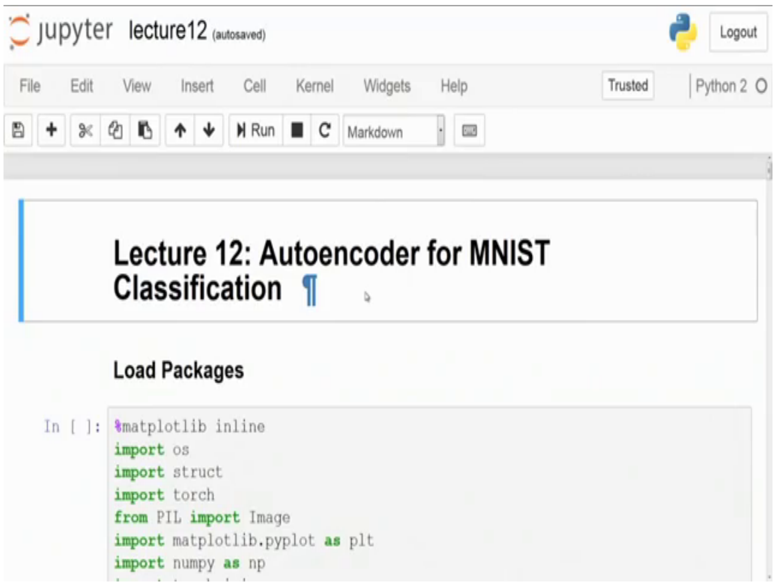
Deep Learning for Visual Computing
Prof. Debdoot Sheet
Department of Electrical Engineering
Indian Institute of Technology, Kharagpur

Lecture – 12
MNIST handwritten digits classification using autoencoders

So, welcome. Today we will be doing hands on session and this is with the use of an Autoencoder.

(Refer Slide Time: 00:21)


jupyter lecture12 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Markdown

Lecture 12: Autoencoder for MNIST Classification ¶

Load Packages

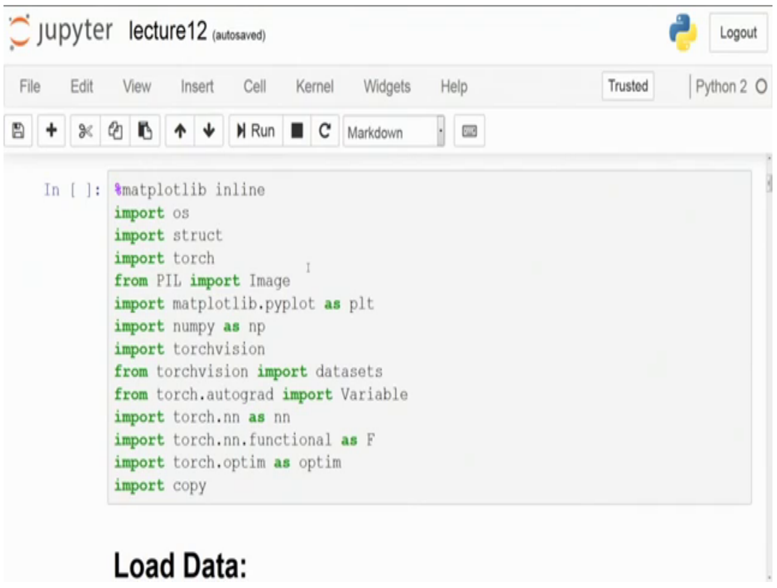
```
In [ ]: %matplotlib inline
        import os
        import struct
        import torch
        from PIL import Image
        import matplotlib.pyplot as plt
        import numpy as np
```

So, in the last few lectures you have understood and studied about auto encoders and previous to that, there was multi layer perceptron and the whole idea was that these auto encoders can be used for 2 purposes. One of them is to do a very efficient representation learning and the other one is to actually understand about how to use these representations as initializations for your multi-layer perceptron.

So today's example which we will be doing is, a pie torch based tutorial and this is on something which is well known and this is called as the MNIST classification challenge. So as we had done on the lectures, I did say about that one of the earliest areas where this whole aspect of deep learning was coming out and working out pretty good, was actually in the aspect of using these kind of classifiers for classifying handwritten digits into 0 to 9 and that is how they were working out fine.

So today's example which we will be doing is, use the standard data set called as MNIST, and use this small patch kind of images for your handwritten digits which are of size 28 cross 28 and then, we will be using them for classification purpose.

(Refer Slide Time: 01:32)


jupyter lecture12 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Markdown

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

Load Data:

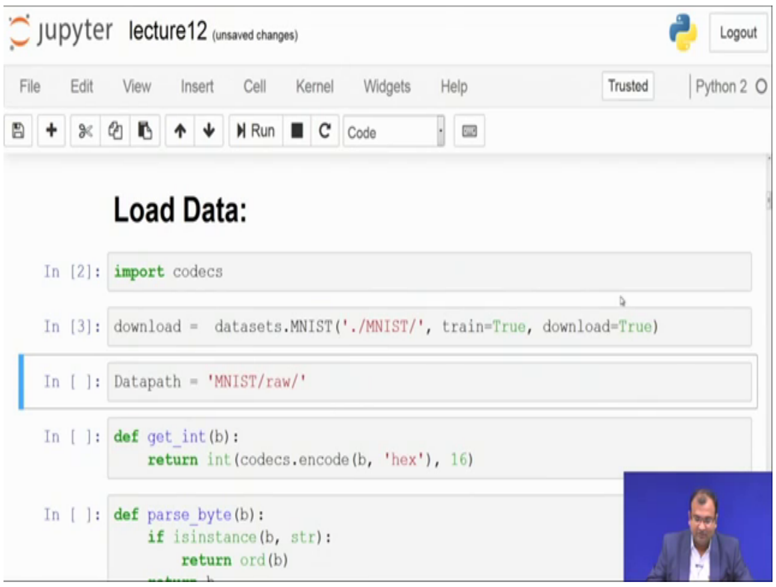
So, let us get started with it. So, as with any of our codes we have the initial part which is a header structure of importing down all the libraries which would be subsequently needing for our work. So, here it does not need much of an introduction over there as such. So, except for one interesting package, which comes down over here which is called as opting package.

So, we are a bit ahead of time in terms of introducing optim on your programming paradigms. But, keep in mind one thing that we will be making use of a standard gradient descent itself, although it is from a stochastic gradient descent with batch size of 1. So, this is just to make, get you introduced to the advanced options available within the library and packages.

Now, as one important aspect, which I need to really point out is that, down the line we will be discussing much more details about optimizers and, all aspects of this package called as opting. But, for the sake of time and to keep it conformal to our subsequent lectures, we are already introducing it over here. And then, this also makes your coding quite compact without having to do, because if you remember from your earlier one on multi layer perceptron, where you had your gradient descent coming in. So, the gradient descent, the major issue was that, you had to put on a lot of lines of code in order to find out your error, then do a derivative of the cost function, do a derivative of the network and then update all the parameters over there. Whereas here, these are all just into one single library and as we keep on going through it, I will be introducing you to further concepts over there.

So, let us run this first part of the header block now. Once that works out, the next part is to load your data.

(Refer Slide Time: 03:14)


jupyter lecture12 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

Load Data:

```
In [2]: import codecs

In [3]: download = datasets.MNIST('./MNIST/', train=True, download=True)

In [ ]: Datapath = 'MNIST/raw/'

In [ ]: def get_int(b):
            return int(codecs.encode(b, 'hex'), 16)

In [ ]: def parse_byte(b):
            if isinstance(b, str):
                return ord(b)
```

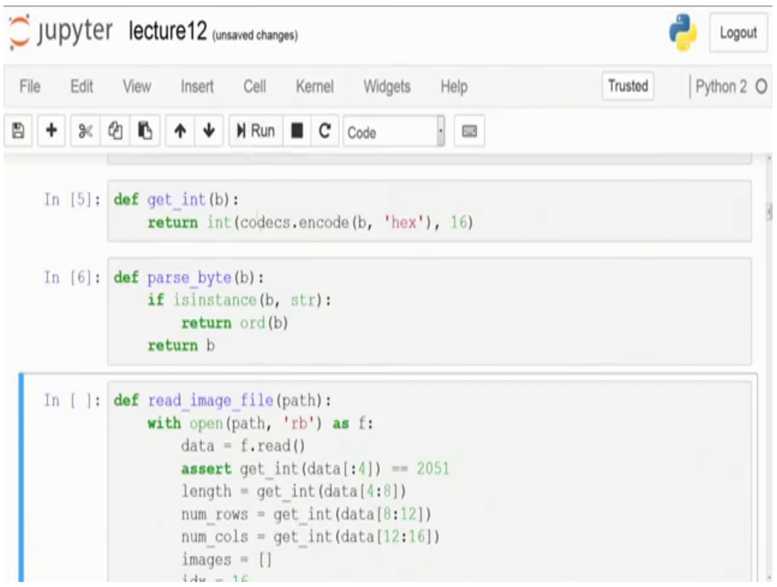
So, this data is again available within your torch vision data sets itself. So, you do not need to download something from an external web resource file and repack it in any ways. So, the first part is that it goes on. So, the codecs option over here is just to do a decoding over here. So, it will come down eventually. So, let us go to this download part over there. So, since it is already downloaded for me and available to me, So I did not see any other command coming down, but if you are doing it for the first time, you would definitely be seeing some downloader command, is it downloading from so and so location, and what you get down is, something called as u byte or unsigned byte type of data file.

Now, it is not much of a concern to be worried about the data file. Because your, codec function over here can actually read from those available direct kind of files. So, the first part over here is, we have a few of these tensors, which are a few of these functions which are more of related to how to handle down the data and get the data into your torch tensors or as and when required even into your numpy arrays subsequently. So, here the first part of it is just a small function written down to fetch down one integer at a time or one image basically at any point of time.

So, what it does is basically there is a codec to read down and it reads in hexadecimal format and gives you back an integer from every single hex code which is written down onto the file. So, typically how it is stored is that, you have 0 to 255 or 8 bytes available over there, 8 bits available over there. So, these 8 bits instead of, when you are storing it down you can have some sort of a binary presentation and everything. So, the u byte basically uses a hexadecimal number representation for storing these 8 bits over there.

Now, as it goes down with the, this hexadecimal representation form over there, you need to decode this hexadecimal representation and get an integer equivalent representation in 8 bits, and that is the purpose of this getting, which is to convert down from a hexadecimal representation to a integer space representation.

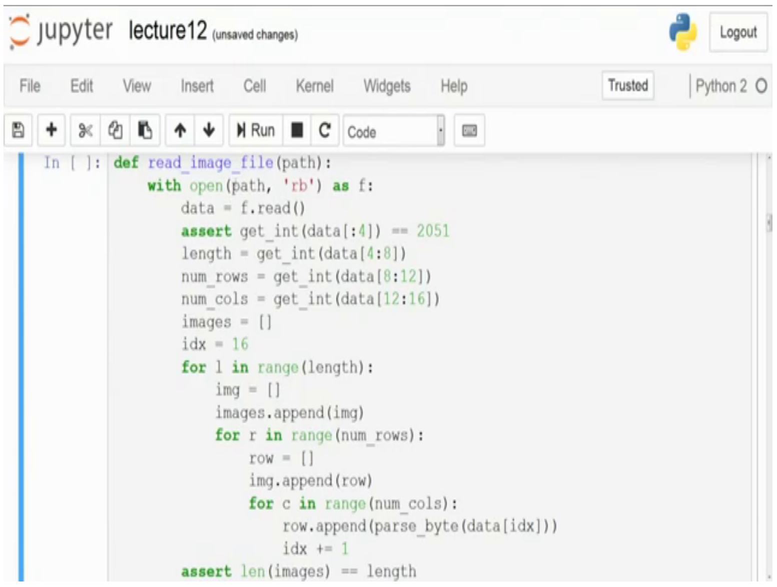
(Refer Slide Time: 05:35)


jupyter lecture12 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

```
In [5]: def get_int(b):
            return int(codecs.encode(b, 'hex'), 16)

In [6]: def parse_byte(b):
            if isinstance(b, str):
                return ord(b)
            return b

In [ ]: def read_image_file(path):
            with open(path, 'rb') as f:
                data = f.read()
                assert get_int(data[:4]) == 2051
                length = get_int(data[4:8])
                num_rows = get_int(data[8:12])
                num_cols = get_int(data[12:16])
                images = []
```

Next is, here this parse byte function whole option is that you get down the whole file in terms of a string whereas, this is a matrix, and you will have to a resize the whole string which comes down into a 2d matrix over there.

(Refer Slide Time: 05:54)

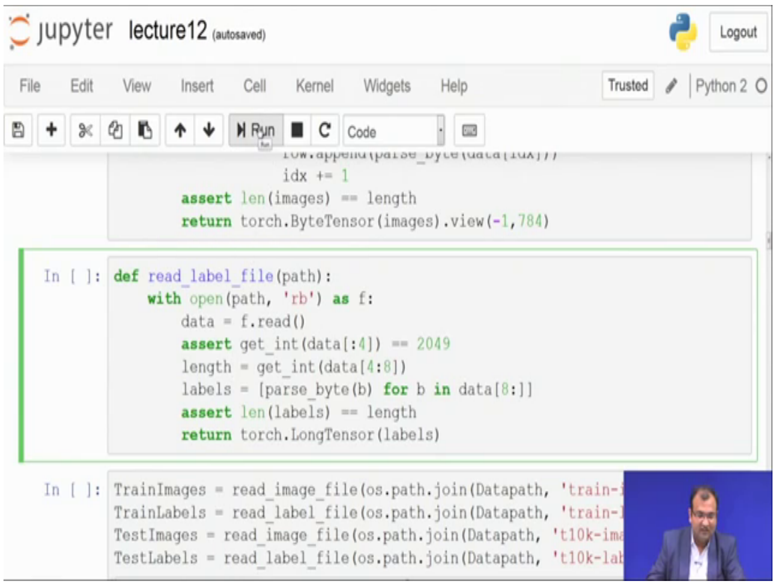

jupyter lecture12 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

```
In [ ]: def read_image_file(path):
            with open(path, 'rb') as f:
                data = f.read()
                assert get_int(data[:4]) == 2051
                length = get_int(data[4:8])
                num_rows = get_int(data[8:12])
                num_cols = get_int(data[12:16])
                images = []
                idx = 16
                for l in range(length):
                    img = []
                    images.append(img)
                    for r in range(num_rows):
                        row = []
                        img.append(row)
                        for c in range(num_cols):
                            row.append(parse_byte(data[idx]))
                            idx += 1
                assert len(images) == length
```

So, that is the part of this, Parse byte function. And next is when you would be reading down the image file over there. So, you need to give down your path name over there, and this path name for the file is something which is given down in data pack, which we will subsequently be using.

So, over here once you give down the data path over there, it reads out the data. Then it gets what is the length of the data, the number of rows, number of columns and these comes down from the data header structure in itself. So, once you go down into say torch vision and get into the data sets you will be getting more and more details of how this whole data is packed over there and this is quite simple. So, the MNIST row format is what we are using over here in it is way.

(Refer Slide Time: 06:34)


jupyter lecture12 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

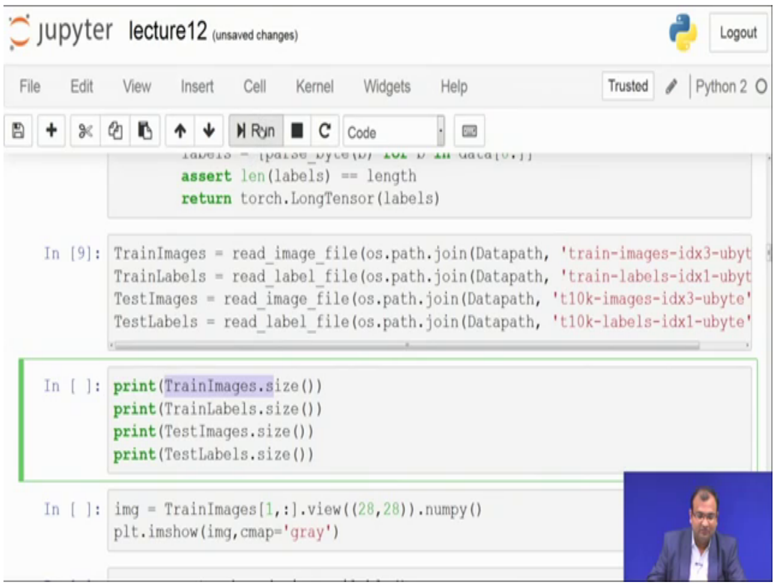
```
            row.append(parse_byte(data[idx]))
            idx += 1
        assert len(images) == length
        return torch.ByteTensor(images).view(-1,784)
```

```
In [ ]: def read_label_file(path):
            with open(path, 'rb') as f:
                data = f.read()
                assert get_int(data[:4]) == 2049
                length = get_int(data[4:8])
                labels = [parse_byte(b) for b in data[8:]]
                assert len(labels) == length
                return torch.LongTensor(labels)
```

```
In [ ]: TrainImages = read_image_file(os.path.join(Datapath, 'train-i
        TrainLabels = read_label_file(os.path.join(Datapath, 'train-l
        TestImages = read_image_file(os.path.join(Datapath, 't10k-ima
        TestLabels = read_label_file(os.path.join(Datapath, 't10k-lab
```

So, that function gets initialized. The next part is to read down the labels as well and which each of these images the labels are also associated with another u byte file and you will have to again pass and get down your single byte level representations. So, that you know whether this image corresponds to 0 or corresponds to 1, 2, 3 up to 9. So, that makes it a 10 class classification problem.

(Refer Slide Time: 06:54)


jupyter lecture12 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

```
        labels = [parse_byte(b) for b in data[8:]]
        assert len(labels) == length
        return torch.LongTensor(labels)
```

```
In [9]: TrainImages = read_image_file(os.path.join(Datapath, 'train-images-idx3-ubyt
        TrainLabels = read_label_file(os.path.join(Datapath, 'train-labels-idx1-ubyt
        TestImages = read_image_file(os.path.join(Datapath, 't10k-images-idx3-ubyte'
        TestLabels = read_label_file(os.path.join(Datapath, 't10k-labels-idx1-ubyte'
```

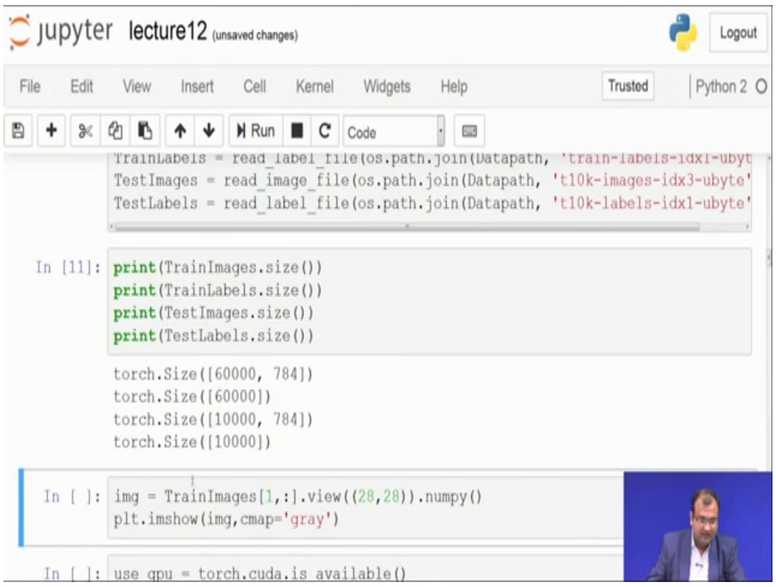
```
In [ ]: print(TrainImages.size())
        print(TrainLabels.size())
        print(TestImages.size())
        print(TestLabels.size())
```

```
In [ ]: img = TrainImages[1,:].view((28,28)).numpy()
        plt.imshow(img,cmap='gray')
```

So, once that is done. So, the next part is you actually execute all of this. The first part is that, I need to get down my training images and my training levels, and for that I will be reading down my image file as well as my label file. And for each of them, the data path for my training is my train images, u byte file. And my, for my training levels it my training labels, u byte file. And for my test images they are 10000 images which are available in your test and you have labels corresponding to each of these 10000 images.

So, for your training you have sixty thousand images, for your testing you have 10000 images. So, let us run this one and you will be able to. So, it would take a bit of time because of this file I O operation going down. So, once this file I O operation is over, then what we would be doing is, we come over here, in this part. Where, the purpose is to print down the size of your training images and labels, and test images and labels. The only purpose we keep on doing it in this way is to see what is the nature of the tensor and whether the number of samples in training images and labels is the same and within your testing images and labels is also the same.

(Refer Slide Time: 08:06)


jupyter lecture12 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

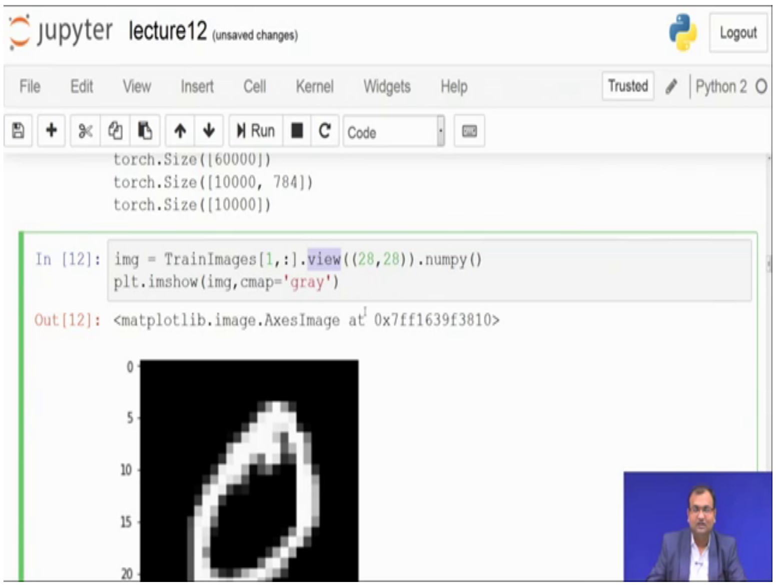
```
TrainLabels = read_label_file(os.path.join(Datapath, 'train-labels-idx1-ubyt
TestImages = read_image_file(os.path.join(Datapath, 't10k-images-idx3-ubyte'
TestLabels = read_label_file(os.path.join(Datapath, 't10k-labels-idx1-ubyte'
```

```
In [11]: print(TrainImages.size())
         print(TrainLabels.size())
         print(TestImages.size())
         print(TestLabels.size())

         torch.Size([60000, 784])
         torch.Size([60000])
         torch.Size([10000, 784])
         torch.Size([10000])

In [ ]:  img = TrainImages[1,:].view((28,28)).numpy()
         plt.imshow(img,cmap='gray')

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

So, you get down your training samples which are 60000 such samples and now, what happens is that each is an image of 28 cross 28 pixels as we had already discussed. So, 28 cross 28 will map down to 784 neurons, linear neurons which you have. So, then you get down your linear neurons coming down over here. So, you know that your training size is 60000 samples cross 784 dimensional feature space, if you would like to put it down on to a pixel wise feature space.

So similarly, you're the size of the test images is. So, there are 10000 samples for your testing and there are 784 pixels on your test part over there. So, doing that, we get into the next part. Here the purpose is basically to try to display it out.

(Refer Slide Time: 08:59)

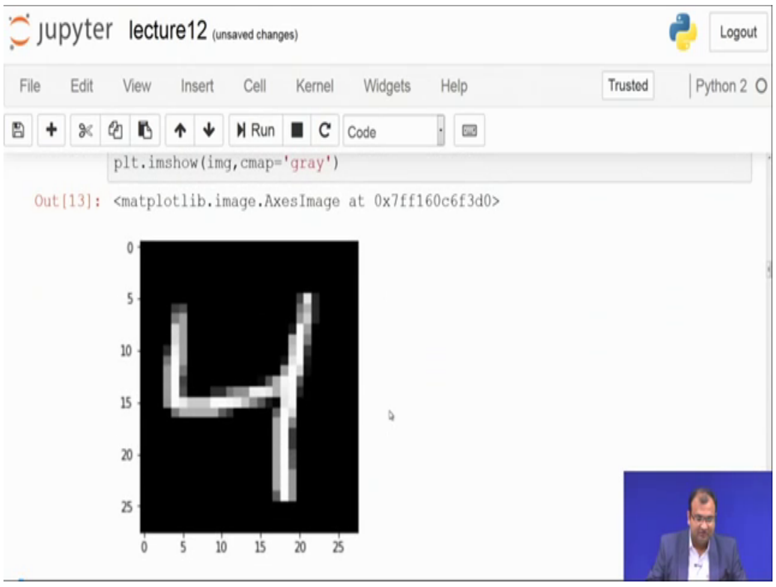

jupyter lecture12 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

```
torch.Size([60000])
torch.Size([10000, 784])
torch.Size([10000])
```

```
In [12]: img = TrainImages[1,:].view((28,28)).numpy()
         plt.imshow(img,cmap='gray')

Out[12]: <matplotlib.image.AxesImage at 0x7ff1639f3810>
```

So, what we do is, we take the first image over there, and then map it down on to a 28 cross 28 form using this view function within torch which is quite interesting. Because what it can do is, as in, if you had used mat lab for your resize or python for your reshape operators, where you can convert one kind of a matrix to another kind of a matrix, say a row matrix or a column matrix into a row cross column or 2D matrix over there. So, we are using the same kind of a function in order to convert my linear matrix available over here into a 2D matrix, and then just type cast it to numpy so that, I can use my pi plot. I am sure in order to look into. So, as you see this is clearly a number 0 which you see over here. I can change this and say make this as 2 and then run this one.

(Refer Slide Time: 09:56)

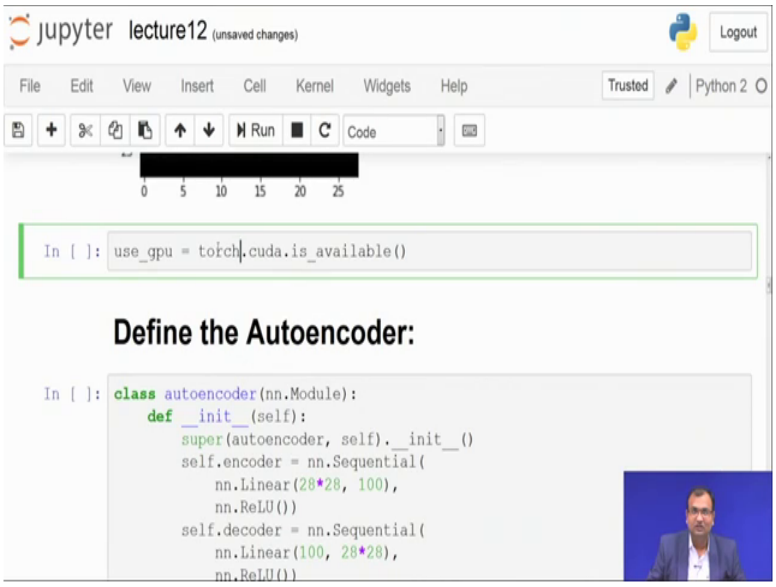

jupyter lecture12 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

```
plt.imshow(img,cmap='gray')
Out[13]: <matplotlib.image.AxesImage at 0x7ff160c6f3d0>
```

So, the second image which I have over here is, something which looks like this and then that is something like number 4. So, basically you can just play around with them and then plot down a few of these images as well, now I can look into my test image as well.

So, I can just go over here and then make that change into image equal to test image and. So, this is my test image. So, the second test image is basically, the number 1, as it comes down. So for us, as humans, it much easier to understand but, the whole point is that, can we make machines actually be that intuitive to understand these using neural networks also.

(Refer Slide Time: 10:34)


jupyter lecture12 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

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

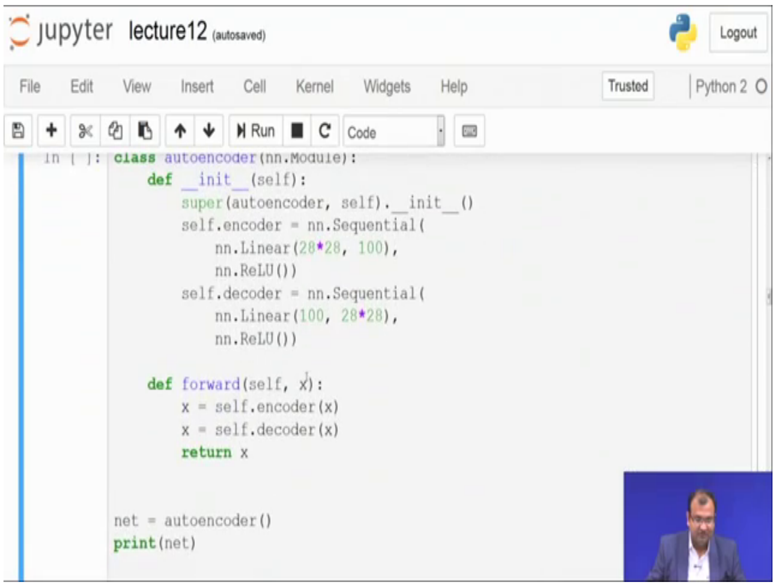
Define the Autoencoder:

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

So, here goes another of our commonly known option, which was to have down. If cuda is available then just find out whether there is a gpu so that we can use this flag of, use gpu in our subsequent works as well.

So, here comes the first part, and which is about defining your network. So, remember that we were speaking about an auto encoder as such.

(Refer Slide Time: 10:53)


jupyter lecture12 (autosaved)
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, 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


        net = autoencoder()
        print(net)
```

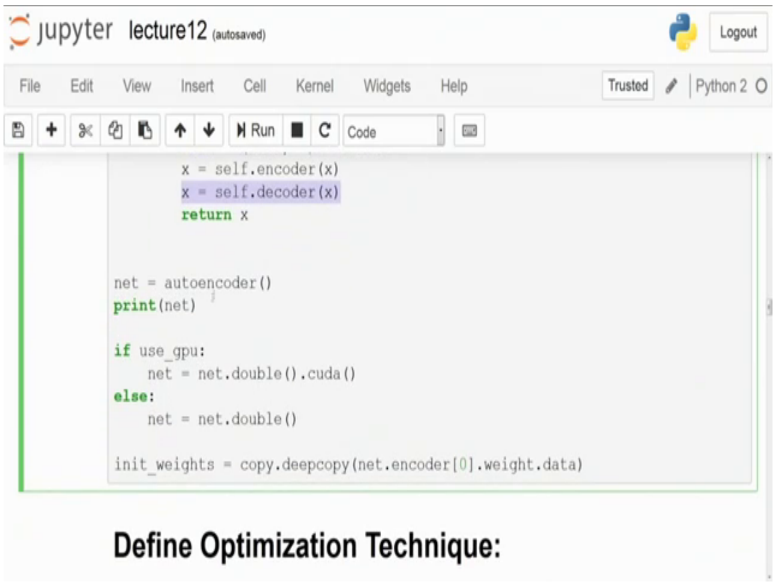
So, an Autoencoder is basically where you have an input, you have a hidden layer and then you have an output and the size of the output is the same as size of the input. And technically what you are trying to do is, you are going to map this input to become. So, whatever is the output that has to be similar to the, as close as possible to the input itself. And that brings is pretty much in to the paradigm of what is called as a regression problem. Where your loss or your cost function over there has to be something in the order of say and l 2 norm, l 1 norm or l 2 norm or an MSE: a mean square error.

So, if is that error over there is 0, it means that all values are mapped and it is on a continuous space. So, here what we try to do is basically we try to represent linear for a fully connected network from 784. So, that is 28 cross 28. Now those get mapped on to 100 hidden neurons over there, that is the first hidden layer and the transformation applied over here is a ReLU or a rectified linear unit to go now.

So, the other part of the network this, typically this part of the network which brings down from the image input to my representation space is, what is called as an encoder unit. Now the other part of it which is my decoder unit which goes down from this representation space the latent representation of space onto my reconstructed space which is called as a decoder.

So, you remember these clearly from our last slides as well. So, that is where it converts from 100 neurons to 28 cross 28. And then the objective is that, within the forward function, whatever is the input that will be going through one forward transformation through the encoder and then a forward transformation through the decoder itself and then you have your Autoencoder form.

(Refer Slide Time: 12:37).


jupyter lecture12 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

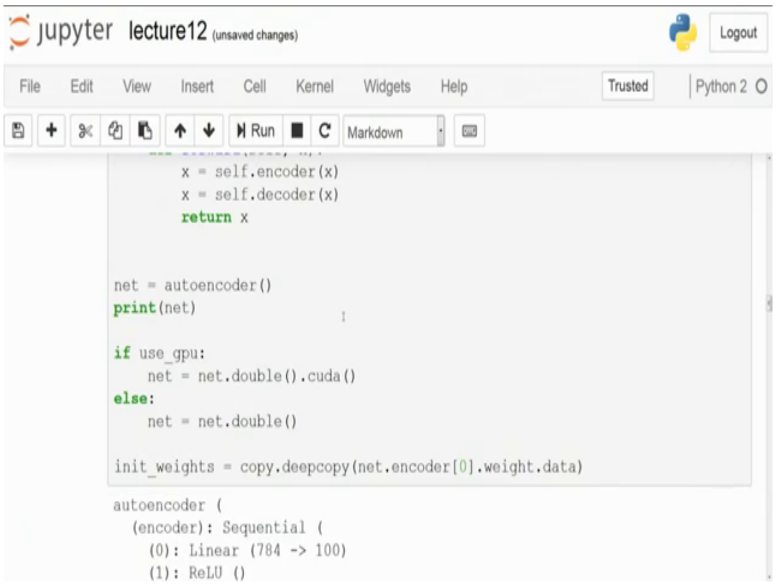
```
        x = self.encoder(x)
        x = self.decoder(x)
        return x


net = autoencoder()
print(net)

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

init_weights = copy.deepcopy(net.encoder[0].weight.data)
```

Define Optimization Technique:

And now, if your gpu exists, then you can convert it otherwise you can just leave that for your CPU options over there. And the next point is that we would try to get down these weights for our own usage at in a bit later on stage down the code.

So, this is using this function called as deep copy. So, what it does is that, it basically copies down all the weights available at this point of time. So, what I would like to show you a bit later on is that, while we are training you would be seeing down the change in weights coming down ok.

(Refer Slide Time: 13:12)

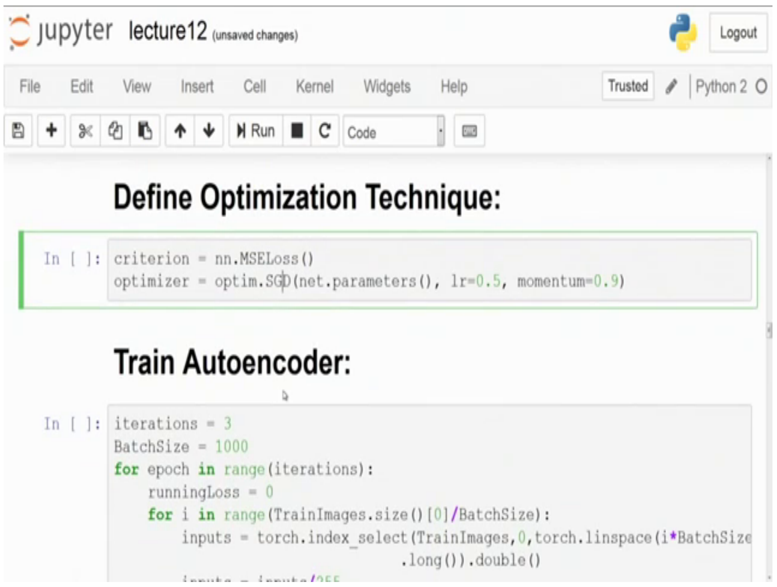

jupyter lecture12 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Markdown

```
        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()

init_weights = copy.deepcopy(net.encoder[0].weight.data)
```

```
autoencoder (
  (encoder): Sequential (
    (0): Linear (784 -> 100)
    (1): ReLU ()
```

So, let us run this function and then do it. So, once the Autoencoder is defined, because I was printing my network over here, this is what the Autoencoder looks like. So, you have linear units which are fully connected neurons which connect from 784 to 100 and then on the decoder you have from 100 to 784.

(Refer Slide Time: 13:36)


jupyter lecture12 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

Define Optimization Technique:

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

Train Autoencoder:

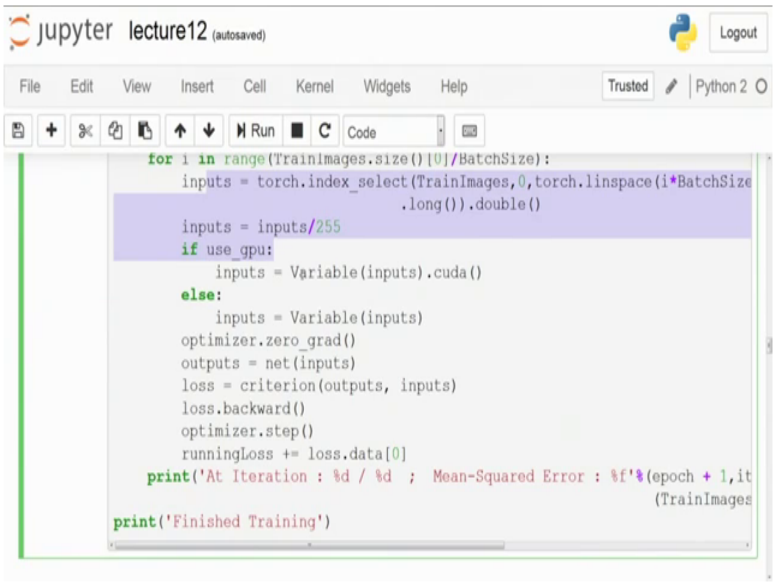
```
In [ ]: iterations = 3
        BatchSize = 1000
        for epoch in range(iterations):
            runningLoss = 0
            for i in range(TrainImages.size()[0]/BatchSize):
                inputs = torch.index_select(TrainImages,0,torch.linspace(i*BatchSize
                                            .long()).double()
```

So, here is where we start by defining the criterion function or the loss function. So, for me the loss function is an MSEloss and the optimizer. So, I said that we will be making use of another new thing which is called as an optimizer. So, this optimizer helps me in writing down, reducing down the bulk of codes which we would be using, but details I would be covering down in a bit later on lectures down the week.

So, as of now what you can remember is that, this optimizer is basically one sync simple way of computing your whole gradient and doing the back propagation in one single go. Now, for the first part is to train down the auto encoder or where we are going to do down the representation learning part over there. So, I had set it down for 3 for convenience, but say let us make it as 10. So, I will train down this auto encoder for 10 epochs over there, and we will take a batch size of 1000 images. So that means, that what will happen is that, you have total 60000 images for your training and I can break it down into 60 different sets of 1000 images each.

So, after 1000 image, I am going to calculate my error and then update my network parameters over there, and within epochs this will happen 60 times basically. Now what I would run is, 1 iterator or the for loop over the range of epochs. So, my epochs is basically numb 10 epochs over there and then within each epoch what I am going to do is, pull down my inputs and once the inputs are available, I would be converting them on to a gpu array.

(Refer Slide Time: 15:11)

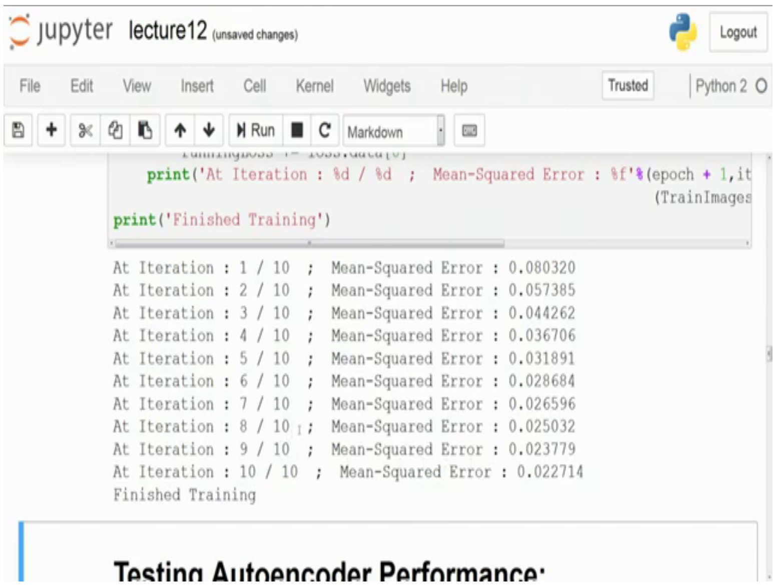


```
    for i in range(TrainImages.size()[0]/BatchSize):
        inputs = torch.index_select(TrainImages,0,torch.linspace(i*BatchSize
                                         .long()).double()
        inputs = inputs/255
        if use_gpu:
            inputs = Variable(inputs).cuda()
        else:
            inputs = Variable(inputs)
        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, inputs)
        loss.backward()
        optimizer.step()
        runningLoss += loss.data[0]
    print('At Iteration : %d / %d  ;  Mean-Squared Error : %f'%(epoch + 1,it
                                                                (TrainImages
print('Finished Training')
```

If my gpu is available, once that is done, do you remember that we had a model zero grad or which was making all the gradients within the model as 0. So, here it becomes as optimize zero grad and my. So, my output was the forward pass over my network, then my loss was coming from my criterion function and then I had to find out my derivative of the loss which is my backward.

Now, the next part is where I had w of n plus 1 is equal to w of n minus eta times of del del w of j at w of n. So, that is what is solved on by this optimizer step function over there. Once that is done, So this is whole update rule or backward back propagation is what is solved over here, then we can get down our running loss calculated down and then eventually, print down per epoch what is the loss coming down.

(Refer Slide Time: 16:16)


jupyter lecture12 (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
                                                                  (TrainImages
print('Finished Training')
```

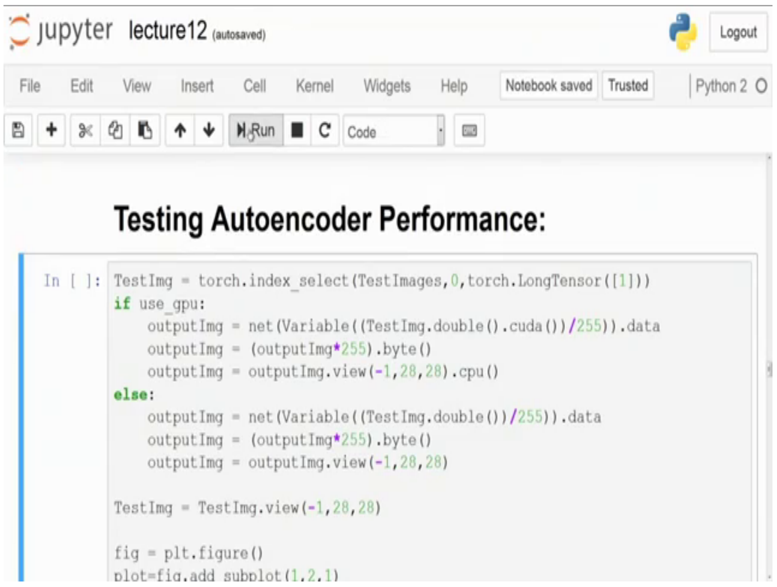
```
At Iteration : 1 / 10  ;  Mean-Squared Error : 0.080320
At Iteration : 2 / 10  ;  Mean-Squared Error : 0.057385
At Iteration : 3 / 10  ;  Mean-Squared Error : 0.044262
At Iteration : 4 / 10  ;  Mean-Squared Error : 0.036706
At Iteration : 5 / 10  ;  Mean-Squared Error : 0.031891
At Iteration : 6 / 10  ;  Mean-Squared Error : 0.028684
At Iteration : 7 / 10  ;  Mean-Squared Error : 0.026596
At Iteration : 8 / 10  ;  Mean-Squared Error : 0.025032
At Iteration : 9 / 10  ;  Mean-Squared Error : 0.023779
At Iteration : 10 / 10  ;  Mean-Squared Error : 0.022714
Finished Training
```

Testing Autoencoder Performance:

So, let us run this part and you would see it running now. So, you see that it is trains over 10 iterations as we had done. So, iteration is basically 1 epoch which we are using down over there. And this is what is plotting down the mean square, because that is the loss over which we are calculating it. You can see it steadily decreasing. It starts where somewhere around 0.08 and then comes down to 0.022 and then there is a steady decline as well.

So, this is not a, we do not know whether it has actually stopped or not, but this was just to show you that, how much it can actually go down. Next is to check down your Autoencoder performances.

(Refer Slide Time: 16:47)


jupyter lecture12 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Notebook saved Trusted Python 2
Run Code

Testing Autoencoder Performance:

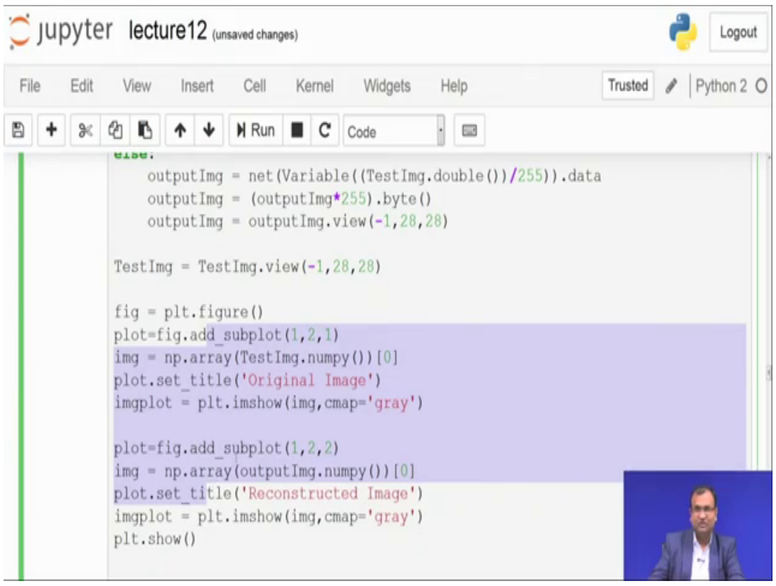
```
In [ ]: TestImg = torch.index_select(TestImages,0,torch.LongTensor([1]))
        if use_gpu:
            outputImg = net(Variable((TestImg.double().cuda())/255)).data
            outputImg = (outputImg*255).byte()
            outputImg = outputImg.view(-1,28,28).cpu()
        else:
            outputImg = net(Variable((TestImg.double())/255)).data
            outputImg = (outputImg*255).byte()
            outputImg = outputImg.view(-1,28,28)

        TestImg = TestImg.view(-1,28,28)

        fig = plt.figure()
        plot=fig.add_subplot(1,2,1)
```

So, let us look into your, what was it trying to reconstruct over that. So, let us give down one image over here. So, what we have done over here is that, we take one of these test images.

So, just as any one of these test images coming down over here now if the test image is coming to us and a gpu is available then we just convert that over here onto my cuda compatible form and then from there make it into a 784. Because it was a 28 cross 28 image which was coming to me. So, I just converted to 784 dimensional input and then that is fed forward through my network and then I am again back converting it as my output image.

Now, the whole point over here is that, I will be, I will try to display them as standard grayscale images and that is what I do using these 2 parts of my function over there.

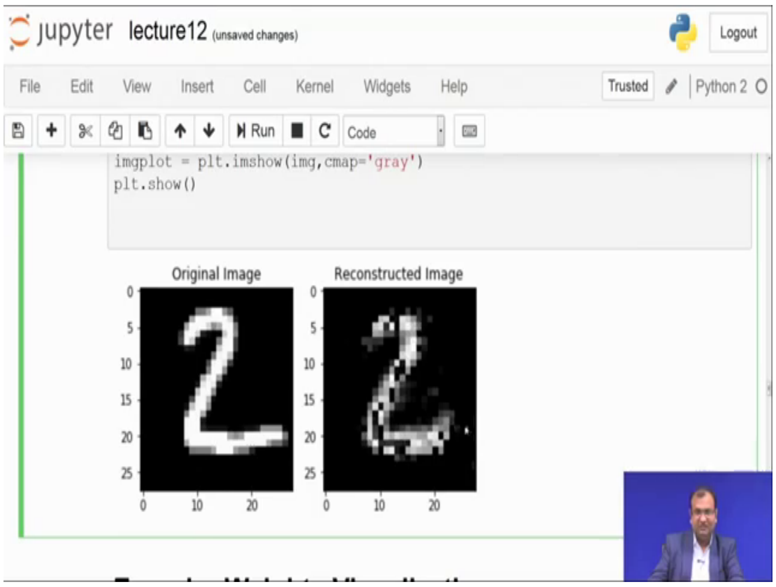
(Refer Slide Time: 17:56).


jupyter lecture12 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted Python 2
Run Code

```
else:
    outputImg = net(Variable((TestImg.double())/255)).data
    outputImg = (outputImg*255).byte()
    outputImg = outputImg.view(-1,28,28)

TestImg = TestImg.view(-1,28,28)

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

plot=fig.add_subplot(1,2,2)
img = np.array(outputImg.numpy())[0]
plot.set_title('Reconstructed Image')
imgplot = plt.imshow(img,cmap='gray')
plt.show()
```

Now pretty straightforward to go through and I am not going into much of a details now. If you look into here, you would see that you see somewhat the shape and structure of this number 2 as preserved which comes down over here, though there are some noisy patches as well.

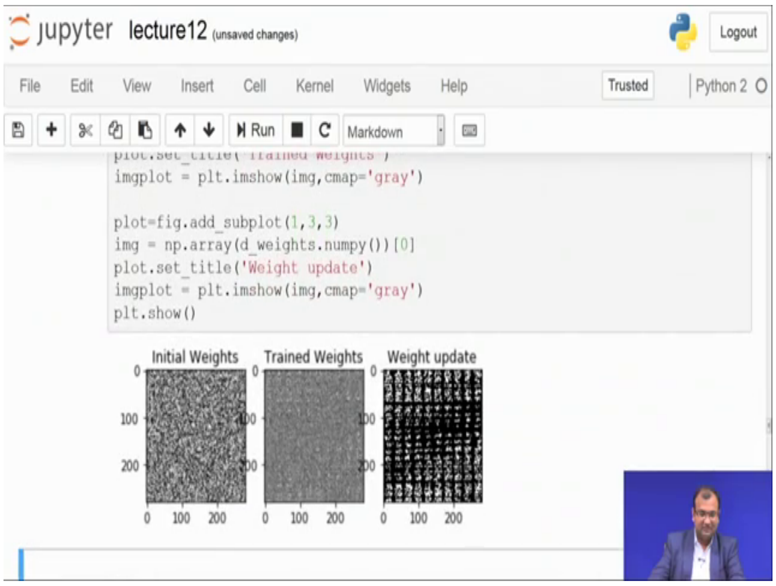
(Refer Slide Time: 18:02)


jupyter lecture12 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted Python 2
Run Code
Original Image
Reconstructed Image
0
5
10
15
20
25
0
10
20

```
imgplot = plt.imshow(img,cmap='gray')
plt.show()
```

So, possibly if you train over a longer number of epochs, you will be seeing down these errors going down to a significant extent as well. So, the next part is that, we try to run down some sort of a visualization in terms of what weights it had learned.

(Refer Slide Time: 18:23)



```
plot.set_title('Trained weights')
imgplot = plt.imshow(img,cmap='gray')

plot=fig.add_subplot(1,3,3)
img = np.array(d_weights.numpy())[0]
plot.set_title('Weight update')
imgplot = plt.imshow(img,cmap='gray')
plt.show()
```

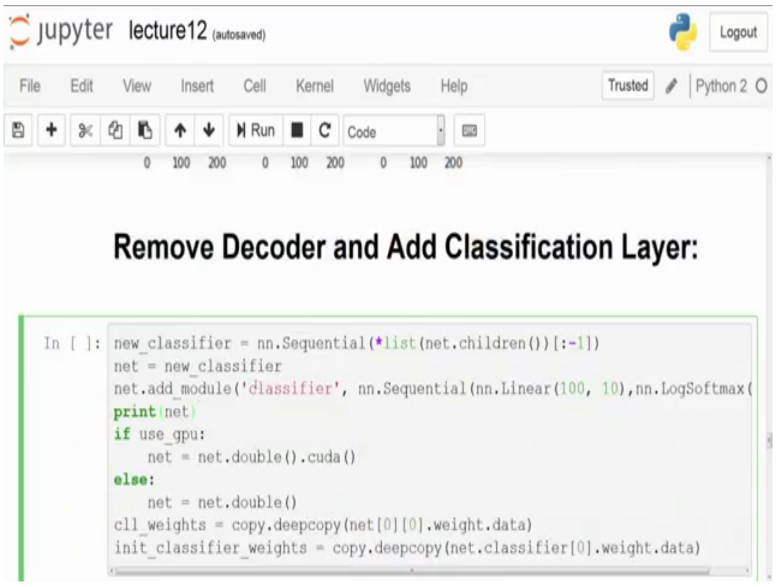
jupyter lecture12 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Markdown
Initial Weights
Trained Weights
Weight update

So, these were basically the sort of weight matrix which was there before we started the training over there. And then this is, sort of the weight matrix which looks into after your training. So, these are basically small square matrices of 28 cross 28. And you have 100 of such app label. So, this axis basically makes it 280, this axis is 280, and this is a small square patches of each of them plotted down.

Now what we try to see is, what is the difference of these two weights and that has what. So, these are the pixels or these are the weight locations which actually got updated, during training. And you see that a significant number of them have been updated and that is a training part which goes on. So, this is for simple visualization, but there are more detailed explanation which are, which where there in the theory part and we have a few of them in the next of the theory part and these are more intuitive for your understanding and how to get more of them running.

So, once that is done, the next part is to get into, using an auto encoder for multi layer perceptron based classification. So, here what I would be doing is that I no more need my whole auto encoding part over there. So, my decoder block can be detached and I can place down a multi layer perceptron in order to just classify. So, here for my multi layer perceptron, I just need 10 layers which will correspond to my outputs over there for each of these 10 classes. So, I get my input, I convert it to 118 variables and from there I bring it down to 10 different classes and it has to be 1 hot as per my definition.

(Refer Slide Time: 19:58)



jupyter lecture12 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code
0 100 200 0 100 200 0 100 200

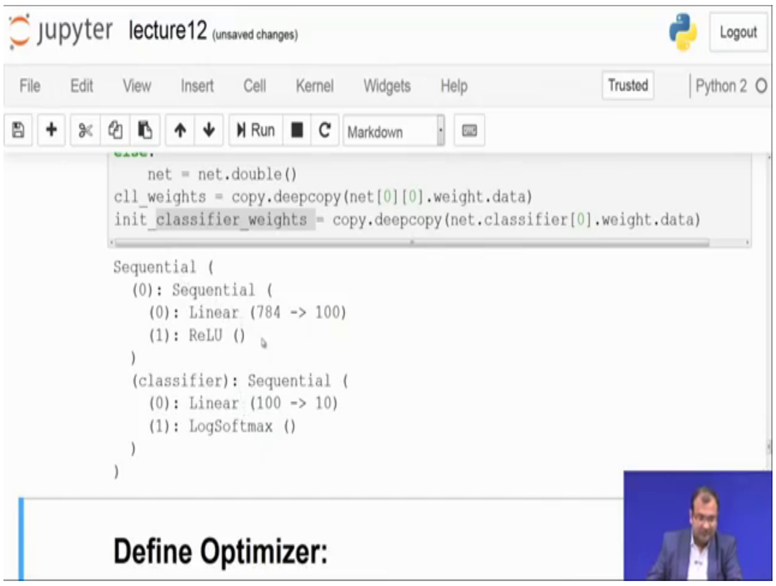
Remove Decoder and Add Classification Layer:

```
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.double().cuda()
        else:
            net = net.double()
        cl1_weights = copy.deepcopy(net[0][0].weight.data)
        init_classifier_weights = copy.deepcopy(net.classifier[0].weight.data)
```

So, what I do over here is a very simple part which is, so we take down the first part of my network, which is my encoder. And from there, I guess add on an extra part which is my classifier or just a mapping from 100 to 10 neurons over there and finally, have a transfer function which is no more a ReLU, but a LogSoftmax over there. And then, the option is basically if everything is available on a gpu, then to convert that and then used on my copied weights from this earlier train part. So, these were my weights on the encoder part of it. So, I just copy them down over there and then use those weights in order to initialize my initial part over there.

Now, once that is done, let us execute this part.

(Refer Slide Time: 20:52)


jupyter lecture12 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Markdown

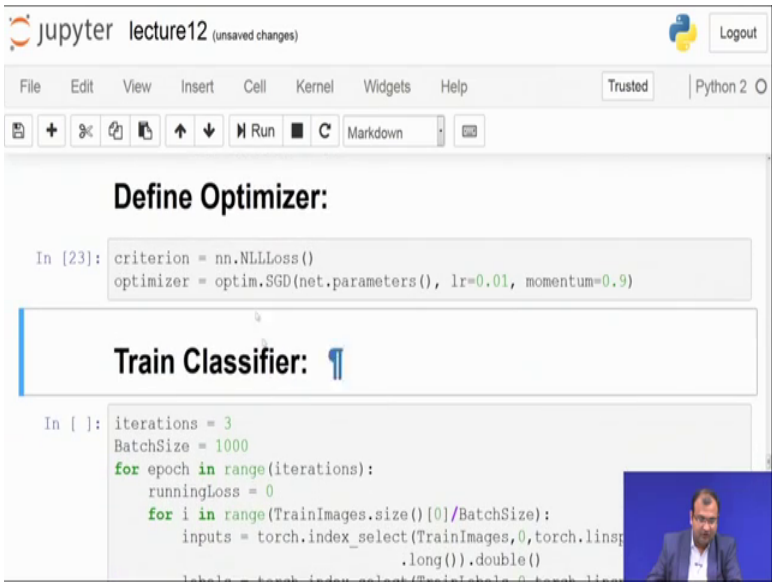
```
    net = net.double()
cll_weights = copy.deepcopy(net[0][0].weight.data)
init_classifier_weights = copy.deepcopy(net.classifier[0].weight.data)
```

```
Sequential (
  (0): Sequential (
    (0): Linear (784 -> 100)
    (1): ReLU ()
  )
  (classifier): Sequential (
    (0): Linear (100 -> 10)
    (1): LogSoftmax ()
  )
)
```

Define Optimizer:

So, you can see the network looks something like this, that I have 784 neurons connected down to 100 neurons and with a ReLU transfer function and then 100 neurons to 10 neurons with the LogSoftmax transfer function.

(Refer Slide Time: 21:09)


jupyter lecture12 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Markdown

Define Optimizer:

```
In [23]: criterion = nn.NLLLoss()
         optimizer = optim.SGD(net.parameters(), lr=0.01, momentum=0.9)
```

Train Classifier: ¶

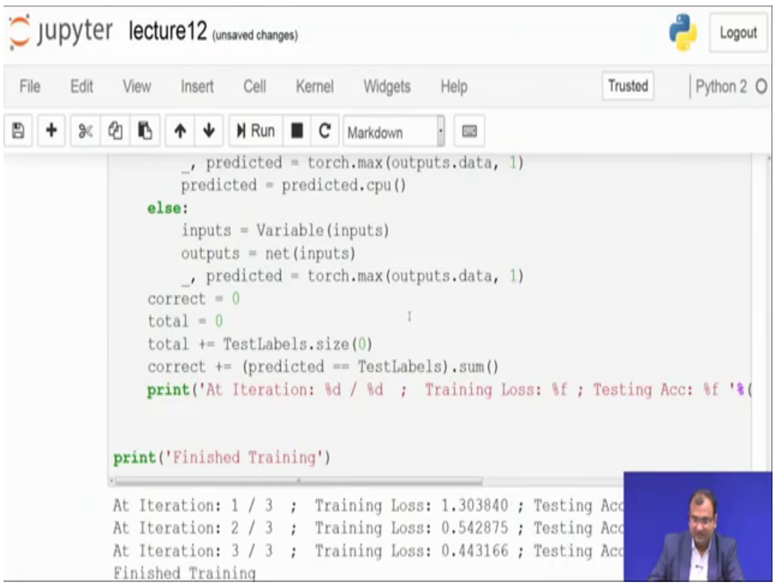
```
In [ ]: iterations = 3
        BatchSize = 1000
        for epoch in range(iterations):
            runningLoss = 0
            for i in range(TrainImages.size()[0]/BatchSize):
                inputs = torch.index_select(TrainImages,0,torch.linsp
                                    .long()).double()
```

Then on this one, I also defined down my loss, but this loss is quite different. This is a negative log likelihood loss or a classification loss, not exactly a mean square error loss.

So, a bit down the line when we are doing cost functions, later on I will be introducing you to, what is the form of a negative log likelihood loss? Or to keep it much more simpler, what you can understand is that, in case of a mean square error, you would see down differences in terms of. So, in a mean square error, we were basically trying to minimize the Euclidean norm between your input and output over here.

Now for a classification, when it is just a 1 hot vector, you know that one of them is supposed to be high and everything else is going to be 0. So, these kind of ones where one of them is high and everything else has a 0 probability, you can use some sort of information theoretic loss. And those kind of losses, or one of those kind of loss is basically a negative log likelihood loss. So, I defined my optimizer as well over there and then get into my classifier. So, here say I just decide to run it down for 3 epochs ok.

(Refer Slide Time: 22:20)

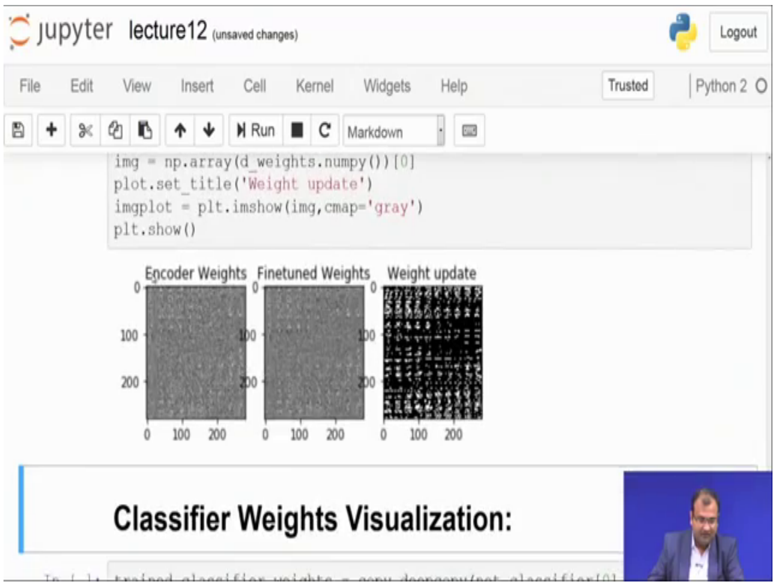

jupyter lecture12 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Markdown

```
            _, predicted = torch.max(outputs.data, 1)
            predicted = predicted.cpu()
        else:
            inputs = Variable(inputs)
            outputs = net(inputs)
            _, predicted = torch.max(outputs.data, 1)
        correct = 0
        total = 0
        total += TestLabels.size(0)
        correct += (predicted == TestLabels).sum()
        print('At Iteration: %d / %d  ;  Training Loss: %f ; Testing Acc: %f '%(


print('Finished Training')
```

```
At Iteration: 1 / 3  ;  Training Loss: 1.303840 ; Testing Acc
At Iteration: 2 / 3  ;  Training Loss: 0.542875 ; Testing Acc
At Iteration: 3 / 3  ;  Training Loss: 0.443166 ; Testing Acc
Finished Training
```

So, I can use my initialize fits and then get into over here. Now if you typically look over here, we were also calculating out what the accuracy of prediction. And you see that while the loss is going down, your accuracy is also quite increasing over there. And then, we come down to visualizing these weights basically. So, these were the weights of this encoder before they were finally used for the classification part over there or end to end update of the whole network and these were the ones after the subject happened. And this is the amount of changes which are there on each of them.

(Refer Slide Time: 22:40)


jupyter lecture12 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Markdown
Encoder Weights
Finetuned Weights
Weight update

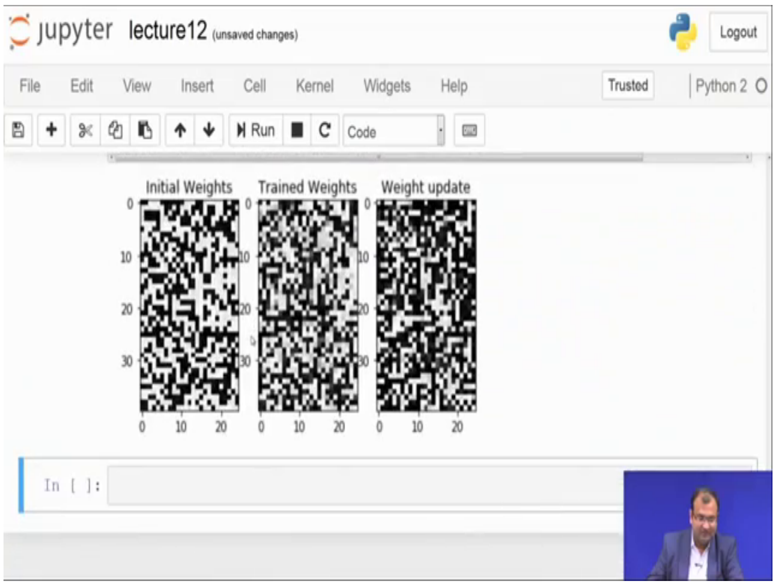
```
img = np.array(d_weights.numpy())[0]
plot.set_title('Weight update')
imgplot = plt.imshow(img,cmap='gray')
plt.show()
```

Classifier Weights Visualization:

So, that does bring in to the fact that, while we are doing a fine tuning of the whole multi layer perceptron using our initialized versions of weights from the Autoencoder, it does significantly impact some of these weights. If you if you had trained this say the feature layer, feature encoding layer for a longer amount of epochs and over here also for a longer amount of, maybe this would go down actually because we have not yet seen a convergence on the encoder.

So, for my typical experiences on this data set was that, run it down for 100 epochs; it would go out pretty fine. But then, we would run out of time if we are trying to do that on this lab class itself.

(Refer Slide Time: 23:39)


jupyter lecture12 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code
Initial Weights
Trained Weights
Weight update
In [ ]:

So, here we try to visualize down the weights for the training for the classification part over there. So, these were the initially random weights which allotted to the classification network which is from 100 to 10 neurons over there. And this is what you see down for the, after the training. And this is amount of updates we just went on.

So, this is one basic scratchpad of how to do it with the MNIST. So, in the next class I will be getting you introduced onto 2 different forms of what are encoder training. So, one of them was, where we have a ladder voice training or 1 network at a time and the other one is, where you have a bunch of hidden layers and you train it as a non encoder decoder network in one single stretch. And then use these initialization in order for your mlp feature extraction or the initial layers over there. But there we will be using another higher order data set and that is called as the fashion MNIST.

So, that is also a gray scale image available and is of the order of MNIST dataset itself except for that they are no more handwritten digits or binary like images present over there. But these are perfect grayscale ones, where you have intensities varying from 0 to 255, over a good linear span as well and these are images of clothes. So, 10 different classes of apparel and small 28 cross 28 snapshots of each of them and 60000 for training and 10000 for testing.

So subsequent to that, we will also be doing on color images with something called as a ALL-IDB data set. So, that would also perfectly work out and subsequent to that. So, these are all patch voice classifications and eventually in the last Autoencoder lab, we will be doing down pixel to pixel classification as well. So, with that it works out quite good.

Thank you and stay tuned for the next lecture.