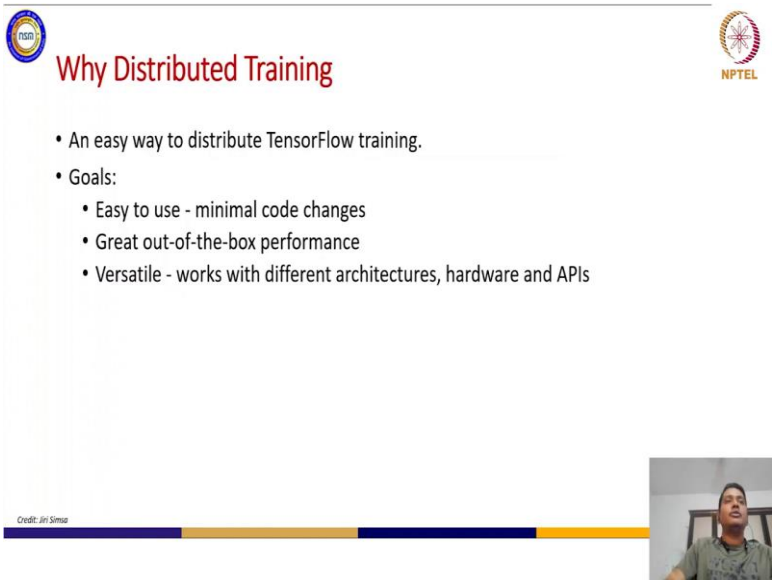
Applied Accelerated Artificial Intelligence
Dr. Satyajit Das
Department of Computer Science and Engineering
Indian Institute of Technology, Palakkad

Lecture - 27
Accelerated TensorFlow Part - 2

(Refer Slide Time: 00:15)

(Refer Slide Time: 00:17)



Next, we will go forward with the distributed training; so, as I was mentioning that it is very easy to distribute the training in tensor flow. But why we are using that that we need to know because the environment you are working with you need some enhanced

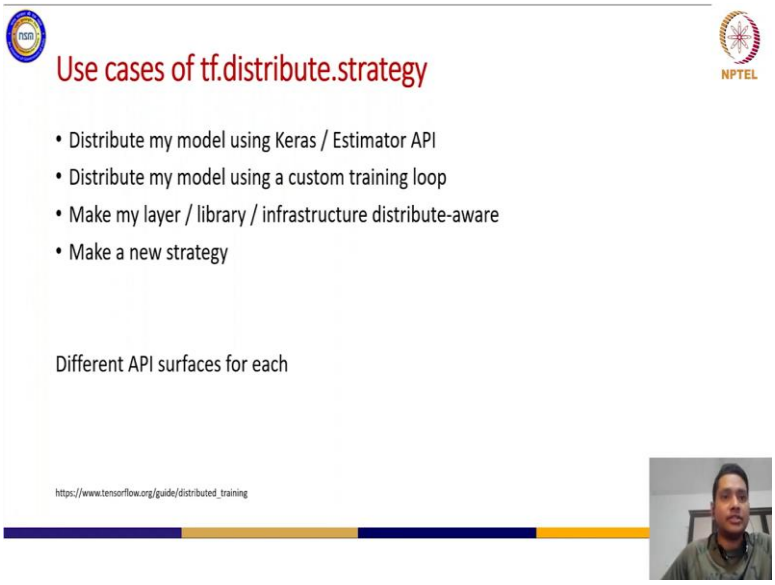
performance in the end. But, but actually you need to also know the environment that you are working with right. So, the ideal performance you want to get let us say very fine tuningly managing the entire pipeline of your training defining.

So, we were mentioning something about this line in the last class that you can a fine tune or finely manage which step will go to which GPU that is available for your a system, but distributed APIs if you are using that you will get out of the box performance which is essentially kind of equivalent if you are doing that fine tuning. Wow what does that mean; that means that using this simple API just one line of code you can distribute the training. But you will get the same performance as if you are fine tuning the each step of your training pipeline.

Now, how it is happening we will not go into that details? Because in the back end as I was mentioning it is called the code is actually developed with C++. And it is very close to system you are the that language is very close to your system. So, the performance you will get with the C++ backend is much higher than Python right.

So, that way the using these APIs you will get the same performance as if you are finding. Now, using this APIs also is versatile in nature, because this the same strategy you can follow for different architectures different hardware different APIs as well as a combination of that ok, so, we will see a few instances of that.

(Refer Slide Time: 02:50)

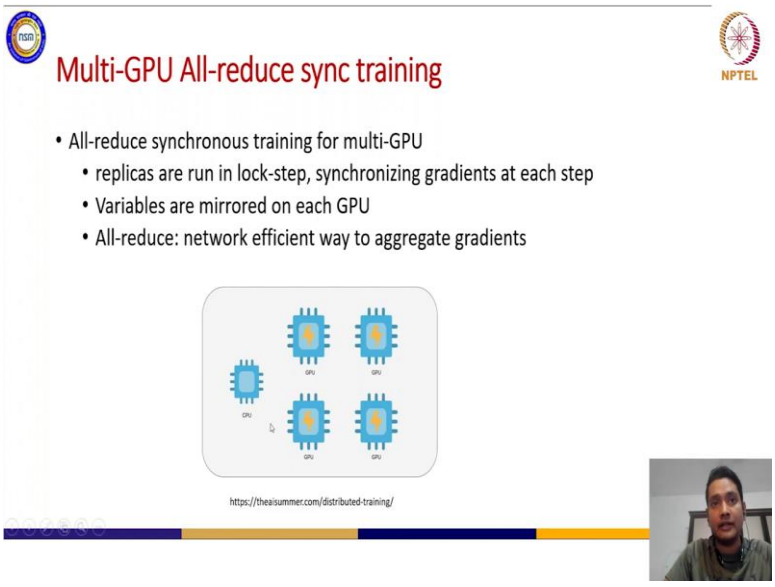


Now, distributed strategy essentially is thus the main task for that distributed.strategy and this strategy is used to distribute any model using Keras API or estimator API. Now, again in the previous class; so, you just follow the sequences in the previous class; we have said that you have you can define, your models using Keras API or estimator API, but estimated API is not being maintained nowadays.

So, you can use only Keras API for efficient distributed manner. Now, distributed my model is also possible using a custom training loop this is very simple and also to get much more flexibility ok. So, you can make your layer your library your infrastructure distribute aware, what is distribute aware we will see. But you can make your different stage of defining your a model you can define which will be the distributed aware module for your case; so, this is much more control.

So, tensor flow is all about abstracting everything to opening everything to you; so, that means, high flexibility and high controllability ok; so, both you can give. Make a new strategy if you want a new strategy to distribute your training process that also you can make it. And that is a also quite valuable if you are targeting let us say multi accelerator things. But again, some APIs are already there which will help you to take advantage of different accelerators that are available in your system yeah, different API surfaces for each of the model that you see now.

(Refer Slide Time: 05:04)

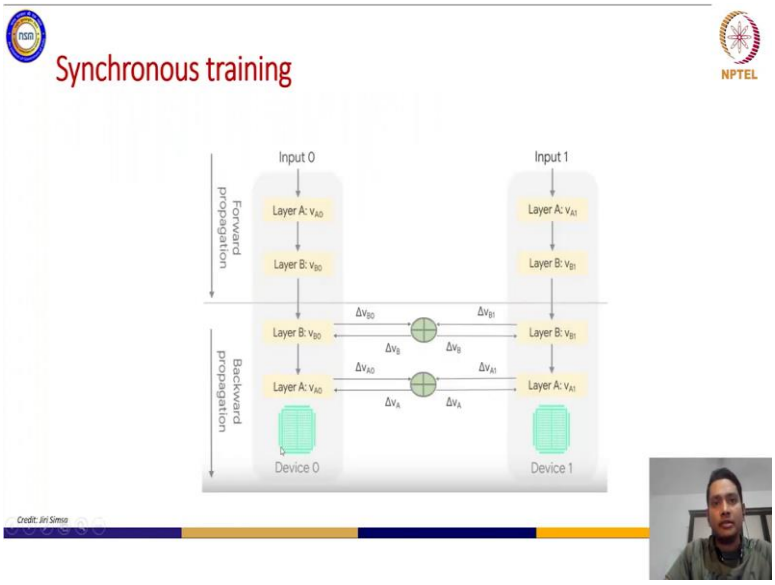


Now, this strategy that first strategy we will talk about is the mirrored strategy; so, mirrored strategy is the multi GPU all reduce sync training. So, it is all about synchronized training, synchronized training means each of the devices where you are distributing the processes will be locked and executed. So, and all the APIs are maintaining everything it for you do not need to worry about the global interrupt lock; so, it is very very efficient.

So, replicas of the variables that you want to a distribute that will be actually made replicas and it will synchronize again at the gradient stage so, when you are trying to compute the gradient then it will meet again. And so, this is all stage where you are actually distributing your variables to or making replicas in each device. So, basically let us say you have one system where you have one CPU and multiple GPUs you want your replicas; that means, replica of the variables that you want to distribute will be available to all GPUs that is all.

And in the reduced stage you actually synchronize right; so, this is all reduce and sync is essentially doing in lock step processes. Variables are mirrored in each GPU as we were discussing here, and all we use this algorithm will actually do the aggregation of the gradients how it is doing it.

(Refer Slide Time: 06:44)



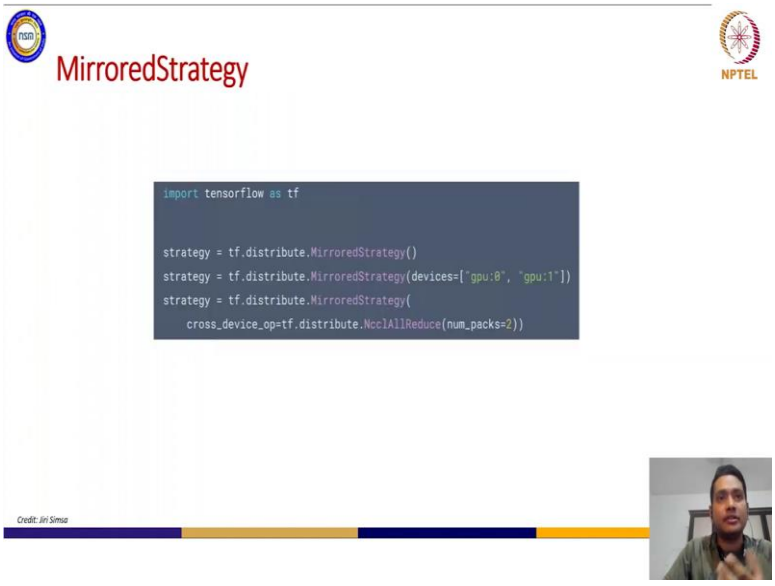
So, let us say we have this one device and second device here and first device we are giving the input here the input 0 and another input 0 input 1. Now, here we have the

same layers A B and B A that is the backward pass and this is the forward pass. Now, I want to replicate the V_A; so, V_{A0} and V_{A1} has been two instances have been created which is replicated actually in the two deviator two devices here. Then the output for the second layer is V B 0, V B 1, then we are doing the backward propagation.

So, now we want to have this V_{B0} and V_{B1} to be met in the middle somewhere where it will be aggregated and again. So, basically this is the reduced stage where it is reduced the variables is all variables will be reduced and then will be distributed again. And then again, the gradient combination will happen and again it will need to get the average of the data, which will help you to improve the to update the weights for particular devices.

So, V_A ΔV_A which is computed here will be broadcasted to all these GPUs; so, this is the synchronous training; so, you see here each. So, this is locked also this is this process is also locked and you will until or unless you get this here you do not know you cannot do the register; so, this is a synchronous training.

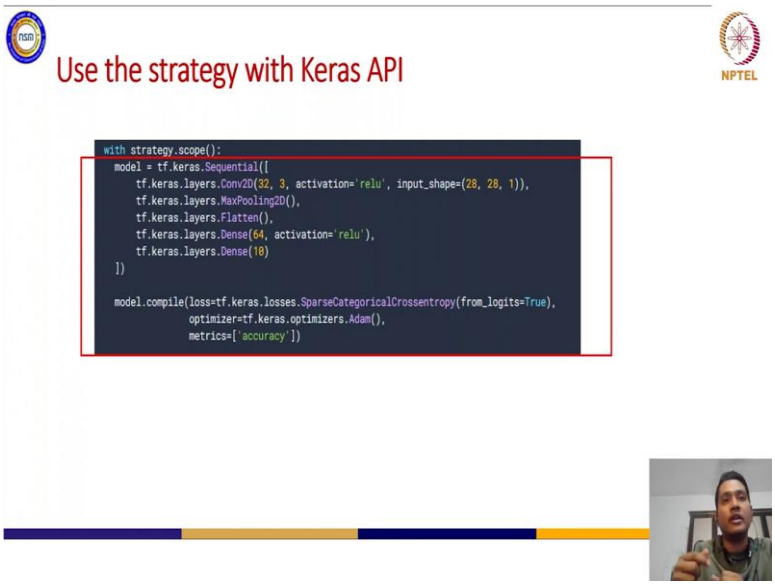
(Refer Slide Time: 08:23)



Now, all reduce synchronous training how you can define it the strategy. So, mirrored strategy; that means, making replicas replica of variables, how we can define? This is one way of defining strategy is equal to tf.distribute.MirroredStrategy. Mirrored strategy is very simple a function which will distribute your or mirror your variables to all the devices that is there. If you want to particularly define, which are the devices that you want to work with, you can also give the arguments which are the particular devices gpu

0, gpu 1 in this case. And also, you can define which reduce algorithm will be used you can define that in the algorithm, these are the three ways to define the strategy.

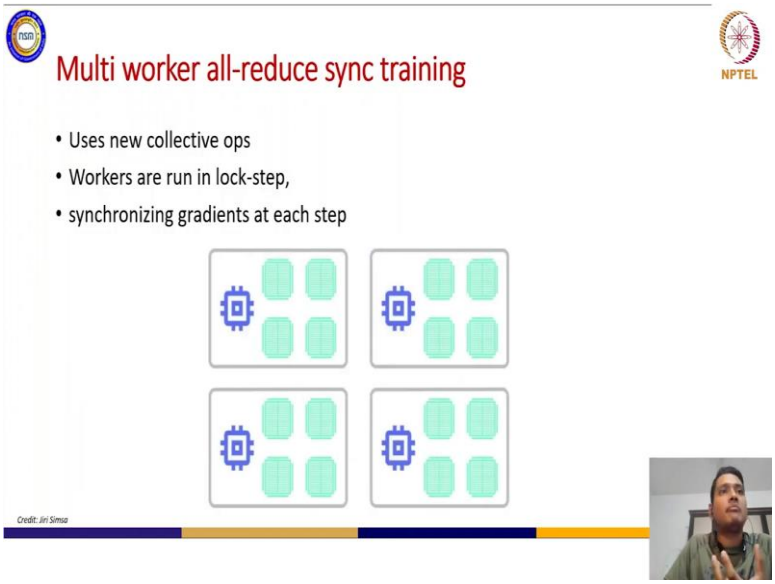
(Refer Slide Time: 09:16)



Once you have defined the strategy, you can go to your model definition and completion this is the sequential way of defining the model that we have seen so far. And to add like this these variables which are used in these layers will be distributed to your machines, if I want to specify that we will just wrap this with this strategy.scope() that is it. So, you define the strategy you wrap, your entire training module inside the strategy.scope().

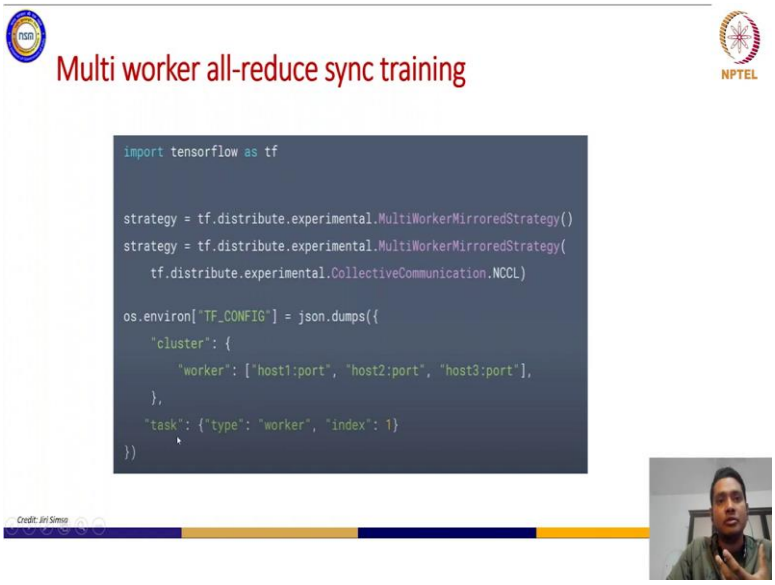
So, this particular block is under this strategy scope that is it simple, and your you will have the mirrored strategy with already is that that is simple. And this particular block which you will see in this red box is essentially the distributed aware box or distributed aware block. So, distribute aware block you can also define that you have seen so far. So, this is the distribute aware block, because now backend knows that this is the block where, I will mirror the values.

(Refer Slide Time: 10:37)



We can have multiple worker; so, that was for single worker and multiple devices, you will have multiple worker multiple devices. And in this case tensor flow collective ops will be updated, again the thing will be synchronized in nature and what. So, workers will run in lock state and you will synchronize the gradients in the in each step ok. So, each step will be distributed to multiple worker multiple devices whatever you define; so, worker which will do the things ok and the this is again synchronous in nature.

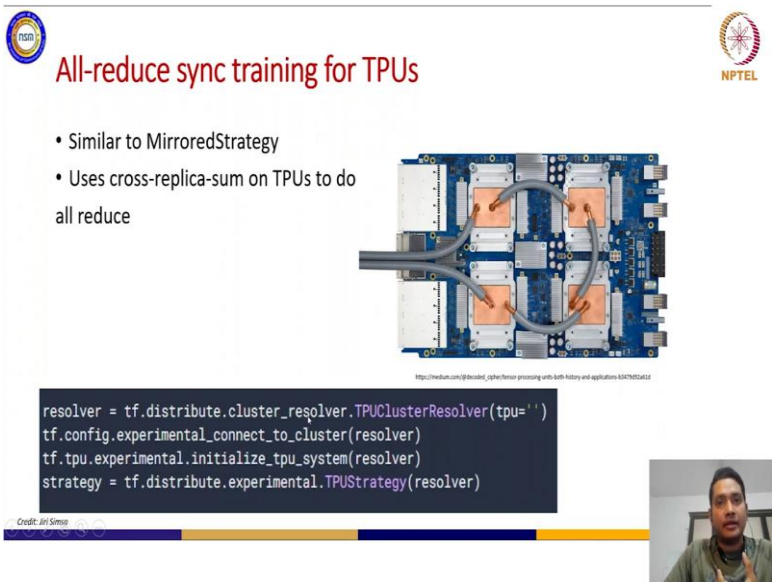
(Refer Slide Time: 11:22)



So, how to define that you want to distribute use the multiple worker mirrored strategy so, this is coming under the experimental module; so, this is just the experiment experimental module, but it will be first part to your a mainstream module in any time ok. But you can use this tf.distribute.experimental.MultiWorkerMirroredStrategy() to define one mirror strategy, which is we will be creating one scope for your distributed module, which will be distributed to multiple worker and multiple devices.

You can also define which back end communication will be happening NCCL can define ok nice. Then you can you also need to define the workers the basically, which will be the task type and the workers and that will be given as json string form ok. So, cluster the specification for your nodes here this will be the worker and this will be. So, this is simple usage of our multi worker mirrored strategy where you can have multiple worker working tandem in a synchronous manner in all the steps will be locked and executed.

(Refer Slide Time: 12:53)



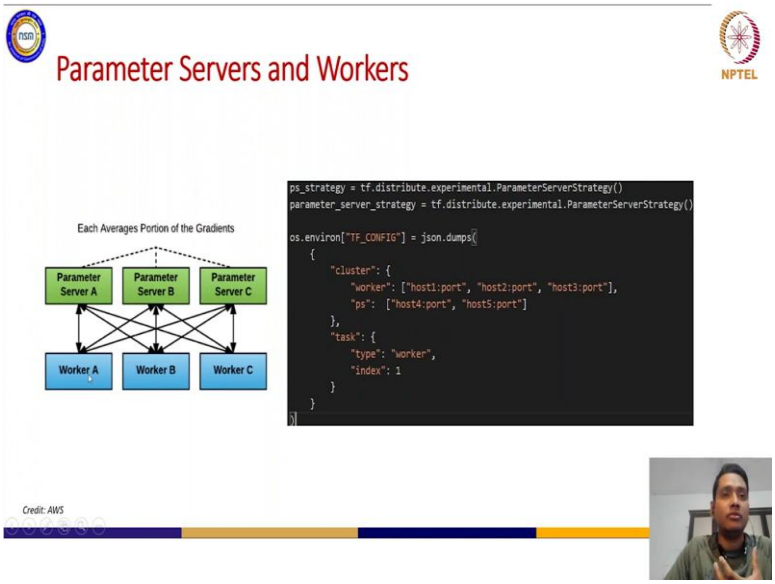
You can have all reduce synchronized training for TPUs as well; so, it is a bit tricky, but since through the colab also you have seen that they you have the flexibility to access GPUs use it is similar to your mirrored strategy. But unlike mirrored strategy it will use cross replica sum to synchronize the aggregations or to different nodes one TPU is TPU and many TPUs is called TPU pods ok; so, you can have both.

So, how to define this strategy; so, again as I was mentioning that once you have defined this strategy you can use strategy.scope() and define your scope that is it. Again, here we

will define the strategy and strategy.scope() you will define your distribute aware block that's it. How you can define it? You can define your resolver, which is essentially cluster resolver class and this will actually be using this TPU cluster resolver for the tpus ok.

So, its bit tricky, but you can use these texts; so, to use this strategy. Now, I how you will get the strategy you will just define to connect to the clusters initialize the tpu system and then use this resolver to create the strategies right. So, this is again coming under distribute dot experimental module where you will have this TPU strategy to be defined for your TPUs and you can use this strategy using strategy.scope().

(Refer Slide Time: 14:53)



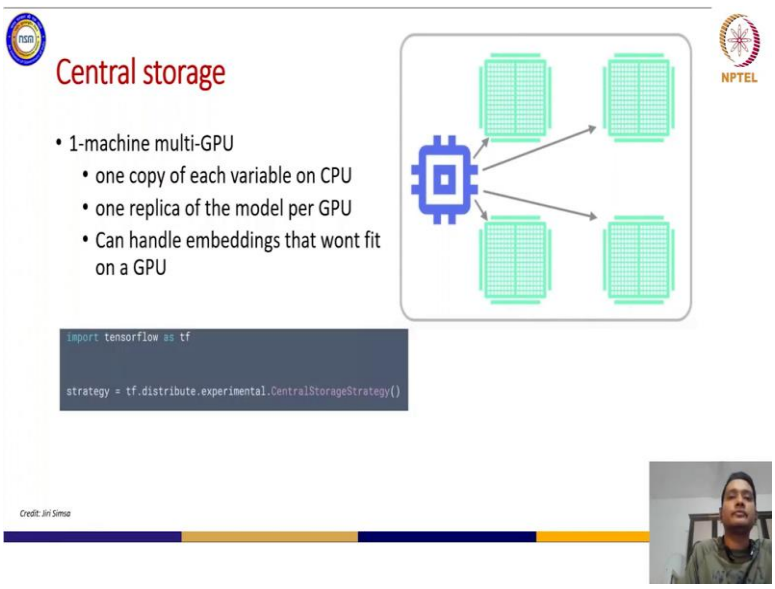
So, we have talked about synchronous execution where your workers are working in a locked state right. What happens if your workers let us say multiple devices you have in one system they are not similar in runtime problem. Let us say some worker will little faster some worker will slower and slower and slower.

So, you will have a different performance workers in your device or in a machine and you can use parameter servers and workers in this case. Parameter servers, maintains the global variables which will be actually synchronized and doing the aggregation. And worker nodes will again do the replication of the variables computing and then it will sync to the server parameter servers to get aggregated and you came back; so, parameter servers will take care of this synchronization.

So, these workers will be working in their own way ok; so this is a bit different from your synchronous way of working things. But once you have the APIs you do not need to worry about anything; so, the parameter servers strategy we are defining it with again it is under coming under distribute dot experimental. And the parameters servers strategy you can define it and we can also define with parameters server strategy with this these two ways ok; so, this is just the name that we are doing.

Now, once you have defined it actually you can use the strategy as a scope and also you need to define in the json string format the workers, which will be the ps the parameter servers and which will be the work. So, these two definitions if you just give in the our configuration string that is it; so, once you have this definition for the tasks and the classroom. So, basically what nodes will be the worker and what are the nodes will be your parameter server; so, that is.

(Refer Slide Time: 17:27)



It one more way of approaching is central storage; so, central storage is basically the one node approach of parameter server one or ps worker strategy that you have seen; so, this is asynchronous also. But we think that; so, here we had multiple devices where we have defined, which will be the parameter servers and which will be the workers.

But here we are thinking from a single node point, in a single node you will have multiple devices which are not exactly same in performance. So, you can use central storage approach where one machine multiple GPU will have one copy of each variable

will be in CPU. So, maybe CPU will be working as parameter server right, and one replica of the model will be in each GPU here and because it might so happen where parameters the number of parameters this is another case.

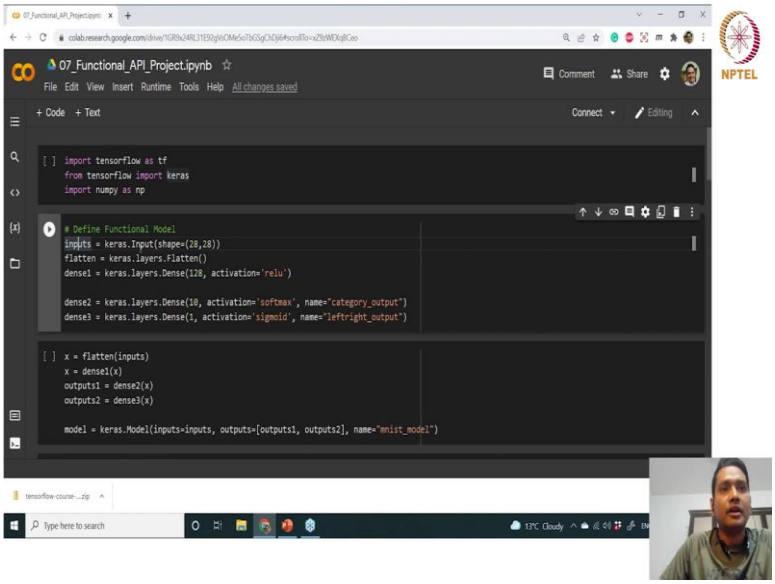
Where the number of parameters is getting exceeded more than the memory that is available for your GPU, what you will do because you are doing the replica you are maintaining the gradients. And then again you are sending back those you are getting all the updated values aggregated values again. So, it might so happen all the embeddings cannot may not feed your GPU memory in that case you can use your CPU to be working as that central storage ok.

And offered everything to your CPU and do release some of the memory pressure from the GPUs. So, how you can define it? Again it is very simple strategy, equal to tf.distribute.experimental.CentralStorageStrategy() and that is it. So, your one pc system will be working as your parameter server strategy and you can use this strategy.scope() to define it ok.

(Refer Slide Time: 19:44)

So, these are the reference codes that is already implemented with engineers from tensor flow for getting to know more about this distribution strategy that is already implemented for several frames that you can do we will go into. Now, one demo for functional API to understand better, some problem approach we will see.

(Refer Slide Time: 20:13)



So, what is the problem that we are referring to solve it with functional API mentioning I may have single input and multiple outputs right. So, one problem I am defining here; so, you have seen the image data set; so, many times for now which is essentially the data set for your handwritten digits right. So, from 0 to 9 we have digits and all the images from different way of writing it is there.

Now, I want to take I want to create one model which will classify first or of course, the digit whether the digit is 5 or 6 or 9 or 10 or 0 like rather. And another head we will predict whether this is being written in left hand or right hand ok; so, this is the problem statement that we are targeting here. So, again one single input, one image you will give as input and it will give two outputs, what is the number? Essentially 0 to 9 and whether the number is written with using right hand or left hand right ok.

So, keep the problem definition in mind and we will go forward with that model definition how you will define such model which will have two outputs and then you a use the functional API. So, simple inputs we will define with Keras, let us say 28 by 28 image that is standard for your image data set. We are flattening this using this flatten Keras.layers.flatten() Keras API.

Again, we are using all the imports of there you can see, first dense layer we are defining, which is dense layer which is 120 output using activation relu function. Second

dense layer it is using 10 outputs and activation; so, 10 outputs you can see here this Dense layer output is essentially will classify which digit is it.

So, we need 10 classes to classify; so, 10 class and activation soft max function we are using because we need to classify in the end because this will convert the logits into your predictions and name is category output so, category of the digit that we have defined now this layer.

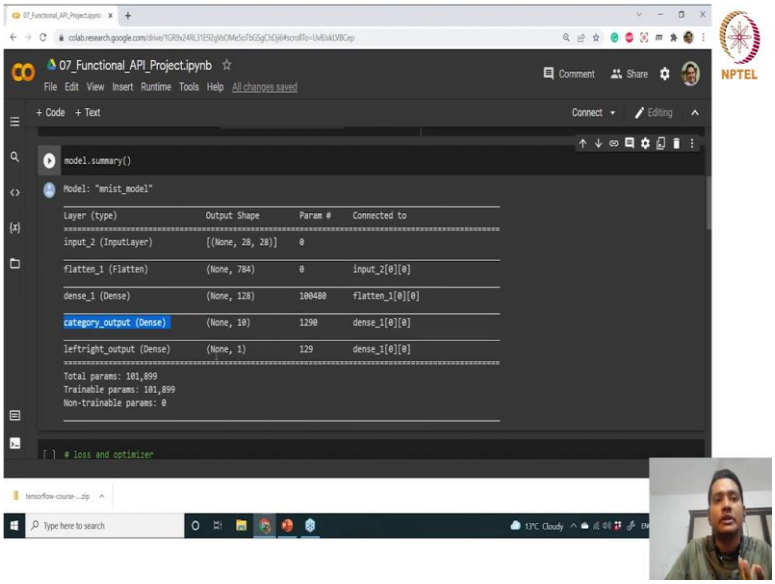
You can name one layer; so, this is just such use case where you can name the layer and you can see the layer definition of this layer. Now, here we are creating two output layers, one output layer for the category output and another layer for left right output. So, there is just one prediction output will be there, because that will give you the a sigmoid activation.

And with this sigmoid activation we can get whether this is just binary classification we are doing it right whether it is being written on the left hand or right hand; so, two heads we have defined. Now, here we have defined the layers as functions; now, we will use these to call on the inputs. So, now, we are calling this right; so, x is flattening the input; so, the input is flat pattern here because this is the fully connected layer we are using dense layers x equal to dense 1.

So, first dense layer, then output 1 is taking this x as input and doing the 10 outputs and this output 2 is taking this layer basically this output of this layer. So, same output is going two head to two heads and giving two outputs right so, this is the important thing that in the function call a function, functional API you can do it this is the thing that I wanted to show you.

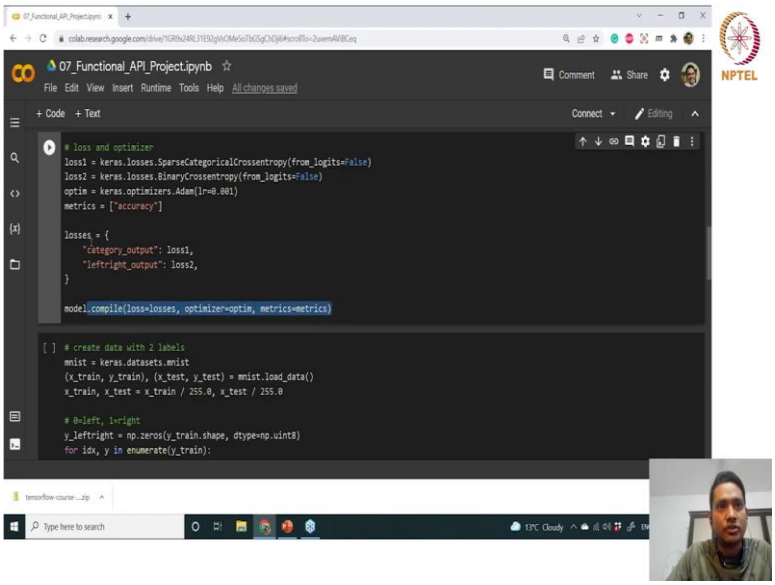
Then you can define the module keras dot model inputs 1 input we have input equal inputs; outputs. Now, you can see we are giving the output list here; so, one model we have created now which has multiple outputs. So, we are giving one list of outputs here outputs 1, outputs 2 name equal to mnist model; so, you can define one model; so, this is the name of the model.

(Refer Slide Time: 25:13)



If I see the model summary what I will see this that input layer the size, flatten layer the size, dense one layer with this number of parameters flatten 1 which is the name. But, category output 1; so, you see this name here category output category output, dense layer and category left to right output dense layer. So, these are the two output layers we have created and; so, total; so, this is the model definition model summary whatever you call it.

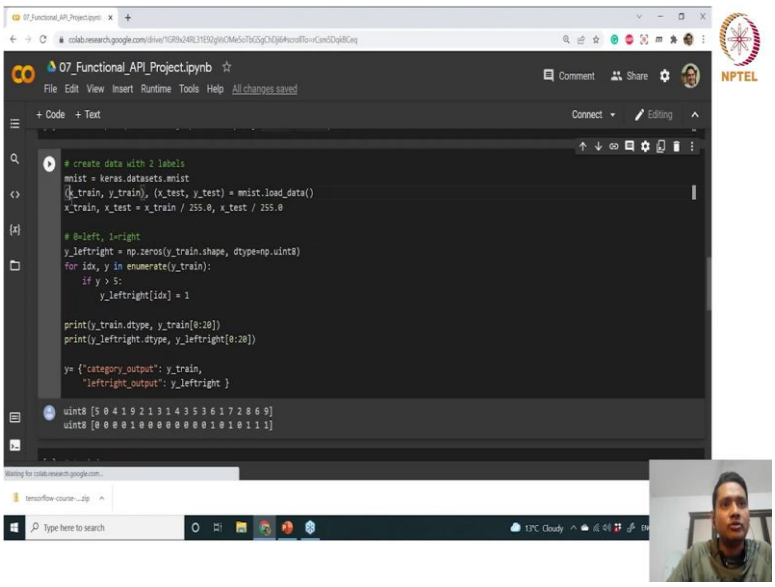
(Refer Slide Time: 25:47)



Then we are defining two losses, because two heads we have; so, for the SoftMax for this category loss category output we have this cross-entropy loss sparse categorical cross entropy loss. From logits equal to false, because we have defined the SoftMax activation, if you did not define the activation here then you need the logits.

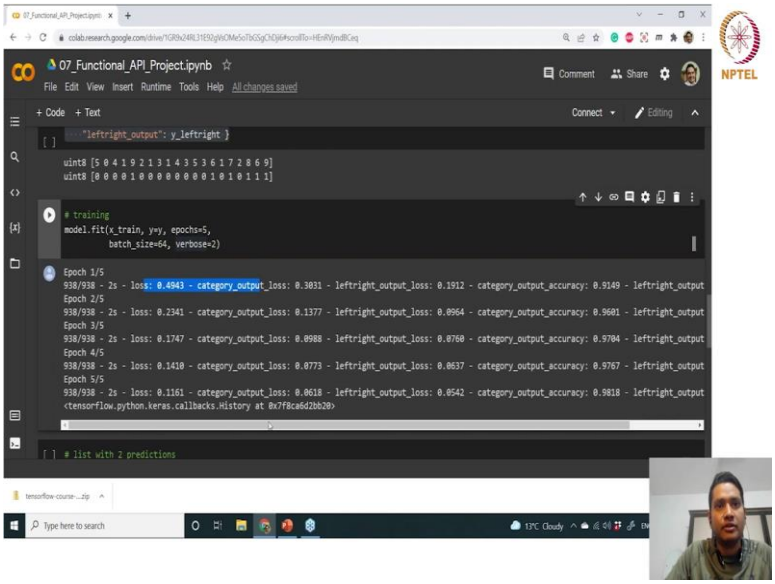
So, logits are the outputs directly coming from the neurons. Now, loss 2 is essentially again binary cross entropy loss because we are doing the binary classification optimizer we are defining, metrics we are defining, what metrics we want to see, accuracy we want to see. Two losses, one loss is for category output and second loss is for left right output. So, for two output layers we have these two losses we are comparing the module model with this loss whichever we have defined here with this optimizer and with this metrics simple as that.

(Refer Slide Time: 26:53)



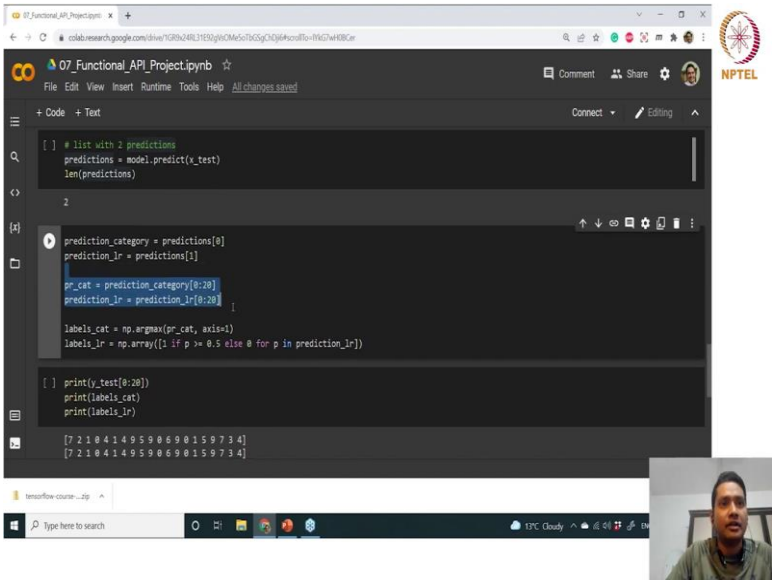
And we are training the module here; so, basically we are creating the data set here. So, keeping the keras.datasets as a source we are taking the I mean mist data set, creating train test modules. And then we are its unpacking and normalizing the images here, then we are now we are creating the two classes. Let us say we are creating the two classes let us say greater than 5 will have written as let us say 1 ok; so, that is right handed and less than 5 is left handed let us say ok.

(Refer Slide Time: 27:34)

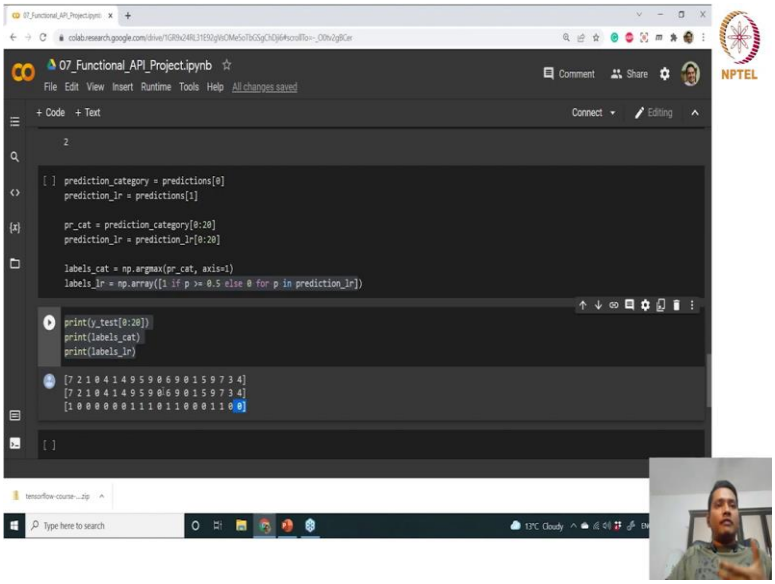


So, you can see here; so, all these digits 5 0 0 0 4 0, but 9 1 6 1; so, all these are written in right hand and these are written in left. So, now, our class levels have been created then we are going into training to just feed call feed with this train function train data set this number of the epochs, batch size, verbose 2. So, now, we are getting the outputs here for the accuracy loss and for different epochs.

(Refer Slide Time: 28:06)



(Refer Slide Time: 28:24)



And we are doing the predictions in the end and this prediction is essentially calling on the test, and how many predictions we are doing? Two predictions here. First prediction is for your left-hand right category and then left-hand right hand. And let us say we are taking the first 20 data and checking what is the maximum prediction for that and what is the left-hand and right-hand prediction for that.

Printing the data, the labels with let us say category and the labels for your right left hand right; so, all the digits have been successfully categorized. So, these are the predicted labels and predicted labels for the left-hand right hand you can see; one is for your 7 0 is for 2, 9 5 1 this is wrong prediction, but again anyway.

So, you have seen how you can use functional API to simply define your module with different number of outputs and you can train the module and extract the predictions and do so, as simple as that. So, if you are doing transfer learning or maybe multi head or multistage training you are creating the pipeline this is the best way to go using functional API; so, it is functional API we will close this session.