Designing High Performance ML Systems

Laurence Moroney

Hi, I'm Laurence and I'm a developer advocate on Google Brain, focused on TensorFlow.

In this course, you are learning the considerations behind architecting and implementing production machine learning systems.

One key consideration is, of course, performance.

Learn how to...

Identify performance considerations for ML models

Choose appropriate ML infrastructure

Select a distribution strategy

In this module, you will learn how to identify performance considerations for machine learning models.

Machine learning models are not all identical. For some models, you will be focused on improving I/O performance, and on others, you will be focused on squeezing out more computational speed.

Depending on what your focus is, you will need different ML infrastructure -- whether you decide to scale out, with multiple machines or scale up on a single machine with a GPU and TPU. Sometimes, you'll do both, by using a machine with multiple accelerators attached to it.

It's not just a hardware choice. The hardware you select will also inform your choice of a distribution strategy.

This is a fast evolving area, and some of the things we show you are still in early alpha stage and involve contributions from the TensorFlow community.

Agenda

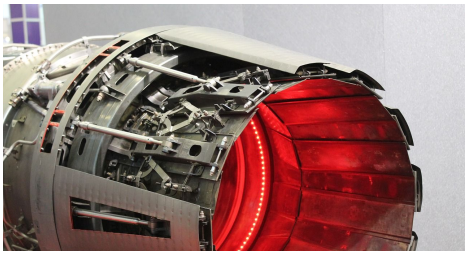
Distributed training

Faster input pipelines

Inference

We will start by talking about what high performance means in this context, and providing a high-level overview of distributed training architectures.

High Performance ML



What does high-performance machine learning mean to you?

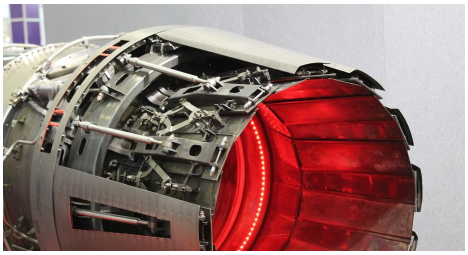
Does it mean “powerful”? the ability to handle large datasets?

Doing it as fast as possible?

The ability to train for long periods of time?

Achieving the best possible accuracy?

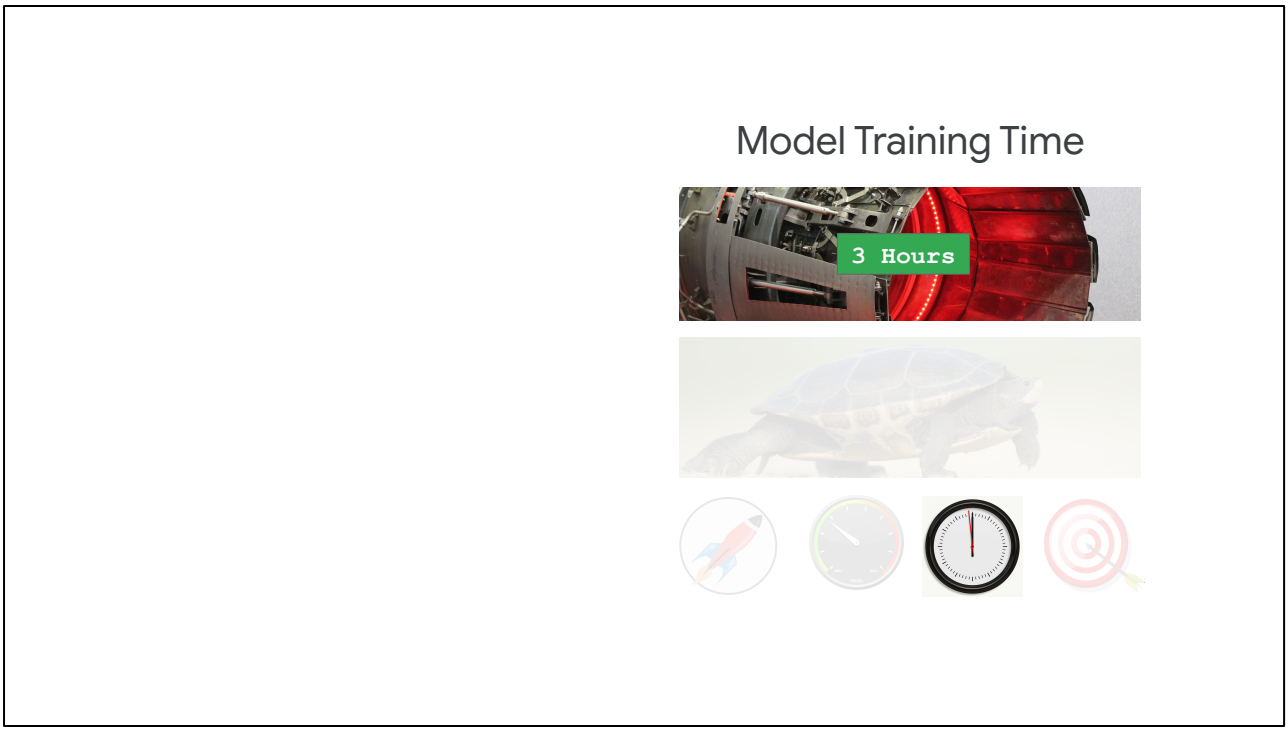
Model Training Time



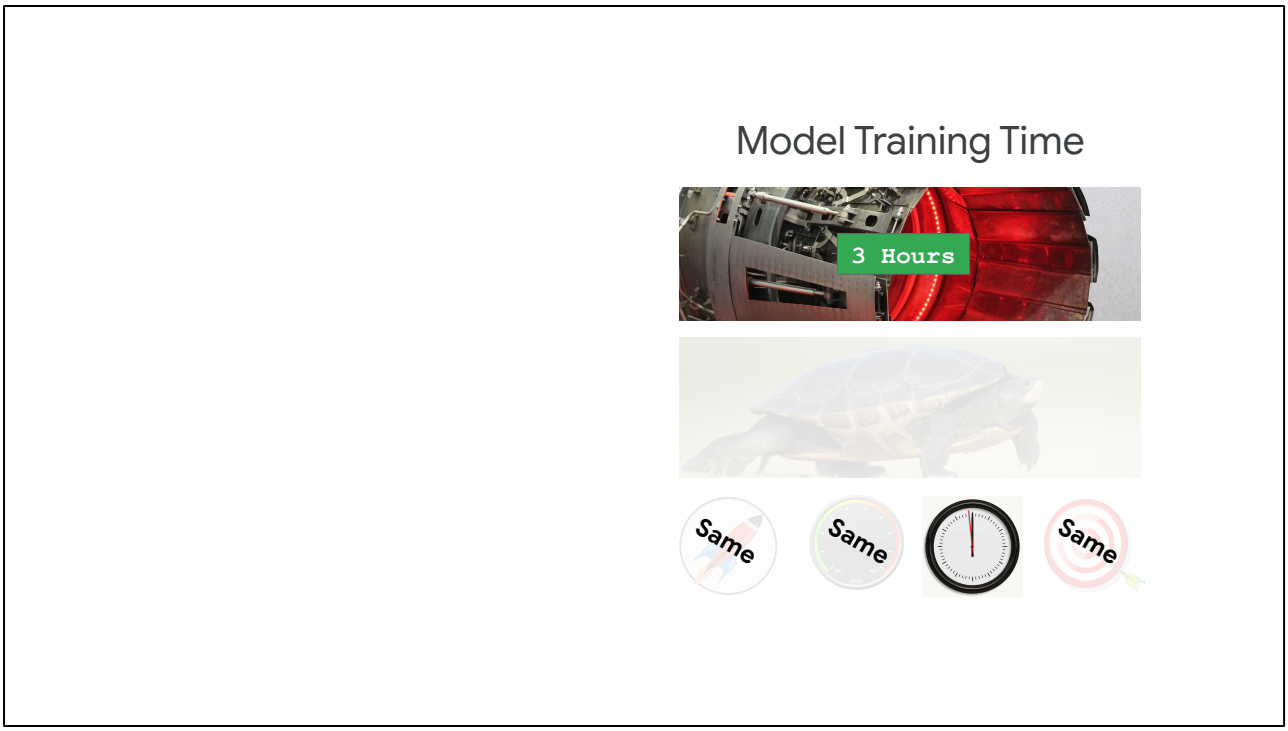
One key aspect is the time taken to train a model.

Model Training Time

If it takes 6 hours to train a model on some hardware/software architecture



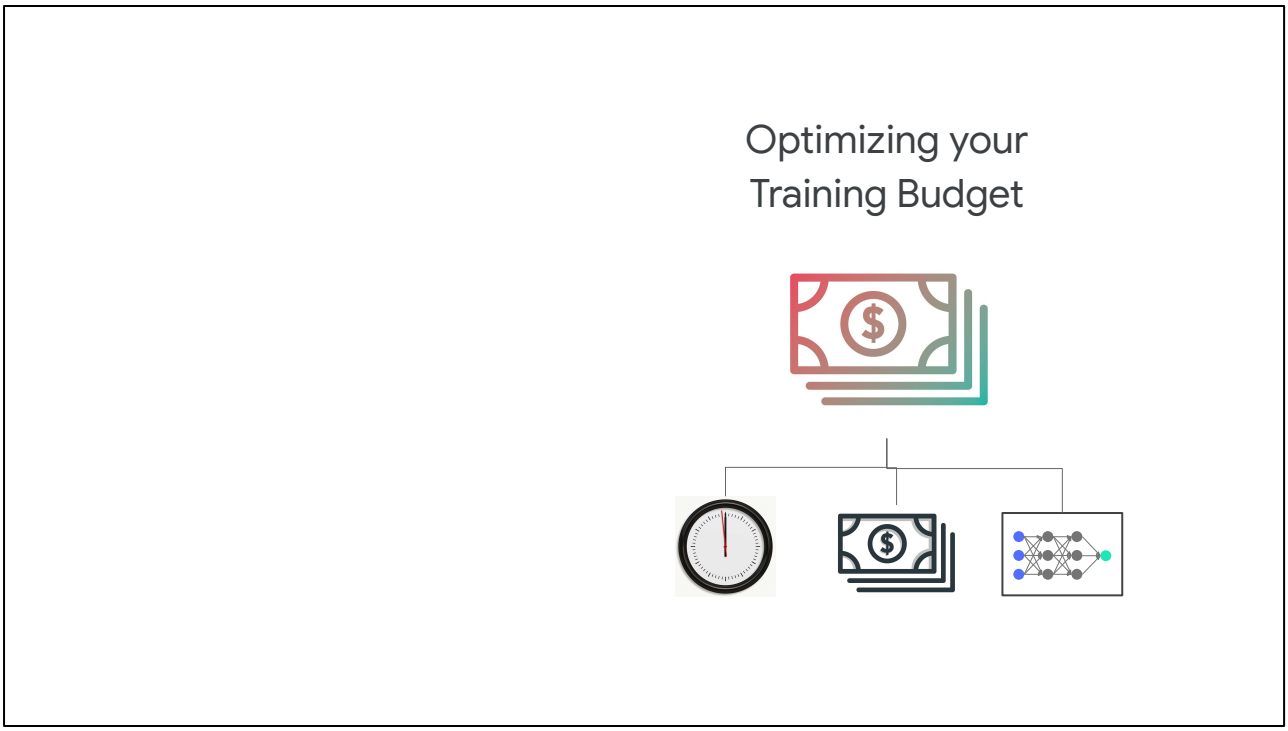
but only 3 hours to train the same model to the same accuracy on some other hardware/software architecture, I think we will all agree that the second architecture is twice as performant as the first one.



Notice that I said “train the model to the same accuracy”. Throughout this module, we will assume that we are talking of models that have the same accuracy or RMSE or whatever your evaluation measure is. Obviously, when we talk about high-performance ML models, accuracy is important.

We just aren’t going to consider that in this module. The rest of the courses in this specialization will look at how to build more accurate ML models, and there we will be looking at model architectures that will help us get to a desired accuracy.

Here, in this course, we will look solely at infrastructure performance.

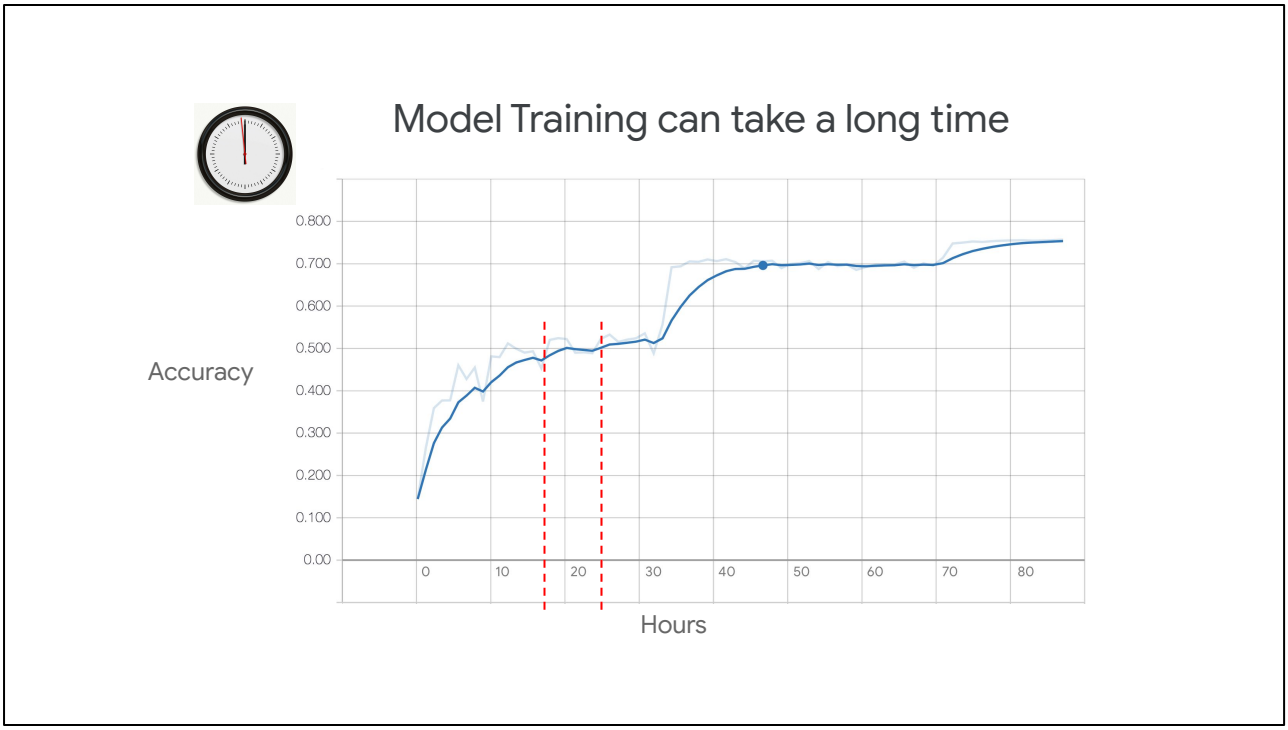


Besides time to train, there is another aspect. Budget.

You often have a training budget. You might be able to train faster on better hardware, but the hardware might cost more, and so you might make the explicit choice to train on slightly slower infrastructure.

When it comes to your training budget, you have three considerations, three levers that you can adjust:

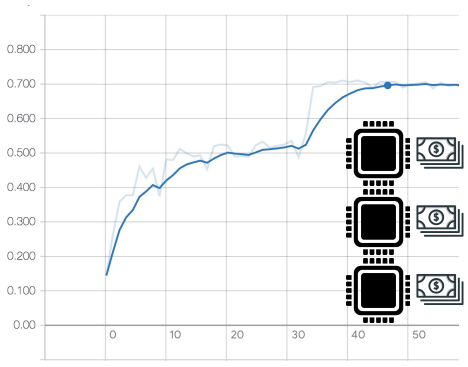
- Time
- Cost
- Scale



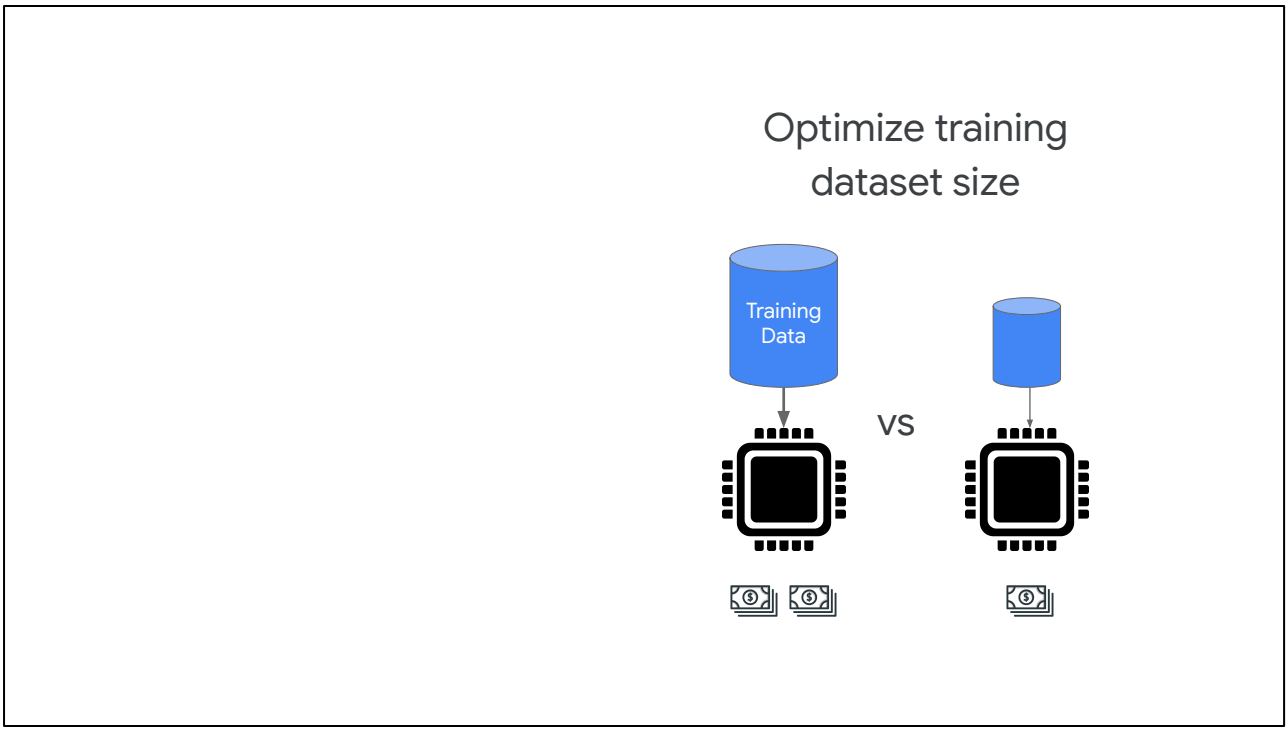
How long are you willing to spend on the model training? This might be driven by the business use case. If you are training a model everyday so as to recommend products to users the next day, then your training has to finish within 24 hours. Realistically, you will need to time to deploy, to A/B test, etc.

So, your actual budget might be only 18 hours.

Analyze Benefit of Model vs Running Cost

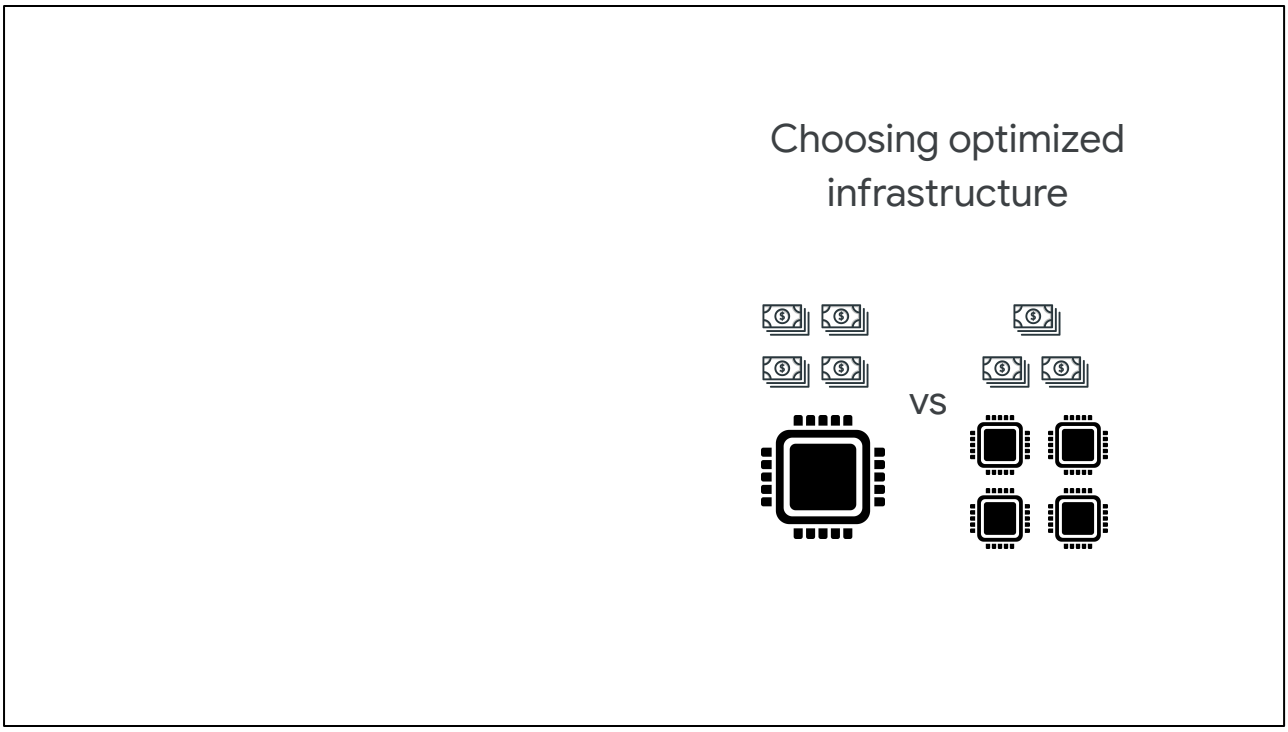


How much are you willing to spend on model training in terms of computing costs? This, too, is a business decision. You don't want to train for 18 hours every day if the incremental benefit is not sufficient.

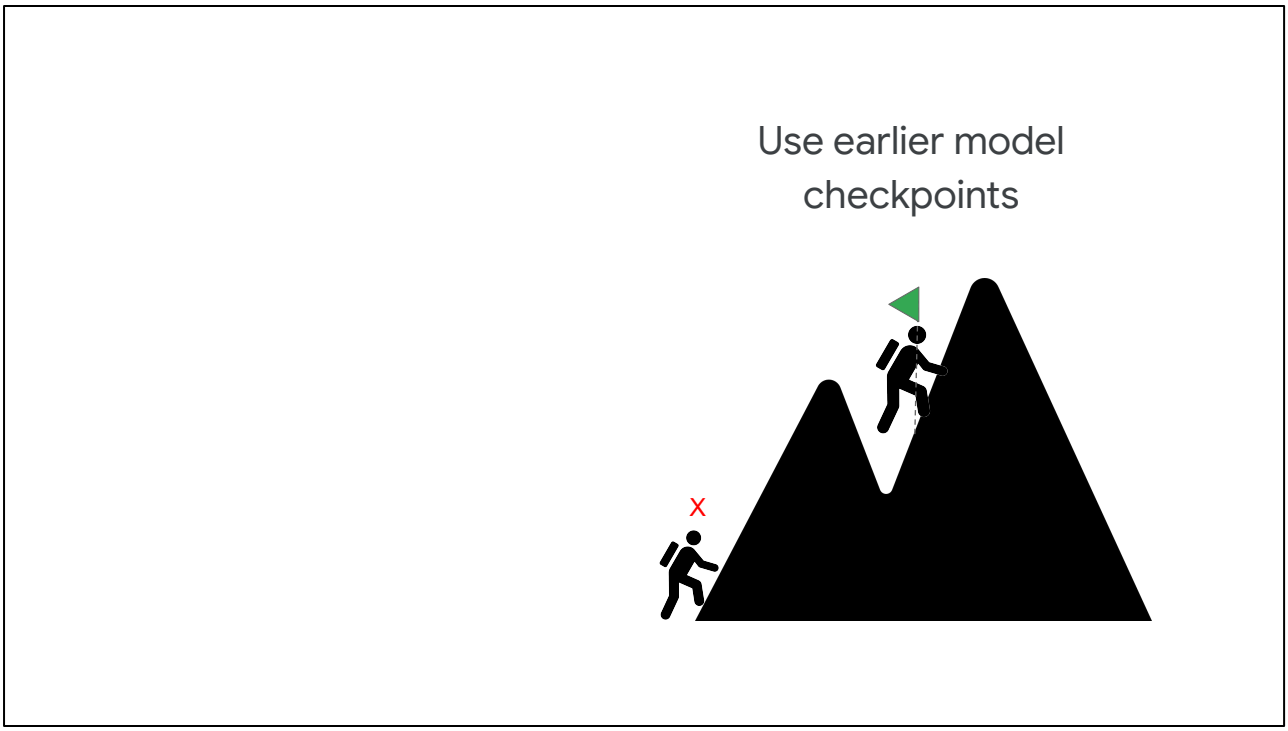


Scale is another aspect of your budget. Models differ in terms of how computationally expensive they are.

Even keeping to the same model, you have a choice of how much data you are going to train on -- generally, the more data, the more accurate the model, but there are diminishing returns to larger and larger data sizes, so, your time and cost budget might dictate the data set size.



Similarly, you often have a choice between training on a single, more expensive machine or multiple, cheaper machines. But to take advantage of this, you may have to write your code somewhat differently. That is another aspect of scale.



Also, you have the choice of starting from an earlier model checkpoint, and training for just a few steps. Typically, this will converge faster than training from scratch each time. This compromise might allow you to reach the desired accuracy faster and cheaper.

Tuning Performance to reduce training time, reduce cost, and increase scale

Constraint	Input / Output	CPU	Memory
Commonly Occurs	Large inputs Input requires parsing Small models	Expensive computations Underpowered Hardware	Large number of inputs Complex model
Take Action	Store efficiently Parallelize reads Consider batch size	Train on faster accel. Upgrade processor Run on TPUs Simplify model	Add more memory Use fewer layers Reduce batch size

In addition, there are ways to tune performance to reduce the time, reduce the cost, or increase the scale.

In order to understand what these are, it helps to understand that model training performance will be bound by one of three things:

- input/output -- how fast can you get data into the model in each training step?
- Cpu -- how fast can you compute the gradient in each training step?
- Memory -- how many weights can you hold in memory, so that you can do the matrix multiplications in-memory on the GPU or TPU?

Your ML training will be I/O bound if the number of inputs is large, heterogeneous (requires parsing), or if the model is so small that the compute requirements are trivial. This also tends to be the case if the input data is on a storage system with low throughput.

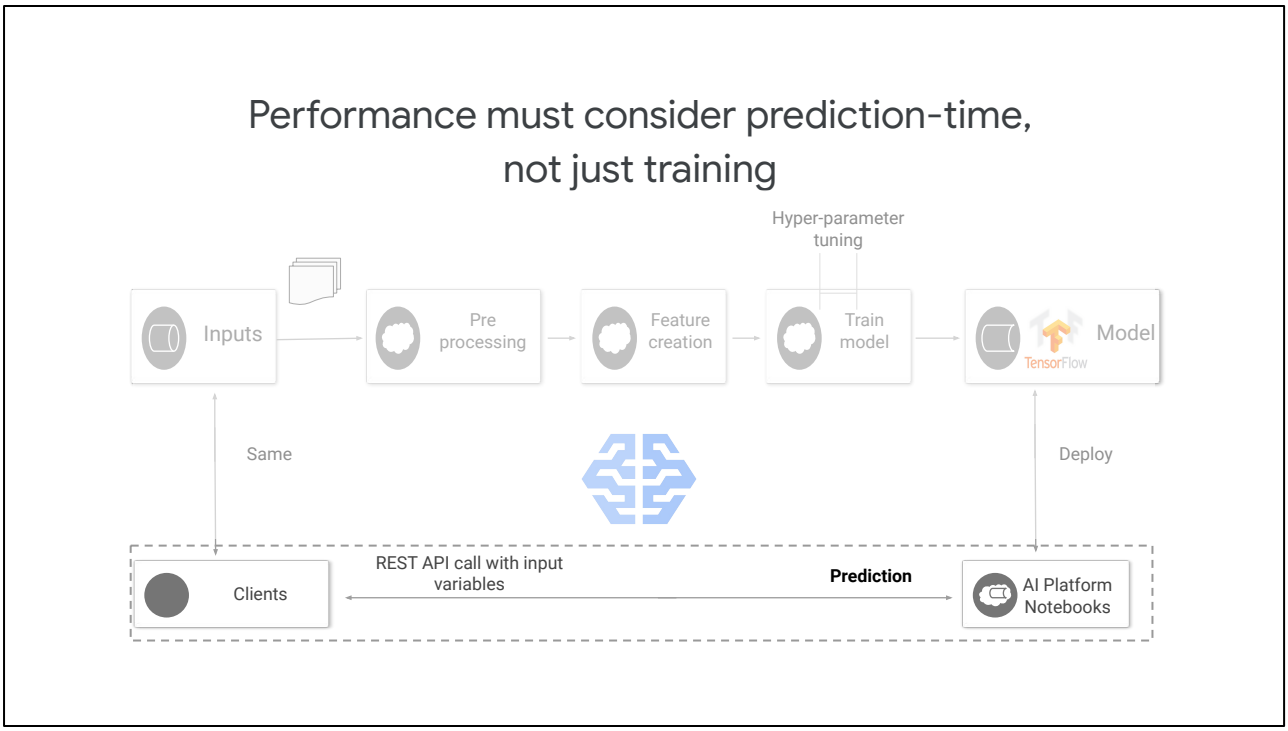
Your ML training will be CPU bound if the I/O is simple, but the model involves lots of expensive computations. You will also encounter this situation if you are running a model on underpowered hardware.

Your ML training might be memory-bound if the number of inputs is large or if the model is complex and has lots of free parameters. You will also face memory limitations if your accelerator doesn't have enough memory.

So, knowing what you are bound by, you can look at how to improve performance. If you are I/O bound, look at storing the data more efficiently, on a storage system with higher throughput, or parallelizing the reads. Although it is not ideal, you might consider reducing the batch size so that you are reading less data in each step.

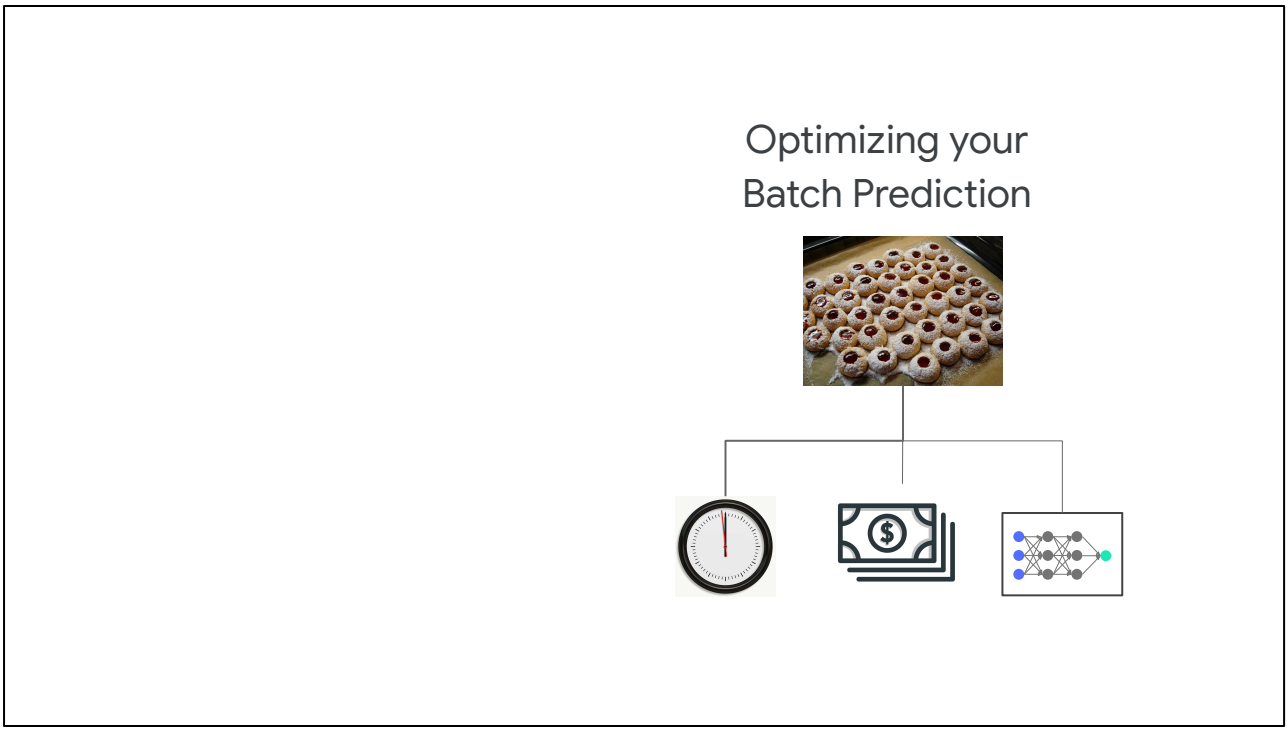
If you are CPU-bound, see if you can run the training on a faster accelerator. GPUs keep getting faster, so move to a newer generation processor. And on Google Cloud, you also have the option of running on TPUs. Even if it is not ideal, you might consider using a simpler model, a less computationally expensive activation function or simply just train for fewer steps.

If you are memory-bound, see if you can add more memory to the individual workers. Again, not ideal, but you might consider using fewer layers in your model. Reducing the batch size can also help with memory-bound ML systems.



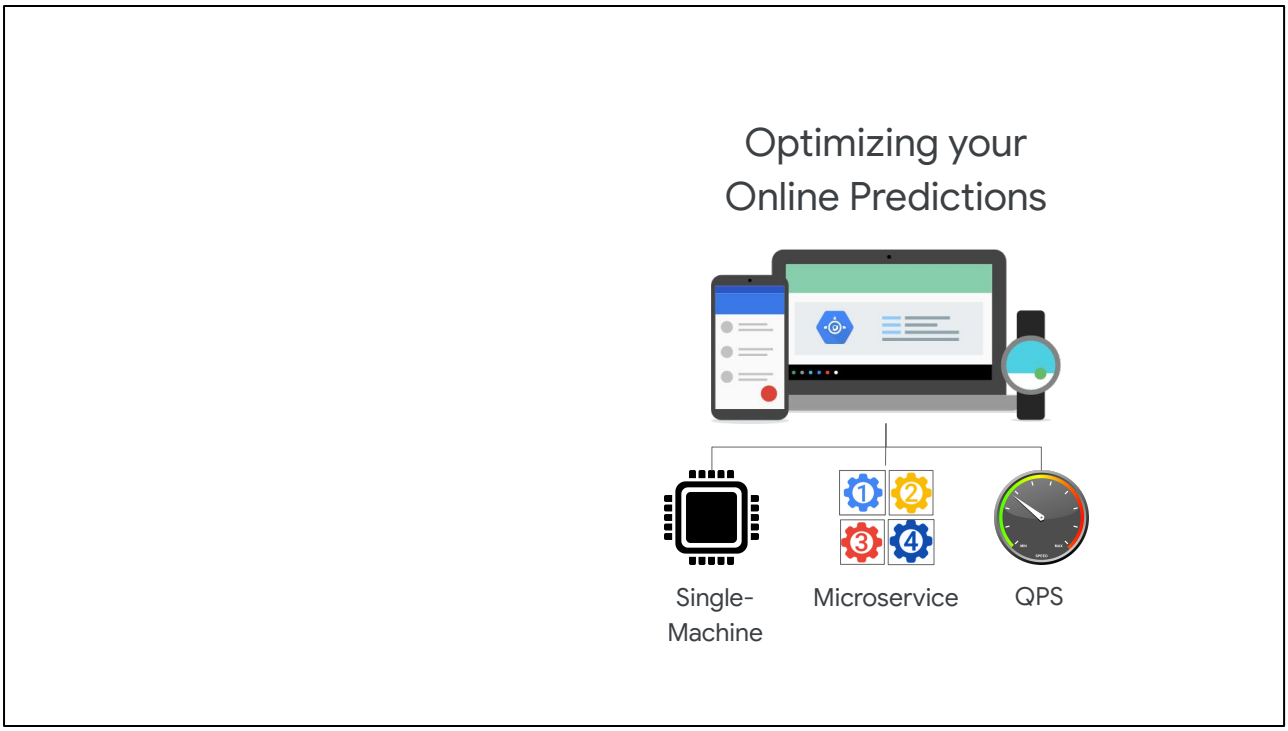
We have talked about time to *train*, but there is another aspect to performance. Predictions.

During inference, you have performance considerations as well.



If you are doing batch prediction, the considerations are very similar to that of training. You are concerned with

- **Time**: How long does it take you to do all the predictions? This might be driven by a business need as well -- if you are doing product recommendations for the next day, you might want recommendations for the top 20% of users precomputed and available in about 5 hours, if it takes 18 hours to train …
- **Cost**: What predictions you are doing, and how much you precompute is going to driven by cost considerations
- **Scale**: do you have to do this all on one machine, or can you distribute it to multiple workers? What kind of hardware are on these workers? Do they have GPUs?



If you are doing online prediction, the performance considerations are quite different. This is because the end-user is waiting for the prediction. How is it different?

- You typically can not distribute the prediction graph; instead, you carry out the computation for one end-user on one machine
- However, you almost always scale out the predictions on to multiple workers. Essentially, each prediction is handled by a microservice, and you can replicate and scale out the predictions using Kubernetes or AppEngine -- Cloud ML Engine predictions are a higher-level abstraction, but they are equivalent to doing this.
- The performance consideration is not how many training steps can you carry out per minute, but how many queries you can handle per second. Queries Per Second or QPS. That's the performance target you need to hit.

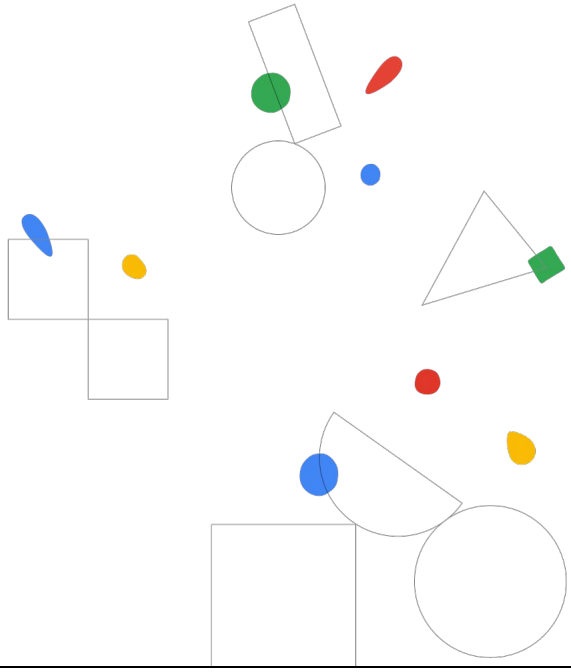
When you design for high performance, you want consider training and prediction separately, especially if you will be doing online predictions.

As I kind-of suggested in my line about precomputing batch predictions for the top 20% of users, and handling the rest of your users via online prediction, performance considerations will often involve striking the right balance. And ultimately, you will know the exact tradeoff (is it 20% or 10% or 25%, that you need to do) only after you build your system and measure things. However, unless you plan to be able to do both batch predictions and online predictions, you will be stuck with a solution that doesn't meet your needs.

The idea behind this module, and this course in general, is so that you are aware of the possibilities. Once you are aware that it can be done, it's not that difficult to accomplish -- the technical part is usually quite straightforward. Especially if you are using TensorFlow on a capable cloud platform.

Google Cloud

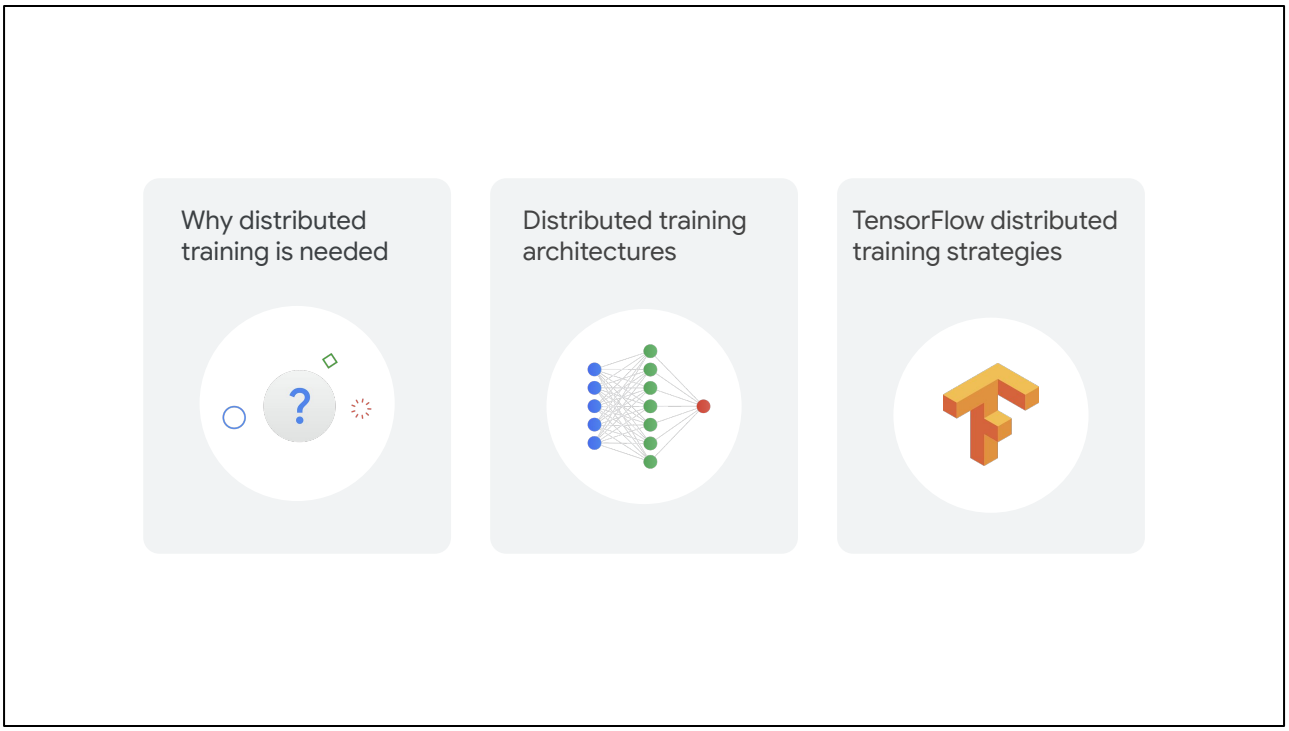
Why distributed training is needed



Module 03

Designing high-performance ML systems

In this module, we'll explore how to run a distributed training job with TensorFlow.

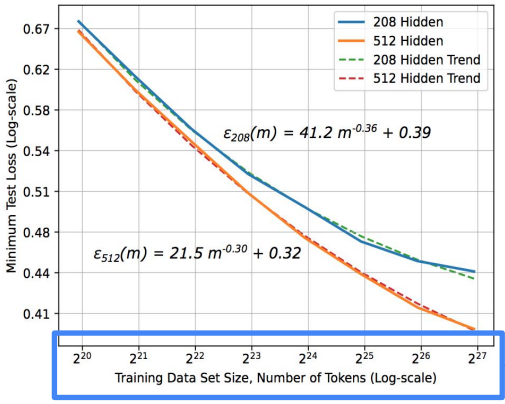


We'll begin with understanding why distributed training is needed.

After that, we'll explore distributed training architectures.

Then lastly, we'll provide an overview of TensorFlow distributed training strategies.

Deep learning works because datasets are large



X-axis is logarithmic

For every doubling in the size of the data, the error rate falls linearly

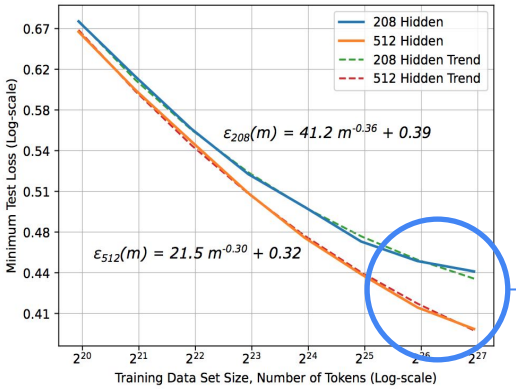
The unreasonable effectiveness of data
https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/35179.pdf

Deep Learning scaling is predictable, empirically
https://arxiv.org/abs/1712.00409

Deep learning works because datasets are large.

Notice that the x-axis here is logarithmic. For every doubling in the size of the data, the error rate falls linearly.

Deep learning works because datasets are large



More complex model

Compute required grows over time

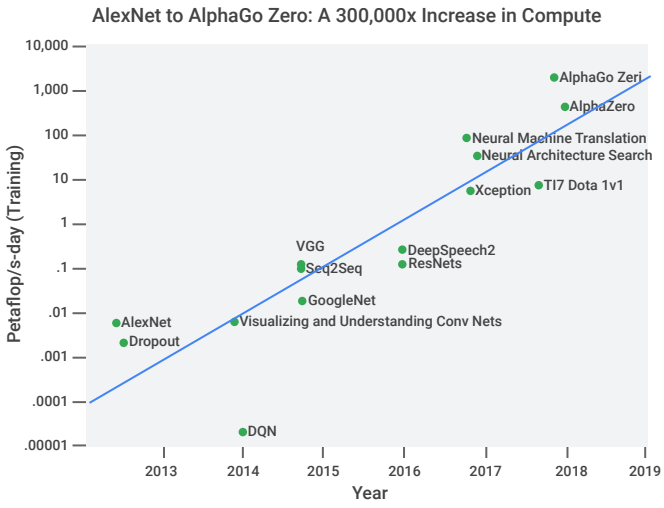
The unreasonable effectiveness of data
https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/35179.pdf

Deep Learning scaling is predictable, empirically
https://arxiv.org/abs/1712.00409

A more complex model also helps -- that is the jump from the blue line to the orange line -- but more data is even more helpful in this situation.

As a consequence of both of these trends, in terms of larger data sizes and more complex models, the compute required to build state-of-the-art models has grown over time. This growth is exponential as well.

The compute required keeps increasing

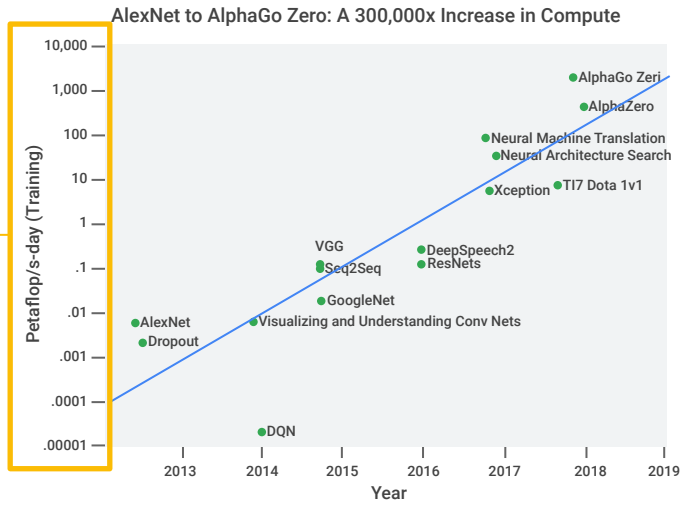


https://blog.openai.com/ai-and-compute/

Each y-axis tick on this graph

The compute required keeps increasing

10x increase in computational need



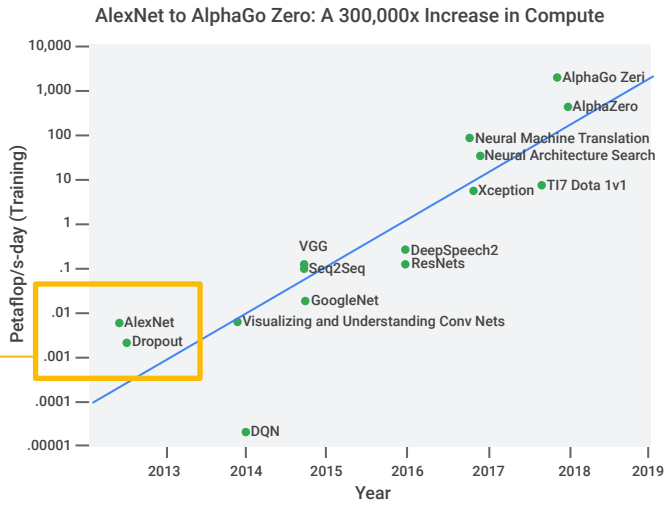
https://blog.openai.com/ai-and-compute/

shows a 10x increase in computational need.

The compute required keeps increasing

AlexNet

<0.01 petaflops per second-day in compute per day for training



https://blog.openai.com/ai-and-compute/

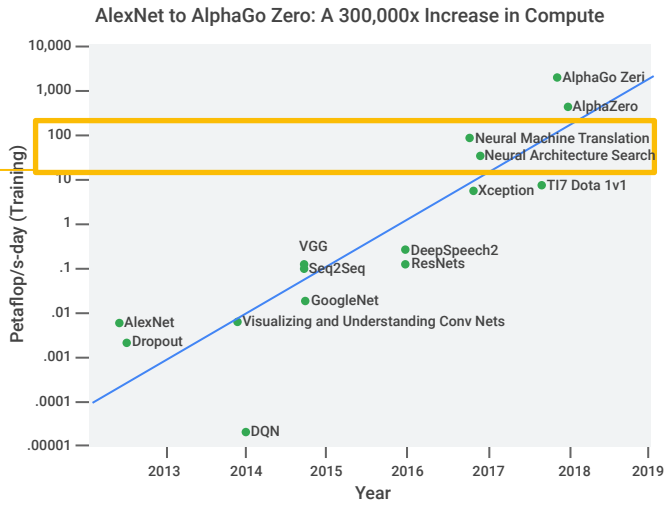
AlexNet, which started the deep learning revolution in 2013, required less than 0.01 petaflops per second-day in compute per day for training.

The compute required keeps increasing

Neural Architecture Search

100 petaflops per second-day

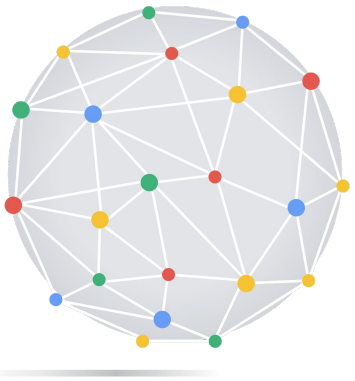
(1000x more compute than you needed for AlexNet)



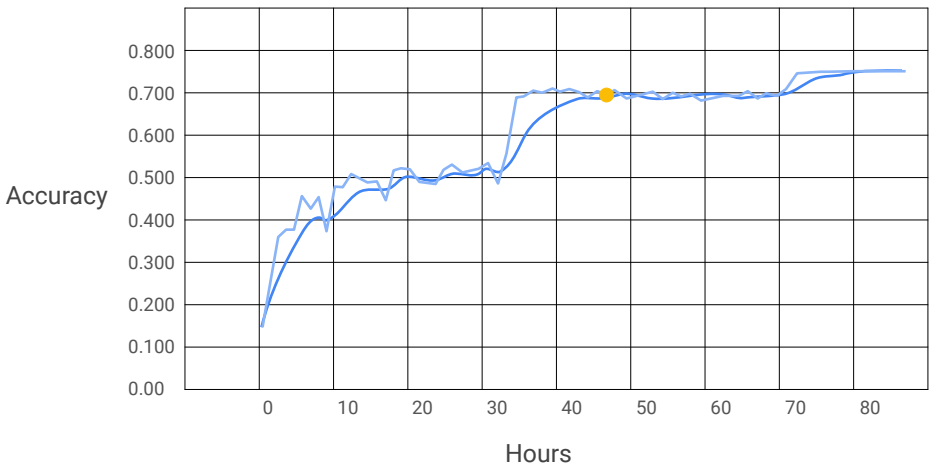
https://blog.openai.com/ai-and-compute/

By the time you get to Neural Architecture Search, the learn-to-learn model published by Google in 2017, you need about 100 petaflops per second-day or 1000x more compute than you needed for AlexNet.

Distributed systems are a necessity for machine learning

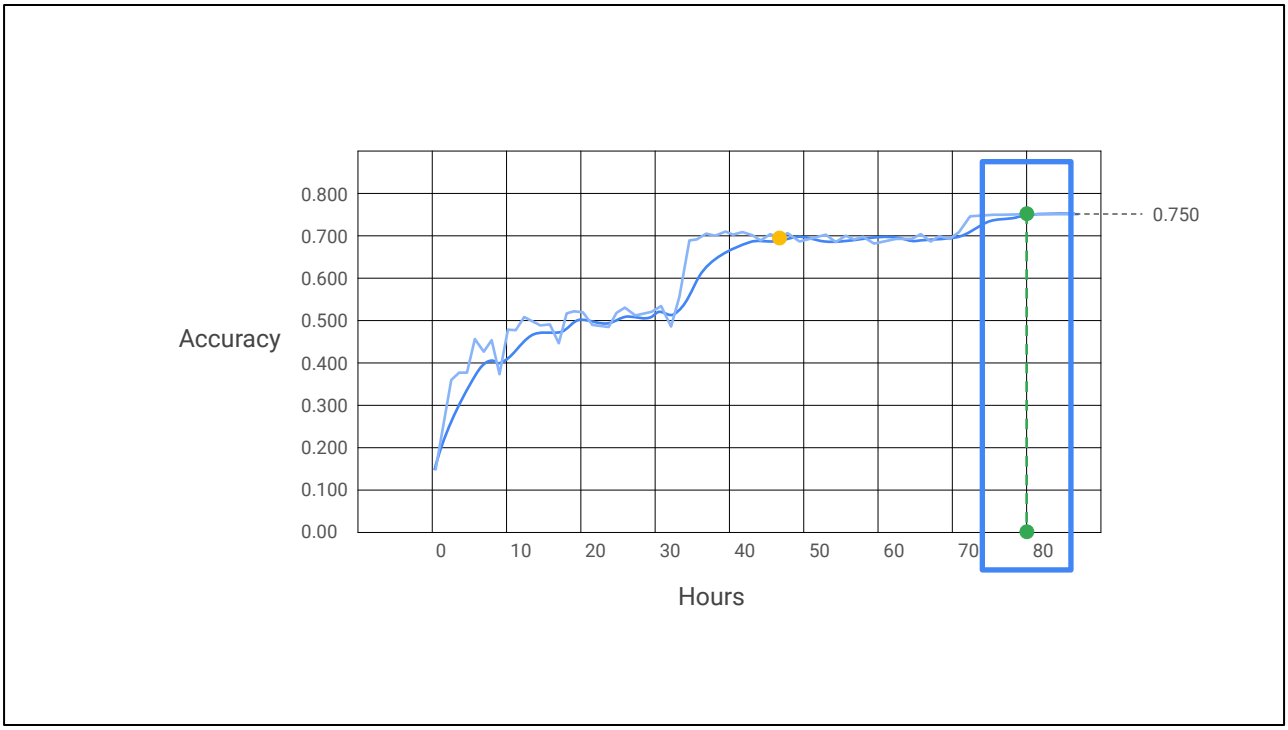


This growth in algorithm complexity and data size means that, with complex models and large data volumes, distributed systems are pretty much a necessity when it comes to machine learning.



Training complex networks with large amounts of data can often take a long time.

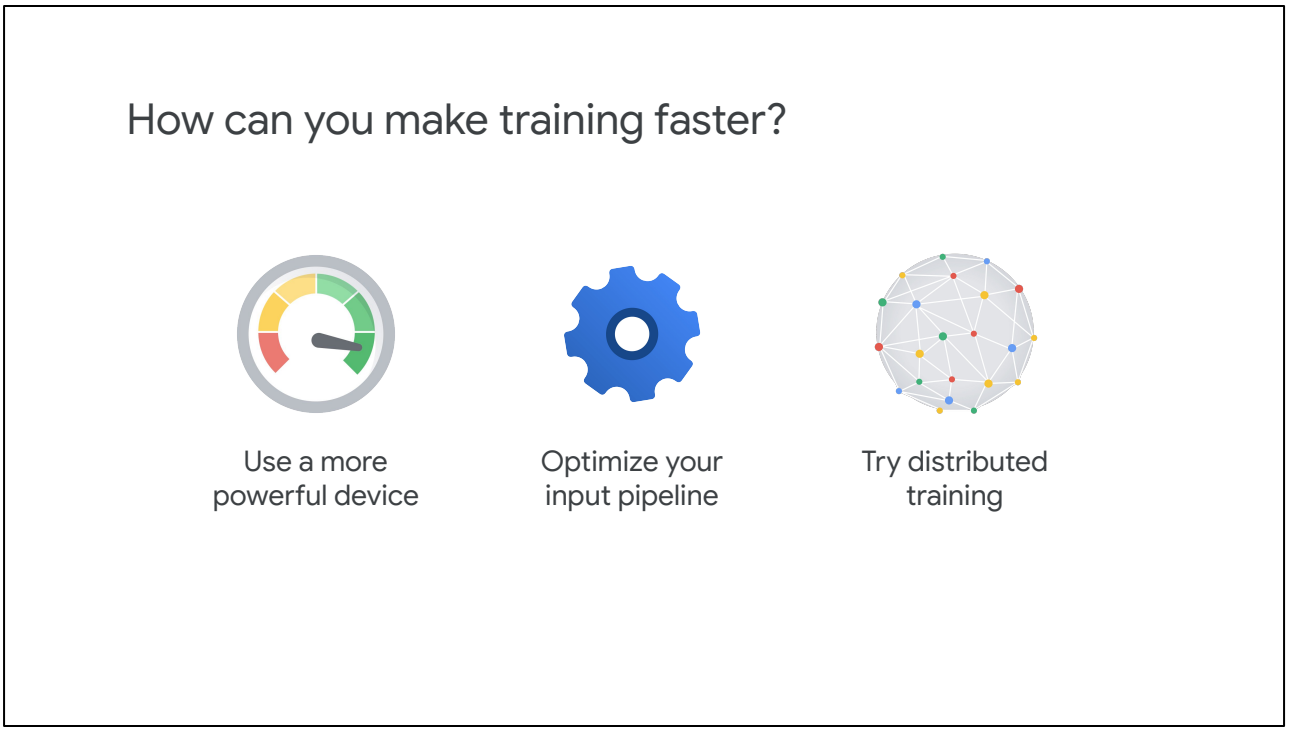
This graph shows training time on the x-axis plotted against the accuracy of predictions on the y-axis, when training an image recognition model on a GPU.



As the dotted line shows, it took around 80 hours to reach 75% accuracy.

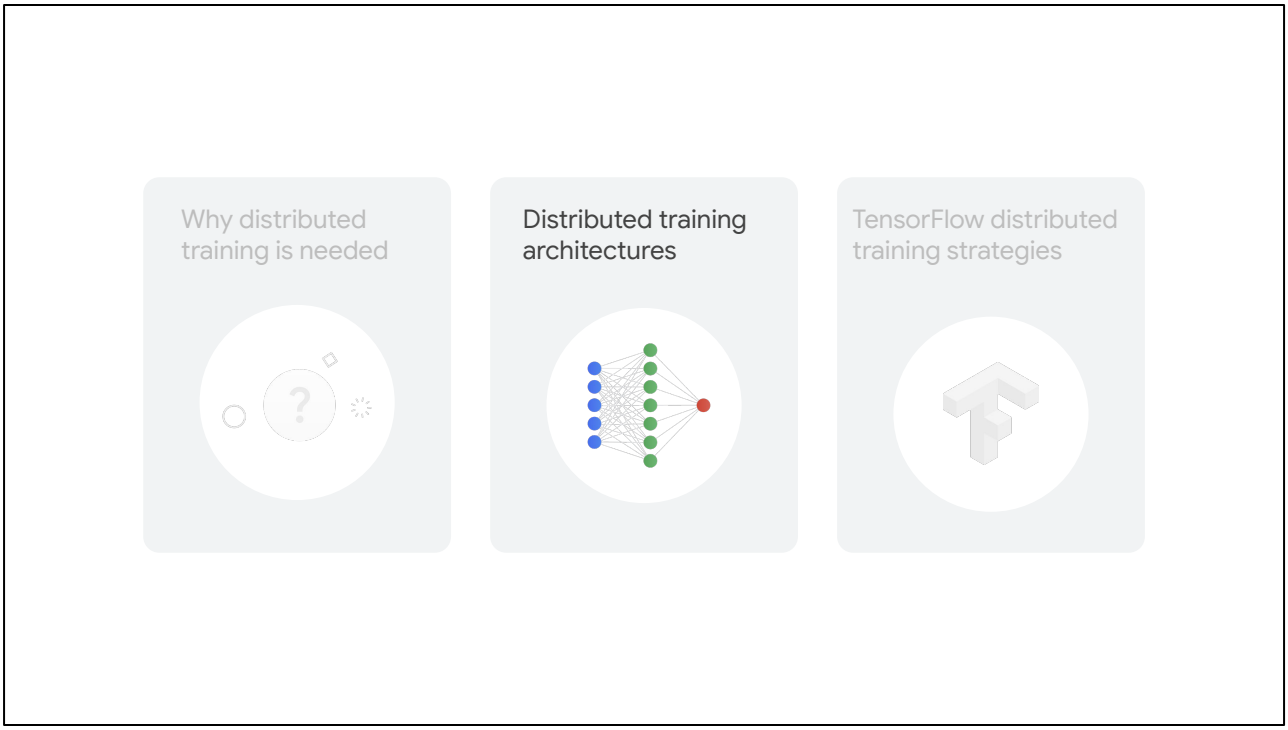
- If your training takes a few minutes to a few hours, it will make you productive and happy, and you can try out different ideas fast.
- If the training takes a few days, you could still deal with that by running a few ideas in parallel.
- If the training starts to take a week or more, your progress will slow down because you can’t try out new ideas quickly.
- And if it takes more than a month… Well that's probably not even worth thinking about!

And this is no exaggeration. Training deep neural networks such as ResNet50 can take up to a week on one GPU.



A natural question to ask is - how can you make training faster?

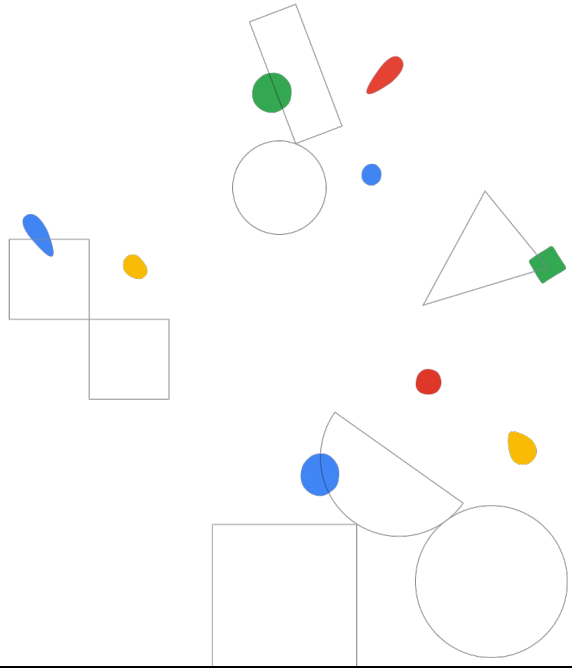
- You can use a more powerful device such as TPU or GPU (accelerator).
- You can optimize your input pipeline.
- Or, you can try out distributed training.



In the next video, we'll explore distributed training architectures.

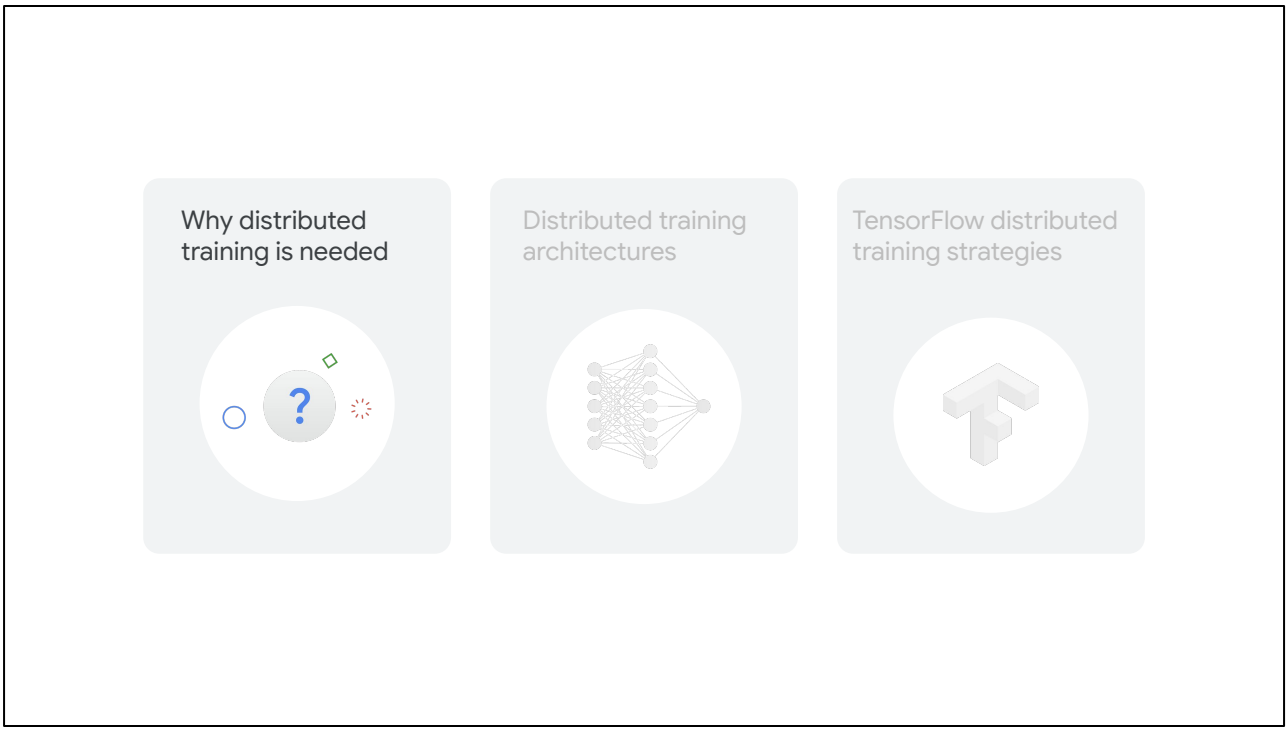
Google Cloud

Distributed training architectures

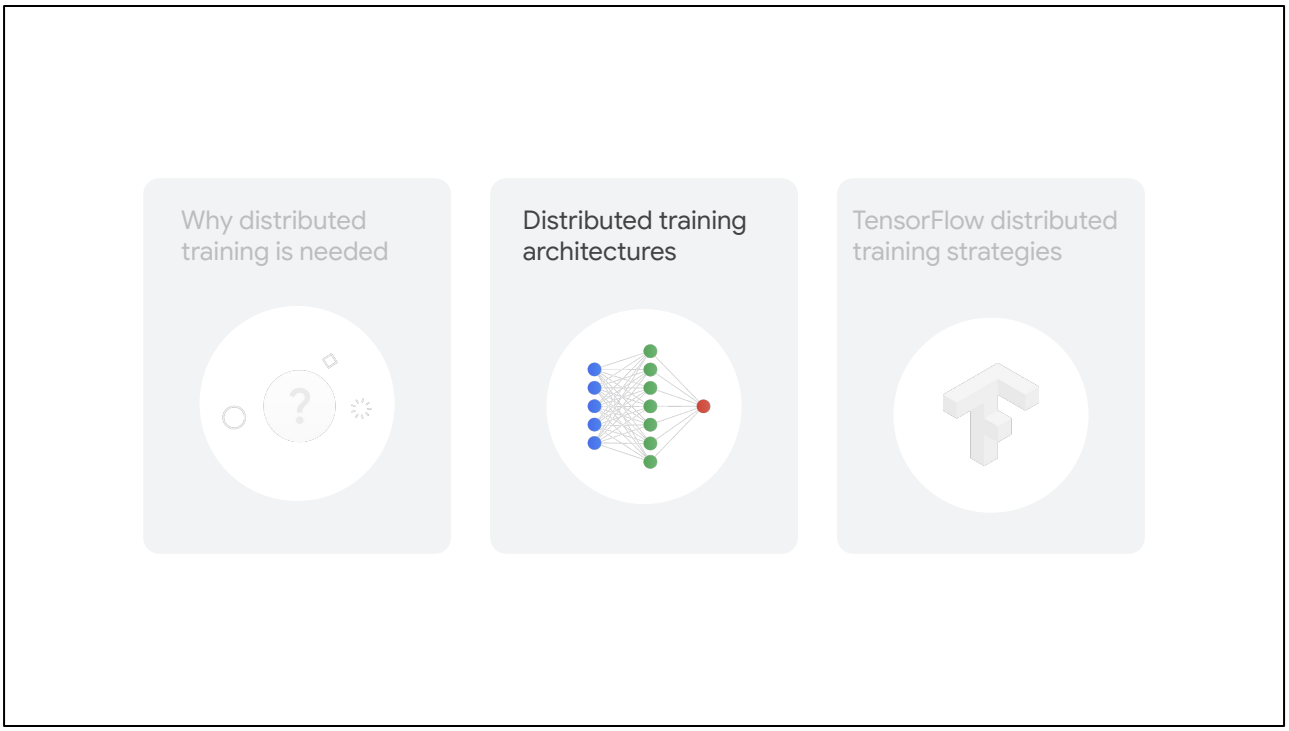


Module 03

Designing high-performance ML systems

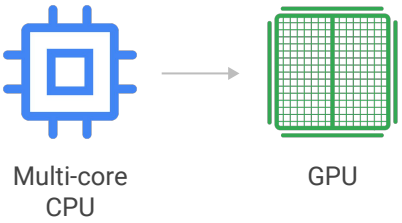


In the previous video we gave an overview of why distributed training is needed.



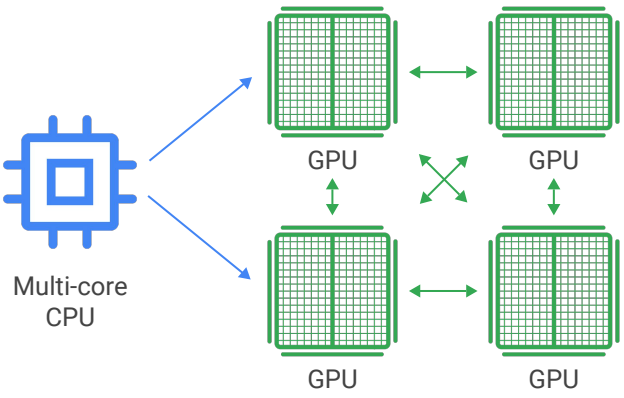
Let’s now take a look at distributed training architectures.

Before we get into the details of how to achieve this scaling in TensorFlow, let’s step back and explore the high level concepts and architectures in distributed training.

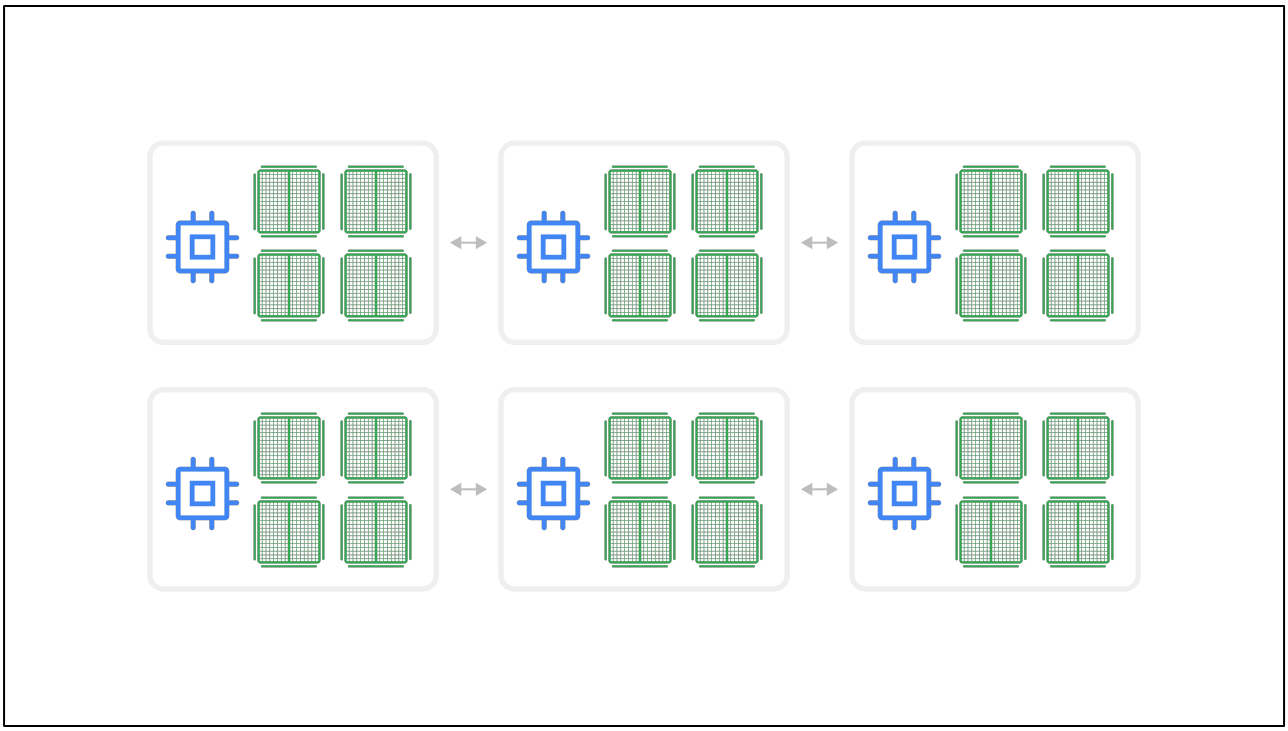


Let’s say you start training on a machine with a multi-core CPU. TensorFlow automatically handles scaling on multiple cores.

You may speed up your training by adding an accelerator to your machine such as a GPU. Again, TensorFlow will use this accelerator to speed up model training with no extra work on your part.



But with distributed training, you can go further. You can go from using one machine with a single device, to a machine with multiple devices attached to it,



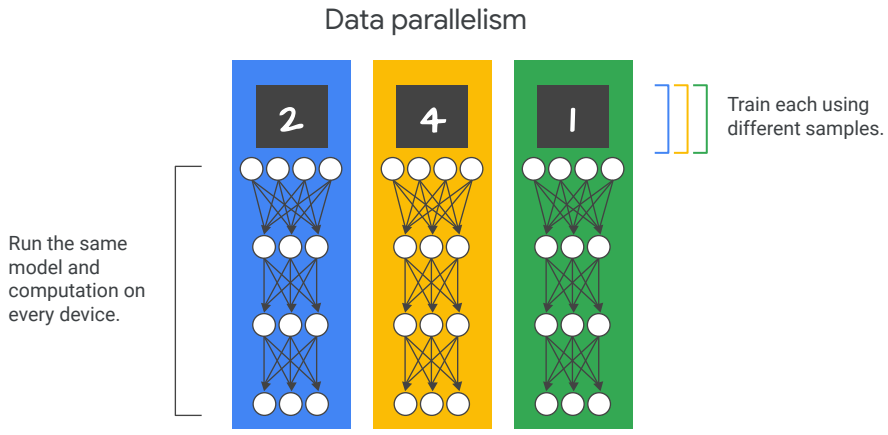
and finally to multiple machines, possibly with multiple devices each, connected over a network.

Eventually, with various approaches, you can scale up to hundreds of devices, and that is in fact what we do in several Google systems.

Simply stated, distributed training distributes training workloads across multiple mini-processors–or worker nodes.

These worker nodes work in parallel to accelerate the training process.

Types of distributed training architectures



Their parallelism can be achieved via two types of distributed training architecture. Let's explore both, starting with the most common, data parallelism.

Data parallelism is model agnostic, making it the most widely used paradigm for parallelizing neural network training.

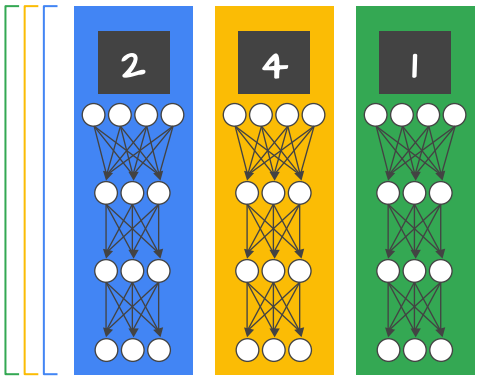
In data parallelism, you run the same model and computation on every device, but train each of them using different training samples.

Types of distributed training architectures

Data parallelism

Each device computes loss and gradients based on the training samples.

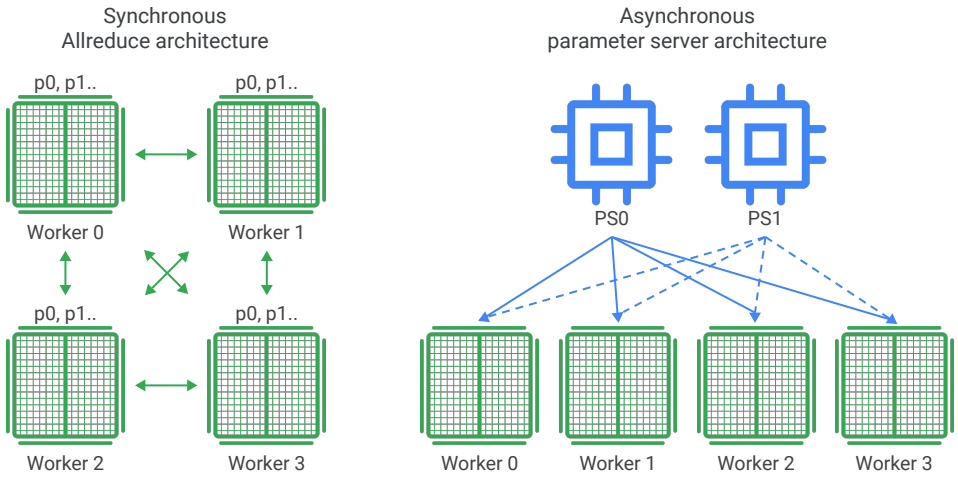
Gradient measures the change in all weights.



Each device computes loss and gradients based on the training samples it sees. Then we update the model's parameters using these gradients. The updated model is then used in the next round of computation.

You'll recall that a gradient simply measures the change in all weights with regard to the change in error. You can also think of a gradient as the slope of a function. The higher the gradient, the steeper the slope and the faster a model can learn. But if the slope is zero, the model stops learning.

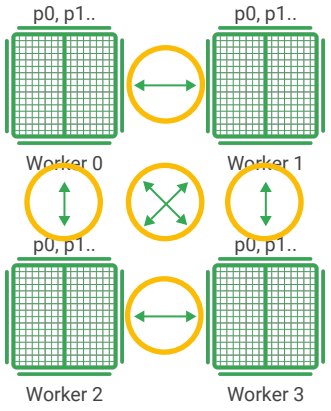
Data parallelism model approaches



There are currently two approaches used to update the model using gradients from various devices: **Synchronous** and **asynchronous**.

In synchronous training, all of the devices train their local model using different parts of data from a single, large mini-batch. They then communicate their locally calculated gradients, directly or indirectly, to all devices.

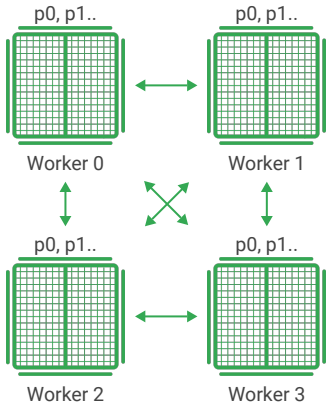
Synchronous Allreduce architecture



- Each worker device computes the forward and backward passes through the model on a different slice of the input data.

In this approach, each worker device computes the forward and backward passes through the model on a different slice of the input data.

Synchronous Allreduce architecture



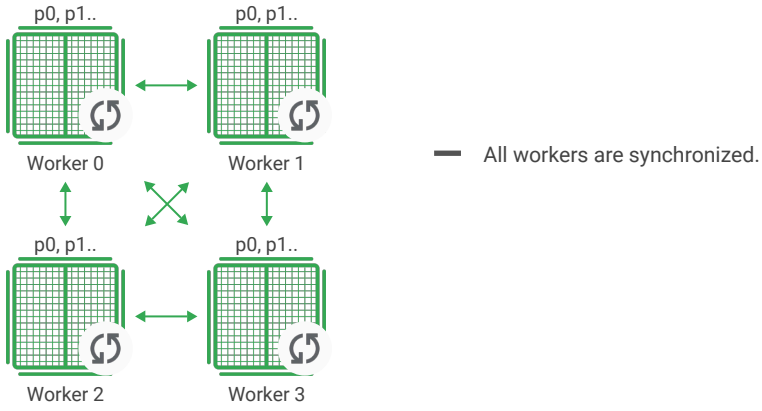
- Each worker device computes the forward and backward passes through the model on a different slice of the input data.
- Each training node exchanges the gradients via an AllReduce operation at the end of each training iteration.
- The optimizer performs the parameter updates with these reduced gradients, keeping the devices in sync.

The computed gradients from each of these slices are then aggregated across all of the devices and reduced, usually using an average, in a process known as AllReduce.

The optimizer then performs the parameter updates with these reduced gradients thereby keeping the devices in sync.

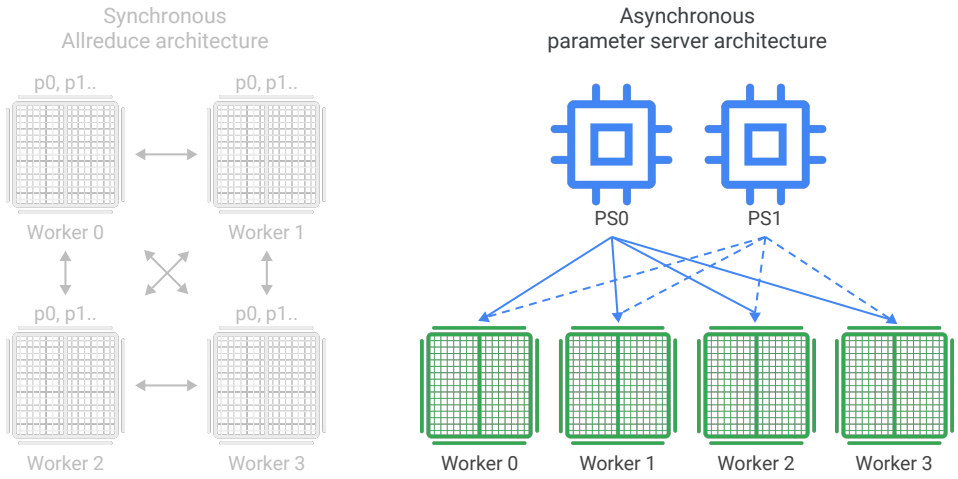
Because each worker cannot proceed to the next training step until all the other workers have finished the current step, this gradient calculation becomes the main overhead in distributed training for synchronous strategies.

Synchronous Allreduce architecture



Only after all devices have successfully computed and sent their gradients so that all workers are synchronized, is the model updated. The updated model is then sent to all nodes along with splits from the next mini-batch. That is, devices train on non-overlapping splits of the mini-batch.

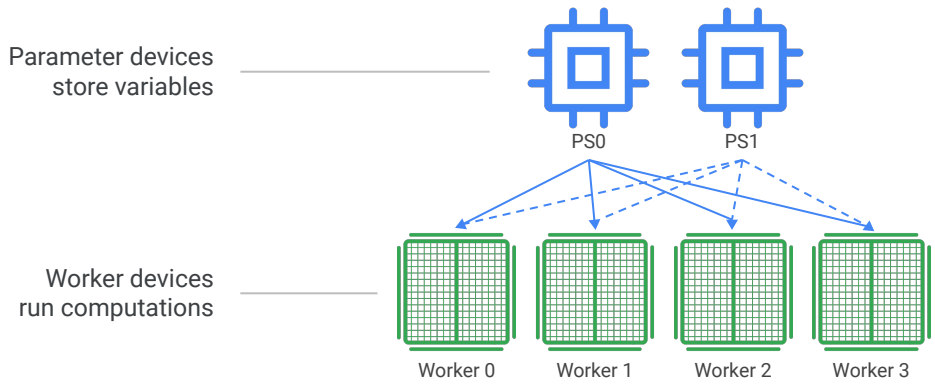
Asynchronous architecture



In asynchronous training, no device waits for updates to the model from any other device. The devices can run independently and share results as peers, or communicate through one or more central servers known as “parameter” servers.

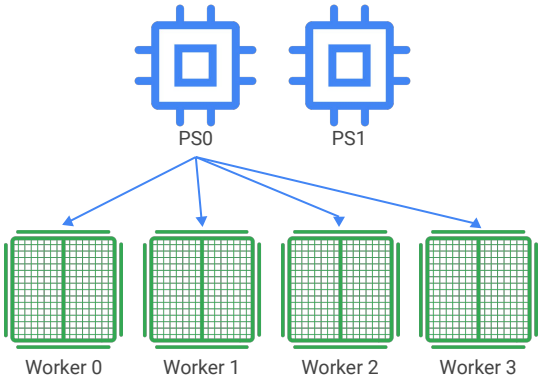
Thus, In an asynchronous parameter server architecture, some devices are designated to be parameter servers, and others as workers.

Asynchronous architecture



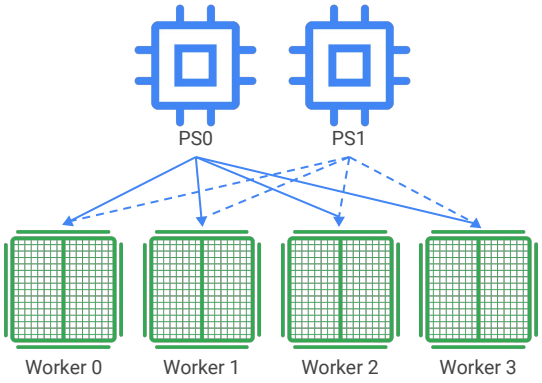
Devices used to store variables are parameter devices, whilst devices used to run computations are called worker devices.

Asynchronous architecture



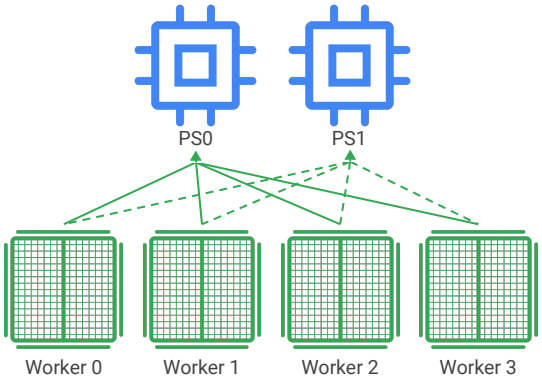
Each worker independently fetches

Asynchronous architecture



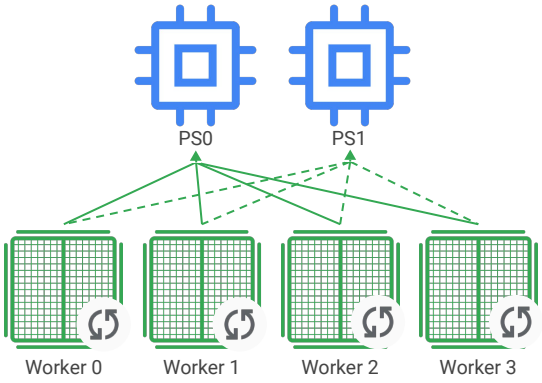
the latest parameters from the parameter servers and computes gradients based on a subset training samples.

Asynchronous architecture



It then sends the gradients back to the PS. Which then updates its copy of the parameters with those gradients.

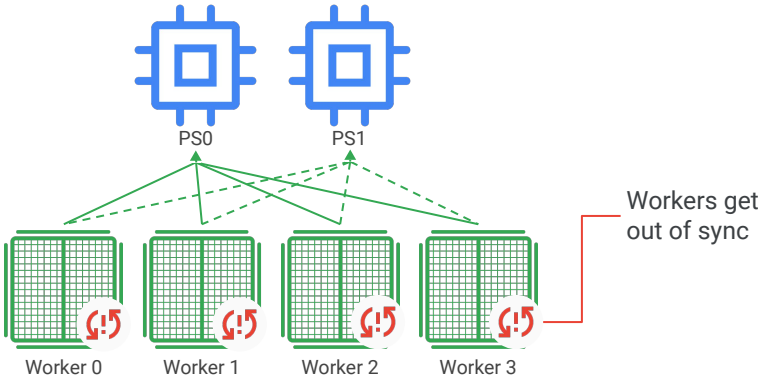
Asynchronous architecture



Each worker does this independently. This allows it to scale well to a large number of workers, where training workers might be preempted by higher priority production jobs, or a machine may go down for maintenance, or where there is asymmetry between the workers.

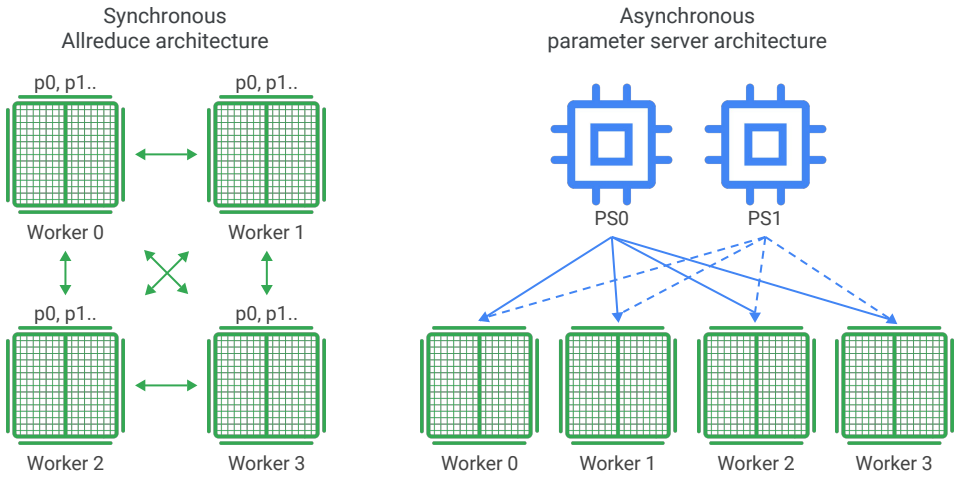
This does not hurt the scaling because workers are not waiting for each other.

Asynchronous architecture



The downside of this approach, however, is that workers can get out of sync. They compute parameter updates based on stale values and this can delay convergence.

Which should you choose?

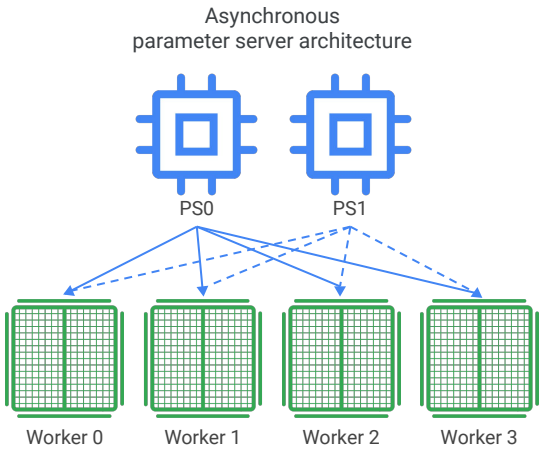


Given these two broad strategies, the asynchronous parameter server approach and the synchronous allreduce approach, which should you choose?

Well, there isn't one right answer, but here are some considerations.

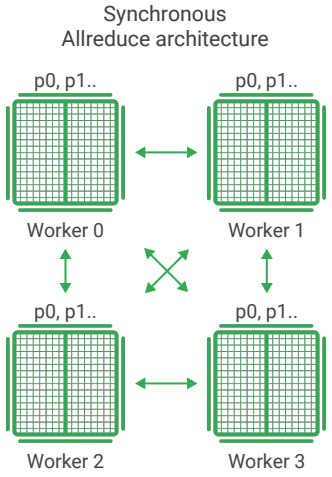
Which should you choose?

For models that use sparse data, contain fewer features, consume less memory, and can run just a cluster of CPUs.



The Asynchronous parameter server approach should be used for models that use sparse data (which contain fewer features, consume less memory, and can run just a cluster of CPUs).

Which should you choose?

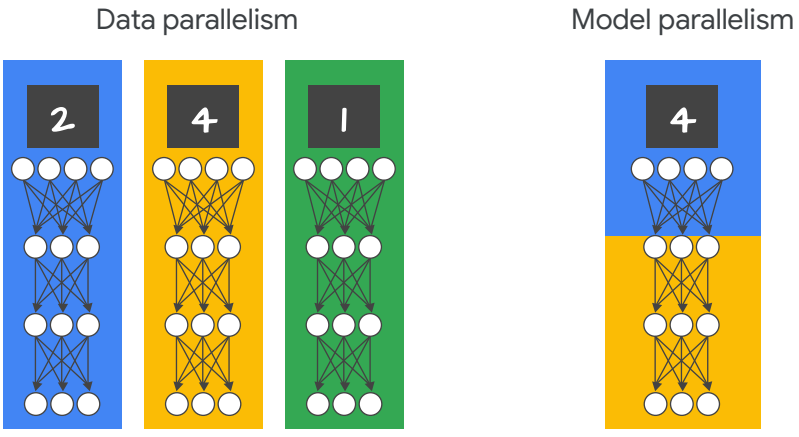


Best for dense models, like BERT, or Bidirectional Encoder Representations from Transformers.

The Sync AllReduce approach should be considered for dense models which contain many features and thus consume more memory.

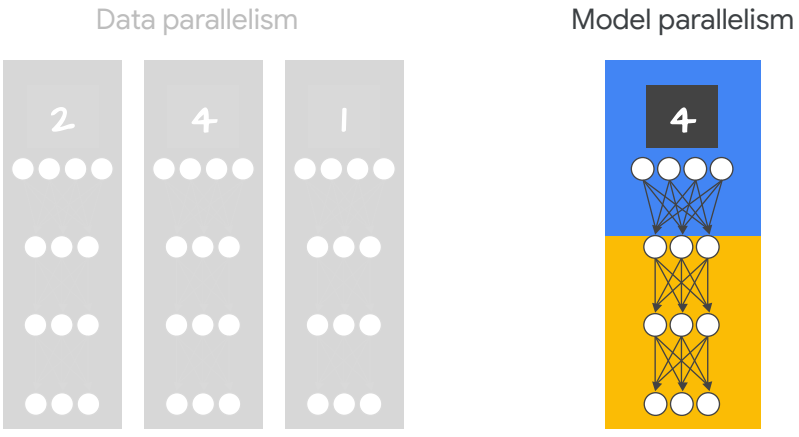
In this approach, all machines share the load of storing and maintaining the global parameters. This makes it the best option for dense models, like BERT (or Bidirectional Encoder Representations from Transformers).

Types of distributed training architectures



When a model is too big to fit on one device's memory, you divide it into smaller parts on multiple devices and then compute over the same training samples.

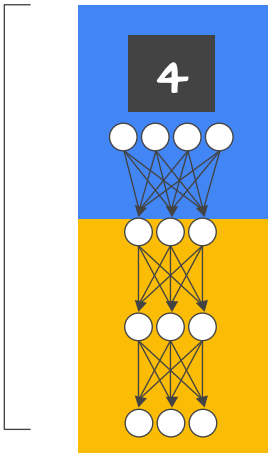
Types of distributed training architectures



This is called model parallelism.

Model parallelism

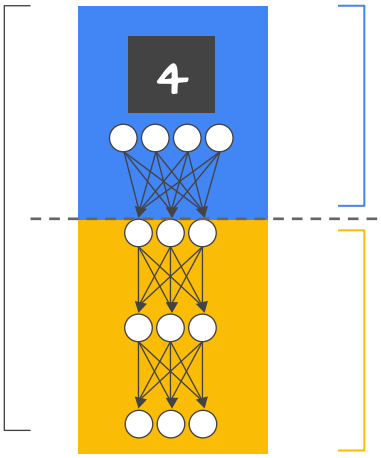
Each GPU has different parameters, and computation, of different parts of a model.



In this approach, each GPU has different parameters, and computation, of different parts of a model. In other words, multiple GPUs do not need to synchronize the values of the parameters.

Model parallelism

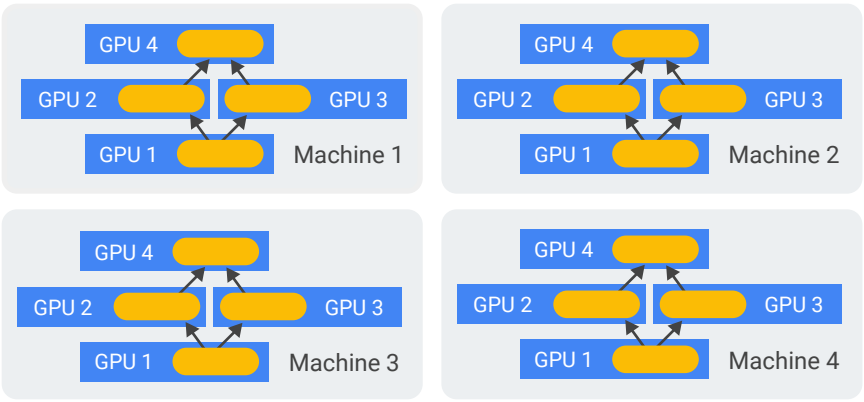
Needs special care when assigning different layers to different GPUs.



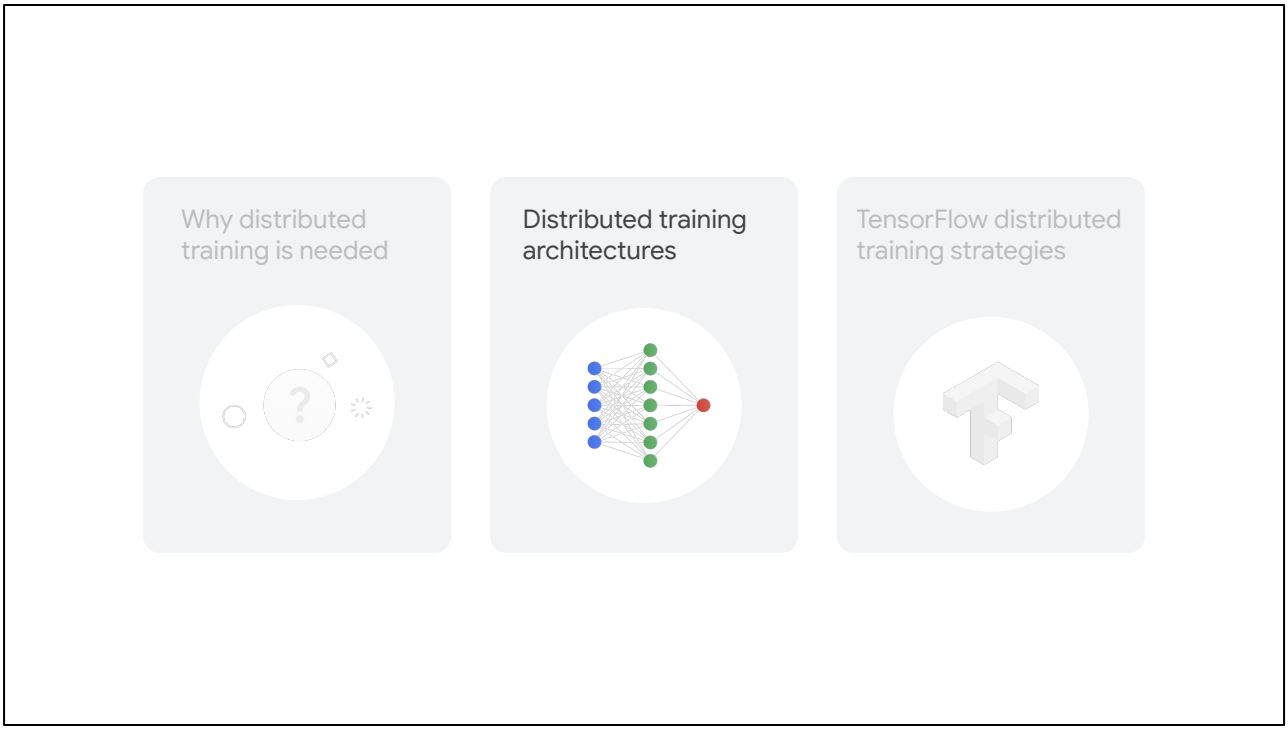
Model parallelism needs special care when assigning different layers to different GPUs, which is more complicated than data parallelism.

The gradients obtained from each model on each GPU are accumulated after a backward process, and the parameters are synchronized and updated.

Hybrid approach: Model and data parallelism



However, a hybrid of the data and model parallelism approaches is sometimes used together in the same architecture.

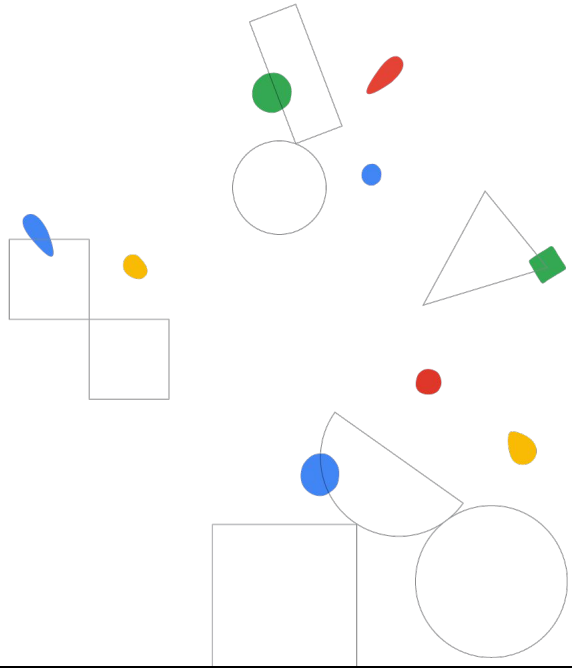


Now that you've been introduced to some of the different distributed training architectures,

in the next video we'll take a look at four TensorFlow distributed training strategies.

Google Cloud

TensorFlow distributed training strategies



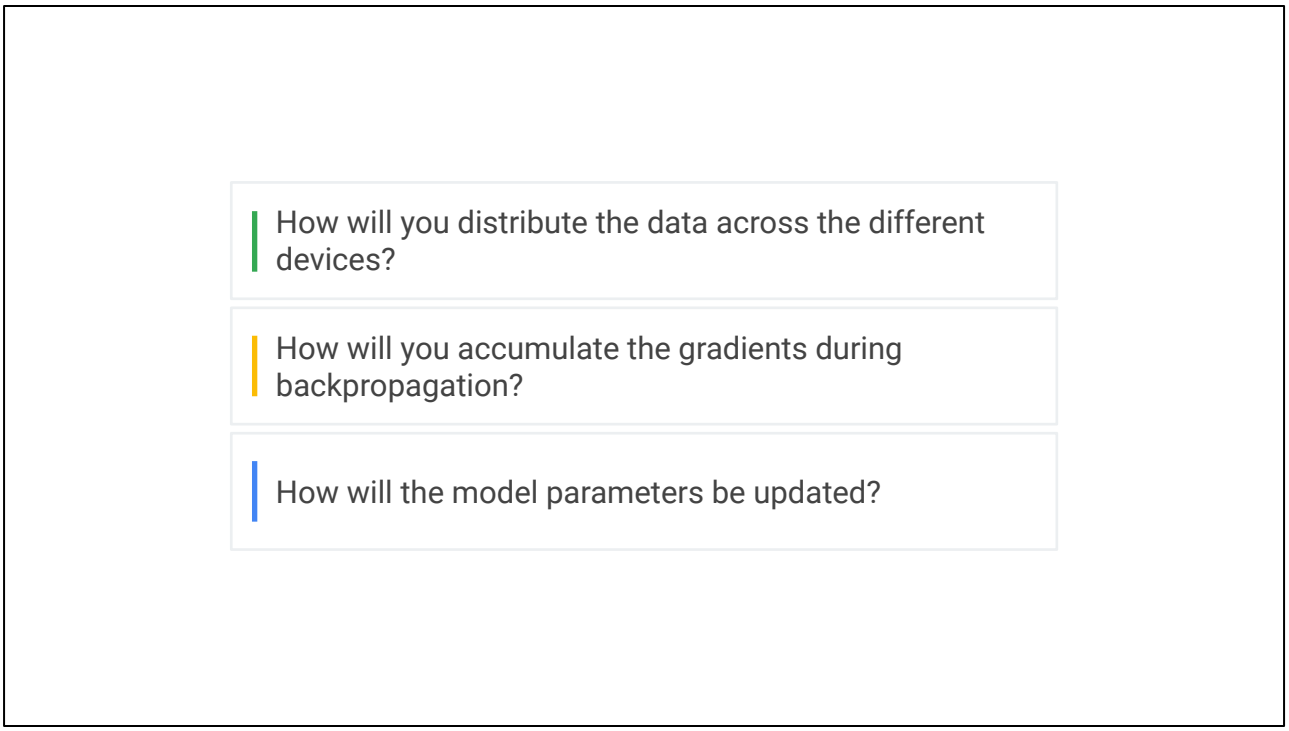
Module 03

Designing high-performance ML systems

Distributed training is particularly useful for very large datasets, because it becomes very difficult, and often unrealistic to perform model training on only a single hardware accelerator, such as a GPU.

TensorFlow's distributed strategies make it easier to seamlessly scale up heavy training workloads across multiple hardware accelerators — be it GPUs or even TPUs.

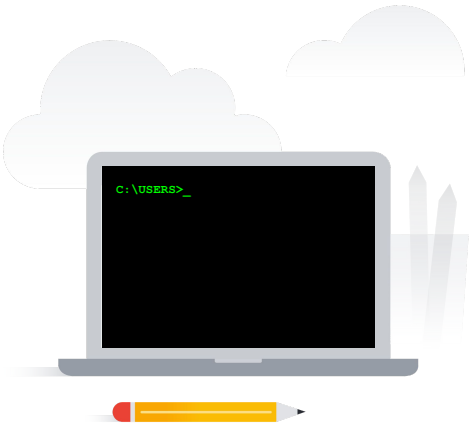
But in doing so, you may face challenges.



For example:

- How will you distribute the model parameters across the different devices?
- How will you accumulate the gradients during backpropagation?
- And how will the model parameters be updated?

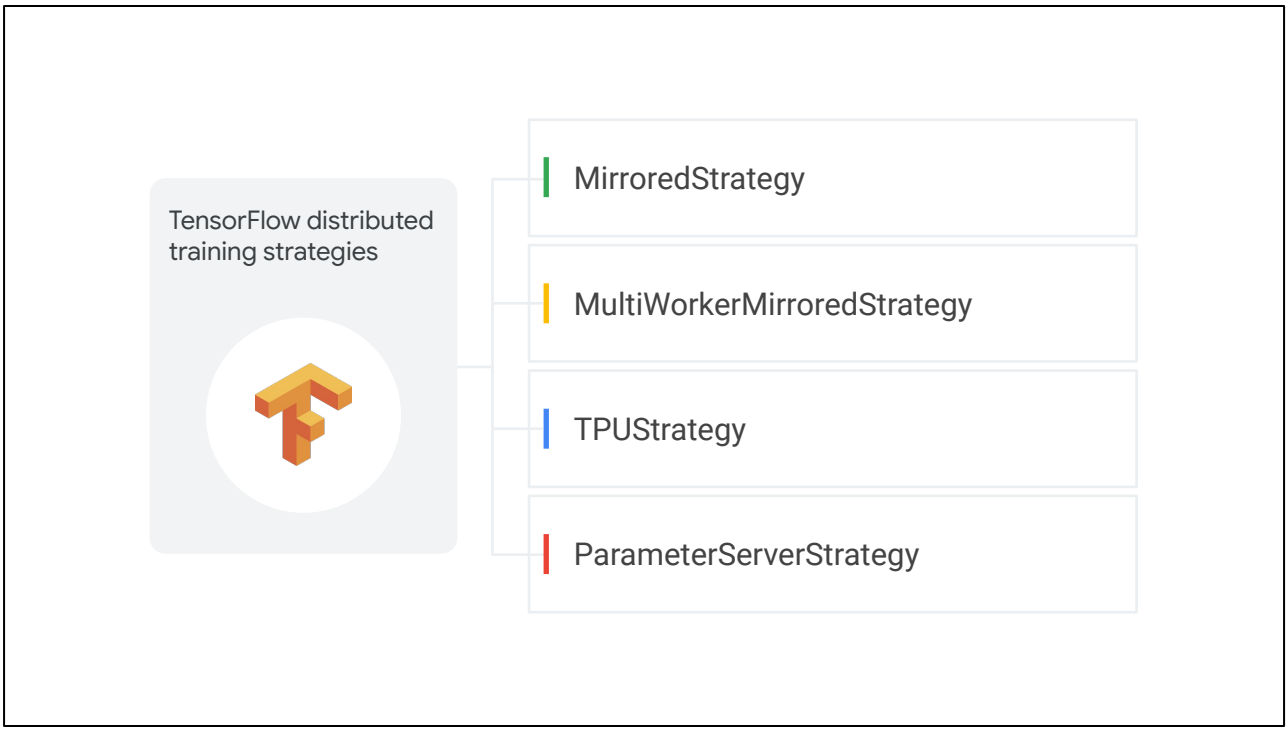
tf.distribute.Strategy API



tf.distribute.Strategy can help with these, and other, potential challenges.is a TensorFlow API to distribute training across multiple GPUs, multiple machines or TPUs.

And there are four TensorFlow distributed training strategies.

The list includes:



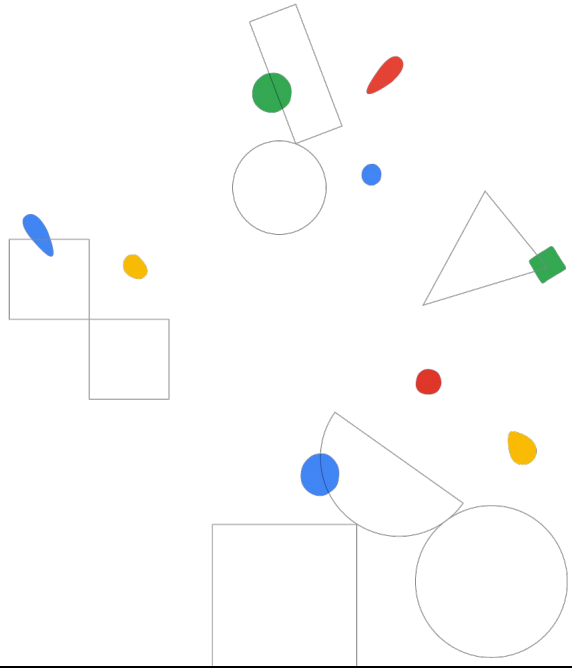
- The mirrored strategy.
- The multi-worker mirrored strategy.
- The TPU strategy, and finally.
- The parameter server strategy.

We'll cover each strategy in more depth in the videos that follow.

	Synchronous or Asynchronous	No. of nodes	No. of accelerators per node
MirroredStrategy	Synchronous	One	Many
TPUStrategy	Synchronous	One	Many
MultiWorkerMirroredStrategy	Synchronous	Many	Many
ParameterServerStrategy	Asynchronous	Many	Many

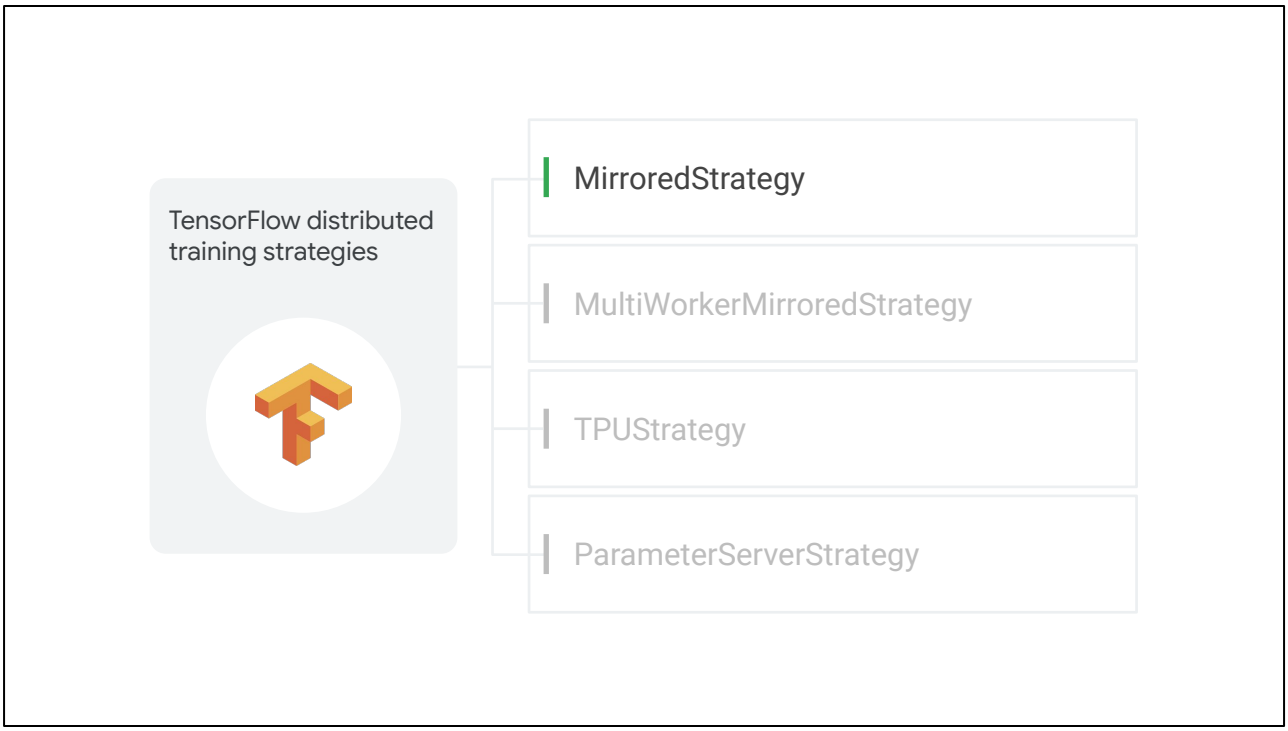
Google Cloud

MirroredStrategy



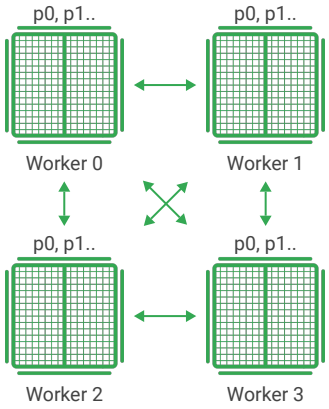
Module 03

Designing high-performance ML systems



The **mirrored strategy** is the simplest way to get started with distributed training.

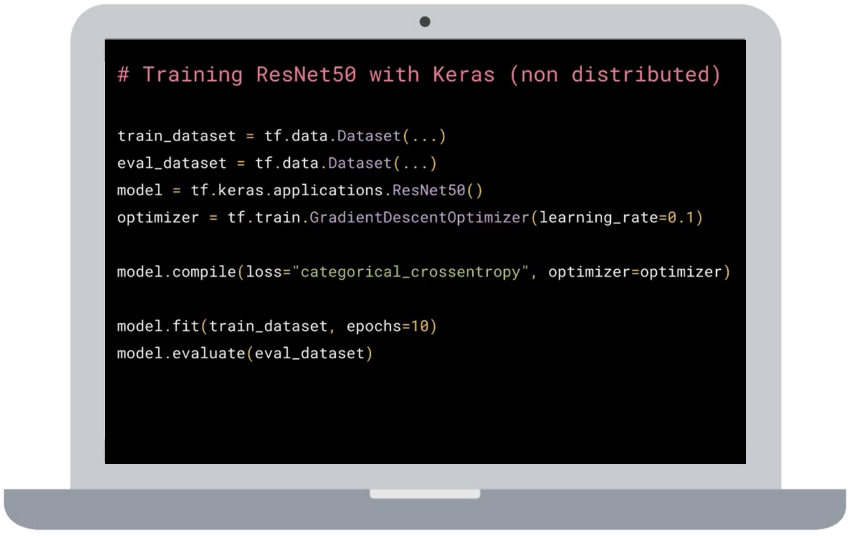
MirroredStrategy



It is a single machine with multiple GPU devices that creates one replica per GPU device.

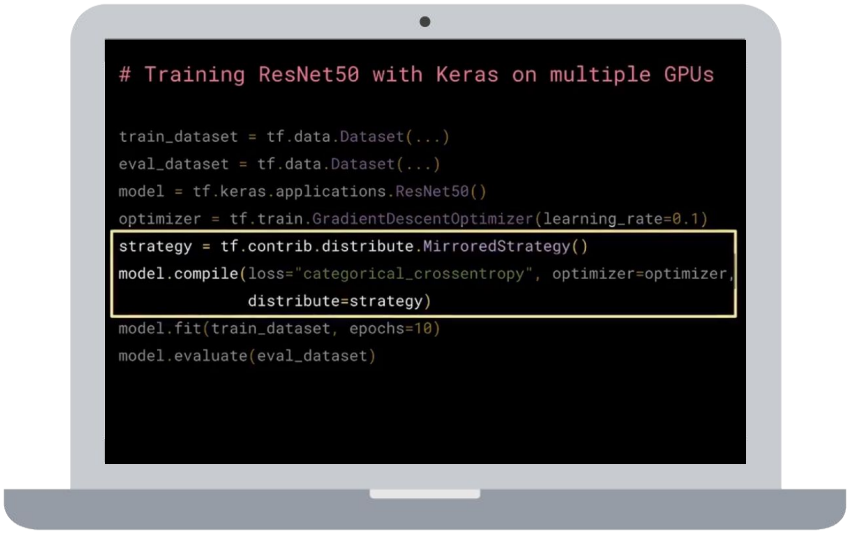
During training, one mini-batch is split into N parts, where N equals the number of GPUS, and each part feeds to one GPU device.

For this setup, the TensorFlow mirrored strategy manages the coordination of data distribution and gradient updates across all of the GPUs.



Here is an example of code showing a Non-distributed strategy.

Here we are training a Residual Network50 (or RESNET 50) with Keras. The code is standard -- we have our datasets, an optimizer, a model.compile, model.fit, and model.evaluation.



Now, to add a distribution strategy, we simply need to add the code here to show the mirrored strategy with multiple GPUs.

And here, you can see how easy it is to add code for a Distributed - MIrrored Strategy - single machines using multiple GPUs.

Keras ResNet model with the functional API

- [x] Download the Cassava dataset from TensorFlow

```
data, info = tfds.load(name='cassava', as_supervised=True, with_info=True)
NUM_CLASSES = info.features['label'].num_classes
```

- [x] Add a preprocess_data function to scale the images

```
def preprocess_data(image, label):
 image = tf.image.resize(image, (300,300))
 return tf.cast(image, tf.float32) / 255., label
```

Let's look at an image classification example where a Keras ResNet model with the functional API is defined.

First, download the Cassava dataset from TensorFlow Datasets.

Then, add a preprocess_data function to scale the images.

Keras ResNet model with the functional API

Define the model

```
def create_model():
 base_model = tf.keras.applications.ResNet50(weights='imagenet',
include_top=False)
 x = base_model.output
 x = tf.keras.layers.GlobalAveragePooling2D()(x)
 x = tf.keras.layers.Dense(1016, activation='relu')(x)
 predictions = tf.keras.layers.Dense(NUM_CLASSES, activation='softmax')(x)
 model = tf.keras.Model(inputs=base_model.input, outputs=predictions)
 return model
```

Then, define the model.

Keras ResNet model with the functional API

- [x] Create the strategy object

```
strategy = tf.distribute.MirroredStrategy()
```

Let’s create the strategy object using tf.distribute MIrroredStrategy.

Keras ResNet model with the functional API

- [x] Create your model variables within the strategy scope

```
with strategy.scope():
  model = create_model()
  model.compile(
      loss='sparse_categorical_crossentropy',
      optimizer=tf.keras.optimizers.Adam(0.0001),
      metrics=['accuracy'])
```

Next, let's create the model with variables within the strategy scope. These variables include the model, spare_categorical_crossentropy for loss, a Keras optimizer, and metrics variables to compute accuracy.

Keras ResNet model with the functional API

- [x] Change the batch size

```
batch_size = 64 * strategy.num_replicas_in_sync
```

The last change you will want to make is to the batch size.

When you carry out distributed training with the tf.distribute.Strategy API and tf.data, the batch size now refers to the global batch size.

Keras ResNet model with the functional API

- [x] Change the batch size

```
batch_size = 64 * strategy.num_replicas_in_sync
```

In other words, if you pass a batch size of 64, and you have two GPUs, then each machine will process 32 examples per step. In this case, 64 is known as the global batch size, and 32 as the per replica batch size.

To make the most out of your GPUs, you will want to scale the batch size by the number of replicas, which is two in this case because there is one replica on each GPU.

Keras ResNet model with the functional API

- [x] Map, shuffle, and prefetch the data

```
train_data =  data['train'].map(preprocess_data)
train_data  = train_data.shuffle(1000)
train_data  = train_data.batch(batch_size)
train_data  = train_data.prefetch(tf.data.experimental.AUTOTUNE)
```

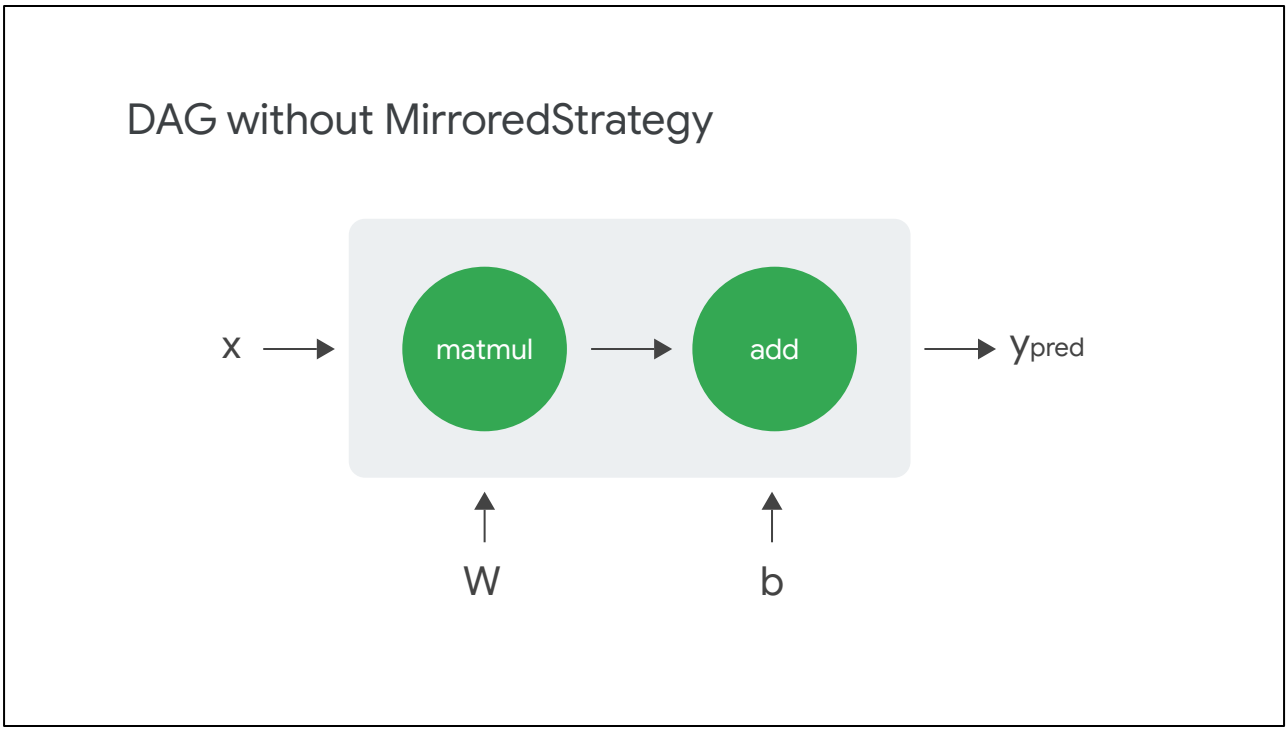
From there, map, shuffle, and prefetch the data.

Keras ResNet model with the functional API

- [x] Call model.fit

```
model.fit(train_data, epochs = 5)
```

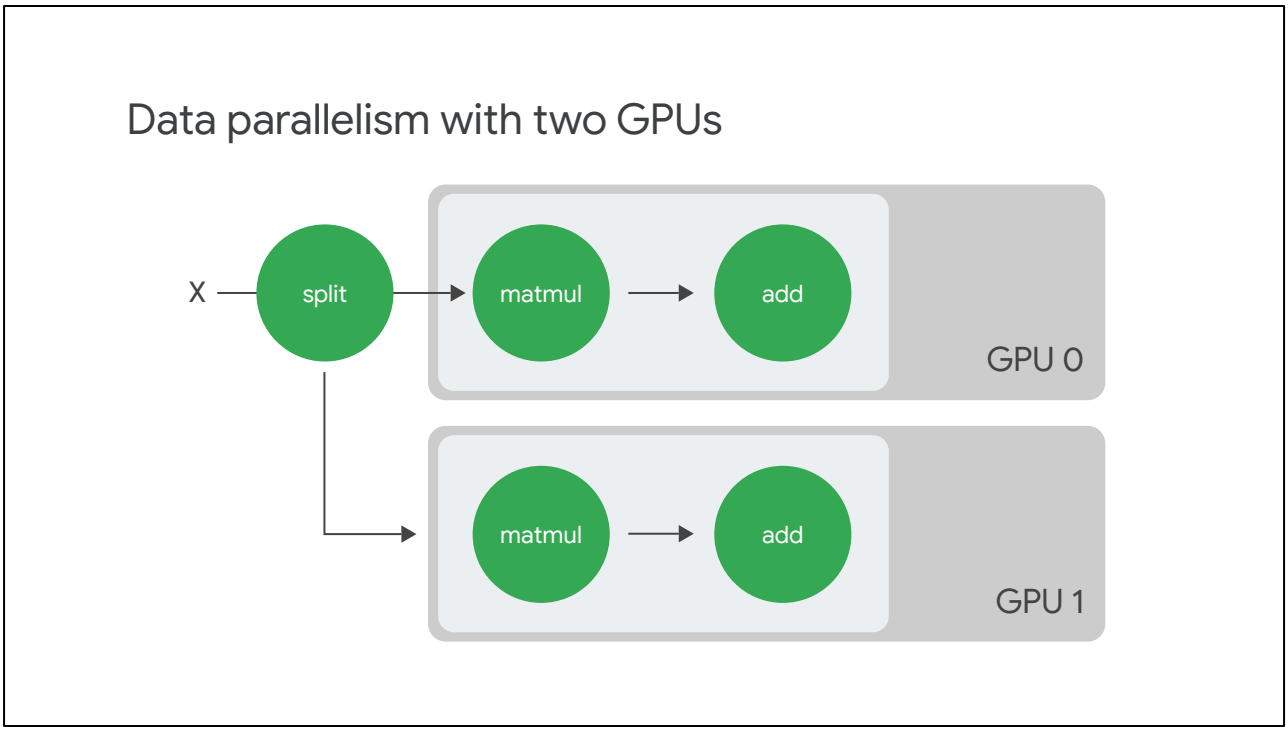
You then call model fit on the training data. Here we are going to run five passes of the entire training dataset.



Let's take a brief look at what actually happens when we call model.fit before adding a strategy.

For simplicity, imagine you have a simple linear model instead of the ResNet50 architecture. In TensorFlow, you can think of this simple model in terms of its computational graph (or Directed Acyclic Graph - or DAG).

Here, the matmul op takes in the X and W tensors, which are the training batch and weights respectively. The resulting tensor is then passed to the add op with the tensor b, which is the model's bias terms. The result of this op is ypred, which is the model's predictions.



Here is an example of data parallelism with two GPUs.

The input batch X is split in half, and one slice is sent to GPU 0, and the other to GPU 1.

In this case, each GPU calculates the same ops but on different slices of the data.

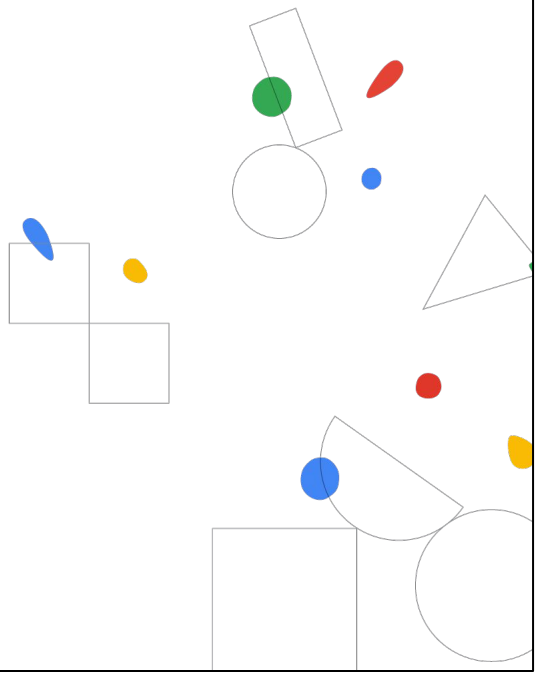
Optimize TensorFlow performance using the Profiler

tensorflow.org/guide/profiler

For more information on making the most of your GPUs, please refer to the guide titled, “Optimize TensorFlow GPU Performance with the TensorFlow Profiler,” found at tensorflow.org/guide/gpu_performance_analysis.

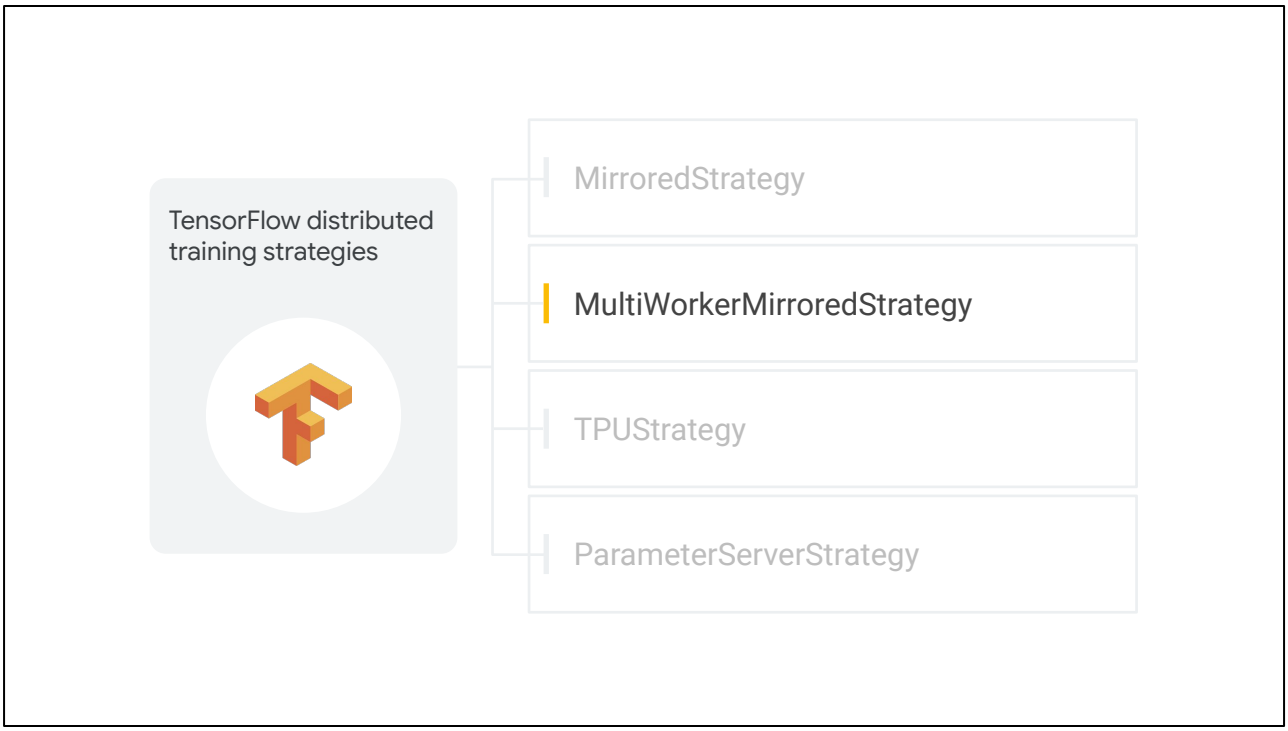
Google Cloud

MultiWorkerMirroredStrategy



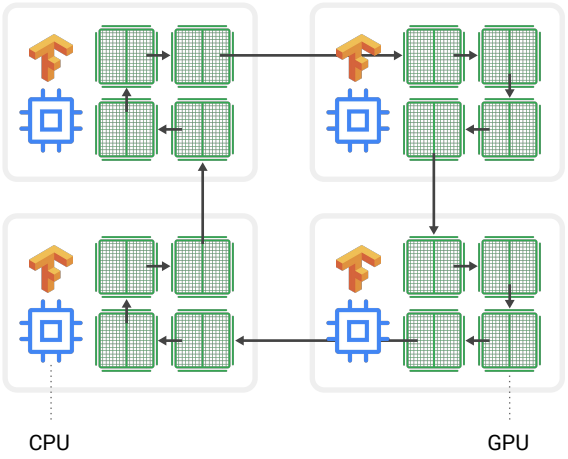
Module 03

Designing high-performance ML systems



The Multi Worker Mirrored Strategy is very similar to the MirroredStrategy. It implements synchronous distributed training across multiple workers, each with potentially multiple GPUs. Similar to MirroredStrategy, it creates copies of all variables in the model on each device across all workers.

Add machines to scale training



If you've mastered single host training and are looking to scale training even further, then adding multiple machines to your cluster can help you get an even greater performance boost.

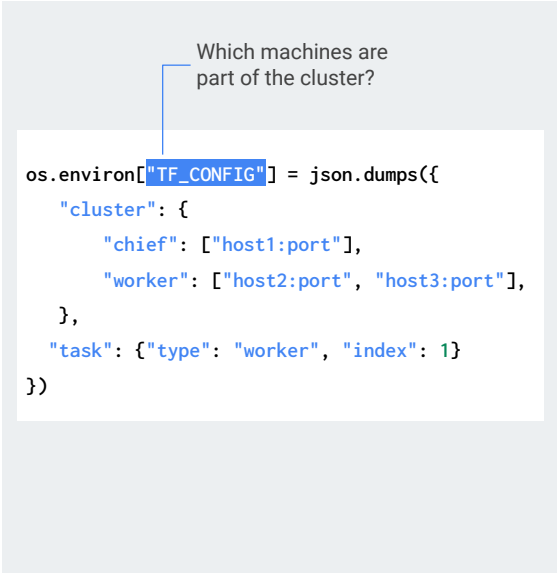
You can make use of a cluster of machines that are CPU only, or that each have one or more GPUs.

MultiWorker MirroredStrategy

```
os.environ["TF_CONFIG"] = json.dumps({
   "cluster": {
       "chief": ["host1:port"],
       "worker": ["host2:port", "host3:port"],
   },
  "task": {"type": "worker", "index": 1}
})
```

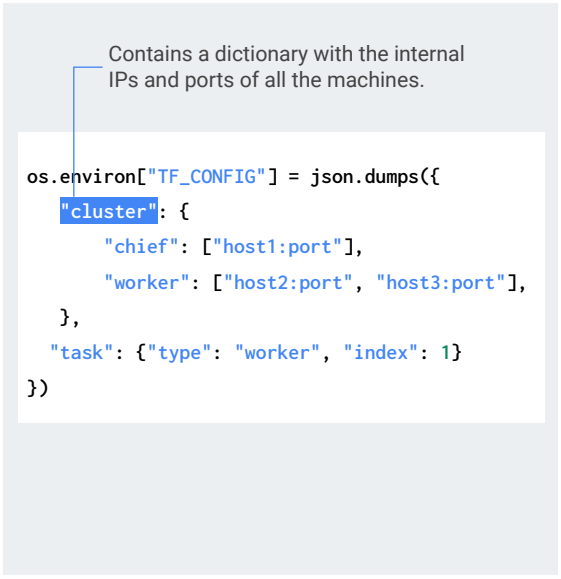
Like its single-worker counterpart, MirroredStrategy, MultiWorkerMirroredStrategy is a synchronous data parallelism strategy that can be used with only a few code changes.

MultiWorker MirroredStrategy



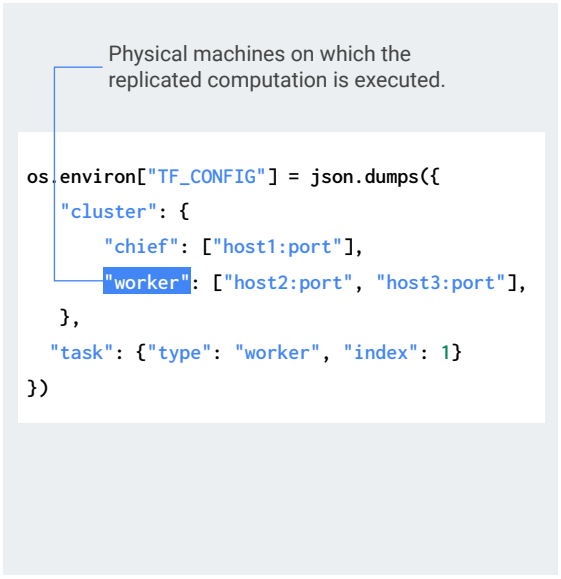
However, unlike MirroredStrategy, for a multi-worker setup TensorFlow needs to know which machines are part of the cluster. In most cases, this is specified with the environment variable TF_CONFIG.

MultiWorker MirroredStrategy



In this simple TF_CONFIG example, the “cluster” key contains a dictionary with the internal IPs and ports of all the machines.

MultiWorker MirroredStrategy



In MultiWorkerMirroredStrategy, all machines are designated as workers, which are the physical machines on which the replicated computation is executed.

MultiWorker MirroredStrategy

One worker that takes on some extra work.

```
os.environ["TF_CONFIG"] = json.dumps({
    "cluster": {
        "chief": ["host1:port"],
        "worker": ["host2:port", "host3:port"],
    },
   "task": {"type": "worker", "index": 1}
})
```

In addition to each machine being a worker, there needs to be one worker that takes on some extra work such as saving checkpoints and writing summary files to TensorBoard. This machine is known as the chief (or by its deprecated name master).

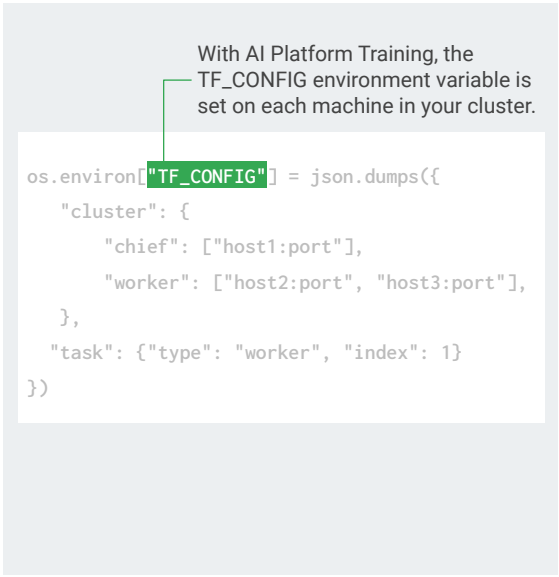
MultiWorker MirroredStrategy

With AI Platform Training, the TF_CONFIG environment variable is set on each machine in your cluster.

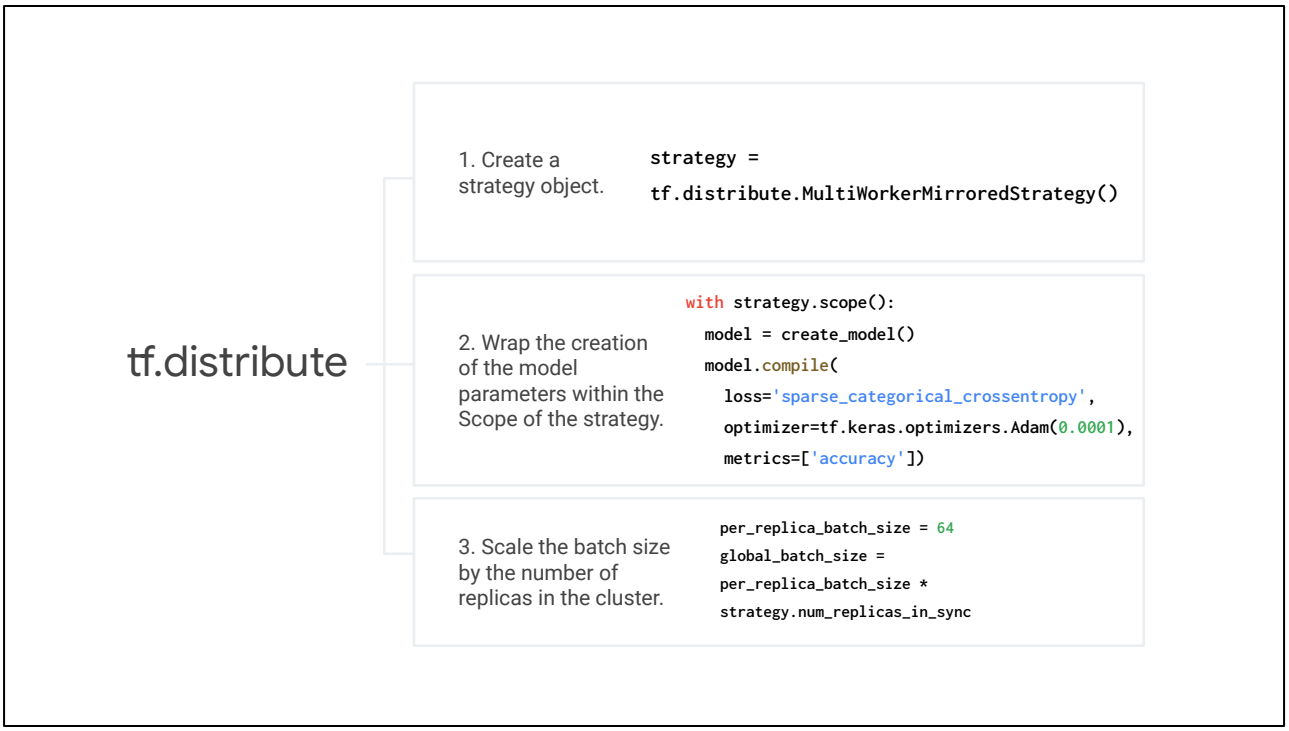
```
os.environ["TF_CONFIG"] = json.dumps({
   "cluster": {
       "chief": ["host1:port"],
       "worker": ["host2:port", "host3:port"],
   },
  "task": {"type": "worker", "index": 1}
})
```

Conveniently, when using AI Platform Training,

MultiWorker
MirroredStrategy



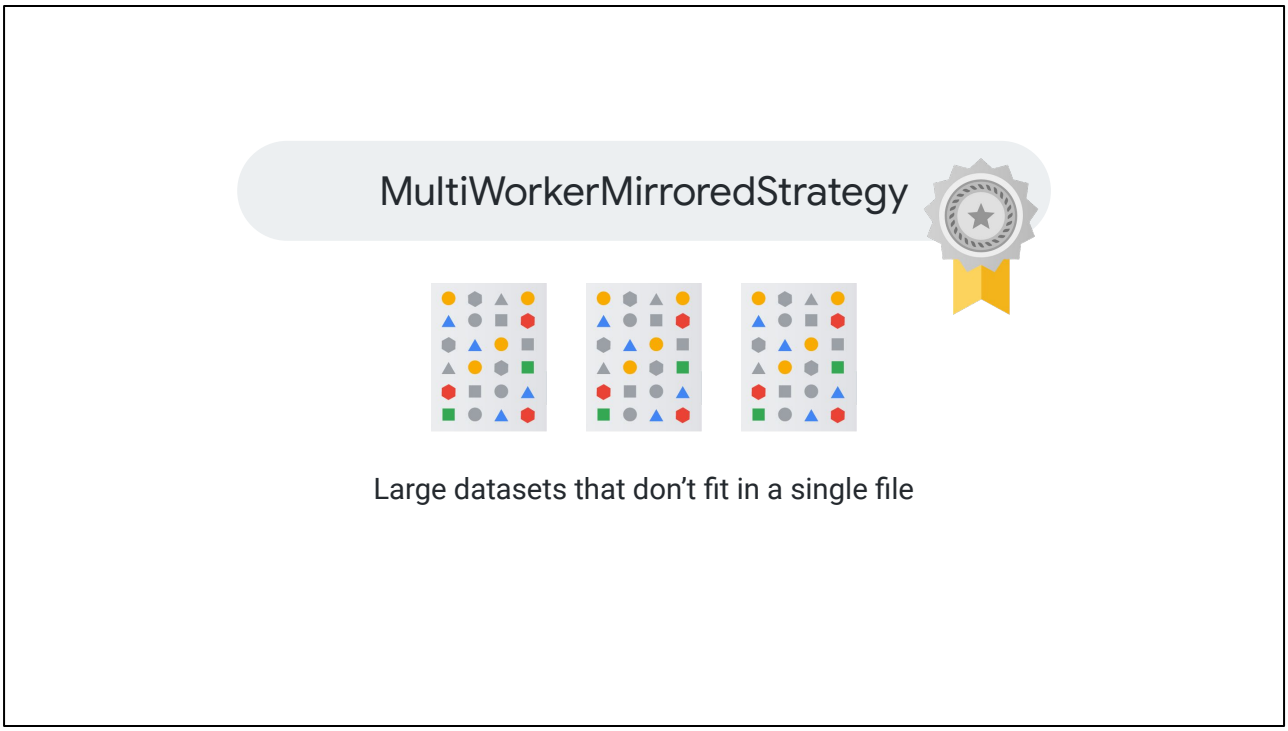
the TF_CONFIG environment variable is set on each machine in your cluster so there's no need to worry about this set up!



- As with any strategy in the tf.distribute module, step one is to create a strategy object.
- Step two is to wrap the creation of the model parameters within the scope of the strategy. This is crucial because it tells MirroredStrategy which variables to mirror across the GPU devices.
- And the third and final step is to scale the batch size by the number of replicas in the cluster. This ensures that each replica processes the same number of examples on each step.

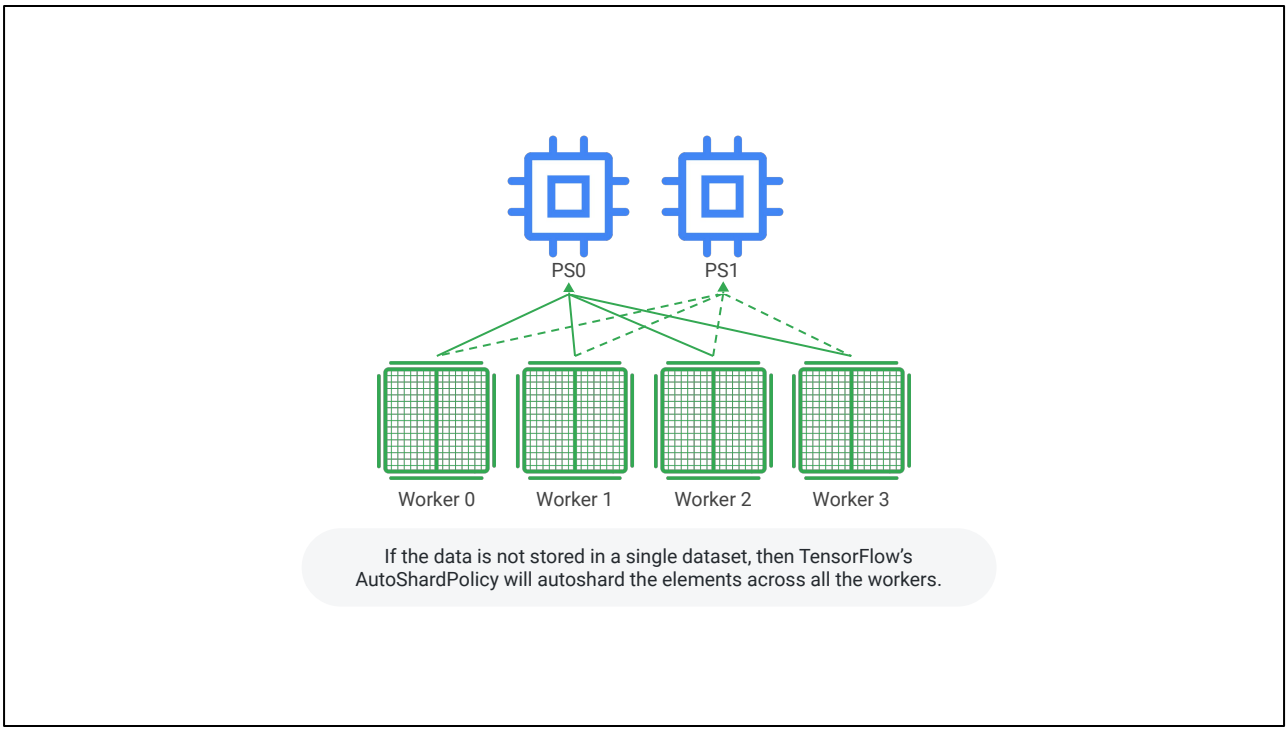
Since we've already covered training with MirroredStrategy, the previous steps should be familiar. The main difference when moving from synchronous data parallelism on one machine to many is that the gradients at the end of each step now need to be synchronized across all GPUs in a machine and across all machines in the cluster.

This additional step of synchronizing across the machines increases the overhead of distribution.

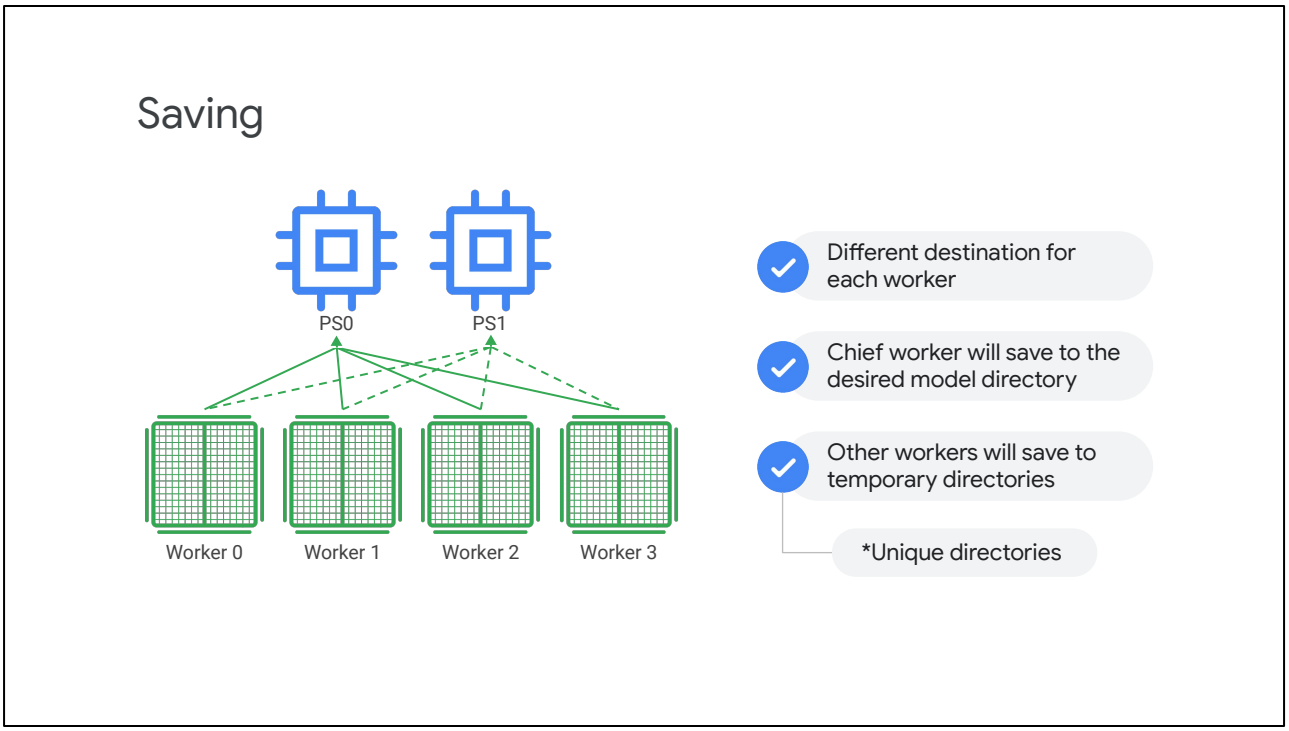


When you are working with large datasets that don’t fit in a single file, using a MultiWorkerMirroredStrategy is the best approach.

However, this option can lead to idle workers if the number of those multiple files is not divisible by the number of workers evenly, or if the length of some files are substantially longer than others.



If the data is not stored in a single dataset, then TensorFlow will autoshard the elements across all the workers. This guards against the potential idle worker scenario, but the downside is that the entire dataset will be read on each worker.



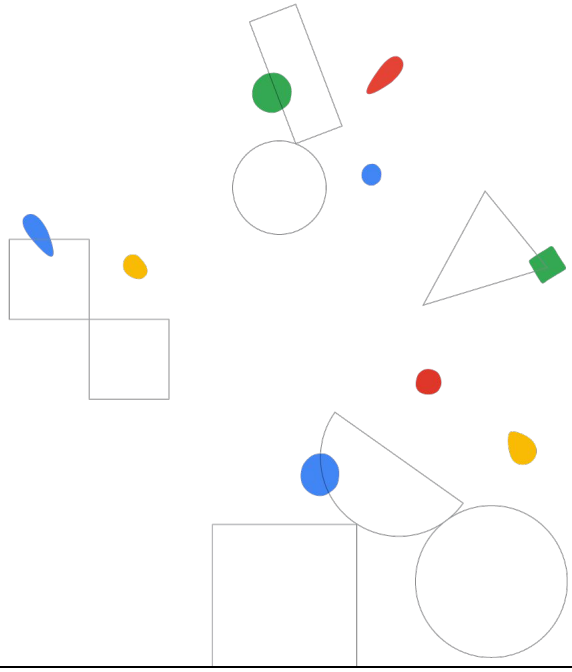
Saving the model is slightly more complicated in the multi-worker case because there needs to be different destinations for each worker.

The chief worker will save to the desired model directory, while the other workers will save the model to temporary directories.

It's important that these temporary directories are unique in order to prevent multiple workers from writing to the same location. Saving can contain collective ops, so all workers must save and not just the chief.

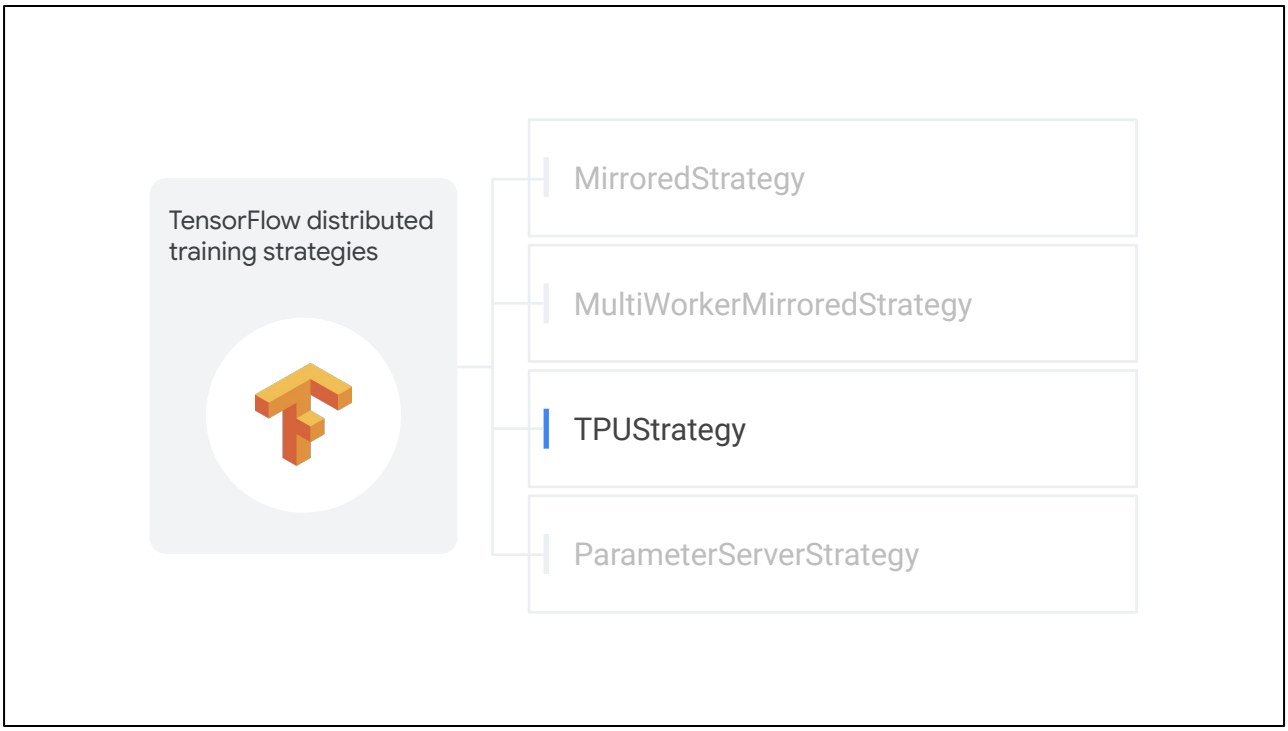
Google Cloud

TPUStrategy



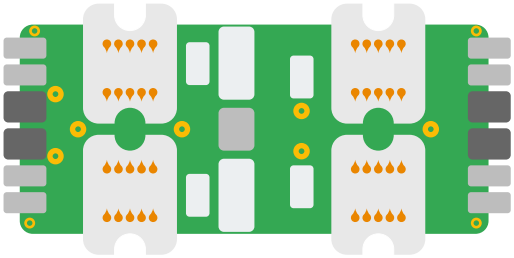
Module 03
Designing high-performance ML systems

Similar to mirrored strategy, TPU strategy uses a single machine



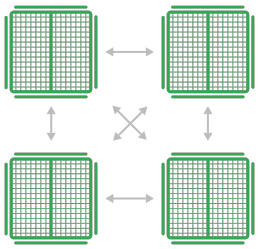
where the same model is replicated on each core, with its variable synchronized mirrored across each replica of the model.

TPUStrategy



All-reduce across TPU cores

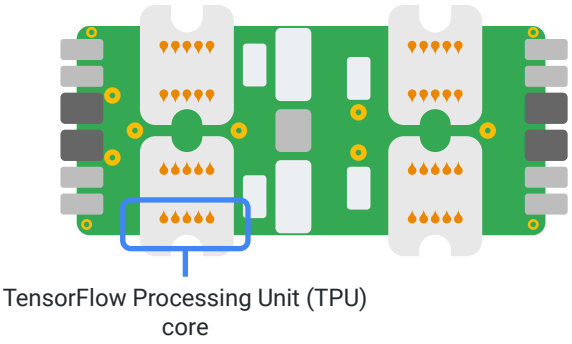
MirroredStrategy



All-reduce across devices

The main difference, however, is that the TPU strategy will all-reduce across TPU cores, whereas the Mirrored Strategy will all-reduce across devices.

tf.distribute.TPUStrategy



tf.distribute.TPUStrategy lets you run your TensorFlow training on Tensor Processing Units (TPUs). TPUs are Google's specialized ASICs designed to dramatically accelerate machine learning workloads.

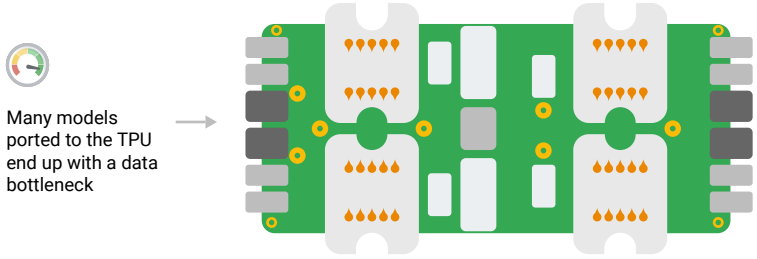
TPUs provide their own implementation of efficient all-reduce and other collective operations across multiple TPU cores, which are used in TPUStrategy.

Call the tf.distribute.TPUStrategy() method

```
strategy = tf.distribute.TPUStrategy()


with strategy.scope():
  model = tf.keras.Sequential(...)
  model.compile(
    loss='sparse_categorical_crossentropy',
    optimizer=tf.keras.optimizers.Adam(0.0001),
    metrics=['accuracy'])
```

You'll also need a variable called strategy but this time you will choose the tf.distribute.TPUStrategy method.

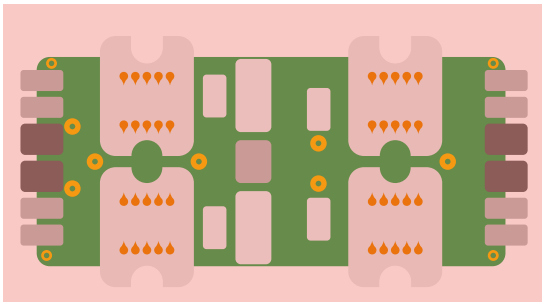
Data considerations



Because TPUs are very fast, many models ported to the TPU end up with a data bottleneck.

Data considerations

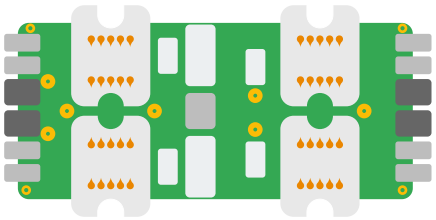
Many models ported to the TPU end up with a data bottleneck



The TPU is sitting idle, waiting for data for the most part of each training epoch.

Data considerations

- TPUs read training data from GCS
- GCS can sustain a large throughput
- Too few files: GCS will not have enough streams to get max throughput
- Too many files: Time will be wasted



TPUs read training data exclusively from Google Cloud Storage (GCS). And GCS can sustain a pretty large throughput if it is continuously streaming from multiple files in parallel.

Following best practices will optimize the throughput. With too few files, GCS will not have enough streams to get max throughput. With too many files, time will be wasted accessing each individual file.

```
model = tf.keras.Sequential(...)
model.compile(loss='mse', optimizer='sgd')
model.fit(dataset, epochs=2)
model.evaluate(dataset)
```

Let's summarize the distribution strategies using code.

Our base scope is a Keras sequential model.

```
strategy = tf.distribute.MirroredStrategy()

with strategy.scope():
  model = tf.keras.Sequential(...)
  model.compile(loss='mse', optimizer='sgd')
```

Now, to improve training, we can use the mirrored strategy.

```
strategy = tf.distribute.MultiWorkerMirroredStrategy()

with strategy.scope():
  model = tf.keras.Sequential(...)
  model.compile(loss='mse', optimizer='sgd')
  model.fit(dataset, epochs=2)
  model.evaluate(dataset)
```

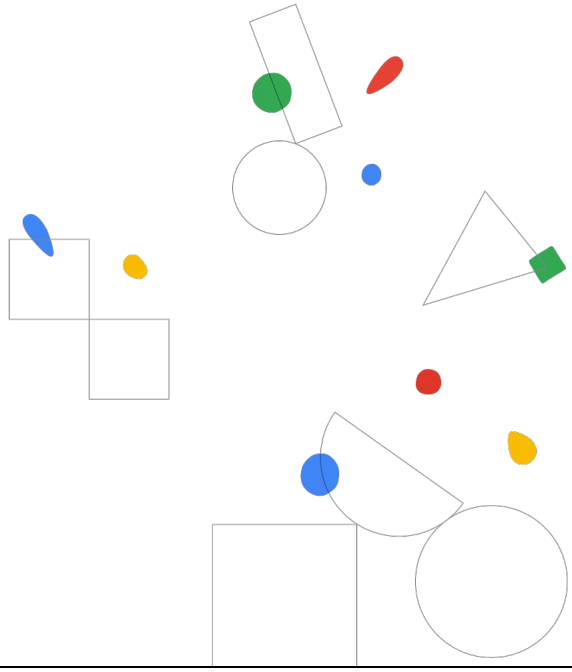
Or for faster training, the multi-worker mirrored strategy.

```
strategy = tf.distribute.TPUStrategy()

with strategy.scope():
  model = tf.keras.Sequential(...)
  model.compile(loss='mse', optimizer='sgd')
  model.fit(dataset, epochs=2)
  model.evaluate(dataset)
```

And for really fast training, the TPU strategy.

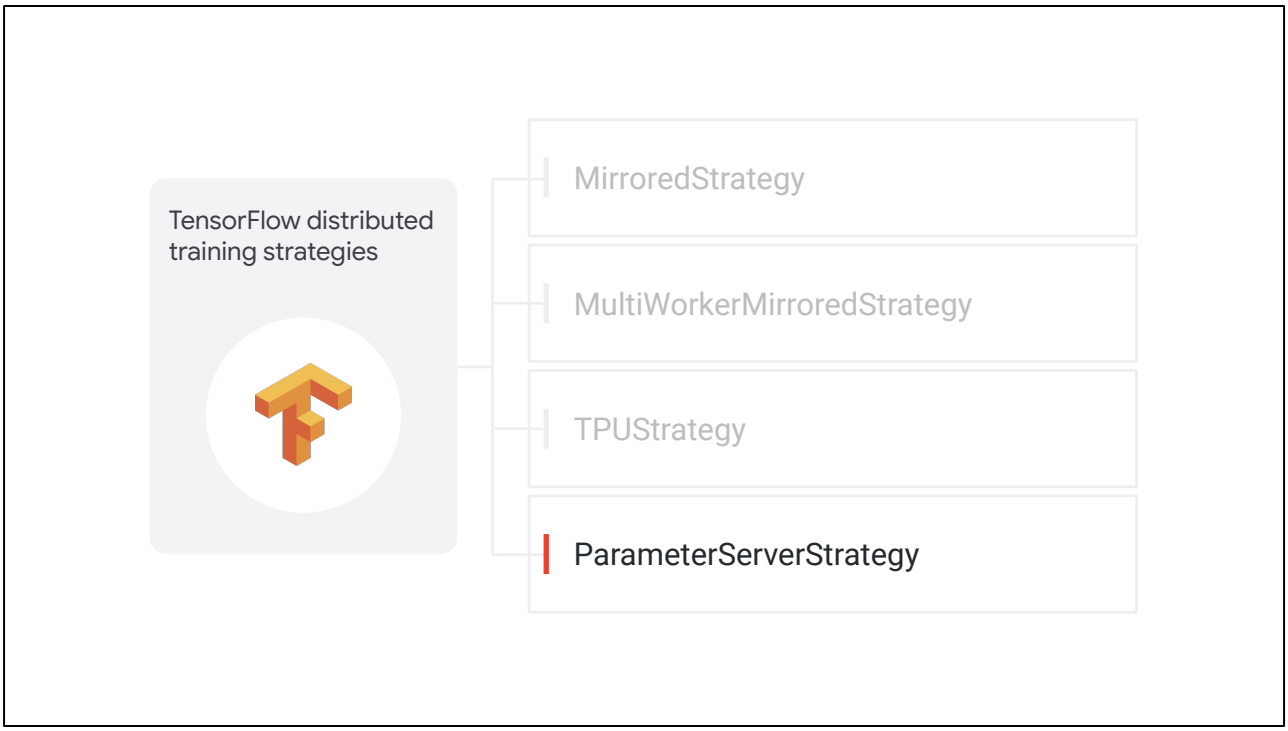
Google Cloud

ParameterServerStrategy

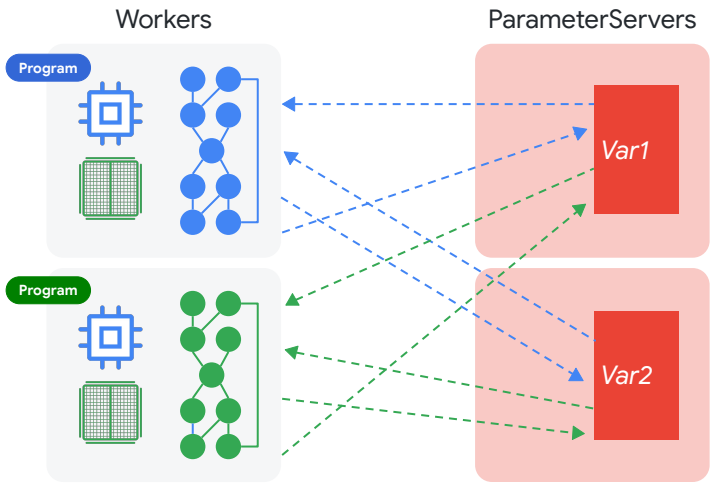


Module 03
Designing high-performance ML systems

Earlier we explored the asynchronous parameter server architecture earlier.



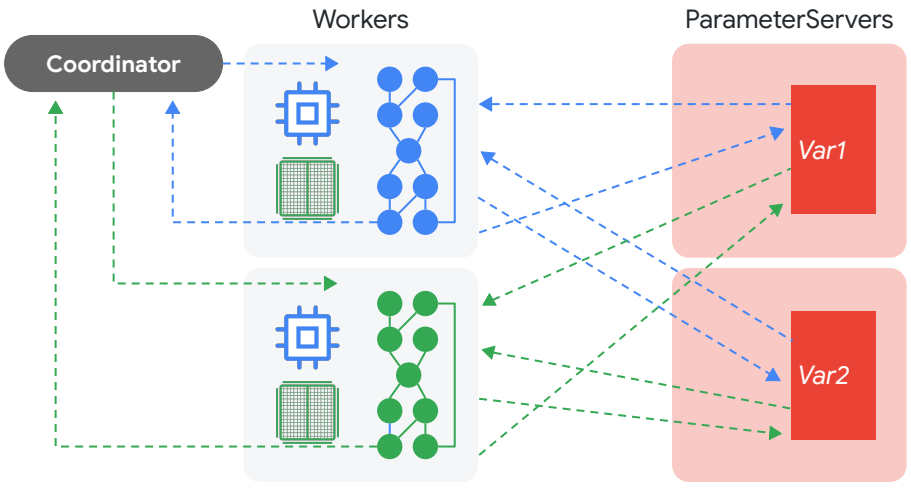
A parameter server training cluster consists of workers and parameter servers.



Earlier we explored the asynchronous parameter server architecture earlier. A parameter server training cluster consists of workers and parameter servers.

Variables are created on parameter servers and they are read and updated by workers in each step. By default, workers read and update these variables independently without synchronizing with each other.

If you used Parameter Server Strategy in TensorFlow 1, you might recall that each worker ran its own training program.



In TensorFlow 2, we introduce a central coordinator.

The coordinator is a special task type that creates resources, dispatches training tasks, writes checkpoints, and deals with task failures.

```
strategy = tf.distribute.experimental.ParameterServerStrategy(
          tf.distribute.cluster_resolver.TFConfigClusterResolver())

def dataset_fn(input_context):
    ...
    return dataset

dc = tf.keras.utils.experimental.DatasetCreator(dataset_fn)

with strategy.scope():
    model = tf.keras.Sequential(...)
    model.compile(loss='mse', optimizer='sgd')

model.fit(dc, epochs=5, steps_per_epoch=20, callbacks=callbacks)
```

You can create your parameter server strategy object just like you would for the other strategies.

Note that you will need to pass in the cluster resolver argument and if training with AI platform, this is just a simple TFConfigClusterResolver.

```
strategy = tf.distribute.experimental.ParameterServerStrategy(
           tf.distribute.cluster_resolver.TFConfigClusterResolver())

def dataset_fn(input_context):
    ...
    return dataset

dc = tf.keras.utils.experimental.DatasetCreator(dataset_fn)

with strategy.scope():
    model = tf.keras.Sequential(...)
    model.compile(loss='mse', optimizer='sgd')

model.fit(dc, epochs=5, steps_per_epoch=20, callbacks=callbacks)
```

Using model fit with parameter server training requires that the input data be provided in a callable object that takes a single argument of type tf.distribute.InputContext and returns a tf.data.Dataset.

We then need to wrap our dataset function in tf.keras.utils.experimental.DatasetCreator.

The code in dataset_fn will be invoked on the input device, which is usually the CPU on each of the worker machines.

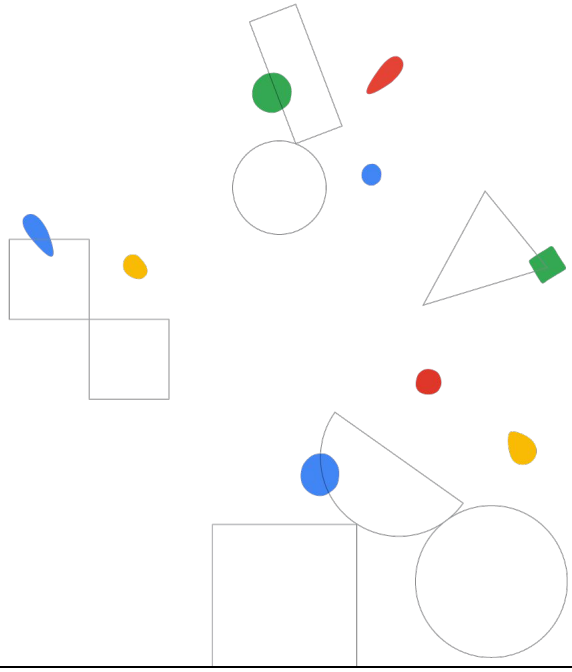
```
strategy = tf.distribute.experimental.ParameterServerStrategy(
          tf.distribute.cluster_resolver.TFConfigClusterResolver())

def dataset_fn(input_context):
    ...
    return dataset

dc = tf.keras.utils.experimental.DatasetCreator(dataset_fn)

with strategy.scope():
    model = tf.keras.Sequential(...)
    model.compile(loss='mse', optimizer='sgd')

model.fit(dc, epochs=5, steps_per_epoch=20, callbacks=callbacks)
```

When using parameter service strategy, it is recommended that you shuffle and repeat your dataset and pass in the steps_per_epoch argument to model.fit.

Google Cloud

Lab

Distributed training with Keras

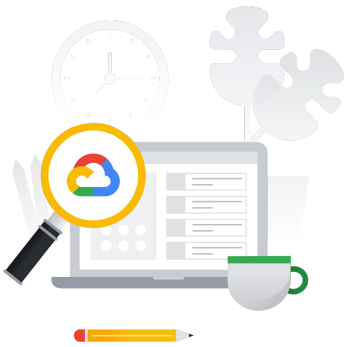


Module 03

Designing high-performance ML systems

This lab provides guidance on how to use distributed training with Keras.

Lab objectives



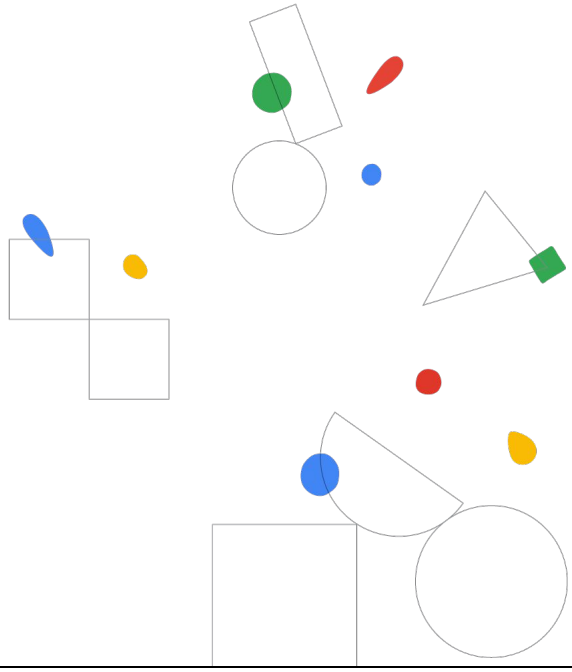
- Define distribution strategy and set input pipelines.
- Create a Keras model.
- Define callbacks.
- Train and evaluate a model.

- You'll begin by defining a distribution strategy and setting an input pipeline,
- then continue on to create a Keras model,
- define callbacks,
- and finally, train and evaluate a model.

Google Cloud

Lab

Distributed training using GPUs on Google Cloud's AI Platform



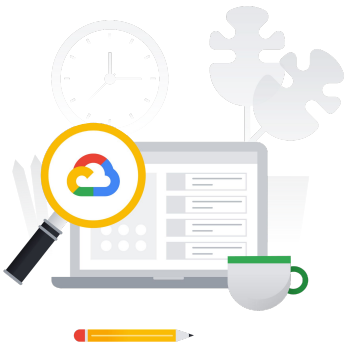
Module 03

Designing high-performance ML systems

This lab provides hands-on practice using Google Cloud's AI Platform to perform distributed training using the MirroredStrategy found within tf.keras.

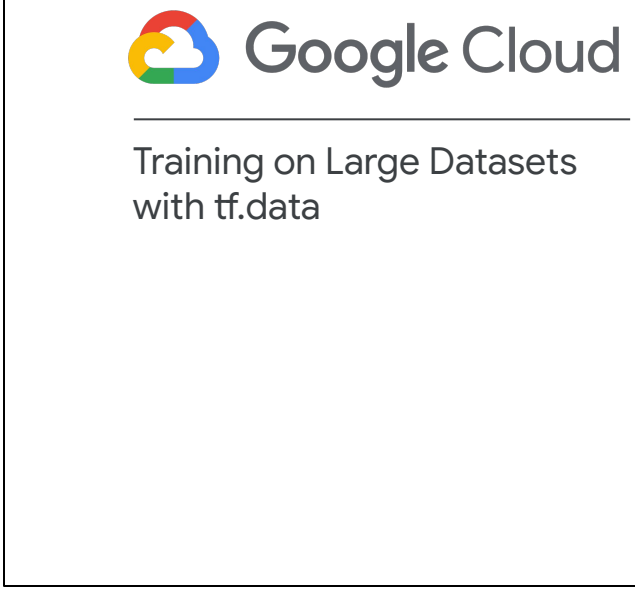
This strategy allows the use of the synchronous AllReduce strategy on a virtual machine with multiple GPUs attached.

Lab objectives



- Set up the environment.
- Create a deep neural network model using the Fashion MNIST dataset.
- Train that model using a MultiWorkerMirroredStrategy running on multiple GPUs.

- You'll start by setting up the environment,
- then continue on to create a deep neural network model using the Fashion MNIST dataset,
- and then, finally, you'll train that model using a MultiWorkerMirroredStrategy running on multiple GPUs.



Data is one of the most crucial components of your machine learning model. Collecting the right data is not enough. You also need to make sure you put the right processes in place to clean, analyze and transform the data, as needed, so that the model can take the most signal from that data as possible.

Models which are deployed in production, specially, require lots and lots of data. This is data that likely won't fit in memory and can possibly be spread across multiple files or may be coming from an input pipeline.

The tf.data API enables you to build those complex input pipelines from simple, reusable pieces. For example, the pipeline for structured data might require normalization, feature crosses or bucketization. An image model might aggregate data from files in a distributed file system, apply random perturbations to each image, and merge randomly selected images into a batch for training. The pipeline for a text model might involve extracting symbols from raw text data, converting them to embedding identifiers with a lookup table, and batching together sequences of different lengths.

The tf.data API makes it possible to handle large amounts of data, read from different data formats, and perform complex transformations.

A tf.data.Dataset allows you to

- Create data pipelines from
  - in-memory dictionary and lists of tensors
  - out-of-memory sharded data files
- Preprocess data in parallel (and cache result of costly operations)

```
dataset = dataset.map(preproc_fun).cache()
```

- Configure the way the data is fed into a model with a number of chaining methods

```
dataset = dataset.shuffle(1000).repeat(epochs).batch(batch_size, drop_remainder=True)
```

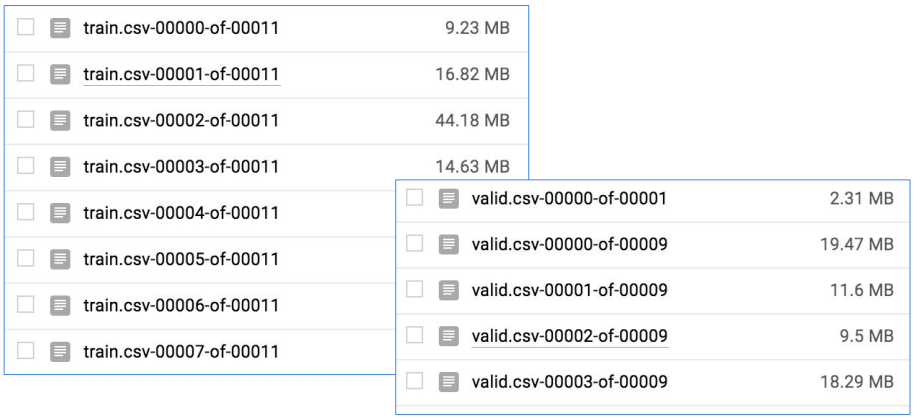
in a easy and very compact way

It's time to look at some specifics. The tf.data API introduces a tf.data.Dataset abstraction that represents a sequence of elements, in which each element consists of one or more components. For example, in an image pipeline, an element might be a single training example, with a pair of tensor components representing the image and its label.

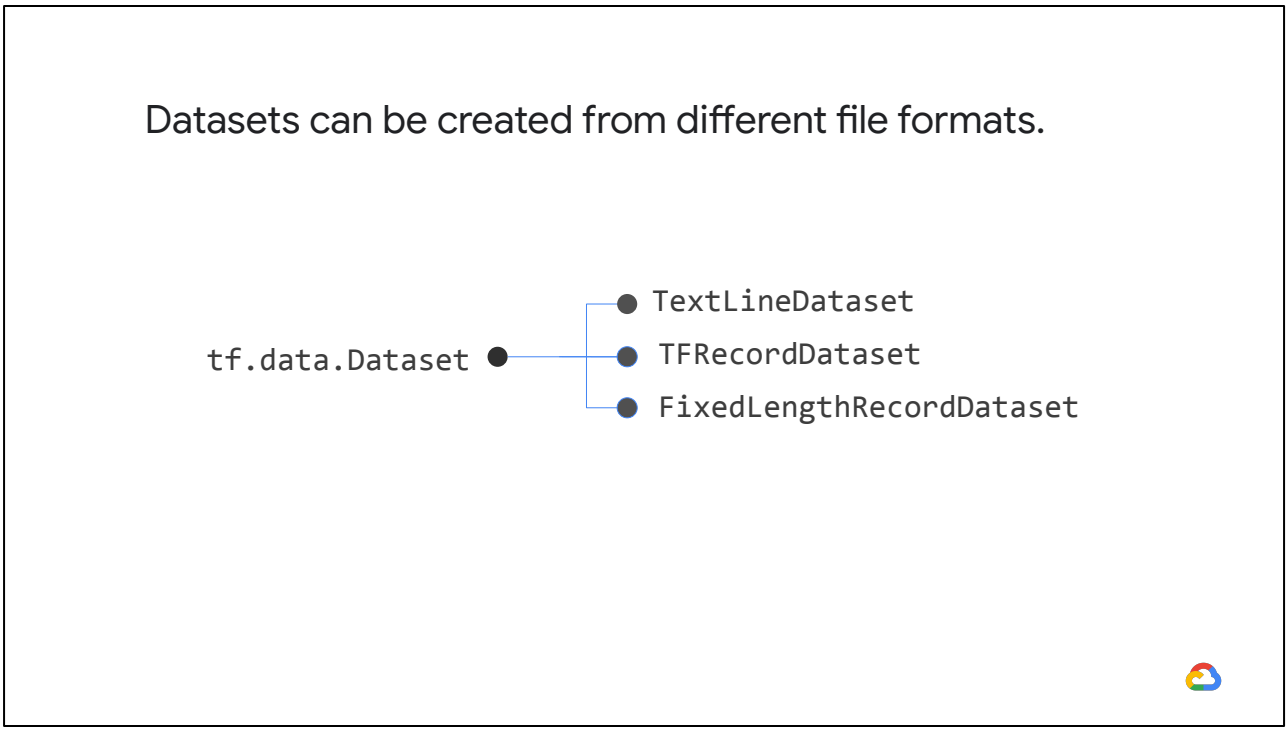
There are two distinct ways to create a dataset:

1. A data source constructs a Dataset from data stored in memory or in one or more files.
2. A data transformation constructs a dataset from one or more tf.data.Dataset objects.

What about out-of-memory sharded datasets?



Large datasets tend to be sharded or broken apart into multiple files which can be loaded progressively. Remember that you train on mini-batches of data. You do not need to have the entire dataset in memory. One mini-batch is all you need for one training step.



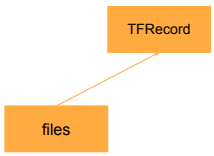
The Dataset API will help you create input functions for your model that will load data progressively. There are specialized Dataset classes that can read data from text files like CSVs, Tensorflow records or fixed length record files.

Datasets can be created from different file formats:

- Use TextLineDataset to instantiate a Dataset object which is comprised of lines from one or more text files.
- TFRecordDataset comprises records from one or more TFRecord files.
- And FixedLengthRecordDataset is a dataset object from fixed-length records from one or more binary files.

For anything else, you can use the generic Dataset class and add your own decoding code.

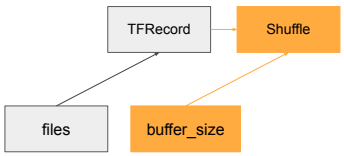
```
dataset = tf.data.TFRecordDataset(files)
```



Let's walk through an example of TFRecordDataset.

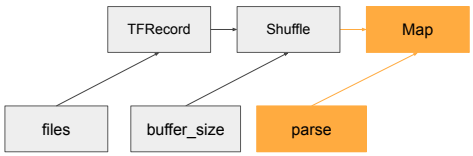
At the beginning, the TFRecord op is created and executed, producing a variant tensor representing a dataset which is stored in the corresponding Python object.

```
dataset = tf.data.TFRecordDataset(files)

dataset = dataset.shuffle(buffer_size=X)
```



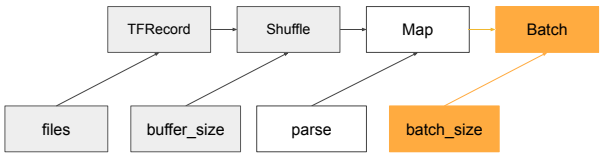
Next, the Shuffle op is executed, using the output of the TFRecord op as its input, connecting the two stages of the input pipeline.

```
dataset = tf.data.TFRecordDataset(files)

dataset = dataset.shuffle(buffer_size=X)

dataset = dataset.map(lambda record: parse(record))
```



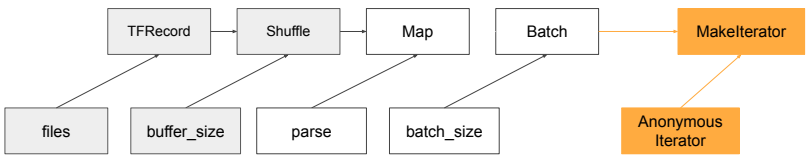
Next, the user-defined function is traced and passed as attribute to the Map operation, along with the Shuffle dataset variant input.

```
dataset = tf.data.TFRecordDataset(files)

dataset = dataset.shuffle(buffer_size=X)

dataset = dataset.map(lambda record: parse(record))

dataset = dataset.batch(batch_size=Y)
```



Finally, the Batch op is created and executed, creating the final stage of the input pipeline.

```
dataset = tf.data.TFRecordDataset(files)

dataset = dataset.shuffle(buffer_size=X)

dataset = dataset.map(lambda record: parse(record))

dataset = dataset.batch(batch_size=Y)


for element in dataset:  #  iter() is called

  ...
```

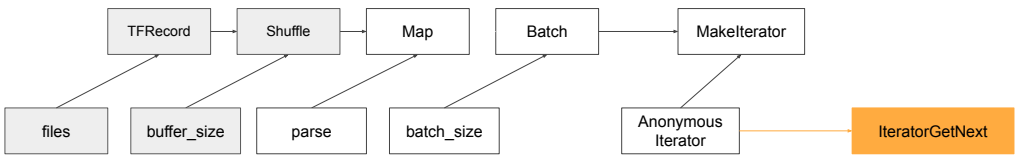


When the for loop mechanism is used for enumerating the elements of dataset, the `iter` method is invoked on the dataset, which triggers creation and execution of two ops.

First an anonymous iterator op is created and executed, which results in creation of an iterator resource.

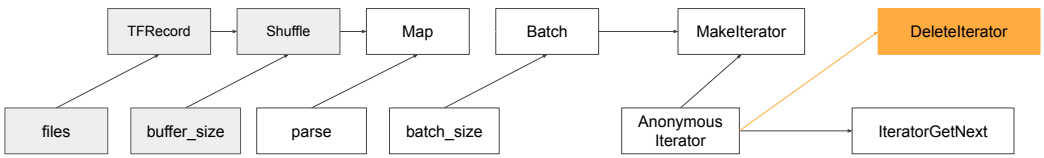
Subsequently, this resource along with the Batch dataset variant is passed into the MakeIterator op, initializing the state of the iterator resource with the dataset.

```
dataset = tf.data.TFRecordDataset(files)

dataset = dataset.shuffle(buffer_size=X)

dataset = dataset.map(lambda record: parse(record))

dataset = dataset.batch(batch_size=Y)


for element in dataset:

  ...  #  next() is called
```



When the `next` method is called, it triggers creation and execution of the IteratorGetNext op, passing in the iterator resource as the input. Note that the iterator op is created only once but executed as many times as there are elements in the input pipeline.

```
dataset = tf.data.TFRecordDataset(files)

dataset = dataset.shuffle(buffer_size=X)

dataset = dataset.map(lambda record: parse(record))

dataset = dataset.batch(batch_size=Y)


for element in dataset:

  ... #  iterator goes out of scope
```

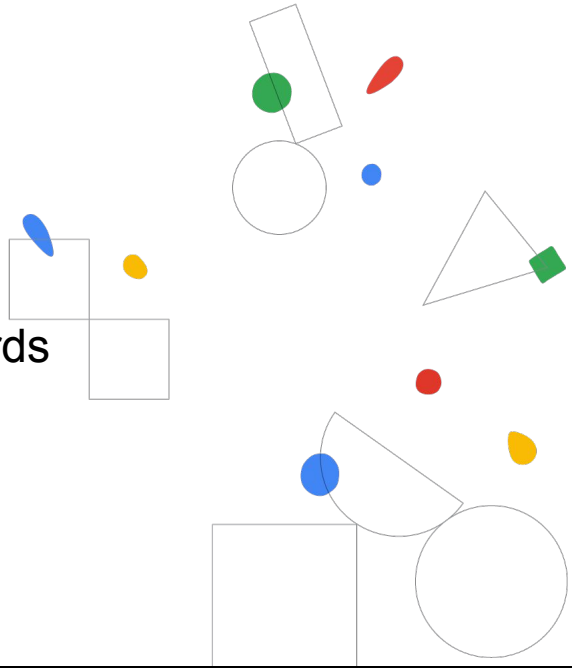


Finally, when the Python iterator object goes out of scope, the DeleteIterator op is executed to make sure the iterator resource is properly disposed of. To state the obvious, properly disposing of the iterator resource is essential as it is not uncommon for the iterator resource to allocate 100MBs to GBs of memory because of internal buffering.

Google Cloud

Lab

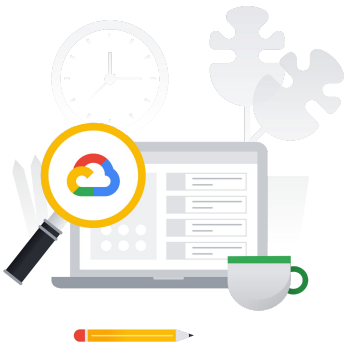
TPU-Speed data pipelines: tf.data.Dataset and TFRecords



Production Machine Learning Systems

In this lab, you'll get practice loading data from Google Cloud Storage with the tf.data.Dataset API to feed a TPU.

Lab objectives



- Use the tf.data.Dataset API to load training data.
- Use TFRecord format to load training data from Google Cloud Storage.

then use the TFRecord format to load training data from Google Cloud Storage.

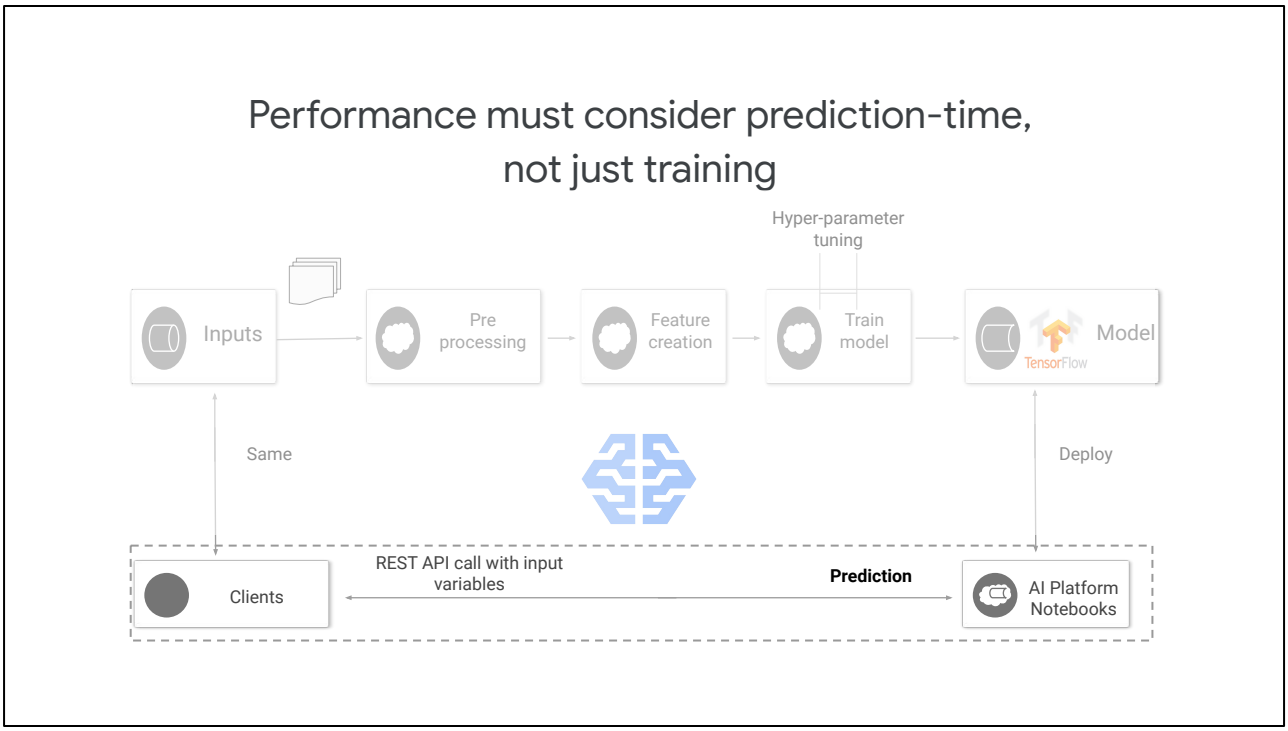
Agenda

Distributed training

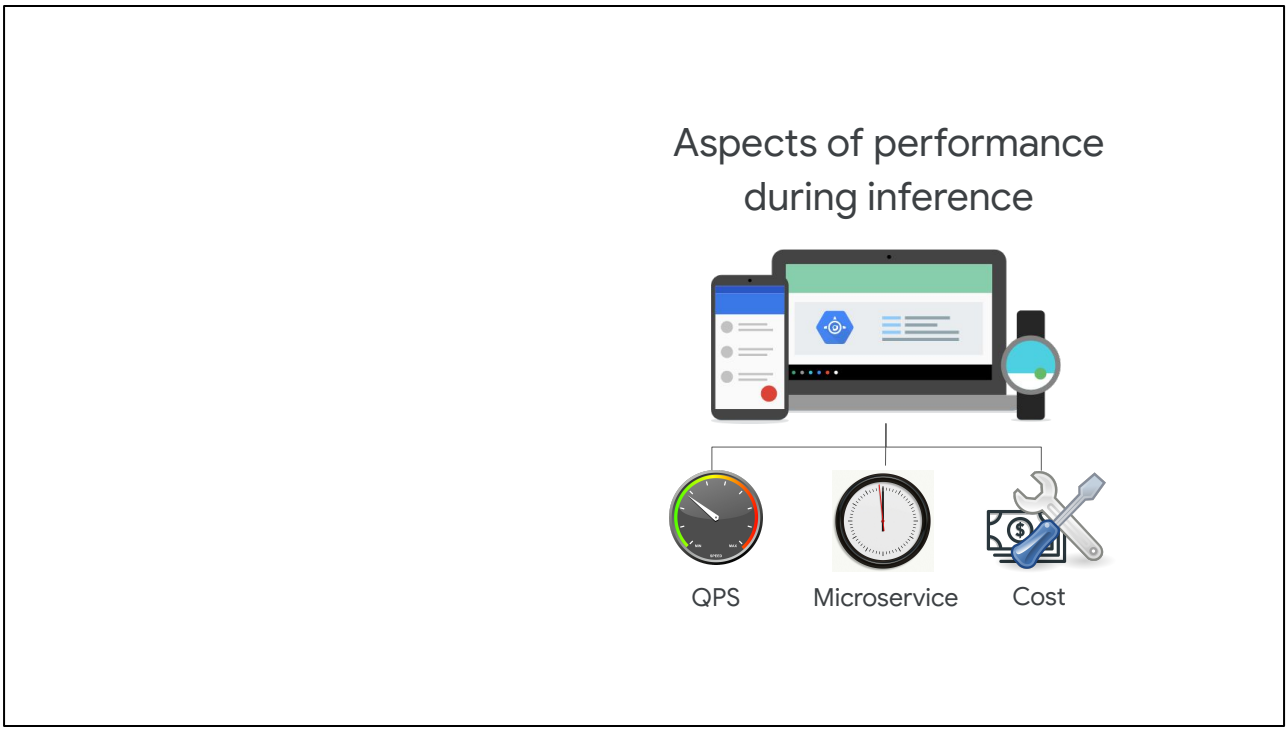
Faster input pipelines

Inference

So far, we have looked at training performance. Now, we will take a look at performance when it comes to predictions.

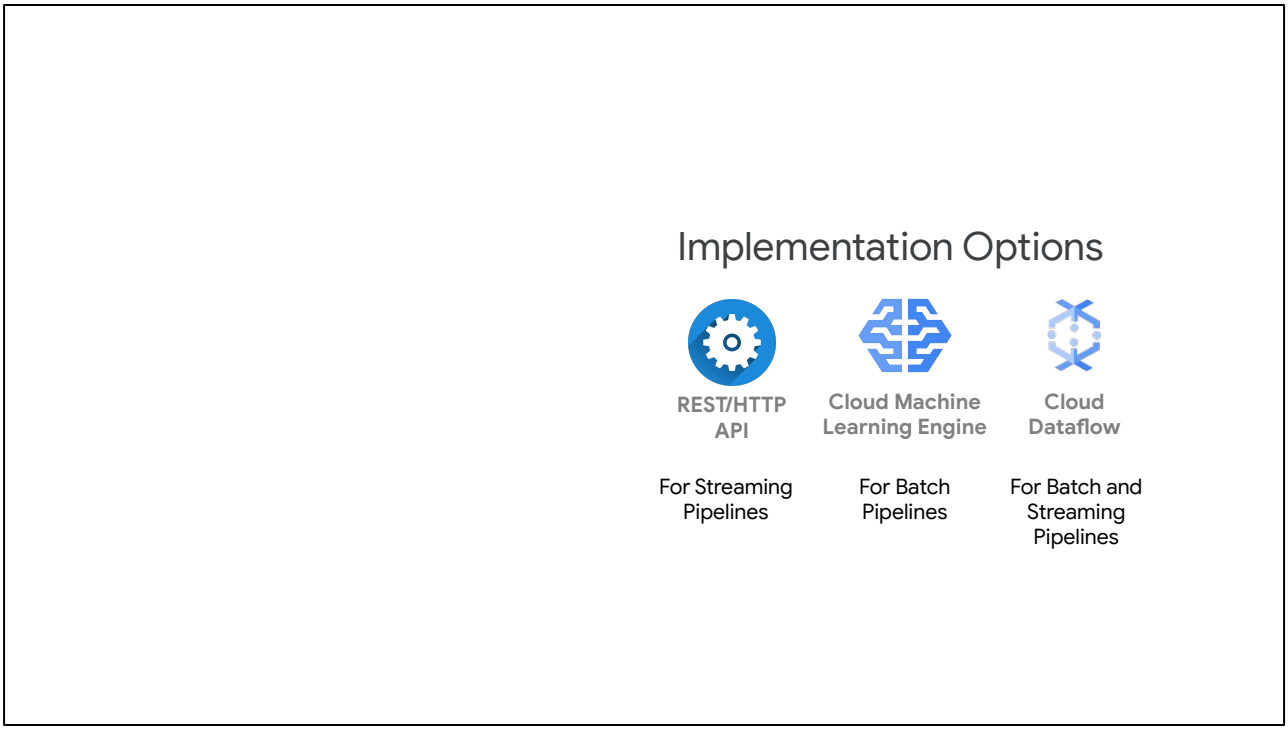


How do you obtain high-performance inference?



You need to consider several aspects:

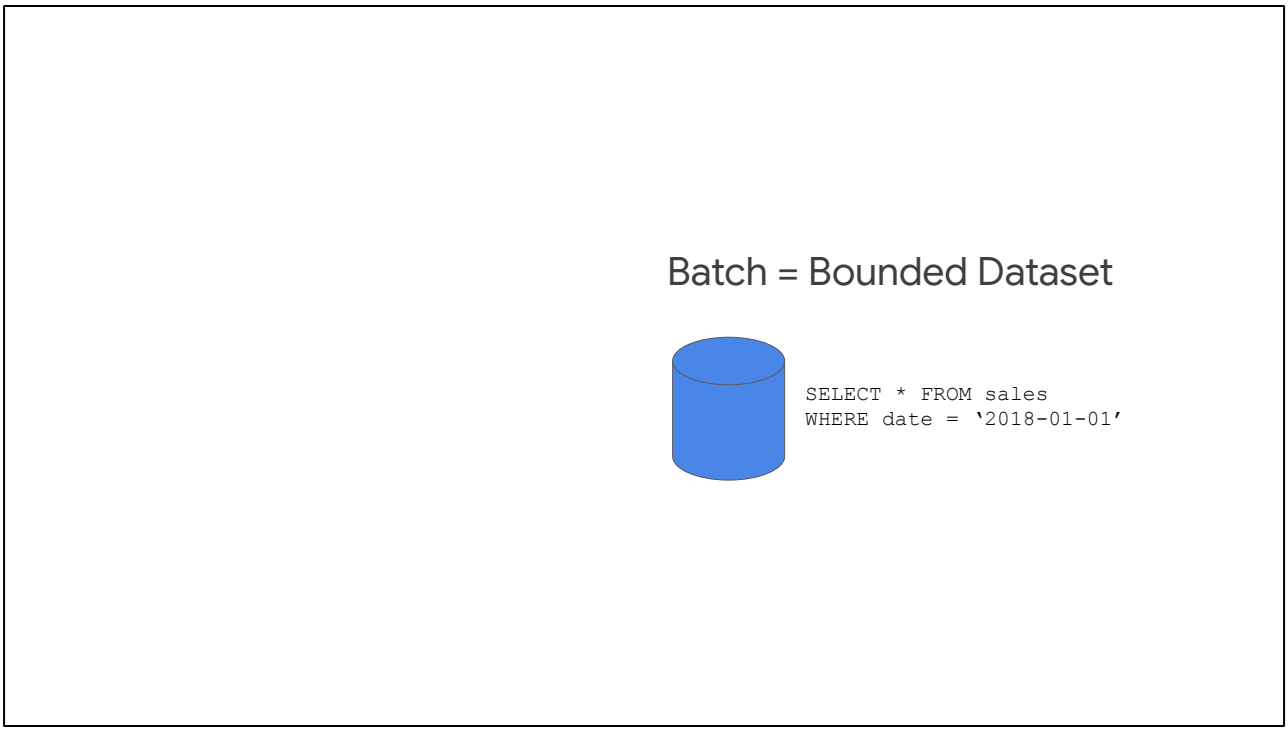
- Throughputs requirements -- how many queries per second do you need to process?
- Latency requirements -- how long can a query take?
- Cost -- in terms of infrastructure *and* in terms of maintenance



There are essentially three approaches to implementing this:

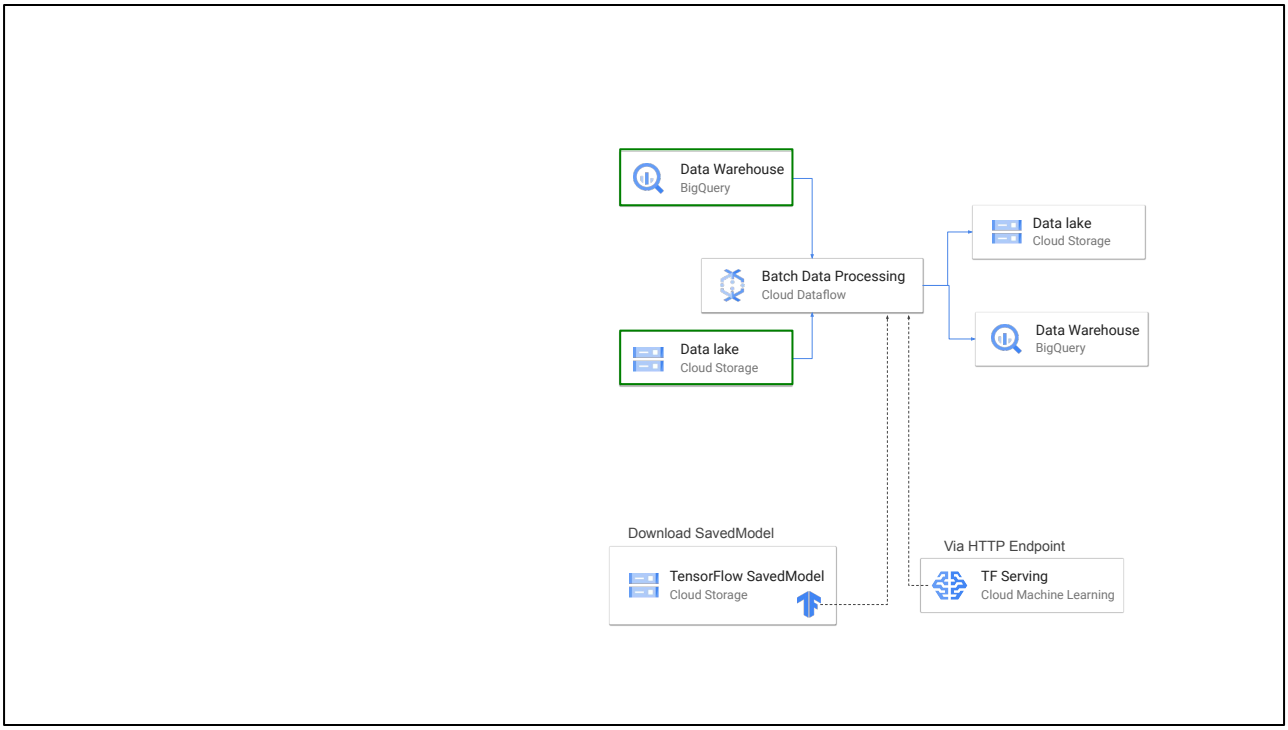
- Using a deployed model REST/HTTP API (for streaming pipelines)
- Using Cloud ML Engine batch prediction jobs (for batch pipelines.)
- Using Cloud Dataflow direct model prediction (for both batch and streaming pipelines)

Let's take the third option and delve into it a bit -- this will help clarify our terminology as well

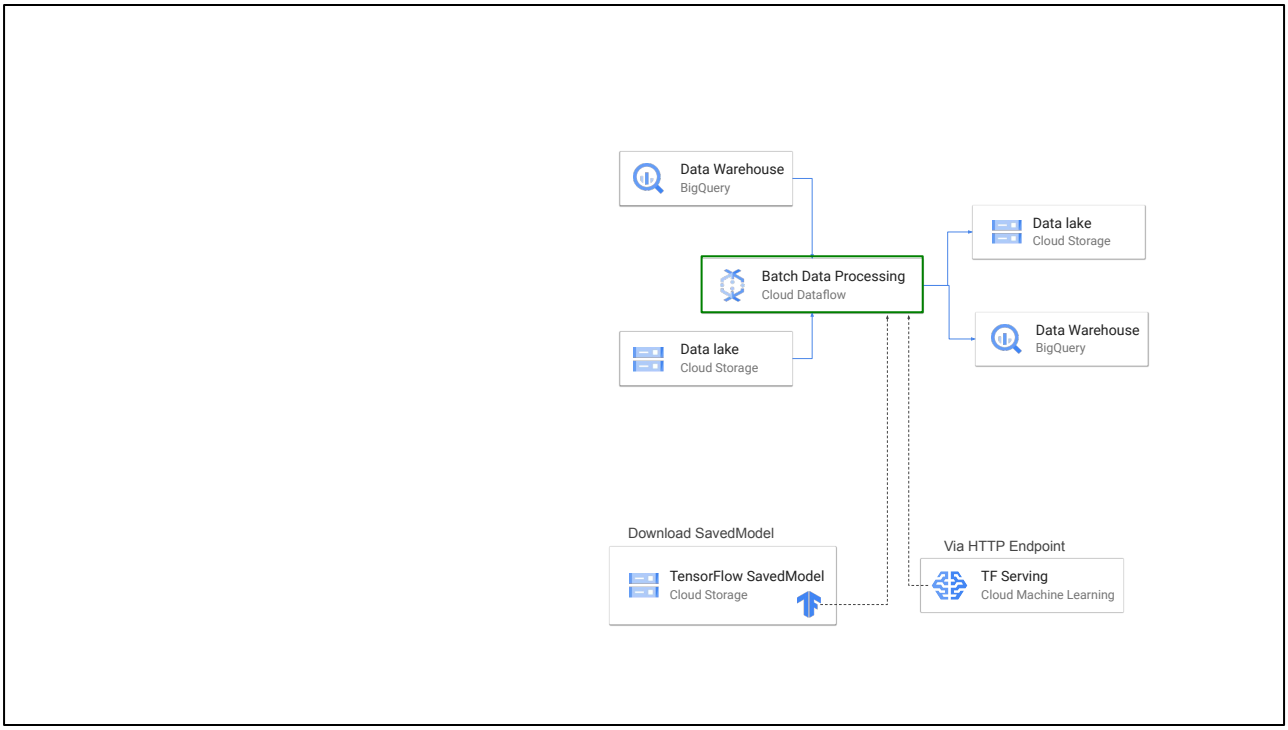


We are using the word “batch” differently from the word “batch” in ML training.

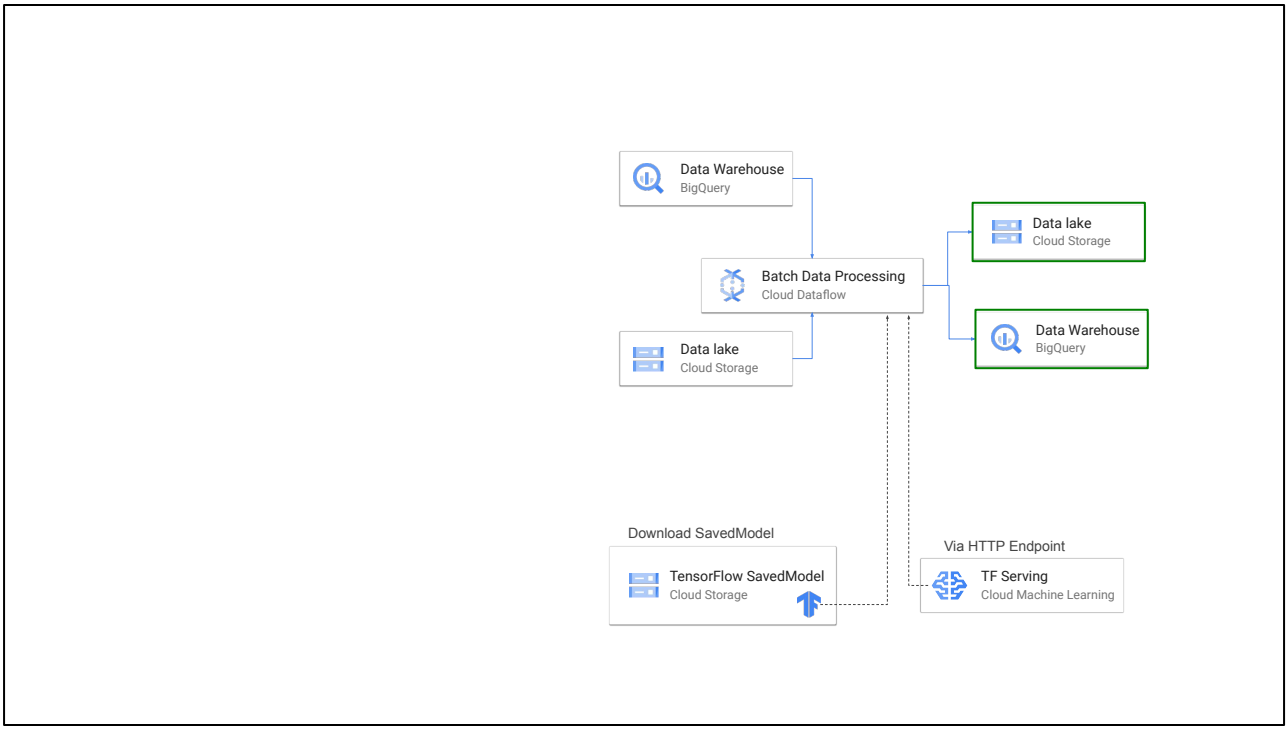
Here, we are using batch to refer to a bounded dataset.



A typical batch data pipeline reads data from some persistent storage, either a data lake like Google Cloud Storage or a data warehouse like BigQuery,

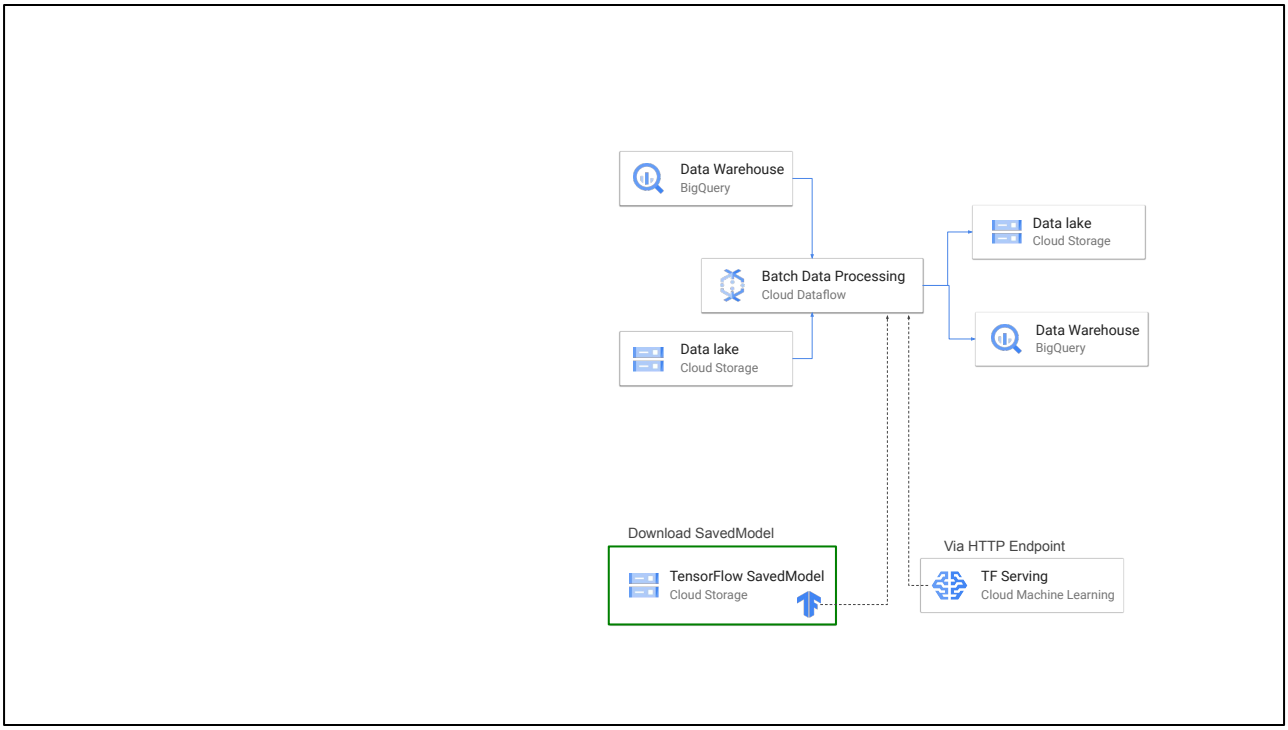


does some processing and

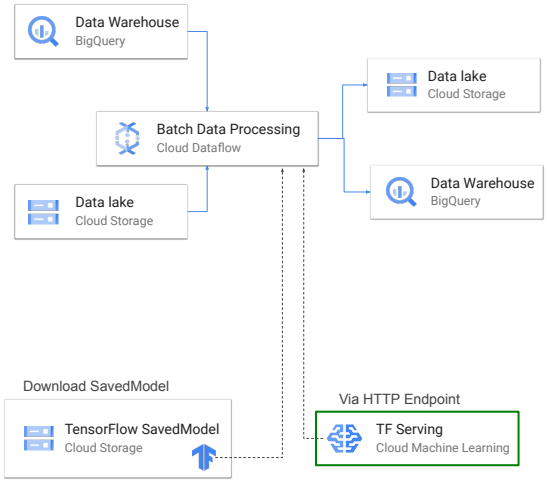


writes it out to the same or different format.

The processing, carried by Cloud Dataflow, typically enriches the data with the predictions of an ML model.

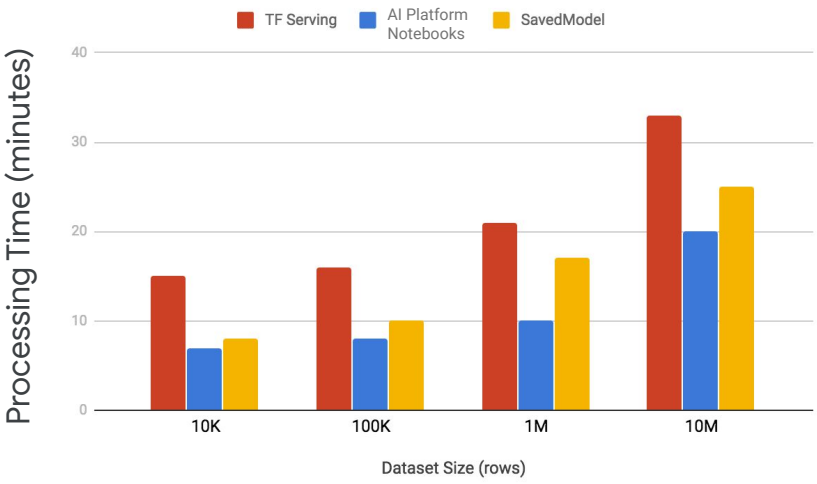


There are two options to do this, either by using a TensorFlow SavedModel and loading it directly into the Dataflow pipeline from Cloud Storage,



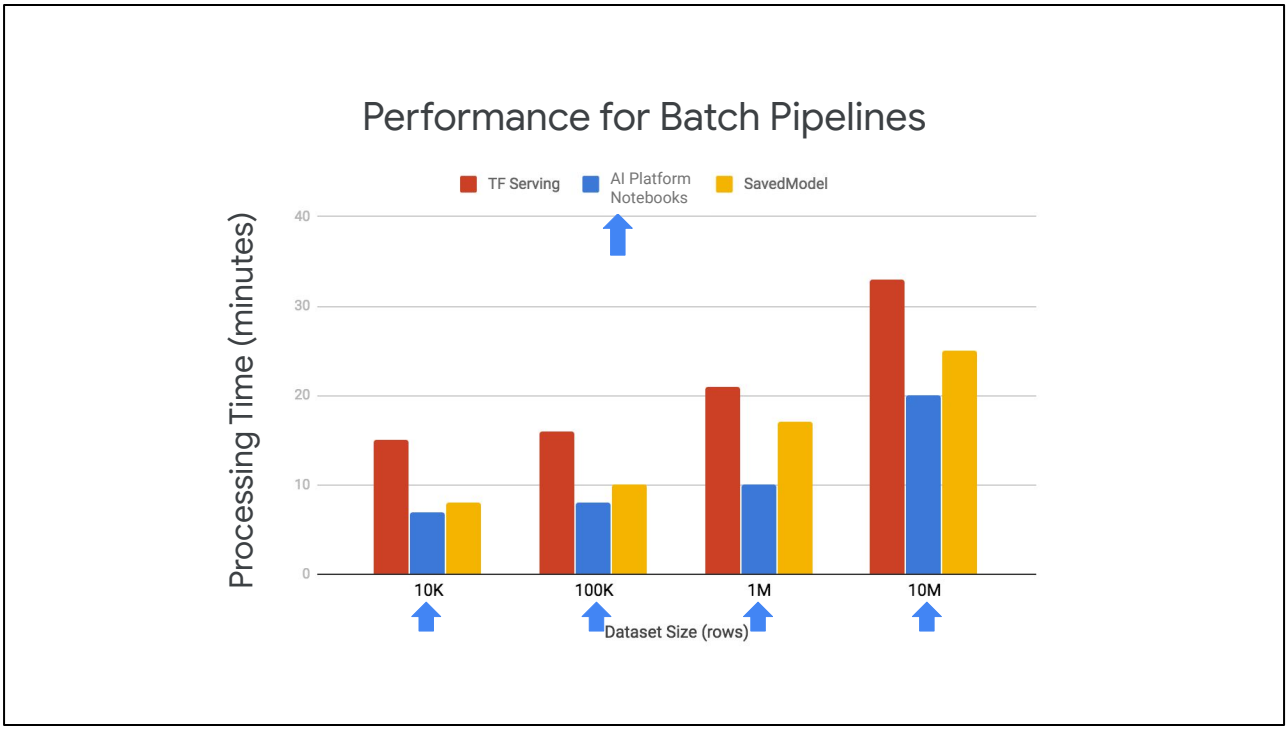
Or by using TF Serving and accessing it via an HTTP end-point as a microservice, either from Cloud ML Engine as shown or using Kubeflow, running on Kubernetes Engine.

So far, we have used the HTTP end-point approach, but for performance reasons you might want to consider the SavedModel approach as well.

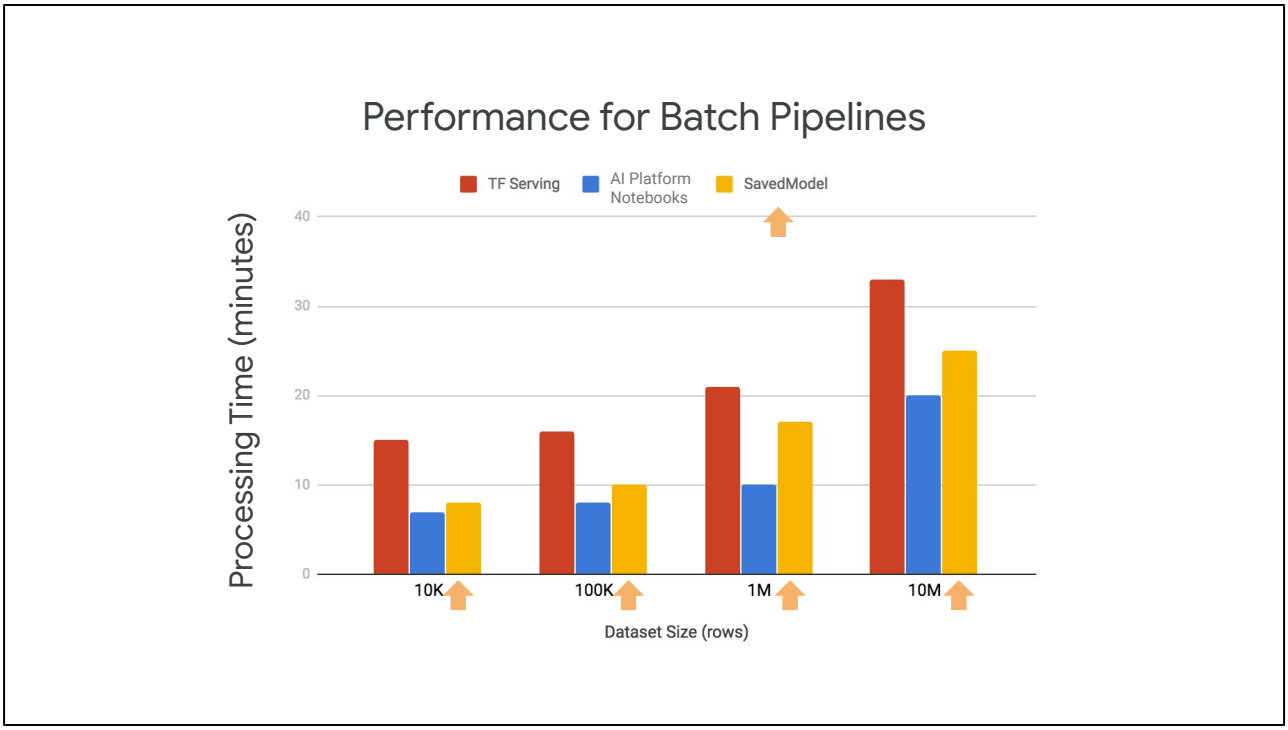


So, what option gives the best performance for batch pipelines?

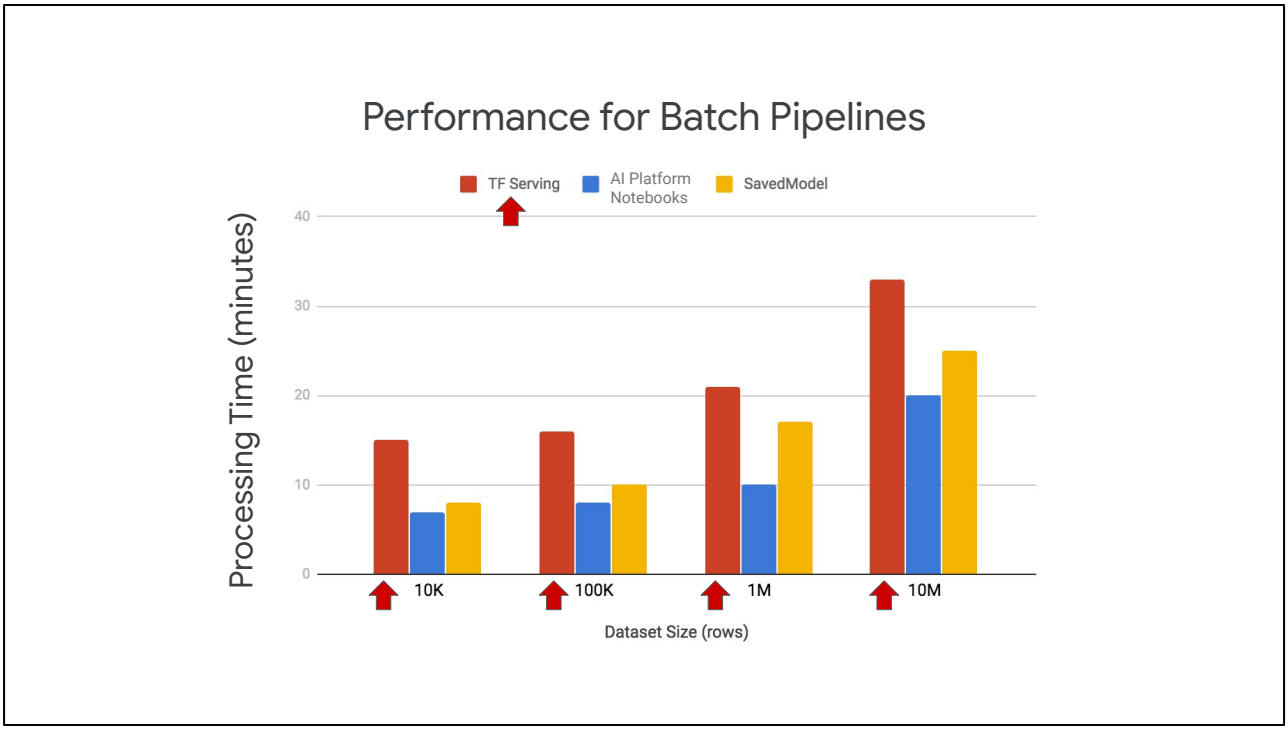
As usual, this depends on which aspect is most important to you.



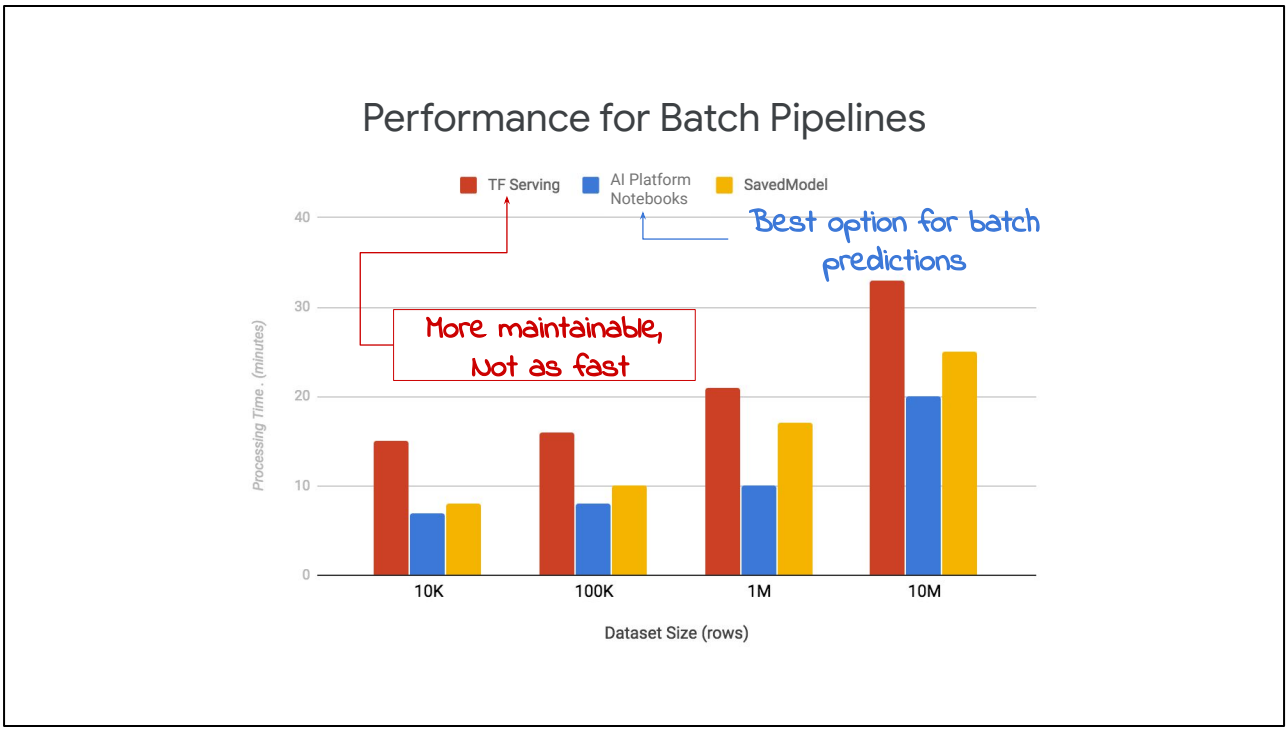
In terms of raw processing speed, you want to use Cloud ML Engine batch predictions.



Next fastest is to directly load the SavedModel into your Dataflow job and invoke it.



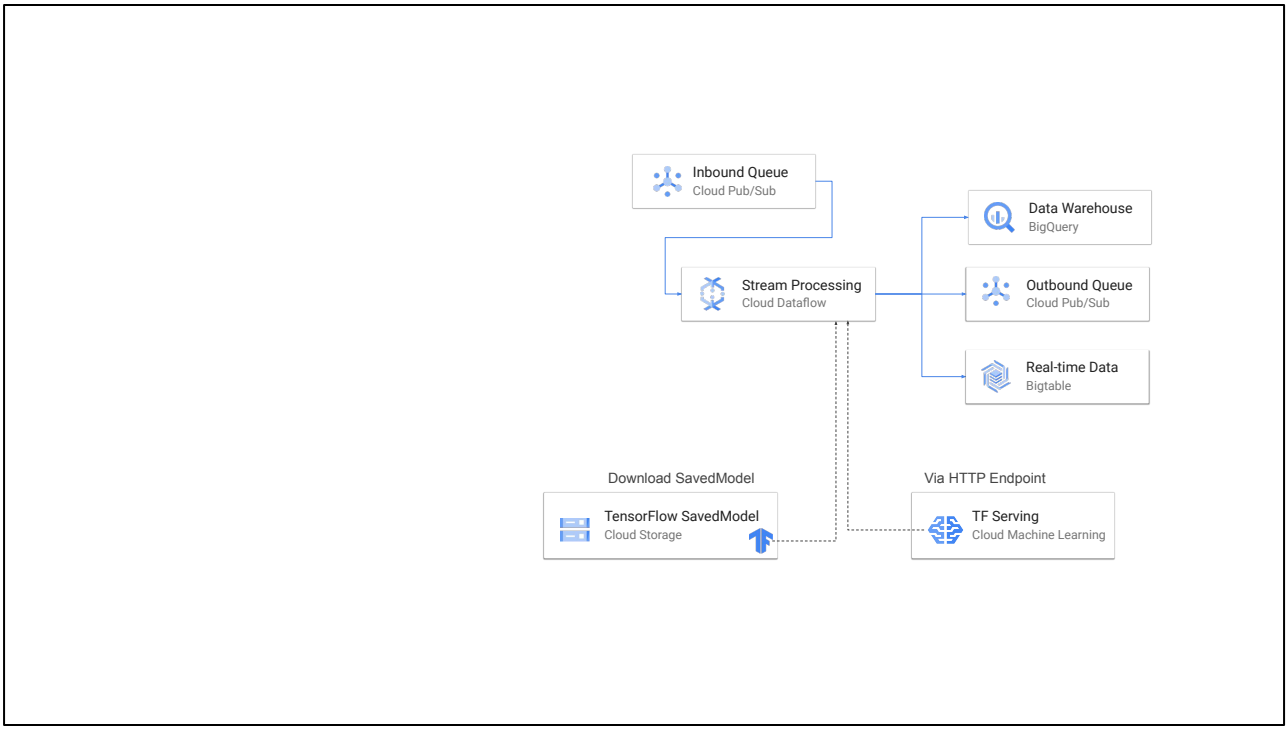
The third option in terms of speed is to use TensorFlow Serving on Cloud ML Engine.



But if you want maintainability, the 2nd and 3rd options reverse. The batch prediction is still the best -- what's not to love about a fully managed service?

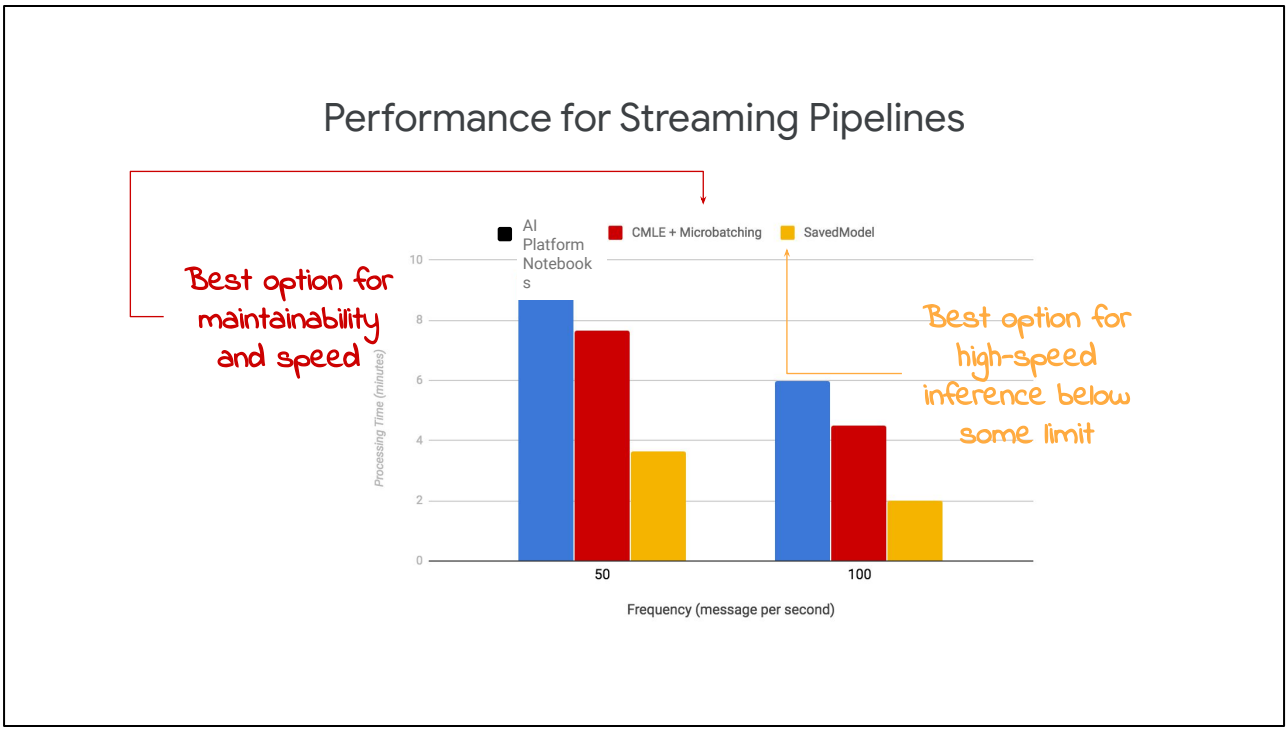
But using online predictions as a microservice allows for easier upgradability and dependency management than loading up the current version into the Dataflow job.

This graph is from an upcoming solution -- see https://cloud.google.com/solutions -- by the time this video is available, the solution might already have been published.



A streaming pipeline is similar, except that the input dataset is not bounded. So, we read it from an unbounded source like Pub/Sub and process it with Dataflow.

You have two options, of SavedModel or TensorFlow serving here as well, with TF serving hosted on Cloud ML Engine.



For streaming pipelines, the SavedModel approach is the fastest.

Using minibatching as we recommended earlier in the module on implementing serving helps reduce the gap between the TF Serving http endpoint approach supported by Cloud ML Engine and directly loading the model into the client.

However, the CMLE approach is much more maintainable especially when the model will be used from multiple clients.

Another thing to keep in mind is that as the number of queries per second keeps increasing, at some point, the SavedModel approach will become infeasible, but the Cloud ML Engine approach should scale indefinitely.