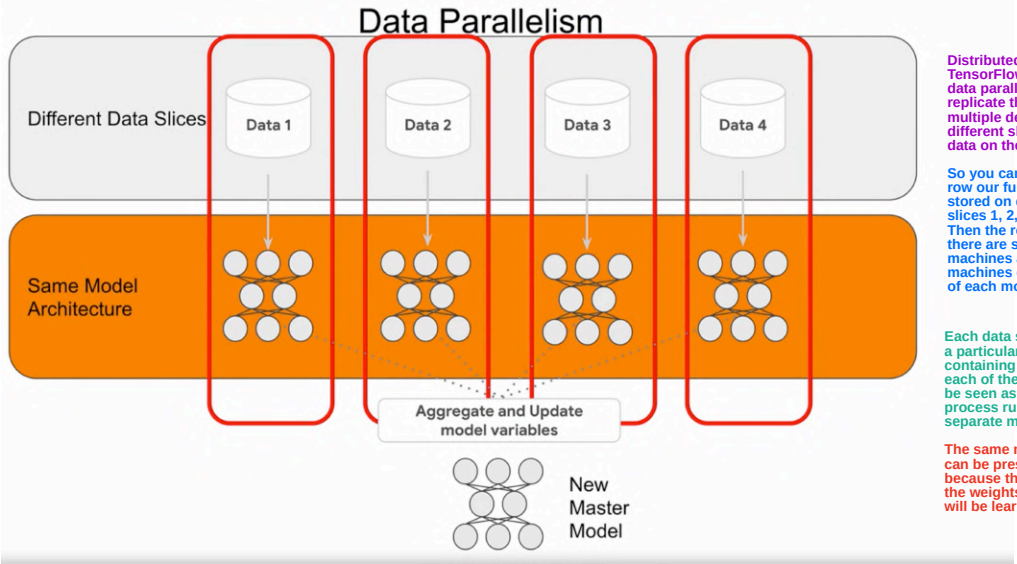
This week, we will go beyond training on a single devices and see what it takes to train on multiple devices. As your models get bigger and more complex, it may become infeasible to train them on a single CPU, GPU, or even TPU. So you might need to figure out ways to distribute the training across multiple ones with a strategy which depends on the hardware available.

To achieve this, TensorFlow has come up with various strategies where you can distribute your data across clusters of machines. Each one of these clusters can have one or more devices, can carry out large scale training on your models accordingly.

This is typically called a distribution strategy and in these sections, you'll start learning how distribution strategies work.

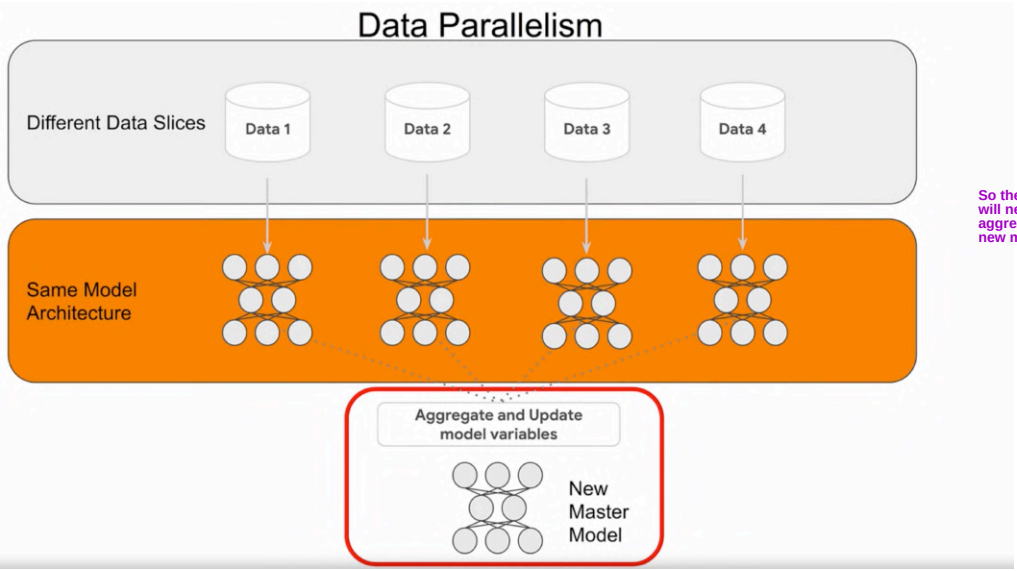


Distributed training in TensorFlow is built around data parallelism where we replicate the same model on multiple devices and run different slices of the input data on them.

So you can see on the top row our full dataset is stored on different data slices 1, 2, 3 and so on. Then the row below that, there are separate machines and each machines contains a copy of each model to be trained.

Each data slice feeds data to a particular machine containing the model. So each of the verticals should be seen as a separate process running on a separate machine.

The same model architecture can be present on each. But because the data is different, the weights in each model will be learned differently.



So these weights will neeed to be aggreated into a new master model.

tf.distribute.Strategy

- High-level APIs
- Custom training loops
- TensorFlow 2: eager mode & graph mode
- Supported on multiple configurations.
- Convenient to use with little to no code changes

Commonly used terms

- **Device**
- Replica
- Worker
- Mirrored variable

CPU
Accelerator: GPU, TPU

We use the term device to ever to any kind of machine that trains machine learning models. This could be CPU, or an accelerator like a GPU or a TPU.

It is possible to have multiple devices on a single computer. For example a machine could have a single cpu and 2 GPUs thereby having 3 total devices available for preprocessing and model training.

Commonly used terms

- Device
- **Replica**
- Worker
- Mirrored variable

CPU
Accelerator: GPU, TPU

During training, the copies of the model's variables are placed on several devices. This copy is often referred to as a replica.

Commonly used terms

A worker is softare running on a device dedicated to training the replica that's on that device.

- Device
- Replica
- **Worker**
- Mirrored variable

CPU
Accelerator: GPU, TPU

Commonly used terms

- Device
- Replica
- Worker
- Mirrored variable

CPU
Accelerator: GPU, TPU

Even though some variables in the replicas are trained independently of the other replicas, there are some variables that you want to be in sync across all devices.

The variables within these models that we want to keep in sync across all of the devices are called mirror variables.

Classifying strategies

Hardware platforms

- Single-machine multi-device
- Multi-machine (with 0 or more accelerators)

Training

- Synchronous (All-reduce)
- Asynchronous (Parameter Server)

One area where parallelism happens is in the hardware platform.

Now there's one setup of the hardware platform where there is a single machine that has multiple devices such as the CPU and one or more GPUs.

Classifying strategies

Hardware platforms

- Single-machine multi-device
- Multi-machine (with 0 or more accelerators)

Training

- Synchronous (All-reduce)
- Asynchronous (Parameter Server)

Another possible setup of the hardware platform is having multiple machines on a network.

Each of these machines can have a different number of accelerators and some might even have none.

Classifying strategies

Hardware platforms

- Single-machine multi-device
- Multi-machine (with 0 or more accelerators)

Training

- Synchronous (All-reduce)
- Asynchronous (Parameter Server)

The second area where parallelism can be a factor is in how you approach the training.

When it comes to training capabilities through data parallelism, there are two types; synchronous training and asynchronous training.

In synchronous training, all workers train over different slices of input data in sync with each other. They'll aggregate gradients at each step using an all-reduce algorithm. That's the type of training that you're mostly going to use in this course.

Classifying strategies

Hardware platforms

- Single-machine multi-device
- Multi-machine (with 0 or more accelerators)

Training

- Synchronous (All-reduce)
- Asynchronous (Parameter Server)

In asynchronous training, all workers are independently training over the input data and they're updating that variables asynchronously. They synchronize the distributed model through something called a parameter server architecture.

MirroredStrategy

- Single-machine multi-GPU
- Creates a replica per ***GPU***
- Each variable is ***mirrored***
- All-reduce ***across devices***

MultiWorkerMirroredStrategy

ParameterServerStrategy

DefaultStrategy

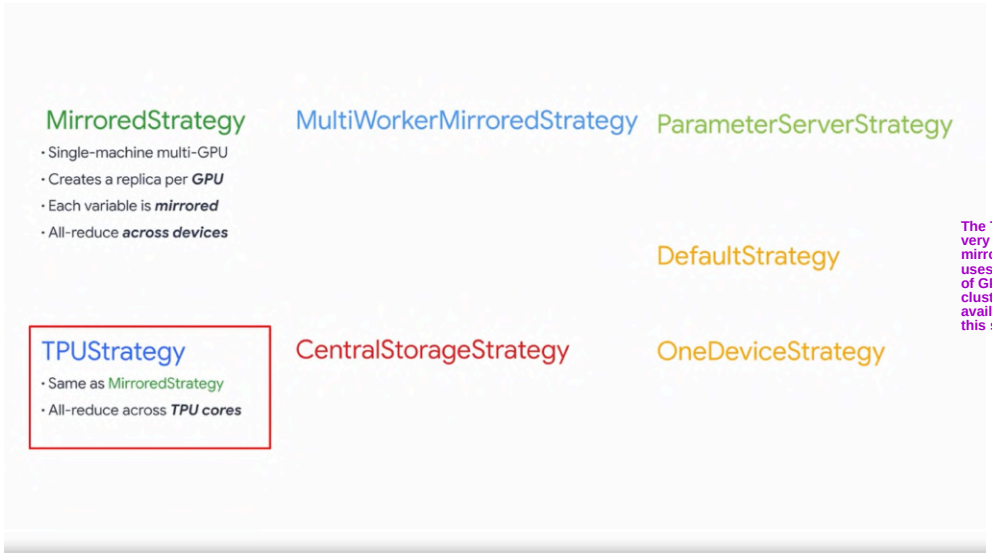
TPUStrategy

CentralStorageStrategy

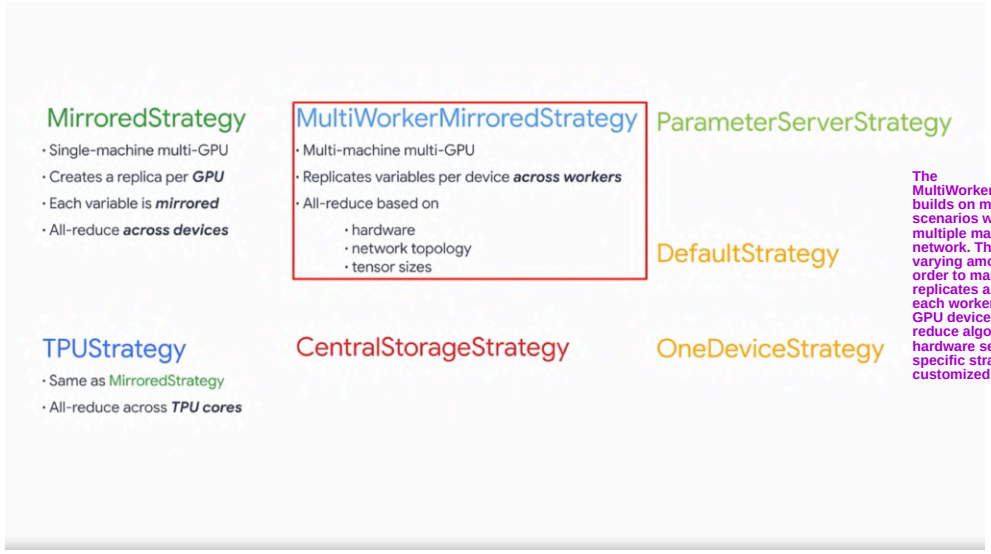
OneDeviceStrategy

TensorFlow supports a number of strategies to help you train using the scenarios in these previous slides. They're listed here. Mirrored strategy is generally what people will get started with distributed training. Where you have a single computer, but it supports multiple GPUs.

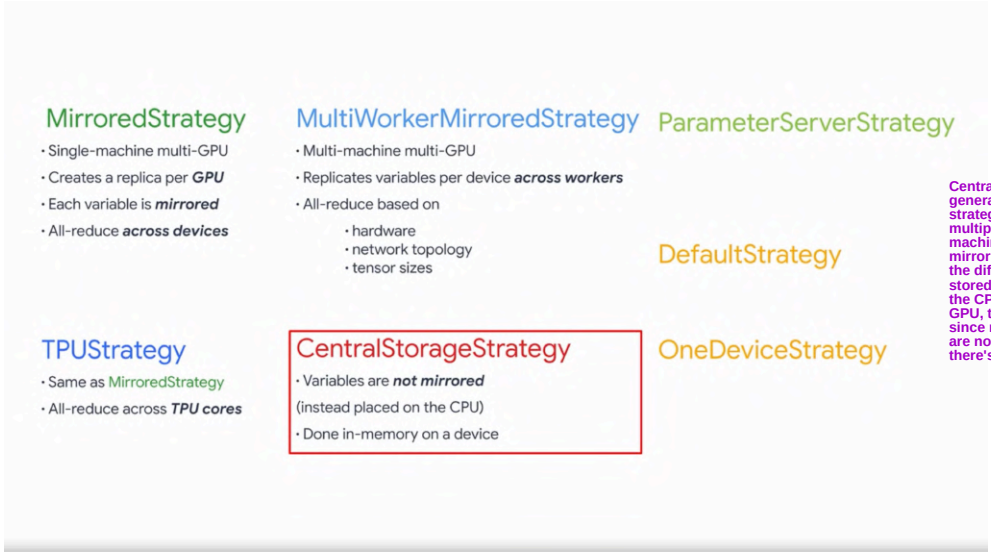
The idea here is that it will create a model replica on each GPU and then mirror the variables. After each epoch of training, the learned parameters are merged using an all-reduce across each of the devices.



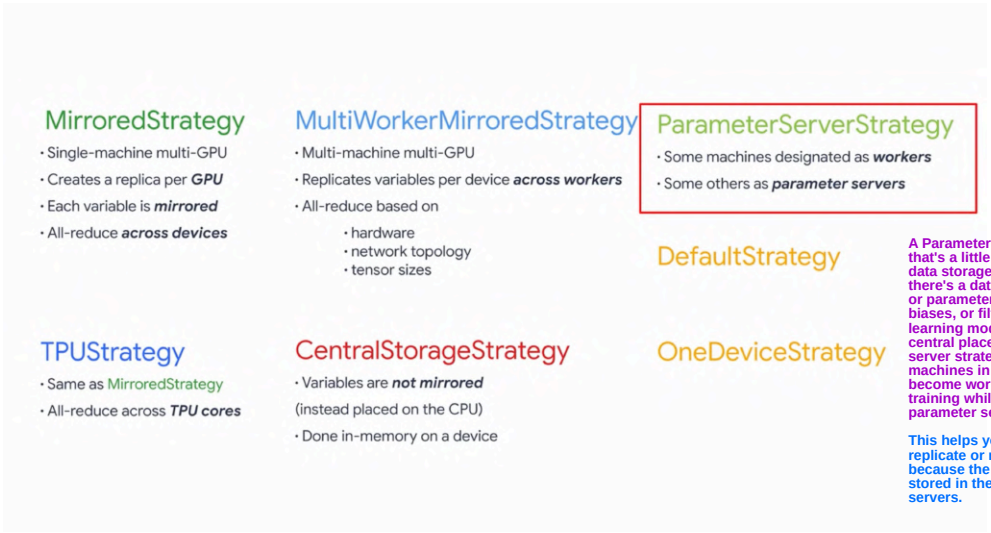
The TPU strategy is very similar to the mirrored one, except it uses TPU cores instead of GPUs. If you have a cluster of them available, you could use this strategy instead.



The MultiWorkerMirroredStrategy builds on mirrored strategy for scenarios where you have multiple machines on the network. These in turn can have varying amounts of GPUs. In order to manage this, it replicates and mirrors across each worker instead of each GPU device. Then it uses an all-reduce algorithm based on the hardware setup. This is a very specific strategy that's customized to your networks

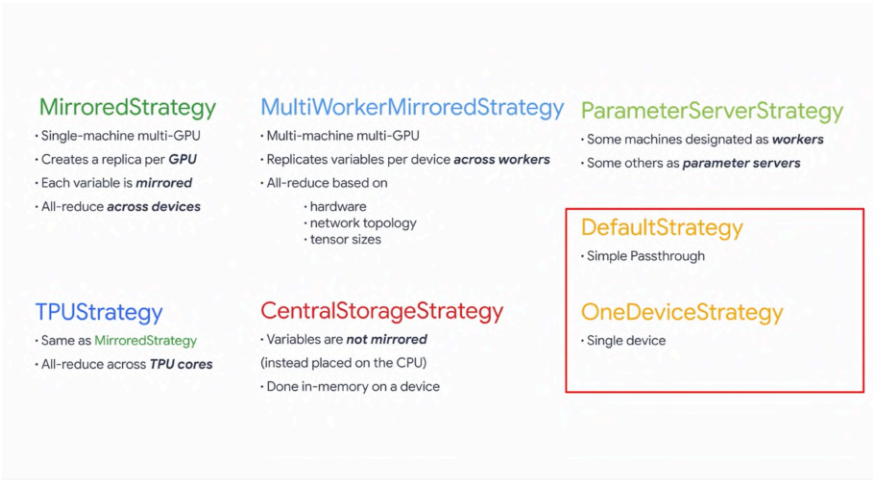


Central Storage Strategy is generally a single machine strategy. But in the case of multiple GPUs on the machine, instead of mirroring variables across the different GPUs, they're stored and processed by the CPU. If there's only one GPU, then the GPU is used since mirrored variables are not needed when there's just one device.



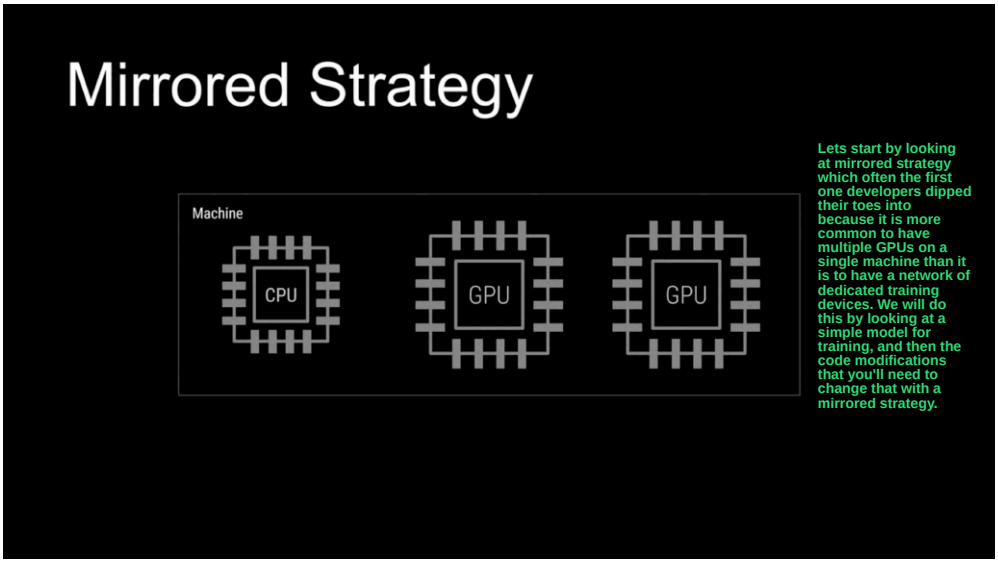
A Parameter server is a machine, that's a little like an independent data storage device. You can see there's a database with a variables or parameters such as weights, biases, or filters in your machine learning model can be stored in a central place. With the parameter server strategies, some of the machines in your network will become workers that perform the training while others are the parameter servers.

This helps you to avoid having to replicate or mirror your variables because the variables are all stored in the central parameter servers.



Finally, there's the default strategy and the OneDeviceStrategy types that are primarily used for prototyping your code on a single machine and you can then later use them on a distributed platform.

Default strategy is just effectively a pass through that allows you to create distributed strategy code without needing any of the supported backends. You don't need multiple GPUs or networks or machines or anything like that. One device strategy explicitly places all of the training parameters, all the data, and all the apps onto a single device. All will be funneled to that device.



Lets start by looking at mirrored strategy which often the first one developers dipped their toes into because it is more common to have multiple GPUs on a single machine than it is to have a network of dedicated training devices. We will do this by looking at a simple model for training, and then the code modifications that you'll need to change that with a mirrored strategy.

Mirrored Strategy

- Model declaration
- Data preprocessing

There's really 2 main places where you change eyour codee to establish a mirror strategy: model declaration and data preprocessing. The next few slides will show the before and after.

```
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, 3, activation='relu', input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(10)
])

model.compile(
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    optimizer=tf.keras.optimizers.Adam(),
    metrics=['accuracy'])
```

Here's typical keras code for setting up a model for training without using any kind of distribution strategy.

We will define the model as a sequential.

```
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, 3, activation='relu', input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(10)
])

model.compile(
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    optimizer=tf.keras.optimizers.Adam(),
    metrics=['accuracy'])
```

You then compile the model specifying a loss function and an optimizier. Then it's ready to train. To get it ready for training across multiple GPUs with a mirrored strategy, you'll have to make some very minor changes

```python
def scale(image, label):
     image = tf.cast(image, tf.float32)
     image /= 255
     return image, label

num_train_examples = info.splits['train'].num_examples
num_test_examples = info.splits['test'].num_examples

BUFFER_SIZE = 10000

BATCH_SIZE = 64

train_dataset = mnist_train.map(scale).cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
eval_dataset = mnist_test.map(scale).batch(BATCH_SIZE)
```

You'll also typically have code like this to preprare your data.

It groups the data up into batches of a particular size, 64 in this case

```python
def scale(image, label):
     image = tf.cast(image, tf.float32)
     image /= 255
     return image, label

num_train_examples = info.splits['train'].num_examples
num_test_examples = info.splits['test'].num_examples

BUFFER_SIZE = 10000

BATCH_SIZE = 64

train_dataset = mnist_train.map(scale).cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
eval_dataset = mnist_test.map(scale).batch(BATCH_SIZE)
```

It then allows you to create random shuffled batches of data that you'll train with. The data is just going to go into a single processor so it doesn't need to be split up. But when using a mirrored strategy with multiple GPUs for training, the data will need to be split up. So this code will neeed to be modified to do this data splitting.

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

print('Number of devices: {}'.format(strategy.num_replicas_in_sync))
```

This is how to modify the code to use a distributed strategy. The first step is to declare which strategy you wish to use, in this case we opt for mirrored stategy.

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

print('Number of devices: {}'.format(strategy.num_replicas_in_sync))
```

The strategy object contains properties that describe your infrastructure. So for example, you can get the number of replicas that are going to be sync for the strategy. If you're running it in colab, this would give you 1 because that's the number of GPUs available to you. If you have a multi-GPU infrastructure, you should see the number of available GPUs when you run this code.

```
def scale(image, label):
  image = tf.cast(image, tf.float32)
  image /= 255

  return image, label

num_train_examples = info.splits['train'].num_examples
num_test_examples = info.splits['test'].num_examples

BUFFER_SIZE = 10000

BATCH_SIZE_PER_REPLICA = 64
BATCH_SIZE = BATCH_SIZE_PER_REPLICA * strategy.num_replicas_in_sync

train_dataset =
mnist_train.map(scale).cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
eval_dataset = mnist_test.map(scale).batch(BATCH_SIZE)
```

For the data processing, we then need to make some minor changes to the code.

Instead of the batch size being 64, we'll have 64 per replica of data.

```
def scale(image, label):
  image = tf.cast(image, tf.float32)
  image /= 255

  return image, label

num_train_examples = info.splits['train'].num_examples
num_test_examples = info.splits['test'].num_examples

BUFFER_SIZE = 10000

BATCH_SIZE_PER_REPLICA = 64
BATCH_SIZE = BATCH_SIZE_PER_REPLICA * strategy.num_replicas_in_sync

train_dataset =
mnist_train.map(scale).cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
eval_dataset = mnist_test.map(scale).batch(BATCH_SIZE)
```

Then the batch size will be batch_size_per_replica times the number of replicas we want to keep in sync. So if we have 2 GPUs, we could make our batch size, set it to 128, then that why each GPU would be 64 per

```
with strategy.scope():
  model = tf.keras.Sequential([
      tf.keras.layers.Conv2D(32, 3, activation='relu', input_shape=(28, 28, 1)),
      tf.keras.layers.MaxPooling2D(),
      tf.keras.layers.Flatten(),
      tf.keras.layers.Dense(64, activation='relu'),
      tf.keras.layers.Dense(10)
  ])


model.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              optimizer=tf.keras.optimizers.Adam(),
              metrics=['accuracy'])
```

Then our model can effectively be unchanged. All we need to do is ensure it's within the scope of the strategy by putting all of the code within a `with strategy.scope()` call.

```
with strategy.scope():
  model = tf.keras.Sequential([
      tf.keras.layers.Conv2D(32, 3, activation='relu', input_shape=(28, 28, 1)),
      tf.keras.layers.MaxPooling2D(),
      tf.keras.layers.Flatten(),
      tf.keras.layers.Dense(64, activation='relu'),
      tf.keras.layers.Dense(10)
  ])


model.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              optimizer=tf.keras.optimizers.Adam(),
              metrics=['accuracy'])
```

Do note, that the model.compile() does not need to be in scope. Neither does model.fit(). Only the declaration.

```
Epoch 1/12
INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to
INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to
INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to
INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to
```

When it starts training with the strategy in colab, you should see something similar to this. Note that each line says `Reduce to` because all reduce is doing is synchronizing the parameters that are being trained on the different devices. The output also says `Replica` followed by a number and this is specifying which replicas are training.

Training across local GPUs

`tf.distribute.MirroredStrategy`

- Each variable in the model is mirrored across all replicas
- Variables are treated as `MirroredVariable`
- **Synchronization** done with NVIDIA NCCL

The mirrored strategy will create one replica for every GPU device and all the trainable parameters are copied across these replicas and thereby it treats them as MirroredVariables.

This means that after each forward pass in all of the replicas in the models, the variables are synced so that they are the same across all devices and all replicas of the model.

Variables created in this strategy are treated as MirroredVariables so the variable will be reflected across all replicas and the values will be kept in sync using an all reduce algorithm.

As only NVIDIA GPUs are presently supported, this algorithm will be NVIDIA's called NCCL. When using colab, you only have access to a single GPU. So it's hard to see how mirrored strategy would work fully.

```
# Create Datasets from the batches
train_dataset = tf.data.Dataset.from_tensor_slices((train_images, train_labels))
                    .shuffle(BUFFER_SIZE).batch(GLOBAL_BATCH_SIZE)

test_dataset = tf.data.Dataset.from_tensor_slices((test_images, test_labels))
                    .batch(GLOBAL_BATCH_SIZE)

# Create Distributed Datasets from the datasets
train_dist_dataset = strategy.experimental_distribute_dataset(train_dataset)
test_dist_dataset = strategy.experimental_distribute_dataset(test_dataset)
```

Let's start by looking at the data. Typically after loading your dataset you shuffle and batch the training datasets. We can also batch the test set. The code for that is here, and it's unchanged from what you would have seen in use previously. Notice the calls to the functions.shuffle.batch.

Previously you saw how TensorFlow implements distribution strategies. You've got an introduction to the mirrored strategy that can commonly be used across GPUs. You saw how to use this with a basic Keras model. While the mirrored strategy worked well, the implementation details were hidden behind the high level APIs, and you could only see it through the impact on training performance.

In this section, we'll go through how to create a model with Custom Training. And then within the training loop, you will better be able to explore how it splits the data training across GPUs, and you can see how the aggregation of lost data is then managed across them.

```
# Create Datasets from the batches
train_dataset = tf.data.Dataset.from_tensor_slices((train_images, train_labels))
                    .shuffle(BUFFER_SIZE).batch(GLOBAL_BATCH_SIZE)

test_dataset = tf.data.Dataset.from_tensor_slices((test_images, test_labels))
                    .batch(GLOBAL_BATCH_SIZE)

# Create Distributed Datasets from the datasets
train_dist_dataset = strategy.experimental_distribute_dataset(train_dataset)
test_dist_dataset = strategy.experimental_distribute_dataset(test_dataset)
```

What's new is this, and this is why we can convert the datasets into distributed ones. We do this by calling the function experimental distributed dataset, passing it the dataset that we want to distribute.

```
# Create Datasets from the batches
train_dataset = tf.data.Dataset.from_tensor_slices((train_images, train_labels))
                    .shuffle(BUFFER_SIZE).batch(GLOBAL_BATCH_SIZE)

test_dataset = tf.data.Dataset.from_tensor_slices((test_images, test_labels))
                    .batch(GLOBAL_BATCH_SIZE)

# Create Distributed Datasets from the datasets
train_dist_dataset = strategy.experimental_distribute_dataset(train_dataset)
test_dist_dataset = strategy.experimental_distribute_dataset(test_dataset)
```

This function is considered experimental at the moment, hence the function name includes the word experiment. The API name will likely change in the future, once it's no longer experimental, so watch out for that.

```
EPOCHS = 10
for epoch in range(EPOCHS):
  # Do Training
  total_loss = 0.0
  num_batches = 0
  for batch in train_dist_dataset:
    total_loss += distributed_train_step(batch)
    num_batches += 1
  train_loss = total_loss / num_batches
```

Next is the typical training loop. And here's an example of a custom training loop, but note two things.

First is that we'll use the distributed dataset for training, so we can read this back by batch.

```
EPOCHS = 10
for epoch in range(EPOCHS):
  # Do Training
  total_loss = 0.0
  num_batches = 0
  for batch in train_dist_dataset:
    total_loss += distributed_train_step(batch)
    num_batches += 1
  train_loss = total_loss / num_batches
```

Typically with a custom model, you would then pass this batch to your training step to calculate the loss and the gradients, and then use the gradients to update the model.

In this case, you're going to write a function called distributed training stamp, which will see in detail soon. This will operate in a very similar way, but our distributed training step, will receive the losses from each of the replicas and then reduce them. Then it will return that value to here

```
EPOCHS = 10
for epoch in range(EPOCHS):
  # Do Training
  total_loss = 0.0
  num_batches = 0
  for batch in train_dist_dataset:
    total_loss += distributed_train_step(batch)
    num_batches += 1
  train_loss = total_loss / num_batches
```

Where it gets added to a cumulative total loss

```
EPOCHS = 10
for epoch in range(EPOCHS):
  # Do Training
  total_loss = 0.0
  num_batches = 0
  for batch in train_dist_dataset:
    total_loss += distributed_train_step(batch)
    num_batches += 1
  train_loss = total_loss / num_batches
```

Which in turn is then averaged out, to give us the training loss for this epoch by dividing the total loss by the number of batches.

```
@tf.function
def distributed_train_step(dataset_inputs):
  per_replica_losses = strategy.run(train_step, args=(dataset_inputs,))
  return strategy.reduce(tf.distribute.ReduceOp.SUM,
                         per_replica_losses, axis=None)
```

In our distributed training scenario, we have the distributed training stamp that's called within the loop. Let's look at that in detail.

```
@tf.function
def distributed_train_step(dataset_inputs):
  per_replica_losses = strategy.run(train_step, args=(dataset_inputs,))
  return strategy.reduce(tf.distribute.ReduceOp.SUM,
                         per_replica_losses, axis=None)
```

First of all, you might be surprised at how little code is in here. You might have been expecting the losses to be calculated, gradients to be made, steps down the gradient slope to optimize parameters to be done and all that kind of stuff. There's none of that in here. Why? Because this function, distributed training step will call another function which is named training step here, and that does all of those calculations.

We'll see that in a moment. You can see that it's unchanged for distributed training. The difference is that the distributed training step is called in the training loop, and then it will in turn called unusual training step that you can see here.

```
@tf.function
def distributed_train_step(dataset_inputs):
  per_replica_losses = strategy.run(train_step, args=(dataset_inputs,))
  return strategy.reduce(tf.distribute.ReduceOp.SUM,
                         per_replica_losses, axis=None)
```

Notice that it calls the training step with the strategy.run, so the distribution strategy class will handle the distributed training for you. You pass it to the train _step function, which will then execute on each replica. You also give strategy.run the dataset inputs for the epoch, which because they're coming from a distributed dataset, will be split up for you according to the number of replicas.

```
@tf.function
def distributed_train_step(dataset_inputs):
  per_replica_losses = strategy.run(train_step, args=(dataset_inputs,))
  return strategy.reduce(tf.distribute.ReduceOp.SUM,
                         per_replica_losses, axis=None)
```

This will return a pair replica losses objects, and this contains the losses per replica. This will simply be an array of losses, with each being the loss calculated on that replica.

```python
@tf.function
def distributed_train_step(dataset_inputs):
  per_replica_losses = strategy.run(train_step, args=(dataset_inputs,))
  return strategy.reduce(tf.distribute.ReduceOp.SUM,
                         per_replica_losses, axis=None)
```

The strategy will then give us a reduce operation method that we can use to aggregate the losses. In this case, we'll simply do a sum of the losses and return back.

```python
def train_step(inputs):
  images, labels = inputs
  with tf.GradientTape() as tape:
    predictions = model(images, training=True)
    loss = compute_loss(labels, predictions)

  gradients = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(gradients, model.trainable_variables))

  train_accuracy.update_state(labels, predictions)
  return loss
```

The training step is unchanged from what you might be used to. Remember that this is executed in parallel across all devices, because we're using a mirrored strategy. But there's no parallel code here. All of that was handled in the distributed training step that you saw earlier on, on which we'll call this train_step function

```python
def train_step(inputs):
  images, labels = inputs
  with tf.GradientTape() as tape:
    predictions = model(images, training=True)
    loss = compute_loss(labels, predictions)

  gradients = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(gradients, model.trainable_variables))

  train_accuracy.update_state(labels, predictions)
  return loss
```

Giving it the appropriate inputs out of which the images and the labels are extracted

```python
def train_step(inputs):
  images, labels = inputs
  with tf.GradientTape() as tape:
    predictions = model(images, training=True)
    loss = compute_loss(labels, predictions)

  gradients = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(gradients, model.trainable_variables))

  train_accuracy.update_state(labels, predictions)
  return loss
```

And then using a gradient tape, we can get our predictions, and use them to calculate the loss against the ground truth

```python
def train_step(inputs):
  images, labels = inputs
  with tf.GradientTape() as tape:
    predictions = model(images, training=True)
    loss = compute_loss(labels, predictions)

  gradients = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(gradients, model.trainable_variables))

  train_accuracy.update_state(labels, predictions)
  return loss
```

Calculate the gradient for the variables against the loss function, and use this to optimize the model,

```python
def train_step(inputs):
  images, labels = inputs
  with tf.GradientTape() as tape:
    predictions = model(images, training=True)
    loss = compute_loss(labels, predictions)

  gradients = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(gradients, model.trainable_variables))

  train_accuracy.update_state(labels, predictions)
  return loss
```

Update the training accuracy state, so that we know how accurate the model currently is...

```
def train_step(inputs):
  images, labels = inputs
  with tf.GradientTape() as tape:
    predictions = model(images, training=True)
    loss = compute_loss(labels, predictions)

  gradients = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(gradients, model.trainable_variables))

  train_accuracy.update_state(labels, predictions)
  return loss
```

Then return the loss.

Don't forget that if this is in a distributed environment, say across two processors, then the distributed training function will call this function twice, once for each processor, get the loss from each, and then reduce that.

```
@tf.function
def distributed_train_step(dataset_inputs):
  per_replica_losses = strategy.run(train_step, args=(dataset_inputs,))
  return strategy.reduce(tf.distribute.ReduceOp.SUM,
                         per_replica_losses, axis=None)
```

Which is what the distributed training step that you saw earlier on. That's pretty much everything you need to change in your code to get it to work across multiple processors. In this case these multiple GPUs. You can get the code on GitHub. But do note that Colab only offers a single GPU. It will be hard for you to see the parallelism.

```
# Detect hardware
try:
    tpu_address = 'grpc://' + os.environ['COLAB_TPU_ADDR']
    tpu = tf.distribute.cluster_resolver.TPUClusterResolver(tpu_address)
    tf.config.experimental_connect_to_cluster(tpu)
    tf.tpu.experimental.initialize_tpu_system(tpu)
    strategy = tf.distribute.experimental.TPUStrategy(tpu)
    print('Running on TPU ', tpu.cluster_spec().as_dict()['worker'])
    print("Number of accelerators: ", strategy.num_replicas_in_sync)

except ValueError:
    print('TPU failed to initialize.')
```

Previously you looked at using merit strategy with GPUs and you saw how you could access the single GPU that's available in Colab. But also if you had multiple GPUs, you can see how it works across them. In this section, I'll take you through using TPUs and Colab and distributed training across them using TPU strategy. Much of the pattern that you used already will work again.

You'll create a custom training loop, and within that you'll have a distributed training stamp whose purpose it is to run the distributed training using the strategy and then reduce the results across all parallel processes. You'll use the TPUs that are available in Colab at this step.

Before we get to training, I just want to show you some code that gets you started with using TPUs and shows you how many cores you'll be able to parallelize for your training.

Before that though, you might wonder, why would I want to use a TPU? The bottom line for this is ultimately that TPUs are chips that are optimized for machine learning. If you have large inexpensive models that can be orders of magnitude cheaper to use and training. Colab supports TPUs so that you can experiment with them. We'll see how to use TPU strategy to make the most out of using them. If you're using Colab, the first thing that you'll want to ensure is that you're using TPUs on the backend. To do this, go to the runtime menu and select Change runtime type. You'll see this dialogue and make sure that you select TPU to be your hardware accelerator. So at the beginning of your Colab, it's good to start with this code to get a handle on exactly what TPU resources are available to you for training. Let's look at this little by little.

The address of the TPU server is available as an environment variable called Colab TPU address, which you can access using os.environ. This will give you the IP address and a port where you can access a TPU over remote procedure call.

```
# Detect hardware
try:
    tpu_address = 'grpc://' + os.environ['COLAB_TPU_ADDR']
    tpu = tf.distribute.cluster_resolver.TPUClusterResolver(tpu_address)
    tf.config.experimental_connect_to_cluster(tpu)
    tf.tpu.experimental.initialize_tpu_system(tpu)
    strategy = tf.distribute.experimental.TPUStrategy(tpu)
    print('Running on TPU ', tpu.cluster_spec().as_dict()['worker'])
    print("Number of accelerators: ", strategy.num_replicas_in_sync)

except ValueError:
    print('TPU failed to initialize.')
```

The TPU address variable is set up to be a GRPC colon slash slash to that IP address and that ports. GRPC stands folk Google Remote Procedure Call.

```
# Detect hardware
try:
    tpu_address = 'grpc://' + os.environ['COLAB_TPU_ADDR']
    tpu = tf.distribute.cluster_resolver.TPUClusterResolver(tpu_address)
    tf.config.experimental_connect_to_cluster(tpu)
    tf.tpu.experimental.initialize_tpu_system(tpu)
    strategy = tf.distribute.experimental.TPUStrategy(tpu)
    print('Running on TPU ', tpu.cluster_spec().as_dict()['worker'])
    print("Number of accelerators: ", strategy.num_replicas_in_sync)

except ValueError:
    print('TPU failed to initialize.')
```

TPUs are available in Google Cloud as Cloud TPU workers, which are different from the process that runs Colab. Thus to get fine-grained control of the TPU, you need to connect to the cluster and initialize that TPU. To find the cluster, you use cluster resolver and the tf.distribute namespace, and you'll pass it the address of the TPU.

```
# Detect hardware
try:
    tpu_address = 'grpc://' + os.environ['COLAB_TPU_ADDR']
    tpu = tf.distribute.cluster_resolver.TPUClusterResolver(tpu_address)
    tf.config.experimental_connect_to_cluster(tpu)
    tf.tpu.experimental.initialize_tpu_system(tpu)
    strategy = tf.distribute.experimental.TPUStrategy(tpu)
    print('Running on TPU ', tpu.cluster_spec().as_dict()['worker'])
    print("Number of accelerators: ", strategy.num_replicas_in_sync)

except ValueError:
    print('TPU failed to initialize.')
```

You can then connect to and initialize the TPU by first connect into the cluster containing the TPU and then calling initialized TPU system on it.

Note that this code is currently considered to be experimental, as you can see in its name. Over time as it matures, these API names may change. So do keep a close eye on them.

```
# Detect hardware
try:
    tpu_address = 'grpc://' + os.environ['COLAB_TPU_ADDR']
    tpu = tf.distribute.cluster_resolver.TPUClusterResolver(tpu_address)
    tf.config.experimental_connect_to_cluster(tpu)
    tf.tpu.experimental.initialize_tpu_system(tpu)
    strategy = tf.distribute.experimental.TPUStrategy(tpu)
    print('Running on TPU ', tpu.cluster_spec().as_dict()['worker'])
    print("Number of accelerators: ", strategy.num_replicas_in_sync)

except ValueError:
    print('TPU failed to initialize.')
```

Having now initialized your TPUs, you can create your TPU strategy. All training is done within this strategy and it will be distributed across replicas available within the TPU.

```
# Detect hardware
try:
    tpu_address = 'grpc://' + os.environ['COLAB_TPU_ADDR']
    tpu = tf.distribute.cluster_resolver.TPUClusterResolver(tpu_address)
    tf.config.experimental_connect_to_cluster(tpu)
    tf.tpu.experimental.initialize_tpu_system(tpu)
    strategy = tf.distribute.experimental.TPUStrategy(tpu)
    print('Running on TPU ', tpu.cluster_spec().as_dict()['worker'])
    print("Number of accelerators: ", strategy.num_replicas_in_sync)

except ValueError:
    print('TPU failed to initialize.')
```

If you want to inspect them, you can do so by querying the strategy objects using tpu.cluster_spec and printing out the number of replicas in sync in this strategy by using Strategy.num_replicas_in_sink

```
INFO:tensorflow:*** Available Device:
_DeviceAttributes(/job:worker/replica:0/task:0/device:TPU_SYSTEM:0,
TPU_SYSTEM, 0, 0)

INFO:tensorflow:*** Available Device:
_DeviceAttributes(/job:worker/replica:0/task:0/device:TPU_SYSTEM:0,
TPU_SYSTEM, 0, 0)

INFO:tensorflow:*** Available Device:
_DeviceAttributes(/job:worker/replica:0/task:0/device:XLA_CPU:0,
XLA_CPU, 0, 0)

INFO:tensorflow:*** Available Device:
_DeviceAttributes(/job:worker/replica:0/task:0/device:XLA_CPU:0,
XLA_CPU, 0, 0)

Running on TPU  ['10.109.132.10:8470']
Number of accelerators:  8
```

Run this code and you'll see a lot of status updates. But at the bottom, you'll see the results of the print statements from the previous slide. Where as one as the address and port of the TPU, you'll also see the number of accelerators that this TPU uses, which in this case it's eight. See your training will be parallelized across all eight.

Training with TPU strategy

- Use a custom training loop
- Call the distributed training function within the loop
  - Use strategy.run to call your usual training function across all replicas
  - Results will be in per-replica-losses structure
  - Use strategy.reduce to reduce losses
- Call the distributed testing function within the loop
  - Use strategy.run to call your usual testing function across all replicas

Once you're set up with your CPU, then the code for training with TPU strategy is very similar to using merit strategy that you used earlier for GPUs.

Training with TPU strategy

- Use a custom training loop
- Call the distributed training function within the loop
  - Use strategy.run to call your usual training function across all replicas
  - Results will be in per-replica-losses structure
  - Use strategy.reduce to reduce losses
- Call the distributed testing function within the loop
  - Use strategy.run to call your usual testing function across all replicas

You'll use a custom training globe instead of relying on Keras as model.fit. This lets you granularly manage how the training works. In particular, measuring the loss and reducing it across the replicas.

Training with TPU strategy

- Use a custom training loop
- Call the distributed training function within the loop
  - Use strategy.run to call your usual training function across all replicas
  - Results will be in per-replica-losses structure
  - Use strategy.reduce to reduce losses
- Call the distributed testing function within the loop
  - Use strategy.run to call your usual testing function across all replicas

In a training loop, you'll typically have a training function that measures loss, plots the current values against that loss, measures the gradients of the loss and attempts to minimize it. That ball rolling down the hill of gradient descent that we discussed last week. Instead of calling this function directly, we'll call it distributed training function first.

Training with TPU strategy

- Use a custom training loop
- Call the distributed training function within the loop
  - Use strategy.run to call your usual training function across all replicas
  - Results will be in per-replica-losses structure
  - Use strategy.reduce to reduce losses
- Call the distributed testing function within the loop
  - Use strategy.run to call your usual testing function across all replicas

The distributed training function will then call your usual training function using the strategy.run. This effectively parallelizes it so that the loss optimization process will happen across all replicas.

Training with TPU strategy

- Use a custom training loop
- Call the distributed training function within the loop
  - Use strategy.run to call your usual training function across all replicas
  - Results will be in per-replica-losses structure
  - Use strategy.reduce to reduce losses
- Call the distributed testing function within the loop
  - Use strategy.run to call your usual testing function across all replicas

You get back to a structure in per-replica-losses. Earlier, we saw that the TPU and Colab and eight cores, though yours might differ. So this structure will have the losses for all eight.

Training with TPU strategy

- Use a custom training loop
- Call the distributed training function within the loop
  - Use strategy.run to call your usual training function across all replicas
  - Results will be in per-replica-losses structure
  - Use strategy.reduce to reduce losses
- Call the distributed testing function within the loop
  - Use strategy.run to call your usual testing function across all replicas

Then to reduce those losses across the infrastructure, you can use Strategy.reduce.

Don't confuse reduce with Minimize. In this context, reduce effectively means merge all of the losses into a single one for measurement and reporting.

Training with TPU strategy

We can do something similar for testing, but in this case, we don't need to report on the loss. We can just create a distributed testing function.

- Use a custom training loop
- Call the distributed training function within the loop
  - Use strategy.run to call your usual training function across all replicas
  - Results will be in per-replica-losses structure
  - Use strategy.reduce to reduce losses
- Call the distributed testing function within the loop
  - Use strategy.run to call your usual testing function across all replicas

Training with TPU strategy

We'll have that called our testing function that updates us on the accuracy of the model using the test set. Let's take a look at that code next.

- Use a custom training loop
- Call the distributed training function within the loop
  - Use strategy.run to call your usual training function across all replicas
  - Results will be in per-replica-losses structure
  - Use strategy.reduce to reduce losses
- Call the distributed testing function within the loop
  - Use strategy.run to call your usual testing function across all replicas

```
@tf.function
def distributed_train_step(dataset_inputs):
  per_replica_losses = strategy.run(train_step,args=(dataset_inputs,))
  return strategy.reduce(tf.distribute.ReduceOp.SUM, per_replica_losses, axis=None)


def train_step(inputs):
  images, labels = inputs
  with tf.GradientTape() as tape:
    predictions = model(images)
    loss = compute_loss(labels, predictions)
  gradients = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(gradients, model.trainable_variables))
  train_accuracy.update_state(labels, predictions)
  return loss
```

Here's the training Functions. First is the distributed training staff whose job it is to perform the training across all of the replicas in parallel.

```
@tf.function
def distributed_train_step(dataset_inputs):
  per_replica_losses = strategy.run(train_step,args=(dataset_inputs,))
  return strategy.reduce(tf.distribute.ReduceOp.SUM, per_replica_losses, axis=None)


def train_step(inputs):
  images, labels = inputs
  with tf.GradientTape() as tape:
    predictions = model(images)
    loss = compute_loss(labels, predictions)
  gradients = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(gradients, model.trainable_variables))
  train_accuracy.update_state(labels, predictions)
  return loss
```

It does this by calling the train_step function, which is at the bottom of the slide. We'll see that shortly within a strategy.run_scope. The strategy then manages splitting the data across all the cores, making replicas, performing the training on these, and getting the loss Report.

```
@tf.function
def distributed_train_step(dataset_inputs):
  per_replica_losses = strategy.run(train_step,args=(dataset_inputs,))
  return strategy.reduce(tf.distribute.ReduceOp.SUM, per_replica_losses, axis=None)


def train_step(inputs):
  images, labels = inputs
  with tf.GradientTape() as tape:
    predictions = model(images)
    loss = compute_loss(labels, predictions)
  gradients = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(gradients, model.trainable_variables))
  train_accuracy.update_state(labels, predictions)
  return loss
```

It will reduce the losses, in this case simply by summing them up. The values can be averaged out later.

```
@tf.function
def distributed_train_step(dataset_inputs):
  per_replica_losses = strategy.run(train_step,args=(dataset_inputs,))
  return strategy.reduce(tf.distribute.ReduceOp.SUM, per_replica_losses, axis=None)


def train_step(inputs):
  images, labels = inputs
  with tf.GradientTape() as tape:
    predictions = model(images)
    loss = compute_loss(labels, predictions)
  gradients = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(gradients, model.trainable_variables))
  train_accuracy.update_state(labels, predictions)
  return loss
```

The training step predicts the value of our data and compares with the ground truth values to get the loss.

```
@tf.function
def distributed_train_step(dataset_inputs):
  per_replica_losses = strategy.run(train_step,args=(dataset_inputs,))
  return strategy.reduce(tf.distribute.ReduceOp.SUM, per_replica_losses, axis=None)


def train_step(inputs):
  images, labels = inputs
  with tf.GradientTape() as tape:
    predictions = model(images)
    loss = compute_loss(labels, predictions)
  gradients = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(gradients, model.trainable_variables))
  train_accuracy.update_state(labels, predictions)
  return loss
```

It then gets the gradients of the values compared to that loss, then uses the optimizer to apply those gradients to update the models trainable variables, and learn to minimize the loss. It then updates the training accuracy tracker with the results returning the loss.

The rest of the code for the TPU strategy is exactly the same as you would expect for distributed training. We can take a look at that in a Colab in a screencast next, and then he can try it out for yourself.

Training on a single device

tf.distribute.OneDeviceStrategy

Input data is distributed

While we've mostly looked at using merit strategy and TPU strategy in this course. That was primarily because they cover the most easily available hardware and can be done simply using Colab. There are some other strategies that will be good to at least get a little bit of an understanding of. You won't get hands-on practice with these here. But we'll take a look at the overall architecture and some simple code.

The first strategy is one device strategy. This particular strategy is used when you deliberately wish to perform training on one specific device of your choice. While it's just a single device, the input data can still be treated as if it were distributed. You can use it to test your code before you then distribute it across multiple devices.

To use it, it's as simple as knowing the name of the device that you want to use. For example, gpu colon zero and then declaring your strategy. The rest of the code will work in exactly the same way as what you've previously looked at particularly if you use a Custom Training.

```
strategy = tf.distribute.OneDeviceStrategy(device="/gpu:0")
```

Training across many machines

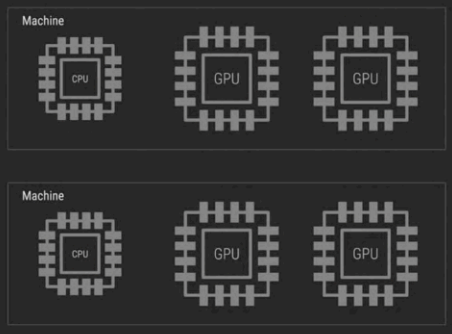
`tf.distribute.experimental.MultiWorkerMirroredStrategy`

- Done across multiple workers, each with multiple GPUs
- Variables are replicated on each device across workers
- Fault tolerance with tf.keras.callbacks.ModelCheckpoint
- **Synchronization** done with CollectiveOps

Next is the MultiWorkerMirroredStrategy. Early you saw how mirrored strategy could be used where training was done across multiple GPUs in a single machine.

MultiWorkerMirroredStrategy is designed when there are multiple GPUs on multiple machines and each machine can have a different number of GPUs on them. It's a complicated environment so things like fault tolerance is vital and this can be achieved using checkpoints. If an issue is hidden training, then he can just revert to the most recent checkpoint.

Synchronization and reduction is also more complex and it is performed using something called CollectiveOps. These APIs are responsible for handling the complexities of keeping everything in sync between all of the devices so you don't have to worry about it.



For example, here's a relatively simple scenario where I have two machines and each has a CPU and two GPUs. If I wanted to distribute training across all of these, I can use a MultiWorkerMirroredStrategy.

```
multiworker_strategy = tf.distribute.experimental.MultiWorkerMirroredStrategy()
```

Multi-worker training

- Run **workers** in a cluster
- **Tasks** (training/input pipelines)
- **Roles** (chief, worker, ps, evaluator)
- Configuring the cluster (next..)

It's called a multi worker strategy. Let's talk about workers for a moment. A worker is a piece of code that runs a task such as managing training or input pipelines. A worker is given a particular role such as a chief or an evaluator and the worker role determines what tasks that it should and it can perform. I won't go into all the details of the roles here. But for example, the chief worker might have a little bit more responsibility than the others and perform things like saving checkpoints.

Cluster specification

The cluster specification looks like this. It's a JSON file where you define your clusters and your workers based on their host address and their port. Within it, you can define the task that the worker will perform.

In this case, we have worker 0 shown and by default, worker 0 is the chief worker. Now this is just illustrative, so if you want to learn more about how to use MultiWorkerMirroredStrategy, please check out the URL at the bottom of the slide.

```
os.environ["TF_CONFIG"] = json.dumps({
    "cluster": {
        "worker": ["host1:port", "host2:port", ...],
    },
   "task": {"type": "worker", "index": 0}
})
```

https://www.tensorflow.org/tutorials/distribute/multi_worker_with_keras

Other strategies

CentralStorageStrategy

Variables - not mirrored, but placed on CPU

Computation - replicated across local GPUs

ParameterServerStrategy

Some machines are designated as workers

... some as parameter servers

Variables - placed on one parameter server (ps)

Computation - replicated across GPUs of all the workers

There are two other strategies that are worth a quick mention. The first is CentralStorageStrategy where the training variables are not mirrored across your clusters and instead they're stored in memory accessed by the CPU.

The computation takes place on the GPUs with the idea of saving the more expensive GPU memory for computation only where the variable storage can be in this cheaper memory that's accessed by the CPU.

Other strategies

CentralStorageStrategy

Variables - not mirrored, but placed on CPU

Computation - replicated across local GPUs

ParameterServerStrategy

Some machines are designated as workers

... some as parameter servers

Variables - placed on one parameter server (ps)

Computation - replicated across GPUs of all the workers

The second called ParameterServerStrategy brings the best of the central Storage strategy and the MultiWorkerMirroredStrategy together and that the computation is across all of the GPUs in your system. But the variables are stored in a distributed database called a parameter server.

Multi-worker training with Keras

- https://www.tensorflow.org/tutorials/distribute/multi_worker_with_keras