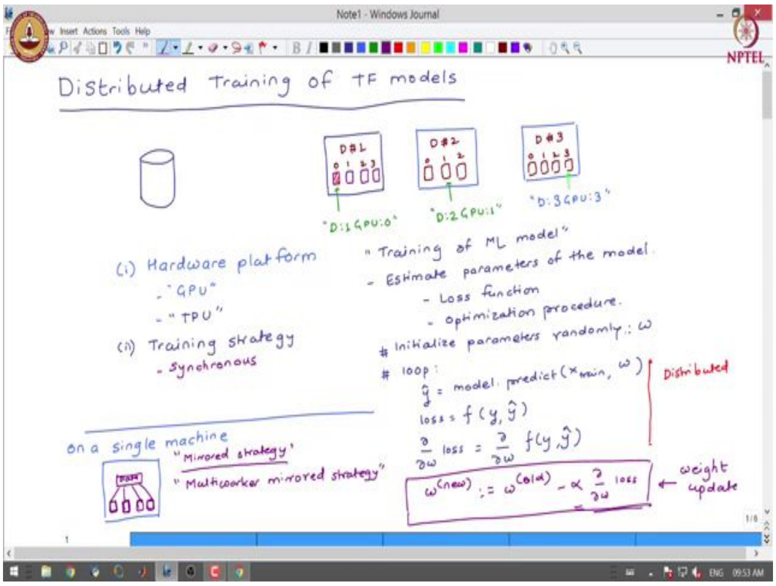
Practical Machine Learning with Tensorflow
Dr. Ashish Tendulkar
Department of Computer Science and Engineering
Indian Institute of Technology, Bombay

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 is 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.

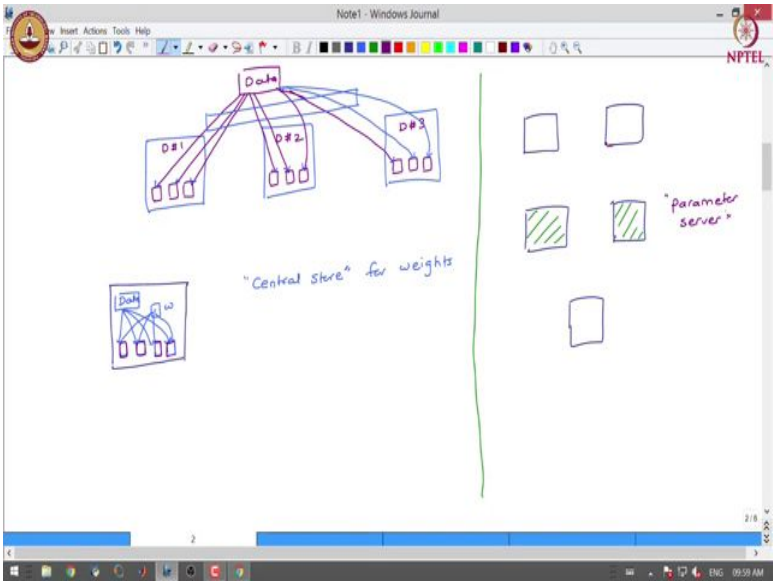
So, what we do is we initialize parameters, randomly or through some intelligent initialization strategy; after initializing the parameters we run a loop and inside the loop what we do is, we first find the predicted value. So, we use model.predict for the training examples with the weight vector that 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 radio 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.

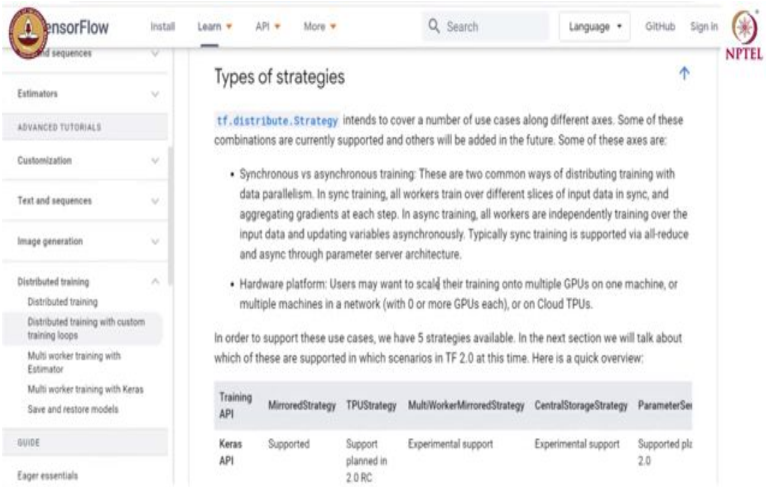
So, what we will do here is, we take the data we replicate it across GPUs; GPUs perform the gradient calculation and in order to do gradient calculation they read the values of the weight from the central store. And then gradient is combined from all the devices through all through all radio strategy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

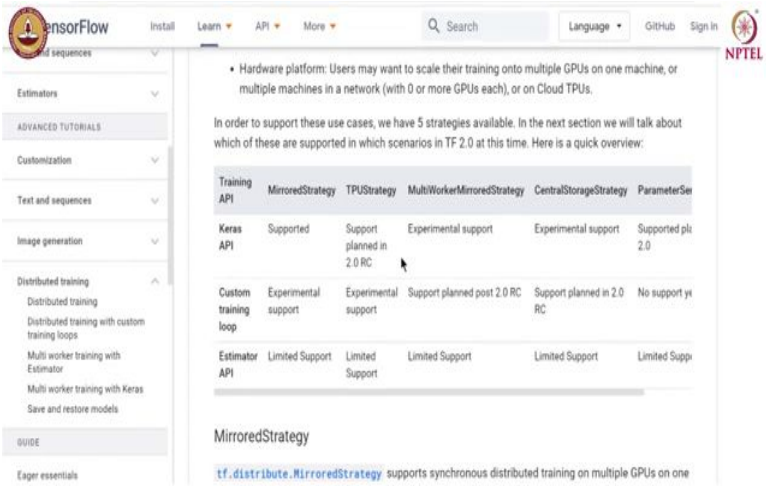
(Refer Slide Time: 18:15)



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 radios and asynch training is supported via parameter server architecture.

(Refer Slide Time: 19:05)

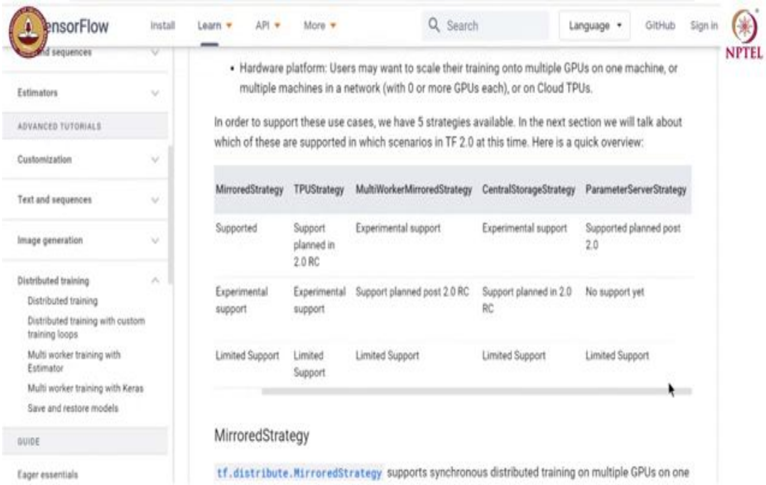


So, in all we have 5 strategies - mirrored strategy, TPU strategy, multi worker mirrored strategy, central storage strategy and parameter server strategy.

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.

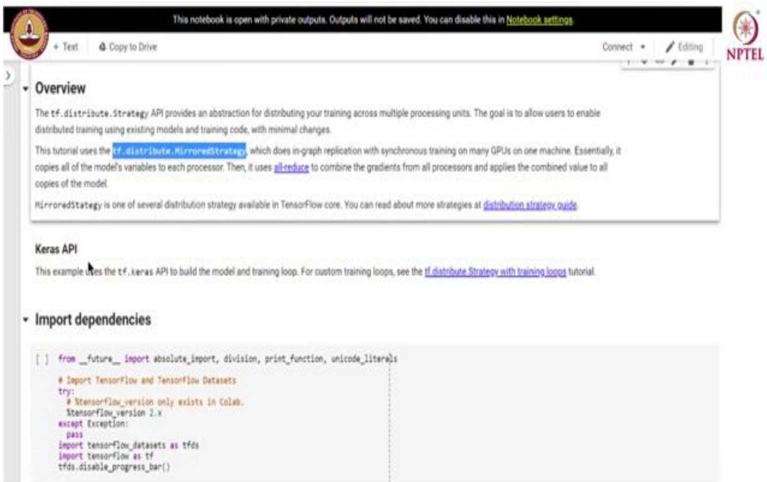
So, keras API supports mirrored strategy, the TPU strategy is planned in the release candidate of 2.0, multi worker strategy has got experimental support, central strategy has also got experimental support.

(Refer Slide Time: 20:05)



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.

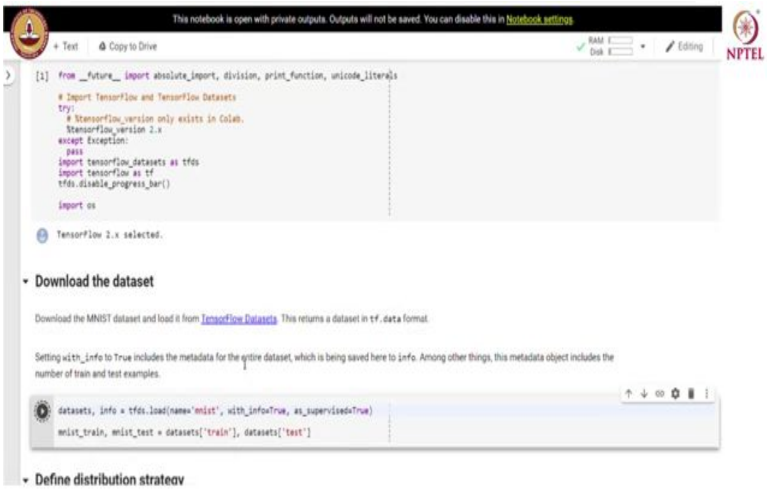
(Refer Slide Time: 20:53)



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.

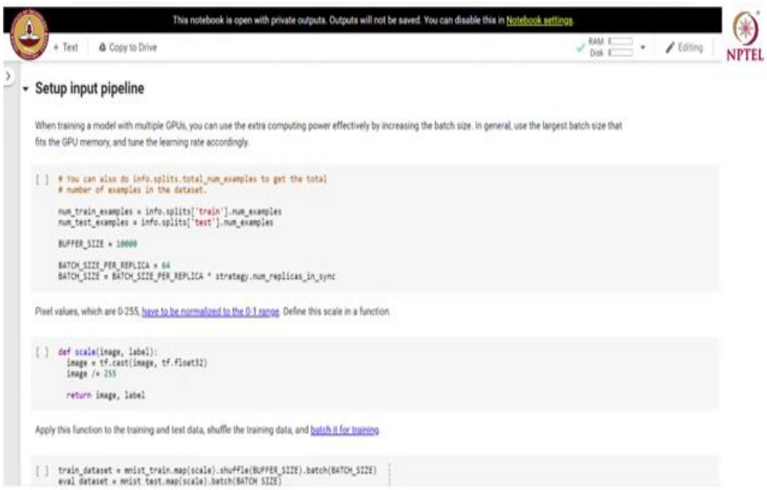
(Refer Slide Time: 22:06)



(Refer Slide Time: 22:12)

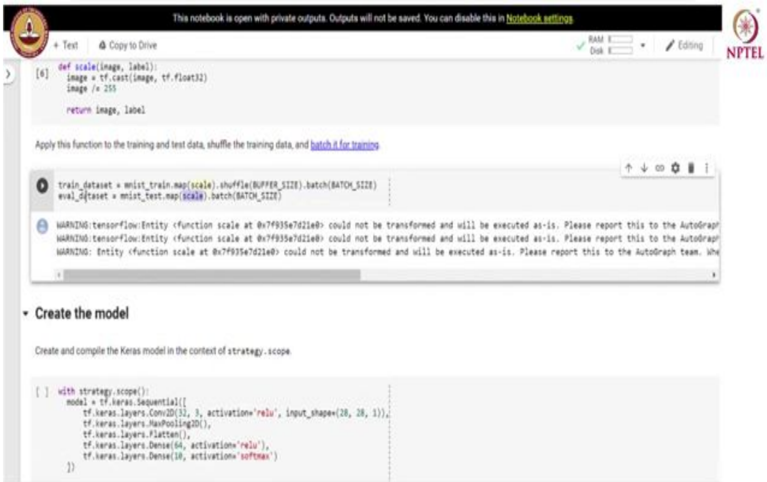
So, let us import all the dependencies, let us download the MNIST dataset. We will create a MirroredStrategy object through this statement; the mirrored strategy will handle distribution and provide a context manager to build our model. The context manager for mirrored strategy is tf.distribute.MirroredStrategy.scope.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

(Refer Slide Time: 22:55)



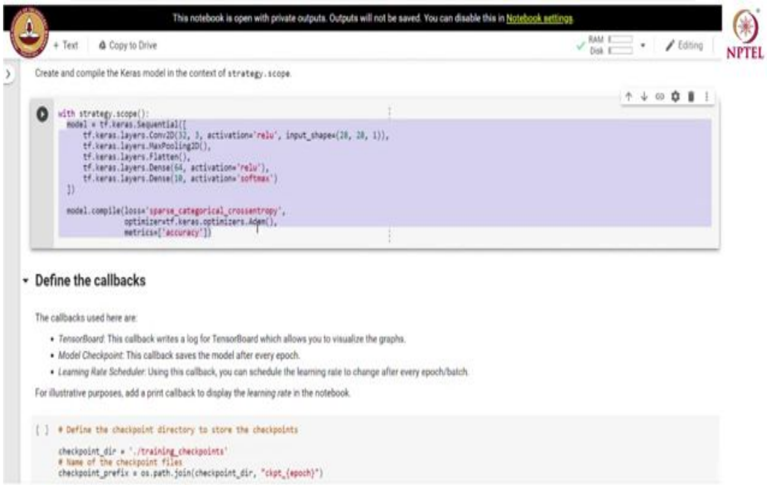
So, let us build input pipeline. When training a model with multiple GPUs we can use extra computing power effectively by increasing the batch size. In general use the largest batch size that fits the GPU memory and tune the learning rate accordingly. So, we have batch size equal to batch size per replica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

(Refer Slide Time: 23:29)



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)



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.

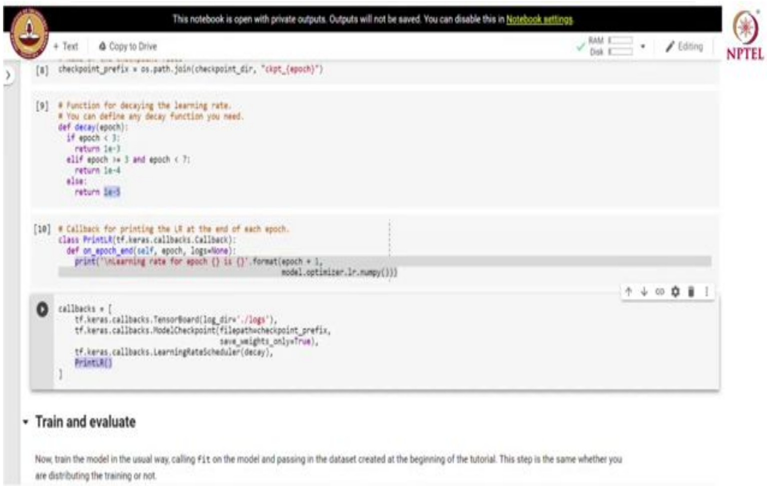
(Refer Slide Time: 24:46)

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.

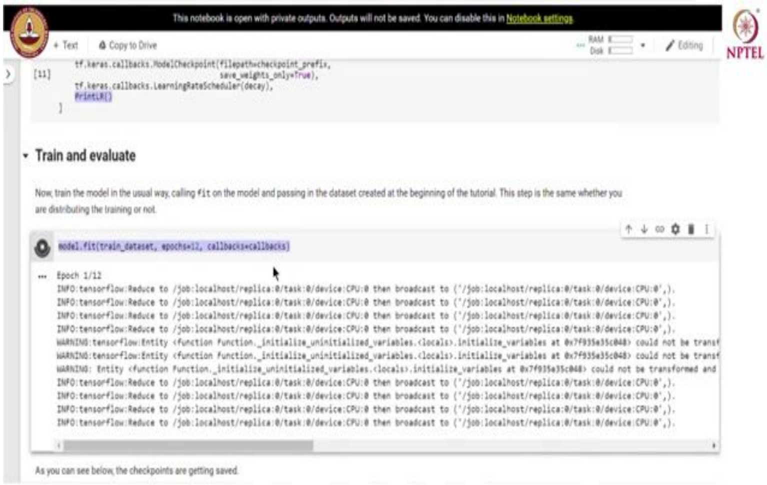
Then we define a callback for printing learning rate at the end of each epoch. So, we write on epoch end event, we capture this particular event and in at this event at the end of epoch we print the learning rate that was used for the epoch. So, if you want to learn more about callbacks there is a guide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

(Refer Slide Time: 26:41)



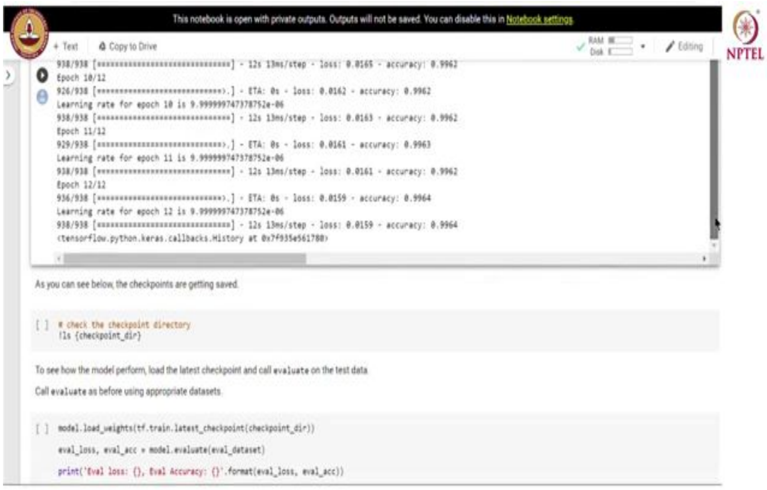
So, we put all callbacks in the callbacks list. So, we have used 3 inbuilt call callbacks and we have implemented one more callback for printing the learning rate at the end of each epoch.

(Refer Slide Time: 27:04)



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)

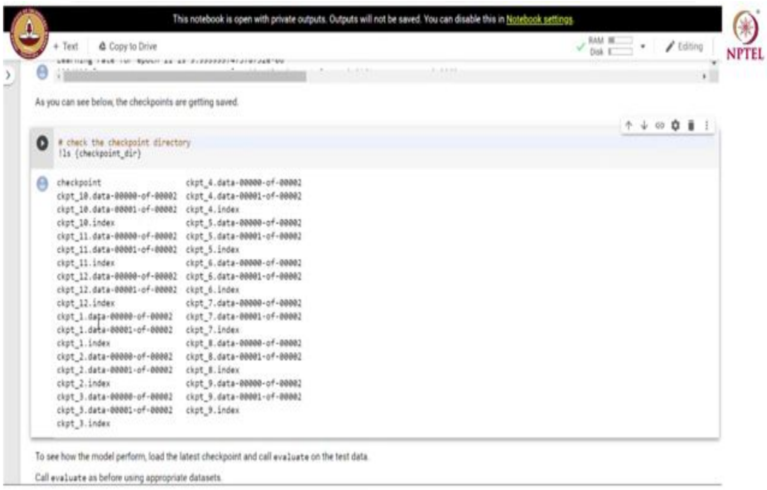


You can see that after 12 epochs the model has reached accuracy of 99.64 and you can see that it is printing the learning rate at the end of each epoch.

(Refer Slide Time: 28:05)

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.

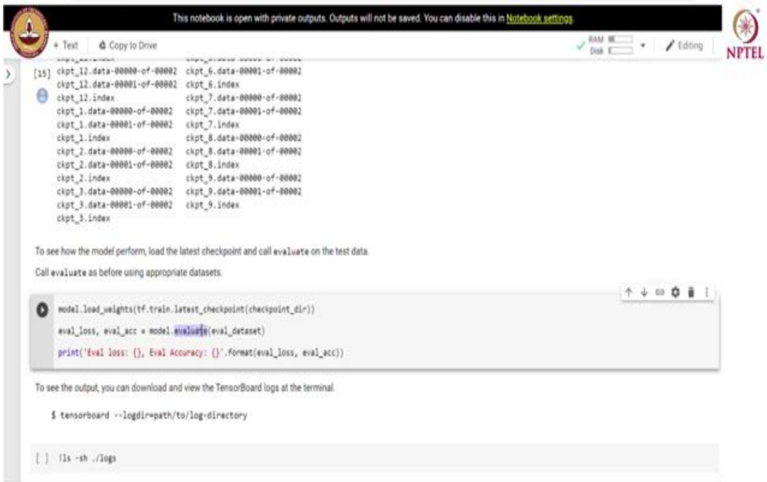
(Refer Slide Time: 28:39)



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.

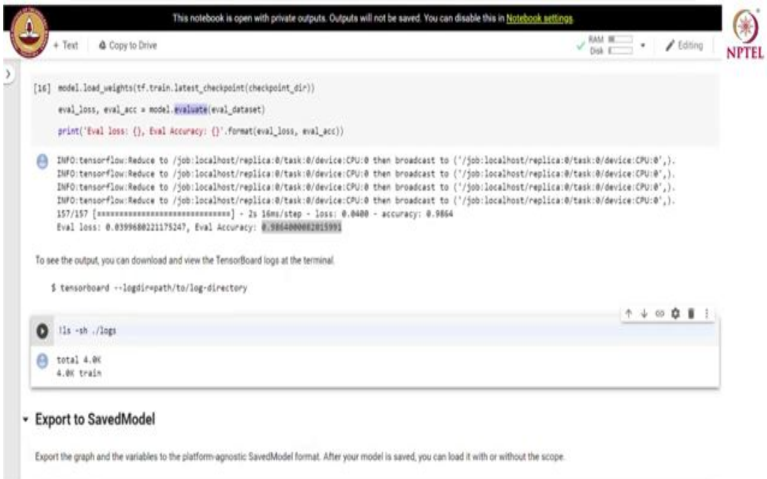
So, after every epoch you are having a single checkpoint. So, there are 12 checkpoints stored for 12 epochs for which we trained a model.

(Refer Slide Time: 29:06)



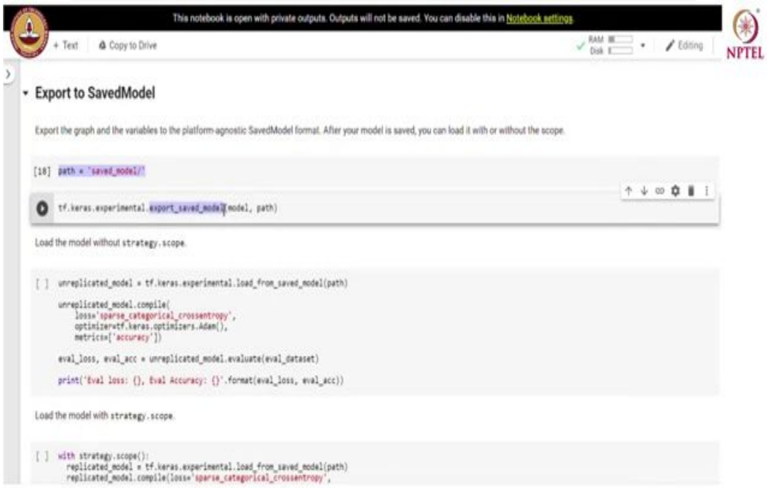
Let us check how the model performs. For that what we will do is, we load the model weights from the latest checkpoint from the checkpoint directory. And, then we will calculate the performance of the model by calling the evaluate function.

(Refer Slide Time: 29:28)



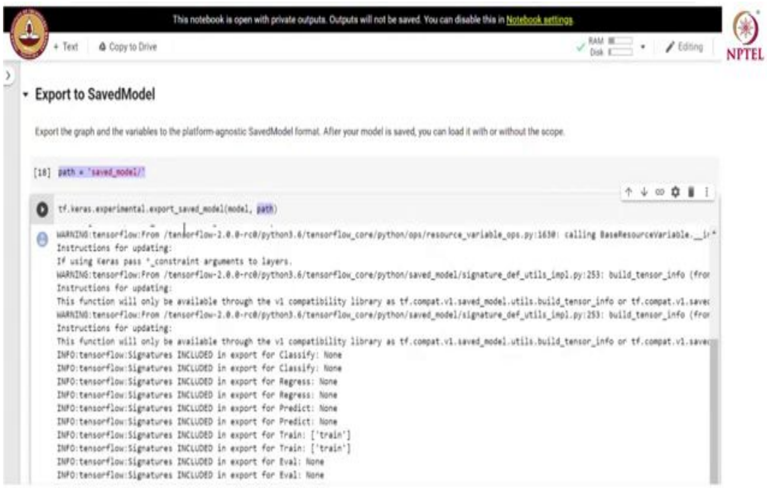
So, you can see that on the evaluation set we achieved 98.64 percent accuracy. Let us look at the log directory; this is where we have stored logs that can be read to TensorBoard.

(Refer Slide Time: 30:02)



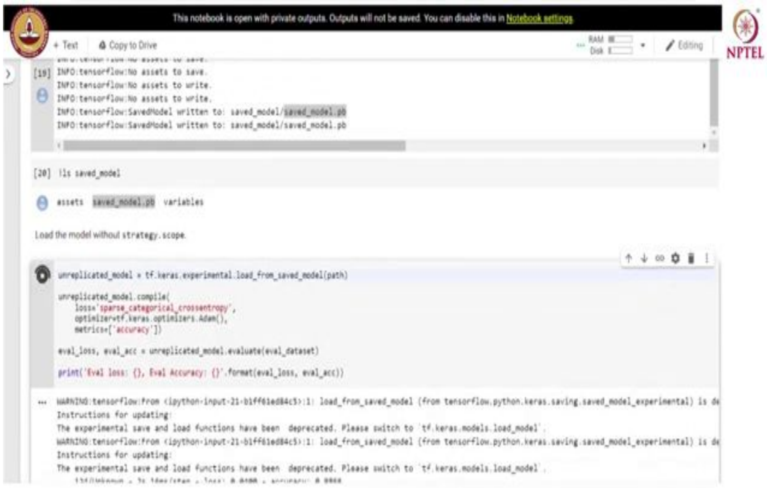
We can export the graph and the variables to the platform agnostic save model format, after the model is saved we can load it with or without the scope.

(Refer Slide Time: 30:30)



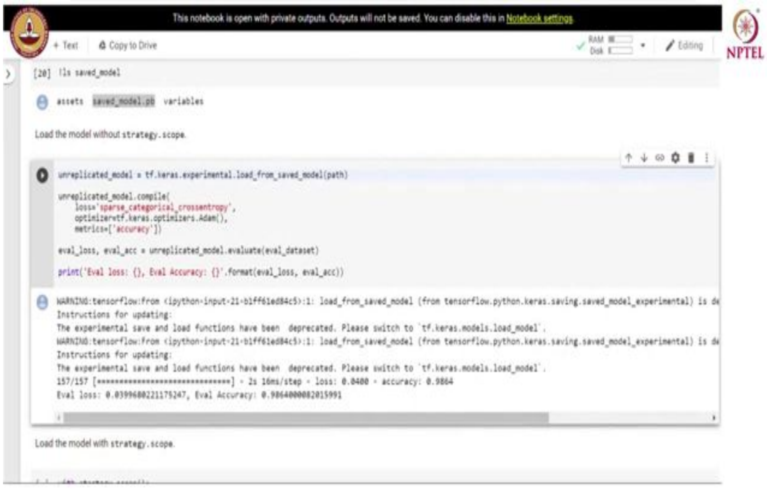
So, we specify path for saving the model and we use export_saved_model from the experimental version and save the model in the specified path.

(Refer Slide Time: 30:33)



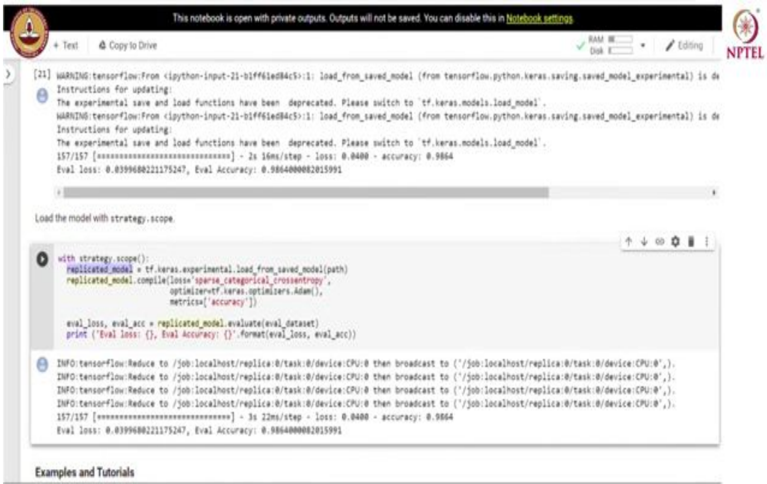
Let us check the content of the saved_model directory. And, you can see that the model has been saved in the saved_model directory. So, the model has been saved to saved_model.pb. Let us load the model without the strategy scope. This is a replicated model that was loaded from the saved model path. After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

(Refer Slide Time: 31:31)



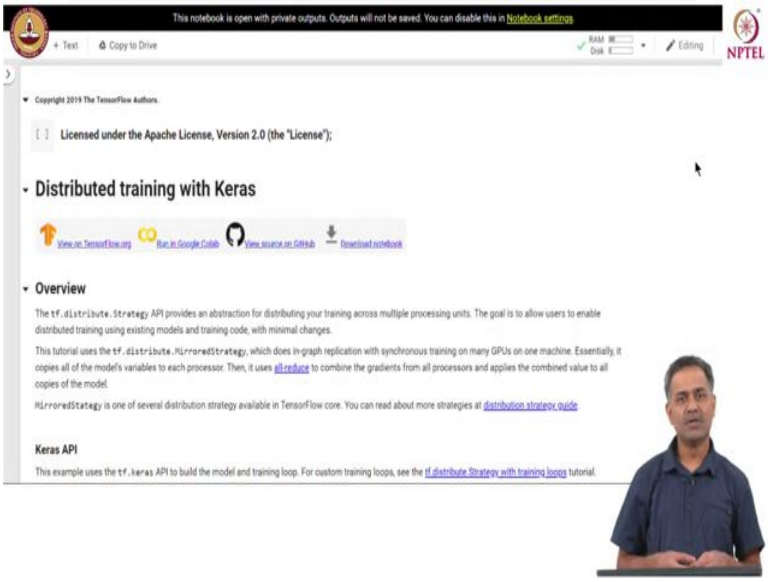
We can see that we achieve the same accuracy as before.

(Refer Slide Time: 31:35)



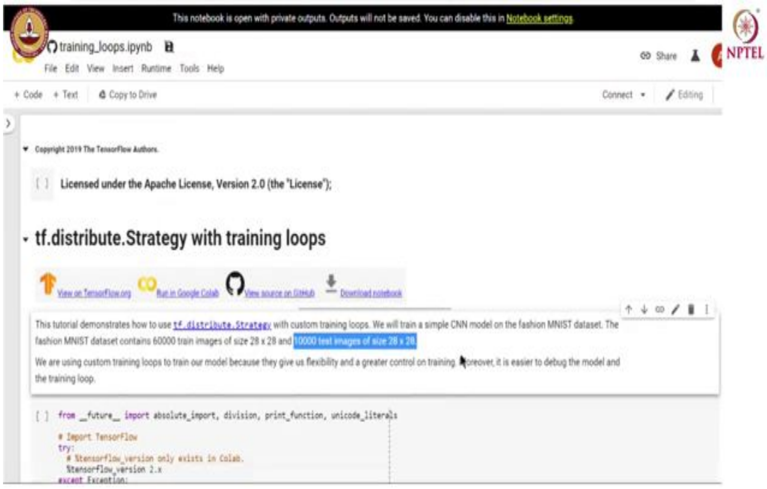
Let us load the model with strategy scope. So, we write with strategy scope everything else remains the same, we have changed the name of the model from the replicated model, we have changed it to replicated_model and so, when we evaluated the model we again achieved almost the same accuracy.

(Refer Slide Time: 32:06)



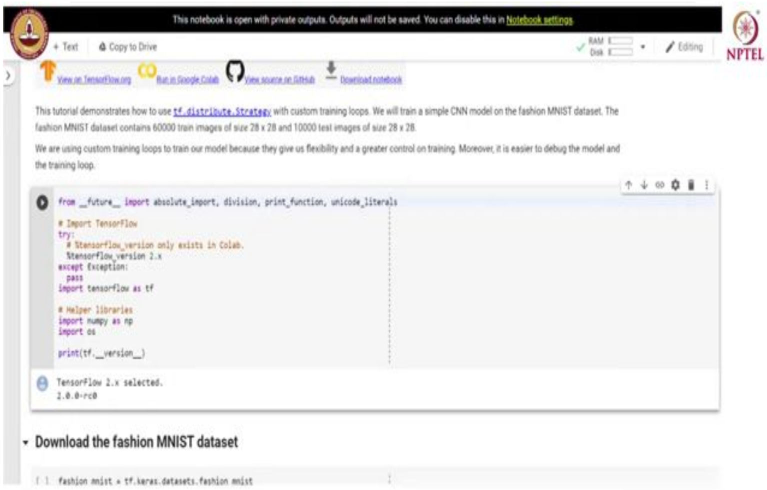
So, this is an example of how to use distributed training strategy with tf.keras API. So, here we use mirrored strategy for training keras model on MNIST dataset. Let us try to see how to use the distributed training strategy for custom training loops. So, we have seen that custom training loop provides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

(Refer Slide Time: 32:28)



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)



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)

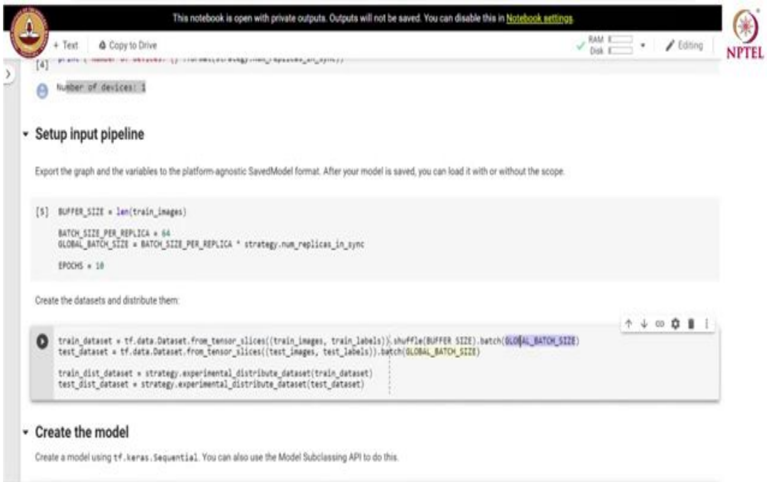
Let us download the fashion MNIST dataset.

(Refer Slide Time: 33:38)

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.

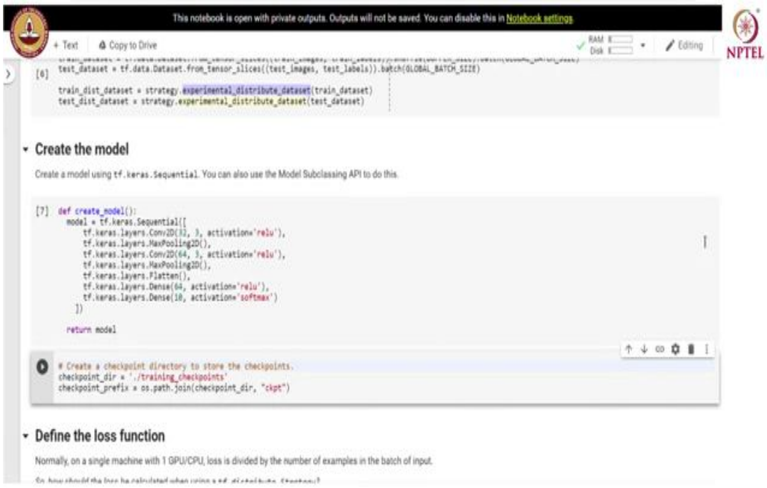
The gradients are synced across all replicas by summing them, after the sync the same update is made to the copies of variable on each replica. So, let us create a strategy object and this strategy object is a mirrored strategy. We can check the number of replicas that are in sync here. So, we have a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

(Refer Slide Time: 34:57)



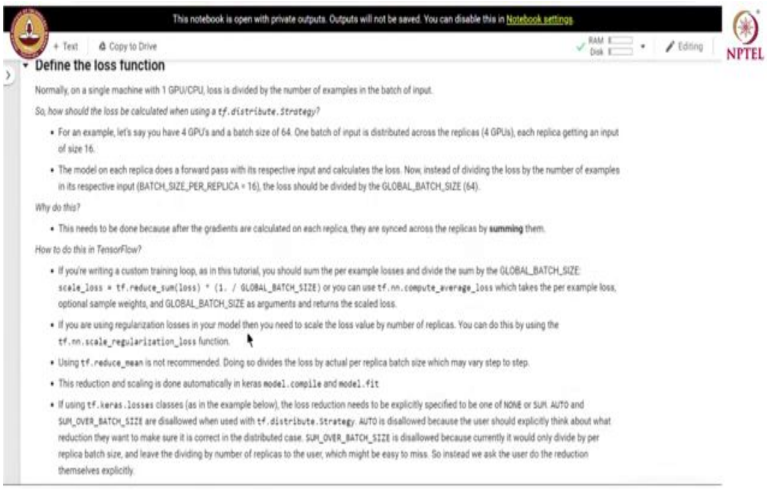
Let us build the input pipeline, let us create database, let us create the datasets and distribute them. So, we use the dataset.from_tensor_slices for creating the dataset. We shuffle it and then we batch according to the batch size specified over here.

(Refer Slide Time: 35:29)



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.

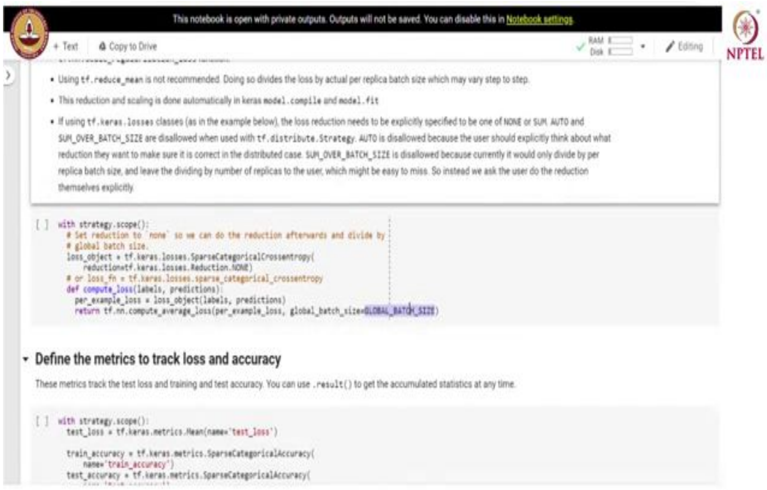
(Refer Slide Time: 35:50)



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)



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.

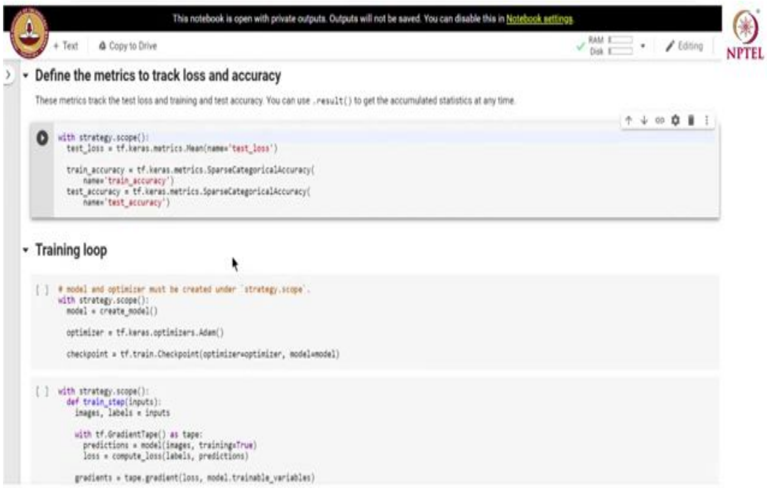
If you are using regularization loss in our model then we need to scale the loss value by the number of replicas. We do this by using tf.nn.scale_regularization_loss function. Using tf.reduce_mean is not recommended. Doing so, divide the loss by actual per replica batch size which may vary from step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

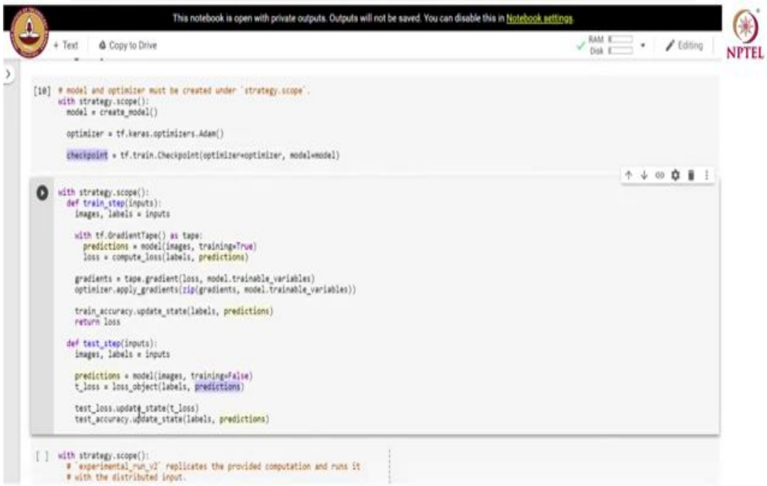
We compute the loss using the loss objet, we supply labels and the predictions and we reduce and we return compute_average_loss, where we take for example, loss and we average it using global_batch_size as specified over here.

(Refer Slide Time: 40:19)



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)



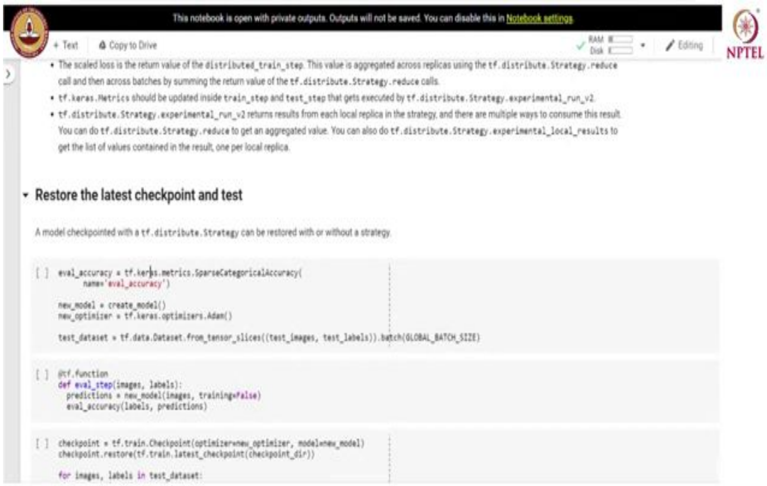
And, under the strategy scope we define a training step; a training step uses a gradient tape that records a forward computation and the loss computation and then we can get gradients with respect to the trainable variables of the model in the gradient list. Then we apply these gradients on the trainable 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)

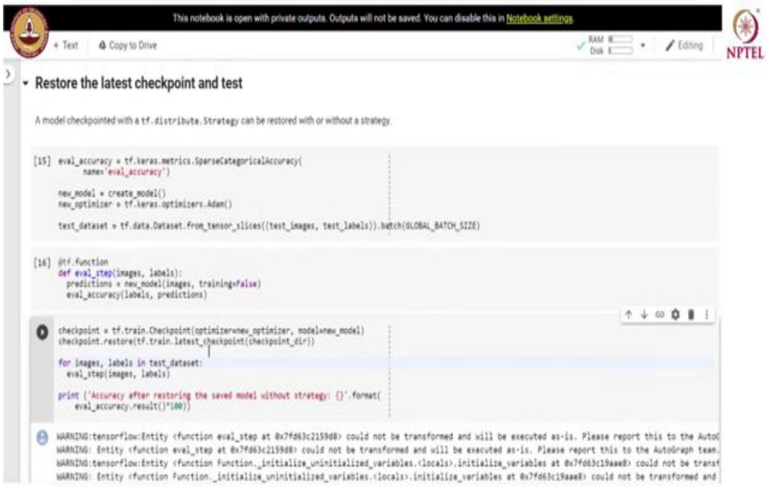
In the strategy scope we define a distributed training step and distributed test step; you are using tf functions over here. So, this is where the training is happening. In every epoch, you are performing distributed training, accumulating the loss and then calculating training loss, then performing 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)



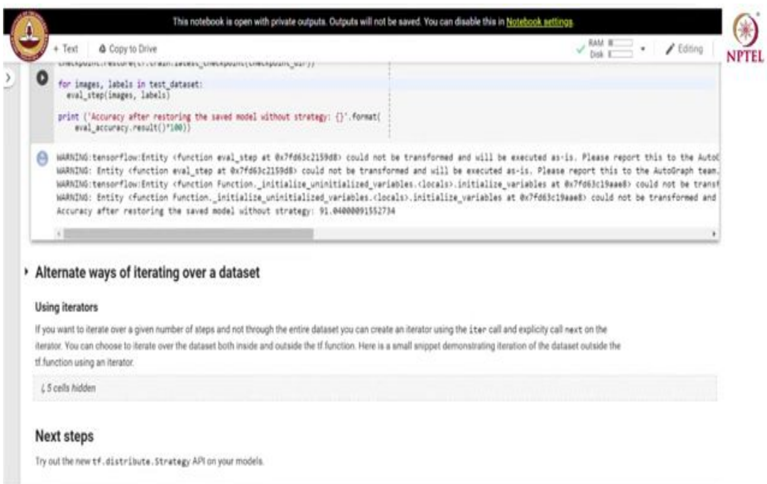
So, let us understand how to restore the model from the latest checkpoint. The model that we checkpointed with tf.distribute.strategy can be restored with or without a strategy.

(Refer Slide Time: 44:17)



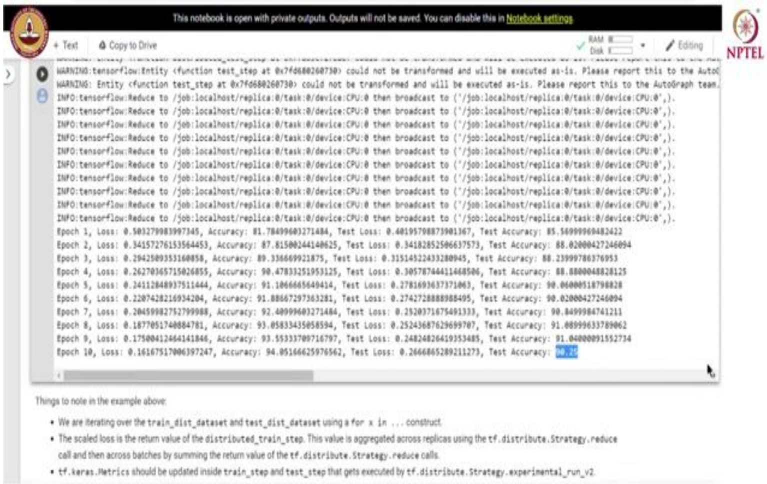
And, we can use the model to perform the inference on new data points. Now, we restore the model with the restore method and the checkpoint object.

(Refer Slide Time: 44:42)



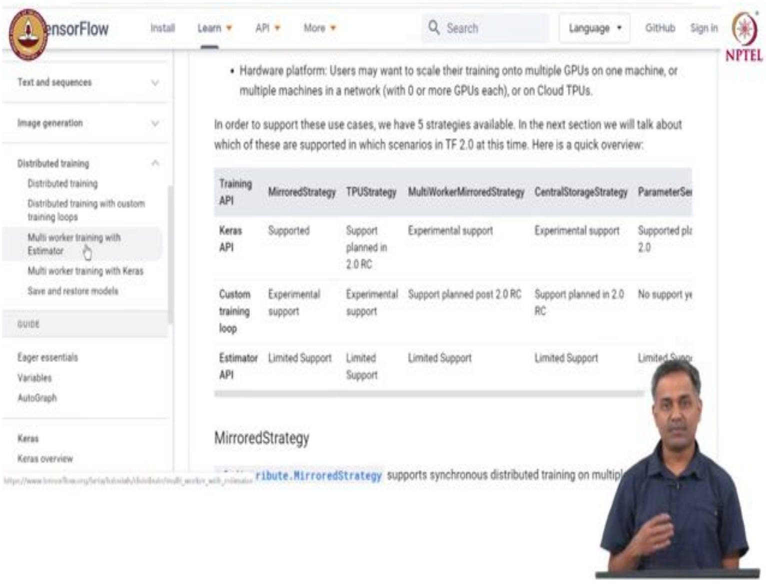
And, we can see that after restoring the model without strategy, we have an accuracy of 91.04 percent and we have a test accuracy of 90.25 percent at the end of training the model.

(Refer Slide Time: 44:53)

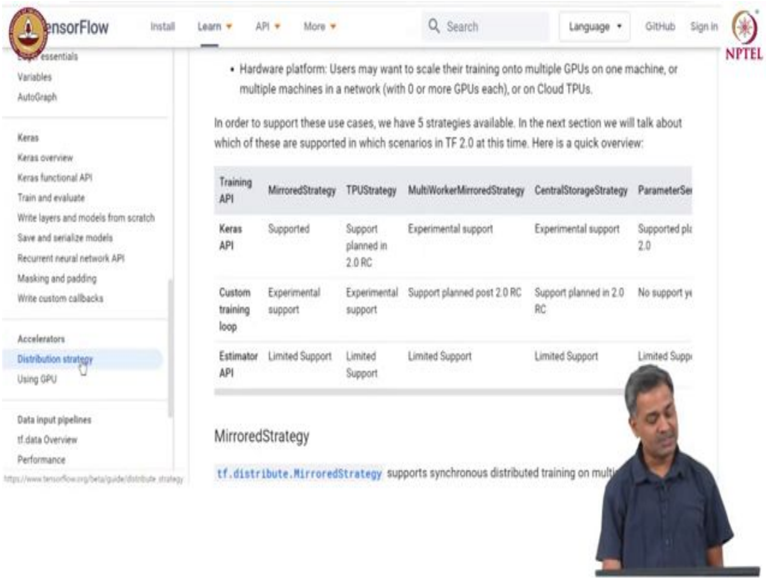


So, in this session we studied how to use distributed training on tf.keras API and on the custom training loop, we use mirrored strategy for synchronous training of the model in a distributed fashion.

(Refer Slide Time: 45:32)



(Refer Slide Time: 45:35)



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.