existing computational resources more wisely and improve the experience of the users.

In practice, distributed model training jobs benefit a lot from scheduling patterns like gang scheduling, which makes sure not to start any of the worker processes until all of them are ready so none of them would be waiting for the remaining worker processes. As a result, we can avoid wasting computational resources. One thing we might be neglecting is that any of these worker processes scheduled by gang scheduling may fail and thus lead to unexpected consequences. Often times it's hard to debug any of these types of failures. In the next section, we'll introduce a pattern that will make debugging and handling failures easy.

6.2.4 Exercises

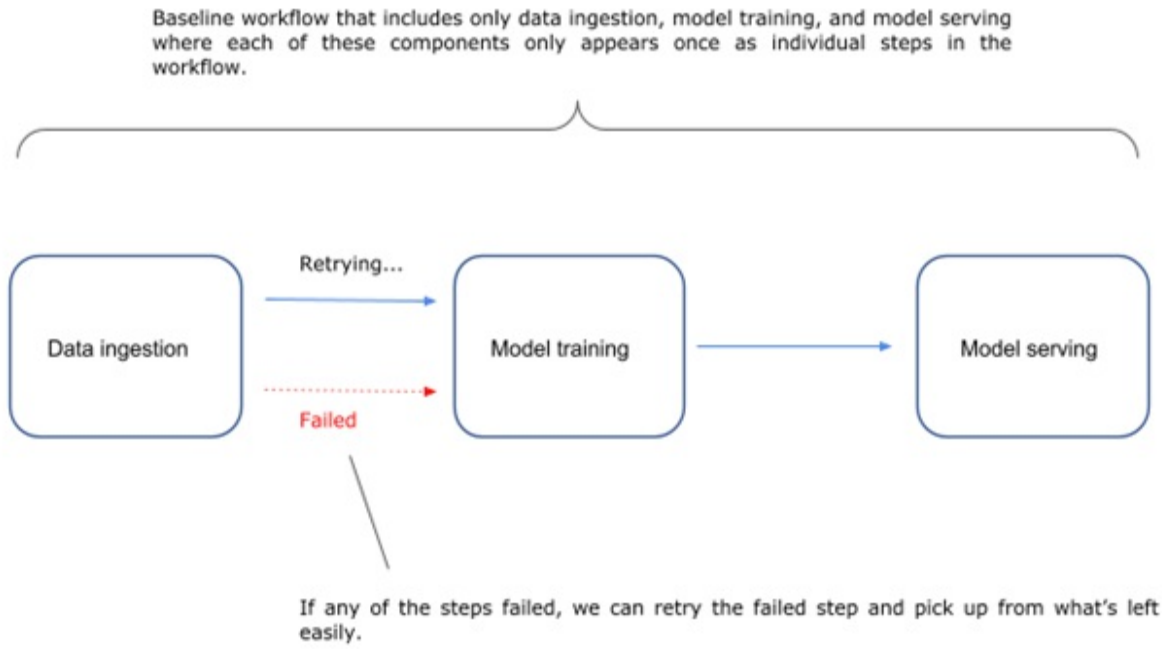
1. Is it correct that we can only apply fair-share scheduling at user level?
2. Is gang scheduling suitable for all distributed model training jobs?

6.3 Metadata pattern: Handle failures appropriately to minimize negative impact on users

When we are building the most basic machine learning workflow that includes only data ingestion, model training, and model serving where each of these components only appears once as individual steps in the workflow, everything seems pretty straightforward.

Each of these steps run sequentially to reach completion. If any of these steps fail, we pick up from where it's left. For example, if the model training step has failed to take the ingested data, e.g. lost the connection to the database where the ingested data is stored, we can retry the failed step and pick up from what's left easily to continue model training without having to re-run the entire data ingestion process, as shown in Figure 6.12.

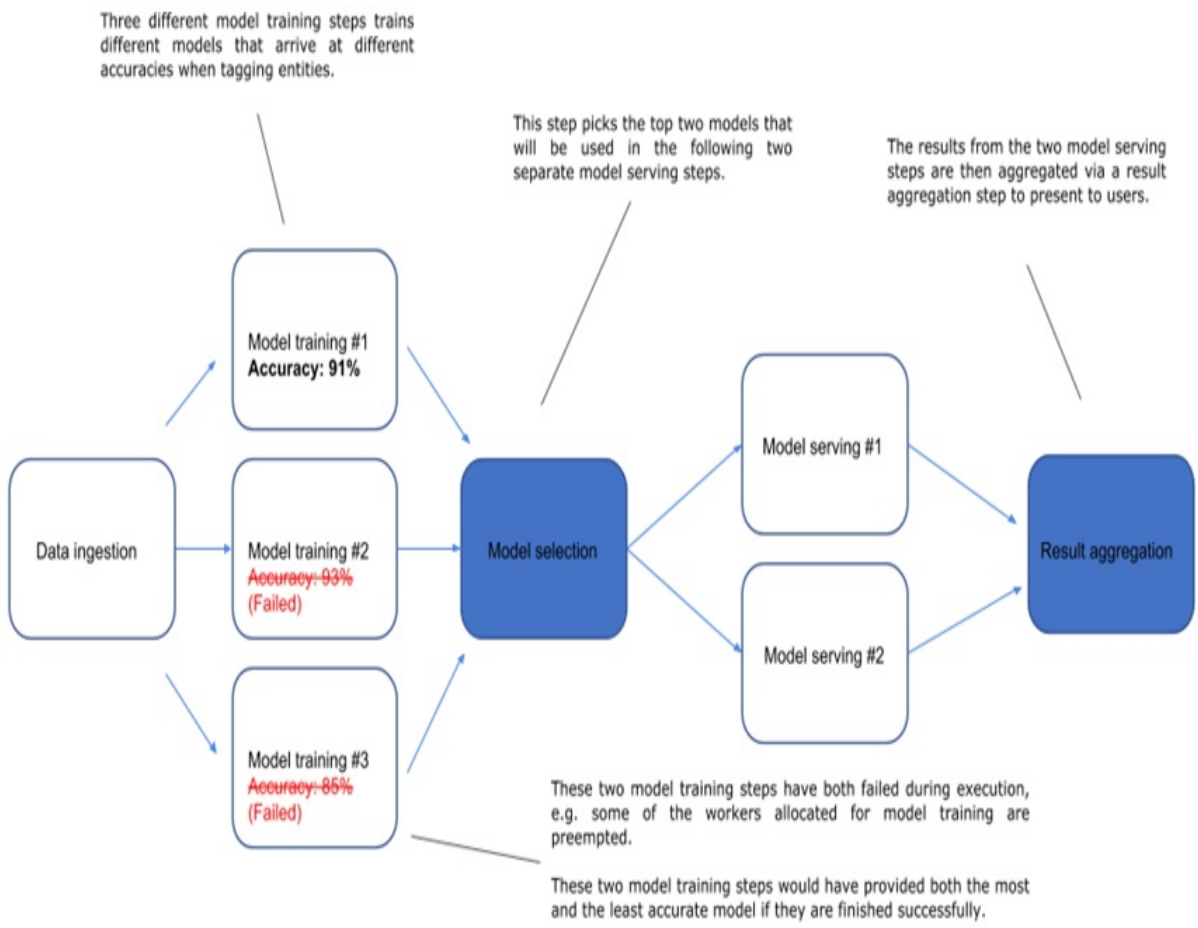
Figure 6.12 A baseline workflow where the model training step has failed to take the ingested data and we retry the failed step and pick up from what's left easily to continue model training without having to re-run the entire data ingestion process.



However, when the workflow gets more complicated, any failures are not trivial to handle. For example, let's take a look at the workflow that we've seen in the previous chapter. This workflow trains different models via three different model training steps that arrive at different accuracies when tagging entities, and then a model selection step picks the top two models with at least 90% accuracy trained from the first two model training steps that will be used in the following two separate model serving steps. The results from the two model serving steps are then aggregated via a result aggregation step to present to users.

Now let's consider the case where the second and the third model training steps have both failed during execution, e.g. some of the workers allocated for model training are preempted. These two model training steps would have provided both the most and the least accurate model if they are finished successfully, as shown in Figure 6.13. Note that the accuracies are crossed out in these two steps since the steps failed without arriving at the expected accuracies.

Figure 6.13 A machine learning workflow that trains different models that arrive at different accuracies when tagging entities and then selects the top two models with at least 90% accuracy to be used for model serving. Note that the accuracies are crossed out in these two steps since the steps failed without arriving at the expected accuracies. The results from the two model serving steps are then aggregated to present to users.



At this point, one might think that we should re-run both of the steps in order to proceed to the model selection and model serving steps. However, in practice, since we already wasted some time training part of the models, we may not want to start everything from scratch since it would take a much longer time before our users can see the aggregated results from our best models. Is there a better way to handle such kinds of failures?

6.3.1 Problem

For complicated machine learning workflows such as the one we've seen in

the previous chapter where we'd like to train multiple models and then select the top performing models for model serving, it's not always trivial to decide on a strategy to handle failures of certain steps due to real-world requirements.

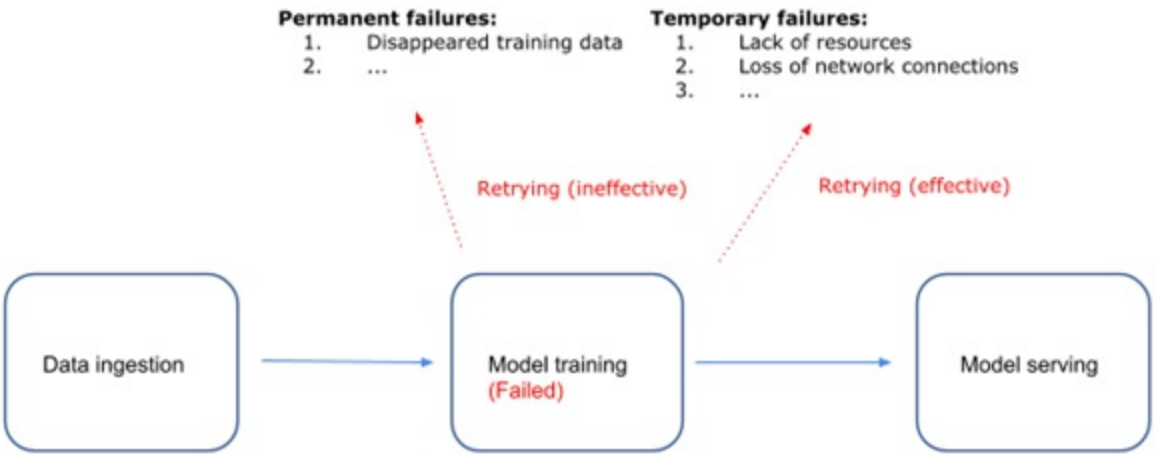
For example, when two out of three model training steps fail due to preempted workers, we cannot simply start training those models from scratch since it would take much longer before our users can see the aggregated results from our best models.

How do we handle failures like this appropriately so that our negative impact on users can be minimized?

6.3.2 Solution

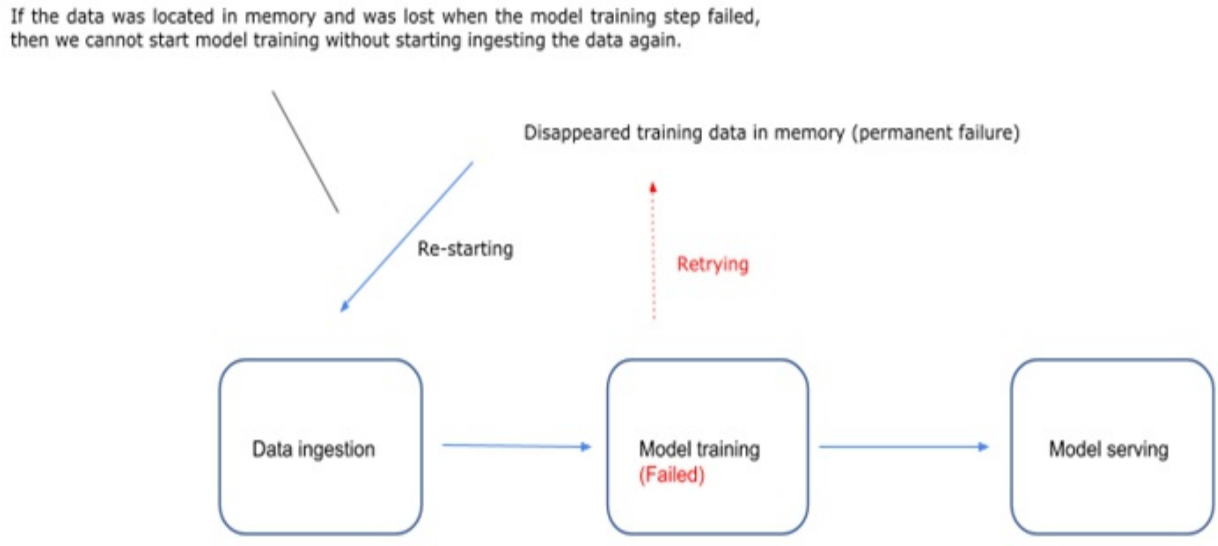
First of all, whenever we encounter failures in a machine learning workflow, we should first understand what the root cause is for the failures, e.g. loss of network connections, lack of computational resources, etc. This is important because we need to understand the nature of the failures well in order to predict whether retrying the failed steps would help at all. If the failures are due to some long-lasting shortages that could very likely lead to repetitive failures when retrying, then we could better utilize the computational resources to run some other tasks. Figure 6.14 illustrates the difference in effectiveness of retrying for permanent and temporary failures. When we retry the model training step when encountering permanent failures, the retries are ineffective and lead to repetitive failures.

Figure 6.14 The difference in effectiveness of retrying for permanent and temporary failures.



For example, in our case, we should first check whether the dependencies of a model training step are met, such as whether the ingested data from the previous step is still available. If the data has been persisted to a local disk to a database, then we can proceed to model training. Otherwise if the data was located in memory and was lost when the model training step failed, then we cannot start model training without ingesting the data again. Figure 6.15 illustrates the process of restarting the data ingestion step when there's permanent failure during model training.

Figure 6.15 The process of restarting the data ingestion step when there's permanent failure during model training.

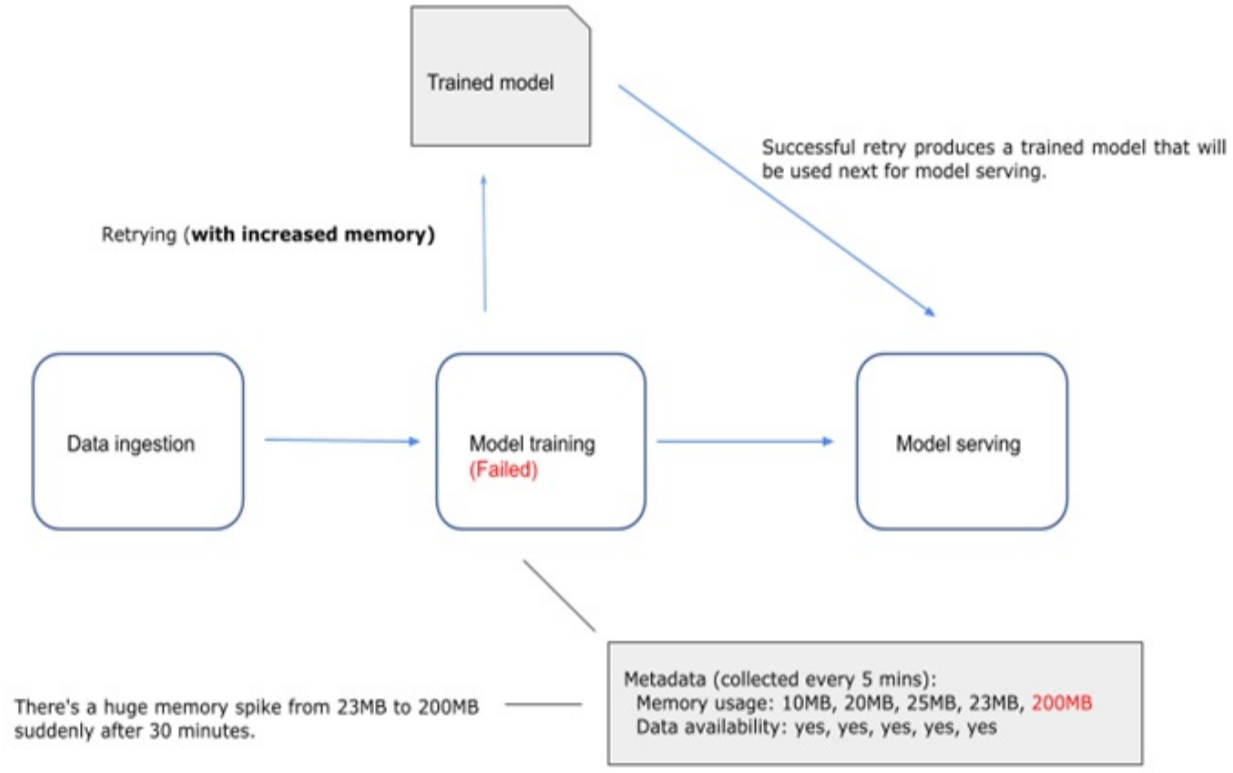


Similarly, if the model training step failed due to preempted training workers or out-of-memory issues, then we need to make sure we still have sufficient computational resources that can be allocated to re-run the model training step.

However, we won’t know that information to analyze the root cause unless we intentionally record it as metadata during the runtime of each of the steps in the entire machine learning workflow. For example, for each of the model training steps, we could record metadata on the availability of the ingested data and whether different computational resources such as memory and CPU usage exceed the limit before the step fails.

Figure 6.16 is a workflow where the model training step failed. Unfortunately there’s metadata collected every 5 minutes regarding the memory usage (in megabytes) and the availability of the training data (yes/no) during the runtime of this step. We can notice that there's a huge memory spike from 23MB to 200MB suddenly after 30 minutes. In this case, we can retry this step with an increased requested memory and this step would then successfully produce a trained model that will be used for the next model serving step.

Figure 6.16 An example workflow where the model training step failed with the metadata collected showing an unexpected memory spike during runtime.



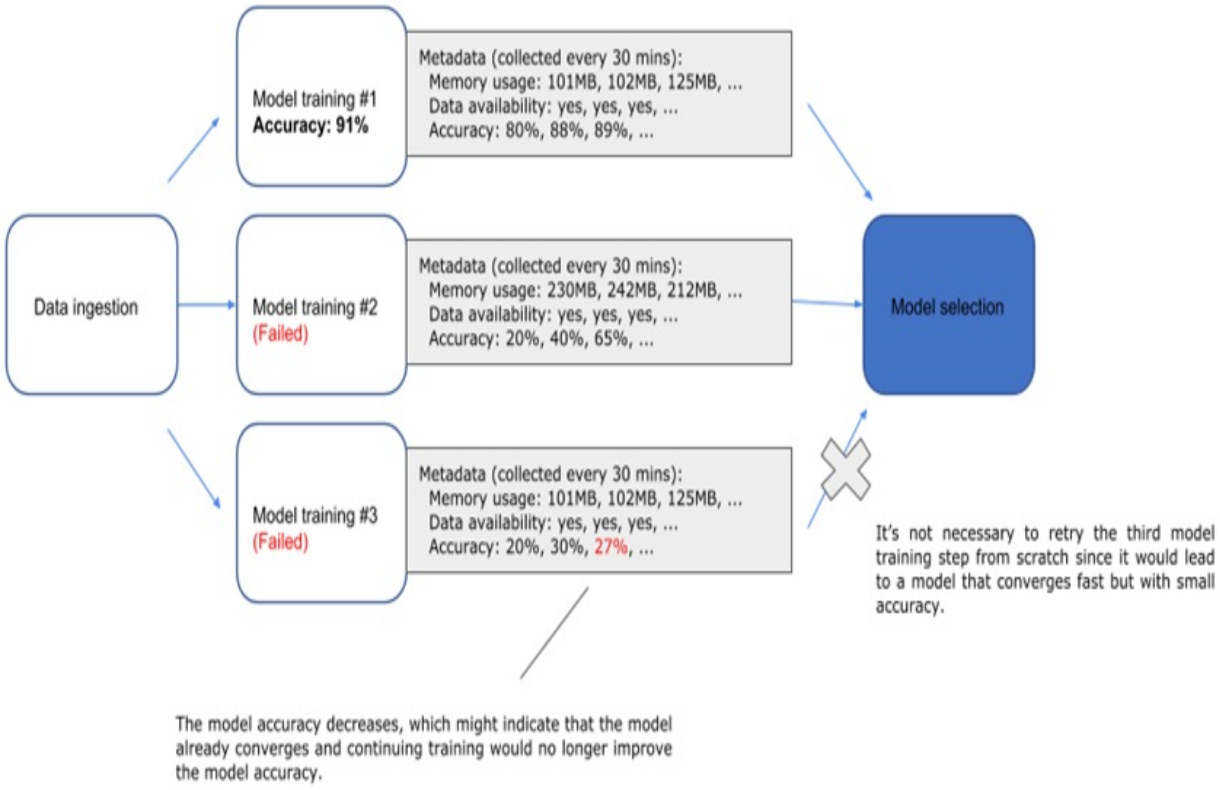
In practice, for complex workflows like in Figure 6.13, even when we know all the dependencies of model training steps are met, e.g. we have enough computational resources and good database connection to access the data source, we should also think about whether we want to handle the failures and how we'd like to handle them.

Recall that we've already spent a huge amount of time on the training steps already and suddenly the steps failed and we lost all the progress. In other words, we don't want to start re-training all the models from scratch since it would take quite some time before our users can see the aggregated results from our best models. Is there a better way to handle this without a huge impact on our user experience?

In addition to the metadata we've recorded for each of the model training steps, we could save more useful metadata that can be used to figure out whether it's worth re-running all the model training steps. For example, the model accuracy overtime indicates whether the model is being trained effectively. If the model accuracy remains steady or even decreases (from

30% to 27%), as shown in Figure 6.17, this might indicate that the model already converges and continuing training would no longer improve the model accuracy. In this example, even though there are two failed model training steps, it's not necessary to retry the third model training step from scratch since it would lead to a model that converges fast but with small accuracy. Another example of metadata that can be potentially useful is the percentage towards model training completion, e.g. if we've iterated through all the requested number of batches and epochs then it's 100% completion.

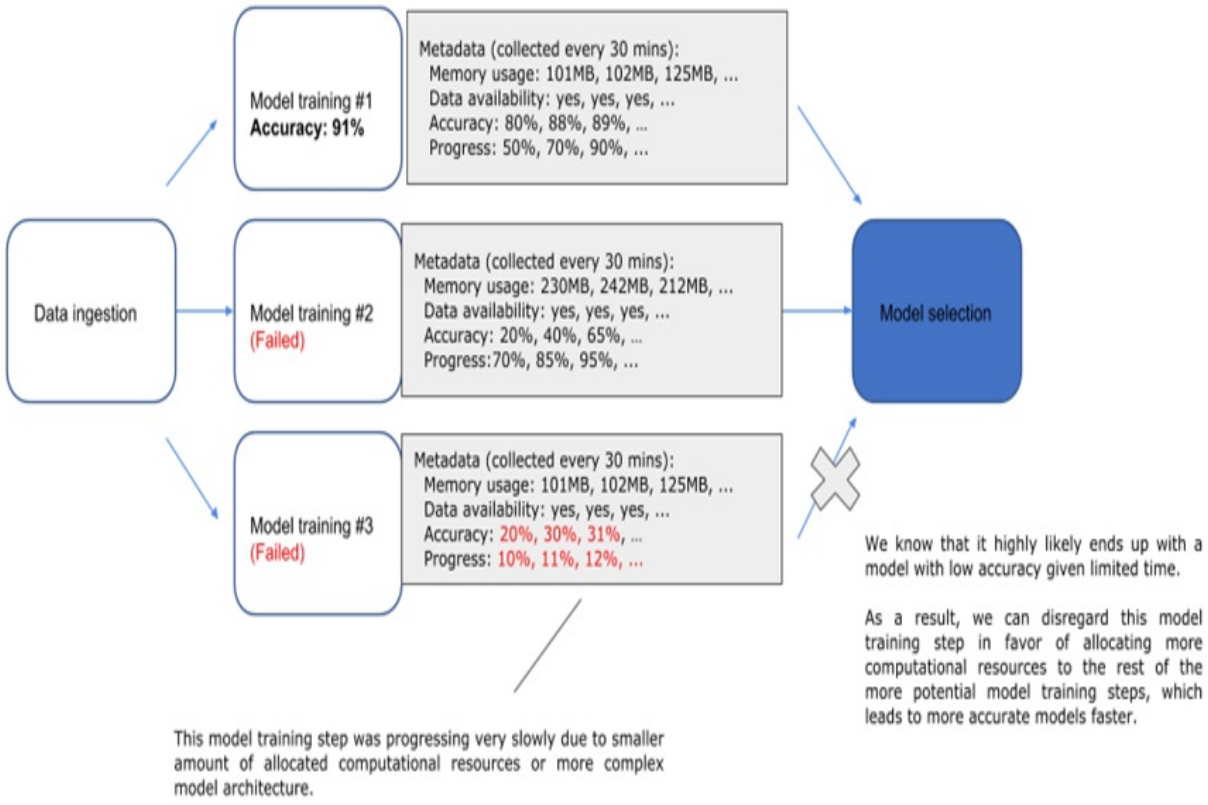
Figure 6.17 An example workflow where two model training steps fail and one of them has decreasing model accuracy.



Once we have these additional metadata on model training steps, we can tell how well each of the started model training steps progress. For example, for the workflow in Figure 6.18, we could potentially conclude ahead of time that the third model training step was progressing very slowly (only 1% of completion every 30 minutes) due to smaller amount of allocated computational resources or more complex model architecture and we know

that it highly likely ends up with a model with low accuracy given limited time. As a result, we can disregard this model training step in favor of allocating more computational resources to the rest of the more potential model training steps, which leads to more accurate models faster.

Figure 6.18 An example workflow where two model training steps fail and one of them is disregarded since it progresses very slowly and is highly likely to end up with a model with low accuracy given limited time.



Recording these metadata could help us derive more insights specific to each of the failed steps in the end-to-end machine learning workflow. We could then decide on a strategy to handle failed steps appropriately to avoid wasting computational resources as well as minimizing the impact on existing users. The metadata patterns provide great visibility into our machine-learning pipelines. They can also be used to search, filter, and analyze the artifacts produced in each step in the future if we are running a lot of pipelines on a regular basis. For example, we might want to know which models are performant or which datasets contribute the most to those models based on

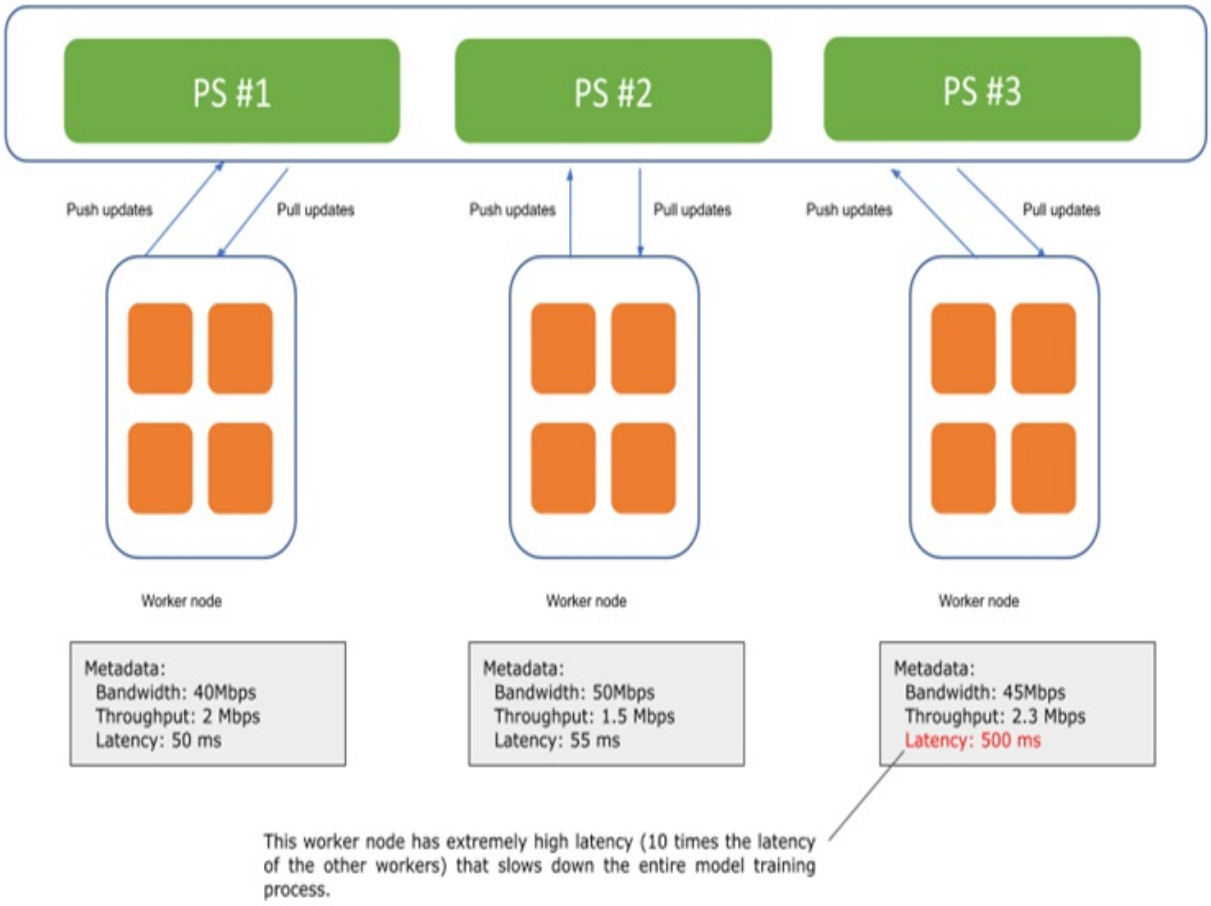
the historical training metrics.

6.3.3 Discussion

With the help of the metadata pattern, we can gain additional insights regarding individual steps in machine learning workflows so that when any of them fail, we can respond based on what's beneficial to our users and thus reduce any negative impact due to the step failures.

So far we've looked at failures at step level. If we zoom in a single model training step, assuming we are leveraging the parameter server pattern where workers communicate with the parameter servers to update the calculated gradients. One common type of metadata is the various network performance metrics while the model is being trained, e.g. bandwidth, throughput, and latency. This type of information is very useful when we want to detect when certain workers experience poor network performance that block the entire training process. When there are slow workers, we can take them down and then start new workers to continue training, assuming the underlying machine learning frameworks support elastic scheduling and fault tolerance that we introduced in Chapter 3. For example in Figure 6.19, based on the metadata, the worker on the right-hand side has extremely high latency (10 times the latency of the other workers) that slows down the entire model training process. Ideally this worker would be taken down and restarted to avoid slowing down the training process.

Figure 6.19 An example parameter server based model training where the worker on the right-hand side has extremely high latency (10 times the latency of the other workers) that slows down the entire model training process.



One additional benefit of introducing the metadata pattern to our machine learning workflows is to leverage the metadata recorded to establish relationships among the individual steps or across different workflows. For example, modern model management tools could leverage the recorded metadata to help users build the lineage of the trained models and visualize what individual steps/factors have contributed to the model artifacts.

6.3.4 Exercises

1. If the training step failed due to the loss of training data source, what should we do?
2. What type of metadata can be collected if we look at individual workers or parameter servers?

6.4 References

1. CPU cycle: https://techterms.com/definition/clockcycle
2. Network performance: https://en.wikipedia.org/wiki/Network_performance

6.5 Summary

- There are different areas of improvements related to operations in machine learning systems, such as job scheduling and metadata.
- Various scheduling patterns, such as fair-share scheduling, priority scheduling, and gang scheduling, can be used to prevent resource starvation and avoid deadlocks.
- We can collect metadata to gain insights from machine learning workflows and handle failures more appropriately to reduce negative impact on users.

Answers to Exercises:

Section 6.2

1. No, we can apply this scheduling strategy at each level of abstraction such as processes, users, groups, etc.
2. No, some machine learning frameworks support elastic scheduling, which allows distributed model training jobs to start with any number of workers available without waiting for all the requested workers to be ready for communication. In this case, gang scheduling is not suitable.

Section 6.3

1. We should re-run data ingestion before retrying the model training step since this is a permanent failure and simply retrying would lead to repetitive failures.
2. Various network performance metrics while the model is being trained, e.g. bandwidth, throughput, and latency. This type of information is very useful when we want to detect when certain workers experience poor network performance that block the entire training process.

7 Project overview and system architecture

This chapter covers

- Getting familiar with our project background and providing a high-level overall design of our system.
- Optimizing the data ingestion component for multiple epochs of the dataset.
- Deciding on the distributed model training strategy that minimizes the overhead.
- Adding model server replicas for high-performance model serving.
- Accelerating the end-to-end workflow of our machine learning system.

In the previous chapters of the book, we learned to choose and apply the correct patterns for building and deploying distributed machine learning systems to handle large scale and gain practical experience in managing and automating machine learning tasks. In Chapter 2, we introduced a couple of practical patterns that can be incorporated into the data ingestion process, which is usually the beginning process of a distributed machine learning system that’s responsible for monitoring any incoming data and performing necessary preprocessing steps to prepare for model training. In Chapter 3, We’ve explored some of the challenges involved in the distributed training component and introduced a couple of practical patterns that can be incorporated into the component, which is the most critical part of a distributed machine learning system and is what makes the system unique from general distributed systems. We also covered the challenges involved in distributed model serving systems and introduced a few established patterns adopted heavily in industries in Chapter 4 to overcome the challenges to achieve horizontal scaling with the help of replicated services and use the sharded services pattern to help the system process large model serving requests, and learned how to assess model serving systems and determine whether the event-driven design would be beneficial with real-world scenarios. In Chapter 5, we discussed machine learning workflows, one of the

most essential components in machine learning systems as it connects all other components in a machine learning system. Last but not latest, in Chapter 6, we discussed some operational efforts and patterns that can be leveraged to greatly accelerate the end-to-end workflow and reduce maintenance and communication efforts when engineering teams are collaborating with teams of data scientists or machine learning practitioners before the systems become production ready.

For the remaining chapters of the book, we will build an end-to-end machine learning system to apply what we learned previously. We will gain hands-on experience implementing many patterns previously learned in this project. We'll learn how to solve problems at a larger scale and take what's developed on our laptop to large distributed clusters. In this chapter, we'll go through the project background and system components. Then we'll go through the challenges in each of these components and share the patterns that we will apply to address them.

Note that we won't be diving into the implementation details in this chapter and we'll use several popular frameworks and cutting-edge technologies, particularly TensorFlow, Kubernetes, Kubeflow, Docker, and Argo Workflows, to build different components of a distributed machine learning workflow in the remaining chapters.

7.1 Project overview

For this project, we will build an image classification system that takes raw images downloaded from the data source, performs necessary data cleaning steps, builds a machine learning model in a distributed Kubernetes cluster, and then deploys the trained model to the model serving system for users to use. We also want to establish an end-to-end workflow that is efficient and reusable. Next, we will introduce the project background and the overall system architecture and components.

7.1.1 Project background

We will build an end-to-end machine learning system to apply what we learned previously. We'll build our data ingestion component that downloads

the Fashion-MNIST dataset and the model training component to train and optimize the image classification model. Once the final model is trained, we'll build a high-performance model serving system to start make predictions using the trained model.

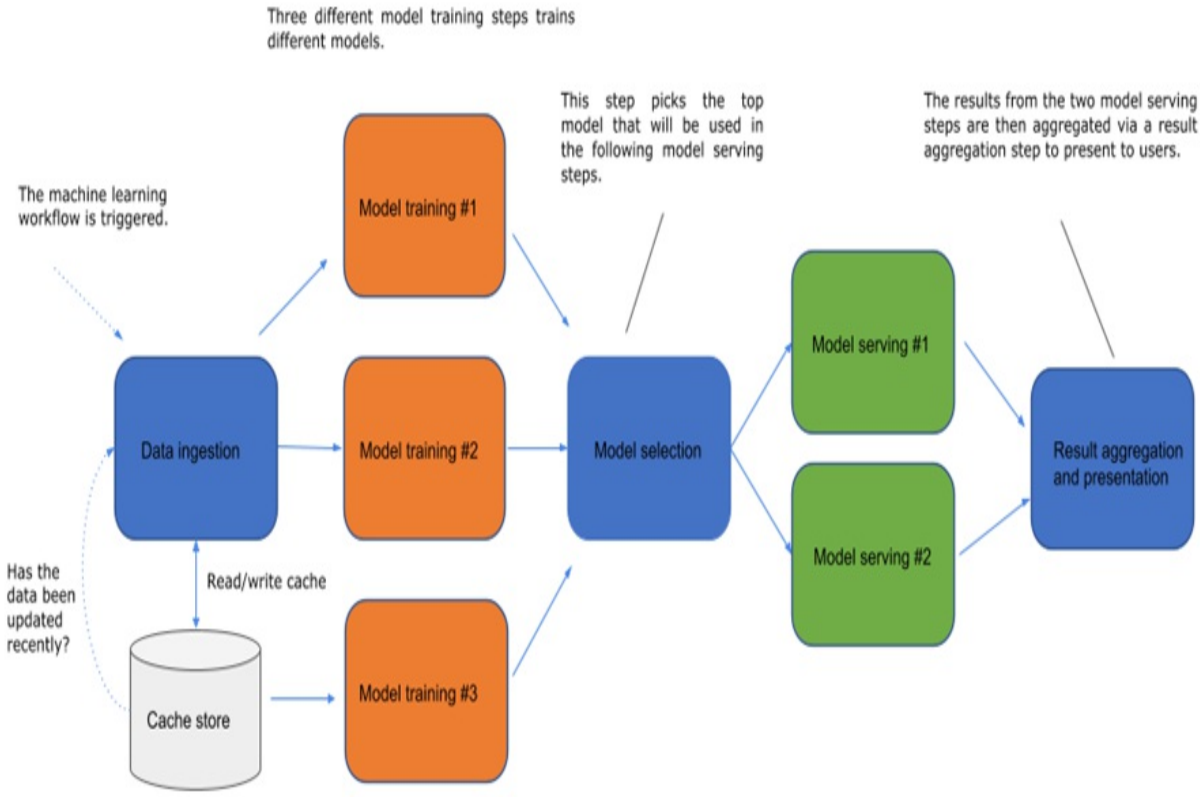
Several popular frameworks and cutting-edge technologies, particularly TensorFlow, Kubernetes, Kubeflow, Docker, and Argo Workflows, will be used to build different components of a distributed machine learning workflow. For example, we'll be using TensorFlow with Python to build the classification model on the Fashion-MNIST dataset, and making predictions. We'll use Kubeflow to run distributed machine learning model training on a Kubernetes cluster. Furthermore, Argo Workflows will be leveraged to build a machine learning pipeline that consists of many important components of a distributed machine learning system. The basics of these technologies will be introduced in the next chapter, and we'll gain hands-on experience on them before diving into the actual implementation of the project in the last chapter of the book.

In the next section, we'll dive into the system components of the project.

7.1.2 System components

Figure 7.1 is the architecture diagram of the system that we will be building in the remaining chapters of the book. First, we will build the data ingestion component responsible for ingesting data and storing the dataset in the cache, leveraging some of the patterns we learned in Chapter 2. Next, we will build three different model training steps that train different models and incorporate the collective communication pattern that we learned in Chapter 3. Once we finish the model training steps, we build the model selection step that picks the top model. The selected optimal model will be used for model serving in the following two steps. At the end of the model serving steps, we aggregate the predictions and present the result to users. Last but not least, we want to make sure all these steps are part of a reproducible workflow that can be executed at any time in any environment.

Figure 7.1 Architecture diagram of the end-to-end machine learning system we will be building.

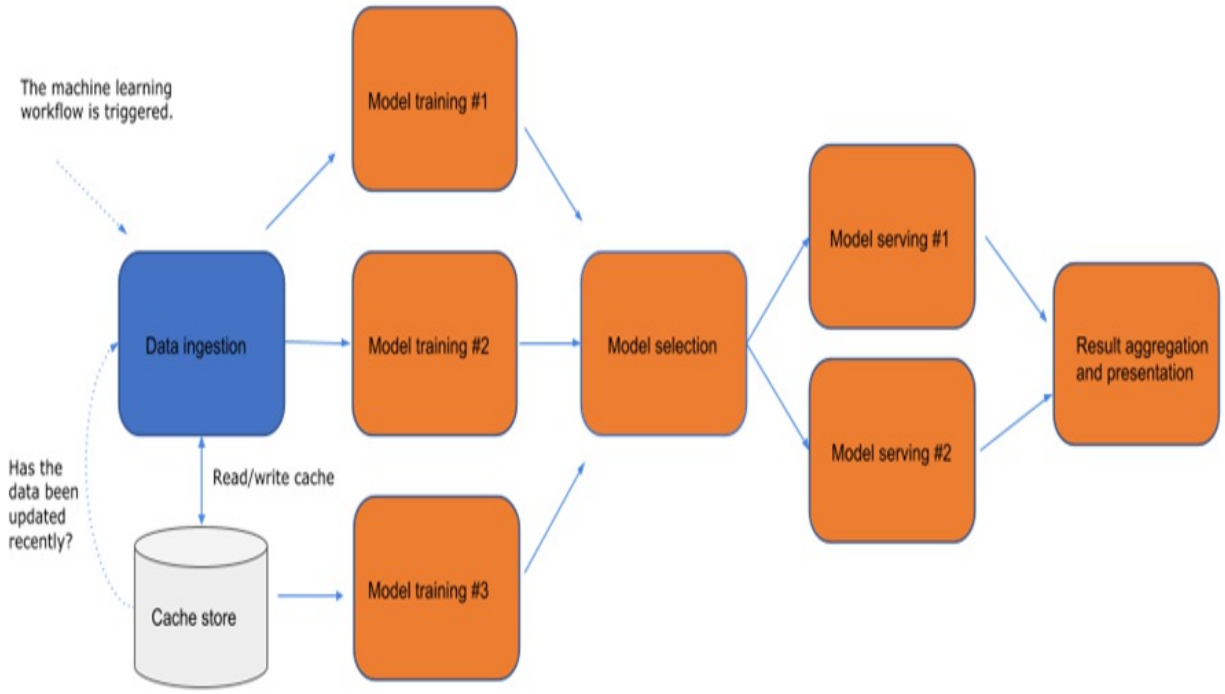


We'll build the system based on the architecture diagram in Figure 7.1 and dive into the details of the individual components. We'll also discuss the patterns we can use to address the challenges in building those components.

7.2 Data ingestion

In this project, we will use the Fashion-MNIST dataset that we introduced in Section 2.2.1 to build the data ingestion component as shown in Figure 7.2. This dataset consists of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image that represents one Zalando's article image, associated with a label from 10 classes. Recall that the Fashion-MNIST dataset is designed to serve as a direct drop-in replacement for the original MNIST dataset for benchmarking machine learning algorithms. It shares the same image size and structure of training and testing splits.

Figure 7.2 Data ingestion component (dark box) in the end-to-end machine learning system.



As a recap, Figure 7.3 is a screenshot of the collection of the images for all 10 classes (t-shirt/top, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag, and ankle boot) from Fashion-MNIST where each class takes three rows in the screenshot.

Figure 7.3 Screenshot of the collection of images from Fashion-MNIST dataset for all 10 classes (t-shirt/top, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag, and ankle boot). Source: https://github.com/zalandoresearch/fashion-mnist.

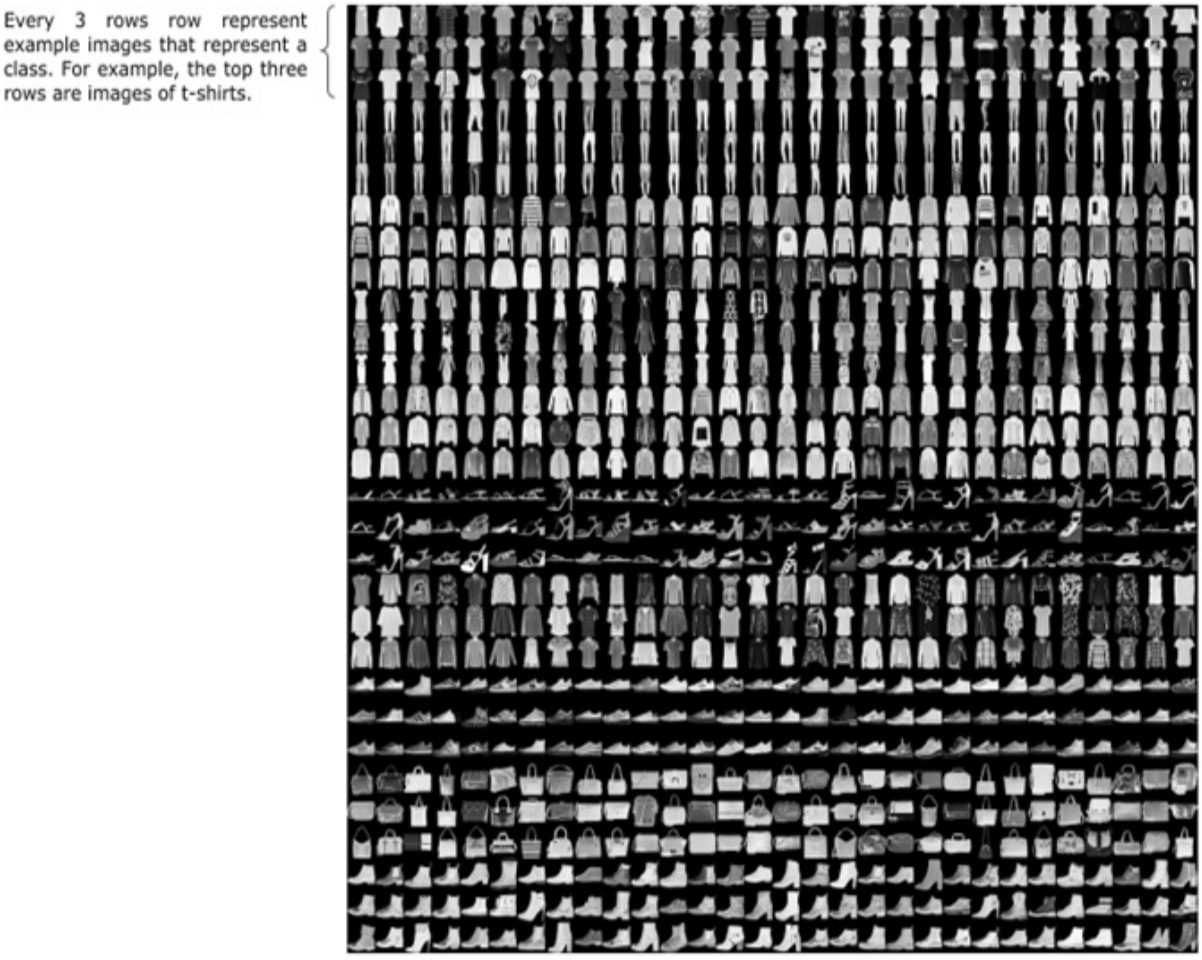


Figure 7.4 is a closer look at the first few example images in the training set together with their corresponding labels in text.

Figure 7.4 A closer look at the first few example images. In the training set with their corresponding labels in text. Source: https://d2l.ai/chapter_linear-networks/image-classification-dataset.html.

The downloaded Fashion-MNIST dataset should only take 30MBs on disk if it's compressed. It's easy to load all the downloaded dataset into memory at once.

7.2.1 Problem

Although the Fashion-MNIST data is not large, we may want to perform additional computations prior to feeding the dataset into the model, which is often common for tasks that often require additional transformations and cleaning. We may want to resize, normalize, convert the images to grayscale, or even perform complex mathematical operations such as convolution operations. These operations may require a lot of additional memory space allocations while there may or may not be sufficient computational resources available after we load the entire dataset in memory, depending on the distributed cluster size.

In addition, the machine learning model we are training from this dataset requires multiple epochs on the training dataset. If training one epoch on the entire training dataset takes 3 hours, then we would need to double the time spent on model training if we would like to train two epochs, as shown in Figure 7.5.

Figure 7.5 Diagram of model training for multiple epochs at time t0, t1, etc. where we spent 3 hours for each one of the epochs.

In real-world machine learning systems, an even larger number of epochs is often needed, and training each epoch sequentially is inefficient. In the next section, we will discuss how we can tackle that inefficiency.

7.2.2 Solution

Let’s take a look at the first challenge we have: the mathematical operations in the machine learning algorithms may require a lot of additional memory space allocations while there may or may not be sufficient computational resources.

Given that we don’t have too much free memory, we should not load the entire Fashion-MNIST dataset into memory directly. Let’s assume that the mathematical operations that we would like to perform on the dataset can be performed on subsets of the entire dataset. Then we could leverage the batching pattern that we introduced in Chapter 2, which could group a number of data records from the entire dataset into batches that will be used to train the machine learning model sequentially on each batch.

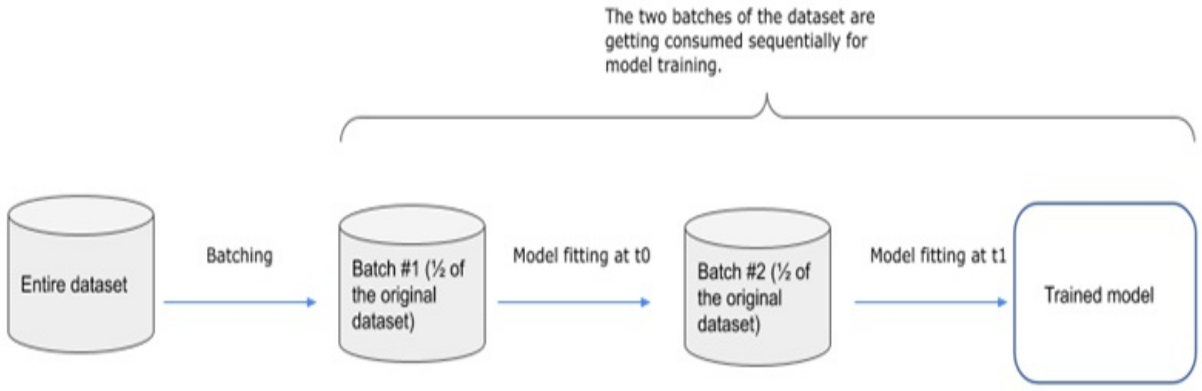
To apply the batching pattern, we first divide the dataset into smaller subsets or mini-batches, load each individual mini-batch of example images, perform expensive mathematical operations on each batch, and then use only one mini-batch of images in each model training iteration. For example, we can perform convolution or other heavy mathematical operations on the first minibatch which consists only 20 images and then send the transformed images to the machine learning model for model training. We then repeat the same process for the remaining mini-batches while continuing performing

model training in the meantime.

Note that since we've divided the dataset into many small subsets or mini-batches, we can avoid any potential issues with out-of-memory when performing various heavy mathematical operations on the entire dataset necessary for achieving an accurate classification model on the Fashion-MNIST dataset. We can then handle even larger datasets using this approach by reducing the size of the mini-batches.

With the help of the batching pattern, we are no longer concerned about potential out-of-memory issues when ingesting dataset for model training. We don't have to load the entire dataset into memory at once and instead we are consuming the dataset batch by batch sequentially. For example, if we have a dataset with a total 1000 records, we can first take 500 of the 1000 records to form a batch and then train the model using this batch of records. Subsequently, we repeat this batching and model training process again for the remaining records. In other words, we make two batches in total where each batch consists of 500 records, and the model we train consumes the batches one by one. Figure 7.6 illustrates this process where the original dataset gets divided into two batches sequentially. The first batch gets consumed to train the model at time t0 and the second batch gets consumed at time t1.

Figure 7.6 Dataset gets divided into two batches sequentially. The first batch gets consumed to train the model at time t0, and the second batch gets consumed at time t1.



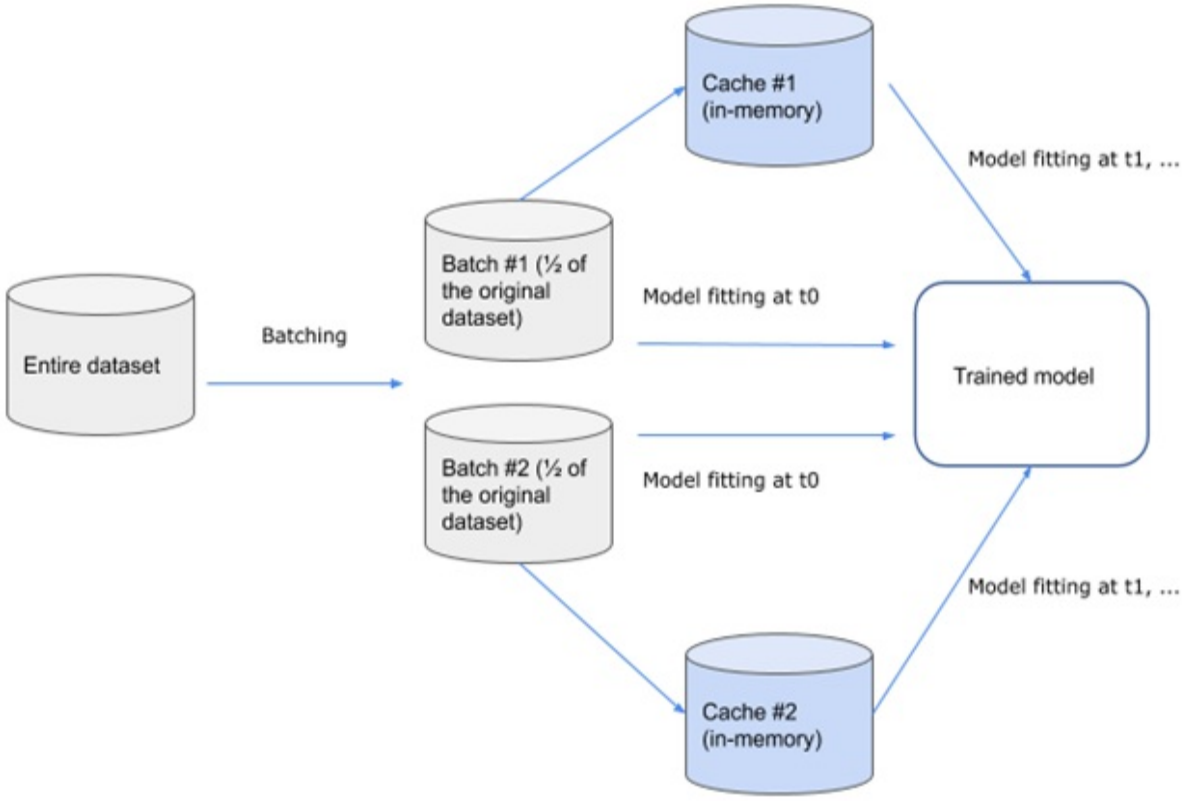
Now let's tackle the second challenge mentioned in Section 7.2.1: we would

waste a lot of time if we ever need to train a machine learning model that involves iterating on multiple epochs of the original dataset.

Recall that in Chapter 2, we've talked about the caching pattern, which would solve this type of problem. With the help of the caching pattern, we can greatly speed up the re-access to the dataset for the model training process that involves training on the same dataset for multiple epochs.

There's nothing special we can do for the first epoch since it's the first time the machine learning model has seen the entire set of the training dataset. We can store the cache of the training examples in the form of in-memory and then it can be much faster to re-access when it comes to the second and subsequent epochs. Let's assume that the single laptop we are using to train the model has sufficient computational resources such as memory and disk space. As soon as the machine learning model consumes each training example from the entire dataset, we can actually hold off recycling and instead keep the consumed training examples in memory. For example, in Figure 7.7, after we have finished fitting the model for the first epoch, we store a cache for both of the two batches that we used for the first epoch of model training.

Figure 7.7 Diagram of model training for multiple epochs at time t0, t1, etc., with cache without having to read from the data source repeatedly.



Then we can start training the model for the second epoch by feeding the stored in-memory cache to the model directly without having to read from the data source over and over again for future epoches. Next, we will discuss the model training component that we will be building in our project.

7.2.3 Exercises

1. Where do we store the cache?
2. Can we use the batching pattern when the Fashion-MNIST dataset gets large?

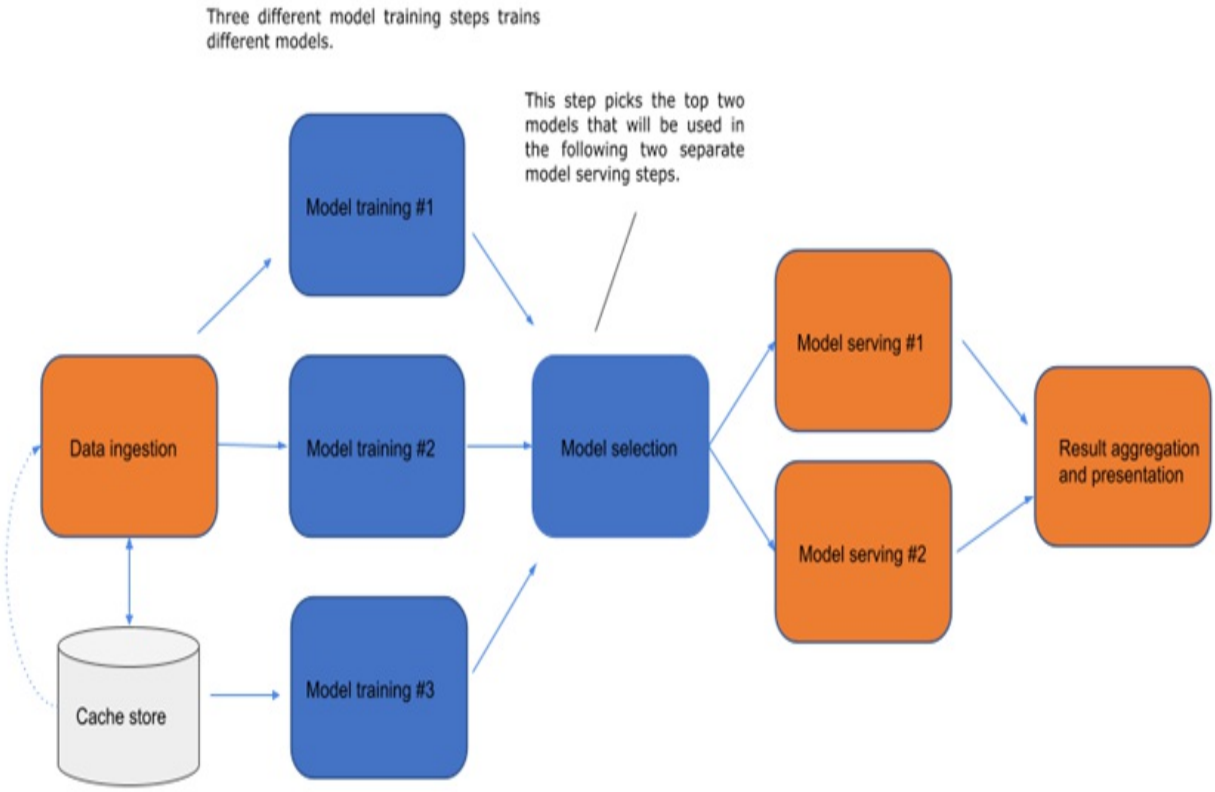
7.3 Model training

In the previous section, we've talked about the data ingestion component of the system we are building and how we could leverage the caching and batching pattern to handle large datasets as well as making the system more efficient. Next let's discuss the model training component that we are

building. Figure 7.8 is a diagram for the model training component in the overall architecture.

Note that in the diagram, there are three different model training steps followed by a model selection step. These model training steps would train three different models competing with each other for better statistical performance. The dedicated model selection step then picks the top model that will be used in the subsequent components in the end-to-end machine learning workflow.

Figure 7.8 Model training component (dark box) in the end-to-end machine learning system.



In the next section, we will dive deeper into the model training component in Figure 7.8 and discuss potential problems when implementing this component.

7.3.1 Problem

In Chapter 3, we've introduced the parameter server pattern and the collective communication pattern. The parameter server pattern is handy for situations where the model is too large to fit in a single machine, such as the one we would have to build for tagging entities in the 8 million YouTube videos discussed in Chapter 3. On the other hand, the collective communication pattern is useful to speed up the training process for medium-sized models when the communication overhead is significant.

Which pattern should we pick for our model training component?

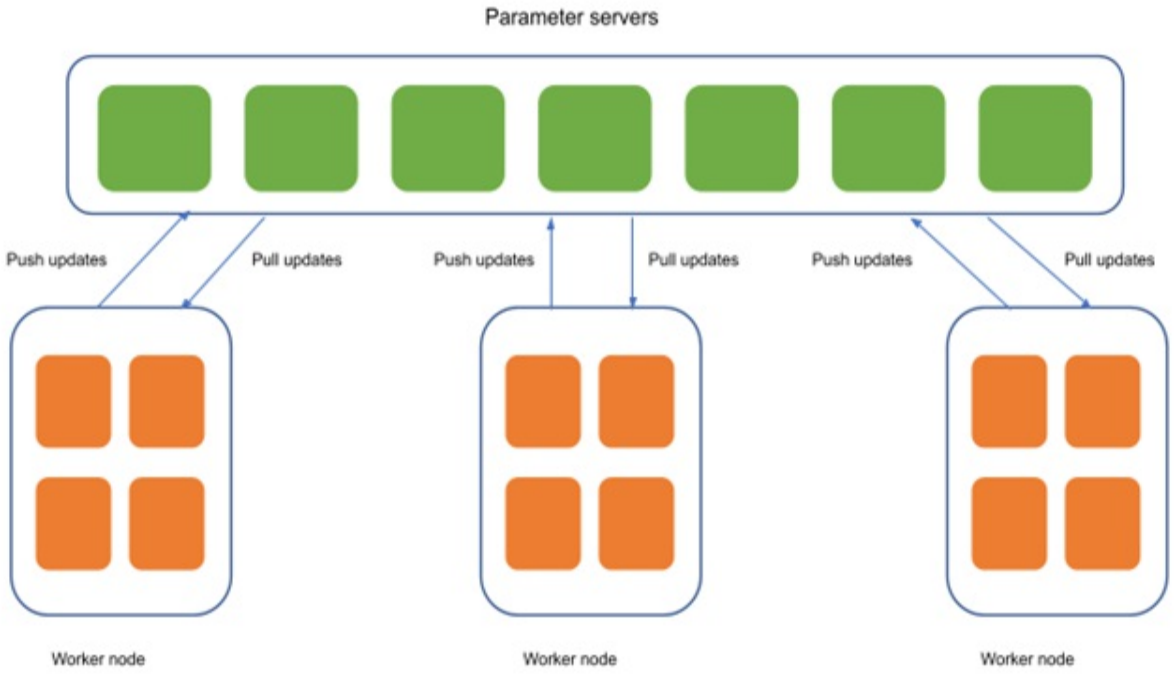
7.3.2 Solution

With the help of parameter servers, we could effectively resolve our challenges when building an extremely large machine learning model that may not fit a single machine.

Even when the model might be too large to fit in a single machine, we could still successfully train the model efficiently with parameter servers. For example, Figure 7.9 is an architecture diagram of the parameter server pattern using multiple parameter servers. Each worker node takes a subset of the dataset, performs calculations required in each of the neural network layers, and then sends the calculated gradients to update one model partition that's stored in one of the parameter servers.

Note that since all workers perform calculations in an asynchronous fashion, the model partitions that each worker node uses to calculate the gradients may not be up-to-date. However, in real-world applications, in order to guarantee that that model partitions each worker node is using or each parameter server is storing is the most recent, we will have to constantly pull and push the updates of the model in between. For instance, two workers can block each other when sending gradients to the same parameter server which makes it hard to gather the calculated gradients on time and requires a strategy to resolve the blocking issue. Unfortunately in real-world distributed training systems where parameter servers are incorporated, multiple workers may be sending the gradients at the same time and thus there are many blocking communications that need to be resolved.

Figure 7.9 A machine learning training component with multiple parameter servers.

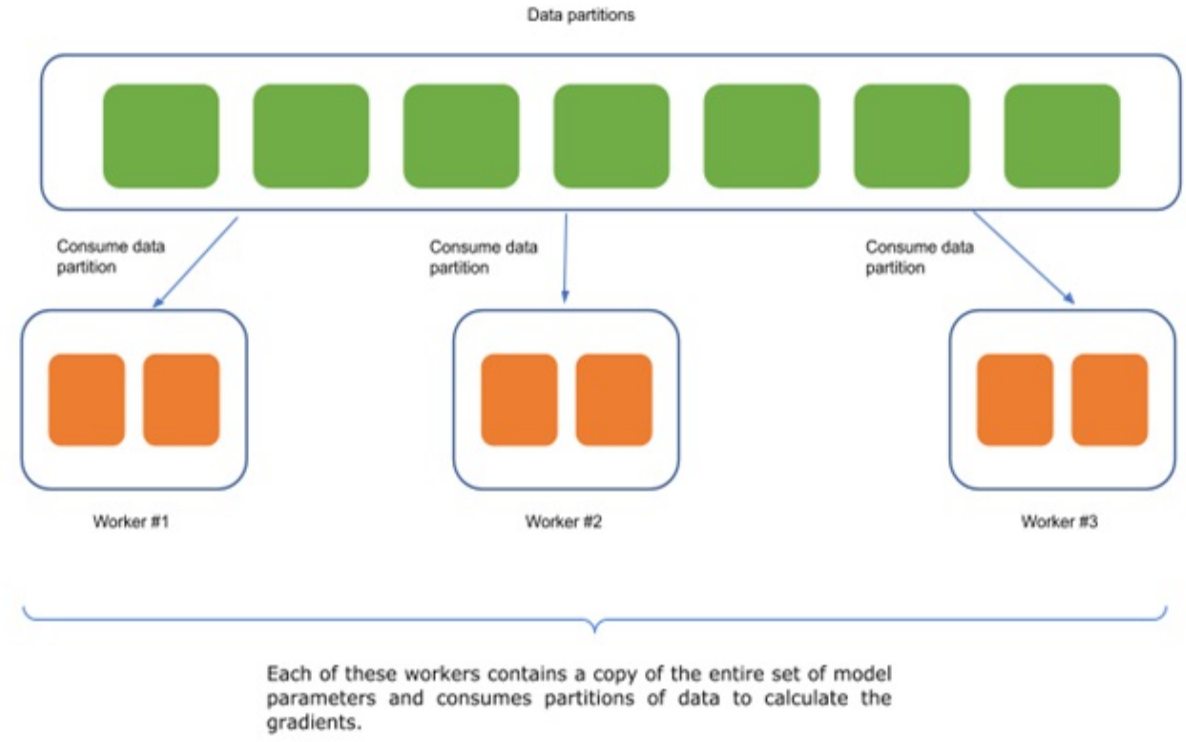


Another challenges comes when deciding the optimal ratio between the number of workers and the number of parameter servers. For example, many workers are sending gradients to the same parameter server at the same time, the problem gets even worse and eventually the blocking communications among different workers or parameter servers become a bottleneck.

Now let's get back to our original application, the Fashion-MNIST classification model we are building is not as large as large recommendation system models and can fit in a single machine easily if we give the machine sufficient computational resources. It's only 30MBs in compressed form. The collective communication model is a perfect for the system we are building.

Now without parameter servers, there are only worker nodes where each worker node stores a copy of the entire set of model parameters, as shown in Figure 7.10.

Figure 7.10 Distributed model training component with only worker nodes where every worker stores a copy of the entire set of model parameters and consumes partitions of data to calculate the gradients.



Recall in Chapter 3 we mentioned that every worker consumes some portions of data and calculates the gradients needed to update the model parameters stored locally on this worker node. As soon as all of the worker nodes have successfully completed the calculation of gradients, we will need to aggregate all the gradients and make sure every worker's entire set of model parameters is updated based on the aggregated gradients. In other words, each worker should store a copy of the exact same updated model.

Going back to the architecture diagram in Figure 7.8, each of the model training steps would leverage the collective communication pattern that takes advantage of the underlying network infrastructure to perform allreduce operations for communicating gradients among multiple workers and allows us to train multiple medium-sized machine learning model in a distributed settings. Once the model is trained, we will start a separate process to pick the top model that will be used for model serving. This step is pretty intuitive and we'll defer the implementation details in the last chapter of the book. In the next section, we will discuss the model serving component of our system.

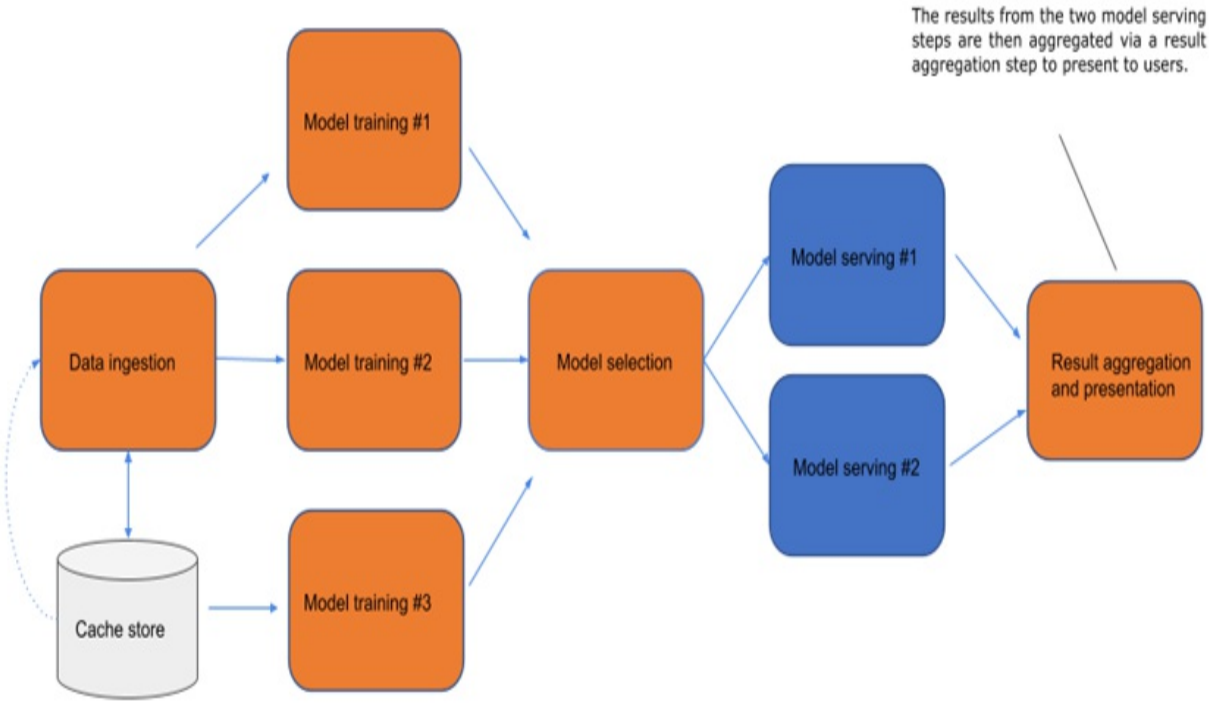
7.3.3 Exercises

1. Why isn’t the parameter server pattern a good fit?
2. Does each worker stores different parts of the model when using the collective communication pattern?

7.4 Model serving

We’ve talked about both the data ingestion and model training component of the system we are building. Next, let’s discuss the model server component we are building that will be essential to the end-user experience. Figure 7.11 shows the model training component in the overall architecture.

Figure 7.11 Model serving component (dark box) in the end-to-end machine learning system.



In Figure 7.11, the model serving components are shown as the two dark boxes between the model selection and result aggregation steps. Let’s take a look at the potential problems and solutions that we will encounter when we actually building this component.

7.4.1 Problem

The model serving system we are building needs to take raw images uploaded by users and then sends the requests to the model server to make inference using the trained model. These model serving requests are being queued and waiting to be processed by the model server.

If the model serving system is a single-node server, it could only serve a limited number of model serving requests on a first-come-first-serve basis. As the number of requests grows in the real-world, the user experience gets bad when users have to wait for a long time to receive the model serving result. In other words, all requests are waiting to be processed by the model serving system but the computational resources are limited to this single node.

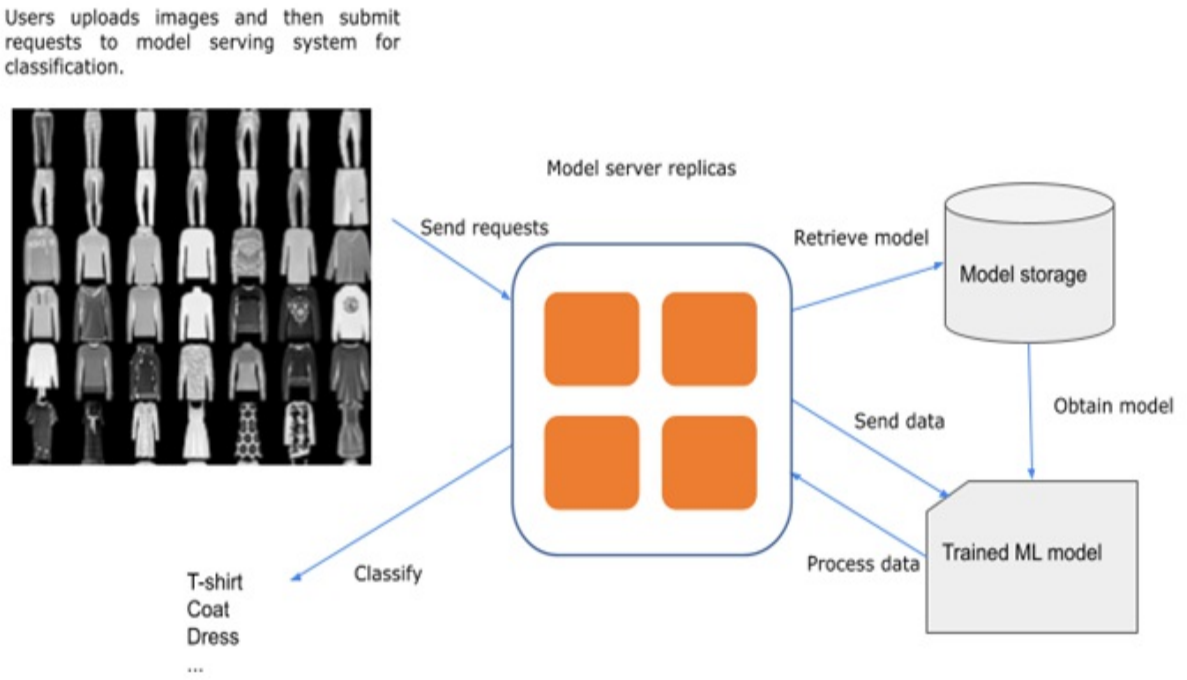
How do we build a more efficient model serving system?

7.4.2 Solution

It's a perfect use case for the replicated services pattern we learned in Chapter 4. Our model serving system takes the images uploaded by users and sends requests to the model server. In addition, unlike the simple single-server design, the system has multiple model server replicas to process the model serving requests asynchronously. Each of the model server replicas takes a single request, retrieves the previously trained classification model from the model training component, and then classifies the images that haven't existed in the Fashion-MNIST dataset.

With the help of the replicated services pattern, we can easily scale up our model server by adding model server replicas to the single-server model serving system. The new architecture is shown in Figure 7.12 below. The model server replicas are capable of handling many requests at a time since each replica can process individual model serving requests independently.

Figure 7.12 The system architecture of the replicated model serving services.

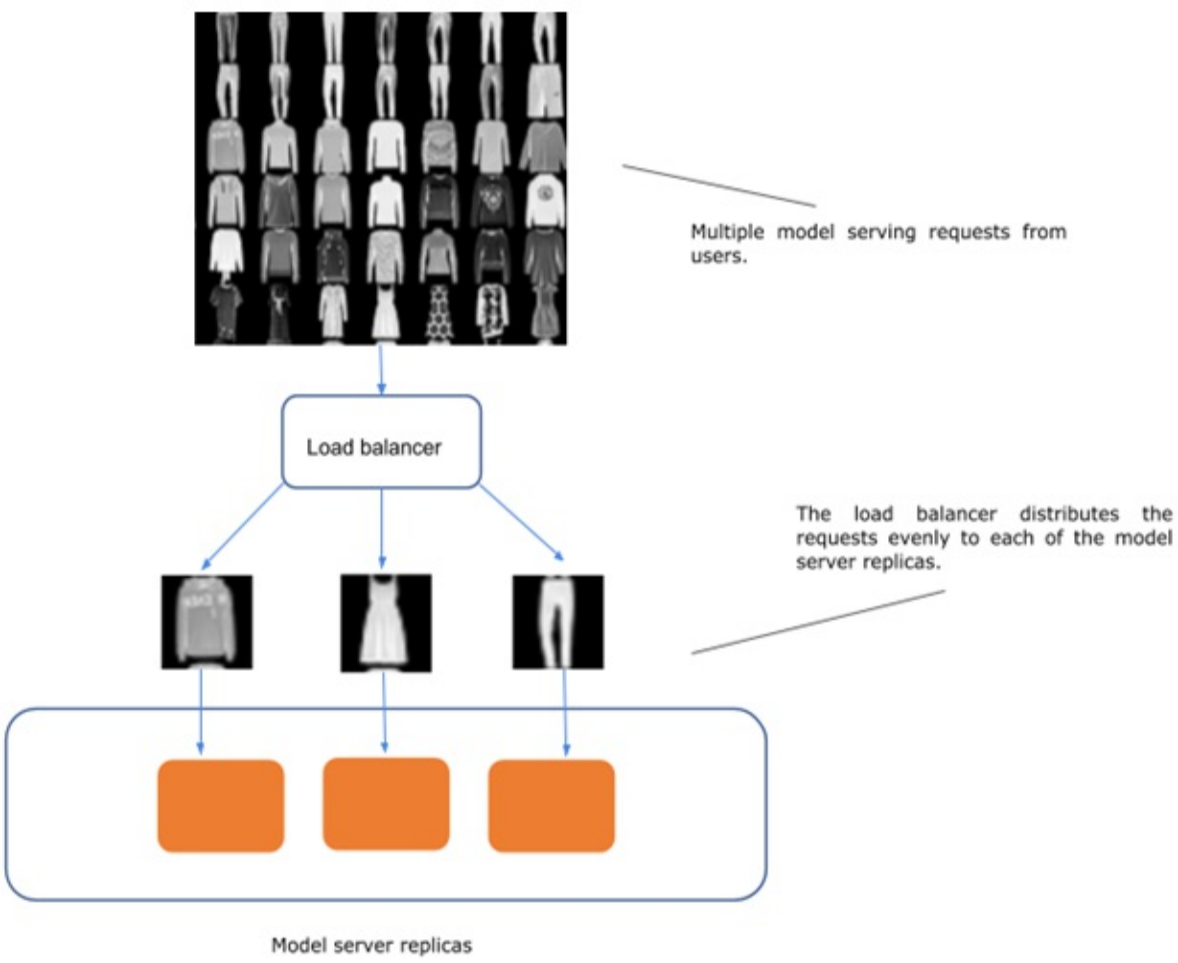


It’s worth noting that multiple model serving requests from users are sent to the model server replicas at the same time after we’ve introduced the replicas. We also need to define a clear mapping relationship among the requests and the model server replicas, which determines which requests are being processed by which of the model server replicas.

In order to distribute the model server requests among the replicas, we need to add an additional load balancer layer. For example, the load balancer takes multiple model serving requests from our users and then distributes the requests evenly to each of the model server replicas, which then are responsible for processing individual requests, including model retrieval and inference on the new data in the request. Figure 7.13 below illustrates this process.

The load balancer uses different algorithms to decide which request goes to which particular model server replica. Example algorithms for load balancing include round robin, least connection method, hashing, etc.

Figure 7.13 A diagram showing how a loader balancer is used to distribute the requests evenly across model server replicas.



Note that from our original architecture diagram in Figure 7.11, there are two individual steps for model serving that each uses different models. Each of those two model serving steps consists of a model serving service with multiple replicas to handle model serving traffic for different models.

7.4.3 Exercises

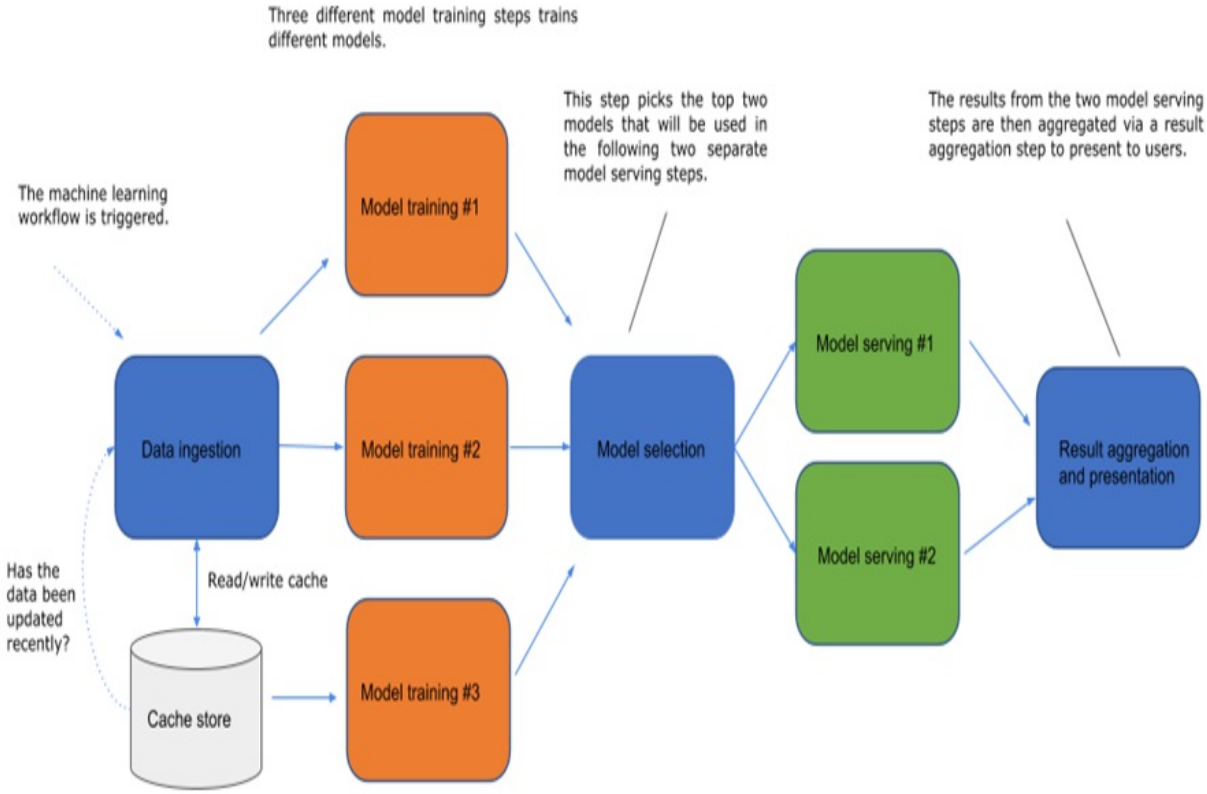
1. What happens when we don’t have a load balancer as part of the model serving system?

7.5 End-to-end workflow

Now that we’ve looked at the individual components. Next, let’s see how to compose an end-to-end workflow that consists of all those components in a

scalable and efficient way. We will also incorporate a few patterns we learned in Chapter 5 into the workflow. Figure 7.14 is a diagram of the end-to-end workflow that we are building.

Figure 7.14 Architecture diagram of the end-to-end machine learning system we will be building.



Instead of paying attention to individual components, we will look at the entire machine learning system we are building that chains all the components together in an end-to-end workflow.

7.5.1 Problem

The Fashion-MNIST dataset is static and does not change over time. However, in order to design a more realistic system, let’s assume that we’ll manually update the Fashion-MNIST dataset regularly. Whenever the updates happen, we may wan to re-run the entire machine learning workflow in order to train a fresh machine learning model that leverages the new data. In other words, we need to execute the data ingestion step every time when

changes happen. In the meantime, when dataset is not updated but we would still like to experiment new machine learning models, we still need to execute the entire workflow that includes the data ingestion step. Data ingestion steps are usually very time-consuming, especially for large datasets. Is there a way to make this workflow more efficient?

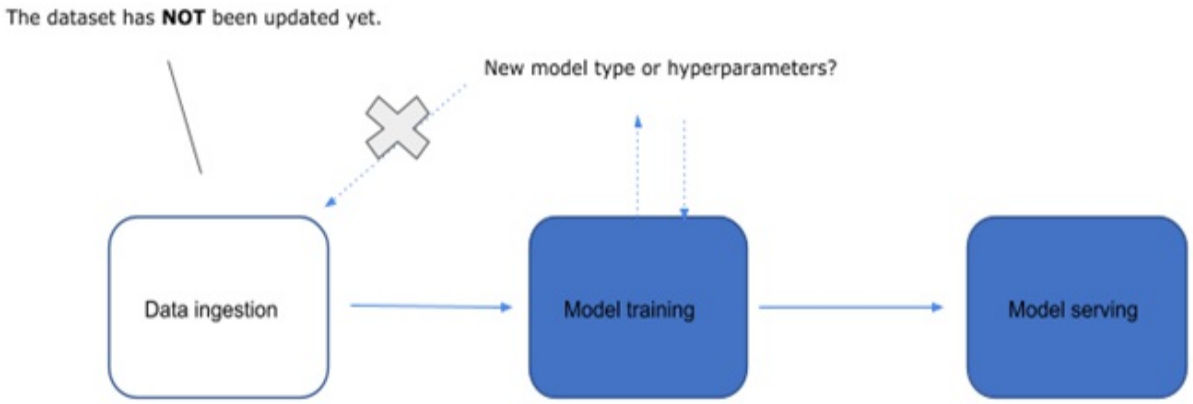
Furthermore, we would like to build a machine learning workflow that would train different models and then select the top model to be used in model serving for users that generates predictions with the knowledge from both of the models. Due to the variance of completion time for each of the model training steps in the existing machine learning workflow, the start of each of the following steps, such as the model selection step and the model serving steps, depend on the completion of its previous steps. However, this sequential execution of steps in the workflow is quite time-consuming and blocks the rest of the steps. For example, when at least one of the model training steps takes much longer to complete than the rest of the steps, the model selection step that follows can only start to execute until this long model training step has completed. As a result, the entire workflow is being delayed by this particular long-running step. Is there a way to accelerate this workflow so it will not be affected by the durations of individual steps?

7.5.2 Solution

For the first problem, we can leverage the step memoization pattern that we learned in Chapter 5. Recall that step memoization could help the system decide whether a step should be executed or skipped. With the help of step memoization, a workflow can identify the steps with redundant workloads that can be skipped without being re-executed and thus greatly accelerate the execution of the end-to-end workflow.

For instance, Figure 7.15 contains a simple workflow that only executes the data ingestion step when we know that the dataset has already been updated. In other words, we don’t want to re-ingest the data that’s already collected again if the new data has not been updated yet.

Figure 7.15 Diagram where the data ingestion step is skipped when the dataset has not been updated yet.



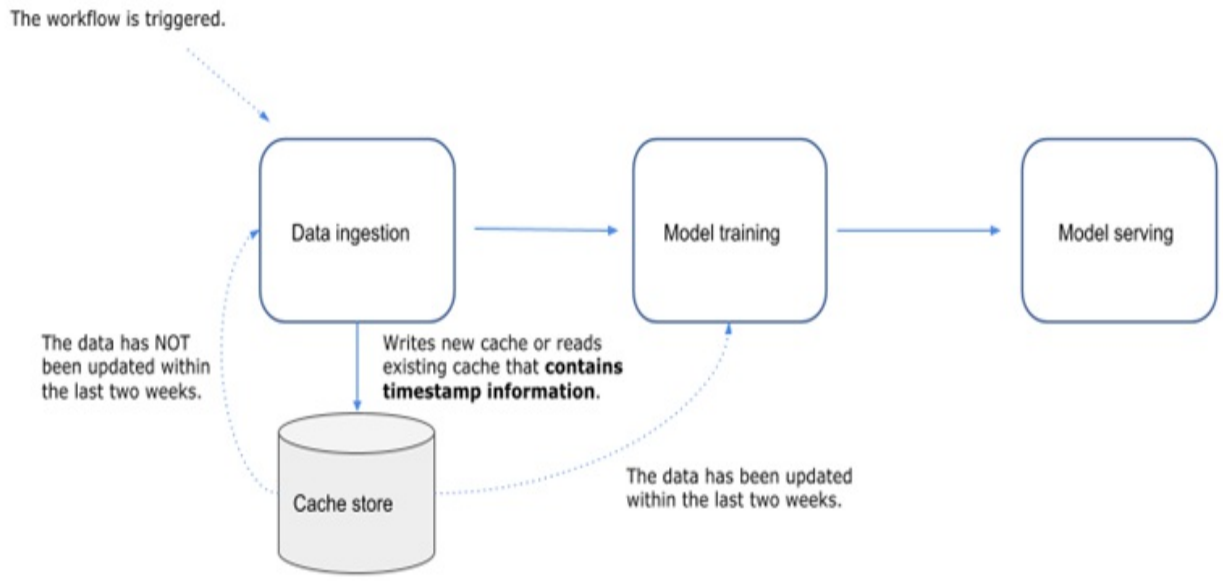
There are many strategies to tell whether the dataset has already been updated. With a pre-defined strategy, we can conditionally re-construct the machine learning workflow and control whether we would like to include a data ingestion step to be re-executed, as shown in the diagram Figure 7.15.

One way to identify whether the dataset has been updated is through the use of cache. Since our Fashion-MNIST dataset is being updated regularly on a fixed schedule, e.g. once a month, we can create a time-based *cache* that stores the location of the ingested and cleaned dataset (assuming the dataset is located in a remote database) and the timestamp of its last updated time.

The data ingestion step in the workflow will then be constructed and executed dynamically based on whether the last updated timestamp is within a particular window. For example, if the time window is set to be two weeks, then we consider the ingested data as fresh if it has been updated within the past two weeks. The data ingestion step will be skipped and the following model training steps will use the already ingested dataset from the location that's stored in the cache. Figure 7.16 illustrates the case where a workflow has been triggered and we check whether the data has been updated within the last two weeks by accessing the cache. If the data is fresh, we skip the execution of the unnecessary data ingestion step and execute the model training step directly. The time window can be used to control how old a cache can be before we consider the dataset as fresh enough that it can be used directly for model training instead of re-ingesting the data again from scratch.

Figure 7.16 The workflow has been triggered and we check whether the data has been updated

within the last two weeks by accessing the cache. If the data is fresh, we skip the execution of the unnecessary data ingestion step and execute the model training step directly.



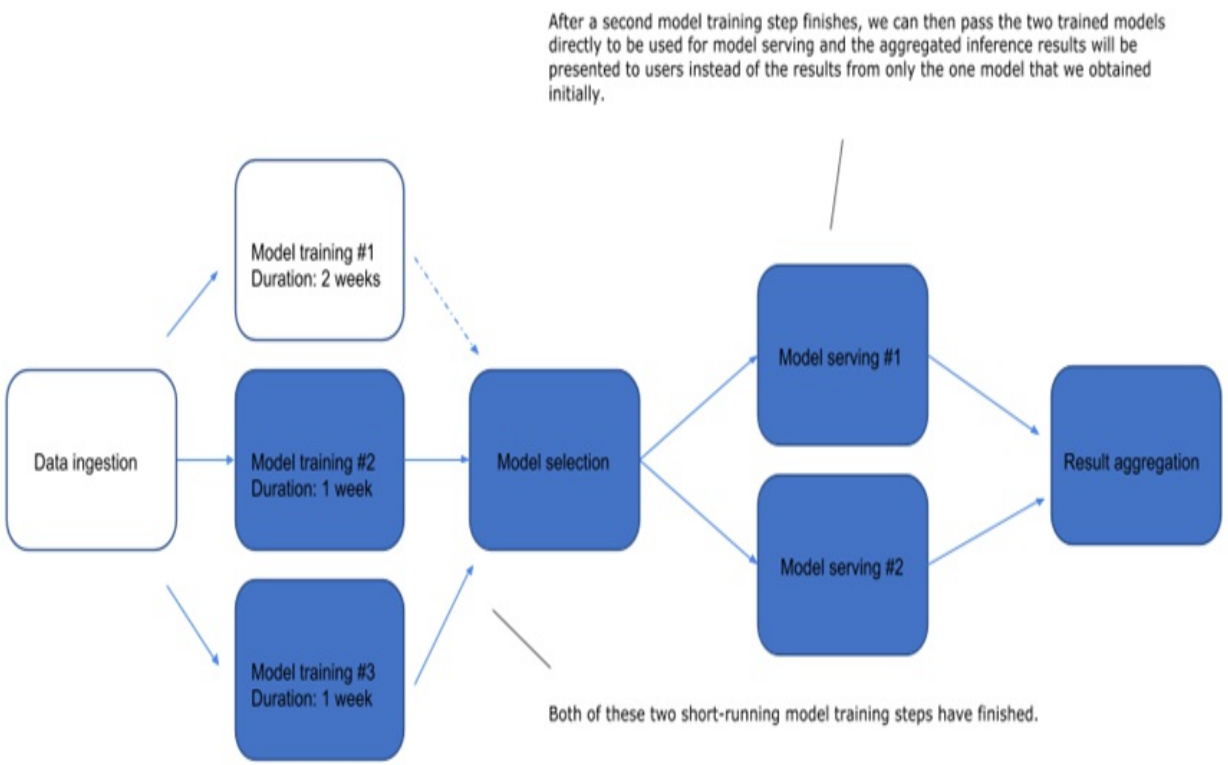
Now let's take a look at the second problem: sequential execution of the steps blocks the subsequent steps in the workflow and is inefficient. This is exactly where the synchronous and asynchronous patterns that we introduced in Chapter 5 can help.

When one of those short-running model training steps finishes, for example, model training step #2 in Figure 7.17, we successfully obtain a trained machine learning model. In fact, we can already leverage this already trained model and use it directly in our model serving system without waiting for the rest of the model training steps to complete. As a result, the users would be able to see the results of image classification from their model serving requests that contain videos as soon as we have trained one model from one of the steps in the workflow.

After a second model training step #3 in Figure 7.17 finishes, we can then pass the two trained models directly to be used for model serving, and the aggregated inference results will be presented to users instead of the results from only the one model that we obtained initially, as shown in Figure 7.17.

Figure 7.17 After a second model training step finishes, we can then pass the two trained models directly to be used for model serving, and the aggregated inference results will be presented to

users instead of the results from only the one model we obtained initially.



As a result, we can continue to use the trained models for model selection and model serving, and in the meantime, the long-running model training steps are still running. In other words, they execute asynchronously without depending on each other's completion. The workflow can proceed and start executing the next step before the previous step finishes. The entire workflow will no longer be blocked by the long-running model training step. Instead, it can continue to use the already trained models from the short-running model training step in the model serving system and can start handling users' model serving requests.

7.5.3 Exercises

1. Which component can be benefited the most from step memoization?
2. How do we tell if a step's execution can be skipped if its workflow has been triggered to run again?

7.6 Summary

- The data ingestion component leverages the caching pattern to speed up the processing of multiple epochs of the dataset.
- The model training component leverages the collective communication pattern to avoid the potential communication overhead among workers and parameter servers.
- We leveraged the replicated model server replicas that are capable of handling many requests at a time since each replica can process individual model serving requests independently.
- We chained all our components into a workflow and used caching to effectively skips time-consuming component such as data ingestion.

Answers to Exercises:

Section 7.2

1. In-memory.
2. Yes.

Section 7.3

1. There are blocking communications among workers and parameter servers.
2. No, each worker stores exactly the same copy of the model.

Section 7.4

1. We cannot balance or distribute the model serving requests among the replicas.

Section 7.5

2. Data ingestion component.
3. Leveraging the metadata in the step cache.

8 Overview of relevant technologies

This chapter covers

- Getting familiar with model building using TensorFlow
- Understanding key terminologies on Kubernetes
- Running distributed machine learning workloads with Kubeflow
- Deploying container-native workflows using Argo Workflows

In the previous chapter, we went through the project background and system components to understand our strategies for implementing each component. We also discussed the challenges for each component and shared the patterns we will apply to address them. As mentioned, we will dive into the project's implementation details in the book's last chapter. However, since we will use different technologies in the project, and it’s not easy to cover all the basics on the fly, in this chapter, we will learn the basic concepts of the four technologies (TensorFlow, Kubernetes, Kubeflow, and Argo Workflows) and gain hands-on experience.

Each of these four technologies has a different purpose, but all will be used to implement the final project in the book's last chapter. TensorFlow will be used for data processing, model building, and evaluation. We will use Kubernetes as our core distributed infrastructure. On top of that, Kubeflow will be used for submitting distributed model training jobs to the Kubernetes cluster, and Argo Workflows will be used to construct and submit the end-to-end machine learning workflows.

8.1 TensorFlow: The machine learning framework

TensorFlow is an end-to-end machine learning platform. It has been widely adopted in academia and industries for different applications and uses cases, such as image classification, recommendation systems, natural language processing, etc. TensorFlow is highly portable, deployable on different hardware, and has multi-language support.

TensorFlow has a large ecosystem. Here are some highlighted projects in this ecosystem:

- TensorFlow.js is a library for machine learning in JavaScript. Users can use machine learning directly in the browser or in Node.js.
- TensorFlow Lite is a mobile library for deploying models on mobile, microcontrollers, and other edge devices.
- TFX is an end-to-end platform for deploying production ML pipelines.
- TensorFlow Serving is a flexible, high-performance serving system for machine learning models, designed for production environments.
- TensorFlow Hub is a repository of trained machine learning models ready for fine-tuning and deployable anywhere. Reuse trained models like BERT and Faster R-CNN with just a few lines of code.

More can be found in the TensorFlow GitHub organization: https://github.com/tensorflow. We will use one of the above projects, TensorFlow Serving, in our model serving component. In the next section, we'll walk through some basic examples in TensorFlow to train a machine learning model locally using the MNIST dataset.

8.1.1 Basics

Let's first install Anaconda for Python 3 for the basic examples that we will go through. Anaconda (https://www.anaconda.com) is a distribution of the Python and R programming languages for scientific computing that aims to simplify package management and deployment. The distribution includes data-science packages suitable for Windows, Linux, and macOS.

Once Anaconda is installed, use the following command in your console to install a Conda environment with Python 3.8:

Listing 8.1 Create Conda environment

```
> conda create --name dist-ml python=3.9 -y
```

Then we can activate this environment with the following:

Listing 8.2 Activate Conda environment

```
> conda activate dist-ml
```

Then we can install TensorFlow in this Python environment:

Listing 8.3 Install TensorFlow

```
> pip install --upgrade pip
> pip install tensorflow==2.10.0
```

If you encounter any issues, please refer to this guide:
https://www.tensorflow.org/install

Note that you may need to uninstall your existing numpy and reinstall it in some cases:

Listing 8.4 Install NumPy

```
> pip install numpy --ignore-installed
```

If you are on Mac, check out Metal plugin for acceleration:
https://developer.apple.com/metal/tensorflow-plugin/

Once we've successfully installed TensorFlow, we can start with a basic image classification example!

Let's first load and preprocess our simple MNIST dataset. Recall that the MNIST dataset contains images for handwritten digits from 0 to 9 where each row represents images for a particular handwritten digit, as shown in Figure 8.1.

Figure 8.1 Screenshot of some example images for handwritten digits from 0 to 9 where each row represents images for a particular handwritten digit. Source: Josep Steffan, licensed under CC BY-SA 4.0.

Note that tf.keras is a high-level API for model training in TensorFlow, and we will use it for both loading the built-in datasets and model training and evaluation.

Listing 8.5 Load the MNIST dataset

```
> import tensorflow as tf
> (x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.
```

The function load_data() will use a default path to save the MNIST dataset if we don't explicitly specify it. This function will return NumPy arrays for training images and labels and testing images and labels. We split the dataset into training and testing so we can run both model training and evaluation in our example. You should see the following log if the dataset is downloaded successfully:

```
Downloading data from https://storage.googleapis.com/tensorflow/t
11490434/11490434 [==============================] - 1s 0us/step
```

NumPy array is a common data type in Python's scientific computing ecosystem. It describes multi-dimensional arrays and has three properties: data, shape, and dtype. Let's use our training images as an example:

Listing 8.6 Inspect dataset

```
> x_train.data
<memory at 0x16a392310>
> x_train.shape
(60000, 28, 28)
> x_train.dtype
dtype('uint8')
> x_train.min()
0
> x_train.max()
255
```

We can observe that x_train is a 60,000x28x28 three-dimensional array. The data type is uint8 from 0 to 255. In other words, this object contains 60,000 gray-scale images with a resolution of 28x28.

Next, we can perform some feature preprocessing on our raw images. Since many algorithms and models are sensitive to the scale of the features, we often center and scale features into a range such as [0, 1] or [-1, 1]. In our case, we can do this easily by dividing the images by 255.

Listing 8.7 Pre-processing function

```
def preprocess(ds):
    return ds / 255.0

x_train = preprocess(x_train)
x_test = preprocess(x_test)

> x_train.dtype
dtype('float64')

> x_train.min()
0.0

> x_train.max()
1.0
```

After preprocessing the images in training and testing set, we can instantiate a simple multi-layer neural network model. We use the Keras API to define the model architecture. First, we use Flatten to expand the two-dimensional images into a one-dimensional array by specifying the input shape to be 28x28. The second layer is densely connected and uses the “relu” activation function to introduce some non-linearity. The third layer is a dropout layer to

reduce overfitting and make the model more generalizable. Since the hand-written digits consist of 10 different digits from 0 to 9, our last layer is densely connected for 10-class classification with softmax activation.

Listing 8.8 Sequential model definition

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation='softmax')
])
```

After we've defined the model architecture, we need to specify three different components: the evaluation metric, loss function, and optimizer.

Listing 8.9 Model compilation with optimizer, loss function, and optimizer

```
model.compile(optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy'])
```

We can then start our model training with 5 epochs as well as evaluation via the following:

Listing 8.10 Model training using the training data

```
model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

We should see the training progress in the log:

```
Epoch 1/5
1875/1875 [==============================] - 11s 4ms/step - loss:
Epoch 2/5
1875/1875 [==============================] - 9s 5ms/step - loss:
Epoch 3/5
1875/1875 [==============================] - 9s 5ms/step - loss:
Epoch 4/5
1875/1875 [==============================] - 8s 4ms/step - loss:
Epoch 5/5
1875/1875 [==============================] - 8s 4ms/step - loss:
10000/10000 [=========] - 0s - loss: 0.0726 - accuracy: 0.9788
```

And the log from the model evaluation:

```
313/313 [==============================] - 1s 4ms/step - loss: 0.
[0.07886667549610138, 0.976300060749054]
```

We should observe that as the loss decreases during training, the accuracy increases to 97.8% on training data. The final trained model has an accuracy of 97.6% on the testing data. Note that your result might be slightly different due to the randomness in the modeling process.

After we've trained the model and are happy with its performance, we can save it so that we don't have to retrain it from scratch next time via the following:

Listing 8.11 Save the trained model

```
model.save('my_model.h5')
```

This saves the model as file "my_model.h5" in the current working directory. When we start a new Python session, we can import TensorFlow and load the model object from "my_model.h5" file.

Listing 8.12 Load the saved model

```
import tensorflow as tf
model = tf.keras.models.load_model('my_model.h5')
```

We've learned how to train a model using TensorFlow's Keras API for a single set of hyperparameters. These hyperparameters remain constant over the training process and directly impact the performance of your machine learning program. Let's learn how to perform hyperparameter tuning for your TensorFlow program with Keras Tuner (https://keras.io/keras_tuner/) to find the best set of hyperparameters.

First, install the Keras Tuner library:

Listing 8.13 Install Keras Tuner package

```
pip install -q -U keras-tuner
```

Once it’s installed, you should be able to import all the required libraries:

Listing 8.14 Import necessary packages

```
import tensorflow as tf
from tensorflow import keras
import keras_tuner as kt
```

We will use the same MNIST dataset and the preprocessing functions for our hyperparameter tuning example.

We then wrap our model definition into a Python function:

Listing 8.15 Model building function using TensorFlow and Keras Tuner

```
def model_builder(hp):
  model = keras.Sequential()
  model.add(keras.layers.Flatten(input_shape=(28, 28)))
  hp_units = hp.Int('units', min_value=32, max_value=512, step=32
  model.add(keras.layers.Dense(units=hp_units, activation='relu')
  model.add(keras.layers.Dense(10))
  hp_learning_rate = hp.Choice('learning_rate', values=[1e-2, 1e-
  model.compile(optimizer=keras.optimizers.Adam(learning_rate=hp_
                loss=keras.losses.SparseCategoricalCrossentropy(f
                metrics=['accuracy'])
  return model
```

This is essentially the same as what we used previously for training model with a single set of hyperparameters, except that we also defined hp_units and hp_learning_rate objects that are used in our dense layer and optimizer.

The hp_units object instantiates an integer that will be tuned between 32 and 512 and used as the number of units in the first densely connected layer. The hp_learning_rate object will tune the learning rate for the “adam” optimizer that will be chosen among these values: 0.01, 0.001, or 0.0001.

Once the model builder is defined, we can then instantiate our tuner. There are several tuning algorithms we can use, e.g. random search, Bayesian optimization, and Hyperband. Here we use the hyperband tuning algorithm. It uses adaptive resource allocation and early-stopping so that it can converge faster on a high-performing model.

Listing 8.16 Hyperband model tuner

```
tuner = kt.Hyperband(model_builder,
                     objective='val_accuracy',
                     max_epochs=10,
                     factor=3,
                     directory='my_dir',
                     project_name='intro_to_kt')
```

We use the validation accuracy as the objective, and maximum number of epochs is 10 during model tuning.

To reduce overfitting, we can create an EarlyStopping callback to stop training early as soon as the model reaches a threshold for the validation loss. Make sure to reload the dataset into memory if you've started a new Python session.

Listing 8.17 Early-stopping callback

```
early_stop = tf.keras.callbacks.EarlyStopping(monitor='val_loss',
```

Now we can start our hyperparameter search via tuner.search():

Listing 8.18 Hyperparameter search with early-stopping

```
tuner.search(x_train, y_train, epochs=30, validation_split=0.2, c
```

Once the search is complete, we can obtain the best set of hyperparameters and then build the model with the optimal hyperparameters and train it on the data for 30 epochs

Listing 8.19 Obtain best hyperparameters and train the model

```
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]
model = tuner.hypermodel.build(best_hps)
model.fit(x_train, y_train, epochs=50, validation_split=0.2)
```

When we evaluate the model on our test data, we should see it's more performant than our baselines model without hyperparameter tuning.

Listing 8.20 Model evaluation on the test data

```
model.evaluate(x_test, y_test)
```

We've learned how to run TensorFlow locally on a single machine. To take the most advantage of TensorFlow, the model training process should be run in a distributed cluster, which is where Kubernetes comes into play. In the next section, we will introduce Kubernetes and provide hands-on examples to learn the fundamentals.

8.1.2 Exercises

1. Can you use the previously saved model directly for model evaluation?
2. Instead of using the Hyperband tuning algorithm, could you try the random search algorithm as well?

8.2 Kubernetes: The distributed container orchestration system

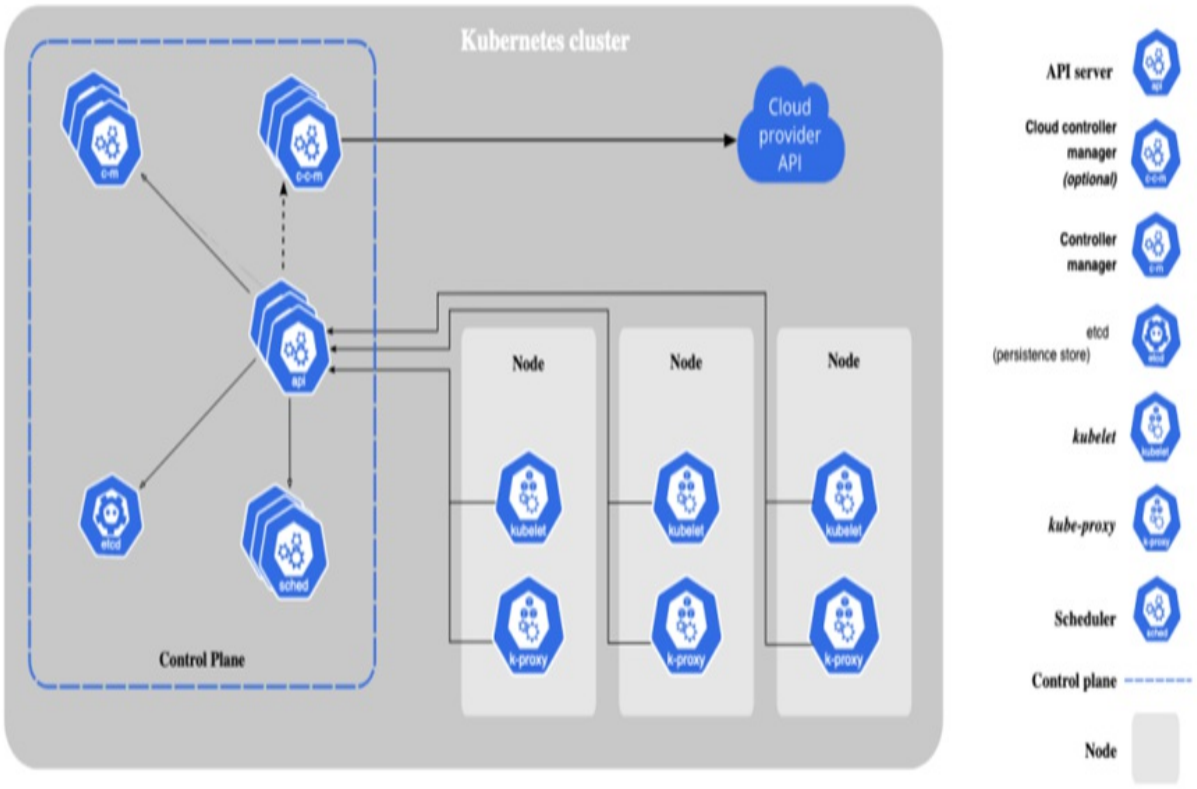
Kubernetes (also known as K8s) is an open-source system for automating the deployment, scaling, and management of containerized applications. It abstracts away complex container management and provides declarative configurations to orchestrate containers in different computing environments.

Containers are grouped into logical units for a particular application for easy management and discovery. Kubernetes builds upon more than 16 years of experience running production workloads at Google, combined with best-of-breed ideas and practices from the community. Its main design goal is to make it easy to deploy and manage complex distributed systems, while still benefiting from the improved utilization that containers enable. The fact that it's open source gives the community the freedom to take advantage of on-premises, hybrid, or public cloud infrastructure and lets you effortlessly move workloads to where it matters.

Kubernetes is designed to scale without increasing your operations team. Figure 8.2 is an architecture diagram of Kubernetes and its components. However, we won't be diving into those components since they are not the focus of this book. One thing to note is that we will use kubectl (on the left-hand side of the diagram), a command line interface of Kubernetes, to

interact with the K8s cluster and obtain information that we are interested in.

Figure 8.2 Architecture diagram of Kubernetes. Source: Kubernetes Website, licensed under Creative Commons Attribution 4.0 International.



We will go through some basic concepts and examples to build our knowledge and prepare the following sections on Kubeflow and Argo Workflows.

8.2.1 Basics

First, let's set up a local Kubernetes cluster. We'll use k3d (https://k3d.io) to bootstrap the local cluster. k3d is a lightweight wrapper to run k3s (a minimal Kubernetes distribution provided by Rancher Lab) in Docker. k3d makes it very easy to create either single-node or multi-node k3s clusters in Docker for local development that requires a Kubernetes cluster.

Let's create a Kubernetes cluster called "distml" via k3s.

Listing 8.21 Create a local K3s cluster

```
> k3d cluster create distml --image v1.25.3+k3s1
```

We can get the list of nodes for the cluster we created via the following:

Listing 8.22 Obtain the list of nodes in the cluster

```
> kubectl get nodes

NAME                  STATUS  ROLES                  AGE  VERSION
K3d-distml-server-0   Ready   control-plane,master   1m   v1.25.3
```

In this case, the node was created one minute ago, and we are running the "v1.25.3+k3s1" version of the k3s distribution. The status is ready so that we can proceed to the next steps.

We can also look at the node's details via "kubectl describe node k3d-distml-server-0". For example, the labels and system info contain information on the operating system and its architecture, whether this node is a master node, etc.

```
Labels:                 beta.kubernetes.io/arch=arm64
                        beta.kubernetes.io/instance-type=k3s
                        beta.kubernetes.io/os=linux
                        kubernetes.io/arch=arm64
                        kubernetes.io/hostname=k3d-distml-server-0
                        kubernetes.io/os=linux
                        node-role.kubernetes.io/control-plane=true
                        node-role.kubernetes.io/master=true
                        node.kubernetes.io/instance-type=k3s

System Info:
  Machine ID:
  System UUID:
  Boot ID:                    73db7620-c61d-432c-a1ab-343b28ab
  Kernel Version:             5.10.104-linuxkit
  OS Image:                   K3s dev
  Operating System:           linux
  Architecture:               arm64
  Container Runtime Version:  containerd://1.5.9-k3s1
  Kubelet Version:            v1.22.7+k3s1
  Kube-Proxy Version:         v1.22.7+k3s1
```

The node's addresses are shown as part of it:

```
Addresses:
  InternalIP:  172.18.0.3
  Hostname:       k3d-distml-server-0

The capacity of the node is also available, indicating how much c

Capacity:
  cpu:                4
  ephemeral-storage:  61255492Ki
  hugepages-1Gi:      0
  hugepages-2Mi:      0
  hugepages-32Mi:     0
  hugepages-64Ki:     0
  memory:             8142116Ki
  pods:               110
```

Then we'll create a *namespace* called "basics" in this cluster for our project. Namespaces in Kubernetes provide a mechanism for isolating groups of resources within a single cluster. Names of resources need to be unique within a namespace, but not across namespaces. Our following examples will be in this single namespace.

Listing 8.23 Create a new namespace

```
> kubectl create ns basics
```

Once the cluster and namespace are set up, we'll use a convenient tool called kubectx to help us inspect and navigate between namespaces and clusters (https://github.com/ahmetb/kubectx). Note that this tool is not required for day-to-day work with Kubernetes but should make it much easier to work with for developers. For example, we can obtain a list of clusters and namespaces that we can connect to via the following:

Listing 8.24 Switch contexts and namespaces

```
> kubectx
d3d-k3s-default
k3d-distml

> kubens
default
kube-system
```

```
kube-public
kube-node-lease
basics
```

For example, we can switch to the “distml” cluster via the “k3d-distml” context and the “basics” namespace that we just created as follows:

Listing 8.25 Activate context

```
> kubectx k3d-distml
Switched to context "k3d-distml".

> kubens basics Active namespace is "basics".
```

Switching contexts and namespaces is often needed when working with multiple clusters and namespaces. We are using the basics namespace to run the examples in this chapter but we will switch to another namespace dedicated to our project in the next chapter.

Next, we will be creating a Kubernetes *pod*. Pods are the smallest deployable units of computing that you can create and manage in Kubernetes. A Pod may consist of one or more containers, with shared storage and network resources, and a specification for how to run the containers. A Pod's contents are always co-located and co-scheduled and run in a shared context. The concept of the pod models an application-specific "logical host", meaning that it contains one or more application containers that are relatively tightly coupled. In non-cloud contexts, applications executed on the same physical or virtual machine are analogous to cloud applications executed on the same logical host.

In other words, a pod is similar to a set of containers with shared namespaces and shared filesystem volumes.

The following is an example of a pod that consists of a container running the image whalesay to print out a hello world message. We save the following pod spec in a file named “hello-world.yaml”

Listing 8.26 Example pod

```
apiVersion: v1
```

```
kind: Pod
metadata:
  name: whalesay
spec:
  containers:
  - name: whalesay
    image: docker/whalesay:latest
    command: [cowsay]
    args: ["hello world"]
```

To create the pod shown above, run the following command:

Listing 8.27 Create the example pod to the cluster

```
> kubectl create -f basics/hello-world.yaml

pod/whalesay created
```

We can then check if the pod has been created by retrieving the list of pods. Note that “pods” is plural so we can get the full list of created pods. We will use the singular form to get the details of this particular pod later.

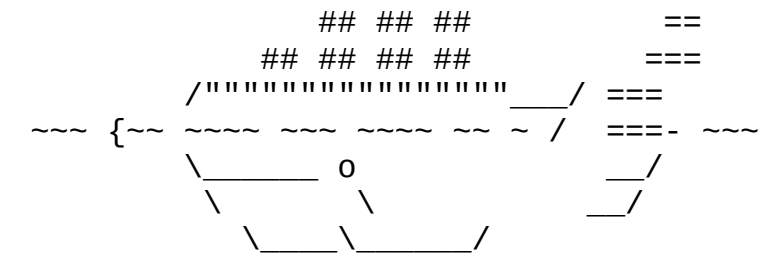
Listing 8.28 Get the list of pods in the cluster

```
> kubectl get pods

NAME       READY   STATUS      RESTARTS      AGE
whalesay   0/1     Completed   2 (20s ago)   37s
```

The pod status is “Completed” so we can take a look at what’s being printed out in the whalesay container like the following:

Listing 8.29 Check the pod logs

```
> kubectl logs whalesay

 _____________
< hello world >
 -------------
        \
         \
          \
                        ##         .
```



We can also retrieve the raw YAML of the pod via kubect. Note that we use “-o yaml” here to get the plain YAML but other formats, such as JSON, are also supported. We use the singular “pod” to get the details of this particular pod instead of the full list of existing pods, as mentioned earlier.

Listing 8.30 Get the raw pod YAML

```
> kubectl get pod whalesay -o yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: "2022-10-22T14:30:19Z"
  name: whalesay
  namespace: basics
  resourceVersion: "830"
  uid: 8e5e13f9-cd58-45e8-8070-c6bbb2dddb6e
spec:
  containers:
  - args:
       - hello world
       command:
       - cowsay
       image: docker/whalesay:latest
       imagePullPolicy: Always
       name: whalesay
       resources: {}
       terminationMessagePath: /dev/termination-log
       terminationMessagePolicy: File
       volumeMounts:
       - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
         name: kube-api-access-x826t
         readOnly: true
  dnsPolicy: ClusterFirst
  enableServiceLinks: true
  nodeName: k3d-distml-server-
```

```
<...truncated…>

  volumes:
  - name: kube-api-access-x826t
      projected:
        defaultMode: 420
        sources:
        - serviceAccountToken:
            expirationSeconds: 3607
            path: token
        - configMap:
            items:
            - key: ca.crt
              path: ca.crt
            name: kube-root-ca.crt
        - downwardAPI:
            items:
            - fieldRef:
                apiVersion: v1
                fieldPath: metadata.namespace
              path: namespace
status:
  conditions:
  - lastProbeTime: null
      lastTransitionTime: "2022-10-22T14:30:19Z"
      status: "True"
      type: Initialized
  - lastProbeTime: null
      lastTransitionTime: "2022-10-22T14:30:19Z"
      message: 'containers with unready status: [whalesay]'
      reason: ContainersNotReady
      status: "False"
      type: Ready
```

You may be surprised how much additional content, such as status and conditions, has been added to the original YAML we used to create the pod. The additional information are appended and updated via the Kubernetes server so that client-side applications know the current status of the pod.

Note that even though we didn’t specify the namespace explicitly, the pod was created in the “basics” namespace since we have used the kubens command to set the current namespace.

That’s it for the basics of Kubernetes! In the next section, we will study how to use Kubeflow to run distributed model training jobs in the local

Kubernetes cluster we just set up.

8.2.2 Exercises

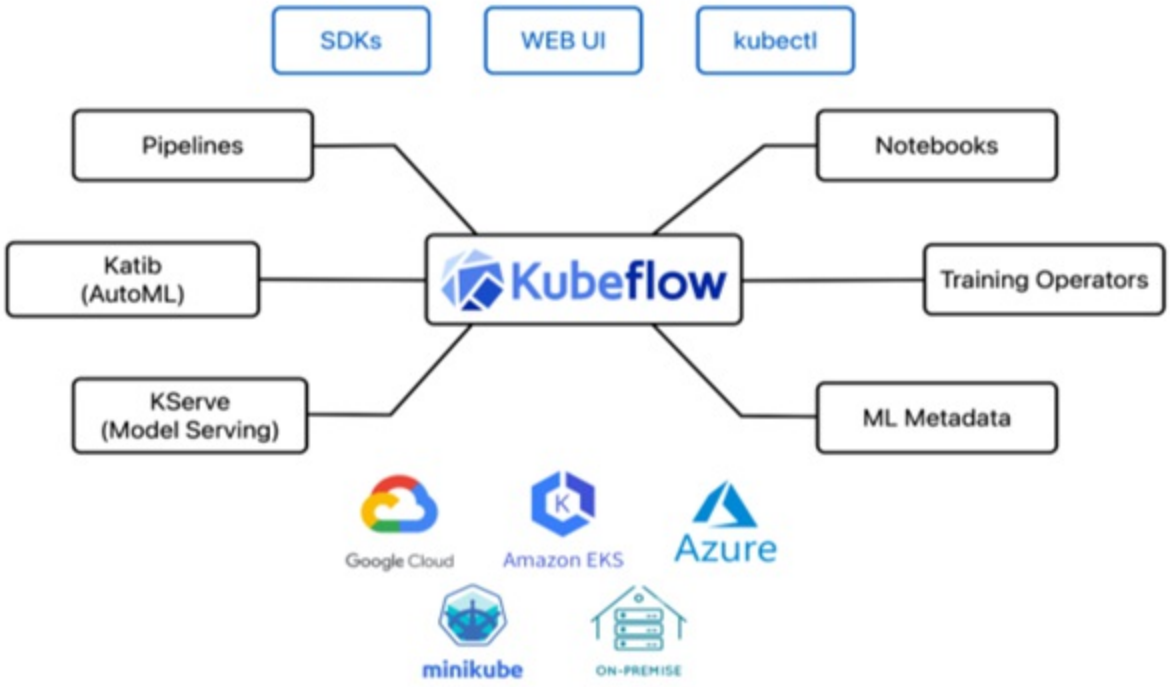
1. How do you get the pod information in JSON format?
2. Can a pod contain multiplier containers?

8.3 Kubeflow: Machine learning workloads on Kubernetes

The Kubeflow project is dedicated to making deployments of machine learning workflows on Kubernetes simple, portable and scalable. The goal of Kubeflow is not to recreate other services, but to provide a straightforward way to deploy best-of-breed open-source systems for ML to diverse infrastructures. Anywhere you are running Kubernetes, you should be able to run Kubeflow. We will be using Kubeflow for submitting distributed machine learning model training jobs to a Kubernetes cluster.

Let’s first take a look at what components Kubeflow provides. Figure 8.3 is a diagram that consists of the main components.

Figure 8.3 Main components of Kubeflow. Source: Kubeflow, licensed under Creative Commons Attribution 4.0 International.



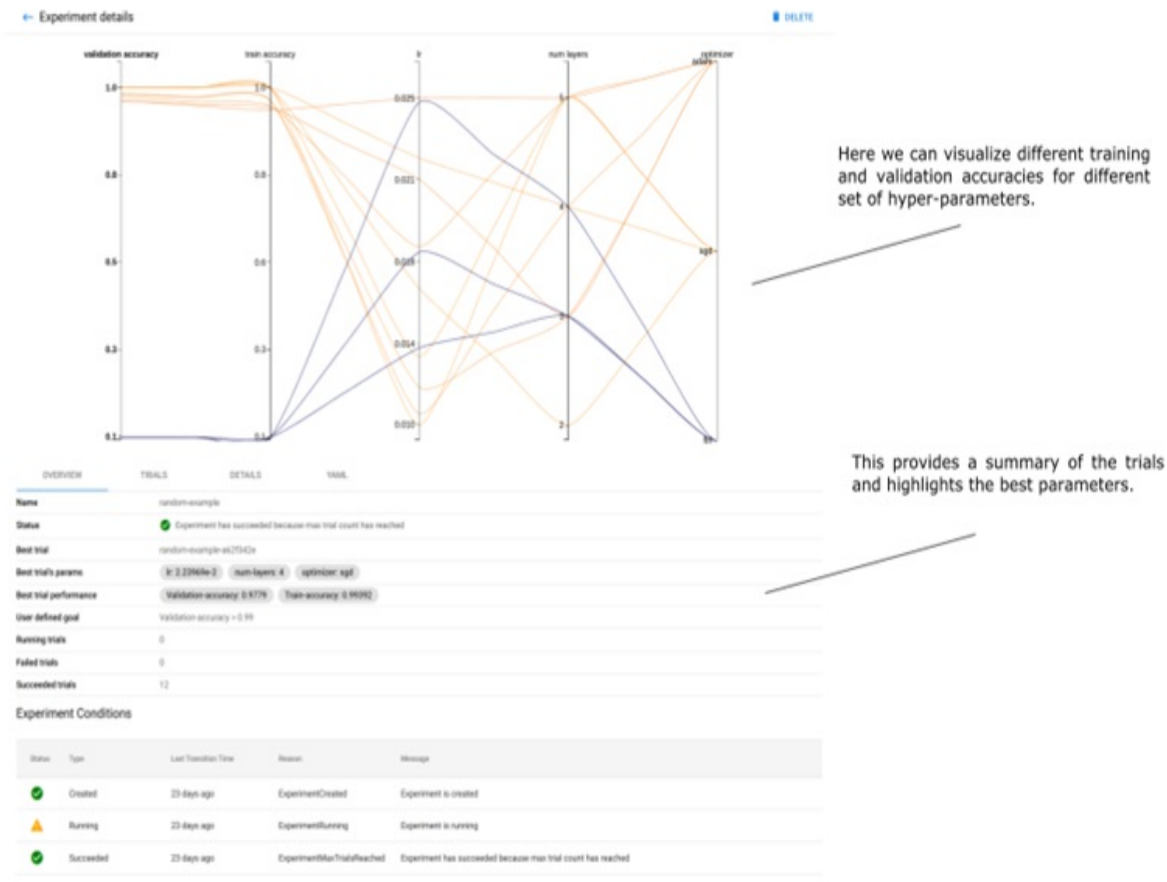
Kubeflow Pipelines (KFP: https://github.com/kubeflow/pipelines) provides Python SDK to make machine learning pipelines easier. KFP is a platform for building and deploying portable and scalable machine learning workflows using Docker containers. The primary objectives of Kubeflow Pipelines are to enable the following:

- End-to-end orchestration of ML workflows
- Pipeline composability through reusable components and pipelines
- Easy management, tracking, and visualization of pipeline definitions, runs, experiments, and ML artifacts
- Efficient use of computing resources by eliminating redundant executions through caching
- Cross-platform pipeline portability through a platform-neutral IR YAML pipeline definition

Note that it uses Argo Workflows as the backend workflow engine, which we will introduce in the next section, and we'll be using Argo Workflows directly instead of using a higher-level wrapper like KFP. The ML metadata project has been merged into KFP and serves as the backend for logging metadata produced in machine learning workflows written in KFP.

Next is Katib (https://github.com/kubeflow/katib). Katib is a Kubernetes-native project for automated machine learning (AutoML). Katib supports hyperparameter tuning, early stopping, and Neural Architecture Search (NAS). Katib is a project which is agnostic to machine learning frameworks. It can tune hyperparameters of applications written in any language of the users' choice and natively supports many ML frameworks, such as TensorFlow, Apache MXNet, PyTorch, XGBoost, and others. Katib can perform training jobs using any Kubernetes Custom Resources with out-of-the-box support for Kubeflow Training Operator, Argo Workflows, Tekton Pipelines, and many more. Figure 8.4 is a screenshot of the Katib UI that performs experiment tracking.

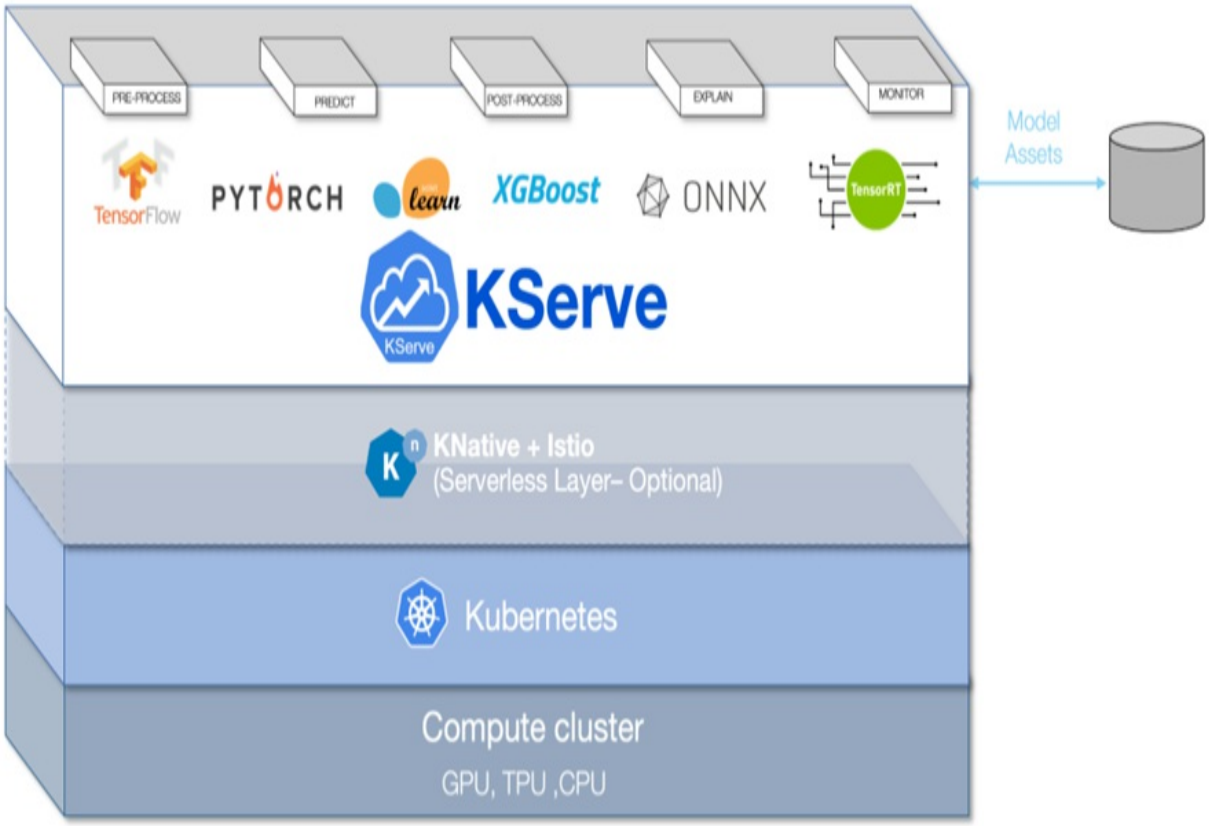
Figure 8.4 Screenshot of Katib UI that performs experiment tracking. Source: Kubeflow, licensed under Creative Commons Attribution 4.0 International.



KServe (https://github.com/kserve/kserve) was born as part of the Kubeflow

project and was previously as KFServing. KServe provides a Kubernetes Custom Resource Definition for serving machine learning models on arbitrary frameworks. It aims to solve production model serving use cases by providing performant, high abstraction interfaces for common ML frameworks. It encapsulates the complexity of autoscaling, networking, health checking, and server configuration to bring cutting-edge serving features like GPU Autoscaling, Scale to Zero, and Canary Rollouts to your ML deployments. Figure 8.5 is a diagram that illustrates the position of KServe in the ecosystem.

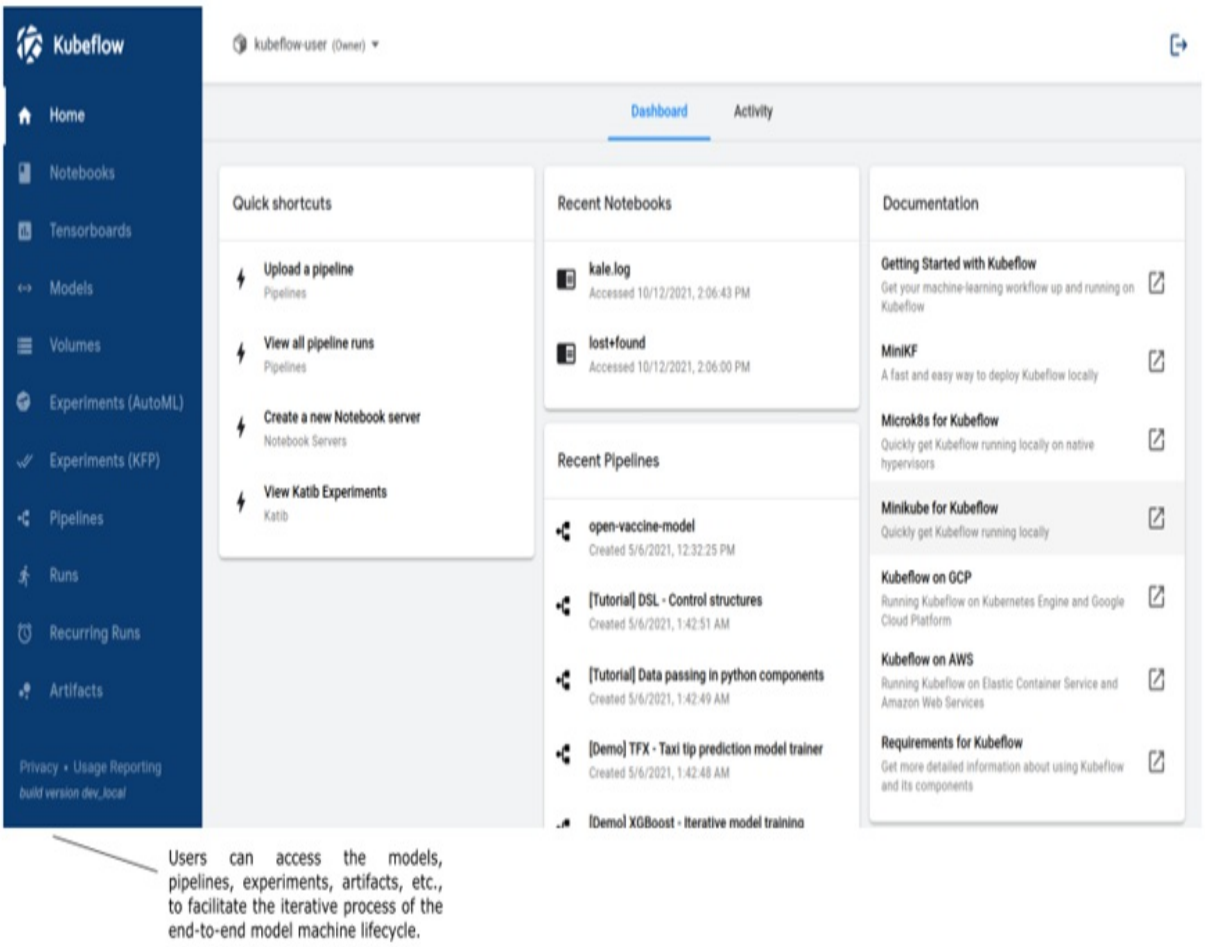
Figure 8.5 KServe positioning in the ecosystem. Source: KServe, licensed under Apache License 2.0.



Kubeflow provides web UI. A screenshot of the UI can be found in Figure 8.6. Users can access the models, pipelines, experiments, artifacts, etc., to facilitate the iterative process of the end-to-end model machine lifecycle in each of the tabs on the left side.

The web UI is integrated with Jupyter Notebooks to be easily accessible. There are also SDKs in different languages to help users integrate with any internal systems. In addition, users can interact with all the Kubeflow components via kubectl since they are all native Kubernetes custom resources and controllers.

Figure 8.6 Screenshot of Kubeflow UI. Source: Kubeflow, licensed under Creative Commons Attribution 4.0 International.



The training operator (https://github.com/kubeflow/training-operator) provides Kubernetes custom resources that make it easy to run distributed or non-distributed TensorFlow, PyTorch, Apache MXNet, XGBoost, or MPI jobs on Kubernetes. We will be using it in this section for submitting distributed model training.

The Kubeflow project has accumulated more than 500 contributors and 20k GitHub stars. It's heavily adopted in various companies and has more than 10 vendors, such as AWS, Azure, Google Cloud, IBM, etc. Seven working groups maintain different subprojects independently. We will be using the training operator to submit distributed model training jobs and KServe to build our model serving component. Once you complete the next chapter, it's recommended to try out the other subprojects in the Kubeflow ecosystem on your own when needed. For example, if you'd like to tune the performance of the model, Katib can be used to leverage its AutoML and hyperparameter tuning functionalities.

8.3.1 Basics

Next, we'll take a closer look at the distributed training operator of Kubeflow and submit a distributed model training job that runs locally in the K3s local cluster we created in the previous section.

Let's first create and activate a dedicated "kubeflow" namespace for our examples and reuse the existing cluster we created earlier:

Listing 8.31 Create and switch to a new namespace

```
> kubectl create ns kubeflow
> kns kubeflow
```

Then we go back to our project folder and apply all the manifests to install all the tools we need:

Listing 8.32 Apply all manifests and install all the tools

```
> cd code/project
> kubectl kustomize manifests | k apply -f -
```

Note that we've bundled all the necessary tools in this "manifests" folder, including

- Kubeflow Training Operator that we will use for this chapter for distributed model training;

- Argo Workflows (https://github.com/argoproj/argo-workflows) for later chapters when we discuss workflow orchestration and chain all the components together in a machine learning pipeline. We can ignore this for now.

As introduced earlier, the Kubeflow Training Operator provides Kubernetes custom resources that make it easy to run distributed or non-distributed jobs on Kubernetes, including TensorFlow, PyTorch, Apache MXNet, XGBoost, MPI jobs, etc.

Before we dive into Kubeflow, we need to understand what *custom resources* are. A custom resource is an extension of the Kubernetes API that is not necessarily available in a default Kubernetes installation. It is a customization of a particular Kubernetes installation. However, many core Kubernetes functions are now built using custom resources, making Kubernetes more modular.

https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/

Custom resources can appear and disappear in a running cluster through dynamic registration, and cluster admins can update custom resources independently of the cluster. Once a custom resource is installed, users can create and access its objects using kubectl, just as they do for built-in resources like pods.

For example, below is a definition for the TFJob custom resource that allows us to instantiate and submit a distributed TensorFlow training job to the Kubernetes cluster.

Listing 8.33 TFJob CRD

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  annotations:
       controller-gen.kubebuilder.io/version: v0.4.1
  name: tfjobs.kubeflow.org
spec:
  group: kubeflow.org
```

```
  names:
       kind: TFJob
       listKind: TFJobList
       plural: tfjobs
       singular: tfjob
```

All instantiated TFJob custom resource objects (TFJobs) will be handled by the training operator. Below is the definition of the deployment of the training operator that runs a stateful controller to continuously monitor and process any submitted TFJobs.

Listing 8.34 Training operator deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: training-operator
  labels:
    control-plane: kubeflow-training-operator
spec:
  selector:
    matchLabels:
      control-plane: kubeflow-training-operator
  replicas: 1
  template:
    metadata:
      labels:
        control-plane: kubeflow-training-operator
      annotations:
        sidecar.istio.io/inject: "false"
    spec:
      containers:
        - command:
            - /manager
          image: kubeflow/training-operator
          name: training-operator
          env:
            - name: MY_POD_NAMESPACE
              valueFrom:
                fieldRef:
                  fieldPath: metadata.namespace
            - name: MY_POD_NAME
              valueFrom:
                fieldRef:
                  fieldPath: metadata.name
          securityContext:
```

```
            allowPrivilegeEscalation: false
          livenessProbe:
            httpGet:
              path: /healthz
              port: 8081
            initialDelaySeconds: 15
            periodSeconds: 20
          readinessProbe:
            httpGet:
              path: /readyz
              port: 8081
            initialDelaySeconds: 5
            periodSeconds: 10
          resources:
            limits:
              cpu: 100m
              memory: 30Mi
            requests:
              cpu: 100m
              memory: 20Mi
      serviceAccountName: training-operator
      terminationGracePeriodSeconds: 10
```

With this abstraction, data science teams can focus on writing the Python code in TensorFlow that will be used as part of a TFJob specification and don't have to manage the infrastructure themselves. For now, we can skip the low-level details and use this to implement our distributed model training.

Next, let's define our TFJob in a file named "tfjob.yaml":

Listing 8.35 Example TFJob definition

```
apiVersion: kubeflow.org/v1
kind: TFJob
metadata:
  namespace: kubeflow
  generateName: distributed-tfjob-
spec:
  tfReplicaSpecs:
    Worker:
      replicas: 2
      restartPolicy: OnFailure
      template:
        spec:
          containers:
```

```
            - name: tensorflow
              image: gcr.io/kubeflow-ci/tf-mnist-with-summaries:1
              command:
                - "python"
                - "/var/tf_mnist/mnist_with_summaries.py"
                - "--log_dir=/train/metrics"
                - "--learning_rate=0.01"
                - "--batch_size=100"
```

In the above spec, we are asking the controller to submit a distributed TensorFlow model training model with two worker replicas where each worker replica follows the same container definition, running the MNIST image classification example.

Once it's defined, we can submit it to our local K8s cluster via the following:

Listing 8.36 Submit the TFJob

```
> kubectl create -f basics/tfjob.yaml
tfjob.kubeflow.org/distributed-tfjob-qc8fh created
```

We can see whether the TFJob has been submitted successfully:

Listing 8.37 Get the list of TFJobs

```
> kubectl get tfjob

NAME                          AGE
Distributed-tfjob-qc8fh       1s
```

When we get the list of pods, we will see two worker pods, "distributed-tfjob-qc8fh-worker-1" and "distributed-tfjob-qc8fh-worker-0" have been created and started running. The other pods can be ignored since they are the pods that are running the Kubeflow and Argo Workflow operators.

Listing 8.38 Get the list of pods

```
> kubectl get pods

NAME                                    READY   STATUS   RESTAR
workflow-controller-594494ffbd-2dpkj  1/1        Running   0
training-operator-575698dc89-mzvwb        1/1        Running   0
```

```
argo-server-68c46c5c47-vfh82              1/1        Running   0
distributed-tfjob-qc8fh-worker-1          1/1        Running   0
distributed-tfjob-qc8fh-worker-0          1/1        Running   0
```

A machine learning system consists many different components. We only used Kubeflow to submit distributed model training jobs but it's not connected to other components yet. In the next section, we'll explore the basic functionalities of Argo Workflows to connect different steps in a single workflow so that they can be executed in a particular order.

8.3.2 Exercises

1. If your model training requires parameter servers, can you express that in a TFJob?

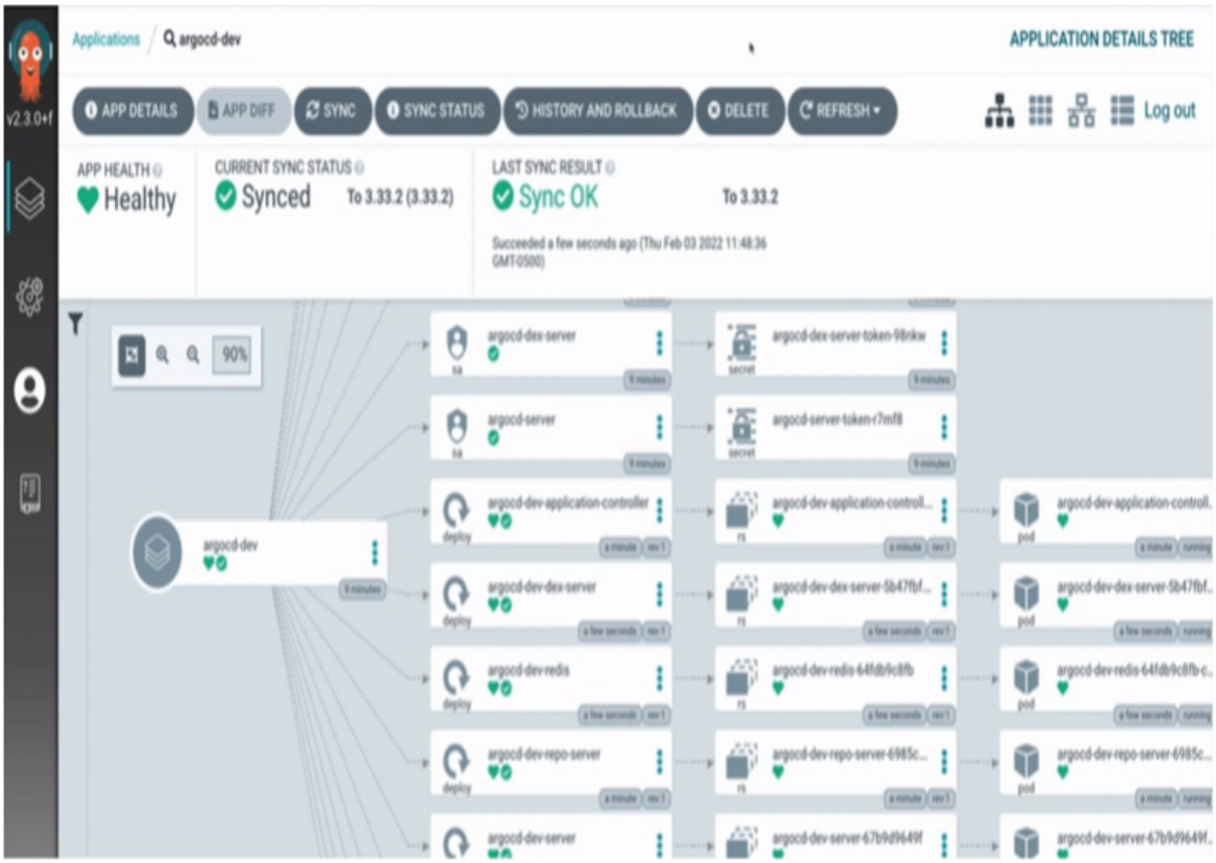
8.4 Argo Workflows: Container-native workflow engine

The Argo Project is a suite of open-source tools for deploying and running applications and workloads on Kubernetes. It extends the Kubernetes APIs and unlocks new and powerful capabilities in application deployment, container orchestration, event automation, progressive delivery, and more. It consists of four core projects: Argo CD, Argo Rollouts, Argo Events, and Argo Workflows. Besides these core projects, many other ecosystem projects are based on Argo, extend Argo, or work well with Argo. A complete list of resources related to Argo can be found here: https://github.com/terrytangyuan/awesome-argo.

Argo CD is a declarative, GitOps application delivery tool for Kubernetes. It manages application definitions, configurations, and environments declaratively in Git. Argo CD user experience makes Kubernetes application deployment and lifecycle management automated, auditable, and easy to grasp. It comes with a UI so that engineers can look at the UI to see what's happening in their clusters and watch for applications deployments, etc. Figure 8.7 is a screenshot of the resource tree in the Argo CD UI.

Figure 8.7 Screenshot of resource tree in Argo CD UI. Source: Argo, licensed under Apache

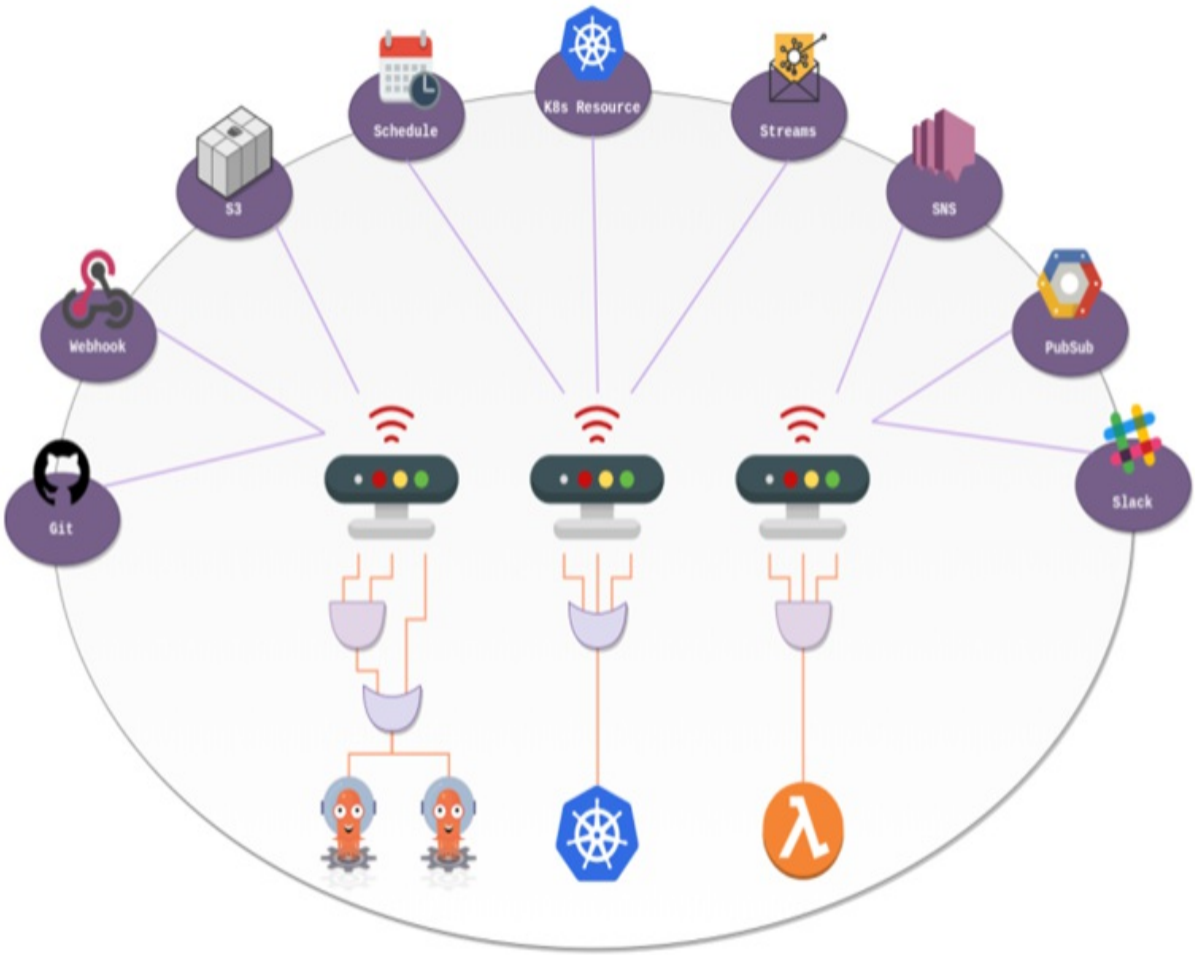
License 2.0.



Argo Rollouts is a Kubernetes controller and set of CRDs which provides progressive deployment capabilities. It introduces blue-green and canary deployments, canary analysis, experimentation, and progressive delivery features to your Kubernetes cluster.

Next is Argo Events. It's an event-based dependency manager for Kubernetes. It can define multiple dependencies from various event sources like webhooks, S3, schedules, streams, and trigger Kubernetes objects after successful event dependencies resolution. A complete list of available event sources can be found in Figure 8.8.

Figure 8.8 Available event sources in Argo Events. Source: Argo, licensed under Apache License 2.0.



Last but not least, Argo Workflows is a container-native workflow engine for orchestrating parallel jobs, implemented as Kubernetes CRD. Users can define workflows where each step is a separate container, model multi-step workflows as a sequence of tasks or capture the dependencies between tasks using a graph, and run compute-intensive jobs for machine learning or data processing. It's worth mentioning that users often use Argo Workflows with Argo Events together to trigger event-based workflows. The main use cases for Argo Workflows are machine learning pipelines, data processing, ETL, infrastructure automation, continuous delivery, and integration.

Argo Workflows also provides interfaces such as CLI, server, UI, and SDKs for different languages. The CLI is useful for managing workflows and performing operations such as submitting, suspending, and deleting workflows through the command line. The server is used for integrating with other services. There are both rest and gRPC service interfaces. The UI is

useful for managing and visualizing workflows and any artifacts/logs created by the workflows, as well as other useful information, such as resource usage analytics.

We will walk through some examples of Argo Workflows to prepare for our project.

8.4.1 Basics

Before we dive into the examples, let's make sure we have the Argo Workflows UI at hand. It's optional since you can still be successful in these examples in the command line to interact directly with Kubernetes via kubectl, but it's nice to see the DAG visualizations in the UI as well as access additional functionalities.

By default, the Argo Workflows UI service is not exposed to an external IP. To access the UI, use the following method:

Listing 8.39 Port-forward Argo Server

```
> kubectl port-forward svc/argo-server 2746:2746
```

Then visit the following URL to access the UI: https://localhost:2746

Alternatively, you can expose a load balancer to get an external IP to access the Argo Workflows UI in your local cluster. Check out the official documentation for more details: https://argoproj.github.io/argo-workflows/argo-server/

Figure 8.9 is a screenshot of what the Argo Workflows UI looks like for a map-reduce style workflow.

Figure 8.9 Argo Workflows UI that illustrates a map-reduce style workflow. Source: Argo, licensed under Apache License 2.0.

Here's a basic hello world example of Argo Workflows. We can specify the container image and the command to run for this workflow and print out a hello world message.

Listing 8.40 Hello-world example

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: hello-world-
spec:
  entrypoint: whalesay
  serviceAccountName: argo
  templates:
  - name: whalesay
    container:
      image: docker/whalesay
```

```
      command: [cowsay]
      args: ["hello world"]
```

Let’s go ahead and submit the workflow to our cluster:

Listing 8.41 Submit the workflow

```
> kubectl create -f basics/argo-hello-world.yaml
workflow.argoproj.io/hello-world-zns4g created
```

We can check if it’s submitted successfully and started running:

Listing 8.42 Get the list of workflows

```
> kubectl get wf

NAME                STATUS    AGE
hello-world-zns4g   Running   2s
```

Once the workflow status has changed to “succeeded,” we can check the statuses of the pods created by the workflow.

First, let’s find all the pods associated with the workflow. We can use a label selector to get the list of pods:

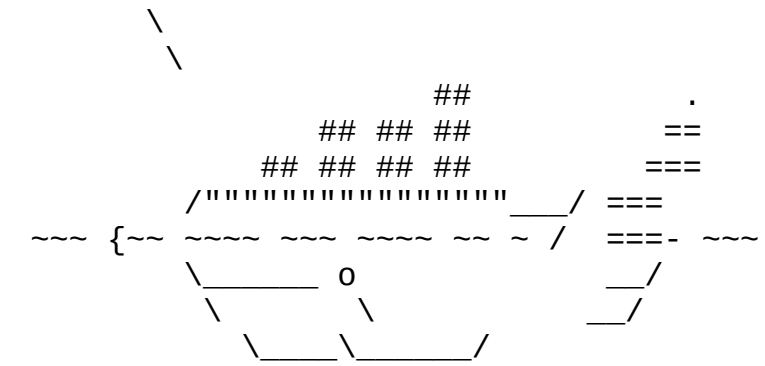
Listing 8.43 Get the list of pods that belong to this workflow

```
> kubectl get pods -l workflows.argoproj.io/workflow=hello-world-

NAME                READY   STATUS     RESTARTS   AGE
hello-world-zns4g   0/2     Completed  0          8m57s
```

Once we know the pod name, we can get the logs of that pod:

Listing 8.44 Check the pod logs

```
> kubectl logs hello-world-zns4g -c main

 _____________
< hello world >
 -------------
       \
```



As expected, we get the same logs as the ones we had with the simple Kubernetes pod in the previous sections since this workflow only runs one hello-world step.

The next example uses a resource template where you can specify a Kubernetes custom resource that will be submitted by the workflow to the Kubernetes cluster. Here we are creating a Kubernetes config map named “cm-example” with a simple key-value pair. Config map is a Kubernetes-native object to store key-value pairs.

Listing 8.45 Resource template

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: k8s-resource-
spec:
  entrypoint: k8s-resource
  serviceAccountName: argo
  templates:
  - name: k8s-resource
    resource:
       action: create
       manifest: |
         apiVersion: v1
         kind: ConfigMap
         metadata:
             name: cm-example
         data:
             some: value
```

This example is perhaps the most useful to Python users. You can write a Python script as part of the template definition. Here we are generating some

random numbers using the built-in random Python module. Alternatively, you can specify this as a container template, as seen in the hello world example.

Listing 8.46 Script template

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: script-tmpl-
spec:
  entrypoint: gen-random-int
  serviceAccountName: argo
  templates:
  - name: gen-random-int
    script:
       image: python:alpine3.6
       command: [python]
       source: |
          import random
          i = random.randint(1, 100)
          print(i)
```

Let’s submit it:

Listing 8.47 Submit the script template workflow

```
> kubectl create -f basics/argo-script-template.yaml
workflow.argoproj.io/script-tmpl-c5lhb created
```

And check its logs to see if a random number is generated:

Listing 8.48 Check the pod logs

```
> kubectl logs script-tmpl-c5lhb
25
```

So far, we’ve only seen examples of single-step workflows. Argo Workflow also allows users to define the workflow as a directed-acyclic graph (DAG) by specifying the dependencies of each task. This can be simpler to maintain for complex workflows and allows for maximum parallelism when running tasks.

Let’s look at an example of a diamond-shaped DAG created by Argo Workflows.

Here this DAG consists of four steps, A, B, C, and D, and each has its dependencies. For example, step C depends on step A, and step D depends on steps B and C.

Listing 8.49 Diamond example using DAG

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: dag-diamond-
spec:
  serviceAccountName: argo
  entrypoint: diamond
  templates:
  - name: echo
    inputs:
      parameters:
        - name: message
    container:
      image: alpine:3.7
      command: [echo, "{{inputs.parameters.message}}"]
  - name: diamond
    dag:
      tasks:
          - name: A
           template: echo
           arguments:
               parameters: [{name: message, value: A}]
          - name: B
            dependencies: [A]
            template: echo
            arguments:
               parameters: [{name: message, value: B}]
          - name: C
            dependencies: [A]
            template: echo
           arguments:
                 parameters: [{name: message, value: C}]
          - name: D
           dependencies: [B, C]
           template: echo
           arguments:
```

```
                parameters: [{name: message, value: D}]
```

Let’s submit it:

Listing 8.50 Submit the DAG workflow

```
> kubectl create -f basics/argo-dag-diamond.yaml
workflow.argoproj.io/dag-diamond-6swfg created
```

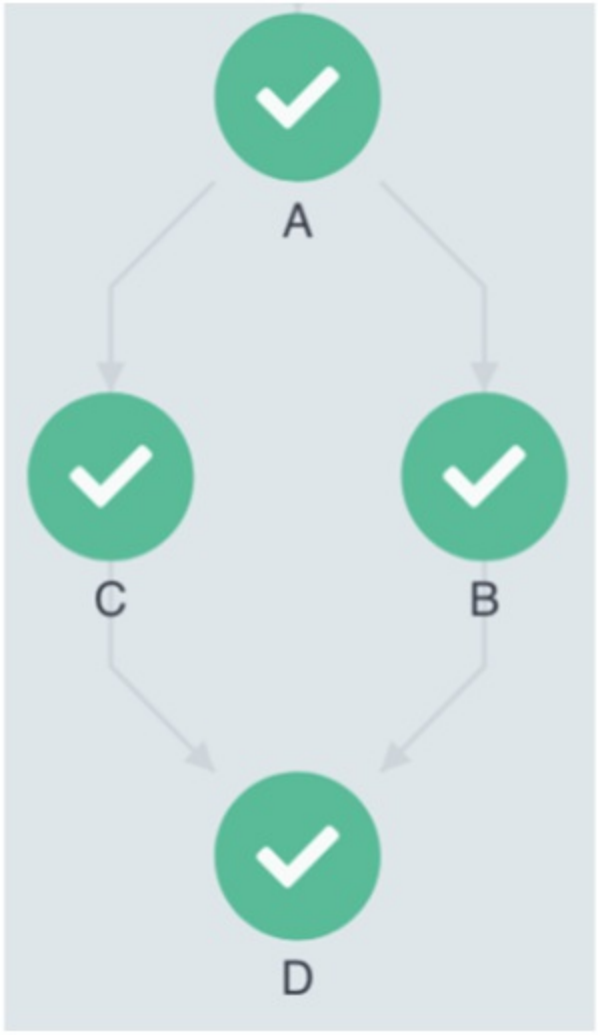
When the workflow is completed, we will see four pods for each of the steps where each step prints out its step name, A, B, C, and D.

Listing 8.51 Get the list of pods that belong to this workflow

```
> kubectl get pods -l workflows.argoproj.io/workflow=dag-diamond-

NAME                                    READY   STATUS     RESTA
dag-diamond-6swfg-echo-4189448097       0/2        Completed   0
dag-diamond-6swfg-echo-4155892859       0/2        Completed   0
dag-diamond-6swfg-echo-4139115240       0/2        Completed   0
dag-diamond-6swfg-echo-4239780954       0/2        Completed   0
```

The visualization of the DAG is available in the Argo Workflows UI. It’s usually more intuitive to see how the workflow is executed in a diamond-shaped flow in the UI, as seen in Figure 8.10.

Figure 8.10 Screenshot of a diamond-shaped workflow in the UI.



Next, we will look at a simple coin flip example to showcase the conditional syntax provided by Argo Workflows. We can specify a condition to indicate whether we want to run the next step or not. For example, we run the flip coin step first, which is the python script we saw earlier, and if the result returns heads, then we run the template called heads which also prints another log saying it was “head.” Otherwise, we print that it was tails. So we can specify these conditionals inside the when clause in the different steps.

Listing 8.52 Coin-flip example

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: coinflip-
```

```
spec:
  serviceAccountName: argo
  entrypoint: coinflip
  templates:
  - name: coinflip
    steps:
    - - name: flip-coin
        template: flip-coin
    - - name: heads
        template: heads
        when: "{{steps.flip-coin.outputs.result}} == heads"
      - name: tails
        template: tails
        when: "{{steps.flip-coin.outputs.result}} == tails"

  - name: flip-coin
    script:
      image: python:alpine3.6
      command: [python]
      source: |
        import random
        result = "heads" if random.randint(0,1) == 0 else "tails"
        print(result)

  - name: heads
    container:
      image: alpine:3.6
      command: [sh, -c]
      args: ["echo \"it was heads\""]

  - name: tails
    container:
      image: alpine:3.6
      command: [sh, -c]
      args: ["echo \"it was tails\""]
```

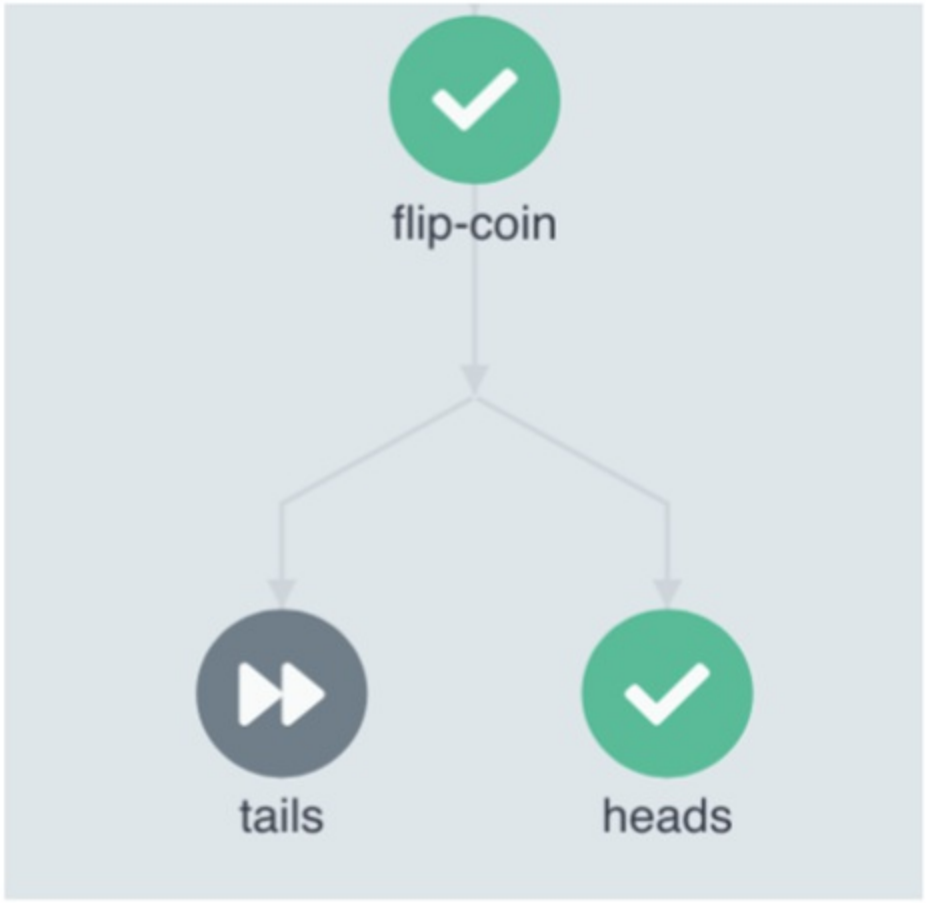
Let’s submit the workflow:

Listing 8.53 Flip-coin example

```
> kubectl create -f basics/argo-coinflip.yaml
workflow.argoproj.io/coinflip-p87ff created
```

Figure 8.11 is a screenshot of what this flip-coin workflow looks like in the UI:

Figure 8.11 Screenshot of flip-coin workflow in the UI.



When we got the list of workflows, only two pods were found:

Listing 8.54 Get the list of pods that belong to this workflow

```
> kubectl get pods -l workflows.argoproj.io/workflow=coinflip-p87

coinflip-p87ff-flip-coin-1071502578   0/2        Completed   0
coinflip-p87ff-tails-2208102039          0/2        Completed   0
```

We can check the logs of the “flip-coin” step to see if it prints out “tails” since the next step executed is the “tails” step:

```
> kubectl logs coinflip-p87ff-flip-coin-1071502578
tails
```

That’s a wrap! We’ve just learned the basic syntax of Argo Workflows,

which should cover all the prerequisites for the next chapter! In the next chapter, we will use Argo Workflows to implement the end-to-end machine learning workflow that consists of the actual system components introduced in Chapter 7.

8.4.2 Exercises

1. Besides accessing the output of each step like {{steps.flip-coin.outputs.result}}, what are other available variables?
2. Can you trigger workflows automatically by Git commits or other events?

8.5 References

- Namespaces: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/

8.6 Summary

- We used TensorFlow to train a machine learning model for the MNIST dataset in a single machine.
- We learned the basic concepts in Kubernetes and gained hands-on experience by implementing them in a local Kubernetes cluster.
- We submitted distributed model training jobs to Kubernetes via Kubeflow.
- We learned about different types of templates and how to define either DAGs or sequential steps using Argo Workflows.

Answers to Exercises:

Section 8.1

1. Yes, via model = tf.keras.models.load_model('my_model.h5'); modele.evaluate(x_test, y_test)
2. You should be able to do it easily by changing the tuner to use kt.RandomSearch(model_builder).

Section 8.2

1. kubectl get pod <pod-name> -o json
2. Yes, you can define additional containers in the pod.spec.containers in addition to the existing single container.

Section 8.3

1. Similar to “worker” replicas, define “parameterServer” replicas in your TFJob spec to specify the number of parameter servers.

Section 8.4

1. The complete list is available here: https://github.com/argoproj/argo-workflows/blob/master/docs/variables.md
2. Yes, you can use Argo Events to watch Git events and trigger workflows.

9 A complete implementation

This chapter covers

- Implementing data ingestion component with TensorFlow
- Defining the machine learning model and submitting distributed model training jobs
- Implementing single-instance model server as well as replicated model servers
- Building an efficient end-to-end workflow of our machine learning system

In the previous chapter of the book, we learned the basics of the four core technologies that we will be using in our project: TensorFlow, Kubernetes, Kubeflow, and Argo Workflows. We learned TensorFlow to perform data processing, model building, and model evaluation. We also learned the basic concepts of Kubernetes and started our local Kubernetes cluster that will be used as our core distributed infrastructure. In addition, we successfully submitted distributed model training jobs to the local Kubernetes cluster using Kubeflow. At the end of the last chapter, we learned how to use Argo Workflows to construct and submit basic hello-world workflow and complex DAG-structured workflow.

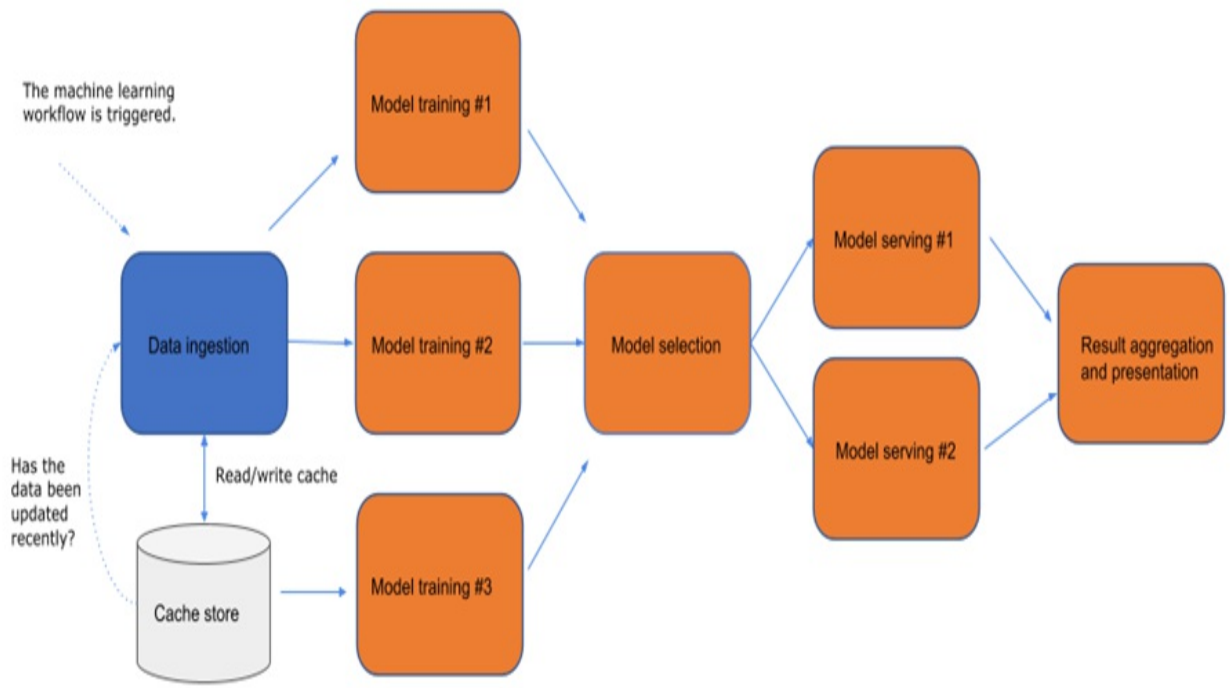
In this last chapter, we'll implement the end-to-end machine learning system with the architecture we designed in Chapter 7. We will provide complete implementation to each of the components that incorporate the patterns that we mentioned previously. We'll use several popular frameworks and cutting-edge technologies, particularly TensorFlow, Kubernetes, Kubeflow, Docker, and Argo Workflows, that we introduced in Chapter 8 to build different components of a distributed machine learning workflow in this chapter.

9.1 Data ingestion

The first component in our end-to-end workflow is data ingestion. We'll be

using the Fashion-MNIST dataset introduced in Section 2.2.1 to build the data ingestion component. Figure 9.1 shows this component in the dark box on the left of the end-to-end workflow.

Figure 9.1 Data ingestion component (dark box) in the end-to-end machine learning system.



Recall that this dataset consists of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image that represents one Zalando's article image, associated with a label from 10 classes. In addition, the Fashion-MNIST dataset is designed to serve as a direct drop-in replacement for the original MNIST dataset for benchmarking machine learning algorithms. It shares the same image size and structure of training and testing splits. Figure 9.2 is a screenshot of the collection of images for all ten classes (t-shirt/top, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag, and ankle boot) from Fashion-MNIST, where each class takes three rows in the screenshot.

Figure 9.2 Screenshot of the collection of images from the Fashion-MNIST dataset for all ten classes (t-shirt/top, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag, and ankle boot). Source: https://github.com/zalandoresearch/fashion-mnist.

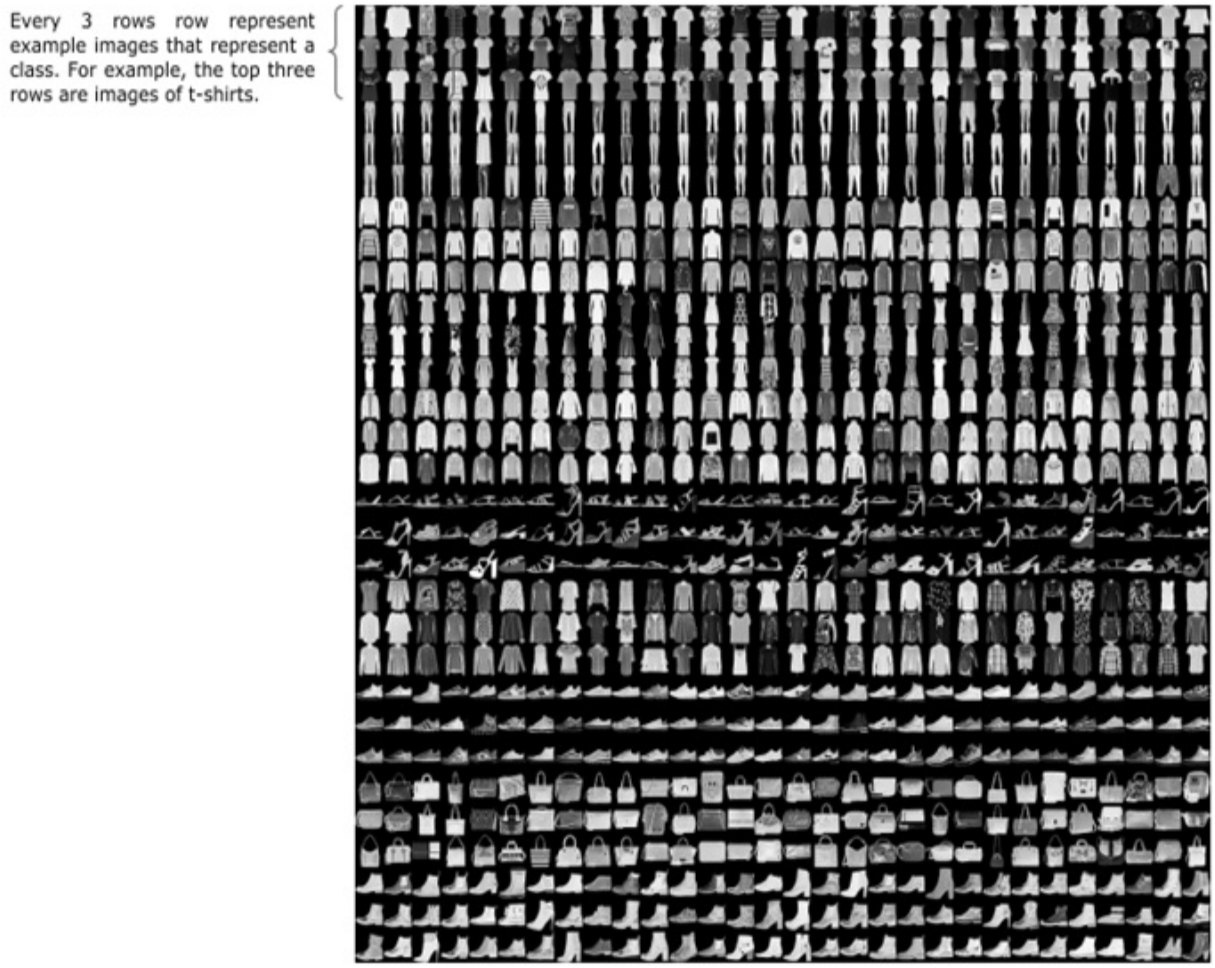
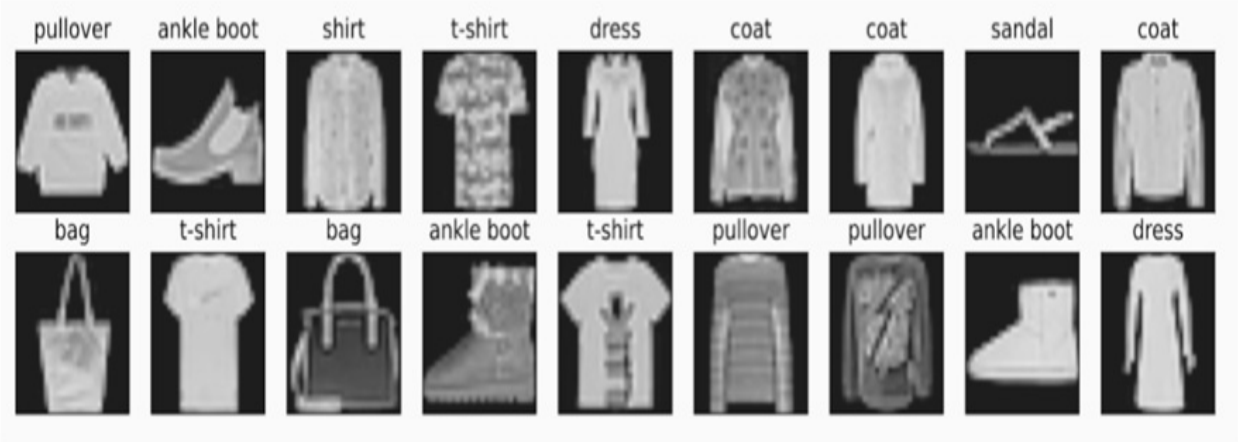


Figure 9.3 is a closer look at the first few example images in training set together with their corresponding labels in text right above each of the images.

Figure 9.3 A closer look at the first few example images. In training set with their corresponding labels in text.



In section 9.1.1, we'll go through the implementation of single-node data pipeline that ingests the Fashion-MNIST dataset. Furthermore, Section 9.1.2 will cover the implementation of the distributed data pipeline to prepare the data for our distributed model training in Section 9.2.

9.1.1 Single-node data pipeline

Let's first take a look at how to build a single-node data pipeline that works locally on your laptop without using a local Kubernetes cluster.

The best way for a machine learning program written in TensorFlow to consume data is through methods in tf.data module. The tf.data API allows users to build complex input pipelines easily. For example, the pipeline for an image model might aggregate data from files in various file systems, apply random transformations to each image, and create batches from the images for model training.

The tf.data API enables it to handle large amounts of data, read from different data formats, and perform complex transformations. It contains a tf.data.Dataset abstraction that represents a sequence of elements, in which each element consists of one or more components. Let's use the image pipeline to illustrate this. An element in an image input pipeline might be a single training example, with a pair of tensor components representing the image and its label.

Below is the code snippet to load the Fashion-MNIST dataset into a

tf.data.Dataset object and performs some necessary preprocessing steps to prepare for our model training:

1. Scale the dataset from the range (0, 255] to (0., 1.]
2. Cast the image multi-dimensional arrays into float32 type that our model can accept
3. Select the training data, cache it in memory to speed up training, and shuffle it with a buffer size of 10,000

Listing 9.1 Load the Fashion-MNIST dataset

```
import tensorflow_datasets as tfds
import tensorflow as tf
def make_datasets_unbatched():
  def scale(image, label):
    image = tf.cast(image, tf.float32)
    image /= 255
    return image, label
  datasets, _ = tfds.load(name='fashion_mnist', with_info=True, a
  return datasets['train'].map(scale).cache().shuffle(10000)
```

Note that we imported tensorflow_datasets module. It is the TensorFlow Datasets which consists of a collection of datasets for various tasks such as image classification, object detection, document summarization, etc., all ready to use with TensorFlow and other Python machine learning frameworks.

The tf.data.Dataset object is a shuffled dataset where each element consists of the images and their labels with the shape and data type information as follows:

Listing 9.2 Inspect the tf.data object

```
>>> ds = make_datasets_unbatched()
>>> ds
<ShuffleDataset element_spec=(TensorSpec(shape=(28, 28, 1), dtype
```

9.1.2 Distributed data pipeline

Now let’s look at how we can consume our dataset in a distributed fashion.

We'll be using tf.distribute.MultiWorkerMirroredStrategy for distributed training in the next section. Let's assume we have instantiated a strategy object. Inside the strategy's scope via Python's "with" syntax, we will instantiate our dataset using the same function we previously defined for the single-node use case.

We will need to tweak a few configurations to build our distributed input pipeline. First, we create repeated batches of data where the total batch size equals the batch size per replica times the number of replicas over which gradients are aggregated. This ensures that we will have enough records to train each batch in each of the model training workers. In other words, the number of replicas in sync equals the number of devices taking part in the gradient allreduce operation during model training. For instance, When a user or the training code calls next() on the distributed data iterator, a per replica batch size of data is returned on each replica. The rebatched dataset cardinality will always be a multiple of the number of replicas.

In addition, we want to configure tf.data to enable automatic data sharding. Since the dataset is in the distributed scope, the input dataset will be sharded automatically in multi-worker training mode. More specifically, each dataset will be created on the CPU device of the worker, and each set of workers will train the model on a subset of the entire dataset when the tf.data.experimental.AutoShardPolicy is set to be AutoShardPolicy.DATA. One benefit of this is that during each model training step, a global batch size of non-overlapping dataset elements will be processed by each worker. Each worker will process the whole dataset and discard the portion that is not for itself. Note that for this mode to partition the dataset elements correctly, the dataset needs to produce elements in a deterministic order, which should already be guaranteed by the TensorFlow Datasets library we use.

Listing 9.3 Configure distributed data pipeline

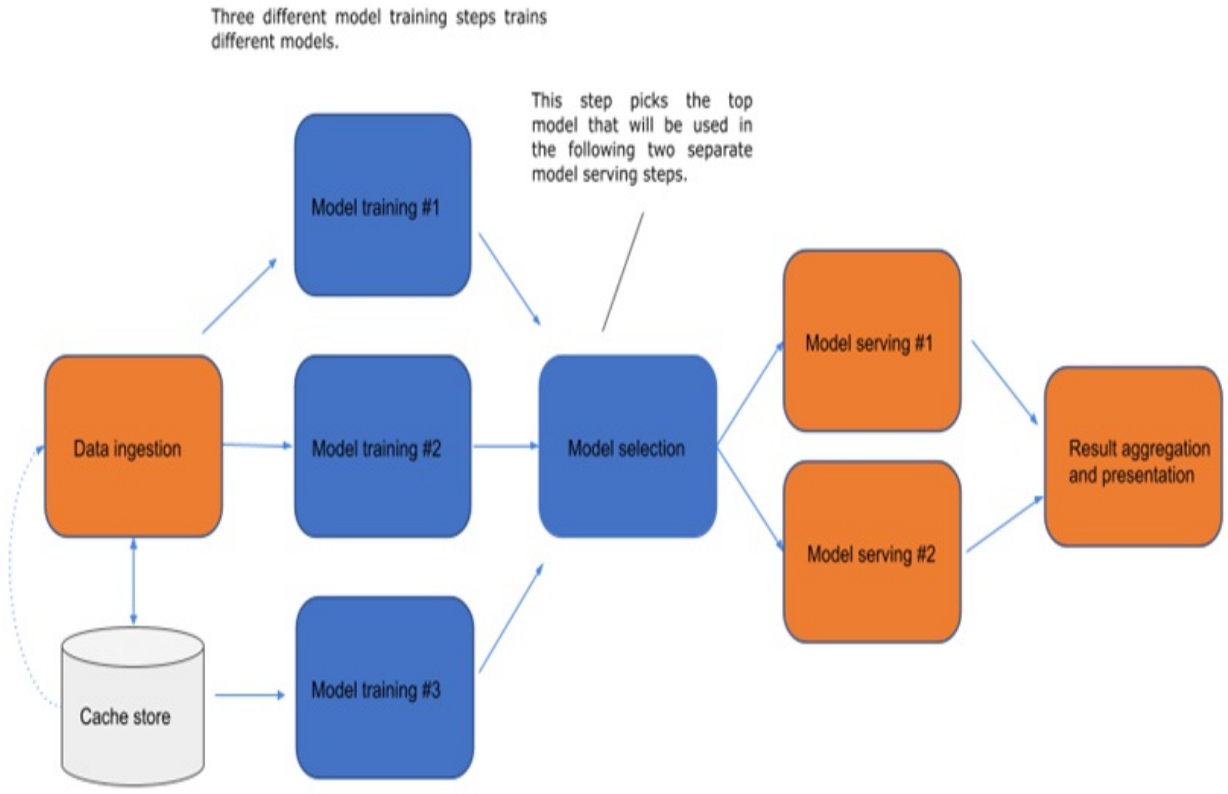
```
BATCH_SIZE_PER_REPLICA = 64
BATCH_SIZE = BATCH_SIZE_PER_REPLICA * strategy.num_replicas_in_sy
with strategy.scope():
  ds_train = make_datasets_unbatched().batch(BATCH_SIZE).repeat()
  options = tf.data.Options()
  options.experimental_distribute.auto_shard_policy = tf.data.exp
  ds_train = ds_train.with_options(options)
```

```
  model = build_and_compile_model()
model.fit(ds_train, epochs=1, steps_per_epoch=70)
```

9.2 Model training

We went through the implementation of the data ingestion component for both local-node and distributed data pipelines and discussed how we can shard the dataset properly across different workers so that it would work with distributed model training. In this section, let's dive into the implementation details for our model training component.

Figure 9.4 is a diagram of the model training component in the overall architecture for us to recap. In the diagram, there are three different model training steps followed by a model selection step. These model training steps would train three different models, namely, CNN, CNN with dropout, and CNN with batch normalization, competing with each other for better statistical performance.



We will learn how to define those three models with TensorFlow in Section 9.2.1 and execute the distributed model training jobs with Kubeflow in

Section 9.2.2. In Section 9.2.3, we will implement the model selection step that picks the top model that will be used in the model serving component in our end-to-end machine learning workflow.

9.2.1 Model definition and single-node training

Below is the TensorFlow code to define and initialize the first model, a Convolutional Neural Network (CNN) model we introduced in previous chapters with three convolutional layers. We initialize the model with Sequential(), meaning we'll add the layers sequentially. The first layer is the input layer, where we specify the shape of the input pipeline that we defined previously. Note that we also explicitly give a name to the input layer so we can pass the correct key in our inference inputs, which we will discuss in more depth in Section 9.3.

After adding the input layer, three convolutional layers, followed by max-pooling layers and dense layers, are added to the sequential model. We'll then print out a summary of the model architecture and compile the model with Adam as its optimizer, accuracy as the metric we use to evaluate the model, and sparse categorical cross-entropy as the loss function.

Listing 9.4 Define the basic CNN model

```
def build_and_compile_cnn_model():
  print("Training CNN model")
  model = models.Sequential()
  model.add(layers.Input(shape=(28, 28, 1), name='image_bytes'))
  model.add(
              layers.Conv2D(32, (3, 3), activation='relu'))
  model.add(layers.MaxPooling2D((2, 2)))
  model.add(layers.Conv2D(64, (3, 3), activation='relu'))
  model.add(layers.MaxPooling2D((2, 2)))
  model.add(layers.Conv2D(64, (3, 3), activation='relu'))
  model.add(layers.Flatten())
  model.add(layers.Dense(64, activation='relu'))
  model.add(layers.Dense(10, activation='softmax'))
  model.summary()
  model.compile(optimizer='adam',
                loss='sparse_categorical_crossentropy',
                metrics=['accuracy'])
  return model
```

We've successfully defined our basic CNN model. Next, we define two models based on the CNN model. One adds a batch normalization layer to force the pre-activations to have zero mean and unit standard deviation for every neuron (activation) in a particular layer. The other with an additional dropout layer where half of the hidden units will be dropped randomly to reduce the complexity of the model and speeds up computation. The rest of the code is the same as the basic CNN model.

Listing 9.5 Define the variations of the basic CNN model

```
def build_and_compile_cnn_model_with_batch_norm():
  print("Training CNN model with batch normalization")
  model = models.Sequential()
  model.add(layers.Input(shape=(28, 28, 1), name='image_bytes'))
  model.add(
              layers.Conv2D(32, (3, 3), activation='relu'))
  model.add(layers.BatchNormalization())
  model.add(layers.Activation('sigmoid'))
  model.add(layers.MaxPooling2D((2, 2)))
  model.add(layers.Conv2D(64, (3, 3), activation='relu'))
  model.add(layers.BatchNormalization())
  model.add(layers.Activation('sigmoid'))
  model.add(layers.MaxPooling2D((2, 2)))
  model.add(layers.Conv2D(64, (3, 3), activation='relu'))
  model.add(layers.Flatten())
  model.add(layers.Dense(64, activation='relu'))
  model.add(layers.Dense(10, activation='softmax'))
]
  model.summary()

  model.compile(optimizer='adam',
                loss='sparse_categorical_crossentropy',
                metrics=['accuracy'])
  return model

def build_and_compile_cnn_model_with_dropout():
  print("Training CNN model with dropout")
  model = models.Sequential()
  model.add(layers.Input(shape=(28, 28, 1), name='image_bytes'))
  model.add(
              layers.Conv2D(32, (3, 3), activation='relu'))
  model.add(layers.MaxPooling2D((2, 2)))
  model.add(layers.Conv2D(64, (3, 3), activation='relu'))
  model.add(layers.MaxPooling2D((2, 2)))
  model.add(layers.Dropout(0.5))
```

```
  model.add(layers.Conv2D(64, (3, 3), activation='relu'))
  model.add(layers.Flatten())
  model.add(layers.Dense(64, activation='relu'))
  model.add(layers.Dense(10, activation='softmax'))
  model.summary()

  model.compile(optimizer='adam',
                loss='sparse_categorical_crossentropy',
                metrics=['accuracy'])
  return model
```

Once the models are defined, we can train them locally on our laptops. Let's use the basic CNN model as an example. We will create three callbacks that will be executed during model training:

1. PrintLR callback print the learning rate at the end of each epoch;
2. TensorBoard callback to start the interactive TensorBoard visualization to monitor the training progress and model architecture;
3. ModelCheckpoint callback to save model weights for model inference later;
4. LearningRateScheduler callback to decay the learning rate at the end of each epoch.

Once these callbacks are defined, we'll pass it to the fit() method for training. The fit() method trains the model with a specified number of epochs and steps per epoch. Note that the numbers here are for demonstration purposes only to speed up our local experiments and may not sufficiently produce a model with good quality in real-world applications.

Listing 9.6 Model training with callbacks

```
single_worker_model = build_and_compile_cnn_model()
checkpoint_prefix = os.path.join(args.checkpoint_dir, "ckpt_{epoc
class PrintLR(tf.keras.callbacks.Callback):
  def on_epoch_end(self, epoch, logs=None):
              print('\nLearning rate for epoch {} is {}'.format(
              epoch + 1, multi_worker_model.optimizer.lr.numpy()

callbacks = [
              tf.keras.callbacks.TensorBoard(log_dir='./logs'),
              tf.keras.callbacks.ModelCheckpoint(filepath=checkp
                                          save_weights_only=True),
```

```
                tf.keras.callbacks.LearningRateScheduler(decay),
                PrintLR()
]

single_worker_model.fit(ds_train,
                                epochs=1,
                                steps_per_epoch=70,
                                callbacks=callbacks)
```

We'll see the model training progress like the following in the logs:

```
Learning rate for epoch 1 is 0.0010000000474974513
70/70 [==============================] - 16s 136ms/step - loss: 1
Here's the summary of the model architecture in the logs:
Model: "sequential"
_________________________________________________________________
 Layer (type)                                Output Shape
=================================================
 conv2d (Conv2D)                                   (None, 26, 26, 32)
 max_pooling2d (MaxPooling2D)                (None, 13, 13, 32)
 conv2d_1 (Conv2D)                                (None, 11, 11, 64)
 max_pooling2d_1                               (MaxPooling2D) (None, 5, 5,
 conv2d_2 (Conv2D)                               (None, 3, 3, 64)
 flatten (Flatten)                                (None, 576)
 dense (Dense)                                    (None, 64)
 dense_1 (Dense)                                  (None, 10)
=================================================
Total params: 93,322
Trainable params: 93,322
Non-trainable params: 0
```

Based on the summary above, 93k parameters will be trained during the process. The shape and the number of parameters in each layer can also be found in the summary.

9.2.2 Distributed model training

Now that we've defined our models and can train them locally in a single machine. The next step is to insert the distributed training logic in the code so that we can run model training with multiple workers using the collective communication pattern that we introduced in the book.

We'll leverage tht tf.distribute module that contains

MultiWorkerMirroredStrategy. It’s a distribution strategy for synchronous training on multiple workers. It creates copies of all variables in the model's layers on each device across all workers. This strategy uses a distributed collective implementation (e.g., all-reduce), so multiple workers can work together to speed up training. If you don’t have appropriate GPUs, you can replace "communication_options" with other implementations. Since we want to ensure the distributed training can run on different machines that might not have GPUs, we’ll replace it with CollectiveCommunication.AUTO so that it will pick any available hardware automatically.

Once we define our distributed training strategy, we’ll initiate our distributed input data pipeline (as mentioned previously in Section 9.1.2) and the model inside the strategy scope. Note that defining the model inside the strategy scope is required since TensorFlow knows how to copy the variables in the model’s layers to each worker adequately based on the strategy. Here we define different model types (basic CNN, CNN with dropout, or CNN with batch normalization) based on the command line arguments we pass to this Python script.

We’ll get to the rest of the flags soon. Once the data pipeline and the model are defined inside the scope, we can use fit() to train the model outside the distribution strategy scope.

Listing 9.7 Distributed model training logic

```
strategy = tf.distribute.MultiWorkerMirroredStrategy(
communication_options=tf.distribute.experimental.CommunicationOpt
implementation=tf.distribute.experimental.CollectiveCommunication

BATCH_SIZE_PER_REPLICA = 64
BATCH_SIZE = BATCH_SIZE_PER_REPLICA * strategy.num_replicas_in_sy

with strategy.scope():
  ds_train = make_datasets_unbatched().batch(BATCH_SIZE).repeat()
  options = tf.data.Options()
  options.experimental_distribute.auto_shard_policy = \
              tf.data.experimental.AutoShardPolicy.DATA
  ds_train = ds_train.with_options(options)
  if args.model_type == "cnn":
    multi_worker_model = build_and_compile_cnn_model()
  elif args.model_type == "dropout":
```

```
    multi_worker_model = build_and_compile_cnn_model_with_dropout
  elif args.model_type == "batch_norm":
    multi_worker_model = build_and_compile_cnn_model_with_batch_n
  else:
    raise Exception("Unsupported model type: %s" % args.model_typ

multi_worker_model.fit(ds_train,
                                epochs=1,
                                steps_per_epoch=70)
```

Once the model training is finished via fit() function, we want to save the model. One common mistake that users will easily make is saving models on all the workers, which may not save the completed model correctly and wastes computational resources and storage. The correct way to do this is only to save the model on the chief worker. We can inspect the environment variable TF_CONFIG, which contains the cluster information, such as the task type and index, to see whether the worker is chief. Note that we want to save the model to a unique path across workers to avoid unexpected errors.

Listing 9.8 Save model with chief worker

```
def is_chief():
  return TASK_INDEX == 0

tf_config = json.loads(os.environ.get('TF_CONFIG') or '{}')
TASK_INDEX = tf_config['task']['index']

if is_chief():
  model_path = args.saved_model_dir
else:
  model_path = args.saved_model_dir + '/worker_tmp_' + str(TASK_I

multi_worker_model.save(model_path)
```

So far, we've seen two command line flags already, namely "saved_model_dir" and "model_type". Here's the rest of the "main" function that will parse those command line arguments. In addition to those two arguments, there's another "checkpoint_dir" argument that we will use to save our model to the TensorFlow SavedModel format that can be easily consumed for our model serving component. We will discuss that in detail in Section 9.3. Note that we also disabled the progress bar for the TensorFlow Datasets module to reduce the logs we will be seeing.

Listing 9.9 Entrypoint main function

```
if __name__ == '__main__':
  tfds.disable_progress_bar()

  parser = argparse.ArgumentParser()
  parser.add_argument('--saved_model_dir',
                              type=str,
                              required=True,
                            help='Tensorflow export directory.')

  parser.add_argument('--checkpoint_dir',
                              type=str,
                              required=True,
                              help='Tensorflow checkpoint direct

  parser.add_argument('--model_type',
                            type=str,
                              required=True,
                              help='Type of model to train.')

  parsed_args = parser.parse_args()
  main(parsed_args)
```

We've just finished writing our Python script that contains the distributed model training logic. Let's containerize it and build the image used to run distributed training in our local Kubernetes cluster. In our Dockerfile, we use Python 3.9 base image, install TensorFlow and TensorFlow Datasets modules via pip, and copy our multi-worker distributed training Python script.

Listing 9.10 Containerization

```
FROM python:3.9
RUN pip install tensorflow==2.11.0 tensorflow_datasets==4.7.0
COPY multi-worker-distributed-training.py /
```

We then build the image from the Dockerfile we just defined. Note that we also need to import the image to K3d cluster since our cluster does not have access to our local image registry.

Listing 9.11 Build and import docker image

```
> docker build -f Dockerfile -t kubeflow/multi-worker-strategy:v0
```

```
> k3d image import kubeflow/multi-worker-strategy:v0.1 --cluster
```

Note that once the worker pods are completed, all files in the pod will be recycled. Since we are running distributed model training across multiple workers in Kubernetes pods, all the model checkpoints will be lost, and we don't have a trained model for model serving. To address that problem, we'll leverage *PersistentVolume* (PV) and *PersistentVolumeClaim* (PVC).

PV is a storage in the cluster that has been provisioned by an administrator or dynamically provisioned. It is a resource in the cluster, just like a node is a cluster resource. PVs are volume plugins like Volumes, but have a lifecycle independent of any individual Pod that uses the PV. In other words, PVs will persist and live even after the pods are completed or deleted.

A PVC is a request for storage by a user. It is similar to a Pod. Pods consume node resources, and PVCs consume PV resources. Pods can request specific levels of resources (CPU and Memory). Claims can request specific size and access modes (e.g., they can be mounted ReadWriteOnce, ReadOnlyMany, or ReadWriteMany).

Let's create a PVC to submit a request for storage that will be used in our worker pods to store the trained model. Here we only submit a request for 1Gi storage with ReadWriteOnce access mode.

Listing 9.12 Persistent volume claim

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: strategy-volume
spec:
  accessModes: [ "ReadWriteOnce" ]
  resources:
    requests:
      storage: 1Gi
```

Listing 9.13 Create PVC

```
> kubectl create -f multi-worker-pvc.yaml
```

Next, let’s define the TFJob spec we introduced in Chapter 7 with the image we just built that contains the distributed training script. We pass the necessary command arguments to the container to train the basic CNN model. The “volumes” field in the worker spec specifies the name of the persistent volume claim that we just created and the “volumeMouts” field in the container spec specifies what folder to mount the files between the volume to the container. The model will be saved in the “/trained_model” folder inside the volume.

Listing 9.14 Distributed model training job definition

```
apiVersion: kubeflow.org/v1
kind: TFJob
metadata:
  name: multi-worker-training
spec:
  runPolicy:
    cleanPodPolicy: None
  tfReplicaSpecs:
    Worker:
      replicas: 2
      restartPolicy: Never
      template:
        spec:
          containers:
            - name: tensorflow
              image: kubeflow/multi-worker-strategy:v0.1
              imagePullPolicy: IfNotPresent
              command: ["python", "/multi-worker-distributed-trai
              volumeMounts:
                - mountPath: /trained_model
                  name: training
              resources:
                limits:
                  cpu: 500m
          volumes:
            - name: training
              persistentVolumeClaim:
                claimName: strategy-volume
```

Then we can submit this TFJob to our cluster to start our distributed model training:

Listing 9.15 Submit TFJob

```
> kubectl create -f multi-worker-tfjob.yaml
```

Once the worker pods are completed, we'll notice the following logs from the pods that indicate we trained the model in a distributed fashion and the workers communicated with each other successfully:

```
2023-01-10 18:01:09.761955: I tensorflow/core/distributed_runtime
2023-01-10 18:01:42.785052: I tensorflow/core/distributed_runtime
2023-01-10 18:01:56.172714: I tensorflow/core/distributed_runtime
2023-01-10 18:01:56.173026: I tensorflow/core/distributed_runtime
```

9.2.3 Model selection

So far, we've implemented our distributed model training component. We'll eventually train three different models, as mentioned in Section 9.2.1, and then pick the top model for model serving. Let's assume that we have trained those models successfully by submitting three different TFJobs with different model types.

Next, we write the Python code that loads the testing data and trained models and then evaluate their performance. We will load each trained model from different folders by keras.models.load_model() function and execute model.evaluate() which returns the loss and accuracy. Once we found the model with the highest accuracy, we copy the model to a new version in a different folder, namely "4", which will be used by our model serving component.

Listing 9.16 Model evaluation

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
import tensorflow_datasets as tfds
import shutil
import os

def scale(image, label):
  image = tf.cast(image, tf.float32)
  image /= 255
  return image, label
```

```
best_model_path = ""
best_accuracy = 0
for i in range(1, 4):
  model_path = "trained_model/saved_model_versions/" + str(i)
  model = keras.models.load_model(model_path)
  datasets, _ = tfds.load(name='fashion_mnist', with_info=True, a
  ds = datasets['test'].map(scale).cache().shuffle(10000).batch(6
  _, accuracy = model.evaluate(ds)
  if accuracy > best_accuracy:
    best_accuracy = accuracy
    best_model_path = model_path

destination = "trained_model/saved_model_versions/4"
if os.path.exists(destination):
  shutil.rmtree(destination)

shutil.copytree(best_model_path, destination)
print("Best model with accuracy %f is copied to %s" % (best_accur
```

Note that The latest version, “4”, in “trained_model/saved_model_versions” folder will be picked up by our serving component. We will talk about that in the next section.

We then add this Python script to our Dockerfile, rebuilds the container image, and create a pod that runs the model selection component. The following is the YAML file that configures the model selection pod.

Listing 9.17 Model selection pod definition

```
apiVersion: v1
kind: Pod
metadata:
  name: model-selection
spec:
  containers:
  - name: predict
    image: kubeflow/multi-worker-strategy:v0.1
    command: ["python", "/model-selection.py"]
    volumeMounts:
    - name: model
      mountPath: /trained_model
  volumes:
  - name: model
    persistentVolumeClaim:
      claimName: strategy-volume
```

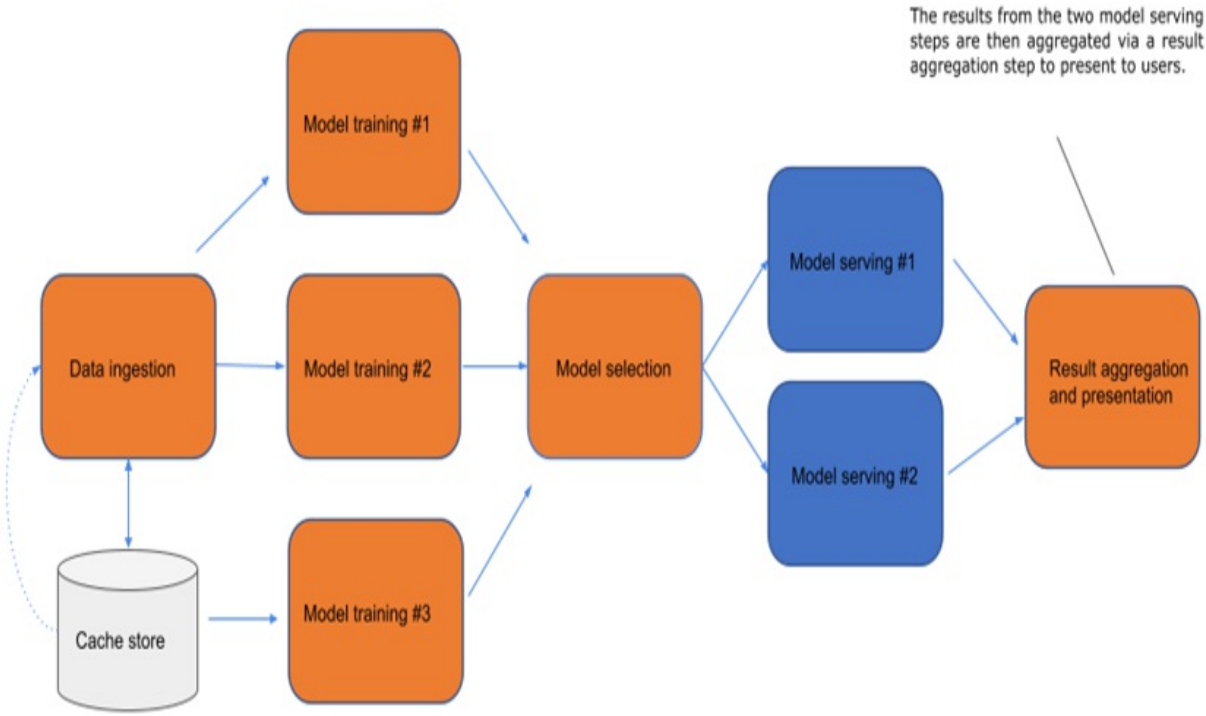
When inspecting the logs, we'll see the third model has the highest accuracy, and will be copied to a new version to be used for the model serving component.

```
157/157 [==============================] - 1s 5ms/step - loss: 0.
157/157 [==============================] - 1s 5ms/step - loss: 0.
157/157 [==============================] - 1s 5ms/step - loss: 0.
```

9.3 Model serving

Now that we have implemented our distributed model training as well as model selection among the trained models. The next component we will implement in this section is the model serving component. The model serving component is essential to the end-user experience since the results will be shown to our users directly and if it's not performant enough, our users will know immediately. Figure 9.5 shows the model training component in the overall architecture.

Figure 9.5 Model serving component (dark box) in the end-to-end machine learning system.



In Figure 9.5, the model serving components are shown as the two dark boxes between the model selection and result aggregation steps. Let's first implement our single-server model inference component in Section 9.3.1 and then make it more scalable and performant in Section 9.3.2.

9.3.1 Single-server model inference

The model inference Python code is very similar to the model evaluation code. The only difference is that we use the model.predict() method instead of evaluate() after we load the trained model. This is an excellent way to test if the trained model can make predictions as expected.

Listing 9.18 Model prediction

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
import tensorflow_datasets as tfds
model = keras.models.load_model("trained_model/saved_model_versio
def scale(image, label):
  image = tf.cast(image, tf.float32)
  image /= 255
  return image, label
datasets, _ = tfds.load(name='fashion_mnist', with_info=True, as_
ds = datasets['test'].map(scale).cache().shuffle(10000).batch(64)
model.predict(ds)
```

Alternatively, you should be able to start a TensorFlow Serving (https://github.com/tensorflow/serving) server locally like the following once it's installed:

Listing 9.19 TensorFlow Serving command

```
tensorflow_model_server --model_name=flower-sample \
         --port=9000 \
         --rest_api_port=8080 \
         --model_base_path=trained_model/saved_model \
         --rest_api_timeout_in_ms=60000
```

This seems straightforward and works well if we are only experimenting locally. However, there are more performant ways to build our model serving

component that will pave our path to running distributed model serving that incorporates the replicated model server pattern that we introduced in previous chapters.

Before we dive into a better solution, let's make sure our trained model can work with our prediction inputs, which will be a JSON-structured list of image bytes with the key "instances" and "image_bytes", like the following:

```
{
    "instances":[
       {
          "image_bytes":{
             "b64":"/9j/4AAQSkZJRgABAQAAAQABAAD
…
<truncated>
/hWY4+UVEhkoIYUx0psR+apm6VBRUZcUYFSuKZgUAf//Z"
          }
       }
    ]
 }
```

Now is the time to modify our distributed model training code to make sure the model has the correct serving signature that's compatible with our supplied inputs. We define the preprocessing function that does the following:

1. Decode the images from bytes
2. Resize the image to 28x28 that's compatible with our model architecture
3. Cast the images to tf.uint8
4. Define the input signature with string type and key as "image_bytes"

Once the preprocessing function is defined, we can define the serving signature via tf.TensorSpec() and then pass it to tf.saved_model.save() method to save the model that is compatible with our input format and preprocess it before TensorFlow Serving makes inference calls.

Listing 9.20 Model serving signature definitions

```
def _preprocess(bytes_inputs):
    decoded = tf.io.decode_jpeg(bytes_inputs, channels=1)
    resized = tf.image.resize(decoded, size=(28, 28))
```

```
    return tf.cast(resized, dtype=tf.uint8)
def _get_serve_image_fn(model):
    @tf.function(input_signature=[tf.TensorSpec([None], dtype=tf.
    def serve_image_fn(bytes_inputs):
        decoded_images = tf.map_fn(_preprocess, bytes_inputs, dty
        return model(decoded_images)
    return serve_image_fn
signatures = {
    "serving_default": _get_serve_image_fn(multi_worker_model).ge
        tf.TensorSpec(shape=[None], dtype=tf.string, name='image_
    )
  }

tf.saved_model.save(multi_worker_model, model_path, signatures=si
```

Once the distributed model training script is modified, we can rebuild our container image and re-train our model from scratch, following the instructions in Section 9.2.2.

Next, we will use KServe as we mentioned in the technologies overview, to create an inference service. Here's the YAML to define the KServe inference service. We need to specify the model format so that KServe knows what to use for serving the model, e.g., TensorFlow Serving. In addition, we need to supply the URI to the trained model. In this case, we can specify the PVC name and the path to the trained model, following the format "pvc://<pvc-name>/<model-path>".

Listing 9.21 Inference service definition

```
apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
metadata:
  name: flower-sample
spec:
  predictor:
    model:
      modelFormat:
        name: tensorflow
      storageUri: "pvc://strategy-volume/saved_model_versions"
```

Let's install KServe and create our inference service!

Listing 9.22 Install KServe and create the inference service

```
> curl -s "https://raw.githubusercontent.com/kserve/kserve/v0.10.
> kubectl create -f inference-service.yaml
```

We can check its status to make sure it's ready for serving:

Listing 9.23 Get the details of the inference service

```
> kubectl get isvc
NAME                  URL
flower-sample   http://flower-sample.kubeflow.example.com
```

Once the service is created, we port-forward it to local so that we can send requests to it locally:

Listing 9.24 Port-forward the inference service

```
> INGRESS_GATEWAY_SERVICE=$(kubectl get svc --namespace istio-sys
> kubectl port-forward --namespace istio-system svc/${INGRESS_GAT
```

You should be able to see the following if the port-forwarding is successful:

```
Forwarding from 127.0.0.1:8080 -> 8080
Forwarding from [::1]:8080 -> 8080
```

Let's open another terminal and execute the following Python script to send a sample inference request to our model serving service and print out the response text.

Listing 9.25 Use Python to send inference request

```
import requests
import json
input_path = "inference-input.json"

with open(input_path) as json_file:
         data = json.load(json_file)
r = requests.post(
 url="http://localhost:8080/v1/models/flower-sample:predict",
 data=json.dumps(data), headers={'Host': 'flower-sample.kubeflow.
print(r.text)
```

Here's the response from our KServe model serving service, which includes

the predicted probabilities for each class in the Fashion-MNIST dataset:

```
{
  "predictions": [[0.0, 0.0, 1.22209595e-11, 0.0, 1.0, 0.0, 7.074
}
```

Alternatively, we can use curl to send requests as well:

Listing 9.26 Use Curl to send inference request

```
# Start another terminal
export INGRESS_HOST=localhost
export INGRESS_PORT=8080
MODEL_NAME=flower-sample
INPUT_PATH=@./inference-input.json
SERVICE_HOSTNAME=$(kubectl get inferenceservice ${MODEL_NAME} -o
curl -v -H "Host: ${SERVICE_HOSTNAME}" "http://${INGRESS_HOST}:${
```

Here’s the output you should expect. The output probabilities should be the same as the ones we just saw:

```
*   Trying ::1:8080...
* Connected to localhost (::1) port 8080 (#0)
> POST /v1/models/flower-sample:predict HTTP/1.1
> Host: flower-sample.kubeflow.example.com
> User-Agent: curl/7.77.0
> Accept: */*
> Content-Length: 16178
> Content-Type: application/x-www-form-urlencoded
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< content-length: 102
< content-type: application/json
< date: Thu, 05 Jan 2023 21:11:36 GMT
< x-envoy-upstream-service-time: 78
< server: istio-envoy
<
{
        "predictions": [[0.0, 0.0, 1.22209595e-11, 0.0, 1.0, 0.0
        ]
* Connection #0 to host localhost left intact
}
```

As mentioned previously, even though we specified the entire directory that

contains the trained model in the KServe InferenceService spec, the model serving service that utilizes TensorFlow Serving will pick the latest version "4" from that particular folder, which is our best model we selected in Section 9.2.3. We can observe that from the logs of the serving pod:

Listing 9.27 Inspect the model server logs

```
> kubectl logs flower-sample-predictor-default-00001-deployment-f
```

Here's the logs:

```
2023-01-05 21:07:32.070697: I external/tf_serving/tensorflow_serv
2023-01-05 21:07:32.072070: I external/tf_serving/tensorflow_serv
2023-01-05 21:07:32.072090: I external/tf_serving/tensorflow_serv
…
<truncated>
2023-01-05 21:07:32.238340: I external/tf_serving/tensorflow_serv
```

9.3.2 Replicated model servers

In the previous section, we successfully deployed our model serving service in our local Kubernetes cluster. This might be sufficient for running local serving experiments, but it's far from ideal if it's deployed to production systems that serve real-world model serving traffic. The current model serving service is a single Kubernetes pod, where the allocated computational resources are limited and requested in advance. When the number of model serving requests increases, the single-instance model server can no longer support the workloads and may run out of computational resources.

To address the problem, we need to have multiple instances of model servers to handle a larger amount of dynamic model serving requests. Fortunately, KServe can autoscale based on the average number of in-flight requests per pod, which leverages the Knative Serving autoscaler.

The following is the inference service spec with auto-scaling enabled. The "scaleTarget" field specifies the integer target value of the metric type the autoscaler watches for. In addition, the "scaleMetric" field defines the scaling metric type watched by autoscaler. The possible metrics are: concurrency, rps, cpu, memory. Here we only allow one concurrent request to be processed

by each inference service instance. In other words, when there are more requests, we will start a new inference service pod to handle each additional request.

Listing 9.28 Replicated model inference services

```
apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
metadata:
  name: flower-sample
spec:
  predictor:
    scaleTarget: 1
    scaleMetric: concurrency
    model:
      modelFormat:
        name: tensorflow
      storageUri: "pvc://strategy-volume/saved_model_versions"
```

Let's assume there's no request no,w and we should only see one inference service pod that's up and running. Next, let's send traffic in 30 seconds spurts, maintaining five in-flight requests. We use the same service host name, and ingress address, as well as the same inference input and trained model. Note that we are using the tool called "hey", which is a tiny program that sends some load to a web application. Please follow the instructions in https://github.com/rakyll/hey to install it before executing the following command.

Listing 9.29 Send traffic to test the load

```
> hey -z 30s -c 5 -m POST -host ${SERVICE_HOSTNAME} -D inference-
```

Below is the expected output from the command, which includes a summary of how the inference service handled the requests. For example, the service has processed 230160 bytes of inference inputs and 95.7483 requests per second. You can also find a nice response time histogram and a latency distribution that might be useful.

```
Summary:
  Total:        30.0475 secs
  Slowest:              0.2797 secs
```

```
  Fastest:       0.0043 secs
  Average:             0.0522 secs
  Requests/sec:  95.7483
  Total data:          230160 bytes
  Size/request:       80 bytes
Response time histogram:
  0.004 [1]             |
  0.032 [1437]   |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.059 [3]             |
0.087 [823] |■■■■■■■■■■■■■■■■■■■■■■■
0.114 [527] |■■■■■■■■■■■■■■■
  0.142 [22]     |■
  0.170 [5]             |
  0.197 [51]     |■
  0.225 [7]             |
  0.252 [0]             |
  0.280 [1]             |
Latency distribution:
  10% in 0.0089 secs
  25% in 0.0123 secs
  50% in 0.0337 secs
  75% in 0.0848 secs
  90% in 0.0966 secs
  95% in 0.1053 secs
  99% in 0.1835 secs
Details (average, fastest, slowest):
  DNS+dialup:        0.0000 secs, 0.0043 secs, 0.2797 secs
  DNS-lookup:        0.0000 secs, 0.0000 secs, 0.0009 secs
  req write:            0.0000 secs, 0.0000 secs, 0.0002 secs
  resp wait:            0.0521 secs, 0.0042 secs, 0.2796 secs
  resp read:           0.0000 secs, 0.0000 secs, 0.0005 secs
Status code distribution:
  [200]   2877 responses
```

As expected, we see five running inference service pods processing the requests concurrently, where each pod handles only one request.

Listing 9.30 Get the list of model server pods

```
> kubectl get pods
NAME                                                    READY   ST
flower-sample-predictor-default-00001-deployment-sr5wd 3/3    Run
flower-sample-predictor-default-00001-deployment-swnk5 3/3    Run
flower-sample-predictor-default-00001-deployment-t2njf 3/3    Run
flower-sample-predictor-default-00001-deployment-vdlp9 3/3    Run
flower-sample-predictor-default-00001-deployment-vm58d 3/3    Run
```

Once the “hey” command is completed, we would only see one running pod:

Listing 9.31 Get the list of model server pods again

```
> kubectl get pods
NAME                                                        READY
flower-sample-predictor-default-00001-deployment-sr5wd      3/3
```

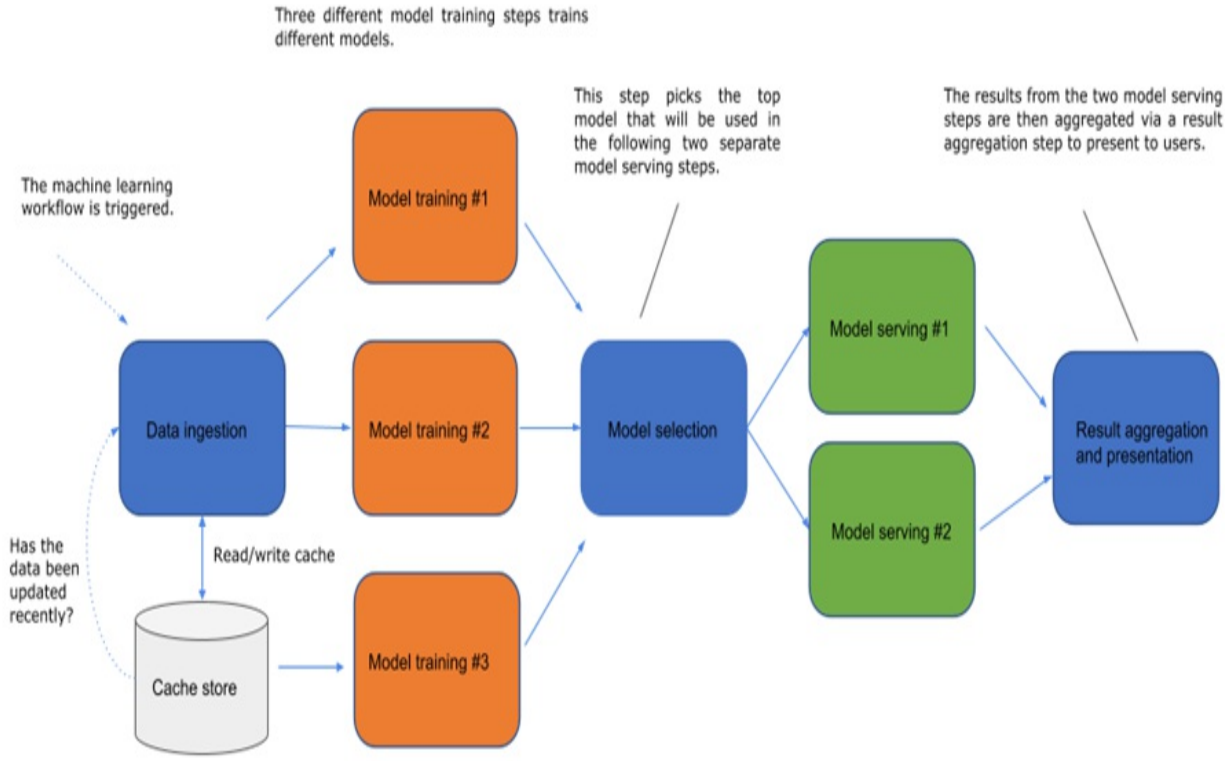
9.4 End-to-end workflow

We have just implemented all the components in previous sections. Now it’s time to put things together!

In this section, we’ll define an end-to-end workflow using Argo Workflows that includes the components we just implemented. Please go back to previous sections if you are still unfamiliar with all the components and refresh your knowledge of basic Argo Workflows from Chapter 8.

Here’s a recap of what the end-to-end that we will be implementing looks like. Figure 9.6 is a diagram of the end-to-end workflow that we are building. Note that there are two model serving steps in the diagram for illustration purposes but we will only implement one step in our Argo workflow and it will autoscale to more instances based on requests traffic as we mentioned in Section 9.3.2.

Figure 9.6 Architecture diagram of the end-to-end machine learning system we will be building.



In the next sections, we will first define the entire workflow by connecting the steps sequentially with Argo and then optimize the workflow for future executions by implementing step memoization.

9.4.1 Sequential steps

First, let's look at the entry point templates and the main steps involved in the workflow. The entry point template name is "tfjob-wf", which consists of the following steps (for simplicity, each step uses a template with the same name):

1. "data-ingestion-step" step contains the data ingestion step, which we will use to download and preprocess the dataset before model training
2. "distributed-tf-training-steps" is a step group that consists of multiple sub-steps where each sub-step represents a distributed model training step for a specific model type
3. "model-selection-step" is a step that selects the top model among the different models we have trained in previous steps
4. "create-model-serving-service" step creates the model serving serve via

Listing 9.32 Workflow entrypoint templates

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: tfjob-wf-
  namespace: kubeflow
spec:
  entrypoint: tfjob-wf
  templates:
  - name: tfjob-wf
    steps:
    - - name: data-ingestion-step
        template: data-ingestion-step
    - - name: distributed-tf-training-steps
        template: distributed-tf-training-steps
    - - name: model-selection-step
        template: model-selection-step
    - - name: create-model-serving-service
        template: create-model-serving-service
  podGC:
    strategy: OnPodSuccess
  volumes:
  - name: model
    persistentVolumeClaim:
      claimName: strategy-volume
```

Note that we specify the podGC strategy to be “OnPodSuccess” since we’ll be creating a lot of pods for different steps within our local K3s cluster with limited computational resources, so deleting the pods right after they are successful can free up computational resources for the subsequent steps. There’s also another “OnPodCompletion” strategy available which will delete pods upon completion regardless of whether they failed or successful. We won’t be using that since we want to keep failed pods so that we can debug what went wrong.

In addition, we also specify our volumes and PVC to ensure we can persist any files that will be used among the steps. We can save the downloaded dataset into the persistent volume for model training and then persist the trained model for the subsequent model serving step.

The first step, the data ingestion step, is very straightforward. It only specifies the container image and the data ingestion Python script to execute. The Python script is a one-line code with tfds.load(name='fashion_mnist') to download the dataset to the container's local storage, which will be mounted to our persistent volume.

Listing 9.33 Data ingestion step

```
- name: data-ingestion-step
    serviceAccountName: argo
    container:
      image: kubeflow/multi-worker-strategy:v0.1
      imagePullPolicy: IfNotPresent
      command: ["python", "/data-ingestion.py"]
```

A next step is a step group that consists of multiple sub-steps where each sub-step represents a distributed model training step for a specific model type, e.g., basic CNN, CNN with dropout, and CNN with batch norm. Here's the template that defines all the sub-steps.

Distributed training steps for multiple models, explain that these will be executed in parallel.

Listing 9.34 Distributed training step groups

```
- name: distributed-tf-training-steps
    steps:
    - - name: cnn-model
        template: cnn-model
      - name: cnn-model-with-dropout
        template: cnn-model-with-dropout
      - name: cnn-model-with-batch-norm
        template: cnn-model-with-batch-norm
```

Let's use the first sub-step as an example, which runs a distributed model training for the basic CNN model. The main content of this step template is the "resource" field, which includes the following:

1. The CRD or manifest to take action upon. In our case, we create a TFJob as part of this step.
2. The conditions which indicate whether the CRD is created successfully.

In our case, we ask Argo to watch the field "status.replicaStatuses.Worker.succeeded" and "status.replicaStatuses.Worker.failed".

Note that inside the container spec in TFJob definition, we specify the model type and save the trained model to a different folder so it's easy for us to pick and save the best model that can be used for model serving in subsequent steps. We also want to make sure to attach the persistent volumes so that the trained model can be persisted.

Listing 9.35 CNN model training step

```
- name: cnn-model
    serviceAccountName: training-operator
    resource:
      action: create
      setOwnerReference: true
      successCondition: status.replicaStatuses.Worker.succeeded =
      failureCondition: status.replicaStatuses.Worker.failed > 0
      manifest: |
        apiVersion: kubeflow.org/v1
        kind: TFJob
        metadata:
          generateName: multi-worker-training-
        spec:
          runPolicy:
            cleanPodPolicy: None
          tfReplicaSpecs:
            Worker:
              replicas: 2
              restartPolicy: Never
              template:
                spec:
                  containers:
                    - name: tensorflow
                      image: kubeflow/multi-worker-strategy:v0.1
                      imagePullPolicy: IfNotPresent
                      command: ["python", "/multi-worker-distribu
                      volumeMounts:
                        - mountPath: /trained_model
                          name: training
                      resources:
                        limits:
                          cpu: 500m
```

```
      volumes:
        - name: training
          persistentVolumeClaim:
            claimName: strategy-volume
```

For the rest of the sub-steps in “distributed-tf-training-steps” step, the spec is very similar except that the saved model directory and model type arguments are different.

The next step is model selection, we supply the same container image but execute the model selection Python script we implemented earlier.

Listing 9.36 Model selection step Caption here

```
- name: model-selection-step
    serviceAccountName: argo
    container:
      image: kubeflow/multi-worker-strategy:v0.1
      imagePullPolicy: IfNotPresent
      command: ["python", "/model-selection.py"]
      volumeMounts:
      - name: model
        mountPath: /trained_model
```

Please ensure these additional scripts are included in your Dockerfile, and that you have rebuilt the image and re-imported it to your local Kubernetes cluster.

Once the model selection step is implemented, the last step in the workflow is the model serving step that starts a KServe model inference service. It’s a resource template similar to the model training steps but with KServe’s InferenceService CRD and a success condition that applies to this specific CRD.

Listing 9.37 Model serving step

```
- name: create-model-serving-service
    serviceAccountName: training-operator
    successCondition: status.modelStatus.states.transitionStatus
    resource:
      action: create
      setOwnerReference: true
```

```
      manifest: |
        apiVersion: serving.kserve.io/v1beta1
        kind: InferenceService
        metadata:
          name: flower-sample
        spec:
          predictor:
            model:
              modelFormat:
                name: tensorflow
              image: "emacski/tensorflow-serving:2.6.0"
              storageUri: "pvc://strategy-volume/saved_model_vers
```

Let’s submit this workflow now!

Listing 9.38 Submit the end-to-end workflow

```
> kubectl create -f workflow.yaml
```

Once the data ingestion step is completed, the associated pod will be deleted. When we list the pods again while it’s executing the distributed model training steps, you’ll see the pods with names prefixed by “tfjob-wf-f4bql-cnn-model-”, which are the pods responsible for monitoring the status of distributed model training for different model types. In addition, each model training for each model type contains two workers with name matching the pattern “multi-worker-training-*-worker-*”.

Listing 9.39 Get the list of pods

```
> kubectl get pods
NAME                                        READY     STATUS
multi-worker-training-2sdgf-worker-0         1/1           Running
multi-worker-training-2sdgf-worker-1         1/1           Running
multi-worker-training-gtwwq-worker-0        1/1           Running
multi-worker-training-gtwwq-worker-1        1/1           Running
multi-worker-training-qwlrw-worker-0         1/1           Running
multi-worker-training-qwlrw-worker-1         1/1           Running
tfjob-wf-f4bql-cnn-model-600587901           1/1           Running
tfjob-wf-f4bql-cnn-model-with-batch-norm 1/1           Running
Tfjob-wf-f4bql-cnn-model-with-dropout        1/1             Running
```

Once the remaining steps are completed, and the model serving has started successfully, the workflow should be in “succeeded” status, and we’ve just

finished the execution of the end-to-end workflow.

9.4.2 Step memoization

In order to speed up future execution of workflows, we can utilize cache and skip certain steps that have recently run. In our case, the data ingestion step can be skipped since we don't have to download the same dataset again and again.

Let's first take a look at the logs from our data ingestion step:

```
Downloading and preparing dataset 29.45 MiB (download: 29.45 MiB,
Dataset fashion_mnist downloaded and prepared to /root/tensorflow
```

Note that the dataset has been download to a path in the container. If the path is mounted to our persistent volume, it will be available for any future workflow runs. Let's use the step memoization feature provided by Argo Workflows to optimize our workflow.

Inside the step template, we supply the "memoize" field with the cache key and age of the cache. When a step is completed, a cache will be saved. When this step runs again in a new workflow in the future, it will check whether the cache is created within the past one hour. If so, this step will be skipped and the workflow will proceed to execute subsequent steps. For our application, our dataset does not change at all so theoretically the cache should be always used and we only specify one hour here for demonstration purposes only. In real-world applications, you may want to adjust that according to how frequent the data is updated.

Listing 9.40 Memoization for data ingestion step

```
- name: data-ingestion-step
    serviceAccountName: argo
    memoize:
      key: "step-cache"
      maxAge: "1h"
      cache:
        configMap:
          name: my-config
          key: step-cache
```

```
    container:
      image: kubeflow/multi-worker-strategy:v0.1
      imagePullPolicy: IfNotPresent
      command: ["python", "/data-ingestion.py"]
```

Let’s run the workflow for the first time and pay attention to the memoization status field in the workflow’s node status. The cache is not hit because this is the first time the step is run:

Listing 9.41 Check the node statuses of the workflow

```
> kubectl get wf tfjob-wf-kjj2q -o yaml
```

Here’s the section for node statuses:

```
Status:
  Nodes:
    tfjob-wf-crfhx-2213815408:
      Boundary ID:  tfjob-wf-crfhx
      Children:
        tfjob-wf-crfhx-579056679
      Display Name:    data-ingestion-step
      Finished At:     2023-01-04T20:57:44Z
      Host Node Name:  distml-control-plane
      Id:              tfjob-wf-crfhx-2213815408
      Memoization Status:
        Cache Name:  my-config
        Hit:            false
        Key:            step-cache
      Name:          tfjob-wf-crfhx[0].data-ingestion-step
```

If we run the same workflow again within one hour, we will notice that the step is skipped (indicated by “hit: true” in memoization status field):

```
Status:
  Nodes:
   tfjob-wf-kjj2q-1381200071:
      Boundary ID:  tfjob-wf-kjj2q
      Children:
        tfjob-wf-kjj2q-2031651288
      Display Name:  data-ingestion-step
      Finished At:   2023-01-04T20:58:31Z
      Id:            tfjob-wf-kjj2q-1381200071
      Memoization Status:
        Cache Name:  my-config
```

```
    Hit:            true
    Key:          step-cache
  Name:           tfjob-wf-kjj2q[0].data-ingestion-step
  Outputs:
    Exit Code:        0
  Phase:              Succeeded
  Progress:           1/1
  Started At:       2023-01-04T20:58:31Z
  Template Name:    data-ingestion-step
  Template Scope:   local/tfjob-wf-kjj2q
  Type:               Pod
```

In addition, note that the finished and started timestamp is the same. Meaning that this step is completed instantly without having to re-execute from scratch.

All the cache in Argo Workflows is saved in a Kubernetes ConfigMap object, let's take a look at it. The cache contains the node ID, step outputs, cache creation timestamp, as well as the timestamp when this cache is last hit.

Listing 9.42 Check the details of the configmap

```
> kubectl get configmap -o yaml my-config
apiVersion: v1
data:
  step-cache: '{"nodeID":"tfjob-wf-dmtn4-3886957114","outputs":{"
kind: ConfigMap
metadata:
  creationTimestamp: "2023-01-04T20:44:55Z"
  labels:
    workflows.argoproj.io/configmap-type: Cache
  name: my-config
  namespace: kubeflow
  resourceVersion: "806155"
  uid: 0810a68b-44f8-469f-b02c-7f62504145ba
```

9.5 Summary

- The data ingestion component implements a distributed input pipeline for the Fashion-MNIST dataset with TensorFlow that would make it easy to integrate with distributed model training.
- Machine learning models and distributed model training logic can be

defined in TensorFlow, and then executed in a distributed fashion in the Kubernetes cluster with the help of Kubeflow.

- Both the single-instance model server and the replicated model servers can be implemented via KServe. The autoscaling functionality of KServe can automatically create additional model serving pods to handle the increasing number of model serving requests.
- We implemented our end-to-end workflow that includes all the components of our system in Argo Workflows and were able to leverage step memoization to avoid time-consuming and redundant data ingestion step.