

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

Lecture – 13
Fashion MNIST classification using autoencoders

Welcome. So, today we would be doing our examples on autoencoder and continuing down from where we had done. So, in the earlier one where we were doing it with a first and the most easiest one which is called as mnist. So, the whole idea was that can we make neural networks which are fully connected and help autoencoders in order to learn a representations and use that auto encoder learnt representations in order to do classification of handwritten digits and here the problem which comes down is going a bit more than that because in case of your handwritten digits you did see that the background was black and on that the digit part whatever pixels were where the handwriting was present over that was what was the (Refer Time 00:56) white.

So, it was more of like a black versus white pixels which and need to be classified and then the tessellation of these points or the arrangement of these points will help me in order to understand what kind of a digit is written down over there send and this is where you are dealing kind of with a binary data because the data is either 0 or a value of 255.

There is not much of a gray level transition which happens over there. Now what we want to do more of try now doing down is that can we have models and can this auto encoders also learn to identify between grayscale data and for that the first one which we will take down is with a modified version of a another dataset which is called as fashion mnist.

So, here it has images which are of the same size of that of a mnist. So, it is 28 cross 28 pixel images these are all grayscale. So, 0 to 255 full scale values which are available over there and these are small images of clothing items or apparels. So, there are 10 classes of apparels which are present over there and keeping in sync with the concept of mnist. So, in fashion mnist we have 60,000 examples for training and 10,000 examples for testing and you have 10 different object categories in which you need to classify. So, let us get started with what we will do. Since, I have already explained about 2 different kinds of training an autoencoder some one of them was called as the end to end training

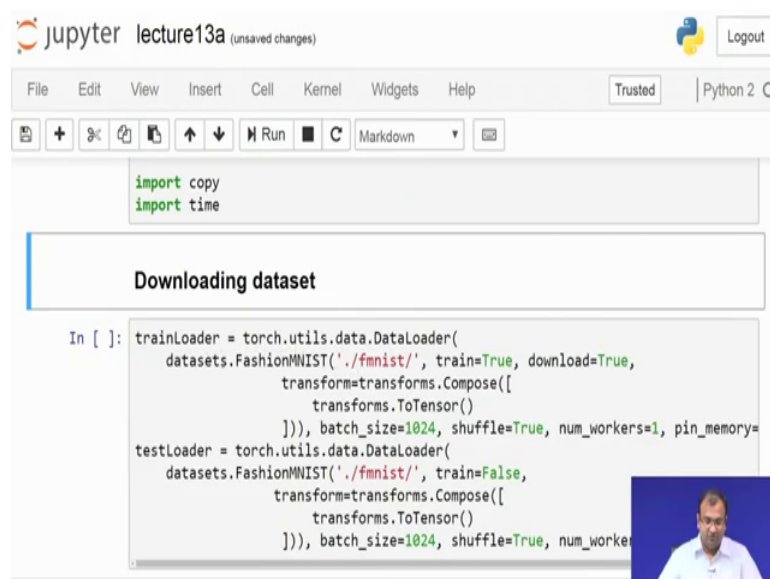
in which you create the complete autoencoder say I have my input layer I have the first hidden layer then the second hidden layer.

Then on my decoder side I should have one hidden one more hidden layer and then my reconstruction output over there. And then I trained this end to end in order to do a representation learning then chop off the decoder part with a hidden layer and the mapping to the output and then I just have something remaining from my input to the first hidden layer to the second hidden layer the third is chopped off or the decoder side first hidden layer is chopped off.

Now, from here I connect it down to a fully connected network neural too. So, this is the first version which is called as an end to end pre training of a auto encoder and then we will use that for MLP initialization in it. So, there is also another version which we have on the example which is called as 13 b. So, on the 13 b example you would be seeing a layer wise pre training of an autoencoder. So, where that was the other way of doing it so that I start with one layer at a time train down the first layer, then train the second layer, then I would be connecting it down to a fully connected layer over there.

So, let us get down to it. The first part is just my header for fetching down all the libraries over there. Let us run the first part of it. And this header is more or less constant for all what I am using. So, there is not much to actually do.

(Refer Slide Time: 03:40)



The screenshot shows a Jupyter Notebook window titled "lecture13a (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 status bar indicating "Trusted" and "Python 2.0". The code cell contains the following Python code:

```
import copy
import time

Downloading dataset

In [ ]: trainLoader = torch.utils.data.DataLoader(
        datasets.FashionMNIST('./fashion1000', train=True, download=True,
                               transform=transforms.Compose([
                                   transforms.ToTensor()
                               ])), batch_size=1024, shuffle=True, num_workers=1, pin_memory=
testLoader = torch.utils.data.DataLoader(
        datasets.FashionMNIST('./fashion1000', train=False,
                               transform=transforms.Compose([
                                   transforms.ToTensor()
                               ])), batch_size=1024, shuffle=True, num_workers=
```

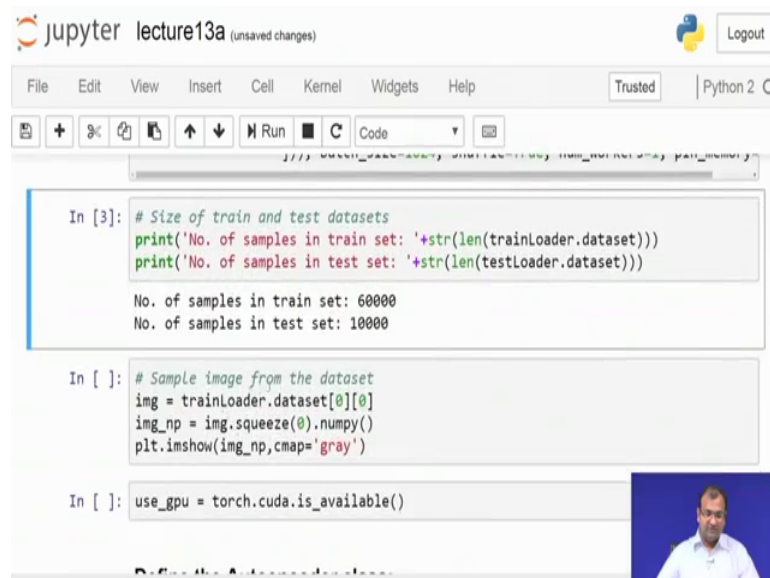
Around over there the data set over here is a bit different. So, you need, but the good thing is that it is available within the torch vision data set itself and you can directly get down get it down. It is called as fashion mnist and now what we do over there is that we just integrate it down multiple of these paths over there. Initially this is the part which was to download all the data that is what I was running down earlier and then once I have the data downloaded then I was again using the path over there and then doing a match loader with it completely.

Now, here we have everything taken down together such that in one single shot I have my training data available to me via the train loader pointer. So, similarly I can do the same thing to get my testing data as well loaded down via the test loader pointer and likewise that I have set my download equal to true. If you are running this for the first time it will start to download the whole data this might take a bit of time, but not significant if you have a good amount of bandwidth available over there.

A couple of minutes and then it downloads and you are good to go and once it is downloaded you can just keep on reusing. The only point is that please do not delete the folder from where the data is placed because every time it is going to look into this one as per this code or if you copy this somewhere then create a fmnist folder or whatever folder where you are just putting on your zip files or tarball images of what got downloaded. Then just put down the exact path over there and that should be solving down your problem.

So, here we have loaded out the data the next is to look into the sample. So, I did mention that we have 60,000 samples in my training set and 10,000 samples in my testing set and that is what comes down over there now from there let us get into the next part.

(Refer Slide Time: 05:24)



The screenshot shows a Jupyter Notebook titled "lecture13a" with the following code and outputs:

```
In [3]: # Size of train and test datasets
print('No. of samples in train set: '+str(len(trainLoader.dataset)))
print('No. of samples in test set: '+str(len(testLoader.dataset)))

No. of samples in train set: 60000
No. of samples in test set: 10000
```

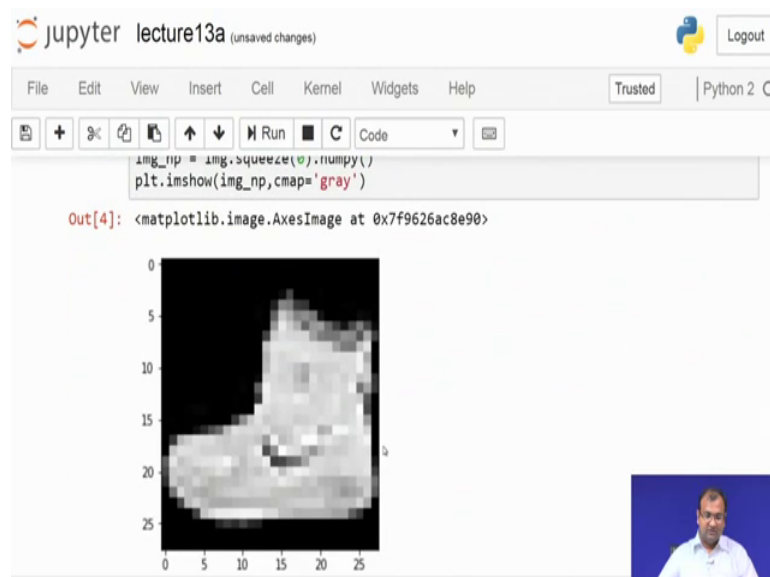
```
In [ ]: # Sample image from the dataset
img = trainLoader.dataset[0][0]
img_np = img.squeeze(0).numpy()
plt.imshow(img_np, cmap='gray')
```

```
In [ ]: use_gpu = torch.cuda.is_available()
```

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


Which is my sample image from the data set.

(Refer Slide Time: 05:30).



The screenshot shows the same Jupyter Notebook interface, but with the output of the image display code cell:

```
Out[4]: <matplotlib.image.AxesImage at 0x7f9626ac8e90>
```

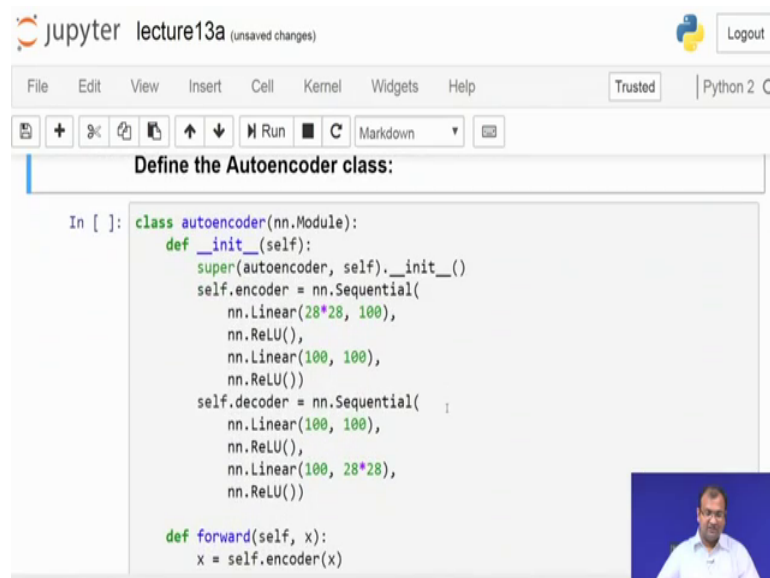


The image shows a grayscale plot of a shoe, likely a sneaker, on a coordinate grid. The x and y axes both range from 0 to 25, with major ticks every 5 units. The shoe is rendered in white and gray pixels against a black background.

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

So, let us try to look into one of these images. This is one image which just comes up for me it looks like a shoe most likely.

(Refer Slide Time: 05:47)



```
Define the Autoencoder class:

In [ ]: class autoencoder(nn.Module):
        def __init__(self):
            super(autoencoder, self).__init__()
            self.encoder = nn.Sequential(
                nn.Linear(28*28, 100),
                nn.ReLU(),
                nn.Linear(100, 100),
                nn.ReLU())
            self.decoder = nn.Sequential(
                nn.Linear(100, 100),
                nn.ReLU(),
                nn.Linear(100, 28*28),
                nn.ReLU())

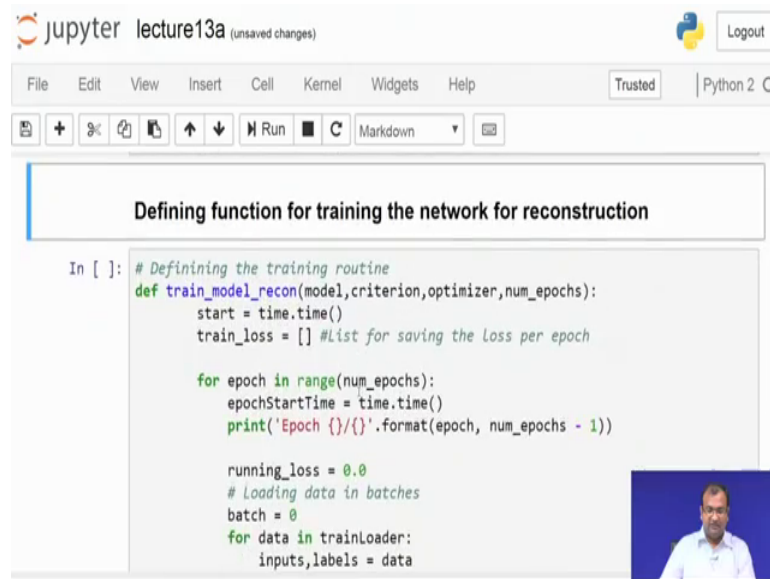
        def forward(self, x):
            x = self.encoder(x)
```

And then if you see it is 28 cross 28 and a perfect grayscale image which is coming down to me.

Now, from there doing a check on the GPU availability or not and here is where I start defining my autoencoder. So, remember that in a autoencoder there were 2 parts of it. One was an encoder another was a decoder. So, here my encoder (Refer Time 05:55) on 784 neurons or 28 cross 28 to 100 neurons that is the first hidden layer from 100 neurons over here after a ReLU or Rectified Linear Unit transformation it connects down to another 100 neurons the second hidden layer over there. Now, similarly my decoder side should have 100 neurons connected to 100 neurons which will map down my like output of the second hidden layer onto something which is conformal and if you vocal to my first hidden layer itself.

Now, once it maps it down I connect them 100 neurons to 784 neurons which goes down to produce an output of the same size as that of the input itself. So, this is my initialization and my definition for my autoencoder over there. Now, with that my forward pass what I need to do is that I would do a forward pass over the encoder then again a forward pass over the decoder and just return that result over there. This is pretty much how my network gets defined. So, let us just define the network and get it running. So, my network gets defined over there the next part is that.

(Refer Slide Time: 07:00)



The screenshot shows a Jupyter Notebook window titled "lecture13a" with "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 main content area displays a Python code cell with the following code:

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

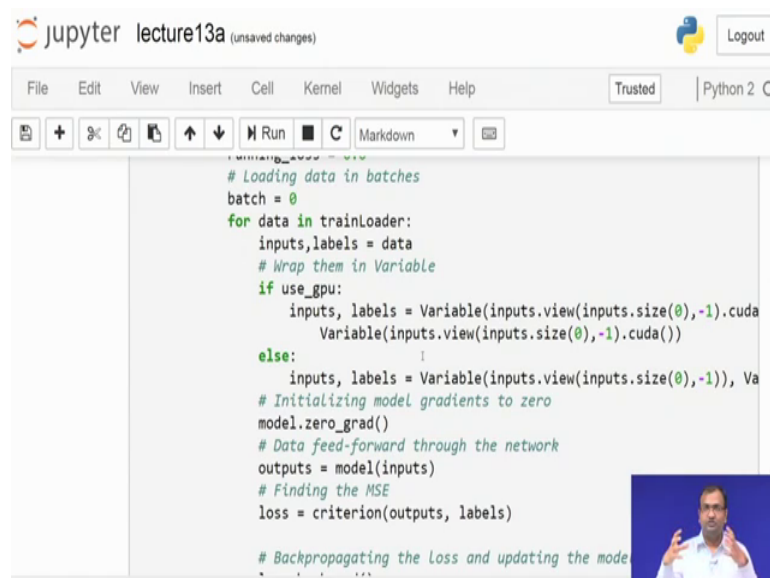
    for epoch in range(num_epochs):
        epochStartTime = time.time()
        print('Epoch {}/{}'.format(epoch, num_epochs - 1))

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

I put start down by training my network.

Now, remember over here. So, we are doing a autoencoder mechanism over there. So, what I would technically be doing is that all my outputs are supposed to be the same as.

(Refer Slide Time: 07:16)



The screenshot shows a Jupyter Notebook window titled "lecture13a" with "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 main content area displays a Python code cell with the following code:

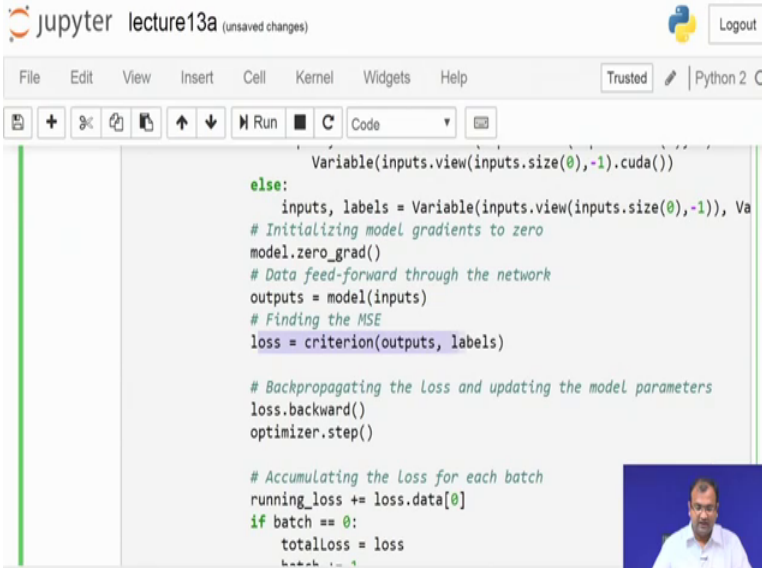
```
# Loading data in batches
batch = 0
for data in trainLoader:
    inputs,labels = data
    # Wrap them in Variable
    if use_gpu:
        inputs, labels = Variable(inputs.view(inputs.size(0),-1).cuda(),
        Variable(inputs.view(inputs.size(0),-1).cuda()))
    else:
        inputs, labels = Variable(inputs.view(inputs.size(0),-1)), Va
    # Initializing model gradients to zero
    model.zero_grad()
    # Data feed-forward through the network
    outputs = model(inputs)
    # Finding the MSE
    loss = criterion(outputs, labels)

# Backpropagating the Loss and updating the mode
```

My input. So, while this is the very generalized form of trainer which has been written down over there though point which we make sure that my labels and my inputs is the same. So, label is technically the term which we would quite often be referring to while saying what is the output coming down from a neuron and because most of the problems

are classification problems. That is the classification label which comes out, but here since I am doing a regression learning problem what I would do is that about I can would put on my input equal to label while my training and then my cost functions appropriately get evaluated over there.

(Refer Slide Time: 07:52)



```
Variable(inputs.view(inputs.size(0),-1).cuda())
else:
    inputs, labels = Variable(inputs.view(inputs.size(0),-1)), Va
# Initializing model gradients to zero
model.zero_grad()
# Data feed-forward through the network
outputs = model(inputs)
# Finding the MSE
loss = criterion(outputs, labels)

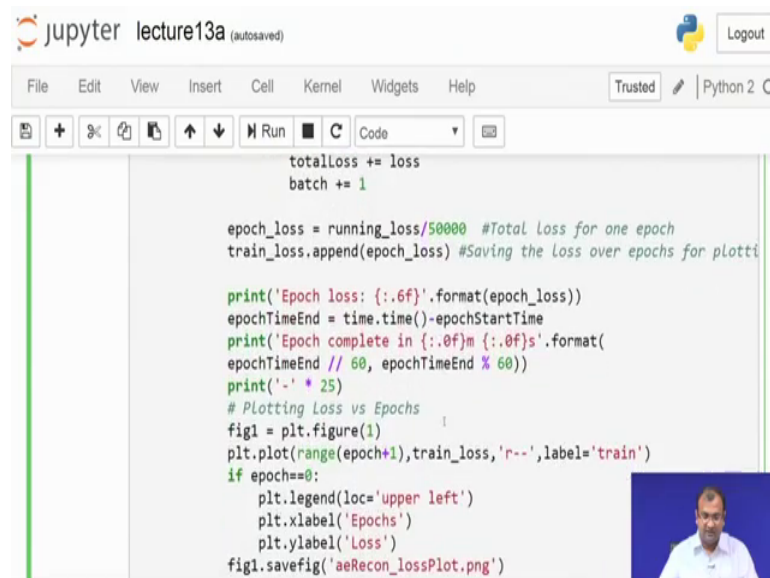
# Backpropagating the Loss and updating the model parameters
loss.backward()
optimizer.step()

# Accumulating the loss for each batch
running_loss += loss.data[0]
if batch == 0:
    totalLoss = loss
    batch = 1
```

So, going through the rest of it is that you zero down the gradients over there and then map down your outputs and inputs over there which gets a forward pass through the model and use a criteria. So, the criterion over here which is defined a bit later on as we had seen in earlier examples here we would be using mean square error MSE criteria because this is just trying to reconstruct itself over there.

Now, from there we go on the learning update rules over there. So, learning update rule is just going to use down my optimizer and a step function over there and we are going to make use of the standard stochastic gradient descent as we have been doing with the other one's as well. Now, that I keep on doing. So, the final part is that I get down my errors and then I can keep on printing them.

(Refer Slide Time: 08:37)



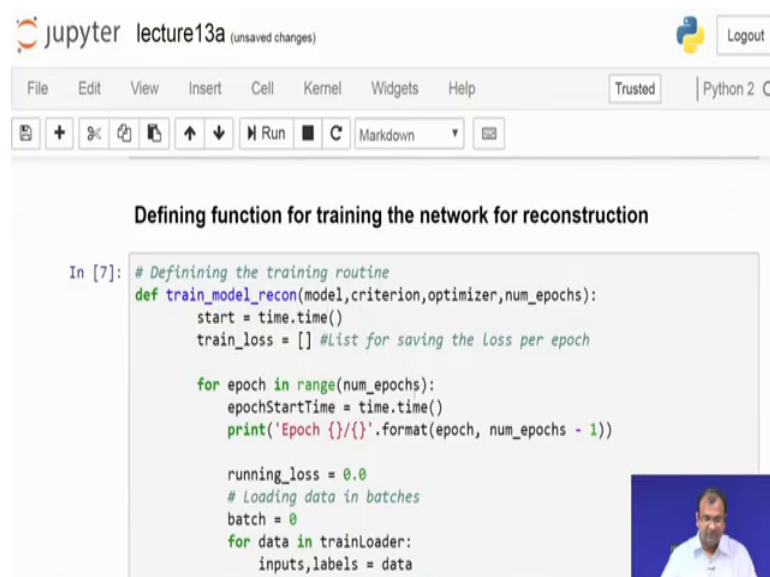
```
totalLoss += loss
batch += 1

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

print('Epoch loss: {:.6f}'.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
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('aeRecon_lossPlot.png')
```

Now, just to see it out. Let us just execute. So, this part gets executed for me.

(Refer Slide Time: 08:48)



```
Defining function for training the network for reconstruction

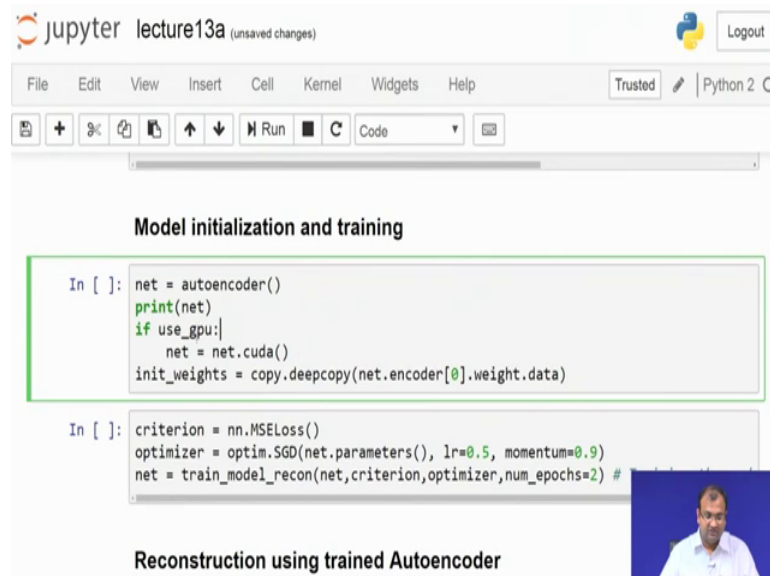
In [7]: # Defining the training routine
def train_model_recon(model,criterion,optimizer,num_epochs):
    start = time.time()
    train_loss = [] #List for saving the loss per epoch

    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
```

And then I have my function for training and this network which gets defined. Now, once and remember that this is just for the auto encoder part we have not yet come into the multilayer perceptron. So, given that I need to initialize my model and then start the training. So, this initialization over here which I do is basically take the whole.

(Refer Slide Time: 09:05)



The screenshot shows a Jupyter Notebook window titled "lecture13a" with "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 is organized into two sections:

Model initialization and training

```
In [ ]: net = autoencoder()
print(net)
if use_gpu:
    net = net.cuda()
init_weights = copy.deepcopy(net.encoder[0].weight.data)
```

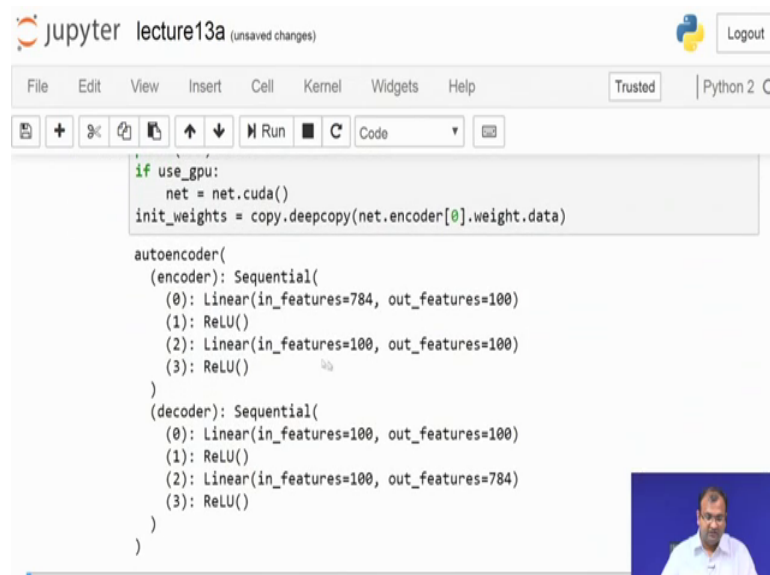
```
In [ ]: criterion = nn.MSELoss()
optimizer = optim.SGD(net.parameters(), lr=0.5, momentum=0.9)
net = train_model_recon(net, criterion, optimizer, num_epochs=2) #
```

Reconstruction using trained Autoencoder

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

Autoencoder model and then see it is randomly initialized. So, one part is just print down and check out whether this exactly what I wanted to see and. You see that this is the one which I wanted to define right 784 neurons going to 100 from 100 to another 100 at the second hidden layer.

(Refer Slide Time: 09:18)



The screenshot shows a Jupyter Notebook window titled "lecture13a" with "unsaved changes". The code defines the structure of an autoencoder model:

```
if use_gpu:
    net = net.cuda()
init_weights = copy.deepcopy(net.encoder[0].weight.data)

autoencoder(
    (encoder): Sequential(
      (0): Linear(in_features=784, out_features=100)
      (1): ReLU()
      (2): Linear(in_features=100, out_features=100)
      (3): ReLU()
    )
    (decoder): Sequential(
      (0): Linear(in_features=100, out_features=100)
      (1): ReLU()
      (2): Linear(in_features=100, out_features=784)
      (3): ReLU()
    )
  )
)
```

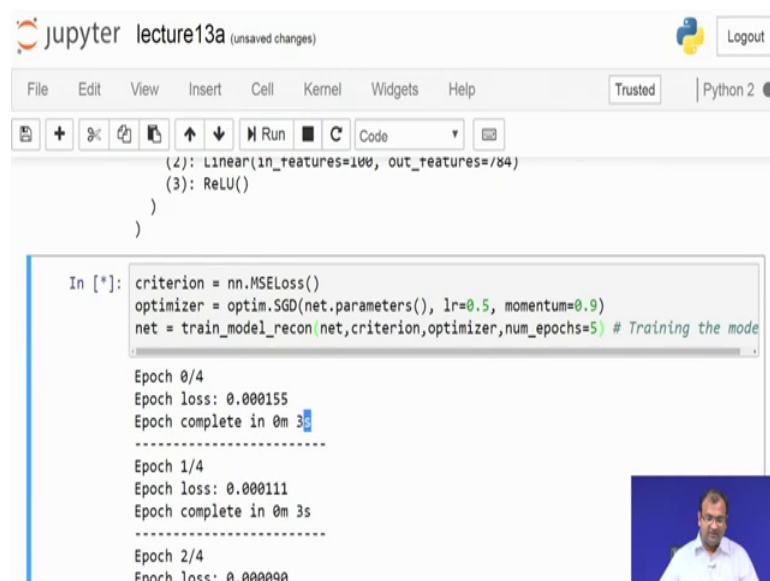
A small video inset of a man is visible in the bottom right corner of the notebook interface.

So, on the decoder side I have a sequential I have a symmetric way of decoding. So, from 100 it goes to 100 and then from there it goes to 784. The weights are again copied down. This is just to check down what is the amount of difference in the weights which

comes down before training and after training. So, from there let us get into the criterion part. So, why criteria for cost function is the Mean Square Error MSE? And the optimizer I am using is SGD with learning rate of 0.5 and a momentum of 0.9. This is quite similar to the one's which we had done in the earlier case with standard mnist as well and then I called on my trainer module and this trainer module function is what was written down over here my reconstruction model or my just an autoencoder model.

Let us run this one and here it is head down to 2. It takes a bit of time. I mean I can change and make it pretty much 5. You cannot run it down for ten's, fifty's, hundred's. But, what we have experienced is more of it goes down to 30 and then whatever it starts reproducing is quite conformal. So, going beyond 30 epochs does not bring any major change over that. However, the only major issue is that when we start doing it takes about 3 seconds in order.

(Refer Slide Time: 10:46)



The screenshot shows a Jupyter Notebook window titled "lecture13a (unsaved changes)". The code cell contains the following Python code:

```
(2): Linear(in_features=100, out_features=784)
(3): ReLU()
)
)

In [*]: criterion = nn.MSELoss()
optimizer = optim.SGD(net.parameters(), lr=0.5, momentum=0.9)
net = train_model_recon(net,criterion,optimizer,num_epochs=5) # Training the mode
```

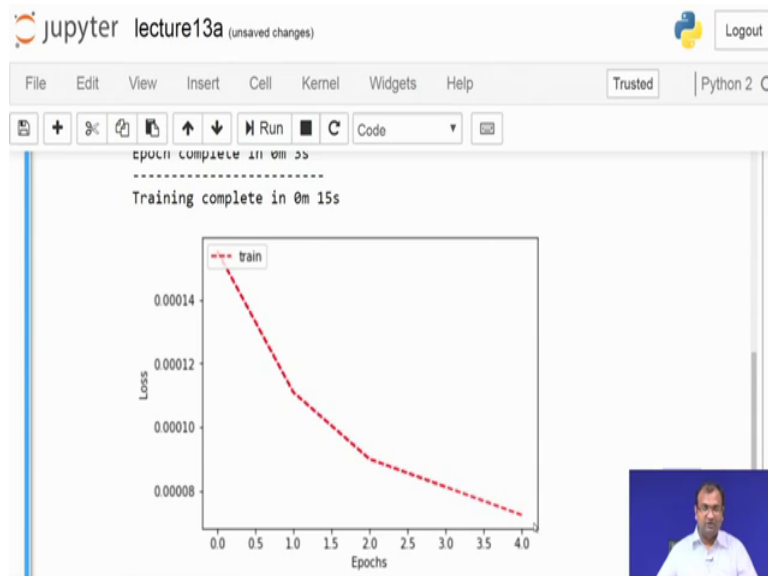
The output of the code cell shows the training progress for 2 epochs:

```
Epoch 0/4
Epoch loss: 0.000155
Epoch complete in 0m 3s
-----
Epoch 1/4
Epoch loss: 0.000111
Epoch complete in 0m 3s
-----
Epoch 2/4
Epoch loss: 0.000090
```

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

To complete each epoch, so that is for the sake of experiments over here within the class we are just keeping it quite lower to just 5 epochs.

(Refer Slide Time: 11:04)



So, 15 seconds and this is where you see that it has come down now it there is a substantial possibility of it to go down even further and keep on decreasing. But I would leave it up to you. So, just make it run down beyond just 5 epochs. Going down to 30 epochs you will get down much slower error coming down. So, this was just the reconstruction error in terms of your Mean Square Error MSE taken out. Now, let us look into.

(Refer Slide Time: 11:18)

```

Reconstruction using trained Autoencoder

In [ ]: TestImg = testloader.dataset[0][0]

if use_gpu:
    outputImg = net(Variable(TestImg.view(TestImg.size(0),-1)).cuda())
else:
    outputImg = net(Variable(TestImg.view(TestImg.size(0),-1)))

outputImg = outputImg.data.view(-1,28,28).cpu()

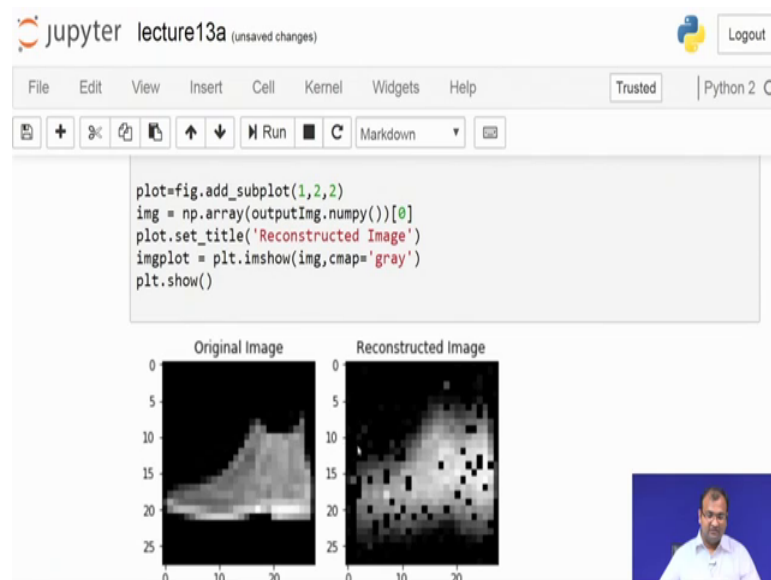
fig = plt.figure()
plot=fig.add_subplot(1,2,1)
img = np.array(TestImg.numpy())[0]
plot.set_title('Original Image')

```

Like how good did the network learn to reconstruct the data it itself.

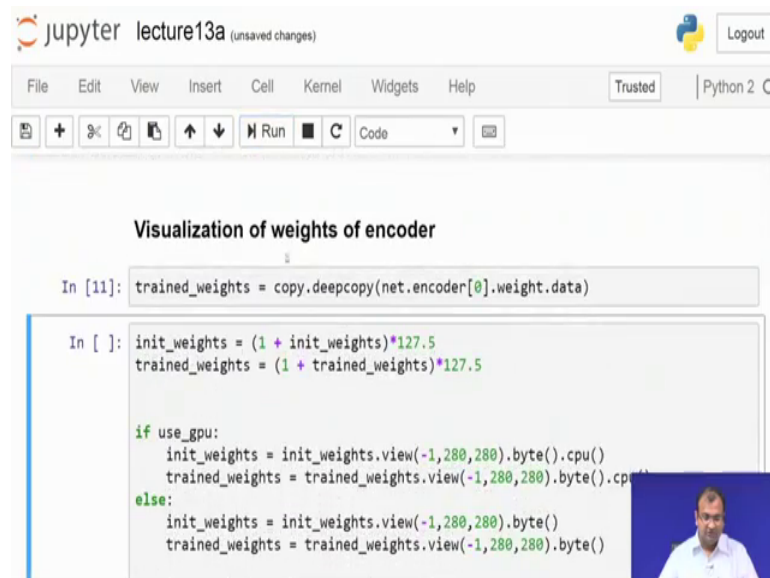
So, if this is the original image which was given to the network this is something what the reconstructed it the foreign factor of it remains almost the same except for that. There are a lot.

(Refer Slide Time: 11:14)



Of these holes which get created over here and there is a bit of blurring and smearing which happens to the original form of the shoe which was there in the original. Now, you keep on training it over a longer period of time you can put down denoising criteria over there and they can pretty much get rid of these small dots coming down and. So, we will be coming to them in a bit later on when we start doing things called as stacked denoising autoencoders and there we will be bringing in the concept in terms of fully connected networks and you can just integrate them and keep on running.

(Refer Slide Time: 12:15).



```
Visualizaton of weights of encoder

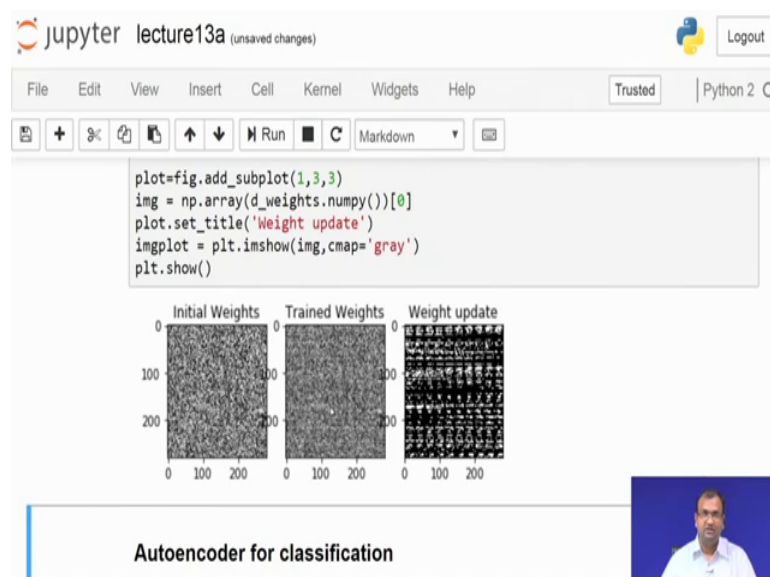
In [11]: trained_weights = copy.deepcopy(net.encoder[0].weight.data)

In [ ]: init_weights = (1 + init_weights)*127.5
        trained_weights = (1 + trained_weights)*127.5

        if use_gpu:
            init_weights = init_weights.view(-1,280,280).byte().cpu()
            trained_weights = trained_weights.view(-1,280,280).byte().cpu()
        else:
            init_weights = init_weights.view(-1,280,280).byte()
            trained_weights = trained_weights.view(-1,280,280).byte()
```

So, here I would just like to look into the weights of the kernel. So, we do a copy of the weights after the whole training process and this is where I just write down a small routine in order to display my initial weights, my train weights and then what are the difference of the weights. Now, if you look into the trained weights as well. So, they too appear as if not learning.

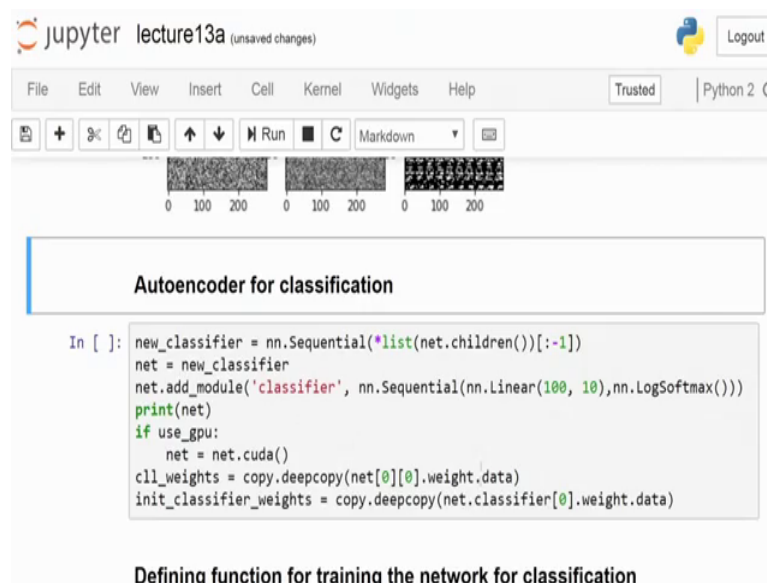
(Refer Slide Time: 12:28)



Down any distinct pattern as such and quite similar to the noisy nature as in the initial weights.

So, but whereas, if you look into the weight difference over then you would see that a substantial number of weights have actually been updated keep on running it for a longer duration of time you will definitely see down more updates coming down. Now, this is where comes the interesting part because here we need to get rid of the decoder part over there and have just the representation learning part available to me and then use this representation learning part as an initialization to my multilayer perceptron. So, I need to connect down my final classification layers over there.

(Refer Slide Time: 13:05)



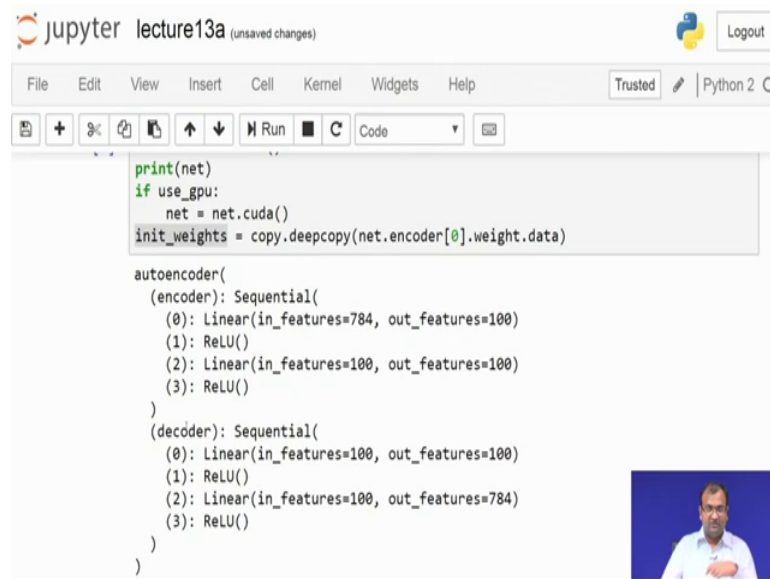
The screenshot shows a Jupyter Notebook interface with the title 'lecture13a (unsaved changes)'. The notebook contains a code cell with the following Python code:

```
In [ ]: new_classifier = nn.Sequential(*list(net.children())[:-1])
net = new_classifier
net.add_module('classifier', nn.Sequential(nn.Linear(100, 10), nn.LogSoftmax()))
print(net)
if use_gpu:
    net = net.cuda()
c1l_weights = copy.deepcopy(net[0][0].weight.data)
init_classifier_weights = copy.deepcopy(net.classifier[0].weight.data)
```

Below the code cell, there is a section titled 'Defining function for training the network for classification'.

So, here what I do is that we would try to define another small network which consists of 100 neurons to just 10 neurons and then append this module on to the initial part over there. So, what I technically do is that I find down just list down all the children on this classifier module n dot sequential this number of children which gets listed over there.

(Refer Slide Time: 13:39)



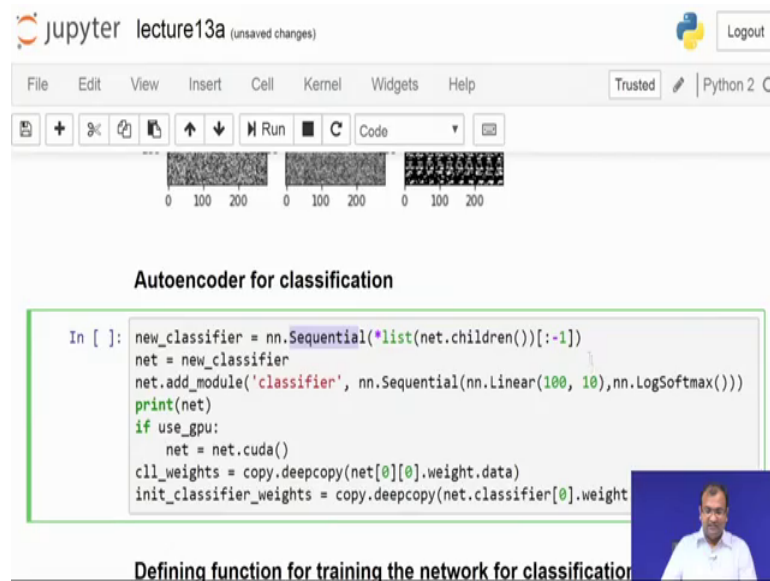
```
print(net)
if use_gpu:
    net = net.cuda()
init_weights = copy.deepcopy(net.encoder[0].weight.data)

autoencoder(
  (encoder): Sequential(
    (0): Linear(in_features=784, out_features=100)
    (1): ReLU()
    (2): Linear(in_features=100, out_features=100)
    (3): ReLU()
  )
  (decoder): Sequential(
    (0): Linear(in_features=100, out_features=100)
    (1): ReLU()
    (2): Linear(in_features=100, out_features=784)
    (3): ReLU()
  )
)
```

If you remember let us get back into the original definition over there. So, my autoencoder over here was defined something which has 2 modules, the first level of the tree. Within encoder module, I have a sequential module and within the sequential model this is how things are in. Then within the decoder, I have a similar kind of a module. So, on my first level of the tree if I try to find out my 2 children then they are encoder and decoder.

Now, if I remove my decoder module then I am just left with my encoder module over here and that is pretty much what has been done over here say remove that point.

(Refer Slide Time: 14:04)



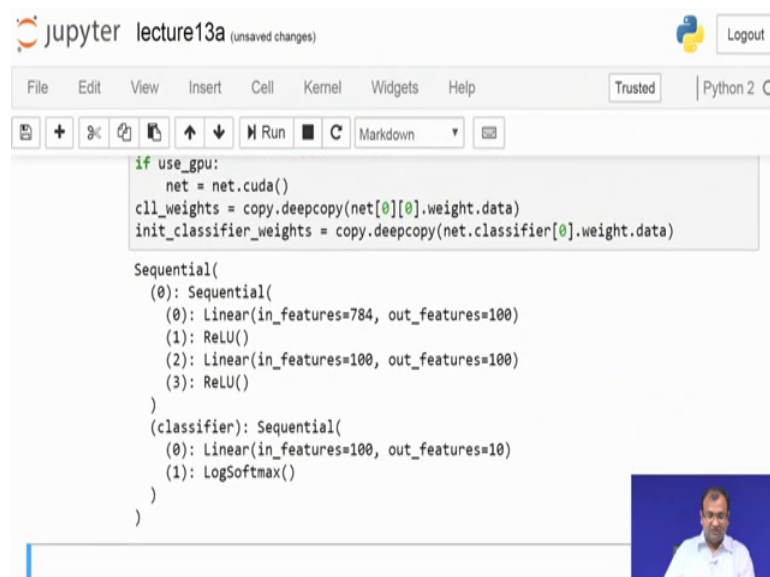
The screenshot shows a Jupyter Notebook window titled "lecture13a" with Python 2 kernel. The code cell contains the following Python code:

```
In [ ]: new_classifier = nn.Sequential(*list(net.children())[:-1])
net = new_classifier
net.add_module('classifier', nn.Sequential(nn.Linear(100, 10), nn.LogSoftmax()))
print(net)
if use_gpu:
    net = net.cuda()
    cll_weights = copy.deepcopy(net[0][0].weight.data)
    init_classifier_weights = copy.deepcopy(net.classifier[0].weight
```

Below the code cell, the text "Defining function for training the network for classification" is visible. A small video inset of a man is in the bottom right corner.

And then just add down my next module. So, from after my encoder I just have this sequential connection and instead of using ReLU which was used just for the purpose of reconstruction we are going to make use of LogSoftmax for classification purpose then just it is just a matter of copying down the weights and keeping them. So, that later on we can as well visualize them.

(Refer Slide Time: 14:30)



The screenshot shows the same Jupyter Notebook window, but the code cell is now empty. The output of the previous code cell is displayed, showing the structure of the neural network:

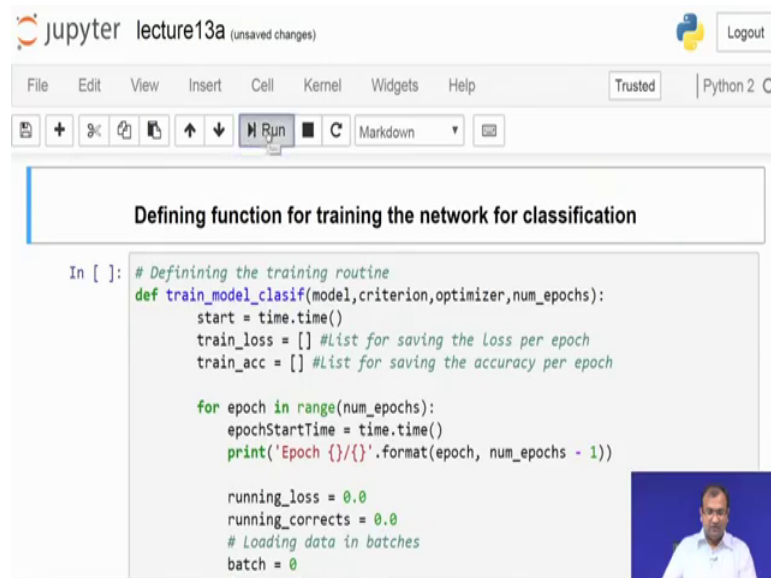
```
Sequential(
  (0): Sequential(
    (0): Linear(in_features=784, out_features=100)
    (1): ReLU()
    (2): Linear(in_features=100, out_features=100)
    (3): ReLU()
  )
  (classifier): Sequential(
    (0): Linear(in_features=100, out_features=10)
    (1): LogSoftmax()
  )
)
```

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

And this is what the network now looks like. So, your initial encoder part remains the same the weights are also preserved over there because I did not play around with the

weights or reinitialize them and then I added down another sequential part which connects 100 neurons to 10 neurons over there and then has a LogSoftmax of non-linearity present.

(Refer Slide Time: 14:54).



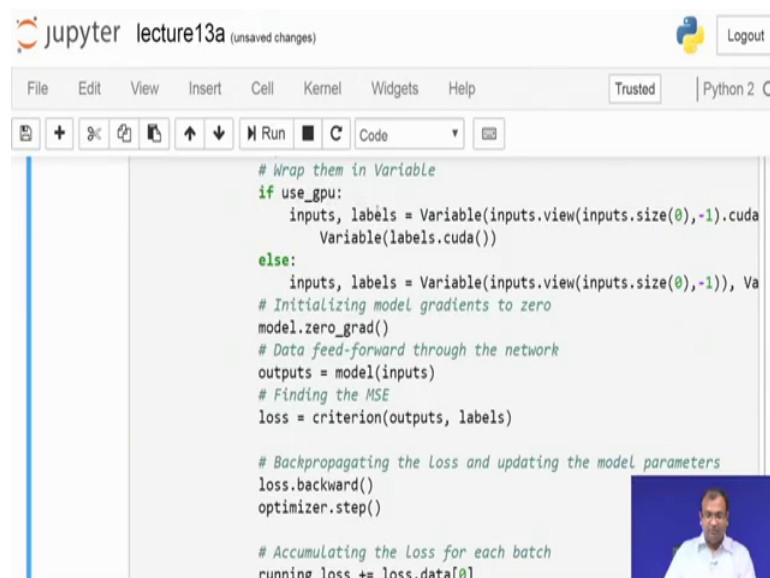
```
In [ ]: # Defining the training routine
def train_model_clasif(model,criterion,optimizer,num_epochs):
    start = time.time()
    train_loss = [] #List for saving the loss per epoch
    train_acc = [] #List for saving the accuracy per epoch

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

        running_loss = 0.0
        running_corrects = 0.0
        # Loading data in batches
        batch = 0
```

So, once done we define the trainer function over here. So, this is quite same except for the fact that my inputs and labels are not different.

(Refer Slide Time: 15:00)



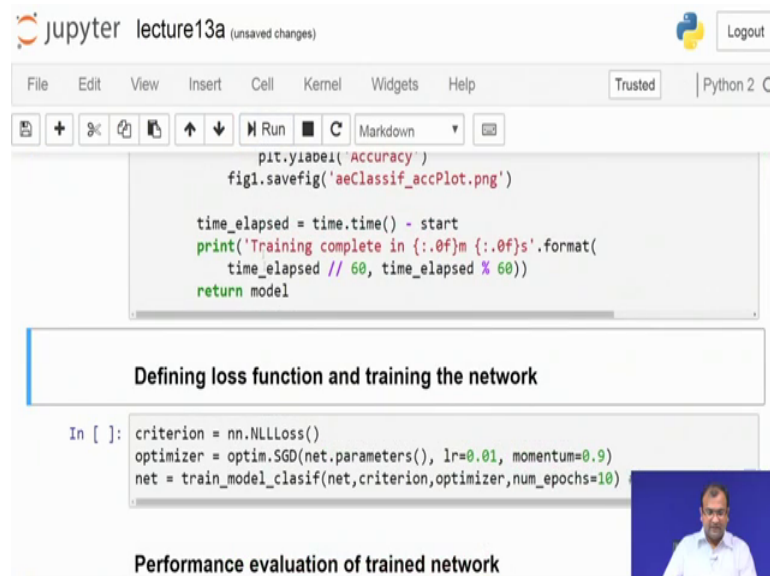
```
# Wrap them in Variable
if use_gpu:
    inputs, labels = Variable(inputs.view(inputs.size(0),-1).cuda(),
                             Variable(labels.cuda()))
else:
    inputs, labels = Variable(inputs.view(inputs.size(0),-1)), Va
# Initializing model gradients to zero
model.zero_grad()
# Data feed-forward through the network
outputs = model(inputs)
# Finding the MSE
loss = criterion(outputs, labels)

# Backpropagating the loss and updating the model parameters
loss.backward()
optimizer.step()

# Accumulating the loss for each batch
running_loss += loss.data[0]
```

So, I will use down labels which are actual class labels coming down over there. My criterion function will actually be changing over there as well. And I will be.

(Refer Slide Time: 15:14)



The screenshot shows a Jupyter Notebook window titled "lecture13a (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 in the editor is as follows:

```
plt.ylabel('Accuracy')
fig1.savefig('aeClassif_accPlot.png')

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

Below the code editor, there is a section titled "Defining loss function and training the network" with the following code:

```
In [ ]: criterion = nn.NLLLoss()
optimizer = optim.SGD(net.parameters(), lr=0.01, momentum=0.9)
net = train_model_clasif(net,criterion,optimizer,num_epochs=10)
```

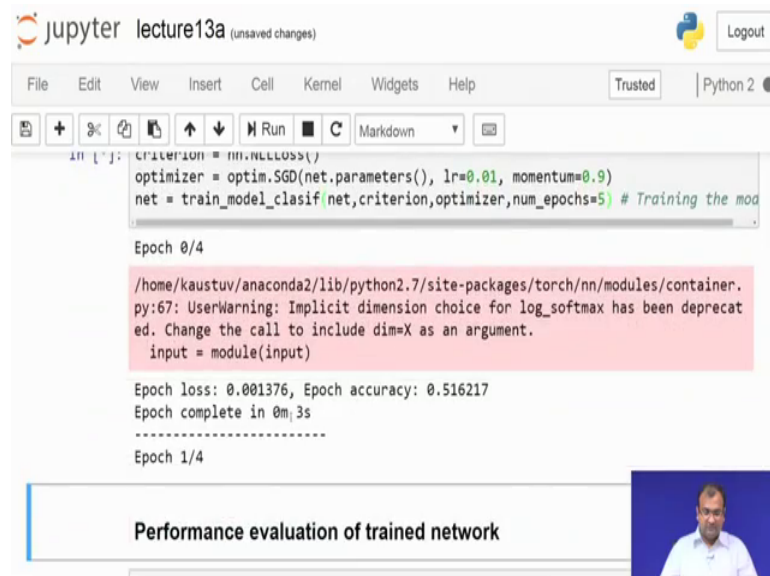
At the bottom of the notebook, there is a section titled "Performance evaluation of trained network" and a small video thumbnail of a man speaking.

Using a different learning rate in order to do, this is just your trainer for the new network which you defined over there and here when I come down to my actual part of what functions to use and how to start training down.

So, I am going to make use of negative log likelihood criterion over there and that corresponds to the final non-linear transformation which was LogSoftmax which is used which is compliant to the dynamic range and the requirements for negative log likelihood to come into play and then I set down my stochastic gradient descent, but here my learning rate is pretty different because earlier I was just using 0.5 here I am using a 0.01 that is to keep down with the dynamic range of the errors which I get down my momentum still remains at the same and I choose to do it with 10 epochs. I can even reduce this and for the sake of time. I would just put down 5 epochs over here.

So, let us run this one and then you see pretty much it takes.

(Refer Slide Time: 16:04)



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

```
in [ ]: criterion = nn.NLLLoss()
optimizer = optim.SGD(net.parameters(), lr=0.01, momentum=0.9)
net = train_model_clasif(net,criterion,optimizer,num_epochs=5) # Training the model
```

Epoch 0/4

```
/home/kaustuv/anaconda2/lib/python2.7/site-packages/torch/nn/modules/container.py:67: UserWarning: Implicit dimension choice for log_softmax has been deprecated. Change the call to include dim=X as an argument.
input = module(input)
```

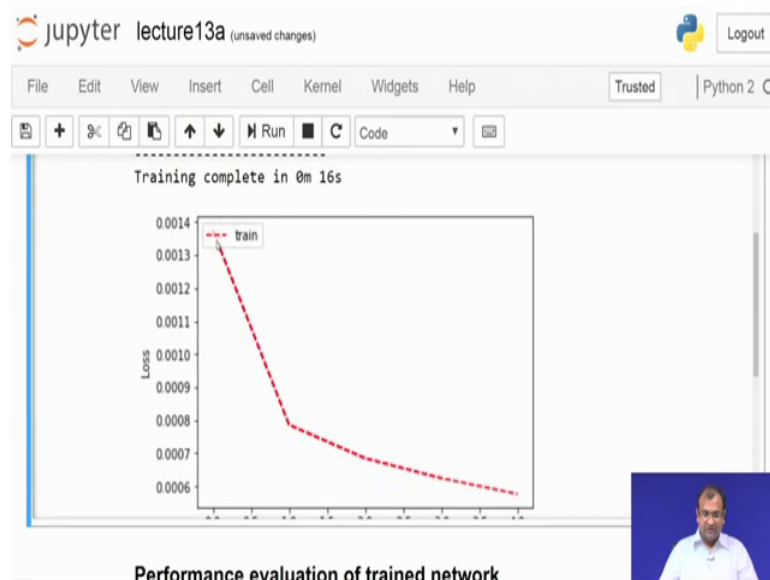
Epoch loss: 0.001376, Epoch accuracy: 0.516217
Epoch complete in 0m 3s

Epoch 1/4

Performance evaluation of trained network

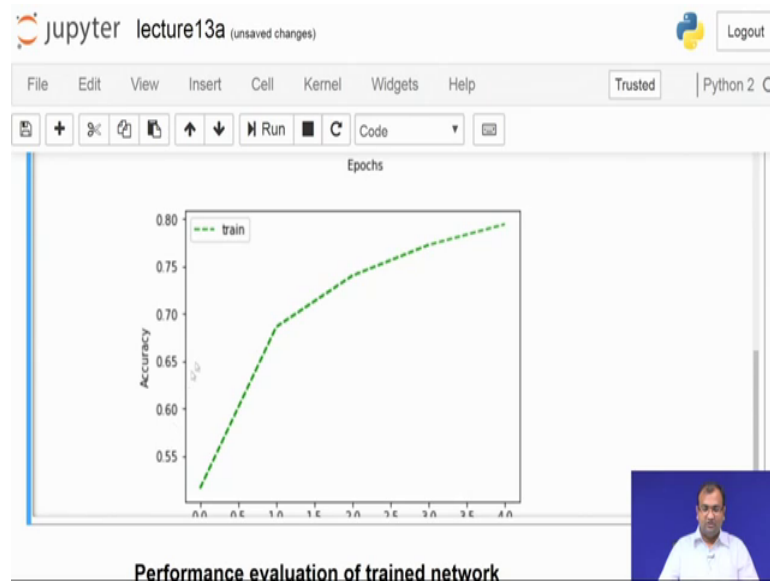
Almost the same amount of time 3 seconds per epoch. So, over 5 epoch it should be done within 15 seconds. And you can now look into the accuracy which keeps on coming down over this. So, as the error keeps on going down your accuracy also keeps on increasing.

(Refer Slide Time: 16:25).



So, this is my training loss curve. This is no more the mean square error, but this.

(Refer Slide Time: 16:39)

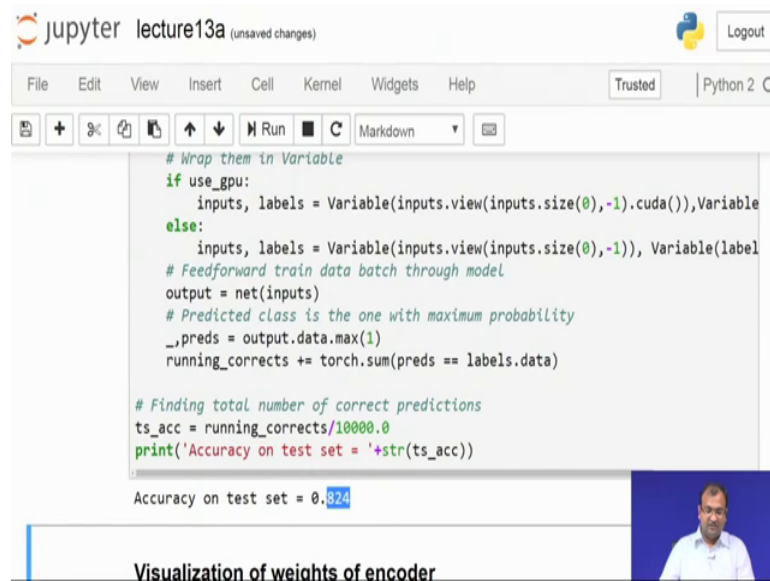


Is a negative log likelihood error which comes down and here is my accuracy. So, this accuracy is not in percentage, but does absolute scale of 0 to 1. You can pretty much see that it is close to 80 percent accuracy just by training down in 5 epochs and that is amazing because the point is that when it was not trained it started down around with the 50 percent accuracy which is just a random chance that anything it just classifies as anything else and from there it from a 50 percent accuracy going down to about 80 percent accuracy is not that bad if it just a few minutes.

So, it is not always that your deep neural networks for solving your problems will actually take a long amount of time it does not and we are not even using a very powerful GPU as such. It is a very standard desktop great gaming GPU with just 2 GB of RAM which is available for us to use. So, here we do the performance evaluation for the network. So, we evaluate `pv` write down the function in order to find out what is the total testing error which comes down.

So, what it is supposed to do over here is that it will get down and load all the images which comes down on your testing data and from there it will feed forward each of these test images onto your network and eventually calculate whether the accuracy has been whether it has been correctly predicted or not now in the training case you did.

(Refer Slide Time: 17:50)



```
# Wrap them in Variable
if use_gpu:
    inputs, labels = Variable(inputs.view(inputs.size(0),-1).cuda()), Variable(labels.view(labels.size(0),-1).cuda())
else:
    inputs, labels = Variable(inputs.view(inputs.size(0),-1)), Variable(labels.view(labels.size(0),-1))
# Feedforward train data batch through model
output = net(inputs)
# Predicted class is the one with maximum probability
_, preds = output.data.max(1)
running_corrects += torch.sum(preds == labels.data)

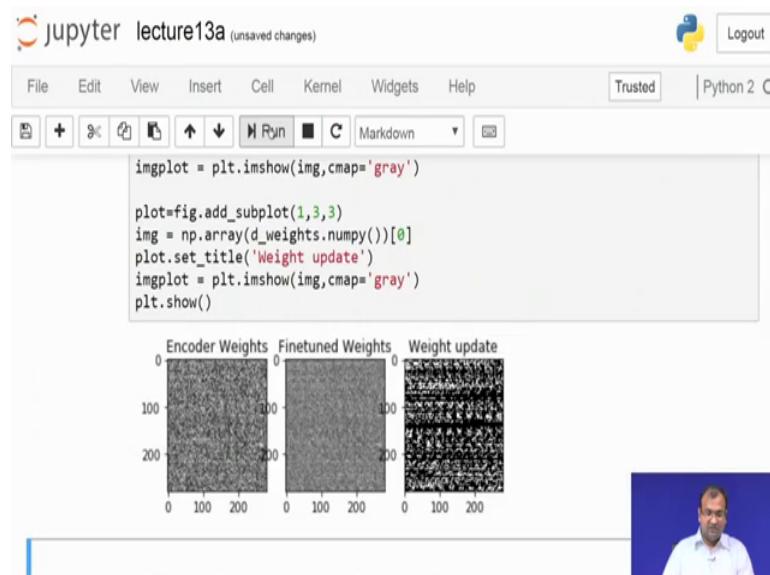
# Finding total number of correct predictions
ts_acc = running_corrects/10000.0
print('Accuracy on test set = '+str(ts_acc))
```

Accuracy on test set = 0.824

Visualization of weights of encoder

See that it was closed to 80 percent and in test it comes down to almost 82.4 percent of accuracy which given the fact that is not a bad because none of these testing examples are what the network had earlier seen in anyways. And let us get into what happened down to the weights after all of this training go down.

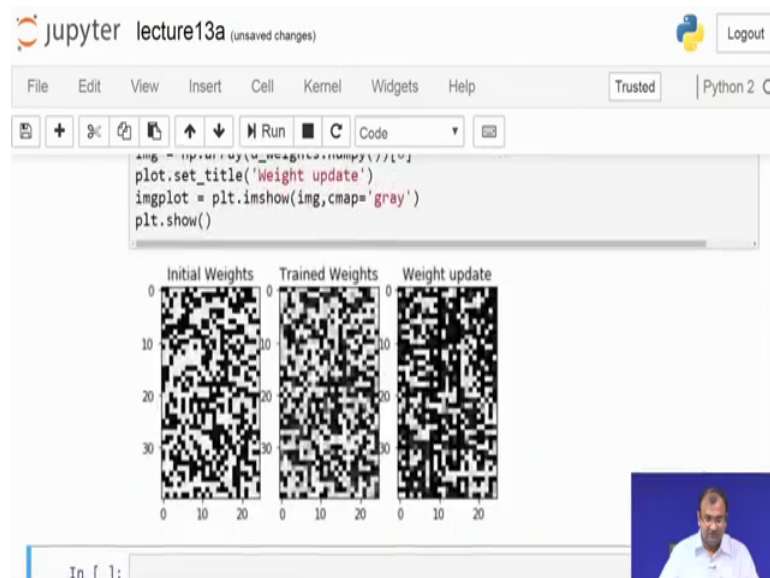
(Refer Slide Time: 18:09)



So, after my fine tuning pitches when I have this multilayer perceptron coming into it and modifying everything, my weights are still working out because this was at the end of the first feature representation learning going down to it and then at the end of

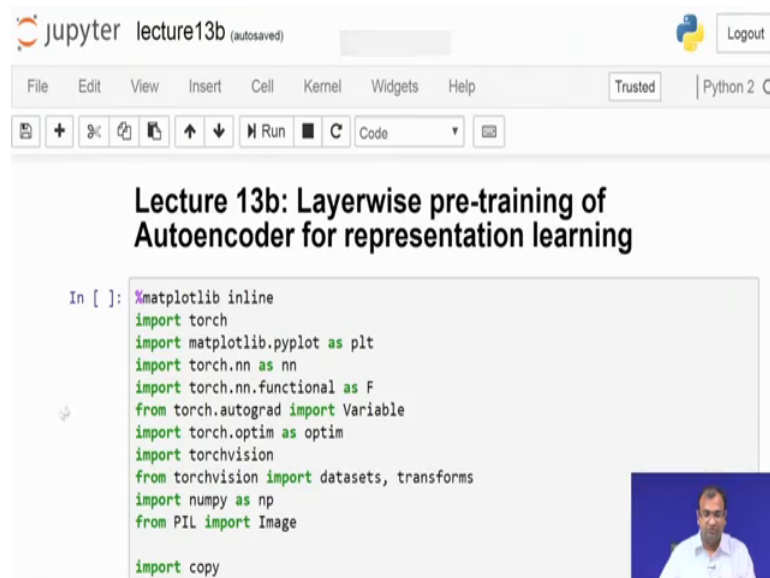
refinement with a classification coming this is what happened down as I change and there have been a significant amount of change you would see that they are getting down more smoother and smoother. Run them down for more number of epochs not just restrict them to 5 epochs, but say 30 epoch you would start getting out beautiful emergent patterns coming which can represent this data in a much better way.

(Refer Slide Time: 18:48)



So, here we try to look into the weights of the classifier. So, this is what the classifier initial weights which were randomly. So, this is the last layer which we added down from 100 to just 10 neurons over there. So, this is what it was initially looking like this is after the training over there and these are the weights which got updated at the magnitude. So, this is one of the versions which works out and the end to end pre training mont. The next one which we do is a 13b and what this does is that instead of doing an end to end we try to look into layer wise pre training or the second approach of doing it.

(Refer Slide Time: 19:22)



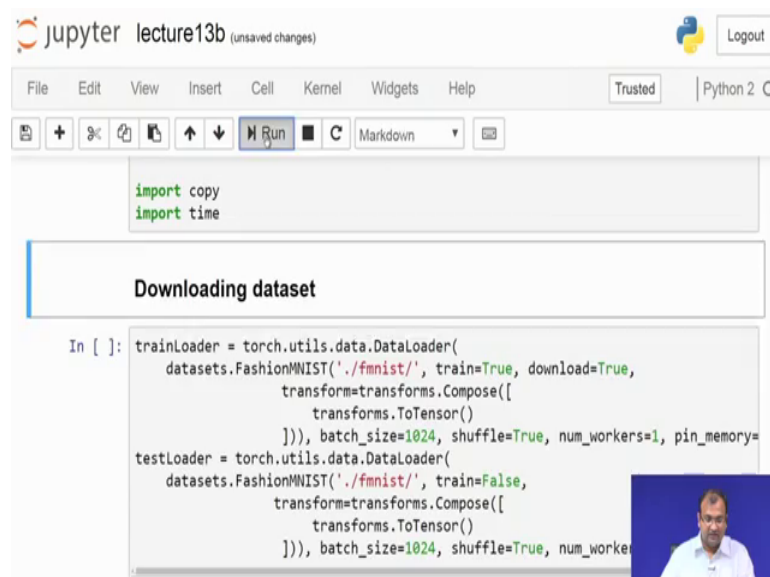
The screenshot shows a Jupyter Notebook window titled 'lecture13b (autosaved)'. The notebook has a menu bar with 'File', 'Edit', 'View', 'Insert', 'Cell', 'Kernel', 'Widgets', and 'Help'. Below the menu bar is a toolbar with icons for file operations and a 'Run' button. The main content area displays the title 'Lecture 13b: Layerwise pre-training of Autoencoder for representation learning' in a large, bold font. Below the title is a code cell with the following Python code:

```
In [ ]: %matplotlib inline
import torch
import matplotlib.pyplot as plt
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
import torch.optim as optim
import torchvision
from torchvision import datasets, transforms
import numpy as np
from PIL import Image

import copy
```

So, going down with the flow over there, this is my header which just works out fine.

(Refer Slide Time: 19:28)



The screenshot shows a Jupyter Notebook window titled 'lecture13b (unsaved changes)'. The notebook has a menu bar with 'File', 'Edit', 'View', 'Insert', 'Cell', 'Kernel', 'Widgets', and 'Help'. Below the menu bar is a toolbar with icons for file operations and a 'Run' button. The main content area displays a code cell with the following Python code:

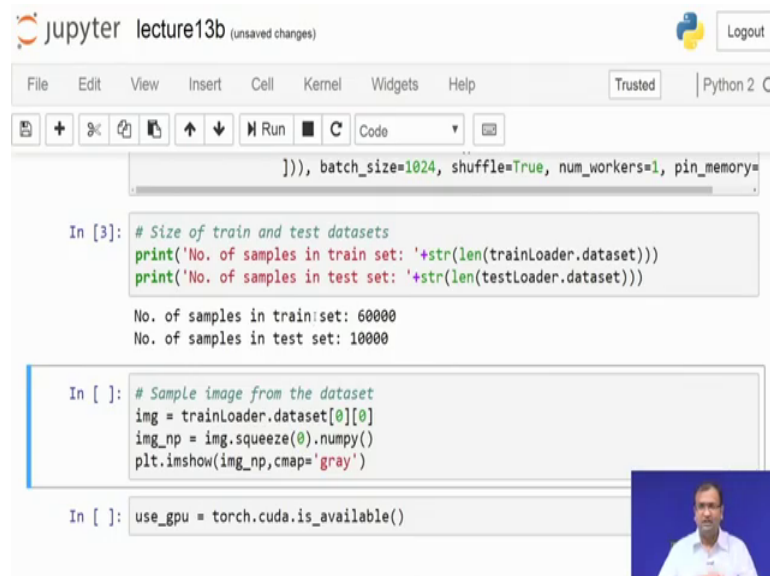
```
import copy
import time
```

Below the code cell is a message box with the text 'Downloading dataset'. Below the message box is another code cell with the following Python code:

```
In [ ]: trainLoader = torch.utils.data.DataLoader(
    datasets.FashionMNIST('./fmnist/', train=True, download=True,
        transform=transforms.Compose([
            transforms.ToTensor()
        ])), batch_size=1024, shuffle=True, num_workers=1, pin_memory=
testLoader = torch.utils.data.DataLoader(
    datasets.FashionMNIST('./fmnist/', train=False,
        transform=transforms.Compose([
            transforms.ToTensor()
        ])), batch_size=1024, shuffle=True, num_worke
```

Then I have my data set loader which also works out fine and.

(Refer Slide Time: 19:32)



```
    ]), batch_size=1024, shuffle=True, num_workers=1, pin_memory=

In [3]: # Size of train and test datasets
print('No. of samples in train set: '+str(len(trainLoader.dataset)))
print('No. of samples in test set: '+str(len(testLoader.dataset)))

No. of samples in train set: 60000
No. of samples in test set: 10000

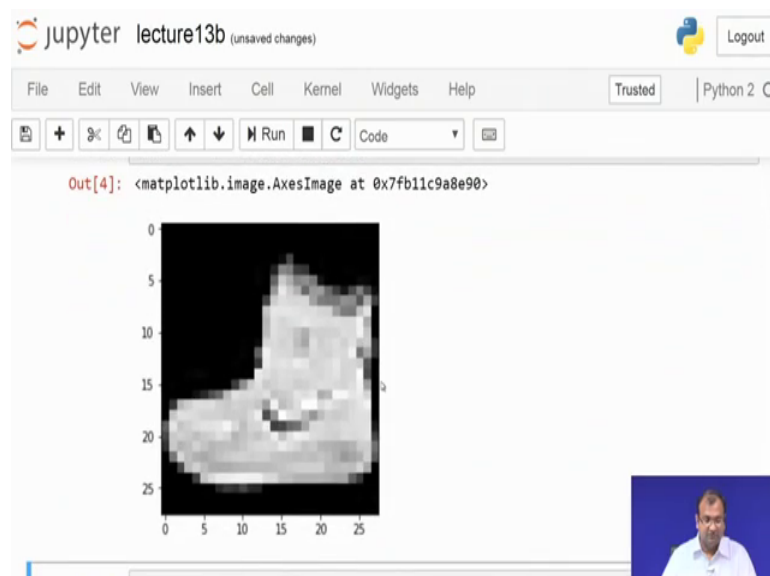
In [ ]: # Sample image from the dataset
img = trainLoader.dataset[0][0]
img_np = img.squeeze(0).numpy()
plt.imshow(img_np, cmap='gray')

In [ ]: use_gpu = torch.cuda.is_available()
```

So, these are all standalone independent codes and we made it a point that instead of juggling letting you juggle between copying down which snippet and pasting it over there and then run down the whole scratch we have written down complete versions of it and which are provided for your cues you can download each notebook and each is a self sufficient exercise with the complete module written down over there.

So, that you do not face a problem for your initial trials and learning exercises. So, we take down.

(Refer Slide Time: 20:00)

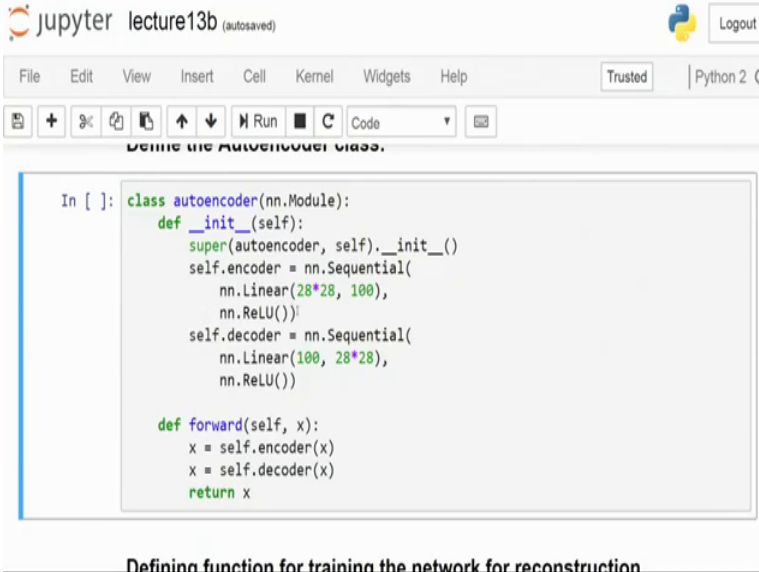


```
Out[4]: <matplotlib.image.AxesImage at 0x7fb11c9a8e90>
```



And it comes down to the same image itself. It because we just did not randomly fetched. We fetched out the first instance of the image from there. So, run that down and here we come down to the definition over here. Now for my autoencoder class in a. So, layer wise pre training the whole idea was that you train one layer at a time. So, if I have 2 hidden layers of 100 neurons, my first job is I will train down my first hidden layer of 100 neurons and then I would go down to.

(Refer Slide Time: 20:31)



The screenshot shows a Jupyter Notebook interface with the title 'lecture13b (autosaved)'. The notebook contains a code cell with the following Python code defining an autoencoder class:

```
In [ ]: class autoencoder(nn.Module):
        def __init__(self):
            super(autoencoder, self).__init__()
            self.encoder = nn.Sequential(
                nn.Linear(28*28, 100),
                nn.ReLU())
            self.decoder = nn.Sequential(
                nn.Linear(100, 28*28),
                nn.ReLU())

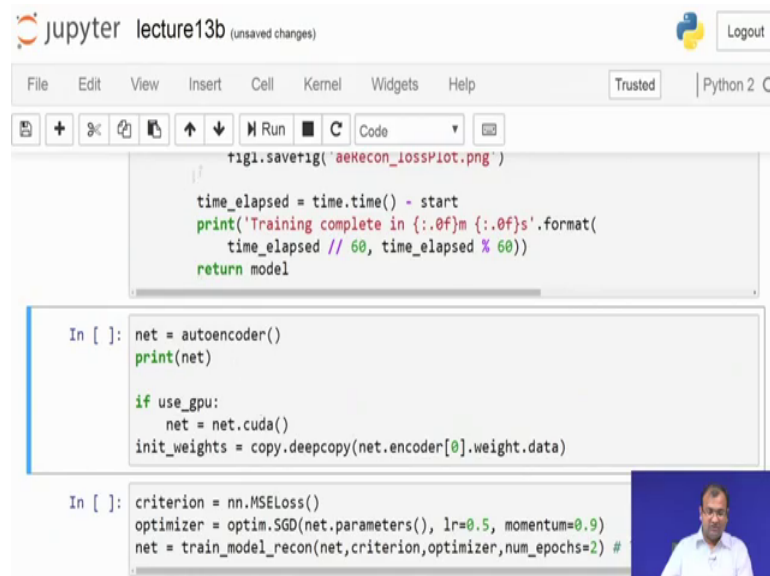
        def forward(self, x):
            x = self.encoder(x)
            x = self.decoder(x)
            return x
```

Below the code cell, the text 'Defining function for training the network for reconstruction' is visible.

My second one, this is how I define my encoder and decoder. So, the and first encoder unit takes in 784 neurons goes to 100 and then from 100 it reconstructs the 784 neurons one second.

Now, that this is defined. So, we will be defining our whole network. Here the input and output is going to be this still the same and this is what comes down. So, if I run and execute my trainer function then I can actually.

(Refer Slide Time: 20:59)



```
fig1.savefig('aeRecon_lossPlot.png')

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

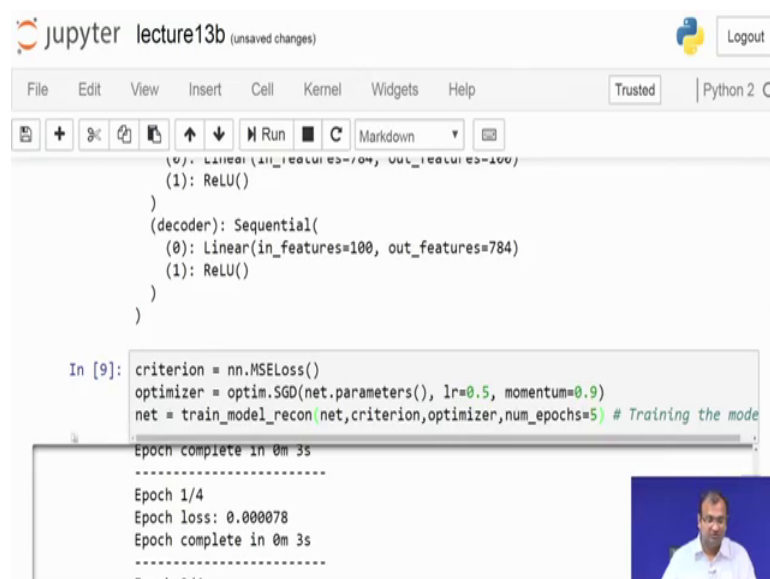
In [ ]: net = autoencoder()
print(net)

if use_gpu:
    net = net.cuda()
init_weights = copy.deepcopy(net.encoder[0].weight.data)

In [ ]: criterion = nn.MSELoss()
optimizer = optim.SGD(net.parameters(), lr=0.5, momentum=0.9)
net = train_model_recon(net,criterion,optimizer,num_epochs=2) #
```

Look into my network get it transported onto a GPU. So, this is what it looks like and copied down the weights in order to visualize it out now subsequent to that I have my standard MSELoss function is SGD as an optimizer with the same momentum and learning rates as I was doing and then I just train it for a few epochs. So, I change it down and make it 5 epochs and I just keep on training.

(Refer Slide Time: 21:28)



```
(0): Linear(in_features=784, out_features=100)
(1): ReLU()
)
(decoder): Sequential(
  (0): Linear(in_features=100, out_features=784)
  (1): ReLU()
)
)

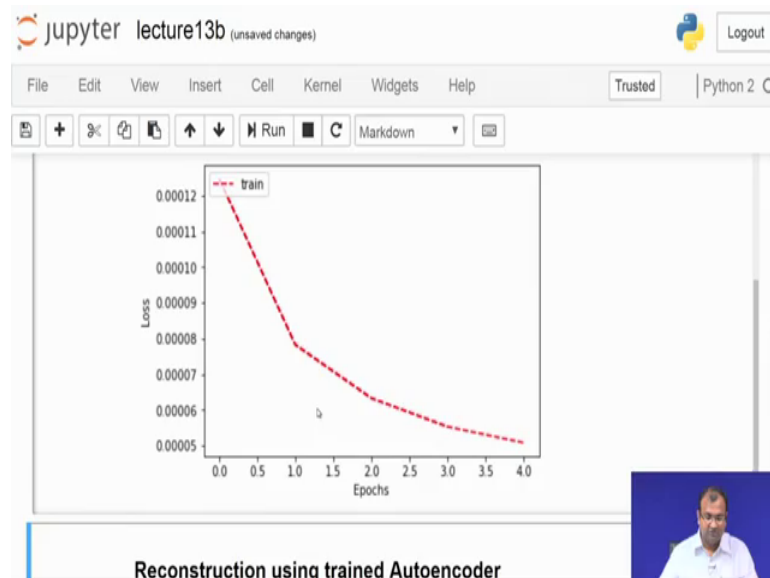
In [9]: criterion = nn.MSELoss()
optimizer = optim.SGD(net.parameters(), lr=0.5, momentum=0.9)
net = train_model_recon(net,criterion,optimizer,num_epochs=5) # Training the mode

Epoch complete in 0m 3s
-----
Epoch 1/4
Epoch loss: 0.000078
Epoch complete in 0m 3s
-----
Epoch 2/4
```

So, the course which will be available on GitHub may have some variations from what is present over there in terms of this number of epochs and arguments they are pretty much

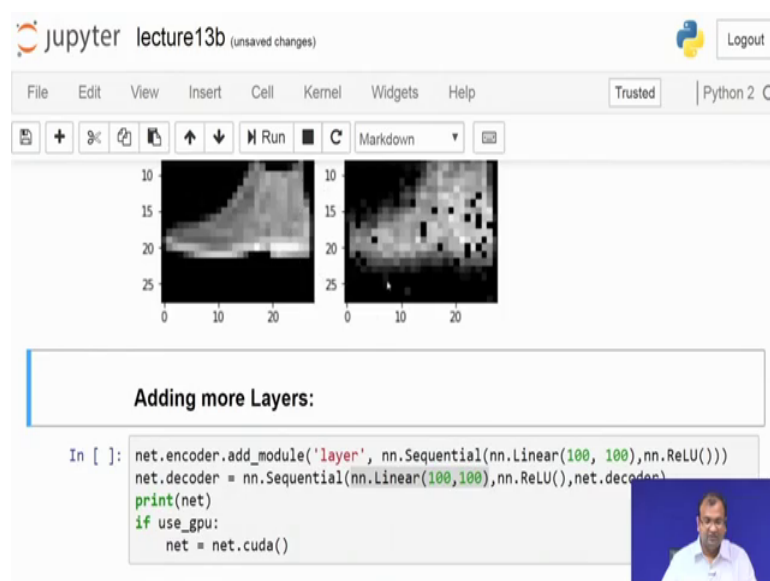
commented and I am showing it to you as to which points to be changed over there. So, just play around with them we might put down deliberately 30 epochs over there. So, that you run initially for 30 epochs and see.

(Refer Slide Time: 21:47)



So, this is what comes down with just one single hidden neuron. Now, here I would like to see just that one single hidden layer what is the kind of reconstruction which I get down.

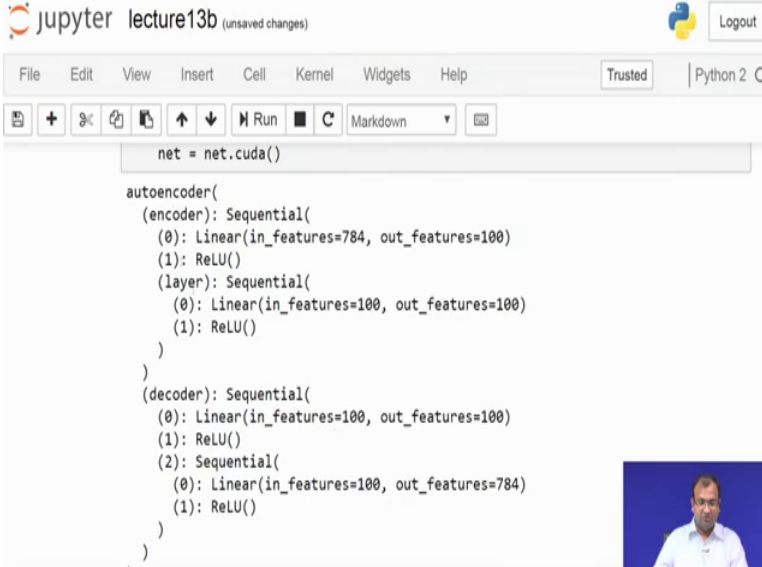
(Refer Slide Time: 21:59)



Now, the next part is this is when my first bunch of hidden layer connections are already trained now the objective is that I would remove my connection from 100 neurons to 784 neurons and rather replace that one with another 100 over there. So, I can look into this as splitting up the decoder encoder and decoder as in the earlier case. So, I have 784 to 100. Now, I will bring in a match of 100 to 100. This gets my second hidden layer. From my second hidden layer to the earlier hidden layer, there will be another neuron connection which has to be established.

That is what I am doing pretty much over here. First, what I am doing is on the encoder model I am adding a module. So, this will append itself to the 784 to 100 neurons. So, after 100 to another 100 in the decoder side I need to redefine this one by saying that first I will add 100 to 100 and then the rest of the decoder gets added over here. And that is what I define over here. Now, once we do that and look into the network this is how the network looks like.

(Refer Slide Time: 23:00)



```
net = net.cuda()

autoencoder(
  (encoder): Sequential(
    (0): Linear(in_features=784, out_features=100)
    (1): ReLU()
    (layer): Sequential(
      (0): Linear(in_features=100, out_features=100)
      (1): ReLU()
    )
  )
  (decoder): Sequential(
    (0): Linear(in_features=100, out_features=100)
    (1): ReLU()
    (2): Sequential(
      (0): Linear(in_features=100, out_features=784)
      (1): ReLU()
    )
  )
)
```

So, I have 784 to 100, 100 to 100 coming down. It looks a bit different because the number of pointer reassignment just went up. So, the hierarchy of the tree is also split up. That is on the code side of it, but as a function it does not make any change which comes down over there.

Now, that I have this. So, let us run the trainer for this one and here it was just set down for 2 epoch. So, we will just run it for 2 epochs and then look into what comes out.

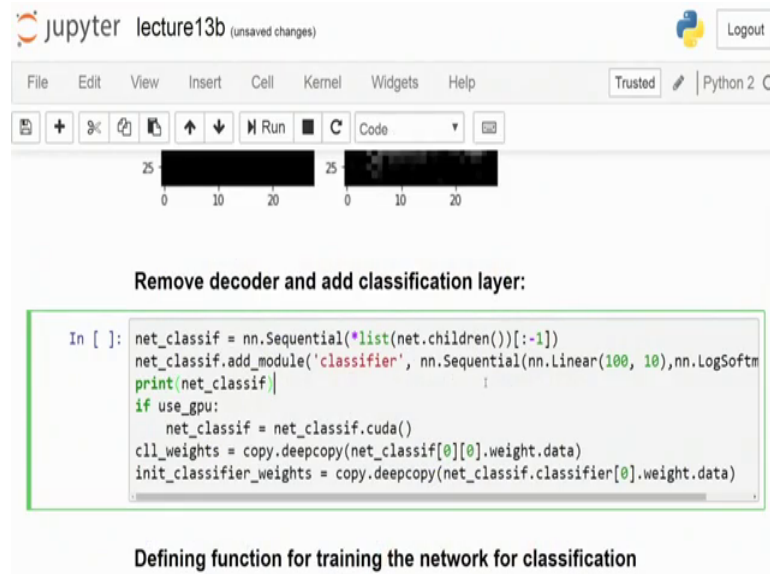
(Refer Slide Time: 23:31)



So, this is how the actually. Pretty much since the error is going down you can keep on training it for even longer duration as well. Here let us look into the reconstruction over there. As I increase a layer, I mean to a lot of people you might say that the blur has increased over there. Yes, but then the network has not yet learned completely.

So, you learned on the network over a longer period of time your errors will keep on decreasing your reconstruction efficacy does definitely increase significantly. Now the next part is when I need to remove all of these decoders. So, I have my 2 hidden layers for representational learning perfectly trained I need to remove the decoder unit over there from.

(Refer Slide Time: 24:09)



The screenshot shows a Jupyter Notebook window titled "lecture13b (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 Python code:

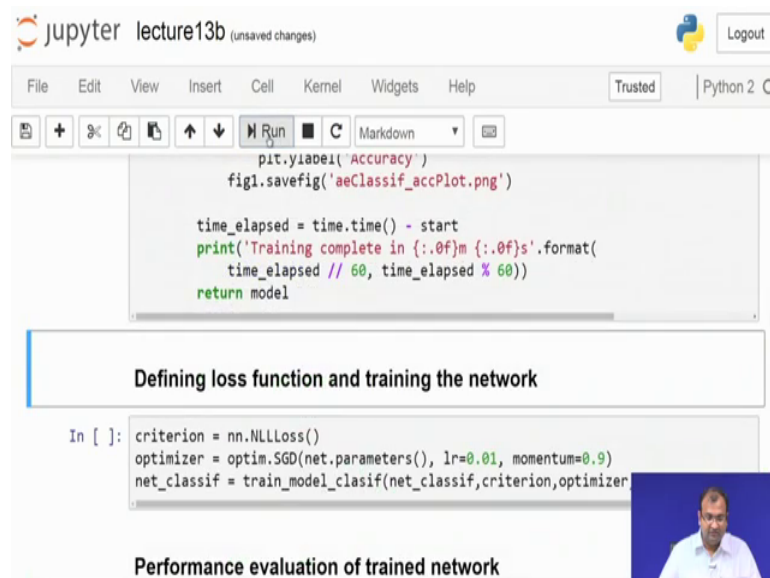
```
In [ ]: net_classif = nn.Sequential(*list(net.children())[:-1])
net_classif.add_module('classifier', nn.Sequential(nn.Linear(100, 10), nn.LogSoftmax))
print(net_classif)
if use_gpu:
    net_classif = net_classif.cuda()
c11_weights = copy.deepcopy(net_classif[0][0].weight.data)
init_classifier_weights = copy.deepcopy(net_classif.classifier[0].weight.data)
```

Below the code editor, the text "Defining function for training the network for classification" is visible.

My auto encoder, what I do is just in the same way come down to the first level over there first level of the tree and just remove the last terminal element and then this is what I am left down with.

So, I remove this part. So, I have 784 to 100 then again the next level 100 to 100. And then my output over here which is my classifier gets 100 to 10 features and then a LogSoftmax which is happened to it and this is pretty much similar to the network which we had in the earlier case. Now doing that we have a function for training it which is pretty similar to what we had earlier.

(Refer Slide Time: 25:36)



The screenshot shows a Jupyter Notebook window titled "lecture13b (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 in the editor is as follows:

```
plt.ylabel('Accuracy')
fig1.savefig('aeClassif_accPlot.png')

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

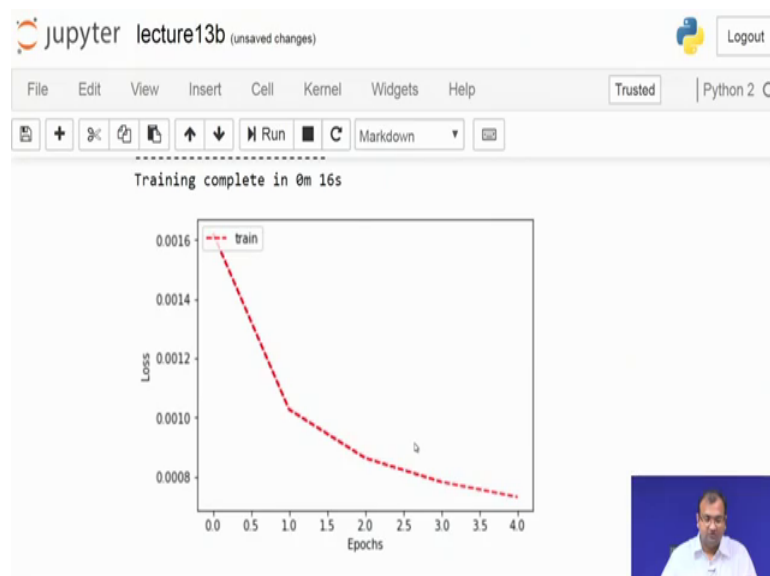
Below the code editor, there is a section titled "Defining loss function and training the network" with the following code:

```
In [ ]: criterion = nn.NLLLoss()
optimizer = optim.SGD(net.parameters(), lr=0.01, momentum=0.9)
net_classif = train_model_clasif(net_classif,criterion,optimizer
```

At the bottom of the notebook, there is a section titled "Performance evaluation of trained network" and a small video feed of the presenter.

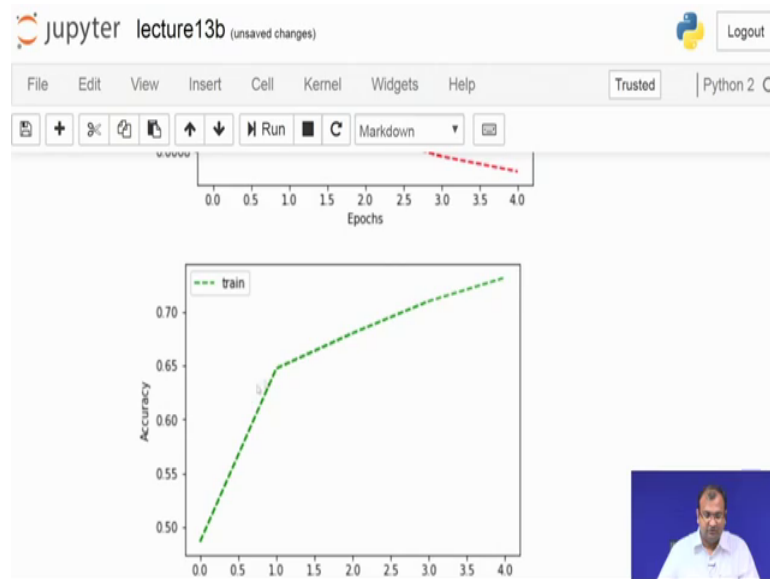
So, I am not discussing much in detail about it and then we get into the loss function and do it. So, well change this epochs and make it to 5 and let us see what comes out over there. It would take you approximately 3 seconds as we had in the earlier cases as well. Now please do not be much worried about these warnings I mean this is just with some syntax errors which keep on creeping in with the change in generation of these libraries which come up.

(Refer Slide Time: 25:15)



So, it took about roughly 16 seconds in order to train down for 5 epochs and.

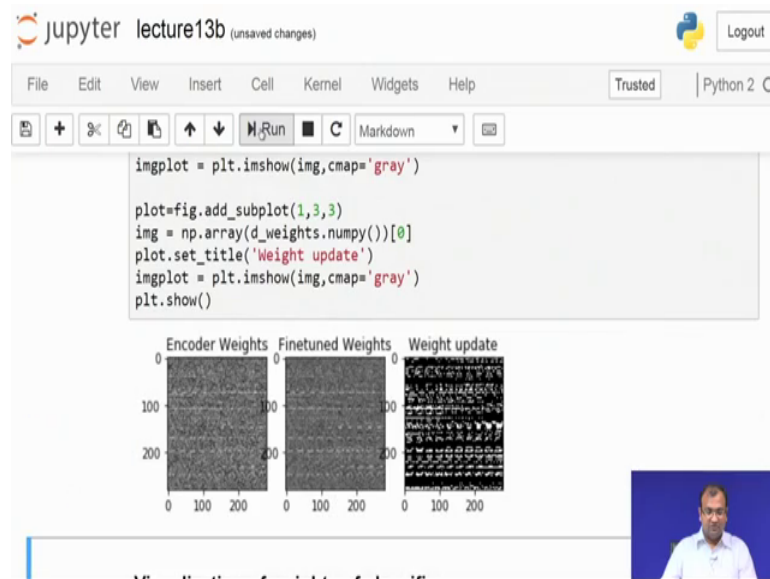
(Refer Slide Time: 25:17)



Is the accuracy which we get which is again quite closer to 80 percent as we see now we would like to even evaluate this network in terms of it is total accuracy and that comes down to about 0.7295 percent. Now you might have an impression that in the earlier case when I was doing an end to end pre training it was giving me an 82 percent whereas, here that I have done a layer by layer pre training I am not getting that high accuracy, but we need to keep one thing in mind that in none of the cases the whole network was trained completely and then if we are able to train it over the whole duration of the epochs then it is pretty much possible that it would train down in a real good way and give you a much better accuracy and that is actually the point over there.

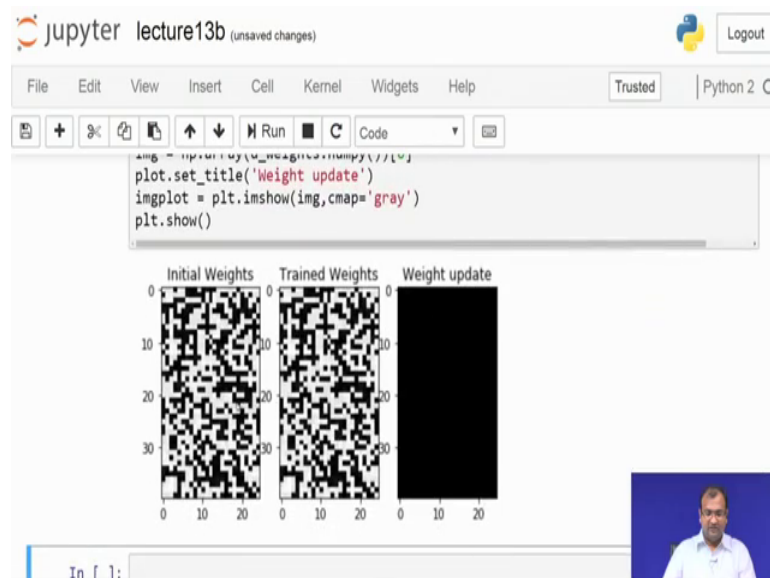
So, with layer wise pre training it does take a bit longer to train, but the convergence criteria's and the total performance is much better than what you can expect on with the end to end pre training though theoretically both of them are supposed to converge on to the same point, but then we are speaking in terms of our trainable function and it is stochastic behaviors which are not necessarily always guaranteed to converge on to the same point.

(Refer Slide Time: 26:24)



So, if we look into the weights over there, between my encoder weights and after this fine tuning there are again different kinds of changes which happen and if you look. So, since we are not yet learning down very fixed kind of patterns over there. Training it over a longer period will definitely give you a much way of doing.

(Refer Slide Time: 26:42)



So, here interestingly what came out that from your initial weights and if you look into your trained weights over there has not been any change. That is really something funny maybe the auto encoder actually learnt out representation. So, well that given any kind of

a random combination or possibly the random combination of weights which came down over there was something which was actually true to the global minimum point and that is why this happened, but then do not expect that. This is the same one which keeps on happening every time over and over again for you all this is fortunately a good case which we just found out for the for our exercise over here today.

So, with that we come to an end with the autoencoders and 2 different kinds of auto encoders. In the next class we will be doing out on colored images using autoencoders and subsequently. Instead of classifying, the next class is on classifying patches of color images. The subsequent next lecture will be where you doing a pixel to pixel annotation or something called as a semantic segmentation using auto encoder.

So, there will be like given an image you go down to any given pixel take a small neighborhood around the pixel using that information over there you are going to classify each pixel location and do a raster scan over the whole image and you are pretty much at a point when you can say print out the complete image. So, with that stay tuned and enjoy down more of coding and then the lectures as we keep on going now.

Thanks.