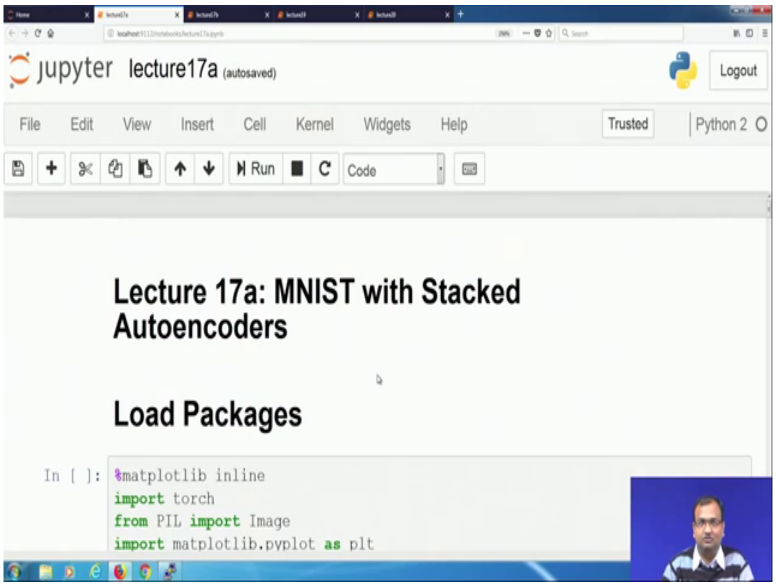
Deep Learning for Visual Computing

Prof. Debdoot Sheet
Department of Electrical Engineering
Indian Institute of Technology, Kharagpur

Lecture – 17
MNIST and Fashion MNIST with Stacked Autoencoders

Welcome. So, today we would be doing a short demo around with a stacked Autoencoders, and for this purpose we are choosing down the standard data set which is on handwritten digits, and that is MNIST data set. And here what I am going to show you around is that.

(Refer Slide Time: 00:30)



Lecture 17a: MNIST with Stacked Autoencoders

Load Packages

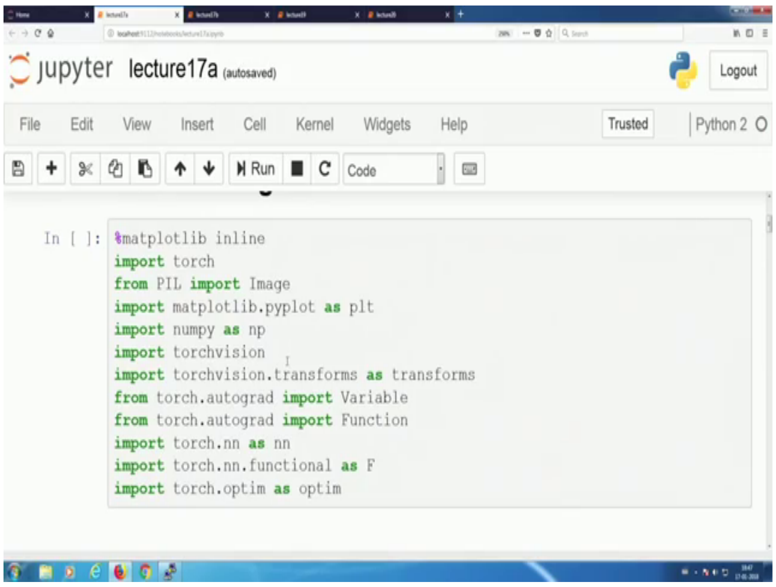
```
In [ ]: %matplotlib inline
        import torch
        from PIL import Image
        import matplotlib.pyplot as plt
```

While we have already discussed theory on how stacking within Autoencoders is used in order to hierarchically keep on building. So, there were two different particular kinds of stacking which you had done, and I used to call them as one method, and the other method or one was end to end pretrained stacking, and other was a ladder wise or greedy growing method over there.

So, and one of them you had all the layers stacked in the encoder side. And similarly you had a cascaded architecture on the decoder which was almost like opposite in philosophy to what was present in that encoder. So, if an encoder you were reducing the size over there in decoder, you are just going to increase the sizes, and then one way is where you train this whole network into it, and then you discard the decoder and have just this front part of it where you are learnt the representations for your classification. The other one was where you could actually grow one layer at a time, and there you have an pair of encoder decoder, encoder decoder being form and you keep on successfully chopping of the decoder side and increasing an another encoder layer, and have a corresponding decoder over there.

So, we are actually going to replicate both of these methods over here within stacking of Auto-encoders. So, one of these approaches which we do so.

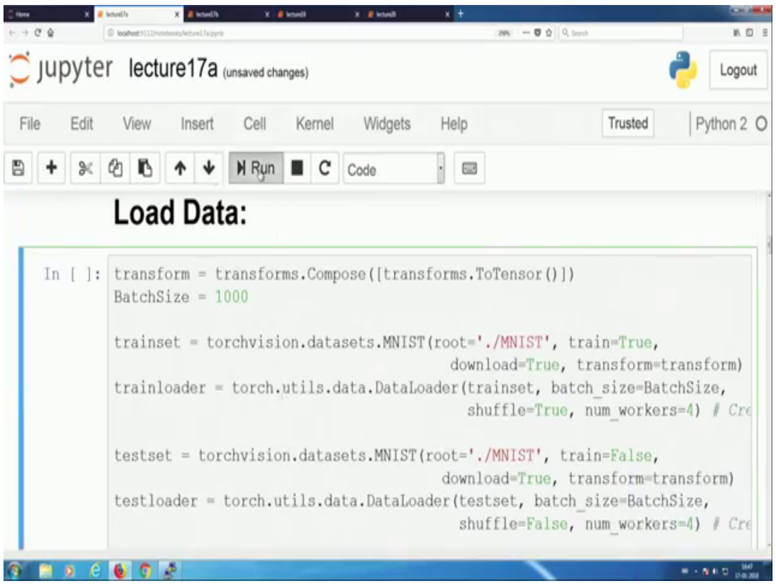
(Refer Slide Time: 01:42)


jupyter lecture17a (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

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

We have the same codes as you had done in the earlier ones, and as with how it goes down is that in your initial header, its present over this. So, on the run let us just keep on executing that.

(Refer Slide Time: 01:53)


jupyter lecture17a (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

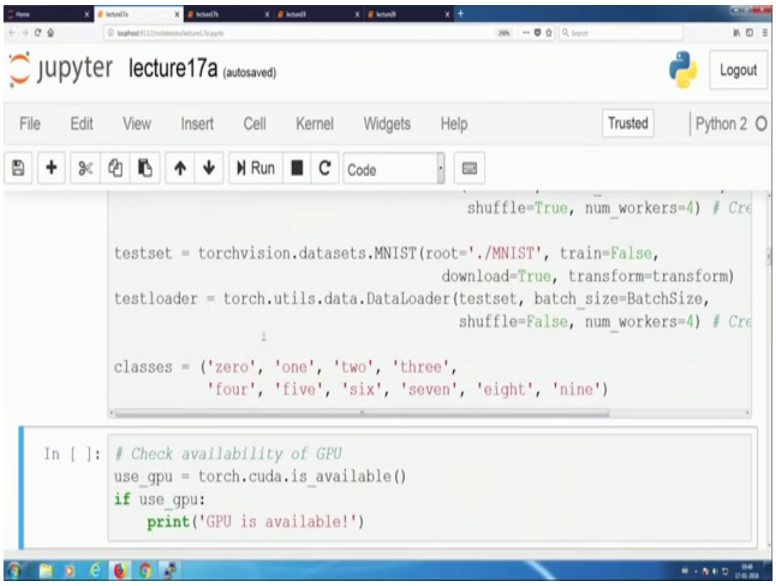
Load Data:

```
In [ ]: transform = transforms.Compose([transforms.ToTensor()])
        BatchSize = 1000

        trainset = torchvision.datasets.MNIST(root='./MNIST', train=True,
                                              download=True, transform=transform)
        trainloader = torch.utils.data.DataLoader(trainset, batch_size=BatchSize,
                                                  shuffle=True, num_workers=4) # Cre

        testset = torchvision.datasets.MNIST(root='./MNIST', train=False,
                                             download=True, transform=transform)
        testloader = torch.utils.data.DataLoader(testset, batch_size=BatchSize,
                                                 shuffle=False, num_workers=4) # Cre
```

Now, comes down to your data set part. And if you remember then this part of the code looks quite similar to what you already had, and that was just with trying to call down your MNIST data set from within your torch vision data sets application and that creates.

So, one was to get your training data set, and then to create down your training loader and which will be just looking into how many number of parallel loaders, you can work it out and as such its equal to number 4, and then you also set down your batch size and which is equal to the number of images it fetches, found in one particular batch. So, that is how it goes down.

(Refer Slide Time: 02:32)

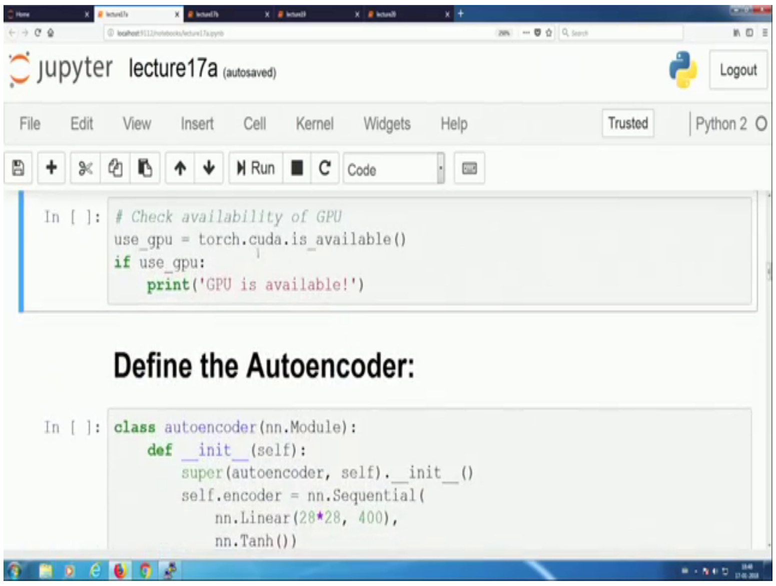

jupyter lecture17a (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

```
                                    shuffle=True, num_workers=4) # Cre

testset = torchvision.datasets.MNIST(root='./MNIST', train=False,
                                     download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=BatchSize,
                                         shuffle=False, num_workers=4) # Cre

classes = ('zero', 'one', 'two', 'three',
           'four', 'five', 'six', 'seven', 'eight', 'nine')
```

```
In [ ]: # Check availability of GPU
        use_gpu = torch.cuda.is_available()
        if use_gpu:
            print('GPU is available!')
```

And you have your classes and everything defined.

(Refer Slide Time: 02:34)


jupyter lecture17a (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

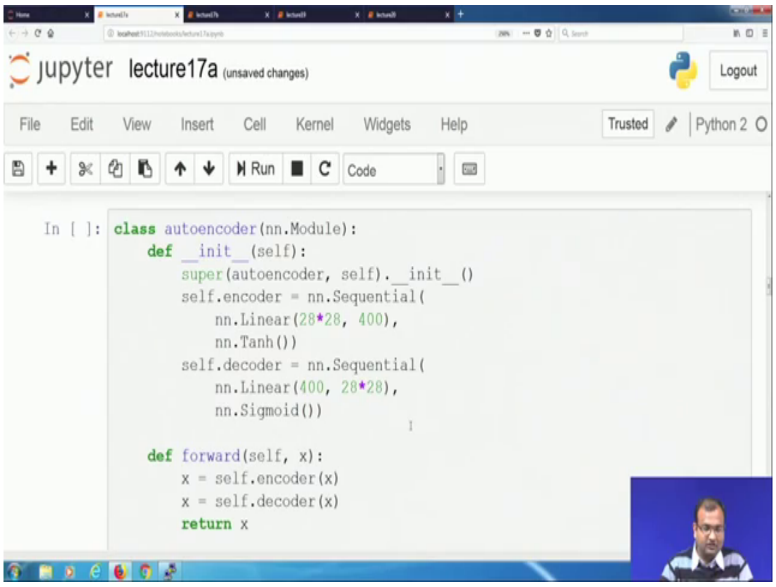
```
In [ ]: # Check availability of GPU
        use_gpu = torch.cuda.is_available()
        if use_gpu:
            print('GPU is available!')
```

Define the Autoencoder:

```
In [ ]: class autoencoder(nn.Module):
            def __init__(self):
                super(autoencoder, self).__init__()
                self.encoder = nn.Sequential(
                    nn.Linear(28*28, 400),
                    nn.Tanh())
```

In the next part is where you have your gpu accessibility and compatibility calls and in. In fact, you by now that you have done a significant number of these exercises. You have been able to understand that this part of the code makes your whole rest of the codes quite generic and easy to work it out on. So, in case your gpu is not present then it just throws you a flag equal to false. So, your used gpu flag turns down as false, and in case your; you used gpu flag is not set down then none of your models or data gets converted on to a gpu array and you can keep on getting it done.

(Refer Slide Time: 03:10)

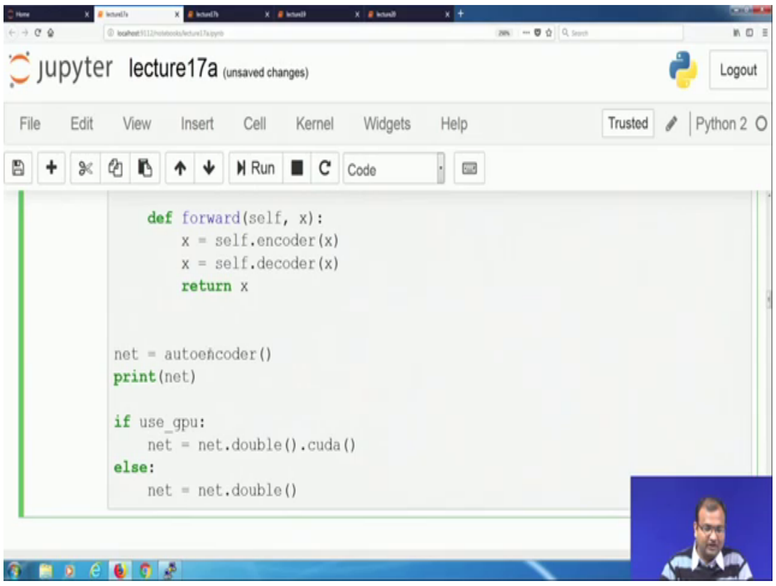

jupyter lecture17a (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

```
In [ ]: class autoencoder(nn.Module):
            def __init__(self):
                super(autoencoder, self).__init__()
                self.encoder = nn.Sequential(
                    nn.Linear(28*28, 400),
                    nn.Tanh())
                self.decoder = nn.Sequential(
                    nn.Linear(400, 28*28),
                    nn.Sigmoid())

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

So, the next module is to get down defining an autoencoder. So, in case of an autoencoder what we have over here is, a very simple approach. So, what we are doing is, initially we start by defining linear network which is just connect down 28 cross 28 number of pixels a 784 pixels connected to 400 neurons over there.

So, 784 to 400 neurons and on the other side of the decoder is 400 to 784, it is just one layer at a time over there. Now once your layer definition and initialization is complete, then the next part is that you write down your forward function and your forward function, what it does is, that it will do a forward pass over the encoder which consists of a linear layer and a tan hyperbolic as a transfer function, and the decoder which has a linear layer and a sigmoid as a transfer function over there. So, once this comes out.

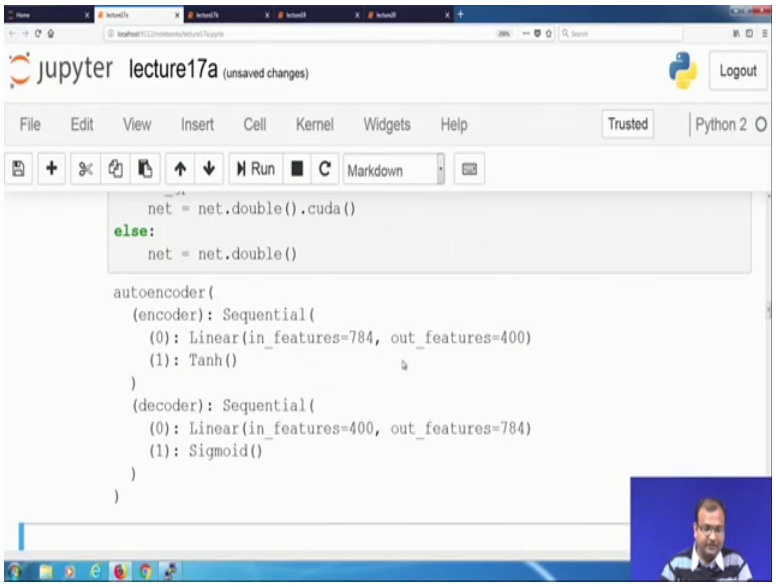
(Refer Slide Time: 04:02)


jupyter lecture17a (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

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


net = autoencoder()
print(net)

if use_gpu:
    net = net.double().cuda()
else:
    net = net.double()
```

Then you have your network which is defined as yours. So, your autoencoder module which is created over here; that is what initializes done down the network and then you can print your network. And eventually if you have good accesses then you can convert it down to your cuda arrays as well.

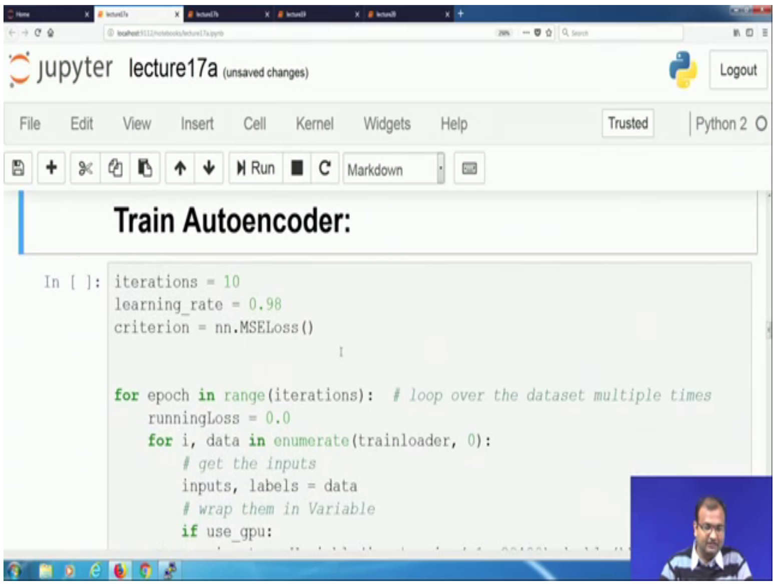
(Refer Slide Time: 04:21)


jupyter lecture17a (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Markdown

```
    net = net.double().cuda()
else:
    net = net.double()

autoencoder(
  (encoder): Sequential(
    (0): Linear(in_features=784, out_features=400)
    (1): Tanh()
  )
  (decoder): Sequential(
    (0): Linear(in_features=400, out_features=784)
    (1): Sigmoid()
  )
)
```

So, we get down this encoder decoder structure over here which just connects. Now 784 neurons onto 400 neurons, a very simple autoencoder architecture as such.

(Refer Slide Time: 04:30)



jupyter lecture17a (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Markdown

Train Autoencoder:

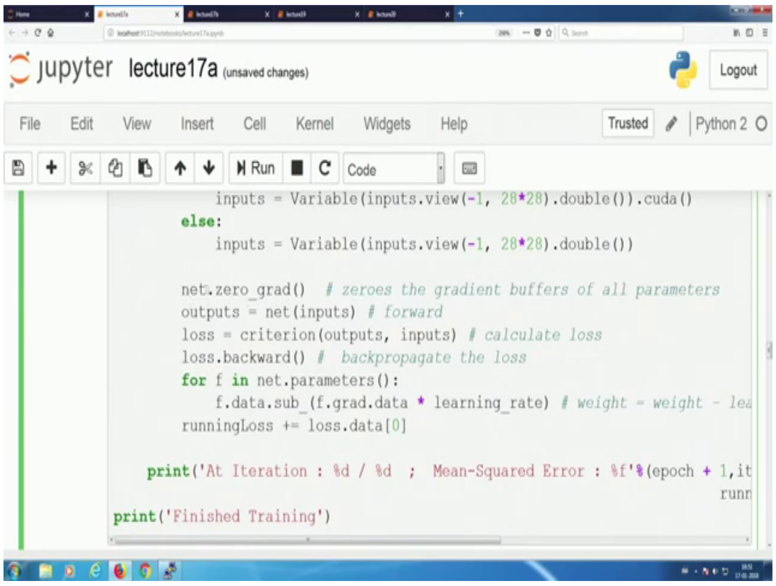
```
In [ ]: iterations = 10
        learning_rate = 0.98
        criterion = nn.MSELoss()

        for epoch in range(iterations):  # loop over the dataset multiple times
            runningLoss = 0.0
            for i, data in enumerate(trainloader, 0):
                # get the inputs
                inputs, labels = data
                # wrap them in Variable
                if use_gpu:
```

Now, what we will do is, we will start with training this autoencoder. So, for the purpose of training it is kept down as small. So, we just have 10 iterations over which it would iterate. So, 10 iterations or 10 epochs is where it would be going through all the 60,000 training data points available on it, and the learning rate has been kept down as 0.98. And since we are in a autoencoder structure which means that on the decoder side, we are just trying to recreate whatever was present on the encoder side itself, and this typically is something which is equivalent to a regression problem which we are trying to solve.

And now we are trying to minimize the error which comes between whatever is decoded and formed and the original; whether it is able to replicate the original image in its perfect way and the best way of doing. It is to take down an l two norm as a loss function or mean square error in your losses. So, that is where we define the criterion as in with the earlier cases. So, here its MSE loss, because you are not doing a classification, but you are just doing a representation learning. So, now, within my iterator over here. So, while I am running down, my whole network over here, over the range of iterations and that is my epoch.

So, I start by doing the first part of it, where you are just going to convert all of these data which is available to gpu, in case you have a gpu available, and you decide to run it on a gpu.

(Refer Slide Time: 05:49)

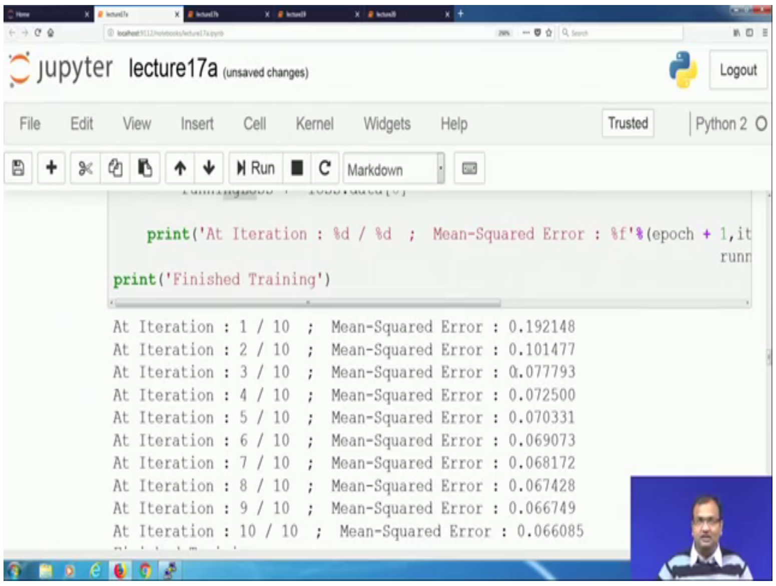


```
            inputs = Variable(inputs.view(-1, 28*28).double()).cuda()
        else:
            inputs = Variable(inputs.view(-1, 28*28).double())

        net.zero_grad()  # zeroes the gradient buffers of all parameters
        outputs = net(inputs) # forward
        loss = criterion(outputs, inputs) # calculate loss
        loss.backward() #  backpropagate the loss
        for f in net.parameters():
            f.data.sub_(f.grad.data * learning_rate) # weight = weight - lea
        runningLoss += loss.data[0]

    print('At Iteration : %d / %d  ;  Mean-Squared Error : %f'%(epoch + 1,it
                                                                         runn
print('Finished Training')
```

The next part as it goes down, is that you know to zero down all the gradient residuals within the network. So, that it does not create any confusion with left over residuals within a particular. So, yeah; so, basically your gradients which were computed in the earlier epoch should not play any role in the current epoch, and that is the whole purpose of this zeroing down on the gradients. Then the next part is, just a forward pass you give your inputs over here which is one image. On the output you again get the same kind of an image, then you have a batch, then you give a batch of images.

You get a batch of images on the output side of it. Next is compute out your mean square error using the criterion function which is just the mean square error between the output and the input image batches, which are formed. And from there you have your loss, and then you do a derivative of the loss using your loss dot backward compute. Once that is done, the next part is basically to update it using the standard gradient descent learning rule, and you have a count over whatever loss is accumulated, over your epochs present over there, and that is how the whole network gets trained. So, let us run this one. Do it for sometime till it finishes off.

(Refer Slide Time: 07:02)


jupyter lecture17a (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Markdown

```
    print('At Iteration : %d / %d  ;  Mean-Squared Error : %f'%(epoch + 1,it
                                                                     runn
print('Finished Training')
```

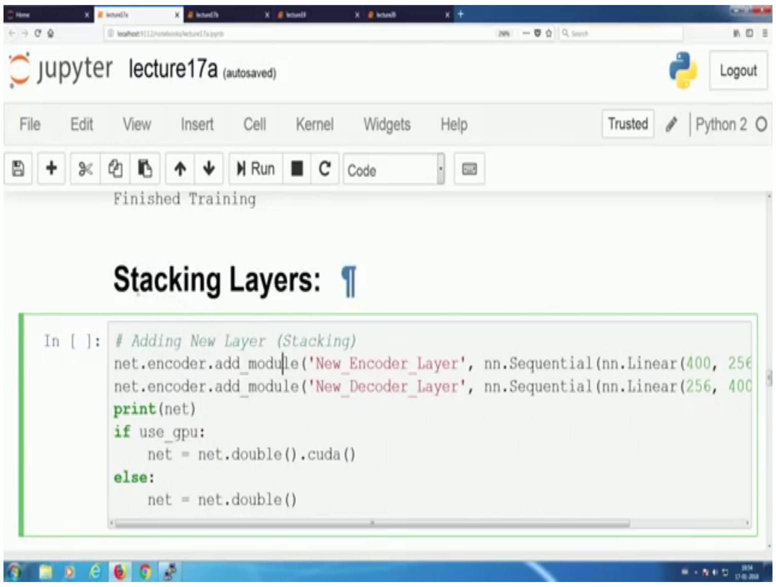
```
At Iteration : 1 / 10  ;  Mean-Squared Error : 0.192148
At Iteration : 2 / 10  ;  Mean-Squared Error : 0.101477
At Iteration : 3 / 10  ;  Mean-Squared Error : 0.077793
At Iteration : 4 / 10  ;  Mean-Squared Error : 0.072500
At Iteration : 5 / 10  ;  Mean-Squared Error : 0.070331
At Iteration : 6 / 10  ;  Mean-Squared Error : 0.069073
At Iteration : 7 / 10  ;  Mean-Squared Error : 0.068172
At Iteration : 8 / 10  ;  Mean-Squared Error : 0.067428
At Iteration : 9 / 10  ;  Mean-Squared Error : 0.066749
At Iteration : 10 / 10  ;  Mean-Squared Error : 0.066085
```

So, you see that one of the iterations are over and it has a mean square error which comes down to around 0.19. So, now, two iterations over it comes down to 0.10. You see, clearly see that there is a decrease in error and then it keeps on steadily decreasing. Now one thing to keep in mind is that, since it is a fully connected layer with lot of neurons. So, it would take a bit of time, and that is where it takes down a bit of time for us as well. And after some time you see that now that it is nearing the actual plateau region, the valley region where it is supposed to come down to an optimal.

So, the relative change, the relative difference between these errors which comes down between two epochs, is also going down quite slower. Now you can actually play around over there. So, what you can do is, you can change down your learning rates and play, and see if this comes down pretty fast or not. So, this is an optimal combination which we are given, but then this is not always necessary that this would be the only combination which is possible. You can set your learning rate as 1 to 10, 100 and then we have already done some exercise it or going to be changing down the learning rates in the earlier part of the classes. So, here you can just make a change around with your learning rates as well, and then you would be seeing that this changes. Now here this was the first approach of training down where the whole thing was that. You have one layer at a time which is being trained.

So, your first hidden layer is now trained and that is at the end of this one. Now my objective is that, I want to introduce a another hidden layer over there, and that is where my stacking of layer comes down.

(Refer Slide Time: 08:36)


jupyter lecture17a (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

```
Finished Training
```

Stacking Layers:

```
In [ ]: # Adding New Layer (Stacking)
net.encoder.add_module('New_Encoder_Layer', nn.Sequential(nn.Linear(400, 256
net.encoder.add_module('New_Decoder_Layer', nn.Sequential(nn.Linear(256, 400
print(net)
if use_gpu:
    net = net.double().cuda()
else:
    net = net.double()
```

So, in case of stacking of layer what I do is quite simple. So, what I would do is, I have my network which is strain which was net. Now I take my encoder part over there and then start a. I add another new module to it. I call this as say a new encoder layer and that is basically sequential connection or a fully connection between the 400 neurons to 256 neurons.

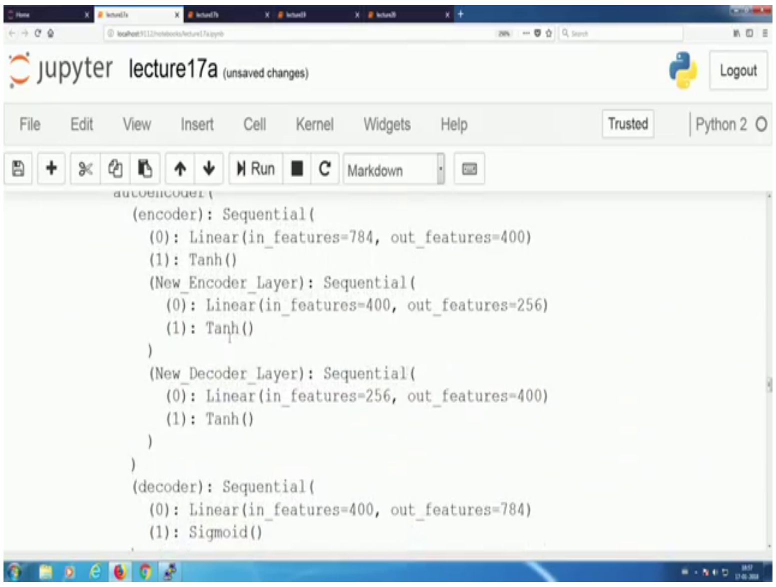
So what now I have within my auto encoder is, in the earlier case you just write the 784 neurons going down to 400 neurons. And then again reconstructed to 784 neurons. Here what I do is, 784 neurons going down to 400 neurons going down to 256 neurons. From there; coming up to 400 neurons, from there going to 784 neurons. So, this becomes my first part of it. Now similarly in the decoder part where I have 256 neurons to be connected down to 400 neurons, this comes down.

Now in my encoder what I do is that I connect down a tan hyperbolic transfer function over there, after the output of this 400 neurons comes out. And then on the other side of my sequential which is to be connected to the decoder, just one layer before to the decoder I also have another tan hyperbolic function. Now if you clearly recall, then in the earlier case when we were trying to define our network over here, you would see that there is a tan hyperbolic function which is connected to the output of these 400 neurons. So; that means, that any of the values which comes down in these 400 neurons over there are in the range of minus 1 to plus 1.

now whatever is the input to this particular layer which connects down 400 to 784 that is also in the same range of minus one to plus one as comes down over here now we need to preserve that whole aspect here as well when we are tracking down neurons and for that reason what we have is that 400 neurons. So, whenever I have 256 connecting down to 400 neurons. So, I have a tan hyperbolic transfer function and not any other transfer function. So, you need to keep this whole concept of dynamic ranges in your mind. So, in case you are not using any transfer function, it does not hinder you in any way, because you just have a linear pass over there.

But in case you are using a transfer function, then you need to keep in mind that my transfer function should be such that the dynamic range of the responses are in the same order, and they do not get confused around with each other. Now once that is done. So, this, these parts are just added over there and then I can print my network. So, let us just see what comes down.

(Refer Slide Time: 11:02)

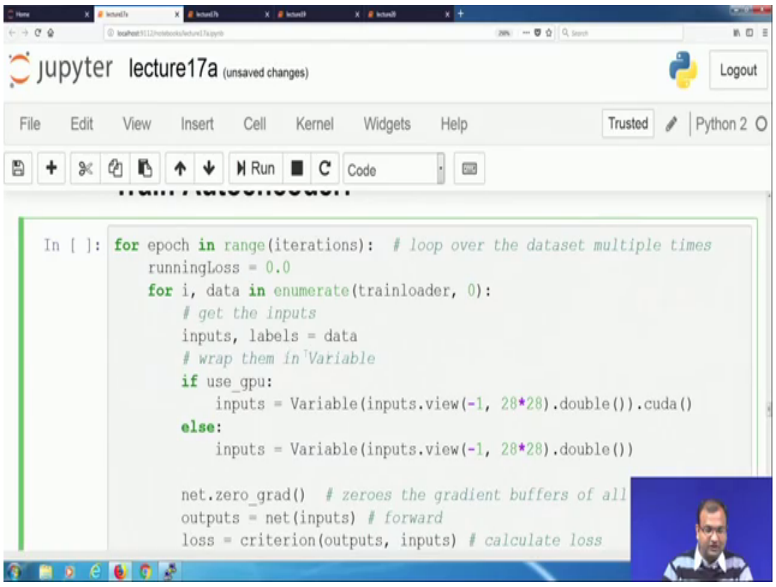

jupyter lecture17a (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Markdown

```
  autoencoder(
    (encoder): Sequential(
      (0): Linear(in_features=784, out_features=400)
      (1): Tanh()
      (New_Encoder_Layer): Sequential(
        (0): Linear(in_features=400, out_features=256)
        (1): Tanh()
      )
      (New_Decoder_Layer): Sequential(
        (0): Linear(in_features=256, out_features=400)
        (1): Tanh()
      )
    )
    (decoder): Sequential(
      (0): Linear(in_features=400, out_features=784)
      (1): Sigmoid()
```

So, you had your earlier case which was the linear connection from 794 neurons to 400 neurons then a tan hyperbolic. The output of this one goes into another sequential connection from 400 to 256 and then tan hyperbolic transfer function.

Now input to the next decoder layer is 256 to 400 and tan hyperbolic function, and subsequent to that is my earlier version of the decoder, which had 400 neurons connect down to 784 neurons and a sigmoid transfer function. So, my philosophy over here, by adding these two new encoder and decoder layer, does not make much of a difference in terms of how the data is propagated. Though if you look into this structure over here. This structure does look quite nested out in a way, because as per the nesting what happens is, this decoder has been nested out with this autoencoder, this encoder layer over there and this is on a separate list level. So, that is up to you how you define your hierarchical strategies, but till it is a linear network. It does not make much of a difference coming down.

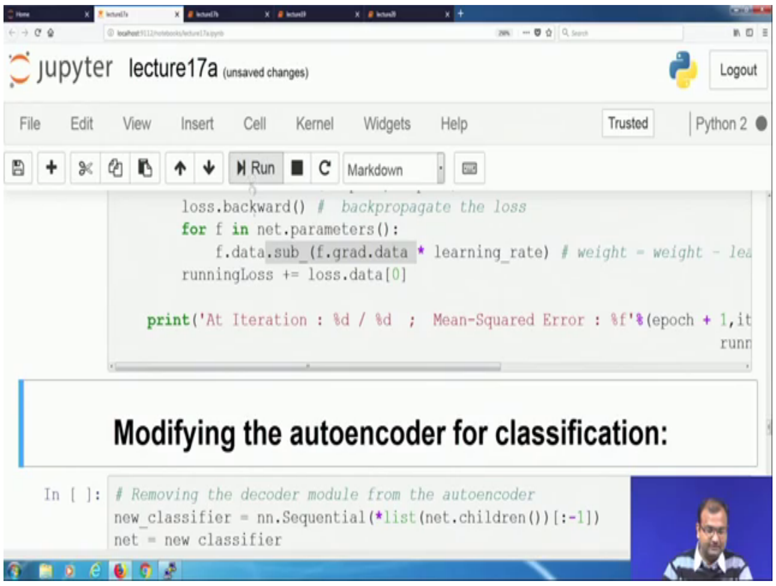
(Refer Slide Time: 12:00)


jupyter lecture17a (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

```
In [ ]: for epoch in range(iterations):  # loop over the dataset multiple times
            runningLoss = 0.0
            for i, data in enumerate(trainloader, 0):
                # get the inputs
                inputs, labels = data
                # wrap them in Variable
                if use_gpu:
                    inputs = Variable(inputs.view(-1, 28*28).double()).cuda()
                else:
                    inputs = Variable(inputs.view(-1, 28*28).double())

                net.zero_grad()  # zeroes the gradient buffers of all
                outputs = net(inputs)  # forward
                loss = criterion(outputs, inputs)  # calculate loss
```

Now, with this one; now we start training down the whole this modified new network once again. So, from the first part of it, it is pretty straightforward and then you do a forward pass and get down your outputs. And then you have your loss function defined and derivative taken down about the loss function, and then you have your learning rates update rule which is defined over here.

(Refer Slide Time: 12:23)


jupyter lecture17a (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Markdown

```
        loss.backward() #  backpropagate the loss
        for f in net.parameters():
            f.data.sub_(f.grad.data * learning_rate) # weight = weight - lea
        runningLoss += loss.data[0]

    print('At Iteration : %d / %d  ;  Mean-Squared Error : %f'%(epoch + 1,it
                                                                        runn
```

Modifying the autoencoder for classification:

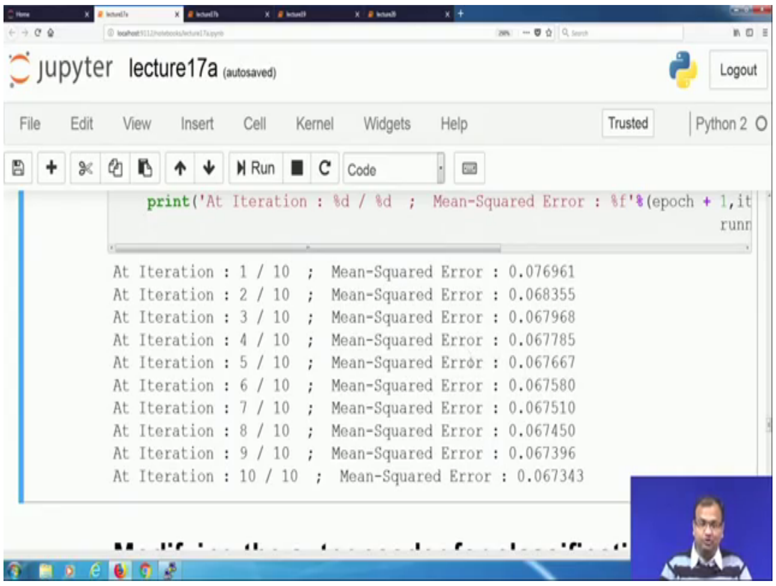
```
In [ ]: # Removing the decoder module from the autoencoder
        new_classifier = nn.Sequential(*list(net.children())[:-1])
        net = new classifier
```

So, let us get this one running. So, what you would see eventually over here, is that this might take slightly tad bit longer than what it had taken in the earlier case, though you have sort of better train model over there.

So, if you look down at the mean square error which comes down around 0.76961. And if you look at the closing error over here; that is about 0.66. Now one question which you might come down and you can definitely ask is, why did this error suddenly jump up? You need to keep in mind that the encoder decoder the newer layers from 400 to 256 which we have now connected; that is randomly initialized. And that does mean that whatever is coming down on the learnt representations and getting forwarded to the decoder layer subsequently, is now pretty much random, that has to learn down exact representations.

And that is why the starting error is a bit more than what was the ending error over that, and then this keeps on going. So, till now we are at a position where we are not yet close to just the error which was received by just using one hidden layer. Now it may be a case that you might not be able to reach down the exact error rates which you had achieved by using just one hidden layer, and there is nothing to panic around it. The only point is that you need to keep some things in mind which is merely stacking down. Layers is not a guarantee that you will have a higher accuracy and lower error rate ok.

(Refer Slide Time: 13:41)


jupyter lecture17a (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

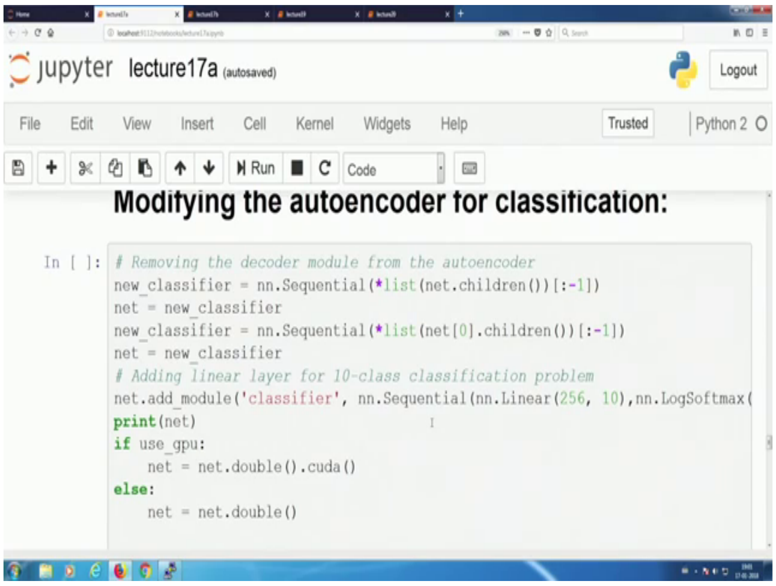
```
print('At Iteration : %d / %d  ;  Mean-Squared Error : %f'%(epoch + 1,it
                                                              runn

At Iteration : 1 / 10  ;  Mean-Squared Error : 0.076961
At Iteration : 2 / 10  ;  Mean-Squared Error : 0.068355
At Iteration : 3 / 10  ;  Mean-Squared Error : 0.067968
At Iteration : 4 / 10  ;  Mean-Squared Error : 0.067785
At Iteration : 5 / 10  ;  Mean-Squared Error : 0.067667
At Iteration : 6 / 10  ;  Mean-Squared Error : 0.067580
At Iteration : 7 / 10  ;  Mean-Squared Error : 0.067510
At Iteration : 8 / 10  ;  Mean-Squared Error : 0.067450
At Iteration : 9 / 10  ;  Mean-Squared Error : 0.067396
At Iteration : 10 / 10  ;  Mean-Squared Error : 0.067343
```

We are trying to solve down the actual problem of classification over here using auto encoder for representation learning. And then we are using a stacking policy and the mean square error estimate, in order to find out my error estimates for this stacked autoencoder. Now we are measuring down the networks for ability to perform against one particular task. Whereas, the network is supposed to be used for another task and that is classification.

So, given all of these things in mind what you need to have, like really outlined over there is that. It is not necessary that just stacking down will bring down your errors; however, clever stacking and change of error rates can actually bring it down. So, at this point, I would leave it to you to actually go up and as update this learning rate, because this learning rate has been kept the same, which is of 0.98 as we had for the first case, where I was just using one single hidden layer.

Now play around with this learning rate and you can definitely see a change coming over here. In fact, with the ability to even modify and go down.

(Refer Slide Time: 14:50)


jupyter lecture17a (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

Modifying the autoencoder for classification:

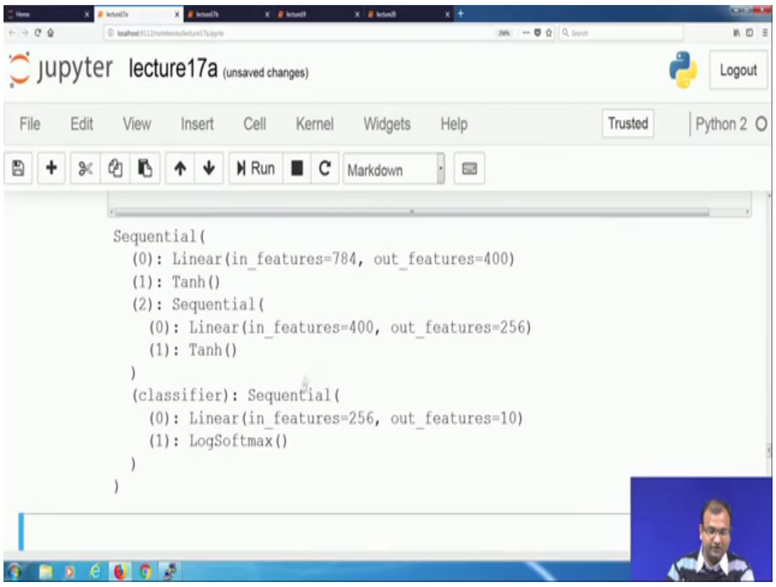
```
In [ ]: # Removing the decoder module from the autoencoder
        new_classifier = nn.Sequential(*list(net.children())[:-1])
        net = new_classifier
        new_classifier = nn.Sequential(*list(net[0].children())[:-1])
        net = new_classifier
        # Adding linear layer for 10-class classification problem
        net.add_module('classifier', nn.Sequential(nn.Linear(256, 10),nn.LogSoftmax(
        print(net)
        if use_gpu:
            net = net.double().cuda()
        else:
            net = net.double()
```

Now, once this part is ready. So, what we have now is an autoencoder, which has two hidden layers. Now my whole objective was to create a neural network, where I have two hidden layers, and then a final output layer for classification. And since, I am using the MNIST case. So, all of us are quite aware that this is just a 10 digits classification problem 0 to 9 written in 100 and 100, and numbers which are in small snapshots of 28 cross 28 pixels, and you have to classify it out. Now over there I would need to modify it down some part of the network, because I do not need the decoder as such anywhere now. So, now, what I can do is, I can start defining something called as a classifier. The whole object of this classifier is actually to get down my sequential network, which comes down from the first part; like just reduce the end part of it. Now if you look into your structure of the network which was given down in the earlier case. So, let us get up over there. So, what you had is this encoded layer over here, which had this newer encoder and this decoder.

Now for me I would just need to preserve out this layer and I would need to preserve out this layer. These two layers are something which I can do away with. And the best way of solving it out is, first delete this layer then delete this layer, and then you just have these two connections present over there. So, that is the typical thing which has been incorporated over here. Now once that is done, the next part is that I would need to add down. So, now what I have is, 784 neurons which goes to 400 neurons, that goes to 256 neurons from 256 neurons. I will have to connect it down to 10 output neurons over.

So, for that I start adding down this extra sequential model, which adds down 256 to 10 neurons. And now my transfer function over here is a log soft max. So, this is taken down from the perspective that I would like to have a classification function, and not necessarily a regression function anymore, and for that purpose. So, let us just run this part of the code and you would see your new layer coming it out.

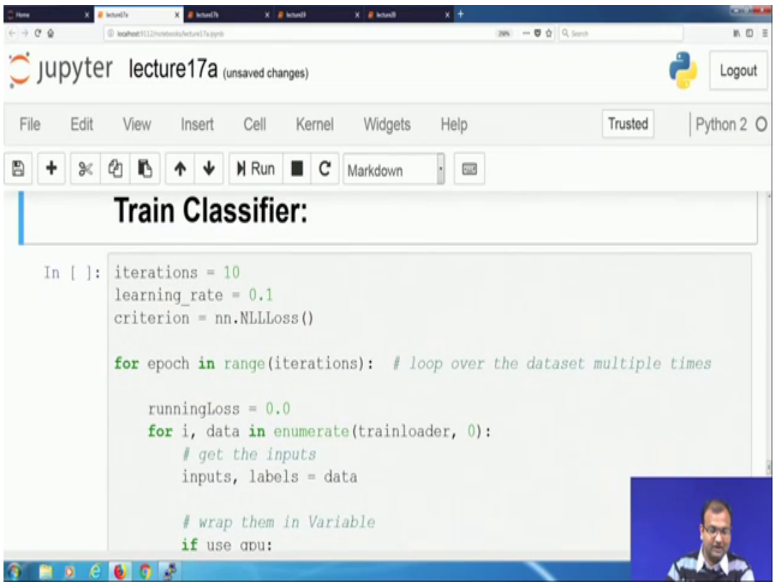
(Refer Slide Time: 16:51).


jupyter lecture17a (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Markdown

```
Sequential(
  (0): Linear(in_features=784, out_features=400)
  (1): Tanh()
  (2): Sequential(
    (0): Linear(in_features=400, out_features=256)
    (1): Tanh()
  )
  (classifier): Sequential(
    (0): Linear(in_features=256, out_features=10)
    (1): LogSoftmax()
  )
)
```

So, you had 784 to 400, 400 to 256 which was preserved as in the previous case. And then you added down this classifier part which was 256 to 10 neurons.

(Refer Slide Time: 17:02)

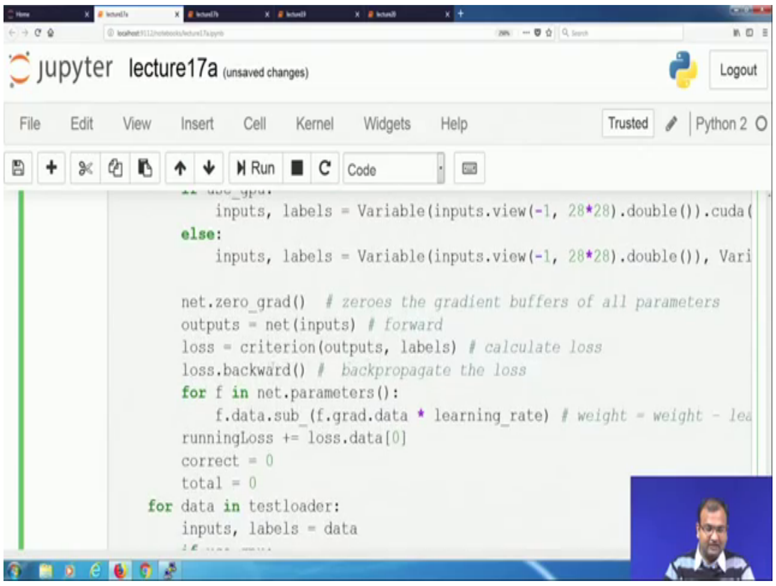

jupyter lecture17a (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Markdown

Train Classifier:

```
In [ ]: iterations = 10
        learning_rate = 0.1
        criterion = nn.NLLLoss()

        for epoch in range(iterations):  # loop over the dataset multiple times

            runningLoss = 0.0
            for i, data in enumerate(trainloader, 0):
                # get the inputs
                inputs, labels = data

                # wrap them in Variable
                if use_gpu:
```

Now I would be putting down my whole classifier to train. Now here I have again changed my learning rate and that has, that is no more as 0.998 which was close to 1, but it is not 0.1, which is say 1 factor, 1 decimal factor lesser than what we had chosen. In the earlier case my criteria also changes, it is no more mean square error loss, but now it becomes class negative likelihood loss function. Now this is just a classification loss function which we have used over there. And in fact, like when we do down in the subsequent lectures on loss functions for and different kind of error measures for classification versus regression, you will be finding out a difference of what comes down.

(Refer Slide Time: 17:46)

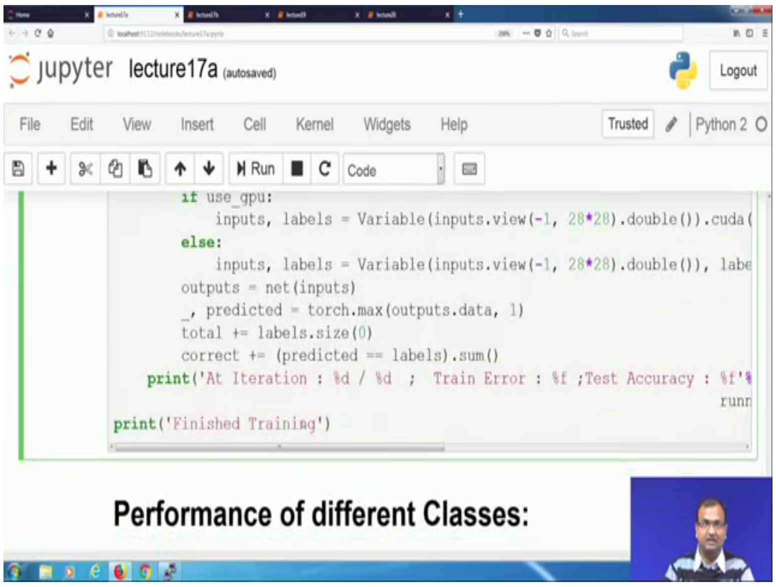


```
            inputs, labels = Variable(inputs.view(-1, 28*28).double()).cuda(
        else:
            inputs, labels = Variable(inputs.view(-1, 28*28).double()), Vari

        net.zero_grad()  # zeroes the gradient buffers of all parameters
        outputs = net(inputs) # forward
        loss = criterion(outputs, labels) # calculate loss
        loss.backward() #  backpropagate the loss
        for f in net.parameters():
            f.data.sub_(f.grad.data * learning_rate) # weight = weight - lea
        runningLoss += loss.data[0]
        correct = 0
        total = 0
    for data in testloader:
        inputs, labels = data
```

So, here the whole aspect is that; now, when I try to look into my training and output. So, I give my images which are my inputs, and then I get my output. Now my outputs in the earlier case, when I was doing representation learning with rotary encoders were just compared along with inputs. Now it is no more. Now whatever is my predicted output is pitched against my ground truth label of that particular image, which has been passed on to it for training purposes, and that helps me in calculating this negative log likelihood error loss function. And then we do gradient of the loss and subsequently, and update using the vanilla approach of gradient descent, the simple gradient descent approach.

Now once this part is done, then what you do is that you keep on repeating this over the total number of images, which are present down over there and then you can keep on training the network.

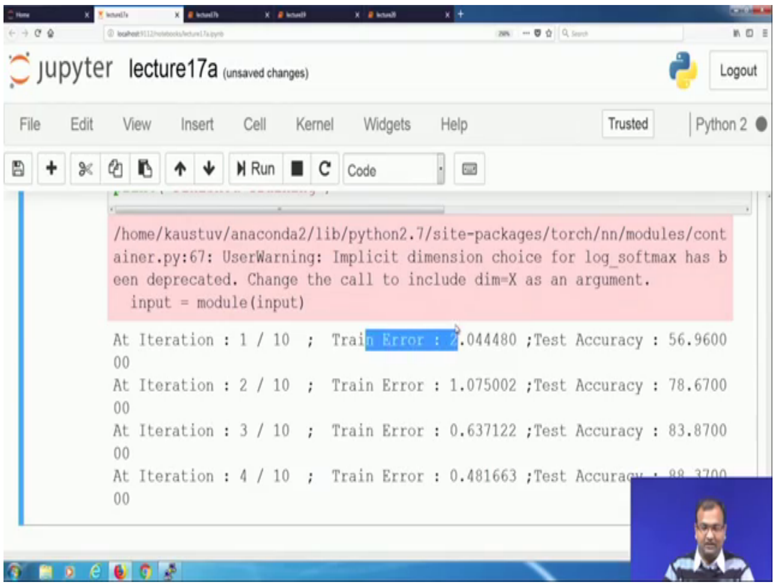
(Refer Slide Time: 18:34)


jupyter lecture17a (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

```
            if use_gpu:
                inputs, labels = Variable(inputs.view(-1, 28*28).double()).cuda(
            else:
                inputs, labels = Variable(inputs.view(-1, 28*28).double()), labe
            outputs = net(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum()
        print('At Iteration : %d / %d  ;  Train Error : %f ;Test Accuracy : %f'%
                                                                            runn
print('Finished Training')
```

Performance of different Classes:

(Refer Slide Time: 18:39)


jupyter lecture17a (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

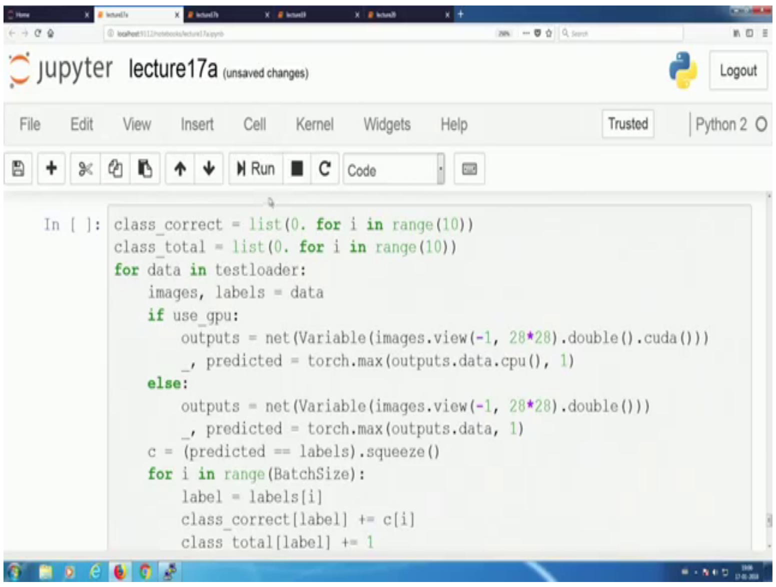
```
/home/kaustuv/anaconda2/lib/python2.7/site-packages/torch/nn/modules/cont
ainer.py:67: UserWarning: Implicit dimension choice for log_softmax has b
een deprecated. Change the call to include dim=X as an argument.
  input = module(input)

At Iteration : 1 / 10  ;  Train Error : 2.044480 ;Test Accuracy : 56.9600
00
At Iteration : 2 / 10  ;  Train Error : 1.075002 ;Test Accuracy : 78.6700
00
At Iteration : 3 / 10  ;  Train Error : 0.637122 ;Test Accuracy : 83.8700
00
At Iteration : 4 / 10  ;  Train Error : 0.481663 ;Test Accuracy : 88.3700
00
```

So, let us just keep on using this one. So, it’s a, these I just answered an implicit calls around with future things which would just become defunct. So, you do not need to worry at this point of time. Now if we look down over here.

So, I have my error as well as my accuracy, both of them being printed, because at each point like while this part was calculating out the losses over here. In the next part it was also finding out the total number of predictions, and whether the predictions are correct or not as in over here. So, if the prediction value is correct, then its accurate, and that is what is being used in order to pitch over here. So, if you see that after having made this classifier, which used pre trained autoencoder network in order to do it. So, we reach down somewhere at 91.83 percent. Actually see in terms of classification and that is associates itself with the negative log, likelihood negative log, likelihood loss of 0.3.

So this is about just training down standard the network, in case of using a stacked autoencoder principle.

(Refer Slide Time: 19:50)

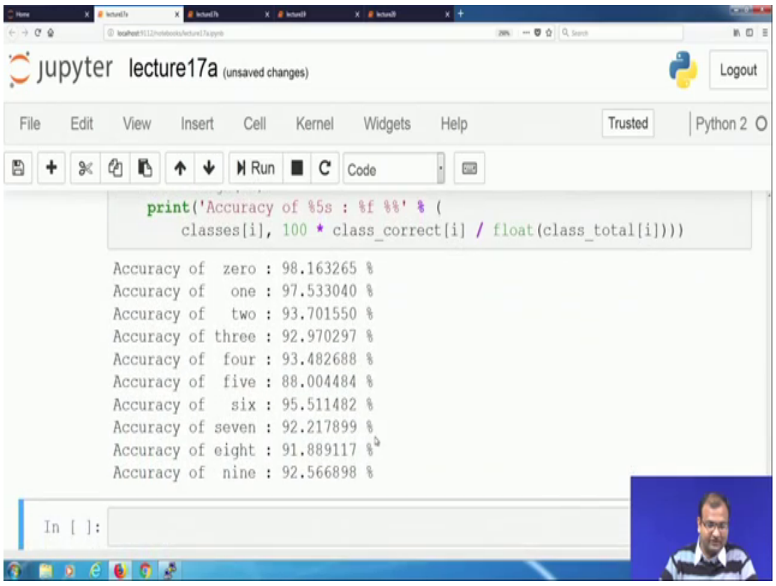


```
In [ ]: class_correct = list(0. for i in range(10))
        class_total = list(0. for i in range(10))
        for data in testloader:
            images, labels = data
            if use_gpu:
                outputs = net(Variable(images.view(-1, 28*28).double().cuda()))
                _, predicted = torch.max(outputs.data.cpu(), 1)
            else:
                outputs = net(Variable(images.view(-1, 28*28).double()))
                _, predicted = torch.max(outputs.data, 1)
            c = (predicted == labels).squeeze()
            for i in range(BatchSize):
                label = labels[i]
                class_correct[label] += c[i]
                class_total[label] += 1
```

Now, what I would also like to look at is what, how was it performing in terms of different classes which is like each of the classes. Was it more prone to make an error while classifying, and for which of the classes it was, where it was not making an error while classifying the whole stuff. So, just let us look and when just wait while the whole thing. Oh sorry; I had actually accidentally said this one running. Once again we just fit for some time. And now one interesting point is that, since I did not update the network and I just used it for running. So, it started actually from the old place where I left it down.

And that is why the first beginning accuracy of the earlier model starts around 892. So, this was an accident thing, but fortunately you will be able to get down the model which has a better accuracy. So, one simple way is that when you are doing, do not restrict it necessarily to just any box, but keep on playing around with more number of epochs.

(Refer Slide Time: 20:56)


jupyter lecture17a (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

```
    print('Accuracy of %5s : %f %%' % (
        classes[i], 100 * class_correct[i] / float(class_total[i])))

Accuracy of  zero : 98.163265 %
Accuracy of   one : 97.533040 %
Accuracy of   two : 93.701550 %
Accuracy of three : 92.970297 %
Accuracy of  four : 93.482688 %
Accuracy of  five : 88.004484 %
Accuracy of   six : 95.511482 %
Accuracy of seven : 92.217899 %
Accuracy of eight : 91.889117 %
Accuracy of  nine : 92.566898 %

In [ ]:
```

Now once that is done. So, let me run this one once again and here comes down the accuracy for each of these classes.

So, if you look down for class 0, its 98 percent accurate for class 1, its 97 percent accurate, and then you see that for class number 5 its just 88 percent accurate. So, this was 1, where the accuracy was going below the average accuracy. So, the average accuracy over here, is about 93.7 percent, and so anything below 93.7 is actually which is below at 92.97 is somewhat close to the average accuracy. Some of them are very high like 98 percent accuracy; some of them are really low which is like 88 percent accuracy. So, these are typical ways in which you can actually look for class accuracies. In fact, you can actually create another different kind of a matrix.

So, we had learned on in our earlier cases about cost functions, and your sensitivity specificity matrices which are quite typical in case of classification. So, here what you can do is, you can actually find out which class was getting more confused with the class. So, you can have a 10 by 0 matrix and create a confusion matrix out of it, all the true classes which are perfectly classified are, what will be located along the diagonal and off diagonals are each other erroneous classifications which come out. So, this was about trying to run it down with the simple example of using a stacked autoencoder. The next example which I have is, basically to show it down using the other data set which is on fashioned MNIST.

So, these were those small images of fashion objects or clothing lines, and which are also available in the same form as a grayscale image, and in the size of 28 cross 28 pixels. So, if we just finish it off, as in the first case you have your first part run down, then you have your data loader and whole thing. And since from the earlier experiment which we had also done with fashion MNIST; you do know that the number of samples, and the distribution of the train, and test class is quite similar to as in MNIST. It is also a 10 class classification problem. So, here my whole network is also defined in the same way, and the whole logic of trying to do it around with fashion MNIST was to show you that similar kind of a network, on a different dataset can actually also work, and that is why we just have a small subsection numbering being created out for the whole experiment, so that you can build around with that.

So, any standard data set or anything which you would like to load and look around, you can actually work it; however, like keeping, keep one thing in mind that you see these errors which come down these error ranges do not necessarily need to be in the same way, as you had for your case of just a simple MNIST space classifier. So, over here the error ranges will be pretty much different, because your inputs are now changed. The range in which your inputs vary they are also quite different. And in fact, for your fashion MNIST you would see that it is not necessarily only white or the gray level 255, and only black for the gray level 0. Here you have good smooth histogram across all the different grade levels present in an 8 bit image, which comes out pretty perfect.

So, this is about just going down with it. So, let us just hope it gets out first. So, two more epochs, and then we are done with the first layer, and then you would eventually be going down to your stacking of layers as we had done in the earlier case as well yeah. So, it comes down to 10 epochs and with one single hidden layer trains up to 0.063. So, in the same way, we do a stacking of the subsequent layers over there. So, this was my stacking thing which I had done in the earlier case, and then I get trained down my stacked out version of the auto encoder getting a starting point from whatever was trained in the earlier case.

So, this would also be running down over 10 iterations as we had defined in the earlier case, and you would see that typically you are lost over here is 0.89 and that is higher than this loss over there, going by the same argument, because you are knew earlier which you introduced over here, that has more number of the; so, whatever were the weights in that particular layer, they are randomly initialized at the start of it, and for that reason you have starting with a higher error than the stopping error in the earlier case. So, this we just need to wait for 10 epochs till it goes down, but you can clearly see that over here. Now by the fourth epoch it has a error value, which is now going below, what we had observed in the earlier case with just one hidden layer.

So, in the earlier case, you had just reached at the end of 10 epochs error point of 0.063 whereas, over here you see that by the fourth epoch you are already much below that particular error point over there, and this is one of those classical examples, where you can see that while in the earlier case we are just using MNIST by stacking down two different layers, you were not necessarily going down to a lower error in terms of understanding representations. Whereas, over here you see that you can actually go down to a lower error rate. So, that is, there is technically no relationship between stacking of layers and whether it would work on your own data set or not.

But there is definitely a lot of relationship between the nature of the data on which you are working, and what is the kind of images you are working; however, let us get another thing quite clear at this point of time that we do not have any empirical rule. So, what nature of a data, and what kind of an architecture would work out good, we seriously do not know, and then that is something which people on the field actually explore out on practical purposes. So, from there we just start with the classification, and then in the same way of modifying the network as we had done in the earlier case. We modify it out, and then you have ten neurons on the output side of it, and then you would be starting down your training program, and since this is a classification problem. So, we are using negative log likelihood loss function over here. So, we start with the training function and then you see that you have an accuracy initially which is of 61 percent.

Then it keeps on going to 69 percent and the error also keeps on decreasing. So, let us just see how much it can climb down. So, in the earlier case with just using MNIST you had seen that during training, it had gone high up to 92 percent of an aggregate accuracy, whereas, here it does not appear that it would be coming up to that level. So, that brings us to another point that training over a large number of epochs is always a, definitely a good idea if you would like to see it, because we have not yet technically saturated out. So, you know that there is a chance of going much higher and looking at the amount of change in which it is happening. So, it is changing in significant places of decimals.

So, this can keep on going higher and higher, if I have it for more number of epochs as well as changing down the learning rate and making it a bigger number, might actually accelerate your whole learning perspective as well. So, this is where I come to an end on using stacked Autoencoders. And in the next lectures we will be continuing with denoising as well as parts Autoencoders. So, till then.

Thanks and check out for the next lectures.