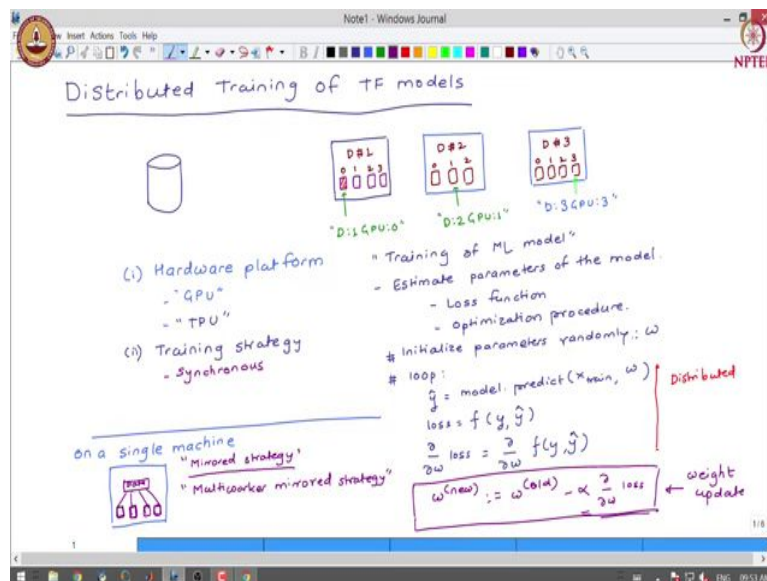


Practical Machine Learning with Tensorflow
Dr. Ashish Tendulkar
Department of Computer Science and Engineering
Indian Institute of Technology, Bombay

Lecture – 38
TensorFlow Distributed Training

Welcome back, to the next module of our course Practical Machine Learning with TensorFlow. This is the last module of the course where we will discuss about scaling strategies for TensorFlow models.

(Refer Slide Time: 00:27)



We will cover distributed training of TensorFlow models in this session. So, let us first understand why we need distributed training of TensorFlow models. Often we have models which are complex, which have very large number of parameters and also we want to train these models on a very large datasets. So far in this course we have written models that usually get trained on a single machine.

In order to do distributed training there are certain changes that we need to do in the TensorFlow code, the distributed strategy is designed in such a way that there are minimal code changes. So, when you actually train TensorFlow model in a distributed manner, you

will see that there are not many changes that are required in the code that we already wrote. Let us understand why we want to do distributed training of TensorFlow models.

So, we have a situation where you have a very large dataset and we have a very complex model to train and let us say we have a machine with multiple GPUs. If we do not do distributed training of TensorFlow model what will happen is; we will not be using all the resources that we have. The model usually get trained on only one of the GPUs.

So, here is a typical situation where we have resources, but we are not using them. So, if you if you want to use all these resources, we have to specify how the training gets distributed across these GPUs. There could be another situation where we might have a cluster of machines and each machine has different number of GPUs.

In the previous module we already studied how to identify each of the GPUs. Each GPU has a device ID and GPU ID.

Now, we will have to see how we can use this infrastructure that we have for performing distributed training. So, we might have a bunch of GPUs and we might have TPUs; TPUs are specialized hardware developed by Google for training large scale machine learning models. TPUs are very similar to GPUs except for certain parts that are not required for performing high end computations.

TPUs are generally an order of magnitude faster than GPUs. So, you might have GPU or TPU clusters and good news here is that we can access GPUs and TPUs from Google Colab; we can also access GPU and TPU machines in large number through cloud providers like Google cloud. So, hardware platform is the first aspect. Second aspect is the training strategy. So, before getting into training strategy let us try to understand what will happen in each of the GPUs.

So, we have a very large amount of data and we have model for which you want to perform parameter estimation. So, the problem that we are trying to solve here is training of machine learning model. By training we mean that you want to estimate parameters of the model. You may recall from our earlier discussion that the parameter estimation involves the following things; one is the loss function and the second is optimization procedure.

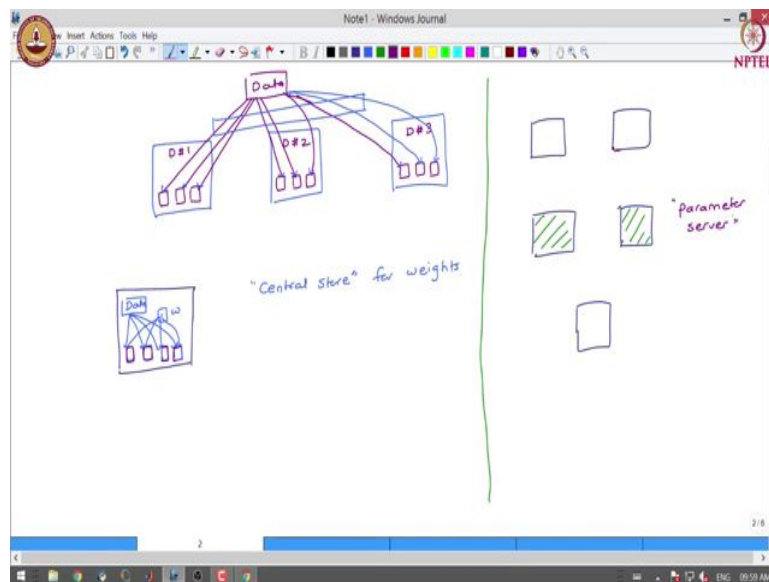
So, what we do is we initialize parameters, randomly or through some intelligent initialization strategy; after initializing the parameters we run a loop and inside the loop what we do is, we first find the predicted value. So, we use `model.predict` for the training examples with the weight vector that we have initialized here and then we calculate loss as some function of actual value and a predicted value.

And, we know in case of regression we use mean squared error as a loss function. Whereas in case of classification we use cross entropy loss as a loss function. And, once we find out the loss function we calculate the gradient of the loss function with respect to the weight vector. So, we will perform these 3 steps in a distributed manner. So, these 3 steps are distributed and finally, what we do is we update the parameter value based on the gradient and we use some learning rate so, this is the parameter update. So, what we will do is whatever data that we get we distribute that data across the available GPUs.

So, let us try to understand how we can do this on a single machine. Let us say we are doing this distributed training on a single machine having multiple GPUs. So, first we distribute the batch of data to all the GPUs and the model graph is already present is also copied to the GPUs. And we calculate the gradient of the loss function in each of the GPUs and then what we do is, we use an algorithm to collect all the gradients and perform the weight update. So, we use all reduce strategy to do the weight update and the updated weights are made available to each of the GPUs.

So, here all the GPUs are working synchronously. This is called as synchronous training strategy. And here we are mirroring data on each of the GPUs and hence this is called as mirrored strategy. So, if you generalize this to multiple machines exactly the same strategy it is called as multi worker mirrored strategy. So, in multi worker mirrored strategy what we do is, let us say these are different machines that we have.

(Refer Slide Time: 12:31)



So, we distribute the data across multiple GPUs, the variables and the model graph is replicated across GPUs and each of the GPU performs the computation of gradient and this gradient is updated through some multi worker all radio strategy. And the weight update is performed and those weights are again copied back to individual GPU in the cluster.

TensorFlow handles all the complexity of the communication as well as failure of the nodes internally. So, programmers do not have to worry about any of the aspect of the distributed computation if we use the TensorFlow library for distributed training. So, this is multi worker mirrored strategy. There can also be a single machine strategy which uses, which uses central store; central store for parameters. In this case, there is a central store that is holding all the weights and these are our GPUs.

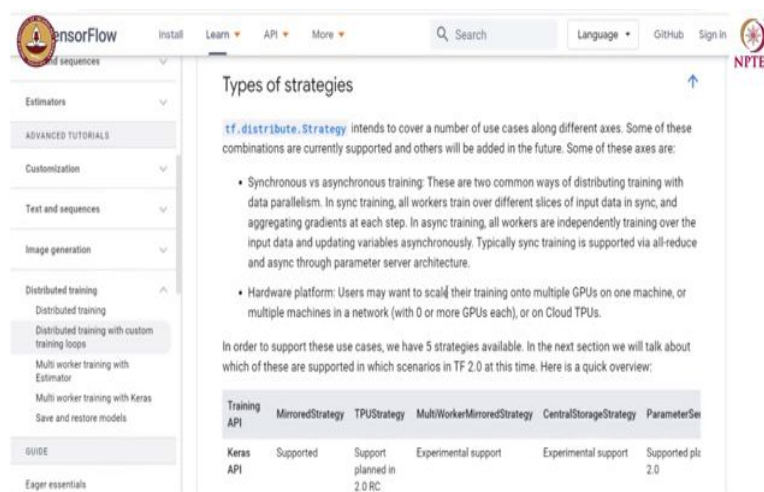
So, what we will do here is, we take the data we replicate it across GPUs; GPUs perform the gradient calculation and in order to do gradient calculation they read the values of the weight from the central store. And then gradient is combined from all the devices through all through all radio strategy and the update is made in the central store.

And, then the values from the central store are read in the next epoch by each of the GPUs this is called as centralized strategy; this is also synchronous strategy. We have one more strategy that involves multiple workers and some of the workers will behave as masters.

Some of the workers are used to keep track of parameters of the model and these machines are called as parameter servers.

So, one set of parameters is kept on one parameter server, and all the machines perform the gradient calculation and the parameters are updated on the respective server this is called as parameter server strategy. And, TPU strategy is also a mirror strategy where instead of GPUs we can think of replacing GPUs by TPUs.

(Refer Slide Time: 18:15)



Types of strategies

`tf.distribute.Strategy` intends to cover a number of use cases along different axes. Some of these combinations are currently supported and others will be added in the future. Some of these axes are:

- **Synchronous vs asynchronous training:** These are two common ways of distributing training with data parallelism. In sync training, all workers train over different slices of input data in sync, and aggregating gradients at each step. In async training, all workers are independently training over the input data and updating variables asynchronously. Typically sync training is supported via all-reduce and async through parameter server architecture.
- **Hardware platform:** Users may want to scale their training onto multiple GPUs on one machine, or multiple machines in a network (with 0 or more GPUs each), or on Cloud TPUs.

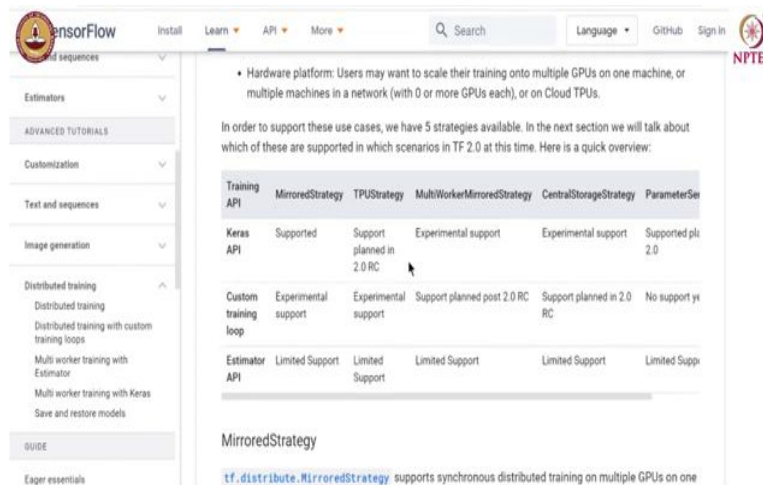
In order to support these use cases, we have 5 strategies available. In the next section we will talk about which of these are supported in which scenarios in TF 2.0 at this time. Here is a quick overview:

Training API	MirroredStrategy	TPUStrategy	MultiWorkerMirroredStrategy	CentralStorageStrategy	ParameterServerStrategy
Keras	Supported	Support planned in 2.0 RC	Experimental support	Experimental support	Supported since 2.0
API					

So, let us summarize this. There is a synchronous versus asynchronous training. In synchronous training; all workers train over different slices of input data in a synchronous manner and aggregate gradients at each step.

In asynchronous training all workers are independently training over the input data and updating variables asynchronously. Typically synchronous training is supported via all-reduce and async training is supported via parameter server architecture.

(Refer Slide Time: 19:05)



The screenshot shows the TensorFlow website's 'Distributed training' section. It features a table summarizing the support for five distribution strategies across different TensorFlow APIs. The strategies are MirroredStrategy, TPUStrategy, MultiWorkerMirroredStrategy, CentralStorageStrategy, and ParameterServerStrategy. The APIs listed are Keras API, Custom training loop, and Estimator API. The table indicates that Keras API supports MirroredStrategy, while TPUStrategy support is planned for the 2.0 release candidate. MultiWorkerMirroredStrategy, CentralStorageStrategy, and ParameterServerStrategy are listed as experimental or planned for future versions.

Training API	MirroredStrategy	TPUStrategy	MultiWorkerMirroredStrategy	CentralStorageStrategy	ParameterServerStrategy
Keras API	Supported	Support planned in 2.0 RC	Experimental support	Experimental support	Supported in 2.0
Custom training loop	Experimental support	Experimental support	Support planned post 2.0 RC	Support planned in 2.0 RC	No support yet
Estimator API	Limited Support	Limited Support	Limited Support	Limited Support	Limited Support

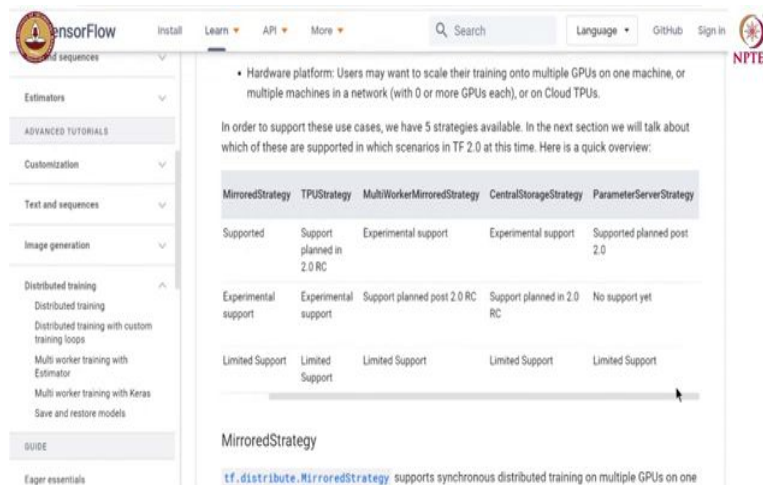
MirroredStrategy
`tf.distribute.MirroredStrategy` supports synchronous distributed training on multiple GPUs on one

So, in all we have 5 strategies - mirrored strategy, TPU strategy, multi worker mirrored strategy, central storage strategy and parameter server strategy.

And, let us look at what kind of strategies are supported in TensorFlow 2.0 through different APIs. We have keras API, we can write our custom training loop or we can use estimator API.

So, keras API supports mirrored strategy, the TPU strategy is planned in the release candidate of 2.0, multi worker strategy has got experimental support, central strategy has also got experimental support.

(Refer Slide Time: 20:05)



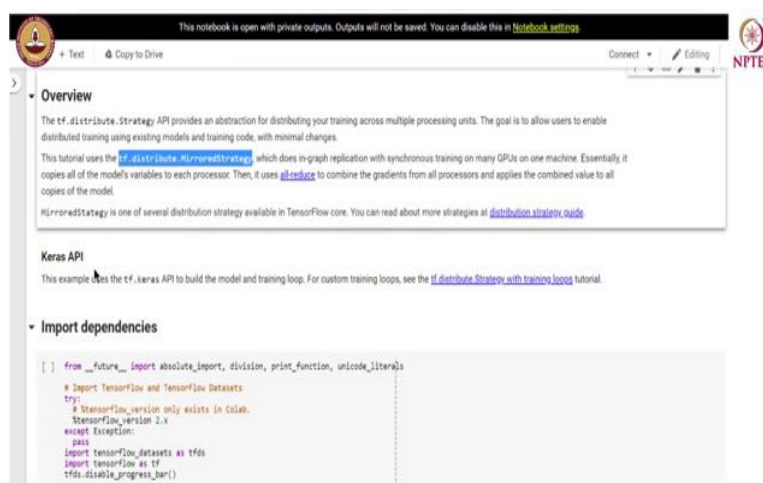
The screenshot shows the TensorFlow website's 'Learn' section. On the left is a navigation menu with categories like 'Text and sequences', 'Image generation', 'Distributed training', and 'GUIDE'. The main content area is titled 'Hardware platform' and discusses scaling training onto multiple GPUs or TPU machines. It lists five strategies: MirroredStrategy, TPUStrategy, MultiWorkerMirroredStrategy, CentralStorageStrategy, and ParameterServerStrategy. A table below details the support status for each strategy across different use cases.

MirroredStrategy	TPUStrategy	MultiWorkerMirroredStrategy	CentralStorageStrategy	ParameterServerStrategy
Supported	Support planned in 2.0 RC	Experimental support	Experimental support	Supported planned post 2.0
Experimental support	Experimental support	Support planned post 2.0 RC	Support planned in 2.0 RC	No support yet
Limited Support	Limited Support	Limited Support	Limited Support	Limited Support

Below the table, the 'MirroredStrategy' section is partially visible, mentioning `tf.distribute.MirroredStrategy` for synchronous distributed training on multiple GPUs.

Parameter server strategy is planned post 2.0 for Keras APIs. Custom training loop has experimental support in mirrored strategy and TPU strategy whereas, multi worker mirrored and central storage strategy are planned post 2.0 release candidate and there is no support for parameter server strategy as far as custom training loop is concerned. The estimators APIs have limited support for all the strategies. Let us look at how we can use the distributed training with `tf.keras` API through a concrete example.

(Refer Slide Time: 20:53)



The screenshot shows a Jupyter Notebook interface. At the top, a message states: 'This notebook is open with private outputs. Outputs will not be saved. You can disable this in [notebook settings](#).' The notebook has tabs for 'Text' and 'Copy to Drive'. The main content is divided into sections: 'Overview', 'Keras API', and 'Import dependencies'. The 'Overview' section explains that `tf.distribute.Strategy` provides an abstraction for distributing training across multiple processing units. It mentions that the tutorial uses `tf.distribute.MirroredStrategy` for in-graph replication with synchronous training on many GPUs on one machine. The 'Keras API' section states that this example uses the `tf.keras` API to build the model and training loop. The 'Import dependencies' section contains a code block for importing necessary modules.

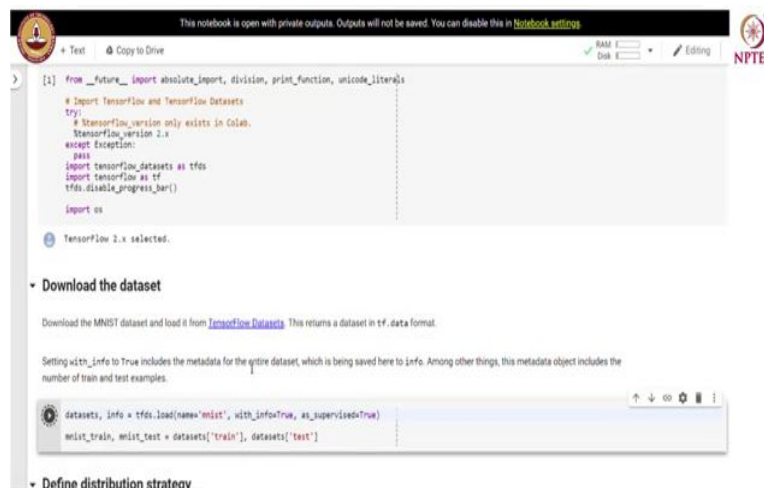
```
[ ] from __future__ import absolute_import, division, print_function, unicode_literals

# Import TensorFlow and TensorFlow Datasets
try:
    # TensorFlow version only exists in Colab.
    tensorflow_version = 2.x
except Exception:
    pass
import tensorflow_datasets as tfds
import tensorflow as tf
tf.disable_progress_bar()
```

So, concretely `tf.distribute.Strategy` API provides an abstraction for distributing your training across multiple processing units. The goal is to allow users to enable distributed training using existing models and training code with minimal changes. So, here we will use `tf.distribute.MirroredStrategy` in this example, this mirrored strategy does graph replication with synchronous training on many GPUs on a single machine.

Essentially, it copies all of the model's variables to each processor and then it uses all radio strategy to combine the gradients from all processors and applies the combined value to all copies of the model. Mirrored strategy is one of several distributed strategies available in TensorFlow.

(Refer Slide Time: 22:06)



The screenshot shows a Jupyter Notebook interface with the following content:

- At the top, a message states: "This notebook is open with private outputs. Outputs will not be saved. You can disable this in [Notebook settings](#)".
- Below the message, there are tabs for "Text" and "Copy to Drive".
- The main code cell contains the following Python code:

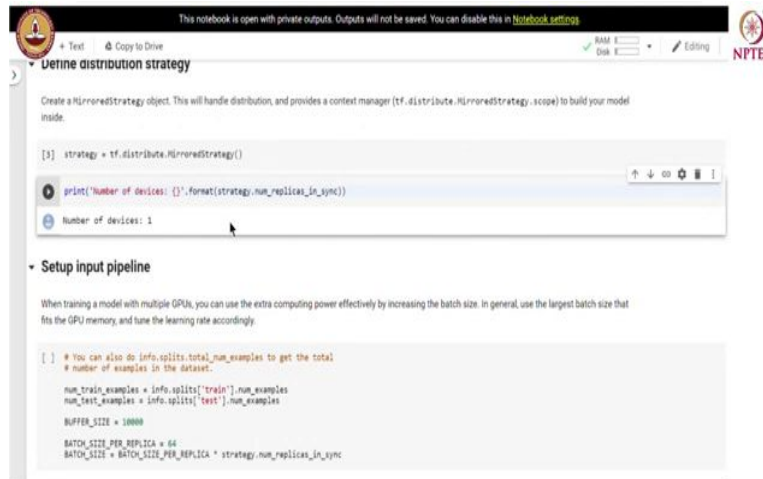
```
[1]: from __future__ import absolute_import, division, print_function, unicode_literals

# Import TensorFlow and TensorFlow Datasets
try:
    # TensorFlow version only exists in Colab
    TensorFlow_version 2.x
except Exception:
    pass
import tensorflow_datasets as tfds
import tensorflow as tf
tf.disable_progress_bar()

import os
```
- Below the code cell, there is a status bar that says "TensorFlow 2.x selected."
- Below the status bar, there is a section titled "Download the dataset".
- Under "Download the dataset", there is a link to "TensorFlow Datasets" and a description: "Download the MNIST dataset and load it from TensorFlow Datasets. This returns a dataset in tf.data format."
- Below the description, there is a note: "Setting with_info to True includes the metadata for the entire dataset, which is being saved here to info. Among other things, this metadata object includes the number of train and test examples."
- Below the note, there is a code cell with the following Python code:

```
datasets, info = tfds.load(name='mnist', with_info=True, as_supervised=True)
mnist_train, mnist_test = datasets['train'], datasets['test']
```
- Below the code cell, there is a section titled "Define distribution strategy".

(Refer Slide Time: 22:12)



The screenshot shows a Jupyter Notebook interface. At the top, there's a header bar with a logo on the left, a status message "This notebook is open with private outputs. Outputs will not be saved. You can disable this in [notebook settings](#)", and a RAM/Disk usage indicator on the right. Below the header, the notebook title "Define distribution strategy" is displayed. The main content area has a text description: "Create a MirroredStrategy object. This will handle distribution, and provides a context manager (tf.distribute.MirroredStrategy.scope) to build your model inside." Below this, there's a code cell with the following code:

```
[1]: strategy = tf.distribute.MirroredStrategy()

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

The output of the code cell is "Number of devices: 1". Below the code cell, there's a section titled "Setup input pipeline" with a text description: "When training a model with multiple GPUs, you can use the extra computing power effectively by increasing the batch size. In general, use the largest batch size that fits the GPU memory, and tune the learning rate accordingly." Below this, there's a code cell with the following code:

```
[ ] # You can also do info.split.total_num_examples to get the total
# number of examples in the dataset.

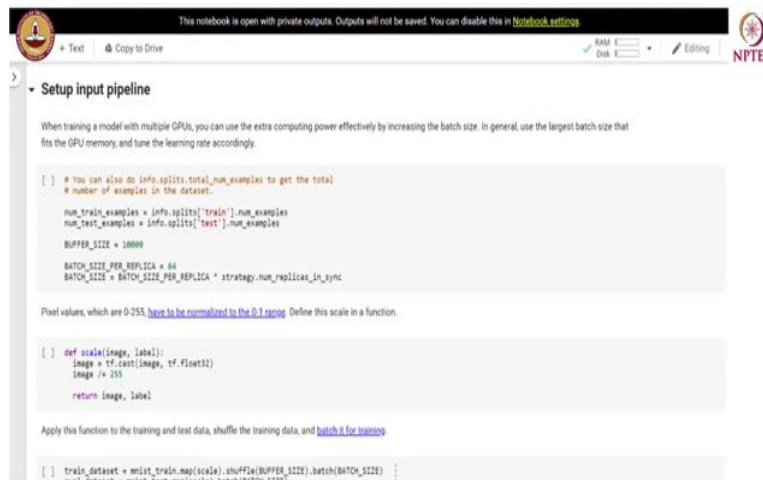
num_train_examples = info.split['train'].num_examples
num_test_examples = info.split['test'].num_examples

BUFFER_SIZE = 10000

BATCH_SIZE_PER_REPLICA = 64
BATCH_SIZE = BATCH_SIZE_PER_REPLICA * strategy.num_replicas_in_sync
```

So, let us import all the dependencies, let us download the MNIST dataset. We will create a MirroredStrategy object through this statement; the mirrored strategy will handle distribution and provide a context manager to build our model. The context manager for mirrored strategy is `tf.distribute.MirroredStrategy.scope`. Let us find out the number of devices that we have we have one device with us.

(Refer Slide Time: 22:55)



The screenshot shows a Jupyter Notebook interface. At the top, there's a header bar with a logo on the left, a status message "This notebook is open with private outputs. Outputs will not be saved. You can disable this in [notebook settings](#)", and a RAM/Disk usage indicator on the right. Below the header, the notebook title "Setup input pipeline" is displayed. The main content area has a text description: "When training a model with multiple GPUs, you can use the extra computing power effectively by increasing the batch size. In general, use the largest batch size that fits the GPU memory, and tune the learning rate accordingly." Below this, there's a code cell with the following code:

```
[ ] # You can also do info.split.total_num_examples to get the total
# number of examples in the dataset.

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

BUFFER_SIZE = 10000

BATCH_SIZE_PER_REPLICA = 64
BATCH_SIZE = BATCH_SIZE_PER_REPLICA * strategy.num_replicas_in_sync
```

Below the code cell, there's a text description: "Pixel values, which are 0-255, [have to be normalized to the 0-1 range](#). Define this scale in a function." Below this, there's a code cell with the following code:

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

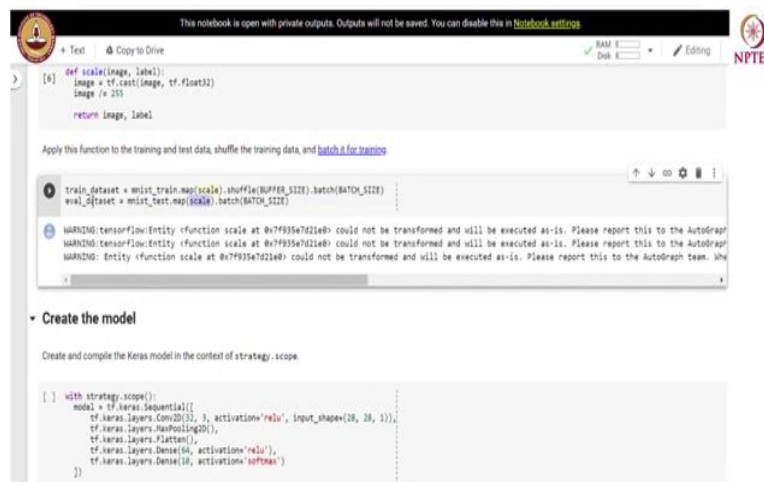
    return image, label
```

Below the code cell, there's a text description: "Apply this function to the training and test data, shuffle the training data, and [batch it for training](#)." Below this, there's a code cell with the following code:

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

So, let us build input pipeline. When training a model with multiple GPUs we can use extra computing power effectively by increasing the batch size. In general use the largest batch size that fits the GPU memory and tune the learning rate accordingly. So, we have batch size equal to batch size per replica into the number of replicas in synch from the strategy object.

(Refer Slide Time: 23:29)



The screenshot shows a Jupyter Notebook interface with the following code and output:

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

Apply this function to the training and test data, shuffle the training data, and [batch it for training](#)

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

Warnings:

```
WARNING:tensorflow:Entity <function scale at 0x7f935e7d21e0> could not be transformed and will be executed as-is. Please report this to the AutoGraph  
WARNING:tensorflow:Entity <function scale at 0x7f935e7d21e0> could not be transformed and will be executed as-is. Please report this to the AutoGraph  
WARNING:tensorflow:Entity <function scale at 0x7f935e7d21e0> could not be transformed and will be executed as-is. Please report this to the AutoGraph team. See
```

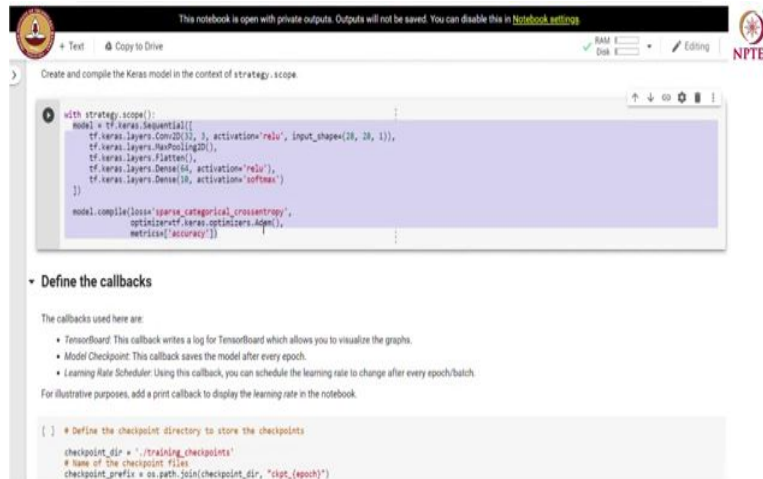
Create the model

Create and compile the Keras model in the context of strategy.scope

```
[ ] 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, activation='softmax')  
    ])
```

We will normalize the data and we apply the scaling function on each and every data point in the dataset then we shuffle the dataset and then we batch with the batch size set before. We go not know from shuffling on the evaluation dataset, we apply scaling on each and every data point in the dataset followed by batching operation.

(Refer Slide Time: 24:02)



The screenshot shows a Jupyter Notebook interface with a black header bar containing the text "This notebook is open with private outputs. Outputs will not be saved. You can disable this in [Notebook settings](#)". The notebook title is "Create and compile the Keras model in the context of strategy.scope". The code cell contains the following Python code:

```
with strategy.scope():
    model = tf.keras.Sequential([
        tf.keras.layers.Conv2D(3, 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, activation='softmax')
    ])
    model.compile(loss='sparse_categorical_crossentropy',
                  optimizer=tf.keras.optimizers.Adam(),
                  metrics=['accuracy'])
```

Below the code cell, there is a section titled "Define the callbacks" with the text "The callbacks used here are:" followed by a bulleted list:

- **TensorBoard** This callback writes a log for TensorBoard which allows you to visualize the graphs.
- **ModelCheckpoint** This callback saves the model after every epoch.
- **Learning Rate Scheduler** Using this callback, you can schedule the learning rate to change after every epoch/batch.

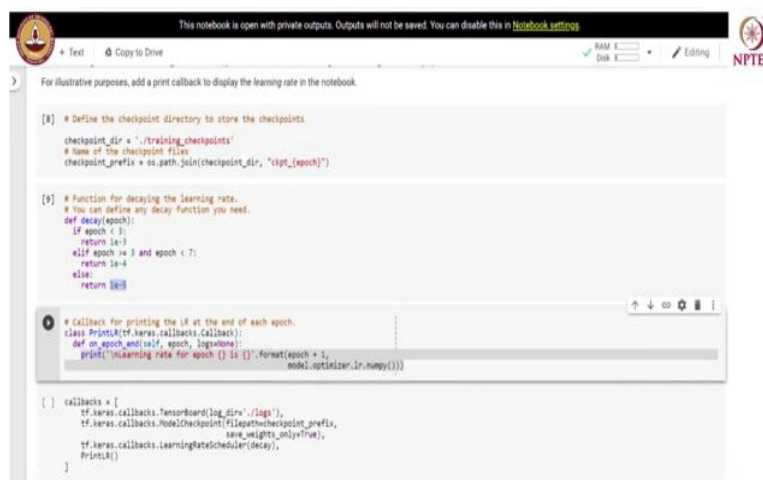
For illustrative purposes, add a print callback to display the learning rate in the notebook.

Below this, there is a code cell with the following Python code:

```
[ ] # Define the checkpoint directory to store the checkpoints
checkpoint_dir = './training_checkpoints'
# Name of the checkpoint files
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt_{epoch}")
```

Let us create a keras model in the context of strategy scope. So, this is one difference as when you want to do distributed training. So, this keras model is exactly the same keras model that we have been using throughout the course, we have used this same keras model for single machine training. Now, the only change that we do for distributed training is we define this model in the scope of a strategy. So, we start with `strategy.scope()` and we define model within this particular scope.

(Refer Slide Time: 24:46)



The screenshot shows a Jupyter Notebook interface with a black header bar containing the text "This notebook is open with private outputs. Outputs will not be saved. You can disable this in [Notebook settings](#)". The notebook title is "For illustrative purposes, add a print callback to display the learning rate in the notebook". The code cell contains the following Python code:

```
[8] # Define the checkpoint directory to store the checkpoints
checkpoint_dir = './training_checkpoints'
# Name of the checkpoint files
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt_{epoch}")

[9] # Function for decaying the learning rate.
# You can define any decay function you need.
def decay(epoch):
    if epoch < 1:
        return 1e-3
    elif epoch >= 1 and epoch < 7:
        return 1e-4
    else:
        return 1e-5

[ ] # Callback for printing the LR at the end of each epoch.
class PrintLR(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs=None):
        print('\nLearning rate for epoch {} is {}'.format(epoch + 1,
            model.optimizer.lr.numpy()))

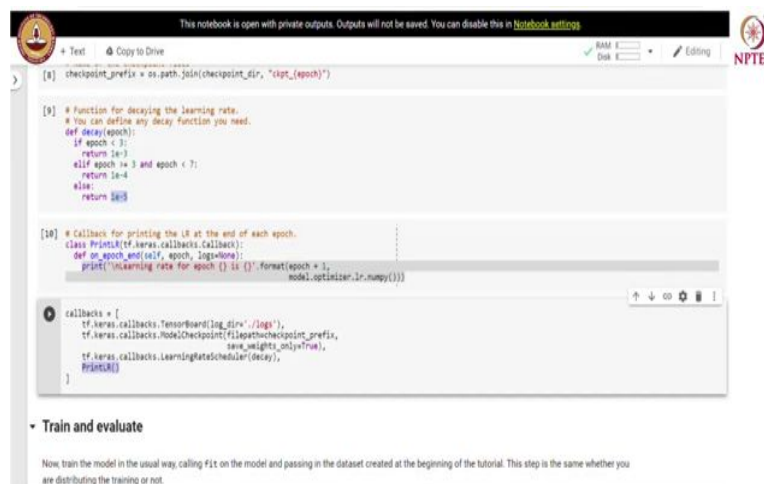
[ ] callbacks = [
    tf.keras.callbacks.TensorBoard(log_dir='./logs'),
    tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_prefix,
        save_weights_only=True),
    tf.keras.callbacks.LearningRateScheduler(decay),
    PrintLR()
]
```

We will use certain callbacks, we will use TensorBoard callback for writing logs for TensorBoard. TensorBoard allows us to visualize the graphs. Then we will use ModelCheckpoint callback for saving the models every epoch and we will also use LearningRateScheduler callback for scheduling learning rate to change after every epoch or batch. So, for illustrative purposes we add a print callback to display the learning rate in this notebook.

So, let us setup the checkpoint directory to store the checkpoint and give the checkpoint prefix. Next we define a function for decaying the learning rate; so, for first two epochs we will use .0001 as a learning rate for third epoch until 7th epoch, we will use some other learning rate and for every other epoch after 6th epoch we use even lesser learning rate.

Then we define a callback for printing learning rate at the end of each epoch. So, we write on epoch end event, we capture this particular event and in at this event at the end of epoch we print the learning rate that was used for the epoch. So, if you want to learn more about callbacks there is a guide available on the TensorFlow website that tells you how to write custom callbacks for tf.keras model.

(Refer Slide Time: 26:41)



```
[8] checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt_{epoch}")

[9] # Function for decaying the learning rate.
# You can define any decay function you need.
def decay(epoch):
    if epoch < 3:
        return 1e-4
    elif epoch >= 3 and epoch < 7:
        return 1e-4
    else:
        return 1e-5

[10] # Callback for printing the LR at the end of each epoch.
class PrintLR(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs=None):
        print("Learning rate for epoch {} is {}".format(epoch + 1,
            model.optimizer.lr.numpy()))

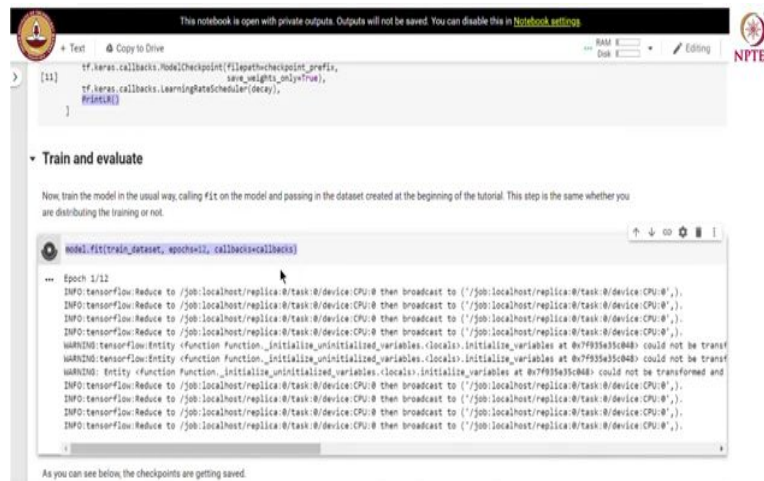
callbacks = [
    tf.keras.callbacks.TensorBoard(log_dir='./logs'),
    tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_prefix,
        save_weights_only=True),
    tf.keras.callbacks.LearningRateScheduler(decay),
    PrintLR()
]
```

• Train and evaluate

Now, train the model in the usual way, calling `fit` on the model and passing in the dataset created at the beginning of the tutorial. This step is the same whether you are distributing the training or not.

So, we put all callbacks in the callbacks list. So, we have used 3 inbuilt call callbacks and we have implemented one more callback for printing the learning rate at the end of each epoch.

(Refer Slide Time: 27:04)



The screenshot shows a Jupyter Notebook interface. At the top, there's a header with a logo and text: "This notebook is open with private outputs. Outputs will not be saved. You can disable this in [Notebook settings](#)". Below the header, there's a toolbar with icons for "Text", "Copy to Drive", "RAM", "Disk", and "Editing". The main code cell contains the following Python code:

```
[11]: tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_prefix,
    save_weights_only=True),
    tf.keras.callbacks.LearningRateScheduler(decay),
    ]
```

Below the code cell, there's a section titled "Train and evaluate". It contains a paragraph: "Now, train the model in the usual way, calling `fit` on the model and passing in the dataset created at the beginning of the tutorial. This step is the same whether you are distributing the training or not."

The next code cell contains the following Python code:

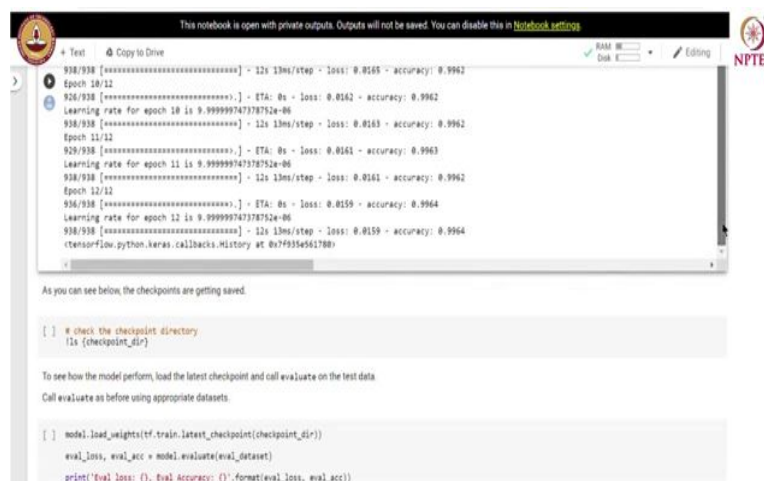
```
model.fit(train_dataset, epochs=12, callbacks=callbacks)
```

Below the code cell, there's a large block of text showing the output of the `fit` function. It starts with "Epoch 1/12" and then shows a series of log messages from TensorFlow, including "INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to ('/job:localhost/replica:0/task:0/device:CPU:0',)", "INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to ('/job:localhost/replica:0/task:0/device:CPU:0',)", "INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to ('/job:localhost/replica:0/task:0/device:CPU:0',)", "INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to ('/job:localhost/replica:0/task:0/device:CPU:0',)", "WARNING:tensorflow:Entity <function Function._initialize_uninitialized_variables.<locals>.initialize_variables at 0x7f935a530440> could not be transformed", "WARNING:tensorflow:Entity <function Function._initialize_uninitialized_variables.<locals>.initialize_variables at 0x7f935a530440> could not be transformed and", "INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to ('/job:localhost/replica:0/task:0/device:CPU:0',)", "INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to ('/job:localhost/replica:0/task:0/device:CPU:0',)", "INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to ('/job:localhost/replica:0/task:0/device:CPU:0',)", "INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to ('/job:localhost/replica:0/task:0/device:CPU:0',)".

At the bottom, there's a note: "As you can see below, the checkpoints are getting saved."

We train and evaluate model exactly like we are doing before. So, we call fit function in the model, you can see that the model is getting trained.

(Refer Slide Time: 27:44)



The screenshot shows a Jupyter Notebook interface. At the top, there's a header with a logo and text: "This notebook is open with private outputs. Outputs will not be saved. You can disable this in [Notebook settings](#)". Below the header, there's a toolbar with icons for "Text", "Copy to Drive", "RAM", "Disk", and "Editing". The main code cell contains the following Python code:

```
[ ]: # check the checkpoint directory
ls (checkpoint_dir)
```

Below the code cell, there's a section titled "To see how the model perform, load the latest checkpoint and call evaluate on the test data". It contains a paragraph: "Call evaluate as before using appropriate datasets."

The next code cell contains the following Python code:

```
[ ]: model.load_weights(tf.train.latest_checkpoint(checkpoint_dir))
eval_loss, eval_acc = model.evaluate(eval_dataset)
print('Eval loss: {}, Eval Accuracy: {}'.format(eval_loss, eval_acc))
```

Below the code cell, there's a large block of text showing the output of the `fit` function. It starts with "Epoch 10/12" and then shows a series of log messages from TensorFlow, including "INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to ('/job:localhost/replica:0/task:0/device:CPU:0',)", "INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to ('/job:localhost/replica:0/task:0/device:CPU:0',)", "INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to ('/job:localhost/replica:0/task:0/device:CPU:0',)", "INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to ('/job:localhost/replica:0/task:0/device:CPU:0',)", "WARNING:tensorflow:Entity <function Function._initialize_uninitialized_variables.<locals>.initialize_variables at 0x7f935a530440> could not be transformed", "WARNING:tensorflow:Entity <function Function._initialize_uninitialized_variables.<locals>.initialize_variables at 0x7f935a530440> could not be transformed and", "INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to ('/job:localhost/replica:0/task:0/device:CPU:0',)", "INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to ('/job:localhost/replica:0/task:0/device:CPU:0',)", "INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to ('/job:localhost/replica:0/task:0/device:CPU:0',)", "INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to ('/job:localhost/replica:0/task:0/device:CPU:0',)".

At the bottom, there's a note: "As you can see below, the checkpoints are getting saved."

You can see that after 12 epochs the model has reached accuracy of 99.64 and you can see that it is printing the learning rate at the end of each epoch.

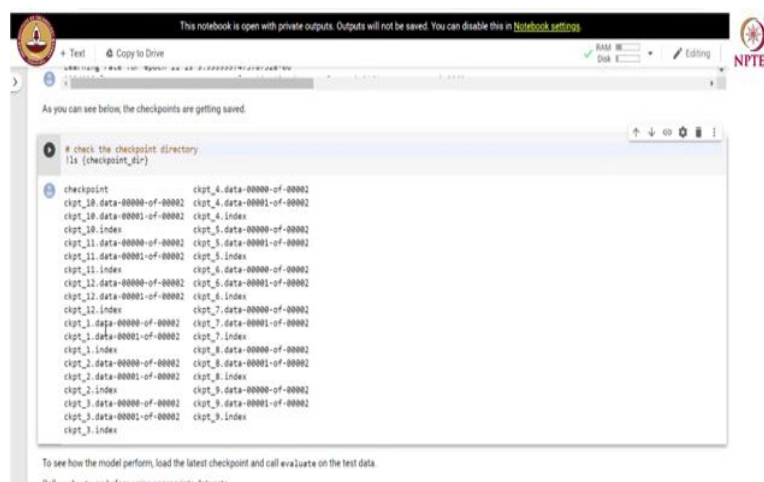
(Refer Slide Time: 28:05)



```
Epoch 3/12
936/936 [#####] - ETA: 0s - loss: 0.0471 - accuracy: 0.9856
Learning rate for epoch 3 is 0.0010000000476974513
938/938 [#####] - ETA: 0s - loss: 0.0470 - accuracy: 0.9857
Epoch 4/12
933/938 [#####] - ETA: 0s - loss: 0.0258 - accuracy: 0.9931
Learning rate for epoch 4 is 9.9999997378752e-05
938/938 [#####] - ETA: 0s - loss: 0.0257 - accuracy: 0.9931
Epoch 5/12
935/938 [#####] - ETA: 0s - loss: 0.0228 - accuracy: 0.9941
Learning rate for epoch 5 is 9.999999747378752e-05
938/938 [#####] - ETA: 0s - loss: 0.0227 - accuracy: 0.9941
Epoch 6/12
934/938 [#####] - ETA: 0s - loss: 0.0209 - accuracy: 0.9945
Learning rate for epoch 6 is 9.999999747378752e-05
938/938 [#####] - ETA: 0s - loss: 0.0209 - accuracy: 0.9945
Epoch 7/12
936/938 [#####] - ETA: 0s - loss: 0.0192 - accuracy: 0.9951
Learning rate for epoch 7 is 9.999999747378752e-05
938/938 [#####] - ETA: 0s - loss: 0.0192 - accuracy: 0.9951
Epoch 8/12
934/938 [#####] - ETA: 0s - loss: 0.0168 - accuracy: 0.9960
Learning rate for epoch 8 is 9.999999747378752e-06
938/938 [#####] - ETA: 0s - loss: 0.0168 - accuracy: 0.9960
Epoch 9/12
933/938 [#####] - ETA: 0s - loss: 0.0165 - accuracy: 0.9962
Learning rate for epoch 9 is 9.999999747378752e-06
938/938 [#####] - ETA: 0s - loss: 0.0165 - accuracy: 0.9962
Epoch 10/12
926/938 [#####] - ETA: 0s - loss: 0.0162 - accuracy: 0.9962
Learning rate for epoch 10 is 9.999999747378752e-06
938/938 [#####] - ETA: 0s - loss: 0.0162 - accuracy: 0.9962
```

So, initially the learning rate was 0.001 and after 4th epoch, it was reduced and we can see that after 7th epoch, it has gone further down. As we get closer and closer to the minima, we are taking smaller steps in the direction of the gradient.

(Refer Slide Time: 28:39)



```
As you can see below, the checkpoints are getting saved.

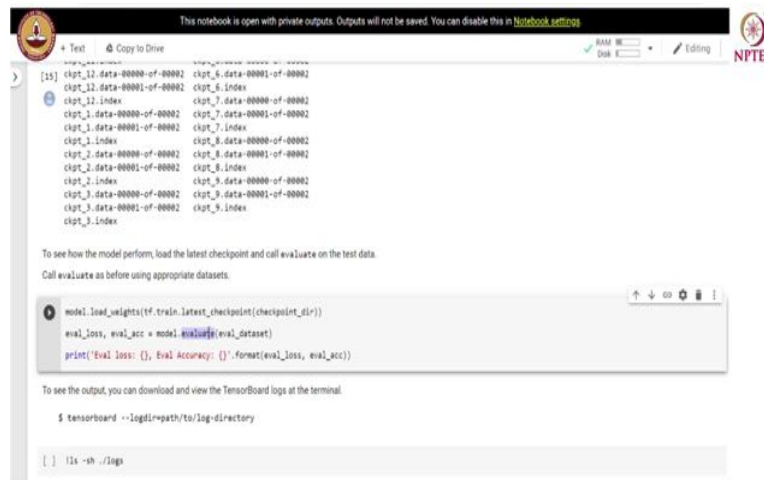
# check the checkpoint directory
ls (checkpoint_dir)

checkpoint
ckpt_10_data-00000-of-00002  ckpt_4_data-00001-of-00002
ckpt_10_data-00001-of-00002  ckpt_4_index
ckpt_10_index                ckpt_5_data-00000-of-00002
ckpt_11_data-00000-of-00002  ckpt_5_data-00001-of-00002
ckpt_11_data-00001-of-00002  ckpt_5_index
ckpt_11_index                ckpt_6_data-00000-of-00002
ckpt_12_data-00000-of-00002  ckpt_6_data-00001-of-00002
ckpt_12_data-00001-of-00002  ckpt_6_index
ckpt_12_index                ckpt_7_data-00000-of-00002
ckpt_1_data-00000-of-00002   ckpt_7_data-00001-of-00002
ckpt_1_data-00001-of-00002   ckpt_7_index
ckpt_1_index                 ckpt_8_data-00000-of-00002
ckpt_2_data-00000-of-00002   ckpt_8_data-00001-of-00002
ckpt_2_data-00001-of-00002   ckpt_8_index
ckpt_2_index                 ckpt_9_data-00000-of-00002
ckpt_3_data-00000-of-00002   ckpt_9_data-00001-of-00002
ckpt_3_data-00001-of-00002   ckpt_9_index
```

Let us see how checkpoints are getting saved. So, we perform the directory listing on the checkpoint directory and you can see that there are multiple checkpoints that we are saved.

So, after every epoch you are having a single checkpoint. So, there are 12 checkpoints stored for 12 epochs for which we trained a model.

(Refer Slide Time: 29:06)



```
[15] ckpt_11.data-00000-of-00002 ckpt_5.data-00001-of-00002
ckpt_11.data-00001-of-00002 ckpt_6.index
ckpt_11.index ckpt_7.data-00000-of-00002
ckpt_1.data-00000-of-00002 ckpt_7.data-00001-of-00002
ckpt_1.data-00001-of-00002 ckpt_7.index
ckpt_1.index ckpt_8.data-00000-of-00002
ckpt_2.data-00000-of-00002 ckpt_8.data-00001-of-00002
ckpt_2.data-00001-of-00002 ckpt_8.index
ckpt_2.index ckpt_9.data-00000-of-00002
ckpt_3.data-00000-of-00002 ckpt_9.data-00001-of-00002
ckpt_3.data-00001-of-00002 ckpt_9.index
ckpt_3.index
```

To see how the model perform, load the latest checkpoint and call evaluate on the test data.
Call evaluate as before using appropriate datasets.

```
model.load_weights(tf.train.latest_checkpoint(checkpoint_dir))
eval_loss, eval_acc = model.evaluate(eval_dataset)
print('Eval loss: {}, Eval Accuracy: {}'.format(eval_loss, eval_acc))
```

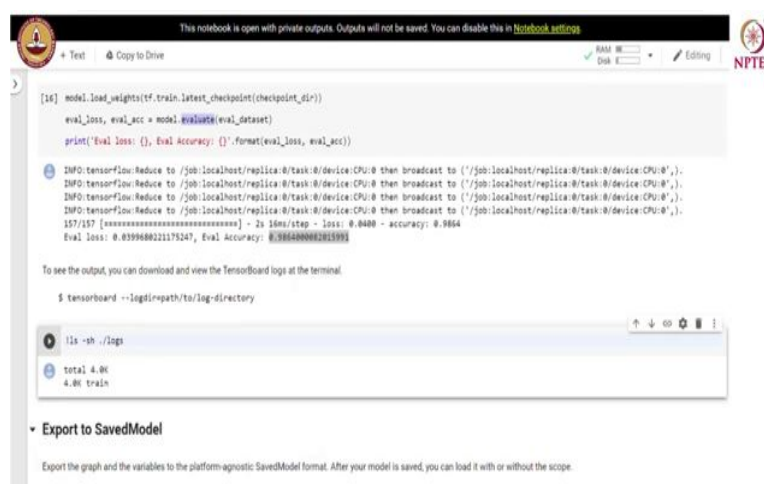
To see the output, you can download and view the TensorBoard logs at the terminal.

```
$ tensorboard --logdir=path/to/log-directory
```

```
[ ] !ls -sh ./logs
```

Let us check how the model performs. For that what we will do is, we load the model weights from the latest checkpoint from the checkpoint directory. And, then we will calculate the performance of the model by calling the evaluate function.

(Refer Slide Time: 29:28)



```
[16] model.load_weights(tf.train.latest_checkpoint(checkpoint_dir))
eval_loss, eval_acc = model.evaluate(eval_dataset)
print('Eval loss: {}, Eval Accuracy: {}'.format(eval_loss, eval_acc))
```

```
INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to (/job:localhost/replica:0/task:0/device:CPU:0),
INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to (/job:localhost/replica:0/task:0/device:CPU:0),
INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to (/job:localhost/replica:0/task:0/device:CPU:0),
INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to (/job:localhost/replica:0/task:0/device:CPU:0),
157/157 [#####] - 2s 16ms/step - loss: 0.0400 - accuracy: 0.9864
Eval loss: 0.039680221175247, Eval Accuracy: 0.98640000201595
```

To see the output, you can download and view the TensorBoard logs at the terminal.

```
$ tensorboard --logdir=path/to/log-directory
```

```
!ls -sh ./logs
```

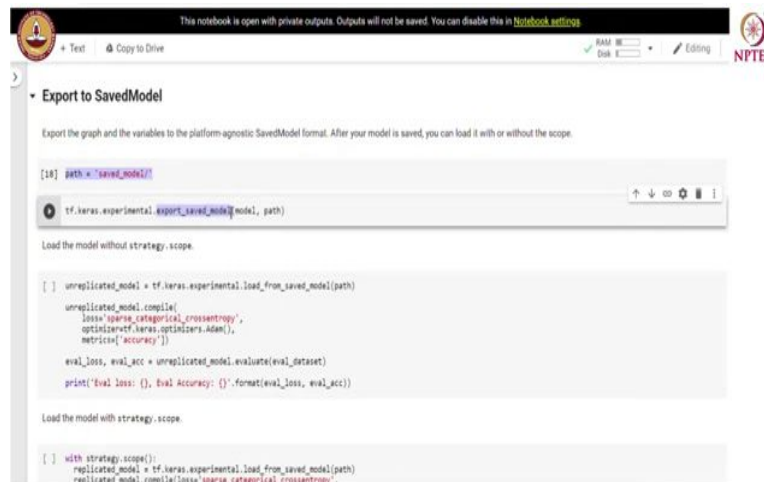
```
total 4.0K
4.0K train
```

Export to SavedModel

Export the graph and the variables to the platform-agnostic SavedModel format. After your model is saved, you can load it with or without the scope.

So, you can see that on the evaluation set we achieved 98.64 percent accuracy. Let us look at the log directory; this is where we have stored logs that can be read to TensorBoard.

(Refer Slide Time: 30:02)



```
This notebook is open with private outputs. Outputs will not be saved. You can disable this in Notebook settings
```

+ Text | Copy to Drive

RAM 16 GB | Disk 100 GB | Editing | NPTEL

Export to SavedModel

Export the graph and the variables to the platform-agnostic SavedModel format. After your model is saved, you can load it with or without the scope.

```
[18]: path = 'saved_model/'

tf.keras.experimental.export_saved_model(model, path)

Load the model without strategy.scope.

[ ] unreplicated_model = tf.keras.experimental.load_from_saved_model(path)

unreplicated_model.compile(
    loss=tf.keras.losses.categorical_crossentropy,
    optimizer=tf.keras.optimizers.Adam(),
    metrics=['accuracy'])

eval_loss, eval_acc = unreplicated_model.evaluate(eval_dataset)

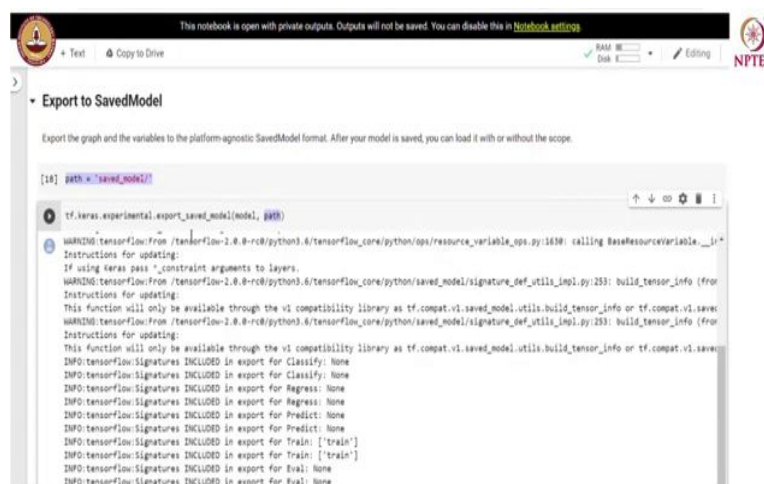
print('Eval loss: {}, Eval Accuracy: {}'.format(eval_loss, eval_acc))

Load the model with strategy.scope.

[ ] with strategy.scope():
    replicated_model = tf.keras.experimental.load_from_saved_model(path)
    replicated_model.compile(loss=tf.keras.losses.categorical_crossentropy,
```

We can export the graph and the variables to the platform agnostic save model format, after the model is saved we can load it with or without the scope.

(Refer Slide Time: 30:30)



```
This notebook is open with private outputs. Outputs will not be saved. You can disable this in Notebook settings
```

+ Text | Copy to Drive

RAM 16 GB | Disk 100 GB | Editing | NPTEL

Export to SavedModel

Export the graph and the variables to the platform-agnostic SavedModel format. After your model is saved, you can load it with or without the scope.

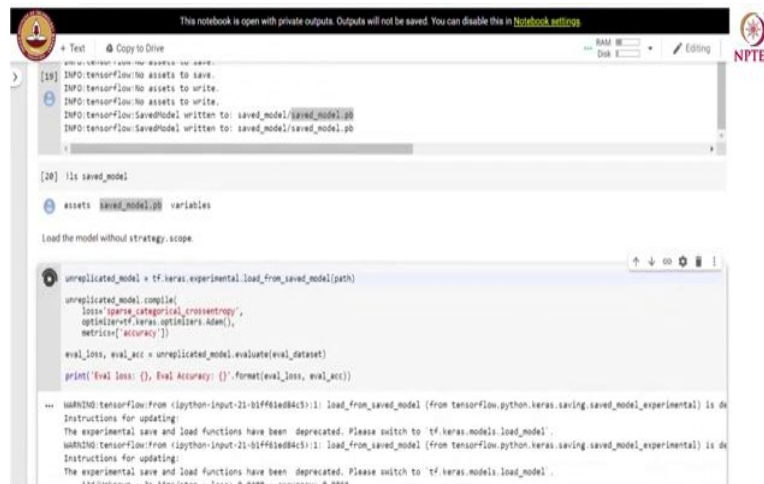
```
[18]: path = 'saved_model/'

tf.keras.experimental.export_saved_model(model, path)
```

```
WARNING:tensorflow:From /tensorflow-2.0.0-rc0/python3.6/tensorflow_core/python/ops/resource_variable_ops.py:1650: calling BaseResourceVariable.__init__
Instructions for updating:
If using keras pass *_constraint arguments to layers.
WARNING:tensorflow:From /tensorflow-2.0.0-rc0/python3.6/tensorflow_core/python/saved_model/signature_def_util_impl.py:253: build_tensor_info (from
Instructions for updating:
This function will only be available through the v1 compatibility library as tf.compat.v1.saved_model.utils.build_tensor_info or tf.compat.v1.saved
WARNING:tensorflow:From /tensorflow-2.0.0-rc0/python3.6/tensorflow_core/python/saved_model/signature_def_util_impl.py:253: build_tensor_info (from
Instructions for updating:
This function will only be available through the v1 compatibility library as tf.compat.v1.saved_model.utils.build_tensor_info or tf.compat.v1.saved
INFO:tensorflow:Signatures INCLUDED in export for Classify: None
INFO:tensorflow:Signatures INCLUDED in export for Regress: None
INFO:tensorflow:Signatures INCLUDED in export for Regress: None
INFO:tensorflow:Signatures INCLUDED in export for Predict: None
INFO:tensorflow:Signatures INCLUDED in export for Predict: None
INFO:tensorflow:Signatures INCLUDED in export for Train: ['train']
INFO:tensorflow:Signatures INCLUDED in export for Train: ['train']
INFO:tensorflow:Signatures INCLUDED in export for Eval: None
INFO:tensorflow:Signatures INCLUDED in export for Eval: None
```

So, we specify path for saving the model and we use `export_saved_model` from the experimental version and save the model in the specified path.

(Refer Slide Time: 30:33)



The screenshot shows a Jupyter Notebook interface with the following content:

- Cell [19]:

```
INFO:tensorflow:No assets to save.  
INFO:tensorflow:No assets to write.  
INFO:tensorflow:SavedModel written to: saved_model/saved_model.pb  
INFO:tensorflow:SavedModel written to: saved_model/saved_model.pb
```
- Cell [20]:

```
!ls saved_model
```

assets saved_model.pb variables

Load the model without strategy scope.

```
unreplicated_model = tf.keras.experimental.load_from_saved_model(path)  
unreplicated_model.compile(  
    loss='sparse_categorical_crossentropy',  
    optimizer=tf.keras.optimizers.Adam(),  
    metrics=['accuracy'])  
eval_loss, eval_acc = unreplicated_model.evaluate(eval_dataset)  
print('Eval loss: {}, Eval Accuracy: {}'.format(eval_loss, eval_acc))
```

At the bottom, there are deprecation warnings from TensorFlow:

```
WARNING:tensorflow:From <ipython-input-21-b1f6f1ed84c5>:1: load_from_saved_model (from tensorflow.python.keras.saving.saved_model_experimental) is deprecated.  
Instructions for updating:  
The experimental save and load functions have been deprecated. Please switch to 'tf.keras.models.load_model'.  
WARNING:tensorflow:From <ipython-input-21-b1f6f1ed84c5>:1: load_from_saved_model (from tensorflow.python.keras.saving.saved_model_experimental) is deprecated.  
Instructions for updating:  
The experimental save and load functions have been deprecated. Please switch to 'tf.keras.models.load_model'.
```

Let us check the content of the `saved_model` directory. And, you can see that the model has been saved in the `saved_model` directory. So, the model has been saved to `saved_model.pb`. Let us load the model without the strategy scope. This is a replicated model that was loaded from the saved model path. After loading the model we compile the model and perform the evaluation on the model.

(Refer Slide Time: 31:31)



This notebook is open with private outputs. Outputs will not be saved. You can disable this in [Notebook settings](#).

Text Copy to Drive

RAM 16 GB Disk 100 GB

Editing

NPTEL

```
[20] !ls saved_model
```

assets saved_model@ variables

Load the model without strategy.scope.

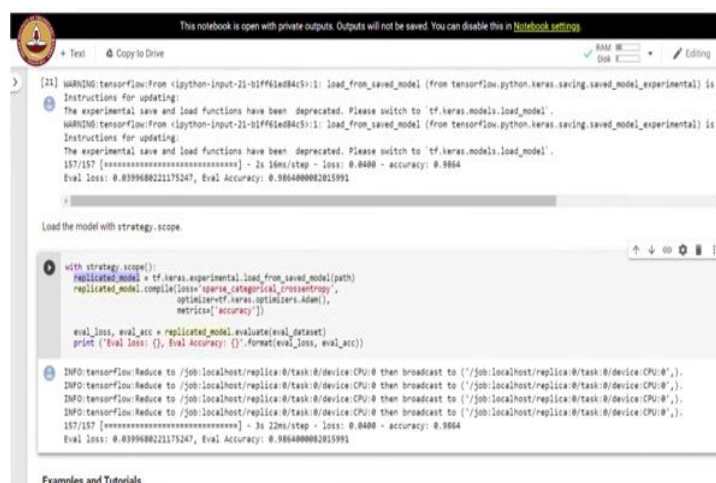
```
unreplicated_model = tf.keras.experimental.load_from_saved_model(path)
unreplicated_model.compile(
    loss='sparse_categorical_crossentropy',
    optimizer=tf.keras.optimizers.Adam(),
    metrics=['accuracy'])
eval_loss, eval_acc = unreplicated_model.evaluate(eval_dataset)
print('Eval loss: {}, Eval Accuracy: {}'.format(eval_loss, eval_acc))
```

```
WARNING:tensorflow:from <ipython-input-21-01ff61ed84c5>:1: load_from_saved_model (from tensorflow.python.keras.saving.saved_model_experimental) is deprecated. Instructions for updating: The experimental save and load functions have been deprecated. Please switch to 'tf.keras.models.load_model'.
WARNING:tensorflow:from <ipython-input-21-01ff61ed84c5>:1: load_from_saved_model (from tensorflow.python.keras.saving.saved_model_experimental) is deprecated. Instructions for updating: The experimental save and load functions have been deprecated. Please switch to 'tf.keras.models.load_model'.
157/157 [#####] - 2s 16ms/step - loss: 0.0400 - accuracy: 0.9864
Eval loss: 0.0399689221175247, Eval Accuracy: 0.9864000002015991
```

Load the model with strategy.scope.

We can see that we achieve the same accuracy as before.

(Refer Slide Time: 31:35)



This notebook is open with private outputs. Outputs will not be saved. You can disable this in [Notebook settings](#).

Text Copy to Drive

RAM 16 GB Disk 100 GB

Editing

NPTEL

```
[21] WARNING:tensorflow:from <ipython-input-21-01ff61ed84c5>:1: load_from_saved_model (from tensorflow.python.keras.saving.saved_model_experimental) is deprecated. Instructions for updating: The experimental save and load functions have been deprecated. Please switch to 'tf.keras.models.load_model'.
WARNING:tensorflow:from <ipython-input-21-01ff61ed84c5>:1: load_from_saved_model (from tensorflow.python.keras.saving.saved_model_experimental) is deprecated. Instructions for updating: The experimental save and load functions have been deprecated. Please switch to 'tf.keras.models.load_model'.
157/157 [#####] - 2s 16ms/step - loss: 0.0400 - accuracy: 0.9864
Eval loss: 0.0399689221175247, Eval Accuracy: 0.9864000002015991
```

Load the model with strategy.scope.

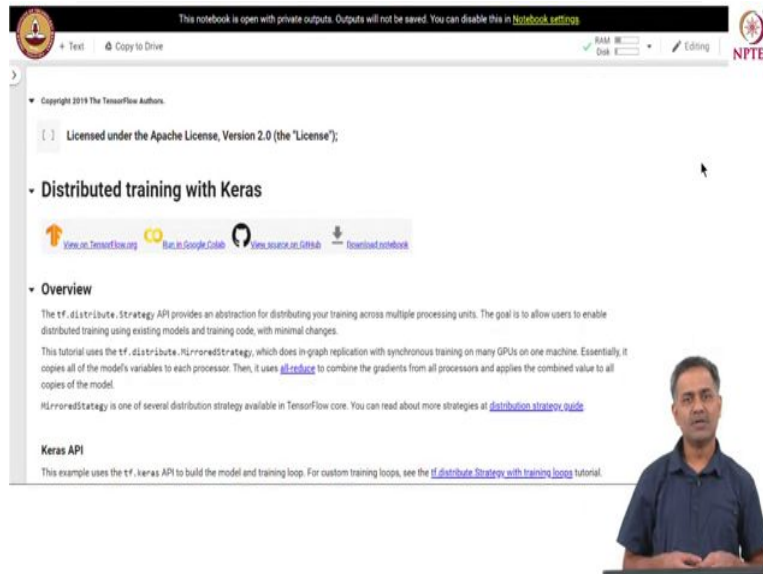
```
with strategy.scope():
    replicated_model = tf.keras.experimental.load_from_saved_model(path)
    replicated_model.compile(loss='sparse_categorical_crossentropy',
                           optimizer=tf.keras.optimizers.Adam(),
                           metrics=['accuracy'])
    eval_loss, eval_acc = replicated_model.evaluate(eval_dataset)
    print('Eval loss: {}, Eval Accuracy: {}'.format(eval_loss, eval_acc))
```

```
INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to (/job:localhost/replica:0/task:0/device:CPU:0).
INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to (/job:localhost/replica:0/task:0/device:CPU:0).
INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to (/job:localhost/replica:0/task:0/device:CPU:0).
INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to (/job:localhost/replica:0/task:0/device:CPU:0).
157/157 [#####] - 3s 22ms/step - loss: 0.0400 - accuracy: 0.9864
Eval loss: 0.0399689221175247, Eval Accuracy: 0.9864000002015991
```

Examples and Tutorials

Let us load the model with strategy scope. So, we write with strategy scope everything else remains the same, we have changed the name of the model from the replicated model, we have changed it to replicated_model and so, when we evaluated the model we again achieved almost the same accuracy.

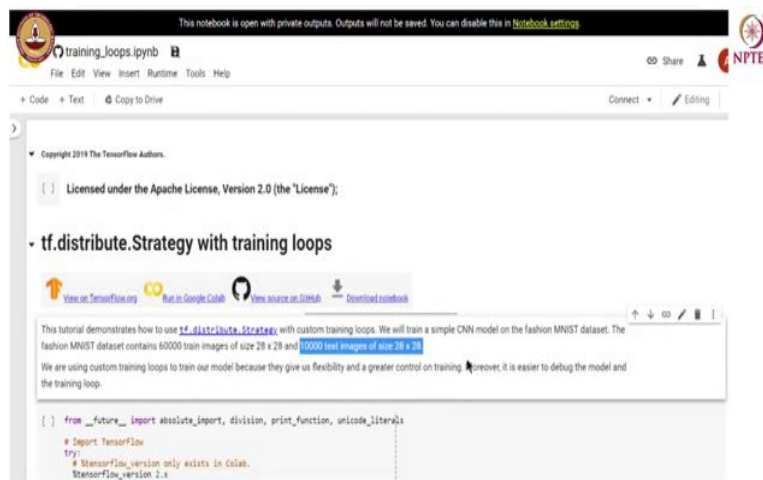
(Refer Slide Time: 32:06)



The screenshot shows a Jupyter Notebook interface. At the top, a black banner reads: "This notebook is open with private outputs. Outputs will not be saved. You can disable this in [notebook settings](#)". The notebook title is "Distributed training with Keras". Below the title, there are links to "View on TensorFlow.org", "Run in Google Colab", "View source on GitHub", and "Download notebook". The "Overview" section explains that the `tf.distribute.Strategy` API provides an abstraction for distributing training across multiple processing units. It mentions that the tutorial uses `tf.distribute.MirroredStrategy`, which does in-graph replication with synchronous training on many GPUs on one machine. It also mentions that `MirroredStrategy` is one of several distribution strategies available in TensorFlow core. The "Keras API" section states that this example uses the `tf.keras` API to build the model and training loop. A man in a blue shirt is visible in the bottom right corner of the notebook interface.

So, this is an example of how to use distributed training strategy with `tf.keras` API. So, here we use mirrored strategy for training keras model on MNIST dataset. Let us try to see how to use the distributed training strategy for custom training loops. So, we have seen that custom training loop provides a way of extending TensorFlow functionality. Here we demonstrate how to use distributed training strategy with custom training loops.

(Refer Slide Time: 32:28)



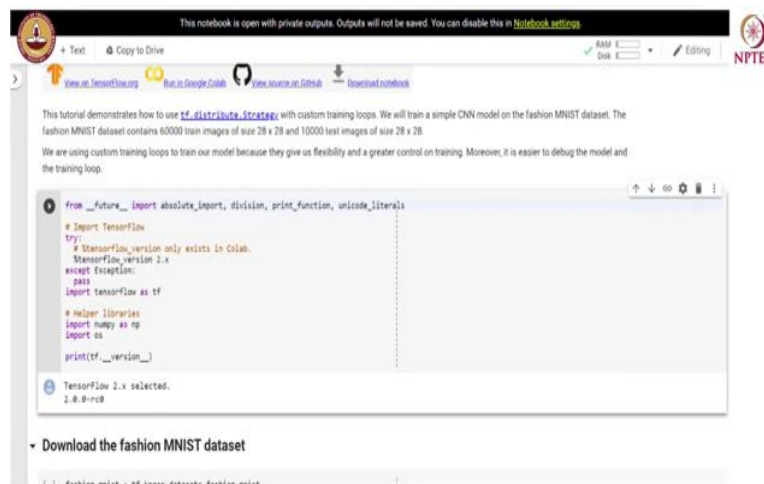
The screenshot shows a Jupyter Notebook interface. At the top, a black banner reads: "This notebook is open with private outputs. Outputs will not be saved. You can disable this in [notebook settings](#)". The notebook title is "training_loops.ipynb". Below the title, there are links to "View on TensorFlow.org", "Run in Google Colab", "View source on GitHub", and "Download notebook". The "tf.distribute.Strategy with training loops" section explains that the tutorial demonstrates how to use `tf.distribute.Strategy` with custom training loops. It mentions that they will train a simple CNN model on the fashion MNIST dataset. The fashion MNIST dataset contains 60000 train images of size 28 x 28 and 10000 test images of size 28 x 28. It also mentions that they are using custom training loops to train their model because they give us flexibility and a greater control on training. Moreover, it is easier to debug the model and the training loop. The code block shows the following code:

```
[ ] from __future__ import absolute_import, division, print_function, unicode_literals

# Import TensorFlow
try:
    # TensorFlow version only exists in Colab.
    TensorFlow_version 2.x
except Exception:
```

We will train a simple CNN model on fashion MNIST dataset. So, fashion MNIST dataset has 60000 training images, if you may recall each image is of size 28x28 and there were 10000 test images of size 28x28.

(Refer Slide Time: 33:09)



```
from __future__ import absolute_import, division, print_function, unicode_literals

# Import TensorFlow
try:
    # TensorFlow version only exists in Colab.
    TensorFlow_version 2.x
except Exception:
    pass
import tensorflow as tf

# Helper libraries
import numpy as np
import os

print(tf.__version__)
```

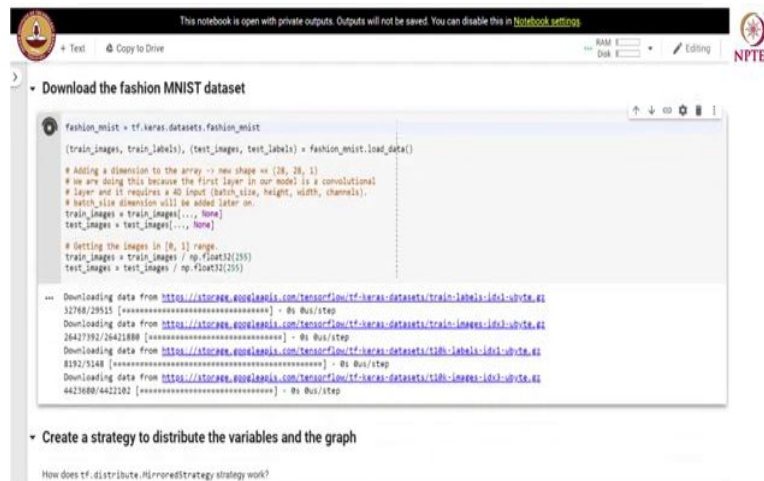
TensorFlow 2.x selected.
2.0.0-rc0

Download the fashion MNIST dataset

```
1. fashion_mnist = tf.keras.datasets.fashion_mnist
```

So, let us import the required libraries, we are using custom training loop to train our model, because they give us flexibility and a greater control on training. Moreover, it is easier to debug the model and the training loop.

(Refer Slide Time: 33:32)



This notebook is open with private outputs. Outputs will not be saved. You can disable this in [Notebook settings](#)

Text Copy to Drive RAM 8 Disk 8 Editing NPTEL

Download the fashion MNIST dataset

```
fashion_mnist = tf.keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()

# Adding a dimension to the array -> new shape == (28, 28, 1)
# We are doing this because the first layer in our model is a convolutional
# layer and it requires a 4D input (batch_size, height, width, channels).
# batch_size dimension will be added later on.
train_images = train_images[..., None]
test_images = test_images[..., None]

# Setting the images in [0, 1] range.
train_images = train_images / np.float32(255)
test_images = test_images / np.float32(255)
```

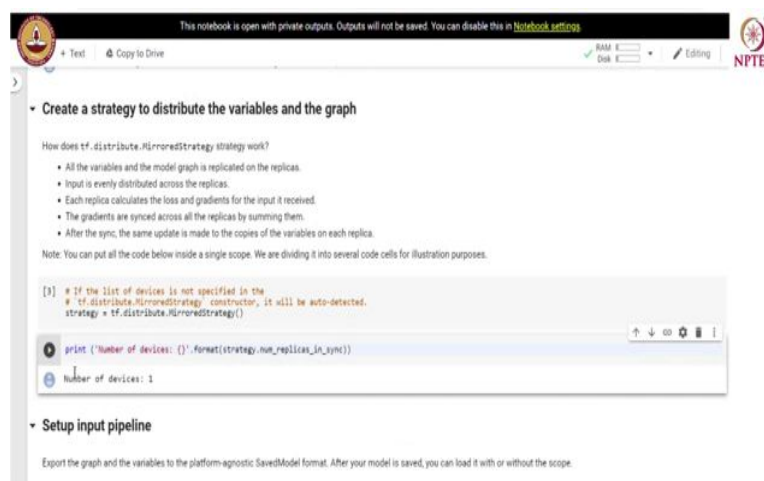
... Downloading data from <https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz>
32768/29513 [=====] - 0s 0us/step
Downloading data from <https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx3-ubyte.gz>
26427392/26421888 [=====] - 0s 0us/step
Downloading data from <https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10h-labels-idx1-ubyte.gz>
8192/5148 [=====] - 0s 0us/step
Downloading data from <https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10h-images-idx3-ubyte.gz>
4423680/4423282 [=====] - 0s 0us/step

Create a strategy to distribute the variables and the graph

How does `tf.distribute.MirroredStrategy` strategy work?

Let us download the fashion MNIST dataset.

(Refer Slide Time: 33:38)



This notebook is open with private outputs. Outputs will not be saved. You can disable this in [Notebook settings](#)

Text Copy to Drive RAM 8 Disk 8 Editing NPTEL

Create a strategy to distribute the variables and the graph

How does `tf.distribute.MirroredStrategy` strategy work?

- All the variables and the model graph is replicated on the replicas.
- Input is evenly distributed across the replicas.
- Each replica calculates the loss and gradients for the input it received.
- The gradients are synced across all the replicas by summing them.
- After the sync, the same update is made to the copies of the variables on each replica.

Note: You can put all the code below inside a single scope. We are dividing it into several code cells for illustration purposes.

```
[1] # If the list of devices is not specified in the
# tf.distribute.MirroredStrategy constructor, it will be auto-detected.
strategy = tf.distribute.MirroredStrategy()
```

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

Number of devices: 1

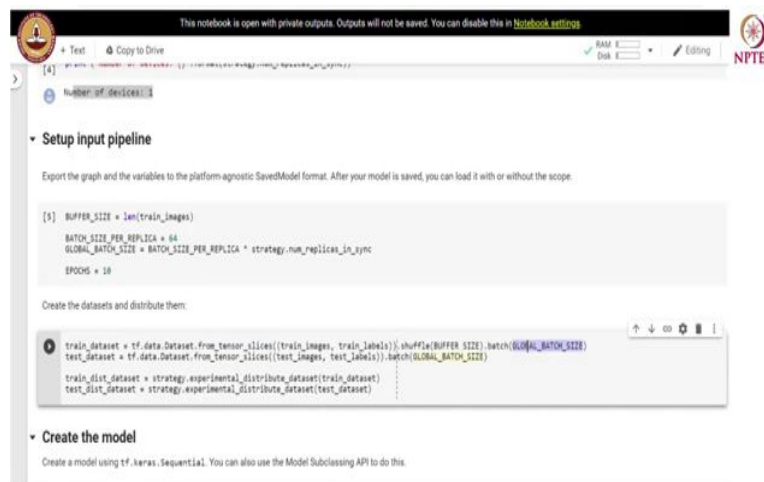
Setup input pipeline

Export the graph and the variables to the platform-agnostic SavedModel format. After your model is saved, you can load it with or without the scope.

Let us create a strategy to distribute the variables and the graph. So, let us recall how the mirrored strategy works. So, first all variables and model graphs are replicated on the GPUs, input is evenly distributed across replicas; each replica calculates the loss and gradient for the input it received.

The gradients are synced across all replicas by summing them, after the sync the same update is made to the copies of variable on each replica. So, let us create a strategy object and this strategy object is a mirrored strategy. We can check the number of replicas that are in sync here. So, we have a single device for our training in this case.

(Refer Slide Time: 34:57)



The screenshot shows a Jupyter Notebook interface with the following content:

- Header: "This notebook is open with private outputs. Outputs will not be saved. You can disable this in [Notebook settings](#)"
- Buttons: "+ Text", "Copy to Drive", "RAM", "Disk", "Editing", and the NPTEL logo.
- Code cell [4]:

```
number_of_devices = 1
```
- Section: "Setup input pipeline"
- Text: "Export the graph and the variables to the platform-agnostic SavedModel format. After your model is saved, you can load it with or without the scope."
- Code cell [5]:

```
BUFFER_SIZE = len(train_images)
BATCH_SIZE_PER_REPLICA = 64
GLOBAL_BATCH_SIZE = BATCH_SIZE_PER_REPLICA * strategy.num_replicas_in_sync
EPOCHS = 10
```
- Text: "Create the datasets and distribute them:"
- Code cell [6]:

```
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)
train_dist_dataset = strategy.experimental_distribute_dataset(train_dataset)
test_dist_dataset = strategy.experimental_distribute_dataset(test_dataset)
```
- Section: "Create the model"
- Text: "Create a model using `tf.keras.Sequential`. You can also use the Model Subclassing API to do this."

Let us build the input pipeline, let us create database, let us create the datasets and distribute them. So, we use the `dataset.from_tensor_slices` for creating the dataset. We shuffle it and then we batch according to the batch size specified over here.

(Refer Slide Time: 35:29)



The screenshot shows a Jupyter Notebook interface with the following content:

- A code cell with the following Python code:

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

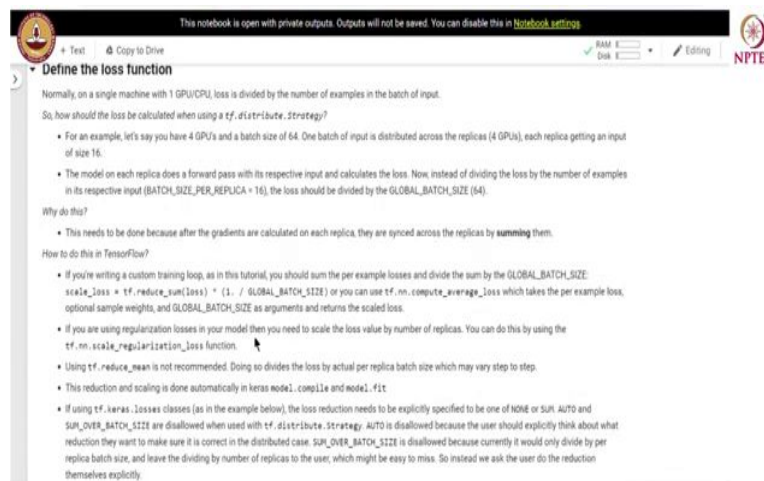
train_dist_dataset = strategy.experimental_distribute_dataset(train_dataset)
test_dist_dataset = strategy.experimental_distribute_dataset(test_dataset)
```
- A section titled "Create the model" with the instruction: "Create a model using `tf.keras.Sequential`. You can also use the Model Subclassing API to do this."
- A code cell with the following Python code:

```
[7] def create_model():
    model = tf.keras.Sequential([
        tf.keras.layers.Conv2D(32, 3, activation='relu'),
        tf.keras.layers.MaxPooling2D(),
        tf.keras.layers.Conv2D(64, 3, activation='relu'),
        tf.keras.layers.MaxPooling2D(),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(64, activation='relu'),
        tf.keras.layers.Dense(10, activation='softmax')
    ])
    return model
```
- A code cell with the following Python code:

```
# Create a checkpoint directory to store the checkpoints.
checkpoint_dir = './training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
```
- A section titled "Define the loss function" with the instruction: "Normally, on a single machine with 1 GPU/CPU, loss is divided by the number of examples in the batch of input."

Later we distribute the dataset across replicas, we create a model, we define a function to create the model it is a CNN model, we create a checkpoint directory to store the checkpoints.

(Refer Slide Time: 35:50)



The screenshot shows a Jupyter Notebook interface with the following content:

- A section titled "Define the loss function" with the instruction: "Normally, on a single machine with 1 GPU/CPU, loss is divided by the number of examples in the batch of input."
- A text block explaining how to calculate the loss in a distributed environment:

So, how should the loss be calculated when using a `tf.distribute.Strategy`?

 - For an example, let's say you have 4 GPUs and a batch size of 64. One batch of input is distributed across the replicas (4 GPUs), each replica getting an input of size 16.
 - The model on each replica does a forward pass with its respective input and calculates the loss. Now, instead of dividing the loss by the number of examples in its respective input (`BATCH_SIZE_PER_REPLICA = 16`), the loss should be divided by the `GLOBAL_BATCH_SIZE` (64).
- A text block explaining why this is done:

Why do this?

 - This needs to be done because after the gradients are calculated on each replica, they are synced across the replicas by **summing** them.
- A text block explaining how to do this in TensorFlow:

How to do this in TensorFlow?

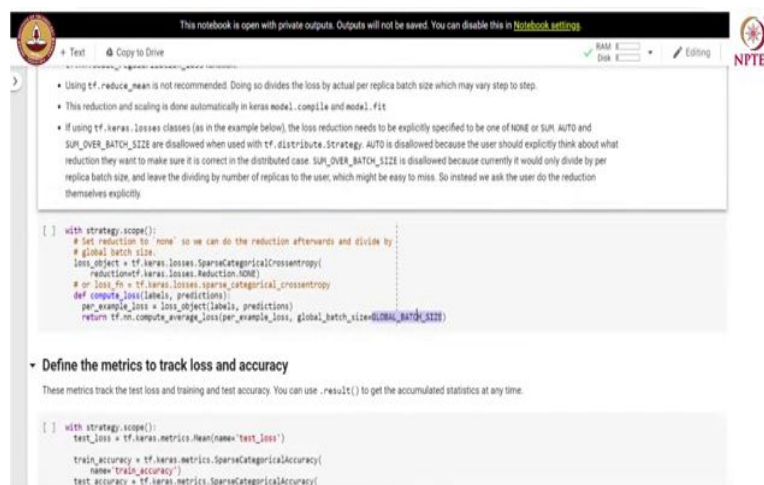
 - If you're writing a custom training loop, as in this tutorial, you should sum the per example losses and divide the sum by the `GLOBAL_BATCH_SIZE`:
`scale_loss = tf.reduce_sum(loss) * (1. / GLOBAL_BATCH_SIZE)` or you can use `tf.nn.compute_average_loss` which takes the per example loss, optional sample weights, and `GLOBAL_BATCH_SIZE` as arguments and returns the scaled loss.
 - If you are using regularization losses in your model then you need to scale the loss value by number of replicas. You can do this by using the `tf.nn.scale_regularization_loss` function.
 - Using `tf.reduce_mean` is not recommended. Doing so divides the loss by actual per replica batch size which may vary step to step.
 - This reduction and scaling is done automatically in `keras.model.compile` and `model.fit`.
 - If using `tf.keras.Losses` classes (as in the example below), the loss reduction needs to be explicitly specified to be one of `NONE` or `SUM`. `AUTO` and `SUM_OVER_BATCH_SIZE` are disallowed when used with `tf.distribute.Strategy`. `AUTO` is disallowed because the user should explicitly think about what reduction they want to make sure it is correct in the distributed case. `SUM_OVER_BATCH_SIZE` is disallowed because currently it would only divide by per replica batch size, and leave the dividing by number of replicas to the user, which might be easy to miss. So instead we ask the user do the reduction themselves explicitly.

Next we define a loss function, normally on a single machine with one GPU or CPU, loss is divided by number of examples in the batch of input. How should we calculate the loss while using the distributed strategy? For an example let us say we have 4 GPUs and a batch size of 16, 1 batch of input is distributed across the replicas; in this case there are 4 GPUs, each GPU

receives 16 inputs. The model on each replica there is a forward pass with it is respective inputs and calculates the loss.

Now, instead of dividing the loss by the number of examples in it is respective input, which is 16 in this case the loss should be divided by the global batch size which is 64. Why do we really do this? This needs to be done, because the gradients are calculated on each replica they are synced across replica by summing them. So, let us see how to do this in TensorFlow.

(Refer Slide Time: 37:14)



The screenshot shows a Jupyter Notebook interface with a black header bar containing a logo on the left and 'NPTEL' on the right. Below the header, there's a text area with two bullet points explaining why `tf.reduce_mean` is not recommended and how to correctly calculate the loss. Below the text, there are two code blocks. The first code block defines a custom loss function `compute_loss` that uses `tf.nn.compute_average_loss` to calculate the loss per example and then divides it by the global batch size. The second code block defines metrics for training and testing loss and accuracy using `tf.keras.metrics`.

```
[ ] with strategy.scope():
    # Set reduction to 'none' so we can do the reduction afterwards and divide by
    # global batch size.
    loss_object = tf.keras.losses.SparseCategoricalCrossentropy(
        reduction=tf.keras.losses.Reduction.NONE)
    # or loss_fn = tf.keras.losses.sparse_categorical_crossentropy
    def compute_loss(labels, predictions):
        per_example_loss = loss_object(labels, predictions)
        return tf.nn.compute_average_loss(per_example_loss, global_batch_size=GLOBAL_BATCH_SIZE)
```

Define the metrics to track loss and accuracy

These metrics track the test loss and training and test accuracy. You can use `.result()` to get the accumulated statistics at any time.

```
[ ] with strategy.scope():
    test_loss = tf.keras.metrics.Mean(name='test_loss')
    train_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(
        name='train_accuracy')
    test_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(
```

So, if you are writing a custom training loop, we should sum the per example losses and divide the sum by the global batch size. So, we define a scale loss where we divide the loss by the global batch size or we can use `tf.nn.compute_average_loss` which takes the per example loss optional sample weights and global batch size as arguments and returns the scaled loss.

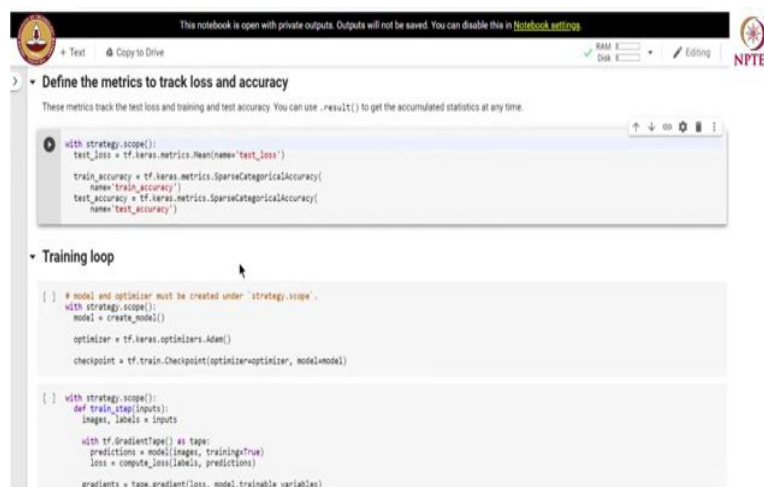
If you are using regularization loss in our model then we need to scale the loss value by the number of replicas. We do this by using `tf.nn.scale_regularization_loss` function. Using `tf.reduce_mean` is not recommended. Doing so, divide the loss by actual per replica batch size which may vary from step to step. This reduction and scaling is done automatically in `keras model.compile` and `model.fit` function.

If we are using `tf.keras.losses` classes; the loss reduction needs to be explicitly specified to be one of `NONE` or `SUM`. `AUTO` is disallowed and `SUM_OVER_BATCH_SIZE` is also disallowed, because currently it would only divide by per replica batch size and leave the division by the number of replicas to the user, which might be easy to miss. So, instead we ask the user to do the reduction themselves explicitly.

So, with `strategy.scope`, here we set reduction to `NONE`. So, we can do the reduction afterward and divide by the global batch size. So, we define sparse categorical cross entropy loss with reduction set to `none`.

We compute the loss using the loss object, we supply labels and the predictions and we reduce and we return `compute_average_loss`, where we take for example, loss and we average it using `global_batch_size` as specified over here.

(Refer Slide Time: 40:19)



The screenshot shows a Jupyter Notebook interface with the following content:

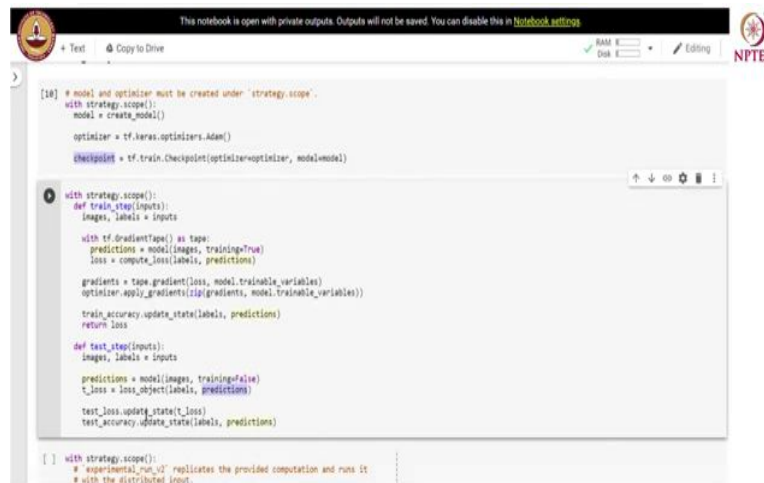
- Header: "This notebook is open with private outputs. Outputs will not be saved. You can disable this in [Notebook settings](#)."
- Section: "Define the metrics to track loss and accuracy". Description: "These metrics track the test loss and training and test accuracy. You can use `.result()` to get the accumulated statistics at any time."
- Code block 1:

```
with strategy.scope():  
    test_loss = tf.keras.metrics.Mean(name='test_loss')  
  
    train_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(  
        name='train_accuracy')  
    test_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(  
        name='test_accuracy')
```
- Section: "Training loop"
- Code block 2:

```
[ ] # model and optimizer must be created under 'strategy.scope'.  
with strategy.scope():  
    model = create_model()  
  
    optimizer = tf.keras.optimizers.Adam()  
    checkpoint = tf.train.Checkpoint(optimizer=optimizer, model=model)  
  
[ ] with strategy.scope():  
    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)
```

Let us define the metrics to track loss and accuracy. So, we again define this metrics with `strategy.scope`. So, here we are using the mean as a metric for test loss; we also use sparse categorical accuracy as another metric for training accuracy and test accuracy. And, we can use `.result` to get the accumulated statistics at anytime. Let us define the training loop; so, we defined a model and optimizer under `strategy.scope`. We create the model using `create_model` function, we define the optimizer and the checkpoint object.

(Refer Slide Time: 41:38)



```
[18] # model and optimizer must be created under 'strategy.scope'.
with strategy.scope():
    model = create_model()
    optimizer = tf.keras.optimizers.Adam()
    checkpoint = tf.train.Checkpoint(optimizer=optimizer, model=model)

with strategy.scope():
    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

    def test_step(inputs):
        images, labels = inputs

        predictions = model(images, training=False)
        t_loss = loss_object(labels, predictions)
        test_loss.update_state(t_loss)
        test_accuracy.update_state(labels, predictions)

[ ] with strategy.scope():
    # experimental_run_v2 replicates the provided computation and runs it
    # with the distributed input.
```

And, under the strategy scope we define a training step; a training step uses a gradient tape that records a forward computation and the loss computation and then we can get gradients with respect to the trainable variables of the model in the gradient list. Then we apply these gradients on the trainable variables and update their values. During the test time, we apply the forward computation, we supply images to the model and we get the prediction.

We calculate loss using the `loss_object` method that takes actual labels and predictions. And, then we update the test loss and the test accuracy based on the labels and predictions.

(Refer Slide Time: 42:54)



The screenshot shows a Jupyter Notebook interface with a code cell containing a Python function for distributed training and testing. The function is named `@tf.function` and `def distributed_train_test(dataset_inputs)`. It returns `strategy.experimental_run_v2(test_step, args=(dataset_inputs,))`. The function includes a `for` loop for epochs, a `for` loop for training steps, and a `for` loop for test steps. It calculates training loss and accuracy, and saves checkpoints. Below the code cell, there is a section titled "Things to note in the example above:" with three bullet points.

```
@tf.function
def distributed_train_test(dataset_inputs):
    return strategy.experimental_run_v2(test_step, args=(dataset_inputs,))

for epoch in range(EPOCHS):
    # TRAIN LOOP
    total_loss = 0.0
    num_batches = 0
    for x in train_dist_dataset:
        total_loss += distributed_train_step(x)
        num_batches += 1
    train_loss = total_loss / num_batches

    # TEST LOOP
    for x in test_dist_dataset:
        distributed_test_step(x)

    if epoch % 2 == 0:
        checkpoint.save(checkpoint_prefix)

    template = ("Epoch {}, Loss: {}, Accuracy: {}, Test Loss: {}, "
               "Test Accuracy: {}")
    print(template.format(epoch+1, train_loss,
                          train_accuracy.result()*100, test_loss.result(),
                          test_accuracy.result()*100))

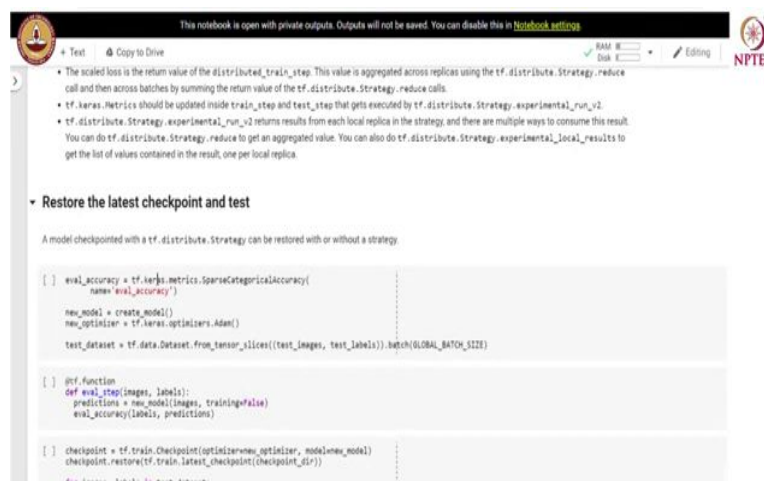
    test_loss.reset_states()
    train_accuracy.reset_states()
    test_accuracy.reset_states()
```

Things to note in the example above:

- We are iterating over the `train_dist_dataset` and `test_dist_dataset` using a `for x in ...` construct.
- The scaled loss is the return value of the `distributed_train_step`. This value is aggregated across replicas using the `tf.distribute.Strategy.reduce` call and then across batches by summing the return value of the `tf.distribute.Strategy.reduce` calls.
- `tf.keras.Metrics` should be updated inside `train_step` and `test_step` that gets executed by `tf.distribute.Strategy.experimental_run_v2`.

In the strategy scope we define a distributed training step and distributed test step; you are using tf functions over here. So, this is where the training is happening. In every epoch, you are performing distributed training, accumulating the loss and then calculating training loss, then performing the distributed test on the test data and then we are saving the checkpoint. So, at the end of the epoch we are asserting test loss training and test accuracies.

(Refer Slide Time: 44:00)



The screenshot shows a Jupyter Notebook interface with a code cell containing Python code for restoring a checkpoint and testing a model. The code includes a list of notes, a section titled "Restore the latest checkpoint and test", and a code block for creating a new model, optimizer, and dataset, and then testing it.

```
[ ] eval_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(
    name='eval_accuracy')

new_model = create_model()
new_optimizer = tf.keras.optimizers.Adam()
test_dataset = tf.data.Dataset.from_tensor_slices((test_images, test_labels)).batch(GLOBAL_BATCH_SIZE)

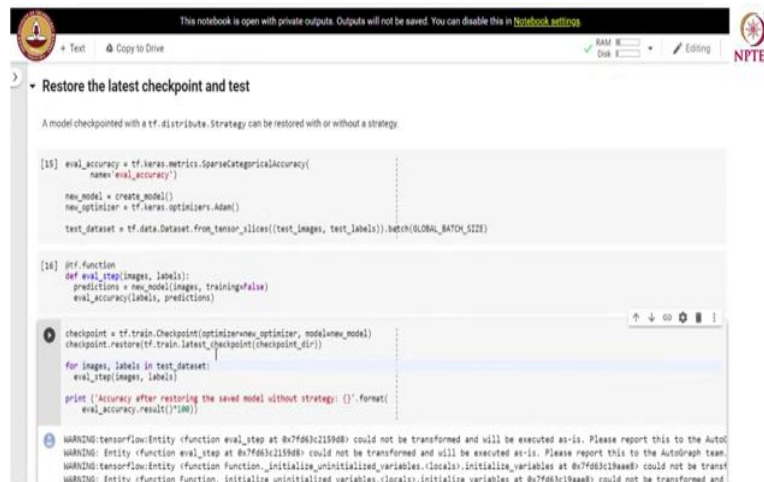
[ ] @tf.function
def eval_step(images, labels):
    predictions = new_model(images, training=False)
    eval_accuracy.update_state(labels, predictions)

[ ] checkpoint = tf.train.Checkpoint(optimizer=new_optimizer, model=new_model)
checkpoint.restore(tf.train.latest_checkpoint(checkpoint_dir))

for images, labels in test_dataset:
```

So, let us understand how to restore the model from the latest checkpoint. The model that we checkpointed with `tf.distribute.Strategy` can be restored with or without a strategy.

(Refer Slide Time: 44:17)



This notebook is open with private outputs. Outputs will not be saved. You can disable this in [Notebook settings](#).

RAM 8 GB
Disk 8 GB
Editing

NPTEL

Restore the latest checkpoint and test

A model checkpointed with a `tf.distribute.Strategy` can be restored with or without a strategy.

```
[15] eval_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(
    name='eval_accuracy')

new_model = create_model()
new_optimizer = tf.keras.optimizers.Adam()
test_dataset = tf.data.Dataset.from_tensor_slices((test_images, test_labels)).batch(GLOBAL_BATCH_SIZE)

[16] @tf.function
def eval_step(images, labels):
    predictions = new_model(images, training=False)
    eval_accuracy(labels, predictions)

checkpoint = tf.train.Checkpoint(optimizer=new_optimizer, model=new_model)
checkpoint.restore(tf.train.latest_checkpoint(checkpoint_dir))

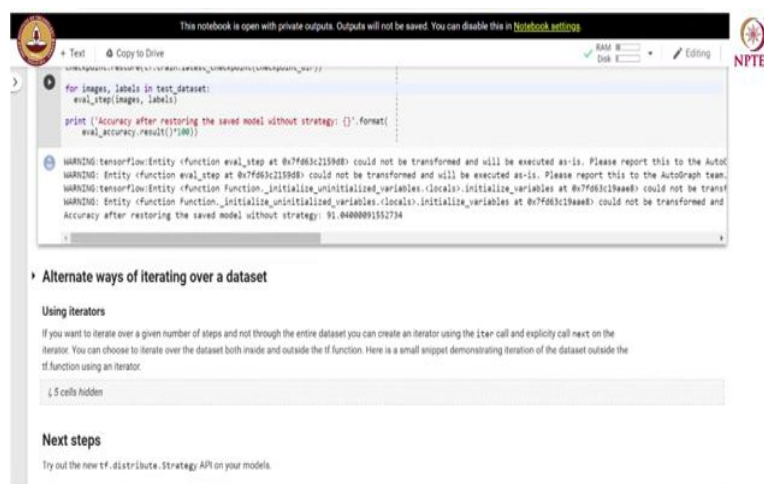
for images, labels in test_dataset:
    eval_step(images, labels)

print('Accuracy after restoring the saved model without strategy: {}'.format(
    eval_accuracy.result()*100))
```

WARNING:tensorflow:Entity <function eval_step at 0x7f063c2196d8> could not be transformed and will be executed as-is. Please report this to the AutoGraph team.
WARNING:tensorflow:Entity <function Function_eval_step at 0x7f063c2196d8> could not be transformed and will be executed as-is. Please report this to the AutoGraph team.
WARNING:tensorflow:Entity <function Function_initialize_uninitialized_variables.<locals>.initialize_variables at 0x7f063c19a6e0> could not be transformed and will be executed as-is. Please report this to the AutoGraph team.
WARNING:tensorflow:Entity <function Function_initialize_uninitialized_variables.<locals>.initialize_variables at 0x7f063c19a6e0> could not be transformed and will be executed as-is. Please report this to the AutoGraph team.

And, we can use the model to perform the inference on new data points. Now, we restore the model with the `restore` method and the `checkpoint` object.

(Refer Slide Time: 44:42)



This notebook is open with private outputs. Outputs will not be saved. You can disable this in [Notebook settings](#).

RAM 8 GB
Disk 8 GB
Editing

NPTEL

```
for images, labels in test_dataset:
    eval_step(images, labels)

print('Accuracy after restoring the saved model without strategy: {}'.format(
    eval_accuracy.result()*100))
```

WARNING:tensorflow:Entity <function eval_step at 0x7f063c2196d8> could not be transformed and will be executed as-is. Please report this to the AutoGraph team.
WARNING:tensorflow:Entity <function Function_eval_step at 0x7f063c2196d8> could not be transformed and will be executed as-is. Please report this to the AutoGraph team.
WARNING:tensorflow:Entity <function Function_initialize_uninitialized_variables.<locals>.initialize_variables at 0x7f063c19a6e0> could not be transformed and will be executed as-is. Please report this to the AutoGraph team.
WARNING:tensorflow:Entity <function Function_initialize_uninitialized_variables.<locals>.initialize_variables at 0x7f063c19a6e0> could not be transformed and will be executed as-is. Please report this to the AutoGraph team.

Accuracy after restoring the saved model without strategy: 91.04000091552734

Alternate ways of iterating over a dataset

Using iterators

If you want to iterate over a given number of steps and not through the entire dataset you can create an iterator using the `iter` call and explicitly call `next` on the iterator. You can choose to iterate over the dataset both inside and outside the `tf` function. Here is a small snippet demonstrating iteration of the dataset outside the `tf` function using an iterator.

5 cells hidden

Next steps

Try out the new `tf.distribute.Strategy` API on your models.

And, we can see that after restoring the model without strategy, we have an accuracy of 91.04 percent and we have a test accuracy of 90.25 percent at the end of training the model.

(Refer Slide Time: 44:53)

```

WARNING:tensorflow:Entity <function test_step at 0x7f0680260730> could not be transformed and will be executed as-is. Please report this to the AutoGraph team.
INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to (/job:localhost/replica:0/task:0/device:CPU:0).
INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to (/job:localhost/replica:0/task:0/device:CPU:0).
INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to (/job:localhost/replica:0/task:0/device:CPU:0).
INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to (/job:localhost/replica:0/task:0/device:CPU:0).
INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to (/job:localhost/replica:0/task:0/device:CPU:0).
INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to (/job:localhost/replica:0/task:0/device:CPU:0).
INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to (/job:localhost/replica:0/task:0/device:CPU:0).
INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to (/job:localhost/replica:0/task:0/device:CPU:0).
Epoch 1, Loss: 0.581279983997345, Accuracy: 81.78499803271484, Test Loss: 0.48195708873981367, Test Accuracy: 85.16899968482422
Epoch 2, Loss: 0.34157276153564453, Accuracy: 87.81590441486125, Test Loss: 0.34182852966637573, Test Accuracy: 88.82000427246094
Epoch 3, Loss: 0.2642809353168988, Accuracy: 89.338669921875, Test Loss: 0.3151452343138945, Test Accuracy: 88.2399978616983
Epoch 4, Loss: 0.2427893575926855, Accuracy: 90.47831251953125, Test Loss: 0.30578744412448586, Test Accuracy: 88.880004828125
Epoch 5, Loss: 0.2411284893751444, Accuracy: 91.1064665449414, Test Loss: 0.2781893637371083, Test Accuracy: 90.80000518798828
Epoch 6, Loss: 0.2287428216934204, Accuracy: 91.8866729763281, Test Loss: 0.274272888988495, Test Accuracy: 90.62000427246094
Epoch 7, Loss: 0.20459982752799888, Accuracy: 92.48999683271484, Test Loss: 0.252837167549333, Test Accuracy: 90.8499984741211
Epoch 8, Loss: 0.1877851748864781, Accuracy: 93.8583343585894, Test Loss: 0.25243687629699787, Test Accuracy: 91.08999633789062
Epoch 9, Loss: 0.17646413464141846, Accuracy: 93.55337907518797, Test Loss: 0.24824826410353485, Test Accuracy: 91.84000091952734
Epoch 18, Loss: 0.16167517066397247, Accuracy: 94.95166625978562, Test Loss: 0.266605289211273, Test Accuracy: 90.25
  
```

Things to note in the example above:

- We are iterating over the `train_dist_dataset` and `test_dist_dataset` using a `for x in ...` construct.
- The scaled loss is the return value of the `distributed_train_step`. This value is aggregated across replicas using the `tf.distribute.Strategy.reduce` call and then across batches by summing the return value of the `tf.distribute.Strategy.reduce` calls.
- `tf.keras.Metrics` should be updated inside `train_step` and `test_step` that gets executed by `tf.distribute.Strategy.experimental_run_v2`.

So, in this session we studied how to use distributed training on `tf.keras` API and on the custom training loop, we use mirrored strategy for synchronous training of the model in a distributed fashion.

(Refer Slide Time: 45:32)

Hardware platform: Users may want to scale their training onto multiple GPUs on one machine, or multiple machines in a network (with 0 or more GPUs each), or on Cloud TPUs.

In order to support these use cases, we have 5 strategies available. In the next section we will talk about which of these are supported in which scenarios in TF 2.0 at this time. Here is a quick overview:

Training API	MirroredStrategy	TPUStrategy	MultiWorkerMirroredStrategy	CentralStorageStrategy	ParameterServerStrategy
Keras API	Supported	Support planned in 2.0 RC	Experimental support	Experimental support	Supported since 2.0
Custom training loop	Experimental support	Experimental support	Support planned post 2.0 RC	Support planned in 2.0 RC	No support yet
Estimator API	Limited Support	Limited Support	Limited Support	Limited Support	Limited Support

MirroredStrategy

`tf.distribute.MirroredStrategy` supports synchronous distributed training on multiple GPUs on a single machine.

(Refer Slide Time: 45:35)

The screenshot shows the TensorFlow website's 'Distribution strategy' page. The left sidebar contains a navigation menu with categories like 'Essentials', 'Keras', 'Accelerators', and 'Data input pipelines'. The main content area features a table titled 'Training API' with columns for different strategies: 'MirroredStrategy', 'TPUStrategy', 'MultiWorkerMirroredStrategy', 'CentralStorageStrategy', and 'ParameterServerStrategy'. The table lists support for 'Keras API', 'Custom training loop', and 'Estimator API'. Below the table, there is a section for 'MirroredStrategy' with a link to the 'tf.distribute.MirroredStrategy' documentation.

Training API	MirroredStrategy	TPUStrategy	MultiWorkerMirroredStrategy	CentralStorageStrategy	ParameterServerStrategy
Keras API	Supported	Support planned in 2.0 RC	Experimental support	Experimental support	Supported since 2.0
Custom training loop	Experimental support	Experimental support	Support planned post 2.0 RC	Support planned in 2.0 RC	No support yet
Estimator API	Limited Support	Limited Support	Limited Support	Limited Support	Limited Support

Apart from synchronous mirrored strategy; we have other strategies and if you are more interested in learning about them, there is a distribution strategy guide available on the TensorFlow website. I would strongly encourage you to go through a couple of colabs for multi worker training.

The challenge with multi worker training is that we will have to set up this multi worker training in a cluster of machines or on cloud. So, if you are interested go through the colabs for the multi worker training and try to set it up on cloud for a practical experience. With this session we concluded our course, hope it was a great learning experience for you to learn practical machine learning with TensorFlow 2.0 with us.