Chapter 7. High-Performance Modeling

In production scenarios, getting the best possible performance from your model is important for delivering fast response times and low costs, with low resource requirements. High-performance modeling becomes especially important when compute resource requirements are large, such as when dealing with large models and/or datasets, and when inference latency and/or cost requirements are challenging.

In this chapter, we'll discuss how models can be accelerated using data and model parallelism. We'll also look at high-performance modeling techniques such as distribution strategies, and high-performance ingestion pipelines such as TF Data. Finally, we'll consider the rise of giant neural nets, and approaches for addressing the resulting need for efficient, scalable infrastructure in that context.

Distributed Training

When you start prototyping, training your model might be a fast and simple task, especially if you're working with a small dataset. However, fully training a model can become very time-consuming. Datasets and model architectures in many domains are getting larger and larger. As the size of training datasets and models increases, models take longer and longer to train. And it's not just the training time for each epoch; often the number of

epochs for a model also increases as a result. Solving this kind of problem usually requires distributed training. Distributed training allows us to train huge models while speeding up training by leveraging more compute resources.

At a high level, there are two primary ways to do distributed training: data parallelism and model parallelism. With *data parallelism*, which is probably the easier of the two to implement, you divide the data into partitions and copy the complete model to all the workers. Each worker operates on a different partition of the data, and the model updates are synchronized across the workers. This type of parallelism is model agnostic and can be applied to any neural network architecture. Usually the scale of data parallelism corresponds to the batch size.

With *model parallelism*, you segment the model into different parts, training it concurrently on different workers. Each worker trains on the same piece of data, and the workers only need to synchronize the shared parameters, usually once for each forward or backpropagation step. You generally use model parallelism when you have larger models that won't fit in memory on your accelerators, such as GPUs or Tensor Processing Units (TPUs). Implementation of model parallelism is relatively advanced compared to data parallelism. Thus, our discussion of distributed training techniques will focus on data parallelism.

Data Parallelism

As noted earlier, with data parallelism, the data is split into partitions, and the number of partitions is usually the total number of available workers in the compute cluster. As shown in Figure 7-1, you copy the model onto each worker node, with each worker training on its own subset of the data. This requires each worker to have enough memory to load the entire model, which for larger models can be a problem.

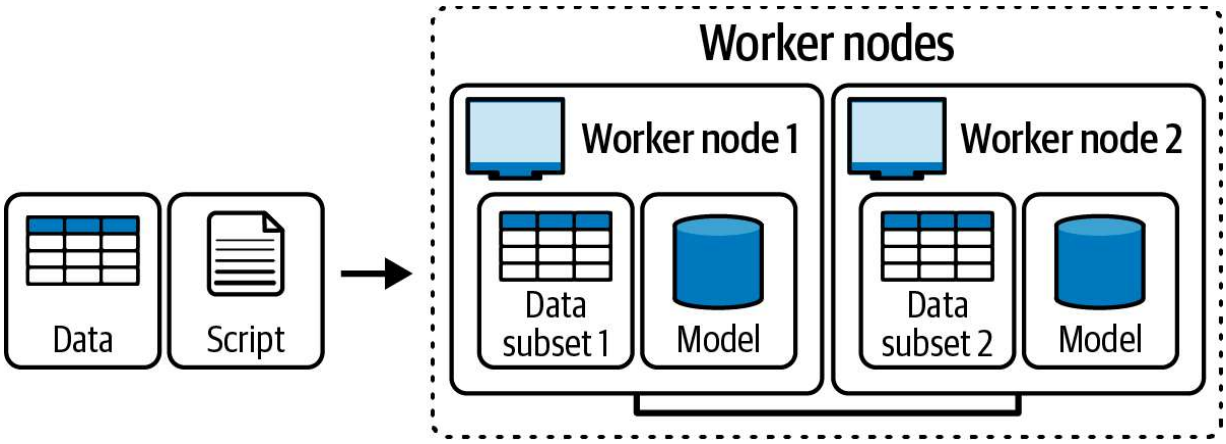

Figure 7-1. Splitting data across worker nodes

Each worker independently computes the errors between its predictions for its training samples and the labeled data, then performs backpropagation to update its model based on the errors and communicates all its changes to the other workers so that they can update their models. This means the workers need to synchronize their gradients at the end of each batch to ensure that they are training a consistent model.

Synchronous versus asynchronous training

The two basic styles of training in data parallelism are synchronous and asynchronous. In *synchronous training*, each worker trains on its current mini batch of data, applies its own updates, communicates its updates to the other workers, and waits to receive and apply all the updates from the other workers before proceeding to the next mini batch. An all-reduce algorithm is an example.

In *asynchronous training*, all workers are independently training over their mini batch of data and updating variables asynchronously. Asynchronous training tends to be more efficient, but it can also be more difficult to implement. A parameter server algorithm is an example of this.

One major disadvantage of asynchronous training is reduced accuracy and slower convergence, which means more steps are needed to converge. Slow convergence may not be a problem, since the speedup in asynchronous training may be enough to compensate. However, the accuracy loss may be an issue, depending on how much accuracy is lost and the requirements of the application.

Distribution awareness

To use distributed training it’s important that models become distribution aware. Fortunately, high-level APIs such as Keras support distributed training.

You can even create your custom training loops to provide more precise control. To make your models capable of performing training or inference in a distributed manner, you need to make them distribution aware with some small changes in code.

Tf.distribute: Distributed training in TensorFlow

To perform distributed training in TensorFlow, you can make use of TensorFlow's `tf. distribute.Strategy` class.

This class supports several distribution strategies for high-level APIs, and also supports training using a custom training loop. The class also supports the execution of TensorFlow code in eager mode and in graph mode, using `tf.function`. In addition to training models, it's also possible to use `tf.distribute.Strategy` to perform model evaluation and prediction in a distributed manner on different platforms.

The `tf.distribute.Strategy` class requires a minimal amount of extra code to adapt your models for distributed training. You can easily switch between different strategies to experiment and find the one that best fits your needs. There are many different strategies for performing distributed training with TensorFlow. The following are the ones used most often:

- `OneDeviceStrategy`
- `MirroredStrategy`

- `ParameterServerStrategy`
- `MultiWorkerMirroredStrategy`
- `TPUStrategy`
- `CentralStorageStrategy`

Here we'll focus on the first three strategies to give you a feel for the basic issues and approaches, as the latter three strategies are derivatives of those. The TensorFlow website has much more information about these strategies.

OneDeviceStrategy

`OneDeviceStrategy` will place any variables created in its scope on the specified device. Input distributed through this strategy will be prefetched to the specified device. Moreover, any functions called via `strategy.run` will also be placed on the specified device.

You might ask: "If it's only one device, what's the point?" Typical usage of this strategy could be testing your code with the `tf.distribute.Strategy` API before switching to other strategies that actually distribute to multiple devices/machines.

MirroredStrategy

`MirroredStrategy` supports synchronous distributed training on multiple GPUs, on one machine. It creates one replica per GPU device, and each variable in the model is mirrored across all the replicas. Together,

these variables form a single conceptual variable called a *mirrored variable*. These variables are kept in sync with one another by applying identical updates.

Efficient all-reduce algorithms are used to communicate the variable updates across the devices. All-reduce aggregates tensors across all the devices by adding them up, and then makes them available on each device. All-reduce is a fused algorithm that is very efficient and can reduce the overhead of synchronization significantly.

With `MultiWorkerMirroredStrategy`, training is distributed on multiple workers, each of which can have multiple GPUs. A `TPUStrategy` is like a `MirroredStrategy` with training distributed on multiple TPUs instead of GPUs. Finally, `CentralStorageStrategy` does not mirror variables, but rather places them on the CPU and replicates operations on all local GPUs.

ParameterServerStrategy

`ParameterServerStrategy` is a common asynchronous data-parallel method for scaling up model training on multiple machines. A parameter server training cluster consists of workers and parameter servers. Variables are created on the parameter servers and are read and updated by the workers in each step. By default, workers read and update these variables independently, without synchronizing with one another. This is

why parameter server–style training is sometimes referred to as asynchronous training.

Fault tolerance

Typically in synchronous training, the entire cluster of workers would fail if one or more of the workers were to fail. Therefore, it's important to consider some form of fault tolerance in cases where workers die or become unstable. This allows you to recover from a failure incurred by preempting workers. This can be done by preserving the training state in the distributed filesystem. Since all the workers are kept in sync in terms of training epochs and steps, other workers would need to wait for the failed or preempted worker to restart in order to continue.

In the `MultiWorkerMirroredStrategy`, for example, if a worker gets interrupted, the whole cluster pauses until the interrupted worker is restarted. Other workers will also restart, and the interrupted worker rejoins the cluster. Then, there needs to be a way for every worker to pick up its former state, thereby allowing the cluster to get back in sync to allow for training to proceed smoothly. For example, Keras provides this functionality in the `BackupAndRestore` callback.

Efficient Input Pipelines

Accelerators are a key part of high-performance modeling, training, and inference. But accelerators are also expensive, so it’s important to use them efficiently. This means keeping them busy, which requires you to supply them with data quickly enough. That’s why efficient input pipelines are important in high-performance modeling.

Input Pipeline Basics

Input pipelines are an important part of many *training* pipelines, but there are often similar requirements for *inference* pipelines as well. In the larger context of a training pipeline, such as a TensorFlow Extended (TFX) training pipeline, a high-performance input pipeline would be part of the Trainer component, and possibly other components such as Transform, that may often need to do quite a bit of work on the data.

In improving input pipeline efficiency, it is important to understand the basic steps that input pipelines take to ingest data. You can view input pipelines as an extract, transform, load (ETL) process. The first step of this process involves extracting data from datastores that may be either local or remote, such as hard drives, solid-state drives (SSDs), cloud storage, and the Hadoop Distributed File System (HDFS).

In the second step, data often needs to be preprocessed or transformed. This includes shuffling, batching, and repeating data, as well as applying element-wise transformations. If these transformations take too long, your accelerators might be underutilized while waiting for data. In addition, the way you order these transformations may have an impact on your pipeline's performance. This is something you need to be aware of when using any data transformation (map, batch, shuffle, repeat, etc.).

The third step of an input pipeline involves loading the preprocessed data into the model, which may be training on a CPU, GPU, or TPU, and starting training. A key requirement for high-performance input pipelines is to parallelize the processing of data across the various systems to try to make the most efficient use of the available compute, I/O, and network resources. Especially for more expensive components such as accelerators, you want to keep them as busy as possible, and not waiting for data.

Input Pipeline Patterns: Improving Efficiency

Let's look at a typical pattern that is easy to fall into, and one that you really want to avoid.

In Figure 7-2, key hardware components including CPUs and accelerators sit idle, waiting for the previous steps to complete. If you think about it, ETL is a good mental model for data performance. To give you some intuition on how pipelining can be carried out, assume that each phase of

ETL uses different hardware components in your system. The extract phase is exercising your disk, or your network if you're loading from a remote system. Transform typically happens on the CPU and can be very CPU hungry. The load phase is exercising the direct memory access (DMA) subsystem and the connections to your accelerator—probably a GPU or a TPU.

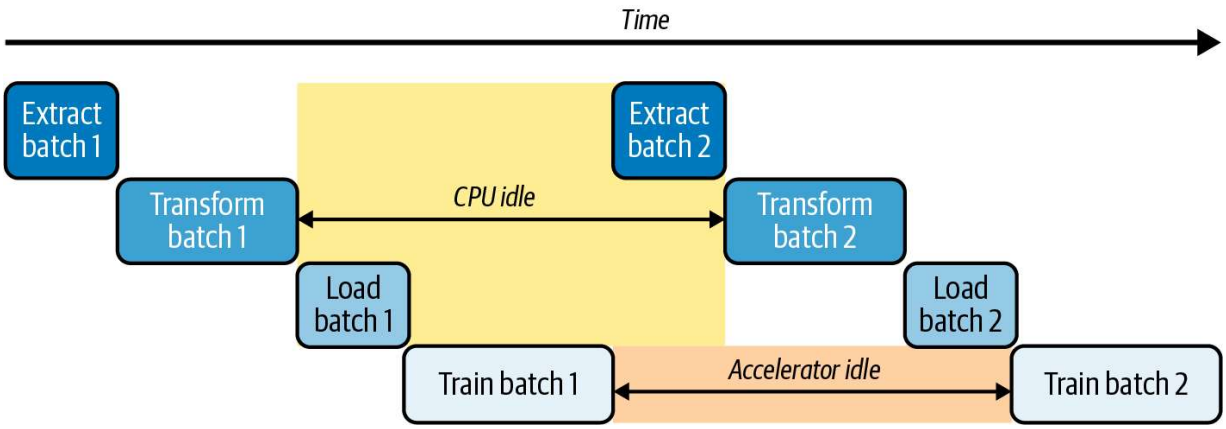

Figure 7-2. An inefficient input pipeline

The approach shown in Figure 7-3 is a much more efficient pattern than the one in Figure 7-2, although it's still not optimal. In practice, though, this kind of pattern may be difficult to optimize further in many cases.

As Figure 7-3 shows, by parallelizing operations you can overlap the different parts of ETL using a technique known as *software pipelining*. With software pipelining, you're extracting data for step 5, while you're transforming for step 4, while you're loading data for step 3, while you're training for step 2, all at the same time. This results in a very efficient use of your compute resources.

As a result, your training is much faster and your resource utilization is much higher. Notice that now there are only a few instances where your hard drive and CPU are actually sitting idle.

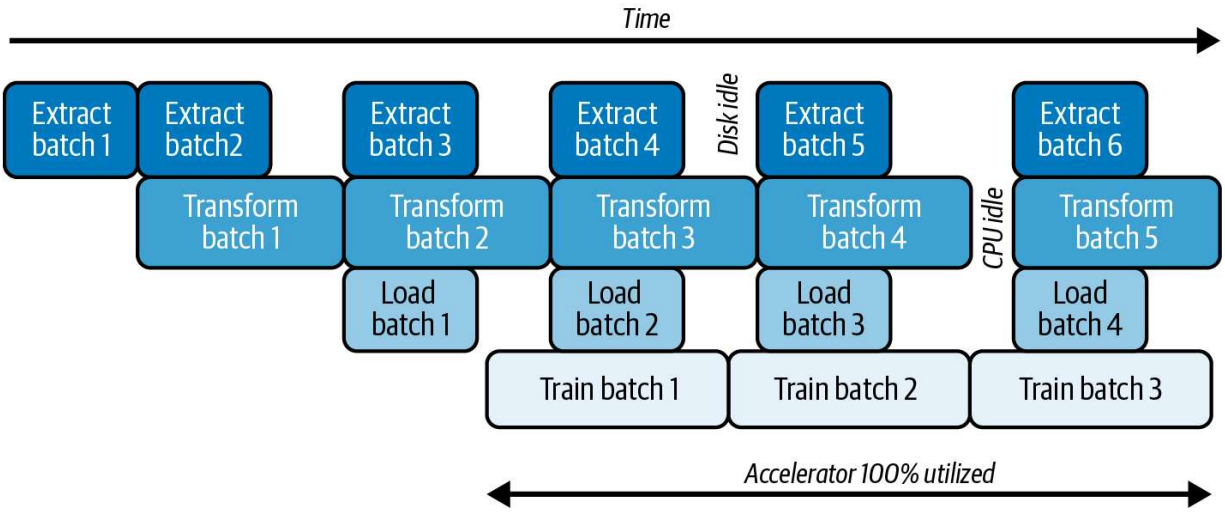

Figure 7-3. A more efficient input pipeline

Optimizing Your Input Pipeline with TensorFlow Data

So, how do you optimize your data pipeline in practice? There are a few basic approaches that could potentially be used to accelerate your pipeline. Prefetching is a good practice, where you begin loading data for the next step before the current step completes. Other techniques involve parallelizing data extraction and transformation. Caching the dataset to get started with training immediately once a new epoch begins is also very effective, when you have enough cache. Finally, you need to be aware of

how you order these optimizations in your pipeline to maximize the pipeline's efficiency.

One framework that can help with these approaches is TensorFlow Data (TF Data). Let's consider TF Data as an example of how to design an efficient input pipeline.

Prefetching

With prefetching, you overlap the work of a "producer" with the work of a "consumer." While the model is executing step *S*, the input pipeline is reading the data for step *S+1*. This reduces the total time it takes for a step to either train the model or extract data from disk (whichever takes the most time).

The TF Data API provides the `tf.data.Dataset.prefetch` transformation. You can use this to decouple the time when data is produced from the time when data is consumed. This transformation uses a background thread and an internal buffer to prefetch elements from the input dataset ahead of time, before the elements are requested. Ideally, the number of elements to prefetch should be equal to, or possibly greater than, the number of batches consumed by a single training step. You could manually tune this value, or you could set it to `tf.data.experimental.AUTOTUNE`, which will configure the TF Data runtime to optimize the value dynamically at runtime.

In a real-world setting, the input data may be stored remotely (e.g., on Google Cloud Storage or HDFS). A dataset pipeline that works well when reading data locally might become bottlenecked on I/O or network bandwidth when reading data remotely because of the following differences between local and remote storage:

Time-to-first-byte

Reading the first byte of a file from remote storage can take orders of magnitude longer than from local storage.

Read throughput

While remote storage typically offers large aggregate bandwidth, reading a single file might only be able to utilize a small fraction of this bandwidth.

To reduce data extraction overhead, the `tf.data.Dataset.interleave` transformation is used to parallelize the data loading step, including interleaving the contents of other datasets. The number of datasets to overlap is specified by the `cycle_length` argument, while the level of parallelism is set with the `num_parallel_calls` argument.

Similar to the prefetch transformation, the interleave transformation supports `tf.data.experimental.AUTOTUNE`, which will delegate the decision about what level of parallelism to use to the TF Data runtime.

Parallelizing data transformation

When preparing data, input elements may need to be preprocessed. For example, the TF Data API offers the `tf.data.Dataset.map` transformation, which applies a user-defined function to preprocess each element of the input dataset. Element-wise preprocessing can be parallelized across multiple CPU cores. Similar to the prefetch and interleave transformations, the map transformation provides the `num_parallel_calls` argument to specify the level of parallelism.

Choosing the best value for the `num_parallel_calls` argument depends on your hardware, the characteristics of your training data (such as its size and shape), the cost of your map function, and what other processing is happening on the CPU at the same time. A simple heuristic is to use the number of available CPU cores. However, as with the prefetch and interleave transformations, the map transformation in TF Data supports `tf.data.experimental.AUTOTUNE`, which will delegate the decision about what level of parallelism to use to the TF Data runtime.

Caching

The `tf.data.Dataset` transformation includes the ability to cache a dataset, either in memory or on local storage. In many instances, caching is advantageous and leads to increased performance. This will save some operations, such as file opening and data reading, from being executed

during each epoch. When you cache a dataset, the transformations before caching (e.g., the file opening and data reading) are executed only during the first epoch. The next epochs will reuse the cached data.

Let’s consider two scenarios for caching:

- If the user-defined function passed into the map transformation is expensive, it makes sense to apply the cache transformation after the map transformation, as long as the resulting dataset can still fit into memory or local storage.
- If the user-defined function increases the space required to store the dataset beyond the cache capacity, either apply it after the cache transformation or consider preprocessing your data before your training job to reduce resource requirements.

Now that we have discussed the basics of distributed training and efficient input pipelines, we will close by discussing the rise of giant neural nets and high-performance modeling strategies that can help train such models efficiently.

Training Large Models: The Rise of Giant Neural Nets and Parallelism

In recent years, the size of ML datasets and models has been continuously increasing, allowing for improved results on a wide range of tasks including speech recognition, visual recognition, and language processing. Recent advances with generative AI (GenAI) models such as Gemini, GPT-4o, and Claude 3.5 in particular have shown the potential of large models. At the same time, hardware accelerators such as GPUs and TPUs have also been increasing in power, but at a significantly slower pace. The gap between model growth and hardware improvement has increased the importance of parallelism.

Parallelism in this context means training a single ML model on multiple hardware devices. Some model architectures, especially small models, are conducive to parallelism and can be divided quite easily among hardware devices. In enormous models, synchronization costs lead to degraded performance, preventing them from being used.

The blog post introducing the open source library GPipe (see "Pipeline Parallelism to the Rescue?") highlighted the enormous increase in model sizes in recent years in achieving performance gains. In that post, the author points to the example of the winners of the ImageNet Large Scale Visual

Recognition Challenge, highlighting the 36-fold increase in the number of parameters between the 2014 and 2017 winners of that challenge.

Massive numbers of weights and activation parameters require massive memory storage. With hardware advances alone not keeping pace with the rapid growth of model sizes, the rise of giant neural nets has only increased the need for effective strategies for addressing memory constraints. But in some ways, this is not a new problem, as we'll discuss next.

Potential Solutions and Their Shortcomings

In this section, we'll examine some older approaches for meeting the needs created by the rise of giant neural nets, and look at the possible shortcomings of such approaches. We'll close by discussing pipeline parallelism and how it can address some of these shortcomings.

Gradient accumulation

One strategy that can overcome problems with insufficient GPU memory is gradient accumulation. *Gradient accumulation* is a mechanism to split full batches into several mini batches. During backpropagation, the model isn't updated with each mini batch, and instead the gradients are accumulated. When a full batch completes, the accumulated gradients of all the previous mini batches are used for backpropagation to update the model. This process is as effective as using a full batch for training the network, since model parameters are updated the same number of times.

Swapping

The second approach is swapping. Here, since there isn't enough storage on the accelerator, you copy activations back to the CPU or memory and then back to the accelerator. The problem with this approach is that it's slow, and the communication between the CPU or memory and the accelerator becomes the bottleneck.

Parallelism, revisited in the context of giant neural nets

Returning to our discussion of distributed training, the basic idea is to split the computation among multiple workers. You've already seen two ways to do distributed training: data parallelism and model parallelism. Data parallelism splits the *input data* across workers. Model parallelism splits the *model* across workers.

In data parallelism, different workers or GPUs work on the same model, but deal with different data. The model is replicated across a number of workers, and each worker performs the forward and backward passes. When it finishes the process, it synchronizes the updated model weights with the other devices and calculates the updated weights of the entire mini batch.

With data parallelism, the input dataset is partitioned across multiple GPUs. Each GPU maintains a full copy of the model and trains on its own partition of data while periodically synchronizing weights with other GPUs, using

either collective communication primitives or parameter servers. The frequency of parameter synchronization affects both statistical and hardware efficiency.

Synchronizing at the end of every mini batch reduces the staleness of weights used to compute gradients, ensuring good statistical efficiency. Unfortunately, this requires each GPU to wait for gradients from other GPUs, which significantly lowers hardware efficiency. Communication stalls are inevitable in data-parallel training due to the structure of neural networks, and the result is that communication can often dominate total execution time. Rapid increases in accelerator speeds further shift the training bottleneck toward communication.

And there’s another problem. Accelerators have limited memory and limited communication bandwidth with the host machine. This means model parallelism is needed for training bigger models on accelerators by dividing the model into partitions and assigning different partitions to different accelerators.

In model parallelism, workers only need to synchronize the shared parameters, usually once for each forward or backpropagation step. Also, larger models aren’t a major concern, since each worker operates on a subsection of the model using the same training data. When using model parallelism in training, the model is divided across K workers, with each worker holding a part of the model. A naive approach to model parallelism

is to divide an N-layered neural network into K workers by simply hosting N/K layers on each worker. More sophisticated methods ensure that each worker is similarly busy by analyzing the computational complexity of each layer. Standard model parallelism enables training of larger neural networks, but it suffers from a large hit in performance since workers are constantly waiting for each other and only one can perform updates at a given time.

In sum, there are issues in achieving high performance with either data parallelism or model parallelism in the neural network context, with each approach having its own shortcomings.

Pipeline Parallelism to the Rescue?

The issues with data parallelism and model parallelism have led to the development of pipeline parallelism. Figure 7-4 shows an example of pipeline parallelism using four accelerators (devices 0–3). The forward passes for training the model are shown as $F_{0\text{-}3}$, and the backpropagation of gradients is shown as $B_{0\text{-}3}$. As the diagram shows, a naive model parallelism strategy leads to severe underutilization due to the sequential nature of the model: only one accelerator is active at a time.

To enable efficient training across multiple accelerators, you need to find a way to partition a model across different accelerators and automatically split a mini batch of training examples into smaller microbatches, as shown

in Figure 7-5. By pipelining the execution across microbatches, accelerators can operate in parallel. In addition, gradients are consistently accumulated across the microbatches so that the number of partitions does not affect the model quality.

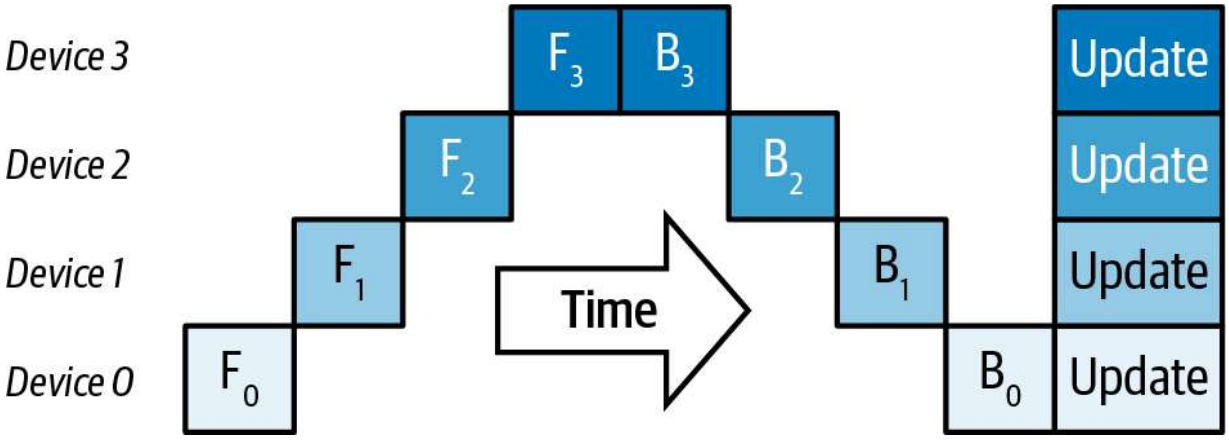

Figure 7-4. Naive model parallelism (source: Huang et al., "GPipe: Easy Scaling with Micro-Batch Pipeline Parallelism," 2019)

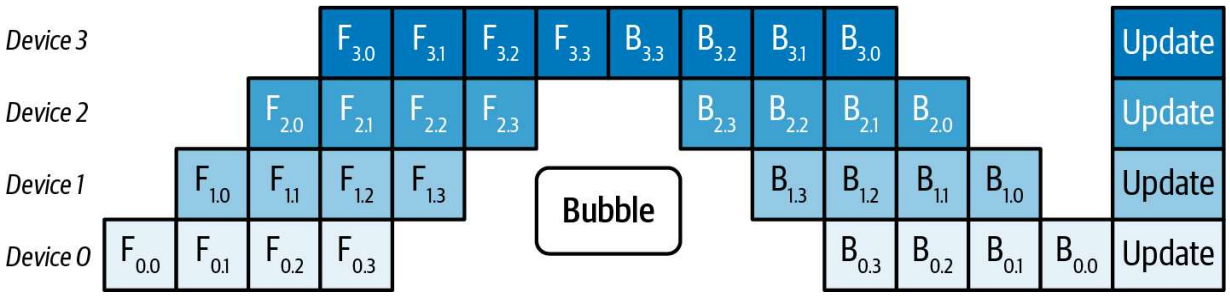

Figure 7-5. More efficient training with microbatches (source: Huang et al., "GPipe: Easy Scaling with Micro-Batch Pipeline Parallelism," 2019)

Google's GPipe is an open source library for efficiently training large-scale models using pipeline parallelism. In Figure 7-5, GPipe divides the input mini batch into smaller microbatches, enabling different accelerators to work on separate microbatches at the same time. GPipe essentially presents

a new way to approach model parallelism that allows training of large models on multiple hardware devices with an almost one-to-one improvement in performance. It also helps models include significantly more parameters, allowing for better results in training. PipeDream from Microsoft also supports pipeline parallelism. GPipe and PipeDream are similar in many ways.

Pipeline parallelism frameworks such as GPipe and PipeDream integrate both data and model parallelism to achieve high efficiency and preserve model accuracy. They do that by dividing mini batches into smaller microbatches and allowing different workers to work on different microbatches in parallel. As a result, they can train models with significantly more parameters on a given set of accelerators. See the Google Research blog post "Introducing GPipe, an Open Source Library for Efficiently Training Large-scale Neural Network Models" for more information about GPipe and the memory and training efficiency gains available through its use.

Conclusion

This chapter has given you a flavor for some of the issues and techniques that are involved in high-performance modeling. This is an area of intense development as the demands for efficient training of extremely large GenAI models such as GPT-4o and Gemini place huge and expensive demands on

computing resources and budgets. New advances in the major areas mentioned in this chapter—distributed training, efficient input pipelines, and training large models—are arriving on an almost weekly basis. With the background in this chapter, you'll be able to better understand and evaluate these advances as the field progresses.