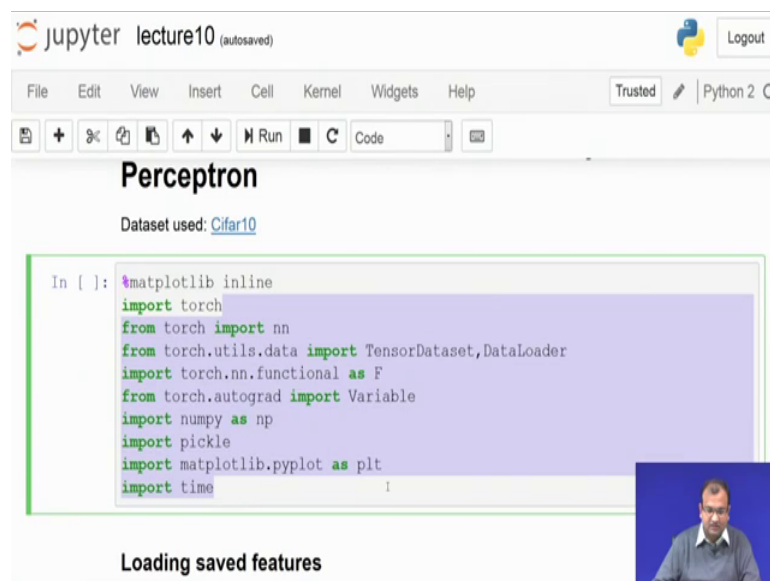


**Deep Learning for Visual Computing**  
**Prof. Debdoot Sheet**  
**Department of Electrical Engineering**  
**Indian Institute of Technology, Kharagpur**

**Lecture - 10**  
**Classification with Multilayer Perceptron**

So welcome, and today we would be learning about Classification with a Multilayer Perceptron.

(Refer Slide Time: 00:21)



```
In [ ]: %matplotlib inline
import torch
from torch import nn
from torch.utils.data import TensorDataset, DataLoader
import torch.nn.functional as F
from torch.autograd import Variable
import numpy as np
import pickle
import matplotlib.pyplot as plt
import time
```

Loading saved features

And while we have done down our basic introductory lectures on what a multilayer perceptron is and then how to use it for classification. So, here the objective is that building upon top of what we had done in the previous class of using down just a set of features, extracted from the images. And then we were feeding it through a neural network and this neural network just had the inputs connected directly to the classification output. So, you had nine features which were going down on the input side of the neural network and it was just connected down to a 1 hot vector of 10 cross 1 dimension and that was basically because it has just 10 classes over there to classify.

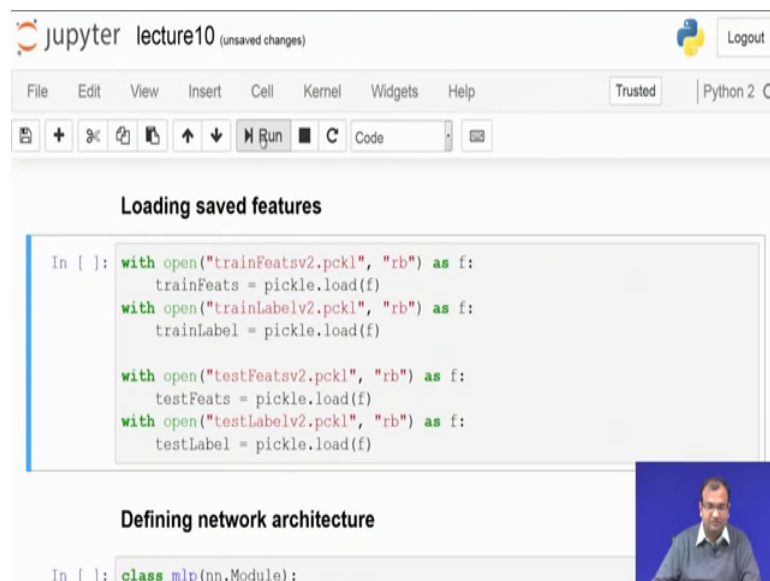
So, any 1 of these classes is going to be 1 based on which particular class is being shown over there, now 1 thing to remember clearly what there was that that was a very simple perceptron model which did not have any number of hidden layers, but then when we were doing with multilayer perceptron, we did realize that you can add down more

number of hidden layers and that would subsequently introduce a lot of non-linearity over that and these non-linearity will be some sort of a hierarchical non-linearity going down over there and that would help you in creating much better separation emerges by learning much better kind of features as well.

So, here today what we would do is write down such a network which will have multiple hidden layers over there and see how it gets trained. So, building upon top of those scratchpads which we had used for creating a simple perceptron we will be starting for that, so let it get into how this code particularly works over here. So, the first part so this is on classification with the multilayer perceptron and this is still for your cifar10 dataset itself.

Now, in this first part we just have the initial header and that is not much to redefine around this header.

(Refer Slide Time: 02:14)



The screenshot shows a Jupyter Notebook window titled "lecture10 (unsaved changes)". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help), a toolbar with icons for file operations and execution, and a "Trusted" status indicator. The notebook content is divided into two sections:

- Loading saved features**: This section contains a code cell with the following Python code:

```
In [ ]: with open("trainFeatsv2.pkl", "rb") as f:
        trainFeats = pickle.load(f)
        with open("trainLabelv2.pkl", "rb") as f:
            trainLabel = pickle.load(f)

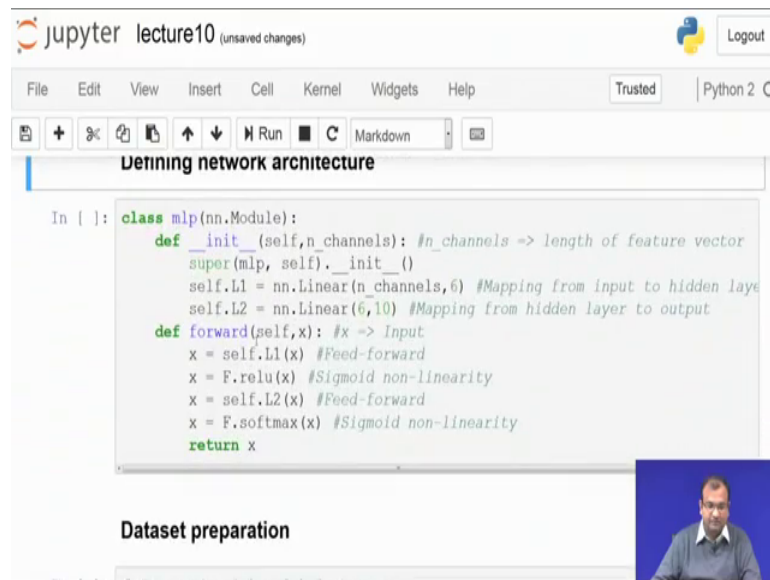
        with open("testFeatsv2.pkl", "rb") as f:
            testFeats = pickle.load(f)
        with open("testLabelv2.pkl", "rb") as f:
            testLabel = pickle.load(f)
```
- Defining network architecture**: This section contains a code cell with the following Python code:

```
In [ ]: class mlp(nn.Module):
```

A small video thumbnail of a person is visible in the bottom right corner of the notebook interface.

As well because we are just keep going to keep this as the same header over there. So, let is just run and get down our initial files into the environment, now once that is done the next part is that you already have your features which were saved down from lecture 3 exercises, where we had basically learned how to collect these features and so we run them and these features come down to me.

(Refer Slide Time: 02:39)



```
In [ ]: class mlp(nn.Module):
def __init__(self, n_channels): #n_channels -> length of feature vector
    super(mlp, self).__init__()
    self.L1 = nn.Linear(n_channels, 6) #Mapping from input to hidden layer
    self.L2 = nn.Linear(6, 10) #Mapping from hidden layer to output
def forward(self, x): #x -> Input
    x = self.L1(x) #Feed-forward
    x = F.relu(x) #Sigmoid non-linearity
    x = self.L2(x) #Feed-forward
    x = F.softmax(x) #Sigmoid non-linearity
    return x
```

Now, once I have loaded down all of my features on and my labels both for the training and the testing over there, the next part comes down to defining this network. Now you clearly remember that in the last case what we had was that there, so this class was called as a perceptron in that case and here we are calling it as mlp. So, mlp stands for multilayer perceptron and then multilayer concept is which comes from that you have multiple number of layers given down over there.

So, multiple number of layers in terms of multiple hidden layers present out. So, the first part is basically an initialization module. So, in this initialization module what you do is that you are going to define how many number of layers over there. So, we have 2 layers and both of them are linearly connected. So, the first 1 is which connects down n number of channels to 6.

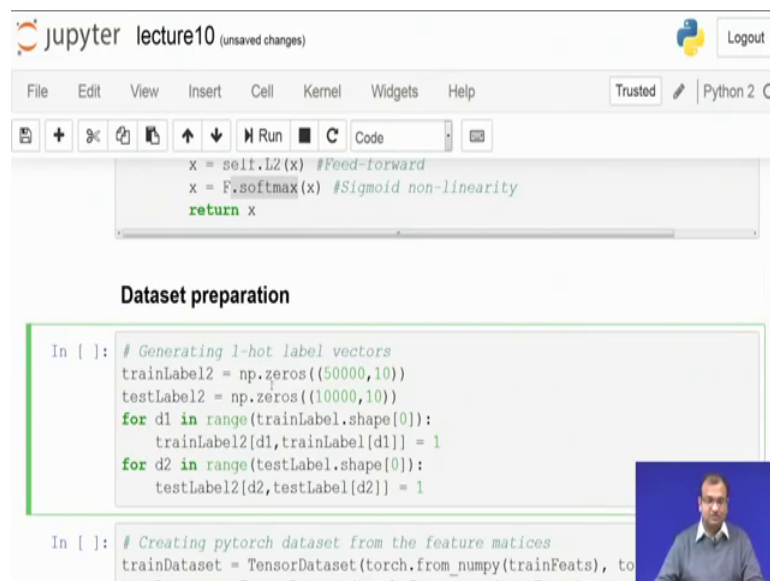
So, n channels over here are the total number of features which you are giving down to this network. So, that is basically 9 for our case, but it can be any variable number. So, this is a 9 to 6 mapping. So, this 6 number of neurons is the number of neurons present down in your first hidden layer, now from that first hidden layer it connects down to 10 which are my output mapping and this 10 mapped down to my total number of classes in the output.

Now once that is done the next part is left with that I need to define; what is my forward pass of this algorithm. So, in order to define my forward pass of the algorithm what I

define is that whatever comes in as in my input over there that has to do the first pass through my l1 and l1 is a n dot linear which is a mapping from n number of channels to 6 channels.

Now once that is done my output is again stored at x, now that will have a non-linearity and the non-linearity imposed over here is a very low kind of non-linearity which we put down. So, this relu or rectified linear unit is something which is for any value less than 0, you will truncate all the values to 0 and for any value which is greater than 0, the value will remain as the same. So, this is basically some sort of form in which it remains linear for positive values of x and become 0 for negative values of x; the next l1 is another feed forward which you do through the next linear layer. And then finally you put down soft max as your non-linearity coming down from the output of it.

(Refer Slide Time: 04:57)



The screenshot shows a Jupyter Notebook window titled "lecture10 (unsaved changes)". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help), a toolbar with icons for file operations and execution, and a code editor. The code editor contains the following code:

```
x = self.L2(x) #Feed-forward
x = F.softmax(x) #Sigmoid non-linearity
return x
```

Below the code editor, there is a section titled "Dataset preparation" with a code cell containing the following code:

```
In [ ]: # Generating 1-hot label vectors
trainLabel2 = np.zeros((50000,10))
testLabel2 = np.zeros((10000,10))
for d1 in range(trainLabel.shape[0]):
    trainLabel2[d1,trainLabel[d1]] = 1
for d2 in range(testLabel.shape[0]):
    testLabel2[d2,testLabel[d2]] = 1
```

At the bottom of the notebook, there is another code cell with the following code:

```
In [ ]: # Creating pytorch dataset from the feature matrices
trainDataset = TensorDataset(torch.from_numpy(trainFeats), to
```

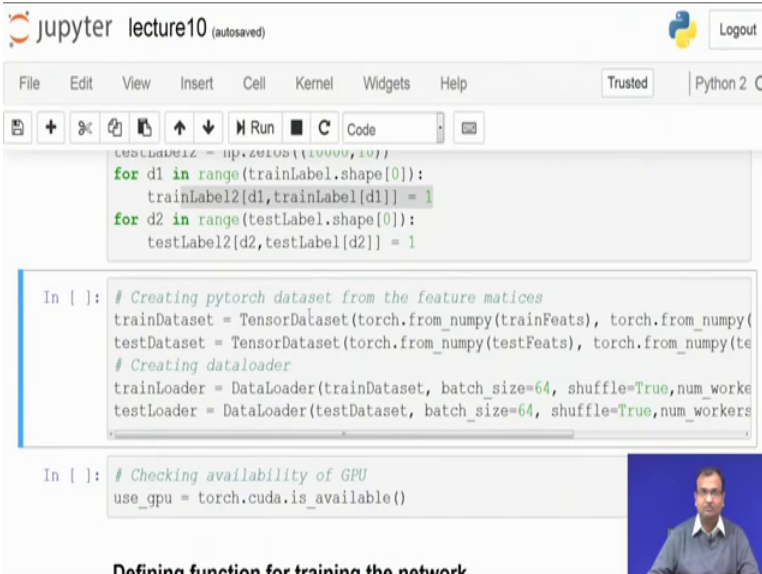
A small video thumbnail of a person is visible in the bottom right corner of the notebook interface.

Now, once that is done the next part goes down in your.

So this is my Network which gets defined over here. So, let is just define my network now comes down the data preparation stage over there. So, in data preparation what we had done in last class was quite simple, so we define 2 matrices 2 sort of like 2 matrices which were 50000 cross 10 and another was 10000 cross 10 and this was just for your labels.

So, you remember clearly that we had labels in the range of 0 to 10, but then when training this network we needed to have a 1 hot vector and that meant that it needs to have 10 number of rows corresponding to 10 number of columns corresponding to each single row and any 1 of these items on these rows is going to be 1 based on which particular class it belongs to. So, we start with the same way so your train label becomes a 2 d matrix of 50000 cross 1 and your test labels also become 2 d matrix of 10000 cross 1 and accordingly over here you do your reassignment.

(Refer Slide Time: 06:02)



```
testLabel2 = np.zeros((10000,10))
for d1 in range(trainLabel.shape[0]):
    trainLabel2[d1,trainLabel[d1]] = 1
for d2 in range(testLabel.shape[0]):
    testLabel2[d2,testLabel[d2]] = 1

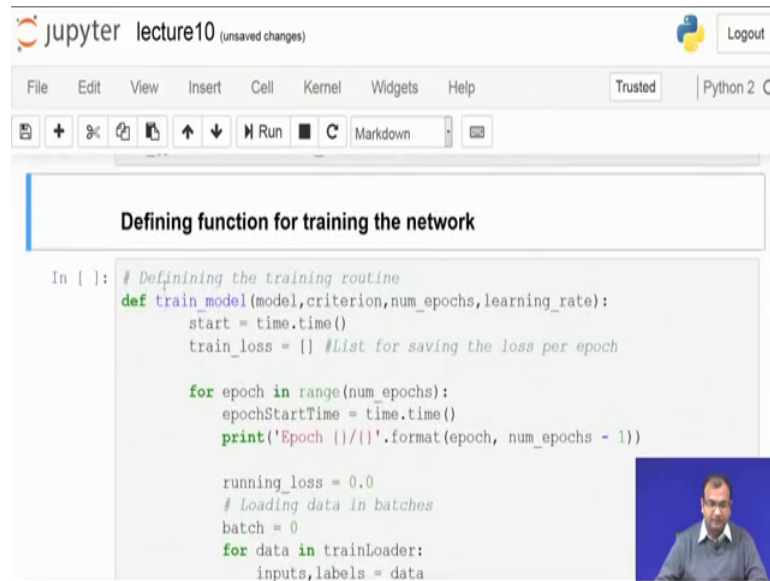
In [ ]: # Creating pytorch dataset from the feature matrices
trainDataset = TensorDataset(torch.from_numpy(trainFeats), torch.from_numpy(
testDataset = TensorDataset(torch.from_numpy(testFeats), torch.from_numpy(te
# Creating dataloader
trainLoader = DataLoader(trainDataset, batch_size=64, shuffle=True,num worke
testLoader = DataLoader(testDataset, batch_size=64, shuffle=True,num_workers

In [ ]: # Checking availability of GPU
use_gpu = torch.cuda.is_available()

Defining function for training the network
```

So, just assign whichever value whichever class is 1 you assign that value as 1 over here. Now once that part is done the next part is to work and create down your convert your data set which is available as of now in as a numpy array to a torch tensor and that is what we were doing it. So, this part of the scratch is pretty much same as what we had done in the earlier class. And then the next one was to check down whether you have a GPU available and a CUDA support on your system or not.

(Refer Slide Time: 06:35)



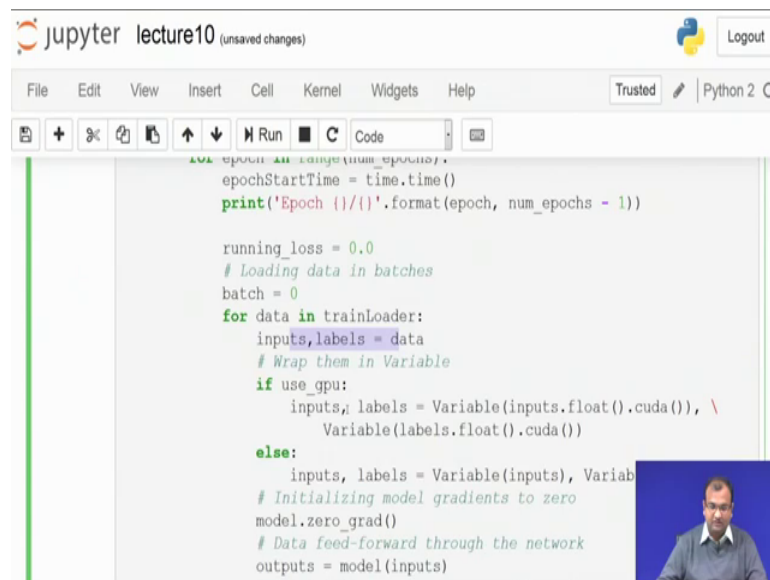
```
In [ ]: # Defining the training routine
def train_model(model,criterion,num_epochs,learning_rate):
    start = time.time()
    train_loss = [] #List for saving the loss per epoch

    for epoch in range(num_epochs):
        epochStartTime = time.time()
        print('Epoch {}/{}'.format(epoch, num_epochs - 1))

        running_loss = 0.0
        # Loading data in batches
        batch = 0
        for data in trainLoader:
            inputs,labels = data
```

Now, once that gets done and so we start by defining at the training routine for our model. So within this training routine for the model the idea was that you would have some sort of an iterator and this is an epoch iterator.

(Refer Slide Time: 06:46)

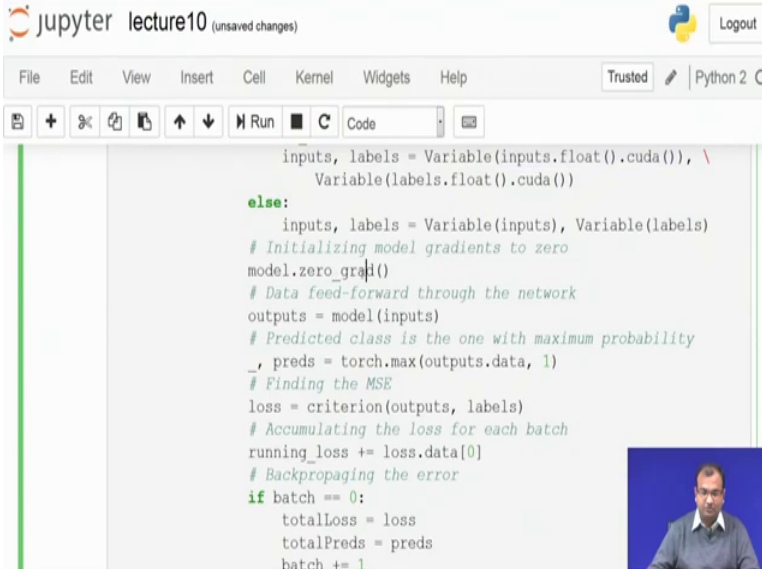


```
running_loss = 0.0
# Loading data in batches
batch = 0
for data in trainLoader:
    inputs,labels = data
    # Wrap them in Variable
    if use_gpu:
        inputs, labels = Variable(inputs.float().cuda()), \
            Variable(labels.float().cuda())
    else:
        inputs, labels = Variable(inputs), Variab
    # Initializing model gradients to zero
    model.zero_grad()
    # Data feed-forward through the network
    outputs = model(inputs)
```

Now, within the epoch iterator there is a fancy sort of a function called over here, for the time in order to check out how many seconds' milliseconds it takes basically to execute each of them.

So, we remember from our last one that it was taking roughly about 1 second and in total for 20 epochs it took down 27 seconds to execute. So, from there the rest of the concepts in terms of running laws then creating a batch over there and using your train loader that remains the same and then if GPU is available then to convert all of these to your CUDA tensor type. So, that you have a memory transfer from your cpu rams on to your GPU ram that is also quite clearly and this is what we are just repeating from the earlier case.

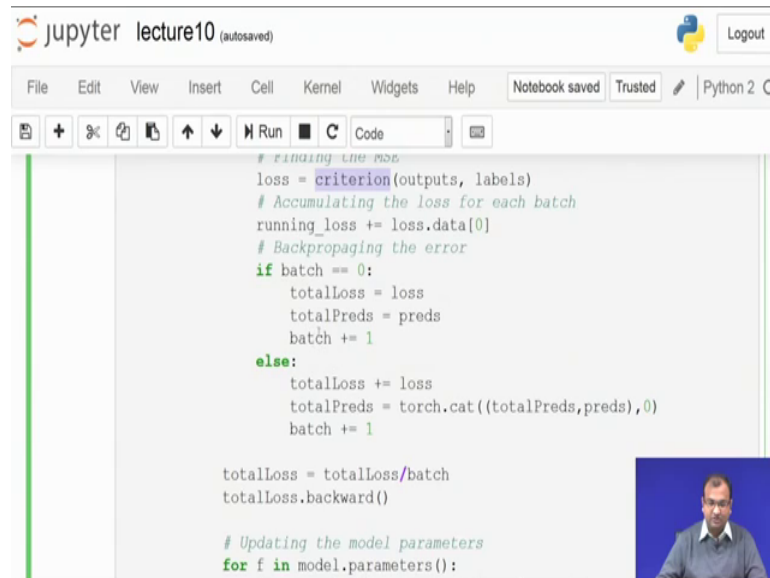
(Refer Slide Time: 07:38)



```
inputs, labels = Variable(inputs.float().cuda()), \
    Variable(labels.float().cuda())
else:
    inputs, labels = Variable(inputs), Variable(labels)
    # Initializing model gradients to zero
    model.zero_grad()
    # Data feed-forward through the network
    outputs = model(inputs)
    # Predicted class is the one with maximum probability
    _, preds = torch.max(outputs.data, 1)
    # Finding the MSE
    loss = criterion(outputs, labels)
    # Accumulating the loss for each batch
    running_loss += loss.data[0]
    # Backpropagating the error
if batch == 0:
    totalLoss = loss
    totalPreds = preds
    batch += 1
```

Next before you start training you will need to always 0 down your gradients once that is done next is the feed forward routine and that defines as output is equal to model of input and model is basically the function which is my multilayer perceptron neural network; given all of that the next part is to use these output in order to get my predictions for each of these classes, and then from there find out my loss which is defined according to this criteria.

(Refer Slide Time: 08:06)



```

# finding the loss
loss = criterion(outputs, labels)
# Accumulating the loss for each batch
running_loss += loss.data[0]
# Backpropagating the error
if batch == 0:
    totalLoss = loss
    totalPreds = preds
    batch += 1
else:
    totalLoss += loss
    totalPreds = torch.cat((totalPreds,preds),0)
    batch += 1

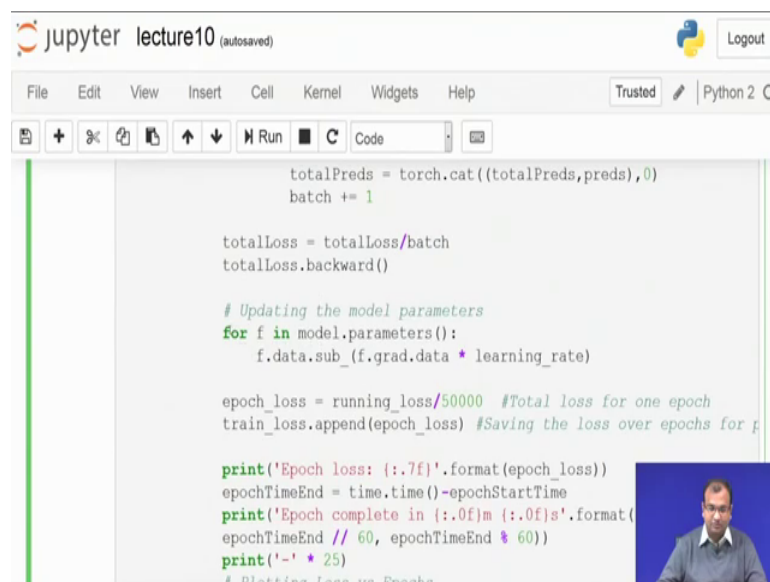
totalLoss = totalLoss/batch
totalLoss.backward()

# Updating the model parameters
for f in model.parameters():

```

So, the criteria function gets defined a bit later on as of now we are just using it as a simple pointer called as criterion and then you find out your batch phase losses. And finally comes down the point where you need to update your parameters.

(Refer Slide Time: 08:23)



```

totalPreds = torch.cat((totalPreds,preds),0)
batch += 1

totalLoss = totalLoss/batch
totalLoss.backward()

# Updating the model parameters
for f in model.parameters():
    f.data.sub_(f.grad.data * learning_rate)

epoch_loss = running_loss/50000 #Total loss for one epoch
train_loss.append(epoch_loss) #Saving the loss over epochs for p

print('Epoch loss: {:.7f}'.format(epoch_loss))
epochTimeEnd = time.time()-epochStartTime
print('Epoch complete in {:.0f}m {:.0f}s'.format(
epochTimeEnd // 60, epochTimeEnd % 60))
print('-' * 25)
# Plotting Loss vs Epochs

```

So, here is where this happens so if you clearly remember you have multiple number of hidden layers over there and there are multiple number of connections between each of these neurons in 1 layer to other neurons in the subsequent layer and this is what is called as those free parameters or the weights and these parameters are what get updated. So,

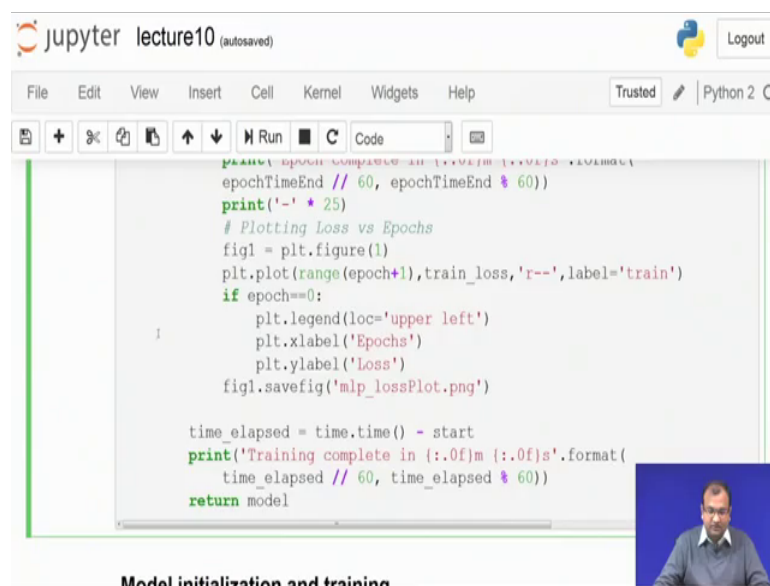


your update rule which was given as  $w$  of  $k$  plus 1 are is equal to  $w$  of  $k$ , where  $w$  of  $k$  is basically the weight of these connections within the neural network at the  $k$ -th iteration and minus  $\eta$  times of  $\Delta w$  of  $j$   $w$  and that  $k$ -th iteration.

So, we need to have our gradient of the data as well as the learning rate and that is going to define down what is my update parameter over there. Now from this once your parameters are updated then you can actually look into your total epoch loss. So, what we are doing over here is that you run down basically 50000 training samples throughout 1 epoch and accumulate all the errors.

So, your epoch loss is basically average of that error and that is where this part comes down and then you can keep on basically creating an array where you are adding down all of those losses and it becomes easier to plot down. Now here you have these printer statements which basically print down what is the total epoch loss which comes down while it is training down, so that you can see that it is convergent or not.

(Refer Slide Time: 09:55)



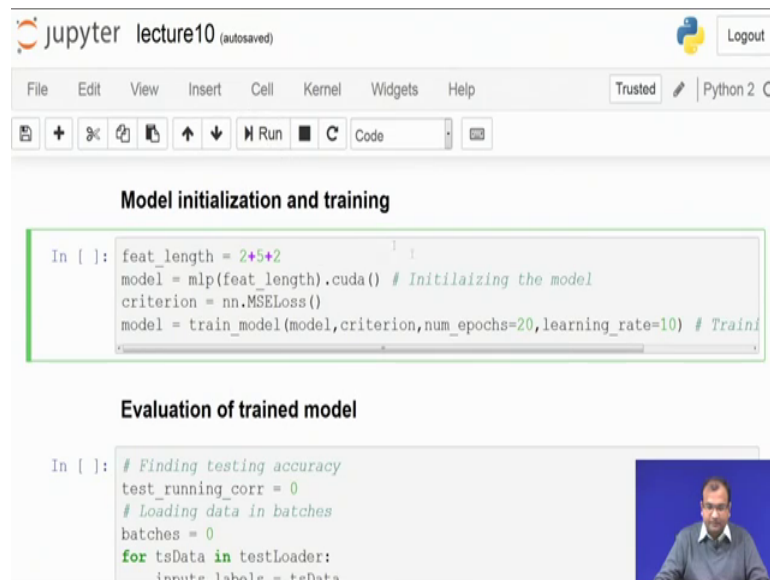
```
print('epoch complete in {:.0f}s'.format(
epochTimeEnd // 60, epochTimeEnd % 60))
print('-' * 25)
# Plotting Loss vs Epochs
fig1 = plt.figure(1)
plt.plot(range(epoch+1),train_loss,'r--',label='train')
if epoch==0:
    plt.legend(loc='upper left')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    fig1.savefig('mlp_lossPlot.png')

time_elapsed = time.time() - start
print('Training complete in {:.0f}s {:.0f}s'.format(
time_elapsed // 60, time_elapsed % 60))
return model
```

Model initialization and training

So, this is where your trainer model gets defined. So, let is just run this part and keep get it be fetched as a function within your environment.

(Refer Slide Time: 10:06)



The screenshot shows a Jupyter Notebook window titled "lecture10 (autosaved)". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help), a toolbar with icons for file operations and execution, and a "Trusted" status indicator. The notebook content is divided into two sections:

### Model initialization and training

```
In [ ]: feat_length = 2+5+2
        model = mlp(feat_length).cuda() # Initilaizing the model
        criterion = nn.MSELoss()
        model = train_model(model,criterion,num_epochs=20,learning_rate=10) # Traini
```

### Evaluation of trained model

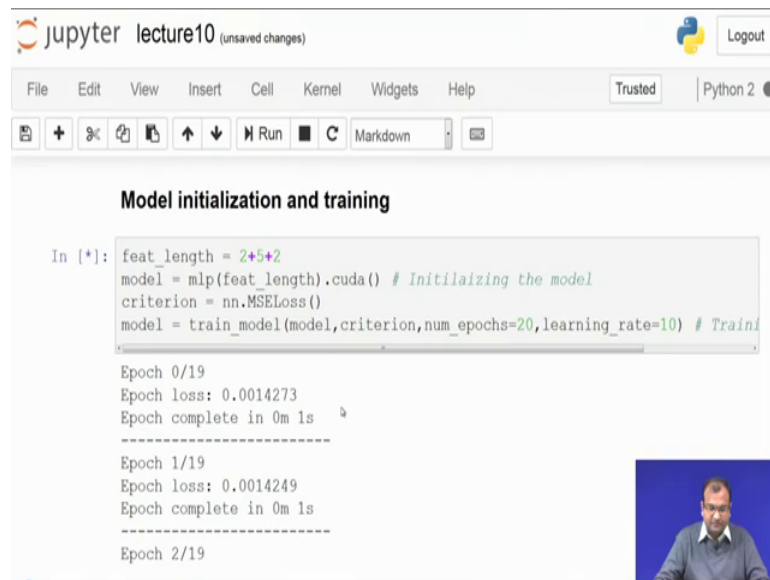
```
In [ ]: # Finding testing accuracy
        test_running_corr = 0
        # Loading data in batches
        batches = 0
        for tsData in testLoader:
            inputs,labels = tsData
```

A small video thumbnail of a man speaking is visible in the bottom right corner of the notebook interface.

Now, once that is done the next part is to actually get into your initialization and training. So, the first part is we need this length of features and this is what we know from our earlier experiences, you can as well look into the dimension second dimension of the pickle file that is and fetch it out as well that is also pretty much possible. Now within your model first thing is that since we have CUDA available.

So, we just defined it type castrated once again into CUDA and then define the criterion or the loss function as MSE loss function and then we set this 1 to train. Now let us and then we are training it again with just 20 epochs in order to look into how it starts to behave.

(Refer Slide Time: 10:48)



The screenshot shows a Jupyter Notebook interface with the title "lecture10 (unsaved changes)". The code cell contains the following Python code:

```
In [*]: feat_length = 2+5+2
model = mlp(feat_length).cuda() # Initilaizing the model
criterion = nn.MSELoss()
model = train_model(model,criterion,num_epochs=20,learning_rate=10) # Traini
```

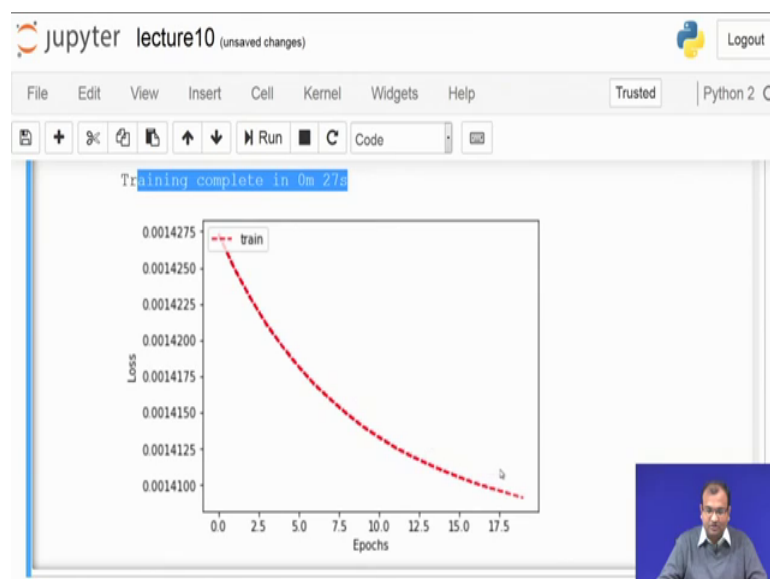
The output of the code cell shows the training progress:

```
Epoch 0/19
Epoch loss: 0.0014273
Epoch complete in 0m 1s
-----
Epoch 1/19
Epoch loss: 0.0014249
Epoch complete in 0m 1s
-----
Epoch 2/19
```

A small video inset in the bottom right corner shows a man speaking.

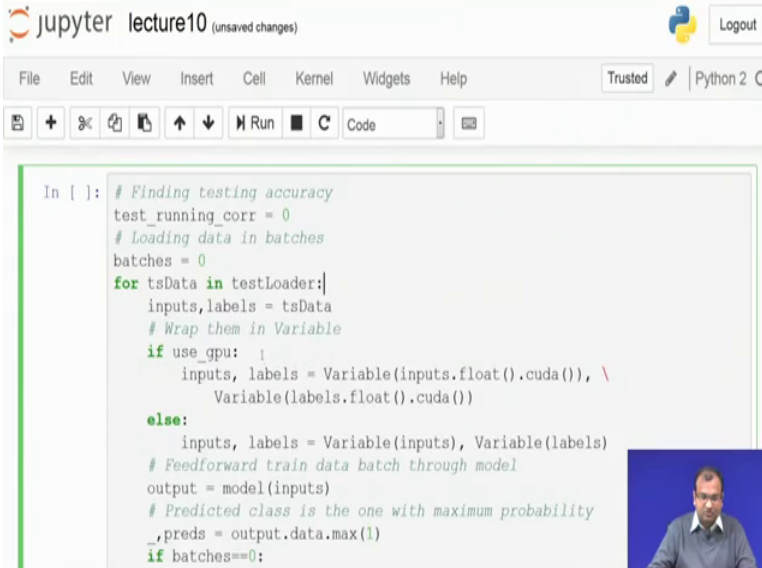
Now, we do see our losses coming down over there and if you look down. So, this is where the error is changing around on these points. And then that is changing down quite slowly I will not say that it is pretty fast, but there is a decent amount of decrease which keeps on going down. And although sometimes it does even look to increase as well and it is roughly taking down it shows as 1 second, but there are a few more milliseconds as well.

(Refer Slide Time: 11:19)



You see that do not take actually much more of a time as compared to the earlier 1 because, you still finished off your t20 epochs within 27 milliseconds and this is how your training loss was falling down. So, you can add down the extra module which we had in the earlier 1 in terms of accuracy and you can see the accuracy growing as well.

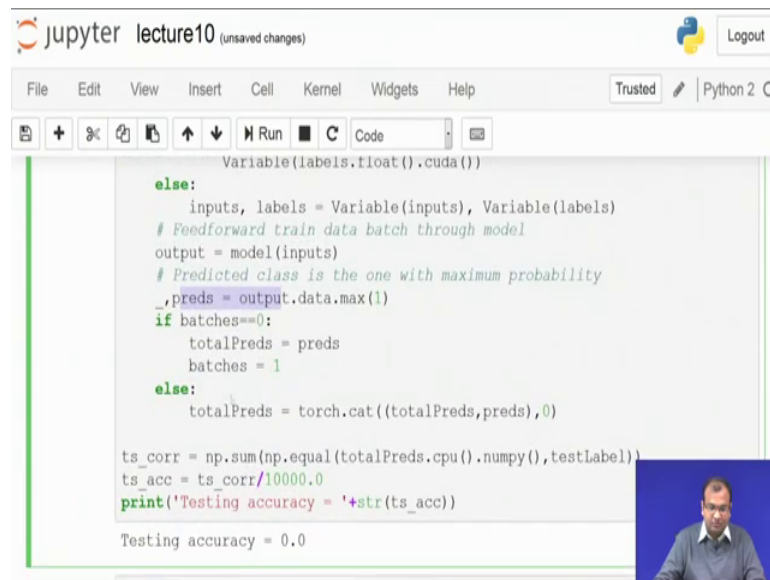
(Refer Slide Time: 11:38)



```
In [ ]: # Finding testing accuracy
test_running_corr = 0
# Loading data in batches
batches = 0
for tsData in testLoader:
    inputs, labels = tsData
    # Wrap them in Variable
    if use_gpu: 1
        inputs, labels = Variable(inputs.float().cuda()), \
            Variable(labels.float().cuda())
    else:
        inputs, labels = Variable(inputs), Variable(labels)
    # Feedforward train data batch through model
    output = model(inputs)
    # Predicted class is the one with maximum probability
    _, preds = output.data.max(1)
    if batches==0:
```

Now, once that is done the next part was to go with your train module and then look into it is testing part. So, over there what we do is we pull down we run an iterator, over the test dataset over the length of the test dataset and then feed forward each input through the model which happens over here.

(Refer Slide Time: 12:06)



```
Variable(labels.float().cuda())
else:
    inputs, labels = Variable(inputs), Variable(labels)
    # Feedforward train data batch through model
    output = model(inputs)
    # Predicted class is the one with maximum probability
    _, preds = output.data.max(1)
    if batches==0:
        totalPreds = preds
        batches = 1
    else:
        totalPreds = torch.cat((totalPreds, preds), 0)

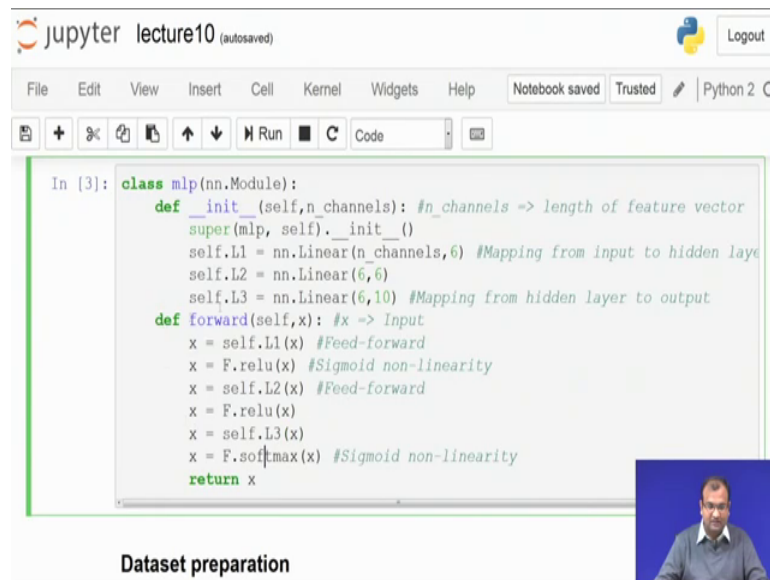
ts_corr = np.sum(np.equal(totalPreds.cpu().numpy(), testLabel))
ts_acc = ts_corr/10000.0
print('Testing accuracy = '+str(ts_acc))

Testing accuracy = 0.0
```

You get down your outputs and then see if your predicted output patches down your exact output and if it is. So, then it is correct or not and interestingly what comes out is that since the training has not yet completed. So, you get down 0 accuracy, but you can keep 1 running this 1 for a longer period of time.

So, for the sake of time constraints we are not running this for longer duration of time, but you can definitely try running this for a longer duration of time. Now there are a few interesting aspects which I would like to reiterate from our earlier understanding, so 1 of the major a point is that if you look into most part of this code, then they are actually something which is quite modularly written down. So, if I want to create a change in my model say I would like to change down my non-linearity.

(Refer Slide Time: 12:48)



```
In [3]: class mlp(nn.Module):
def __init__(self, n_channels): #n_channels -> length of feature vector
    super(mlp, self).__init__()
    self.L1 = nn.Linear(n_channels, 6) #Mapping from input to hidden layer
    self.L2 = nn.Linear(6, 6)
    self.L3 = nn.Linear(6, 10) #Mapping from hidden layer to output
def forward(self, x): #x -> Input
    x = self.L1(x) #Feed-forward
    x = F.relu(x) #Sigmoid non-linearity
    x = self.L2(x) #Feed-forward
    x = F.relu(x)
    x = self.L3(x)
    x = F.softmax(x) #Sigmoid non-linearity
    return x
```

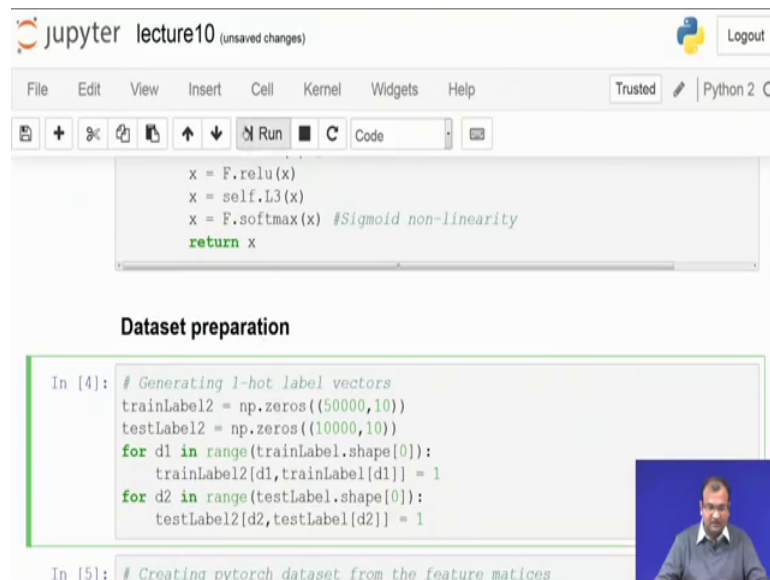
Dataset preparation

So, I can just change make a change over here or over here, if I want to add down a few models over there. So, if I say I want to add down a few more linear numbers of channels, so I can just keep on adding down over there. So, say like I just want to add down another layer I can write it as nn dot linear.

So, I have 6 number of outputs from there from that 6 number of outputs I can again think of going down to say another 6 number of outputs over there. That is also pretty much possible then I can do a nn dot linear from and so this will be self dot l3 is equal to nn dot linear of 6 to 10.

Now once that is done over here till this point I just had with the self dot l2 and what I can add is x is equal to self sorry, I can add another value and get a non-linearity imposed over there; now once this part is solved over here the next part is that I will have to again feed that through another part of the layer and that becomes x equal to self dot l3 of x and from here whatever comes down that goes into my soft max and then this will create down a new model, so we can just do a run on this 1.

(Refer Slide Time: 14:22)



The screenshot shows a Jupyter Notebook window titled "lecture10 (unsaved changes)". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for file operations and execution. The code cell contains the following Python code:

```
x = F.relu(x)
x = self.L3(x)
x = F.softmax(x) #Sigmoid non-linearity
return x
```

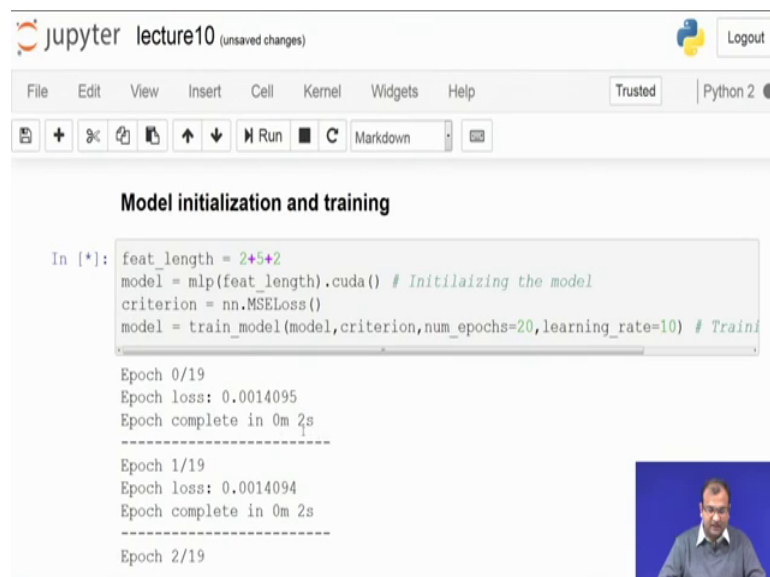
Below the code cell, the notebook displays the heading "Dataset preparation" and the output of an execution cell (In [4]):

```
# Generating 1-hot label vectors
trainLabel2 = np.zeros((50000,10))
testLabel2 = np.zeros((10000,10))
for d1 in range(trainLabel.shape[0]):
    trainLabel2[d1,trainLabel[d1]] = 1
for d2 in range(testLabel.shape[0]):
    testLabel2[d2,testLabel[d2]] = 1
```

The output of the next cell (In [5]) is partially visible: "# Creating pytorch dataset from the feature matrices". A small video inset of a man is visible in the bottom right corner of the notebook interface.

Now, once that is done I can again do the subsequent runs over the next and here comes my training module, you see increasing that I just got a tag bit more increase it came down to somewhere around 2 seconds and so we just need to wait for a few more moments till it gets over.

(Refer Slide Time: 14:51)



The screenshot shows a Jupyter Notebook window titled "lecture10 (unsaved changes)". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for file operations and execution. The code cell contains the following Python code:

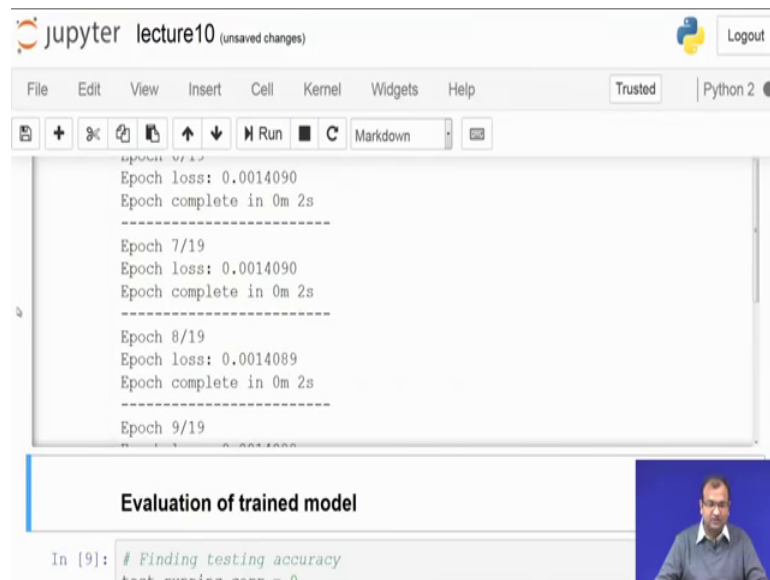
```
feat_length = 2+5+2
model = mlp(feat_length).cuda() # Initilaizing the model
criterion = nn.MSELoss()
model = train_model(model,criterion,num_epochs=20,learning_rate=10) # Traini
```

The output of the execution cell shows the following text:

```
Epoch 0/19
Epoch loss: 0.0014095
Epoch complete in 0m 2s
-----
Epoch 1/19
Epoch loss: 0.0014094
Epoch complete in 0m 2s
-----
Epoch 2/19
```

A small video inset of a man is visible in the bottom right corner of the notebook interface.

(Refer Slide Time: 14:53)



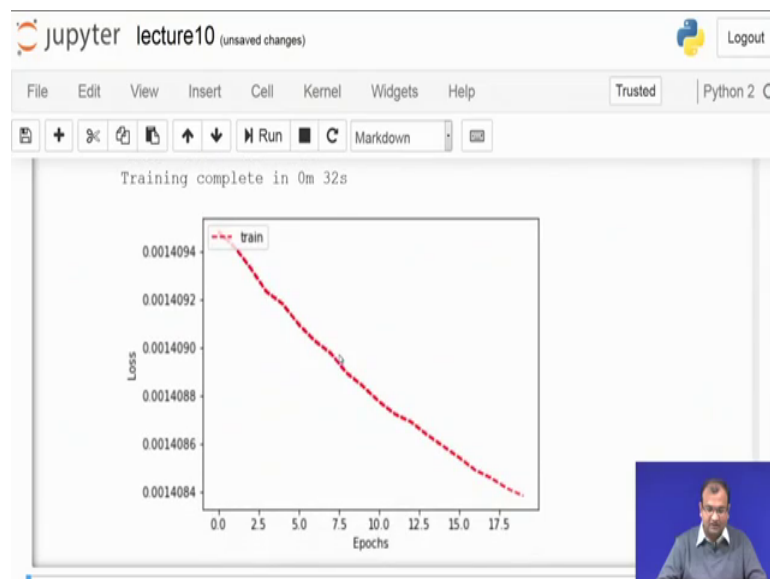
```
Epoch 7/19
Epoch loss: 0.0014090
Epoch complete in 0m 2s
-----
Epoch 8/19
Epoch loss: 0.0014089
Epoch complete in 0m 2s
-----
Epoch 9/19
Epoch loss: 0.0014088
Epoch complete in 0m 2s
```

**Evaluation of trained model**

```
In [9]: # Finding testing accuracy
test_running_corr = 0
```

So, it would like to you it should be a much easier way because, within this library all though it looks like a lot of lines of codes but most of these are modularly written. So, if you want to make change onto your architecture there is just 1 function over there, which defines these architectures and you will just have to make a shorter change.

(Refer Slide Time: 15:12)



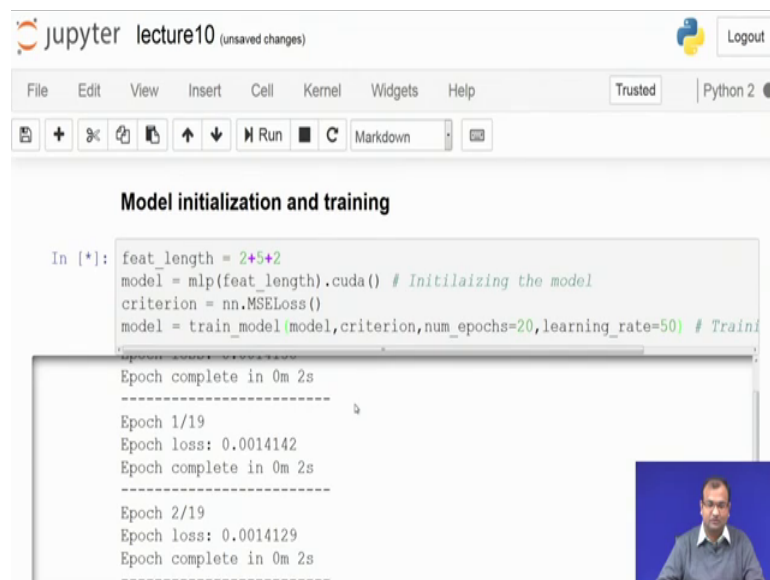
So, by now it is complete it took down about 32 seconds and you can see that it had a bit of jagged behavior and it was much slower in actually falling down. So, the error did decrease to a much lower front as well, but then the rate at which it was decreasing also



we can slower. So, it is not always necessary that just by increasing the number of neurons you will have a much better decrease over there.

So, I can even play around with the learning rate over here say I make down my learning rate as 1 and I can again set that 1 running, so I just need to execute only this part of it keep 1 thing in mind, if you are making some changes to a previous 1 then you need to execute all the subsequent blocks so that it comes down. So, this is on I python notebook so it gives you a much interactive way of solving it out and a much better way of learning, you can always copy all of these into a direct python code for you and it will appear in the same modular way within your python environment.

(Refer Slide Time: 16:15)



The screenshot shows a Jupyter Notebook window titled "lecture10 (unsaved changes)". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for file operations and execution. The notebook content is titled "Model initialization and training" and contains the following code and output:

```
In [*]: feat_length = 2+5+2
model = mlp(feat_length).cuda() # Initilaizing the model
criterion = nn.MSELoss()
model = train_model(model,criterion,num_epochs=20,learning_rate=50) # Traini
```

```
Epoch complete in 0m 2s
-----
Epoch 1/19
Epoch loss: 0.0014142
Epoch complete in 0m 2s
-----
Epoch 2/19
Epoch loss: 0.0014129
Epoch complete in 0m 2s
-----
```

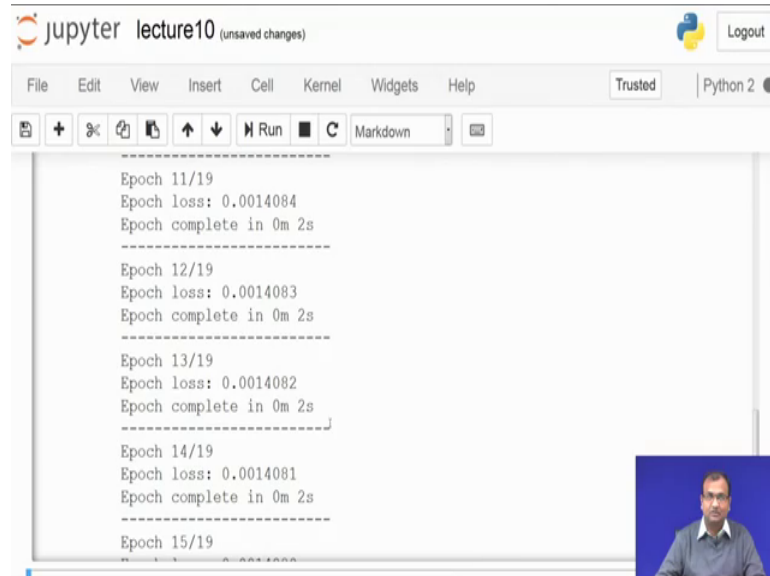
A small video inset in the bottom right corner shows a man speaking.

And you can just execute that py file within whatever is your choice of environment most likely most of us would be making use of anaconda python in order to do that and that becomes easier. So, you see that when I change it to a learning rate of 1, this is how it was falling down and then that was not even a steady fault. So, it became much more linearly falling down and much jagged and the error was also much higher.

So, I can make a change over there and say make this as 50 and see. So, these changes which I am making over there on the learning rate that is the factor, which is going to update my eta in my learning rule and that does make a very important statement in terms of how fast it is going to learn and how fast it is going to converge; all though the

major theoretical reasoning behind that is that this eta is going to scale down my gradients as being proportional to the weights which it needs to update.

(Refer Slide Time: 17:03)



```
jupyter lecture10 (unsaved changes)
File Edit View Insert Cell Kernel Widgets Help Trusted Python 2
Epoch 11/19
Epoch loss: 0.0014084
Epoch complete in 0m 2s
-----
Epoch 12/19
Epoch loss: 0.0014083
Epoch complete in 0m 2s
-----
Epoch 13/19
Epoch loss: 0.0014082
Epoch complete in 0m 2s
-----
Epoch 14/19
Epoch loss: 0.0014081
Epoch complete in 0m 2s
-----
Epoch 15/19
```

There are definitely some more questions which a lot of you can ask which is that every single layer will have a weight of its own dynamic range. So, eta should be different and yes that is also pretty much true and that is what we do observe. So, it is not necessary that always getting down a slower eta is going to make you a better convergent or even getting down of bigger eta is going to give you. So, if I change that even to 500 or 5000 maybe it does not come like this, but it starts jittering over there. And they are some of the interesting stuff you can play around with your learning rates.

So, down the future lectures we would be getting more of a clearer understanding into how this happens and why this happens in more of a way. So, till then keep on enjoying with more of these lectures and do keep on coding as we continue with the theoretical lectures as well and so. Then, we see you for the next class.