

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

Lecture – 19
Sparse Autoencoders for MNIST classification

So, welcome and now today, we would be doing one exercise, which is on Sparse Autoencoders and while in the last lecture, I have explained you exactly where and how Sparse Autoencoders come to place. So, while we are doing this particular exercise using the MNIST digits classification problem, you would be exposed on to how to incorporate Sparsity within your code and then how does this have a subsequent down role to play down.

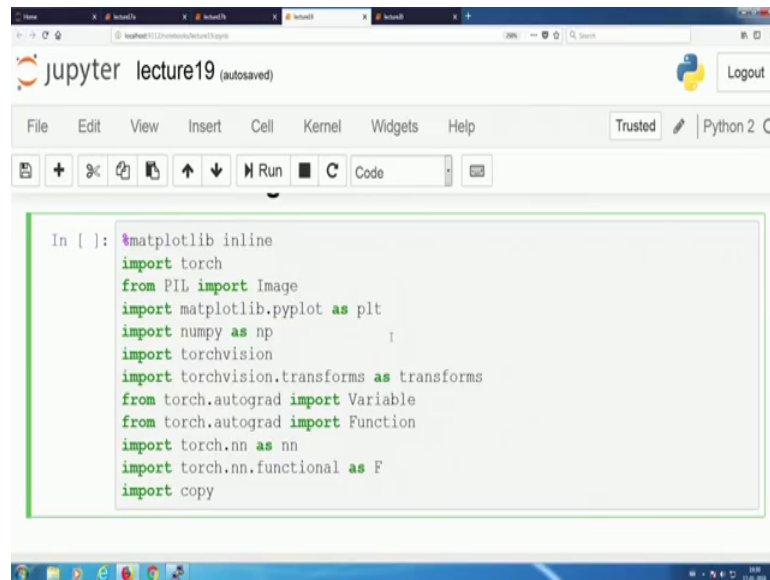
So, you clearly remember, when we were doing down our earlier lecture on Sparse Autoencoder. So, there was L1 norm, which was introduced over there or what was also called as a L1 penalty loss and then that is basically to find out the total number of zeros, which are present down and then just do some sort of a kl divergence between the total number of zeros present and the total actual desired number of zeros to come down. So, that was a extra loss, which we were adding down as an extra amount of error to the whole system and the whole objective was that till we come down to the point where we desired to have. We actually have that many number of 0s within the weight matrices as we would desire to have over there. This error is still going to be a higher value and then it would come down.

So, technically, if I desired that 30 percent of my weights over there are completely 0s and if it is not up to that level, then it has a higher error and on the other side of it say, I desired that it is 30 percent of them are 0s, but then 80 percent of my weights turn over 0. So, that's mean that I have a system which is learning to be heavily redundant, but I do not want the system to be that rate on it. So, I can actually, because of a positive error coming down, because of that k l divergence function over there you would see this metric actually coming to a point, where it would come down to that 30 percent point over there.

So, that is a typical example, which we will be doing; however, given one thing in mind that we have just looked into one kind of our cost function, one kind of a loss function

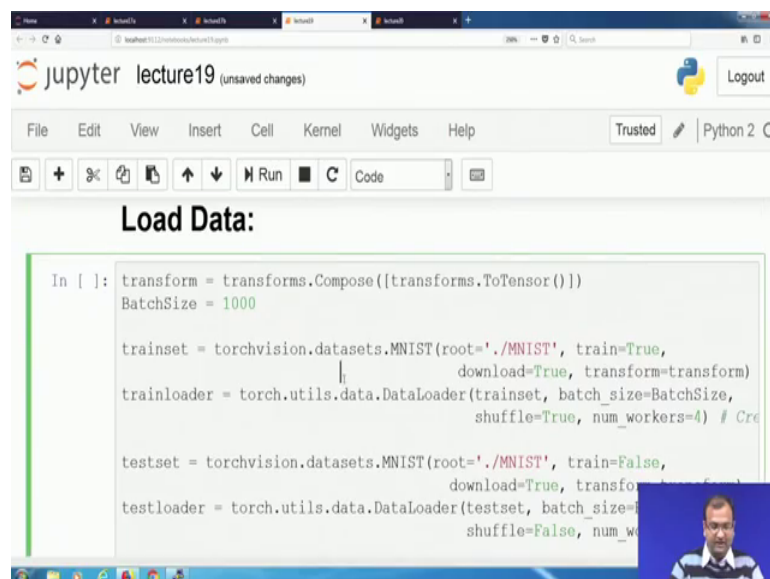
and not these two kinds of different loss functions. So, how to incorporate that within your network and whether that has any changes in the forward pass and in the backward pass is, what we will be doing today.

(Refer Slide Time: 02:21)



```
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 copy
```

(Refer Slide Time: 02:30)



```
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)

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)
```

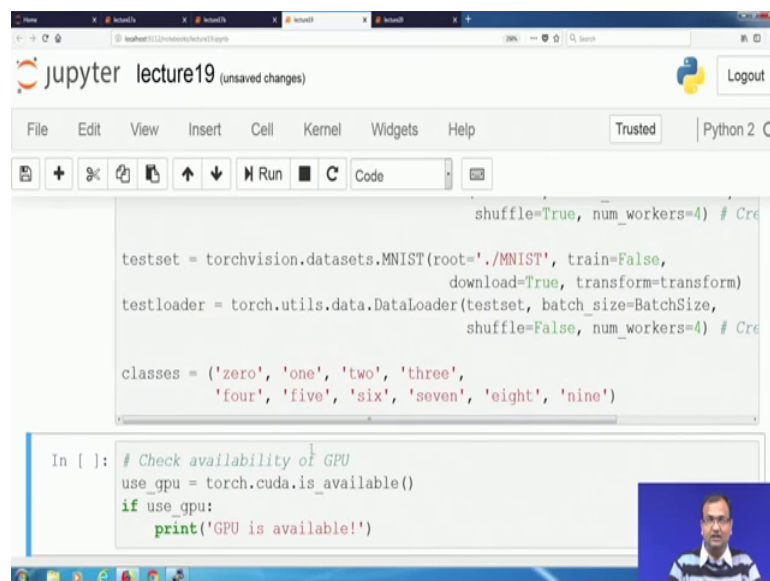
So, starting from the first part which is quite simple and just on loading down your libraries and then eventually coming down to you are loading down the data. So, we are using the same architecture and same kind of a notion as in any of our earlier experiments with MNIST. So, you have 60,000 training samples and 10,000 testing

samples and what we are doing is, we call up the data set from our torch vision data, sets point and the advantage which we get down is that I have already downloaded everything and kept it over there.

So, with iterative runs I do not need to. So, if you are not doing a refresh clone or not just deleting the older archive, wherever it is, put down and then putting down the newer one, then you know that pretty much everything is over there. So, your data sets are also preserved and at this point of thing one suggestion which I would give, is that you can keep on downloading newer assignments as they keep on coming, on the jit link over here, but then please do not delete the older folders over that in which your earlier data sets were kept, otherwise every time you do a run and if your data set folder is completely deleted then you would see this whole messing up. So, one simple thing is that always JIT clone onto the same location.

So, you know that it is always refreshed by the newer version of files, which have come down from the JIT repository and if you have made some modifications and kept it then please make a copy on your separate JIT account and upload and keep it for your own purposes and do not disturb the main repository, which is downloaded and kept on. So, here we start with the same way of you, having for workers in parallel loading down in batch sizes of 1000 samples.

(Refer Slide Time: 04:06)



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

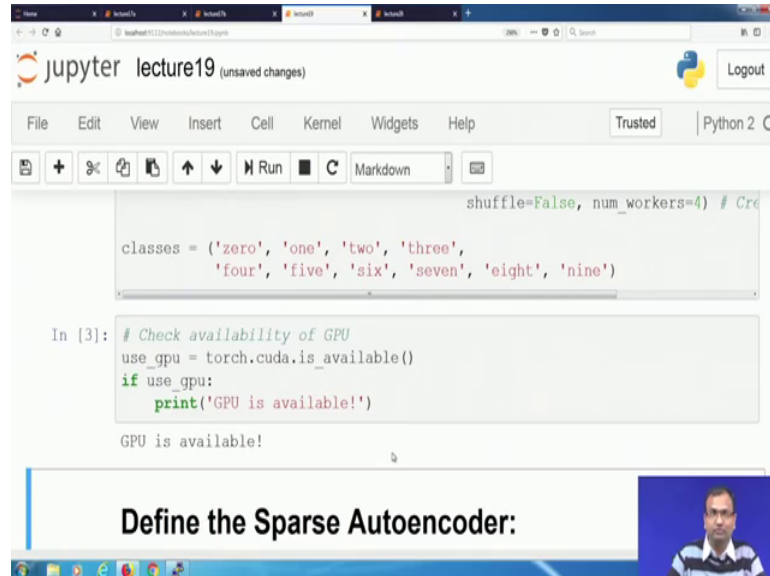
testset = torchvision.datasets.MNIST(root='./MNIST', train=False,
                                     download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=BatchSize,
                                          shuffle=False, num_workers=4) # Cre

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!')
```

So, that goes out and then I have my GPU availability, check over here which checks out my GPU is available.

(Refer Slide Time: 04:13)



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

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

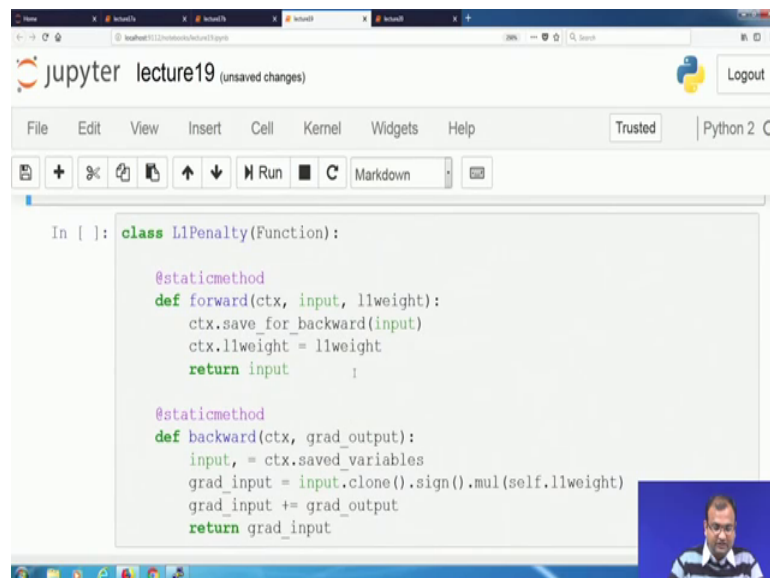
In [3]: # Check availability of GPU
use_gpu = torch.cuda.is_available()
if use_gpu:
    print('GPU is available!')

GPU is available!
```

Define the Sparse Autoencoder:

So, I can actually use my best way of accelerating out my codes. Now, we start with defining what comes down as a sparse auto encoder.

(Refer Slide Time: 04:27)



```
In [ ]: class L1Penalty(Function):

    @staticmethod
    def forward(ctx, input, llweight):
        ctx.save_for_backward(input)
        ctx.llweight = llweight
        return input

    @staticmethod
    def backward(ctx, grad_output):
        input, = ctx.saved_variables
        grad_input = input.clone().sign().mul(self.llweight)
        grad_input += grad_output
        return grad_input
```

Now, you see two different parts over there, one is that I initially start by defining a newer kind of a class and this class of this particular class, which I define over here is what is called as a L1 penalty. Now, it becomes a bit confusing at this point of time to a

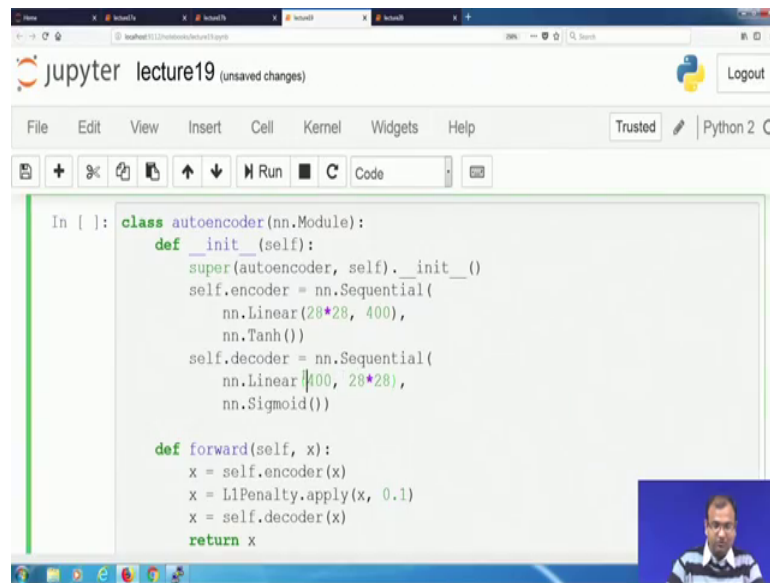
lot of people, but let us keep it simple and do it. So, here the whole point was that while I am actually doing this autoencoder. So, where is the first time, where you would be looking into your sparsity.

So, you had say some 784 neurons, which were connected down to 400 neurons and you just had these 400 neurons, again mapping back to 784 neurons and this was the simple auto encoder, which you were creating. Now, I would like to add down sparsity at the end of this first layer or which will look into the sparsity of the weights in my connections from the first hidden layer to my output, which is being created over there, great and the whole objective is that these layers will be my connections from my hidden layer to my output layer, will be sparse only if there is a over complete representation present in my earlier layer, which is for my input to my hidden layer.

This has to be there, otherwise we cannot ensure sparsity in any way and it goes around the web in order to ensure sparsity, you will have an over complete representation and you will, whenever you have an over complete representation you will have a sparsity. So, this has to go hand in hand, with the learning paradigm over there. Now, comes the two different aspects of it. So, one is that I would need to define, what is on my forward pass? So, in my forward pass, what I am trying to get down is basically, I find out that the total number of weights, which have a value of closer to 0.

This is what I just find it out whereas, when I do my reverse pass or my back propagation then I do not want any of these to have any impact and changes, while I need to keep one thing in mind that these parts autoencoders. They would ensured that, you keep on like wherever it becomes a 0, then that 0 is stored over there and you, while in back propagation you also have the same thing in that.

(Refer Slide Time: 07:22)

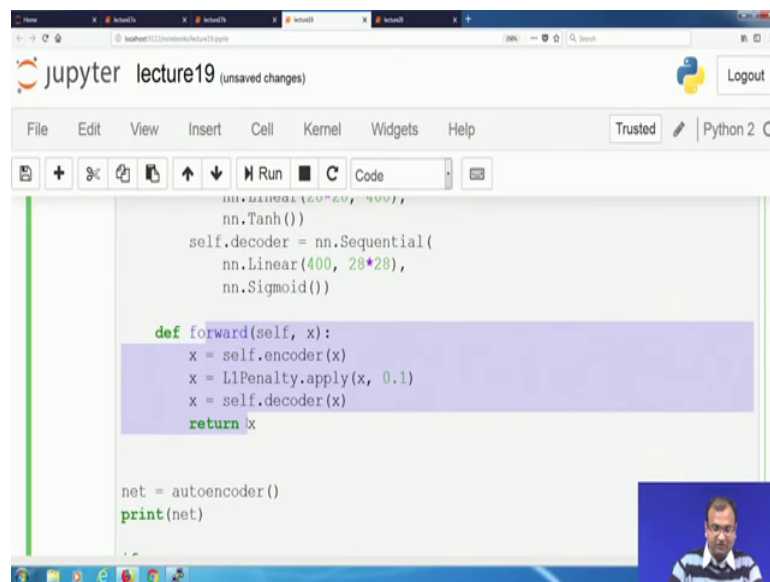


```
In [ ]: class autoencoder(nn.Module):
        def __init__(self):
            super(autoencoder, self).__init__()
            self.encoder = nn.Sequential(
                nn.Linear(28*28, 400),
                nn.Tanh())
            self.decoder = nn.Sequential(
                nn.Linear(400, 28*28),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

        def forward(self, x):
            x = self.encoder(x)
            x = L1Penalty.apply(x, 0.1)
            x = self.decoder(x)
            return x
```

So, my auto encoder was that I had 784 neurons connected down to 400 neurons and 400 neurons to 784 neurons on my decoder set.

(Refer Slide Time: 07:32)



```
nn.Tanh())
self.decoder = nn.Sequential(
    nn.Linear(400, 28*28),
    nn.Sigmoid())

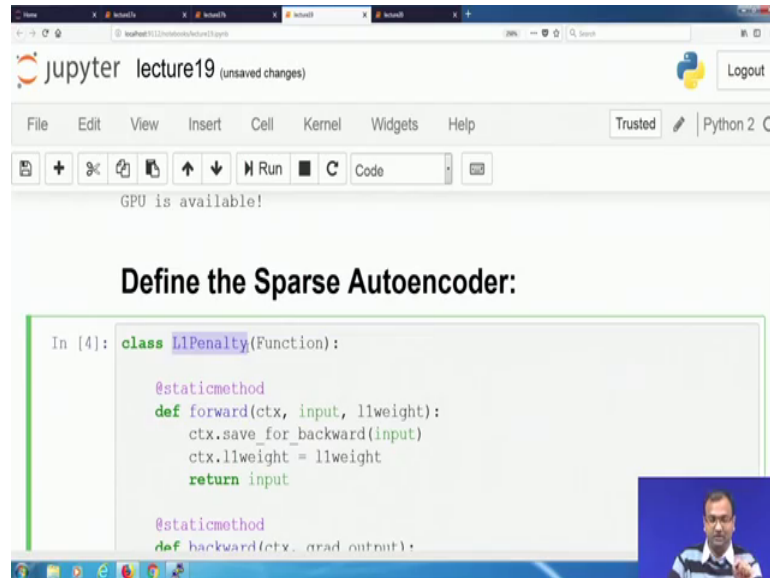
def forward(self, x):
    x = self.encoder(x)
    x = L1Penalty.apply(x, 0.1)
    x = self.decoder(x)
    return x

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

Now, comes my forward pass of the auto encoder. So, what I do, is I have my first forward pass, which is from my 784 neurons, whatever is my input they go down to 400 neurons. Now, at 400 neurons, I will be putting out my sparsity for the first time. So, and I want to have a sparsity in position of 0.1 or 10 percent of my weights are supposed to be 0 that is by definition what I am saying down over here. Now, this L1 penalty is

basically, a function, which has been defined in this earlier case over here and I have defined both my forward and my backward definitions for this particular new layer, which we are proposing for the first thing.

(Refer Slide Time: 08:04)

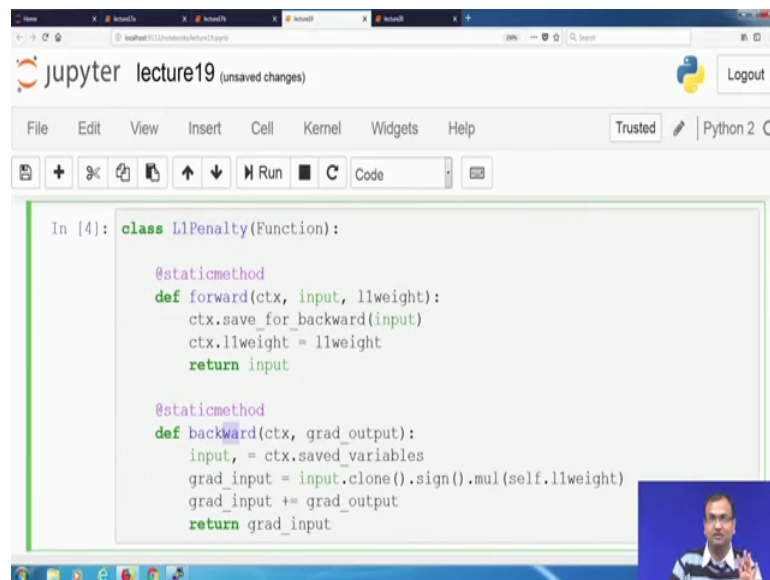


The screenshot shows a Jupyter Notebook window titled "lecture19 (unsaved changes)". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for file operations and execution. A message at the top states "GPU is available!". The main content area displays the following Python code:

```
In [4]: class L1Penalty(Function):  
  
    @staticmethod  
    def forward(ctx, input, llweight):  
        ctx.save_for_backward(input)  
        ctx.llweight = llweight  
        return input  
  
    @staticmethod  
    def backward(ctx, grad_output):
```

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

(Refer Slide Time: 08:07)

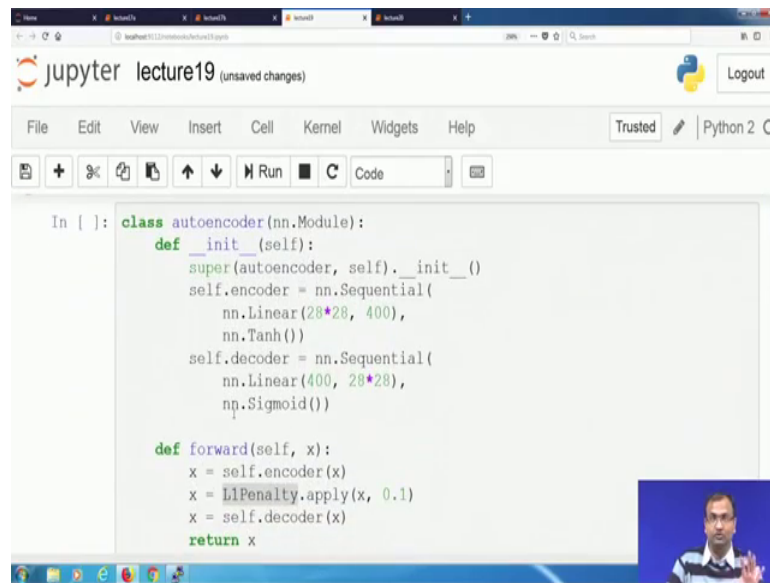


The screenshot shows the same Jupyter Notebook window, now displaying the complete implementation of the L1 Penalty layer, including the backward pass. The code is as follows:

```
In [4]: class L1Penalty(Function):  
  
    @staticmethod  
    def forward(ctx, input, llweight):  
        ctx.save_for_backward(input)  
        ctx.llweight = llweight  
        return input  
  
    @staticmethod  
    def backward(ctx, grad_output):  
        input, = ctx.saved_variables  
        grad_input = input.clone().sign().mul(self.llweight)  
        grad_input += grad_output  
        return grad_input
```

The same video inset of the man speaking is visible in the bottom right corner.

(Refer Slide Time: 08:11)

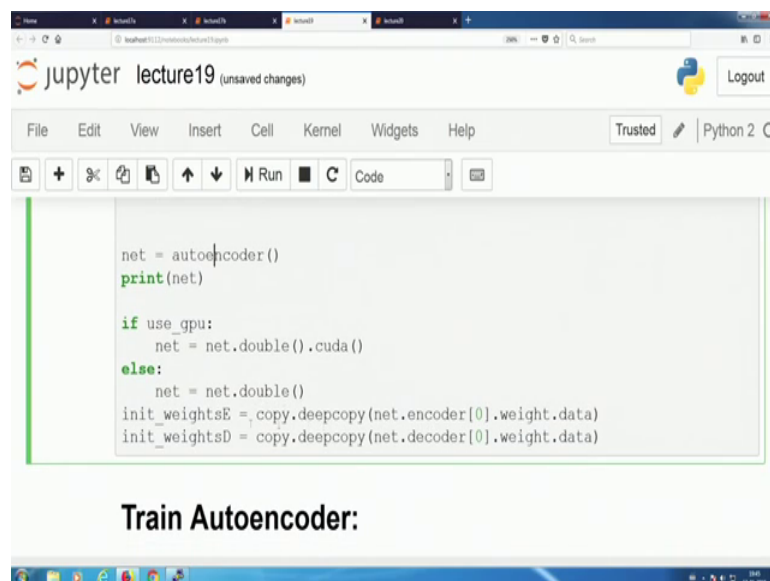


```
In [ ]: class autoencoder(nn.Module):
def __init__(self):
    super(autoencoder, self).__init__()
    self.encoder = nn.Sequential(
        nn.Linear(28*28, 400),
        nn.Tanh())
    self.decoder = nn.Sequential(
        nn.Linear(400, 28*28),
        nn.Sigmoid())

def forward(self, x):
    x = self.encoder(x)
    x = L1Penalty.apply(x, 0.1)
    x = self.decoder(x)
    return x
```

So, you can actually use this as a standard definition for proposing any newer kind of a layer architecture as well. So, maybe you come up with your own neural layer architectures at any point of time and you can actually use a separate function for defining that, we do the same thing in order to do this L1 penalty loss over here and then, whatever is the output which comes down over here, which is now sparse in some sense and you have a decoder through which it goes down. Next, I can just print my auto encoder. And then if I have GPU available, I would just be using GPU to do it.

(Refer Slide Time: 08:40)



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

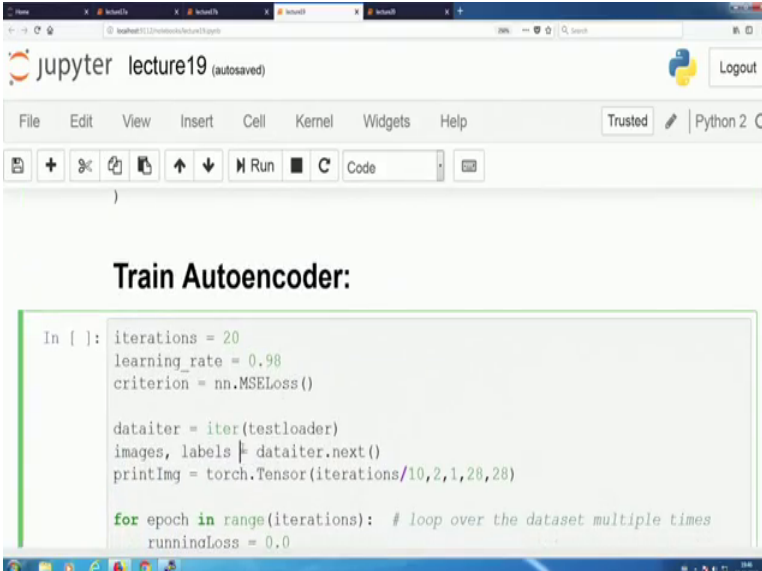
if use_gpu:
    net = net.double().cuda()
else:
    net = net.double()
init_weightsE = copy.deepcopy(net.encoder[0].weight.data)
init_weightsD = copy.deepcopy(net.decoder[0].weight.data)
```

Train Autoencoder:

Now, from there what I do is I use two temporary variables and these are basically, to copy down my weights, the whole rationale for copying down these two weights are like before I start my whole training process. I would actually like to see like how they are initialized and how they look like and what happened after my training with this kind of a sparsity coming now. So, let us run this part and. So, you print down your network architecture. Now, if you look into your network architecture at no point of time, this L1s penalty loss is coming down over here, because this is just a function, which has been defined for the forward pass.

So, you do not have a modification over the architecture, but you are just modifying the way your data would be passing in the forward pass of your network.

(Refer Slide Time: 09:32)



```
In [ ]: iterations = 20
learning_rate = 0.98
criterion = nn.MSELoss()

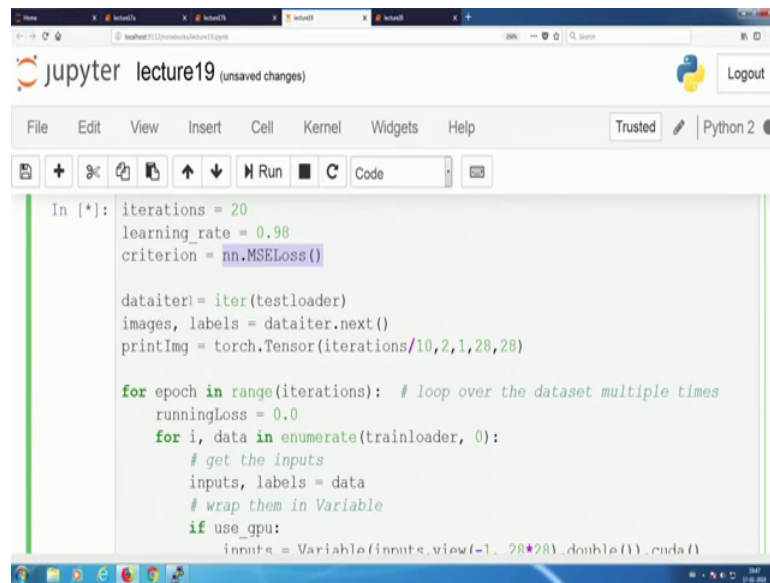
dataiter = iter(testloader)
images, labels = dataiter.next()
printImg = torch.Tensor(iterations/10, 2, 1, 28, 28)

for epoch in range(iterations): # loop over the dataset multiple times
    runningLoss = 0.0
```

Now, my next part is to go on and train this autoencoder. Now, based on our experience we did see that over 20 iterations, it does run out. Now, again from our prior experiences with the earlier examples, which were doing it, does take a significant amount of time to finish it out. So, a few seconds, I will just set this running, while I actually start explaining you, what goes down over here.

So, we decided to go with 20 epochs of learning and a learning rate of 0.98 and since I am training just a simple autoencoder architecture over here, which is just trying to find out the reconstruction loss and trying to minimize it; so, my criterion, my loss function over here becomes MSE loss function.

(Refer Slide Time: 10:13)



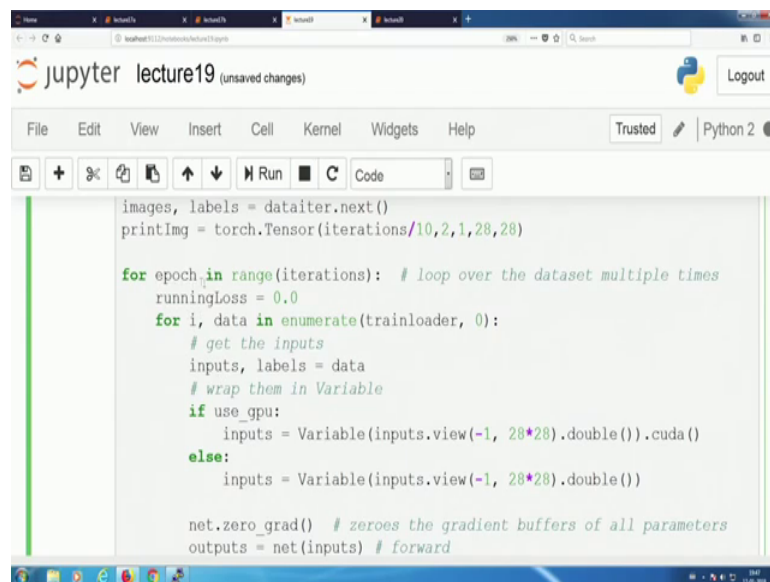
```
In [*]: iterations = 20
learning_rate = 0.98
criterion = nn.MSELoss()

dataiter = iter(testloader)
images, labels = dataiter.next()
printImg = torch.Tensor(iterations/10, 2, 1, 28, 28)

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()
```

Now, what I do over here is that, I would like to have like these data points over there also stored for my like a later on point of viewing over there.

(Refer Slide Time: 10:28)



```
images, labels = dataiter.next()
printImg = torch.Tensor(iterations/10, 2, 1, 28, 28)

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 parameters
    outputs = net(inputs) # forward
```

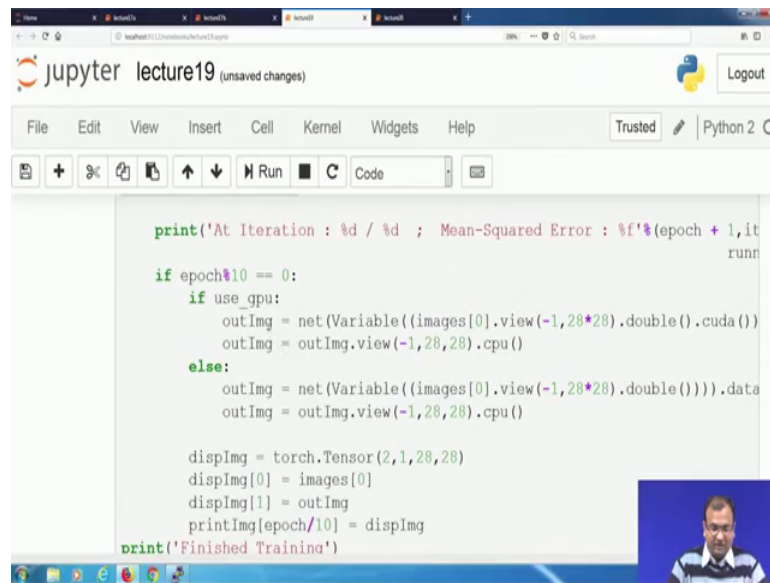
Now, what I start from that is, I will start my training, will just be something loop which keeps on rotating over the iterating over the number of epochs and that is my epochs counter, which is running down . So, now, as in with the earlier case, if you have a GPU then we convert it onto a GPU variable, a cuda variable which sends it over to the GPU, otherwise you just still have it residing on the C P U, then you do a 0 of all the residual

gradients present down over there and then comes down the first part, which is a forward passing.

So, there is one forward pass of the whole batch in one single epoch, after that what I do is, I find out what is the loss function. So, using my criterion which is a MSE loss function and this is just the difference between the input and output, because I was just trying to reconstruct whatever was given as input over there. Now, I need to do a backward or the derivative of my loss ∇ of my loss function, a ∇ of J and here, I get my derivative of the loss coming down and then is my vanilla gradient descent or the simple strain plain method of gradient descent, which I am using if you remember from my last lecture, where I was explaining you the theory of Sparse Autoencoders, you do not have any specific kind of a change coming down in terms of your learning rule.

So, the only thing is that we add a certain term onto the cost function and how it gets added is, because you have this forward pass and the backward pass both of them defined and in the backward pass is where I am adding this extra part on the gradient over there and nothing more. So, what this introduces necessarily is that my gradient is what comes down from the backward operator over the criterion function and backward operator over the network. Now, that I have my gradient of my input available, I would just be adding this extra gradient, which is the loss computed in case of when I have my sparsity incorporated over there. So, that is just the addition of the loss, which I do over there and that is what includes my equations, whatever we had done over there .

(Refer Slide Time: 12:37)



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

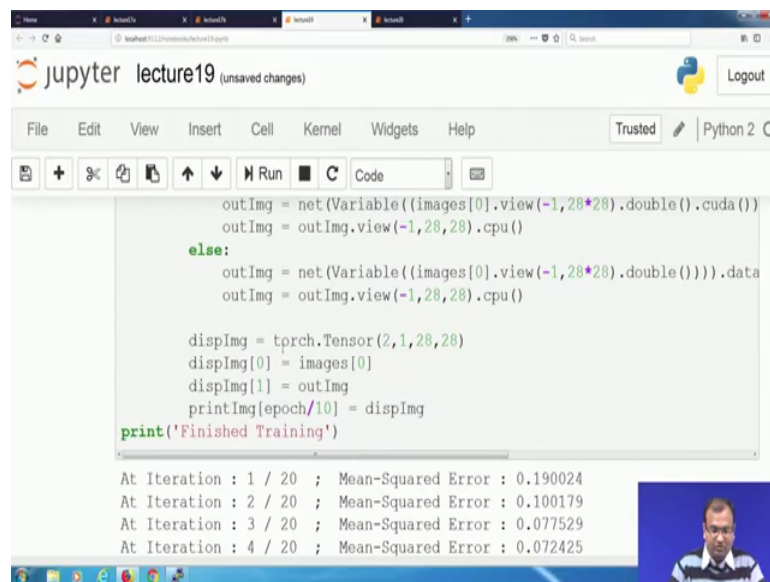
if epoch%10 == 0:
    if use_gpu:
        outImg = net(Variable((images[0].view(-1,28*28).double()).cuda()))
        outImg = outImg.view(-1,28,28).cpu()
    else:
        outImg = net(Variable((images[0].view(-1,28*28).double()))).data
        outImg = outImg.view(-1,28,28).cpu()

dispImg = torch.Tensor(2,1,28,28)
dispImg[0] = images[0]
dispImg[1] = outImg
printImg[epoch/10] = dispImg

print('Finished Training')
```

Now, from there what I do over here is that, this is just a simple trick to display down your outputs after a certain given point of time and then nothing beyond it.

(Refer Slide Time: 12:49)



```
outImg = net(Variable((images[0].view(-1,28*28).double()).cuda()))
outImg = outImg.view(-1,28,28).cpu()

else:
    outImg = net(Variable((images[0].view(-1,28*28).double()))).data
    outImg = outImg.view(-1,28,28).cpu()

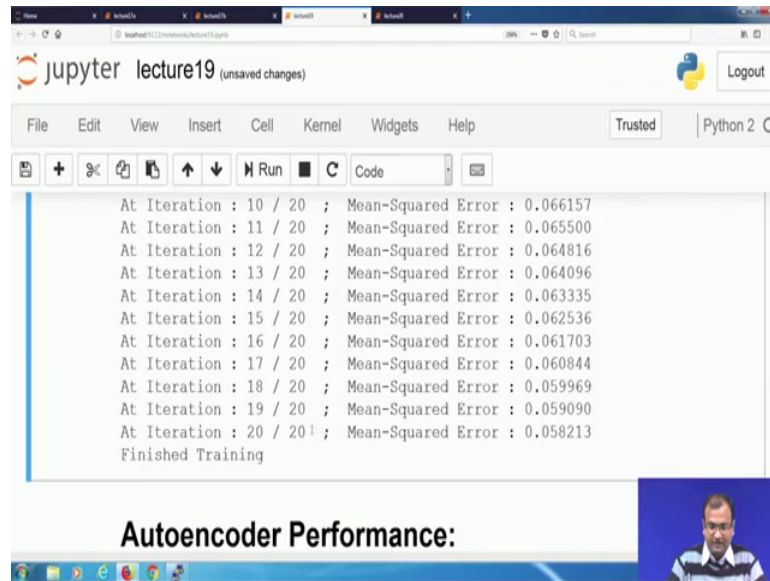
dispImg = torch.Tensor(2,1,28,28)
dispImg[0] = images[0]
dispImg[1] = outImg
printImg[epoch/10] = dispImg

print('Finished Training')
```

At Iteration : 1 / 20 ; Mean-Squared Error : 0.190024
At Iteration : 2 / 20 ; Mean-Squared Error : 0.100179
At Iteration : 3 / 20 ; Mean-Squared Error : 0.077529
At Iteration : 4 / 20 ; Mean-Squared Error : 0.072425

So, let us just see how it has gone down. So, it was good that you did finish training the whole thing and over with 20 epochs we have somewhere achieved a loss of 0.058.

(Refer Slide Time: 12:55)



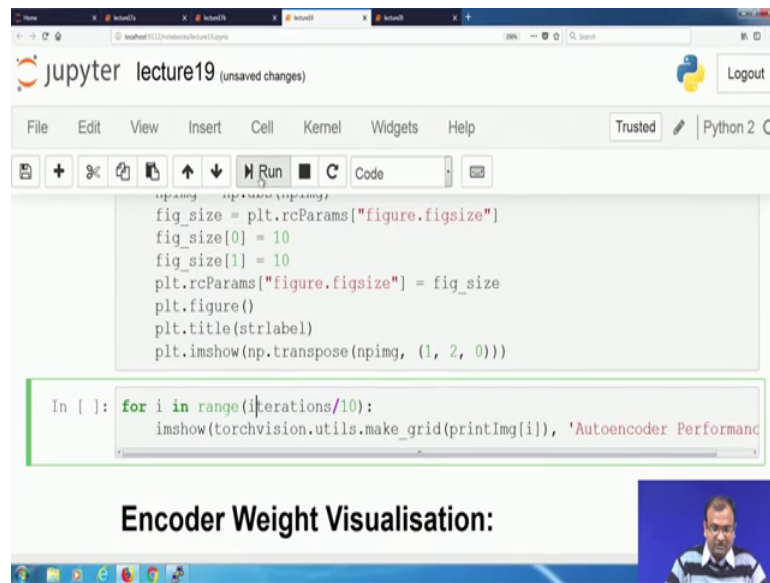
```
At Iteration : 10 / 20 ; Mean-Squared Error : 0.066157
At Iteration : 11 / 20 ; Mean-Squared Error : 0.065500
At Iteration : 12 / 20 ; Mean-Squared Error : 0.064816
At Iteration : 13 / 20 ; Mean-Squared Error : 0.064096
At Iteration : 14 / 20 ; Mean-Squared Error : 0.063335
At Iteration : 15 / 20 ; Mean-Squared Error : 0.062536
At Iteration : 16 / 20 ; Mean-Squared Error : 0.061703
At Iteration : 17 / 20 ; Mean-Squared Error : 0.060844
At Iteration : 18 / 20 ; Mean-Squared Error : 0.059969
At Iteration : 19 / 20 ; Mean-Squared Error : 0.059090
At Iteration : 20 / 20 ; Mean-Squared Error : 0.058213
Finished Training
```

Autoencoder Performance:

Now, if you remember in your earlier MNIST case with training epochs, you were coming down to somewhere around 0.61, if I clearly remember whereas, here with using a sparsity.

We have not been able to come down to that level, but after 20 epochs, we did go down and clearly from the earlier example where you had two different data set which are of the same empirical size in terms of the number of pixels and almost in the same dynamic range of gray values, you also did recall that it is not necessary that you would come down to the same error limits for both of them. So, here also it is something of the same sort. Now, here what I try to do is basically, in order to look down into what is the performance of my auto encoder over here, I would just like to check out

(Refer Slide Time: 13:46)



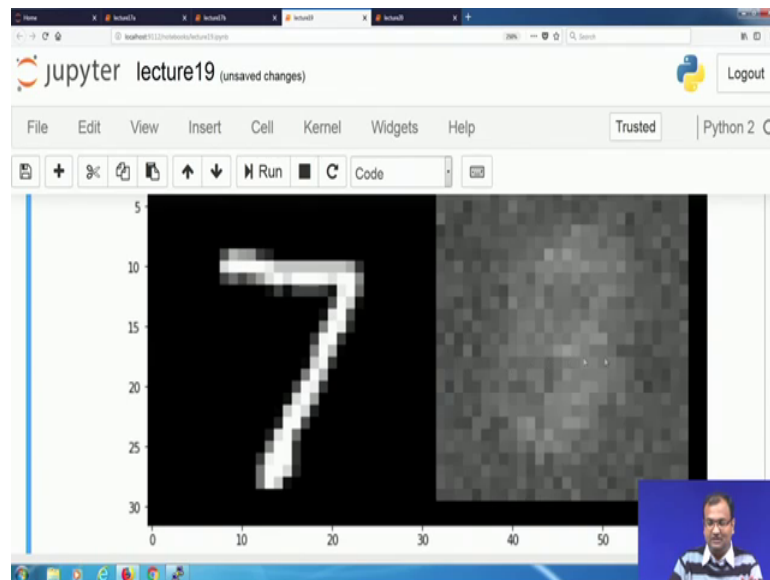
```
fig_size = plt.rcParams["figure.figsize"]
fig_size[0] = 10
fig_size[1] = 10
plt.rcParams["figure.figsize"] = fig_size
plt.figure()
plt.title(strlabel)
plt.imshow(np.transpose(npimg, (1, 2, 0)))

In [ ]: for i in range(iterations/10):
        imshow(torchvision.utils.make_grid(printImg[i]), 'Autoencoder Performanc
```

Encoder Weight Visualisation:

So, let us look into the performance of the autoencoder. So, initially when it was trained on like at the start of the training, which is the zeroth epoch. So, you have all rates, which are randomized and taken down from some sort of a Gaussian distribution and plot it down over there. So, with that if this is my input.

(Refer Slide Time: 14:03)



Then this is something which comes down as my output over there and you cannot necessarily make out anything out of it whereas, after training, this is something which I get it comes down as if a consolidated blob, not necessarily. So, noisy it is a blob which

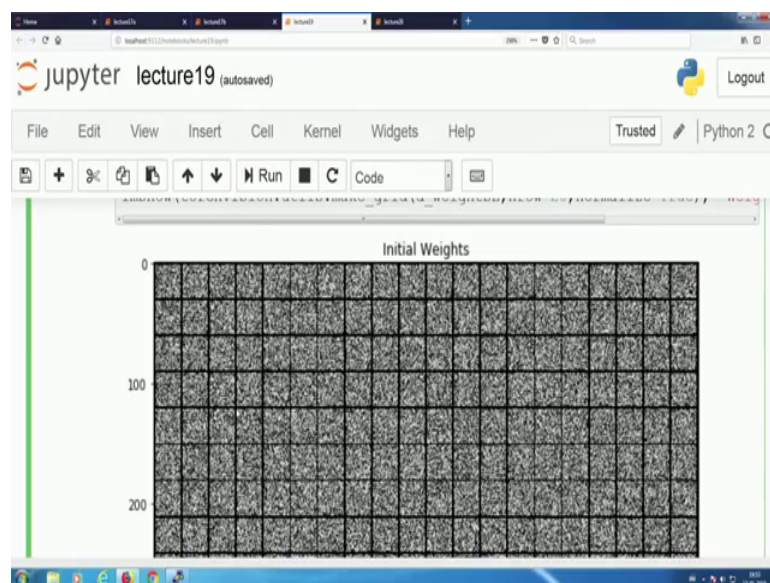
comes out, it looks like a mixture of maybe 7398, a lot of things and some decent probability around being a 7.

So, provided that you are training it for really longer period of time or you are using some different kind of training method, instead of the gradient descent, you use some of the other optimization techniques, which I deleted on point of time, when we would be using. We study about them, you will be knowing that using some other techniques, it might come down to a better point, but as of now let us freeze at this point. So, this comes down one where you have achieved some sort of fidelity in the whole reconstruction. Now, I would like to look into my visualization of these weights.

Now, you remember that what we had done earlier, somewhere over here is we had actually copied down all the weights, while the network was defined and these are all the random initializations, which my network gets randomly initialized. So, I actually bring those weights over here, one is I can have my trained weights, which I can extract actually after the training over there and then I have my initialized initialization weights, which were before the whole process were started that is what is given down over there.

Now, when I write down this capital E that is actually the weights of the encoder, when I write down those as d then that is weights on the decoder, which comes down. So, here let us look into these initial weights.

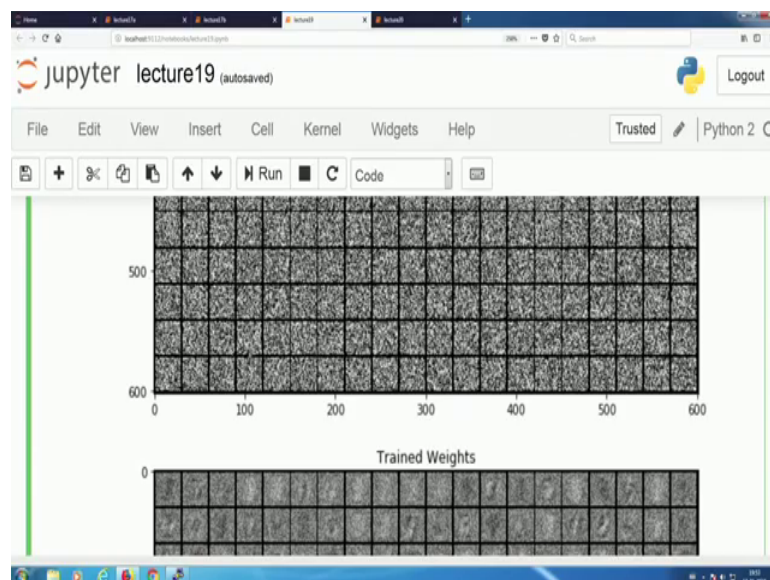
(Refer Slide Time: 12:45)



So, now, this is what it was like at the start of the whole problem and you remember that you had 28 cross 28, which were connected down to 400 neurons over there. So, technically; that means, that I have 28 cross 28 or 700 and 784 weights, which connect down to 1 single neuron and this is that 28 cross 28 weight matrix, which you see and typically you like, if you look across the spacings.

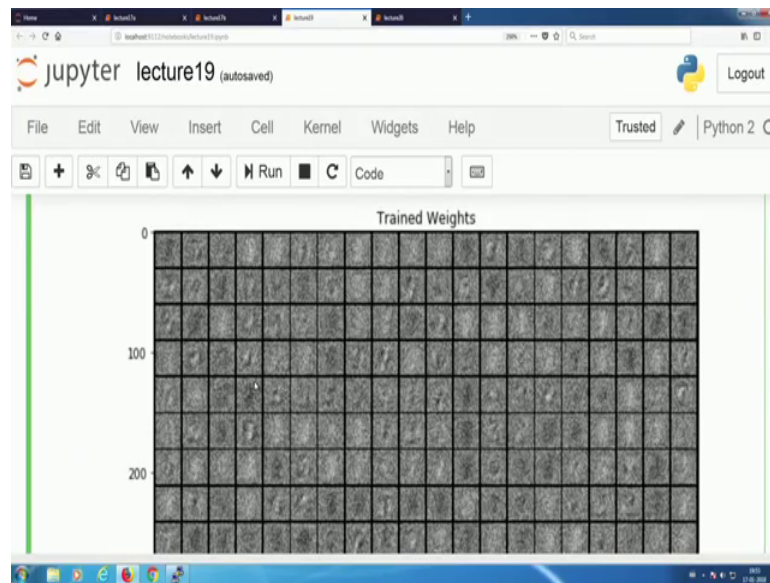
So, you have such 20 cross 20 tiles and you can just count it 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 such ones and on this side also you have 20 over there. So, in total that should be 400 such small tiles present over there.

(Refer Slide Time: 16:31)



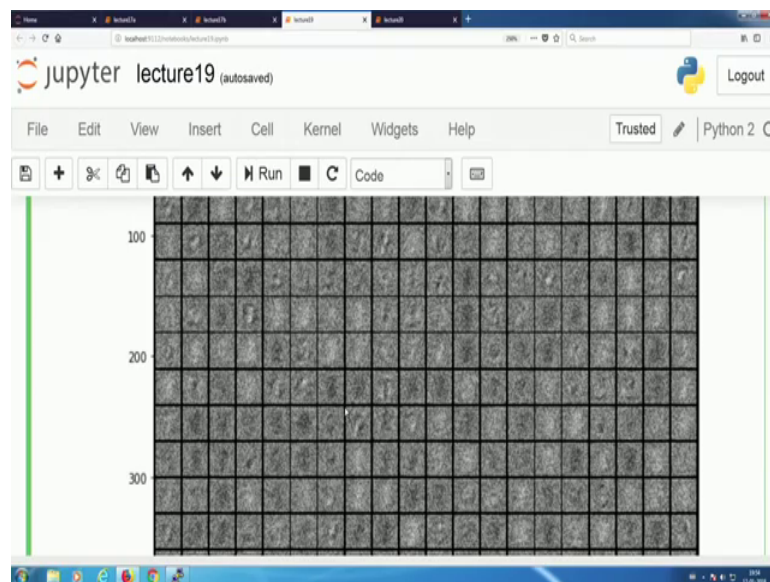
Now, once the whole training is process, training process is over, after 20 iterations over, there this is what these weights start looking like.

(Refer Slide Time: 16:37)



And one thing what you can see is that they have changed not necessarily, that they have come down to a convergent point or something.

(Refer Slide Time: 16:50)



But there has been subtle changes coming down and some of them look like as if they are taking structures like 0 1 7 something over here, something rough. So, provided you train them over a really long period of time maybe, set it down for 200 epochs and go and have your cup of coffee, tea or finish off your dinner, lunch, whatever you are doing and

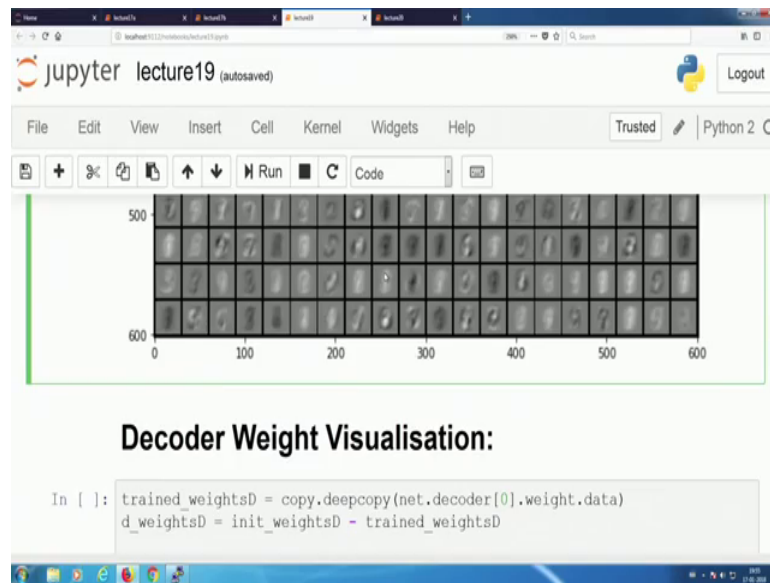
then come back in 30 minutes or. So, in most of your systems, it would be trained on for 200 epochs by then.

So, deep learning does not take that much of time as a lot of people are scared about it. So, then you would be able to actually see down the changes, which come down the next part. What we looked into is; what is the actual way of weights updates, which have happened. Now, this is a very crucial point, because what we have done over here is just subtracted the original random weights from the current version of the weights are after training and these indicates those particular locations, where there has been changes. Now, if you look into this changes over here, these changes are something where you would be able to see down numbers.

Now, typically most of the updates have been in the center region and that is quite obvious, because whatever images of these numbers 100 and digits, you had, they had most of these actual line pandas, which were at the center point, they were not located at corner places or anywhere and that was the reason why you did not have any updates coming down in the corner, but most of the updates were from the center region also. You see that they are somehow congruent to the numbers where it was trying to come down and this is the first belief that if you are training it down over a longer period of time. So, this would keep on converging over and over again, this would be the sort of a guidance principle of how the weights are getting updated and eventually the final thing would start looking something like this where each of these bits are tuned down to a particular number.

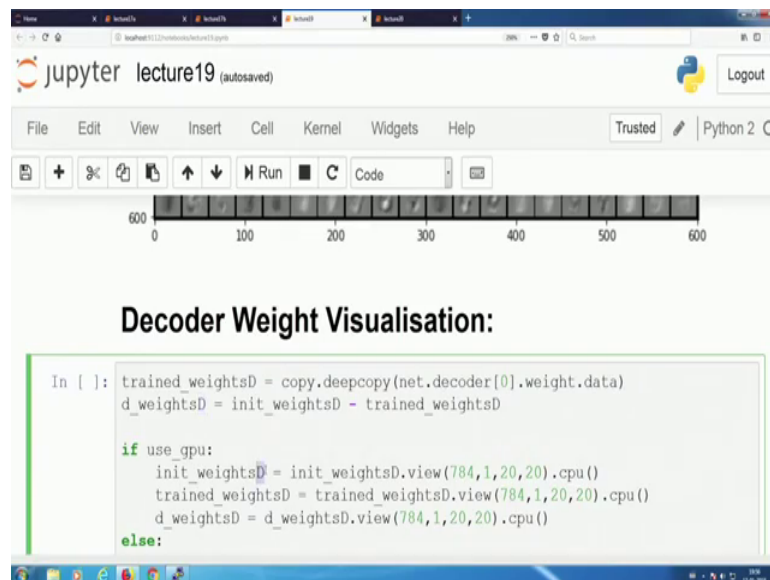
So, this is what I leave on to you guys to do out of your own interest. So, this is about how all of these things are connected from my encoder to decoder.

(Refer Slide Time: 18:38)



This is one part of it, the next part is to look into the weight visualization for my decoder and that is where my this capital D comes from. So, all my weights from the decoder. So, initially I had already stored down my weights from the decoder.

(Refer Slide Time: 18:52)



So, that was something, if we go up over here, when I had defined it. So, I have my weights of the decoder as well which I had stored it down and then after my whole training is over, I again pull out those decoder weights from my network.

Now, my whole objective is to repeat the same thing and then start looking into the decoder weights as well now, this is something which is present on the initial part of it and clearly if you look into these patches. So, I have 784 neurons which were connected to 400 neurons in my encoder. Now, I have 400 neurons in my encoder as output of my encoder, which I connected down to 784 neurons on my decoder.

So, now what I would have, is that there is a small matrix of 20 cross 20 or 400 bits. So, that is, this small matrix over here and then since I am connecting it to seven eighty four. So, that should be 784 such small patches or matrices of weights. So, it is a 28 cross 28 array, which should be there, let us just count it 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 and countdown on the vertical direction, you would see such 28 tiles again.

Now, over here is where I see my weights, which are after my training. Now, it does not look much of a difference, technically saying like most of you would say it is as noisy as it was in the earlier case as well, yes it is noisy, but then the intensity of the noise has somewhat decreased. One thing you need to keep in mind is you remember that these gray values, what we had seen in the earlier, cases also I was explaining that they are what are zero values and all the blacks are negative values, all the whites are positive values which are other and this is normalized into it is dynamic range and that is why.

We do not have that, this is just for visualization purpose, you cannot technically infer out on the values just looking over here and then if you look into these weight updates over there you do not necessarily see to, I tend to see any patterns, there are some wavy things and some really discreet way; however, there is one interesting point, which you need to look into over here, when you see in this one most of the values were either highly negative and highly positive, here a majority of these values tend to become down as zeros and that is the point of this L1 sparsity, which comes down.

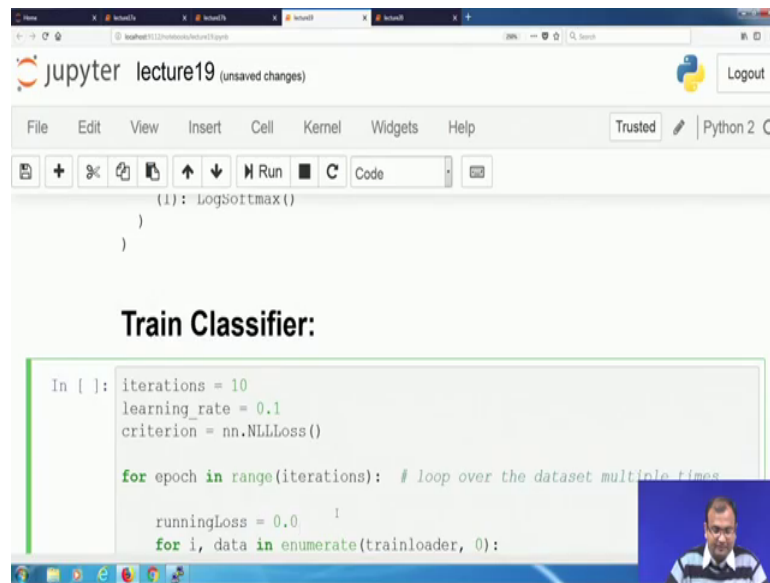
Because now, that you have all of these gray values, which you see they are zero value numbers, which comes from and this is something which gets imposed from here, while you had just high positive and high negative values to hear. When majority of the values are actually zero valued on the weight matrix over there. So, this whole thing comes down, because you have an L1 penalty, you had imposed over there and that was possible, because a lot of these weights you see over here, they look similar to each

other, say this weight looks very similar to this weight. It looks pretty similar to this weight, it looks pretty similar this weight, it looks pretty similar to these weights. Now, the moment is you will be able to get down whereas, this way it looks very dissimilar.

So, you will have very less number of such unique weights, which are available and then since whatever you are doing over here, is just a combination of weighted summation of the outputs, coming down from one of these layers. Now, most of these neurons, they would actually have a similar kind of an output, which comes down and you can pretty much do away with most of the neurons and keep done 1 or 2 neurons over there and still get an output coming down and that was the reason the whole rationale, why over here? Do you see a lot of these bits, actually go down to a **zero** value and that **is** really helpful in case of dealing with sparsity.

Now, you would see that the weight updates, which happen over here, they are also quite near to zero value, but there are certain high and low changes coming down, because you had to bring down those high positive and high negative values on to quite closer down to actual zero values coming out. So, that was all about these visualizations and finally, what we do over here is the simple old trick, which was I have trained down my whole encoder with L1 sparsity over there and now, I would like to modify this autoencoder, in order to form a classifier. So, what I need to do is I necessarily need to remove out my last decoder layer and then add my classifier module which connects from 400 neurons to 10 neurons.

(Refer Slide Time: 23:12)



The screenshot shows a Jupyter Notebook window titled "lecture19 (unsaved changes)". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for file operations and execution. The code cell contains the following Python code:

```
(1): LogSoftmax()
)
)

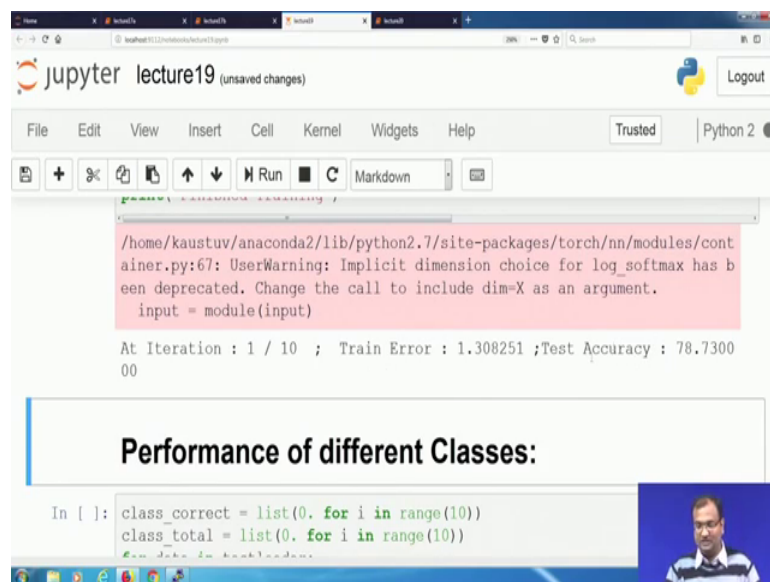
Train Classifier:

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):
```

And then I can start my classification trainer and this classification trainer over here, trains over this standard 10 iterations, using negative log likelihood loss function, as we had done in the earlier case. So, let us just set this running over here.

(Refer Slide Time: 23:28)



The screenshot shows the same Jupyter Notebook window. The code cell is now in "Markdown" mode, displaying the output of the training process. The output includes a warning message and performance metrics:

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

At Iteration : 1 / 10 ; Train Error : 1.308251 ; Test Accuracy : 78.730000
```

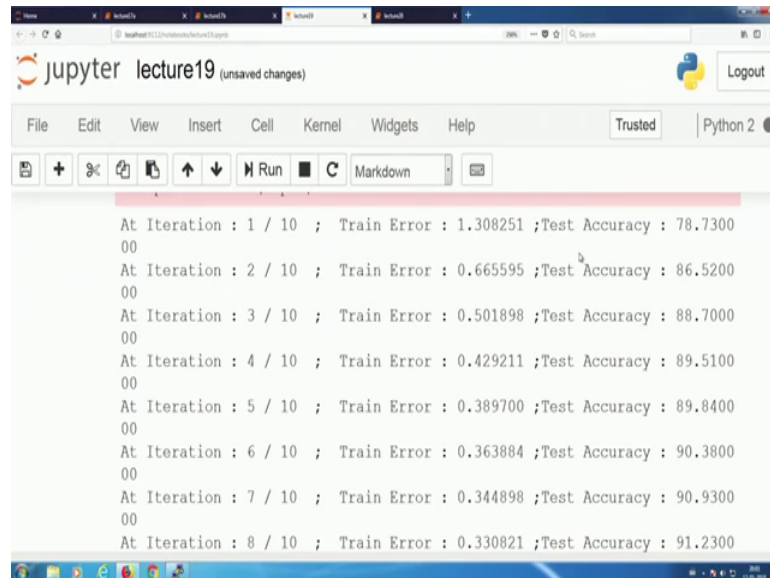
Performance of different Classes:

```
In [ ]: class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))
```

Now as I see over here, you would see that it starts with a starting accuracy of 78 percent. Now, if this is, if you remember it from the earlier one, this is really high those earlier case, we started somewhere around 66 percent on the starting accuracy whereas,

here we have started down with a starting accuracy itself of 78 percent. Now, one question you might ask is why is it? So, why did we start with a higher accuracy?

(Refer Slide Time: 23:52)

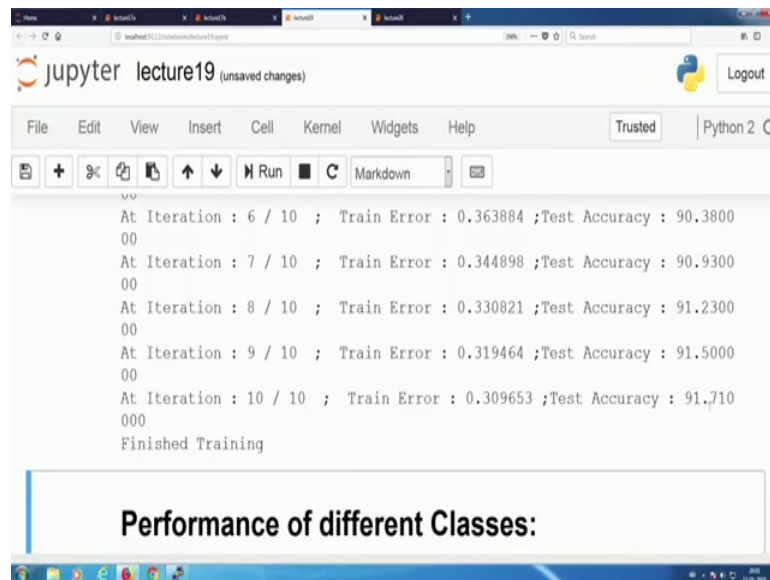


```
At Iteration : 1 / 10 ; Train Error : 1.308251 ;Test Accuracy : 78.7300
00
At Iteration : 2 / 10 ; Train Error : 0.665595 ;Test Accuracy : 86.5200
00
At Iteration : 3 / 10 ; Train Error : 0.501898 ;Test Accuracy : 88.7000
00
At Iteration : 4 / 10 ; Train Error : 0.429211 ;Test Accuracy : 89.5100
00
At Iteration : 5 / 10 ; Train Error : 0.389700 ;Test Accuracy : 89.8400
00
At Iteration : 6 / 10 ; Train Error : 0.363884 ;Test Accuracy : 90.3800
00
At Iteration : 7 / 10 ; Train Error : 0.344898 ;Test Accuracy : 90.9300
00
At Iteration : 8 / 10 ; Train Error : 0.330821 ;Test Accuracy : 91.2300
```

One thing you need to keep in mind is that these kind of networks, the moment we started introducing this sparsity, there was one thing in order for the network to be sparse. You need the earlier part before sparsity to be over, complete, otherwise the network cannot be sparse. Now the moment you are having, an over complete representation. In the earlier case, you see that you tend to have similar representations grouped down. So, you are now coming down to dominant group of representations and; that means, that the noisy representations are getting ruled out in the whole process.

So, as we keep on getting noisy representations ruled out, which means that features which are really irrelevant to this particular problem of classification. They do not come into play and for that reason; we start with a very higher accuracy.

(Refer Slide Time: 24:32)



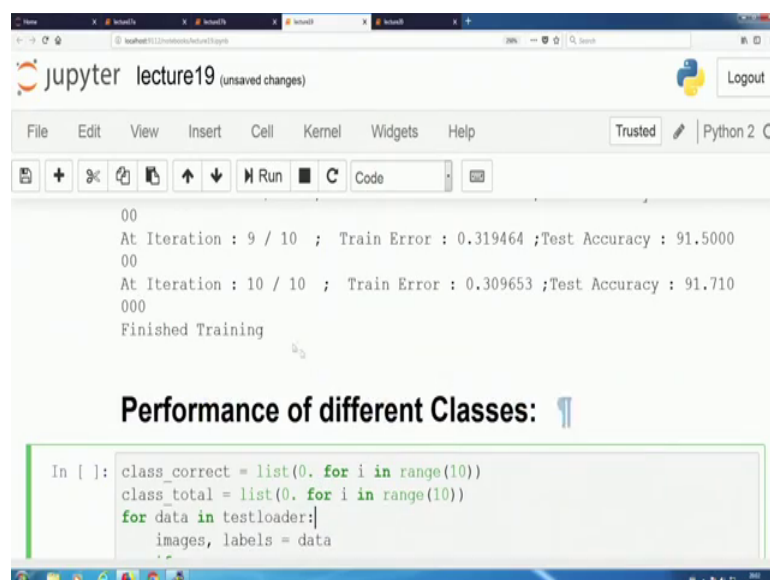
The screenshot shows a Jupyter Notebook interface with the title 'lecture19 (unsaved changes)'. The notebook contains a code cell with the following output:

```
At Iteration : 6 / 10 ; Train Error : 0.363884 ;Test Accuracy : 90.3800
00
At Iteration : 7 / 10 ; Train Error : 0.344898 ;Test Accuracy : 90.9300
00
At Iteration : 8 / 10 ; Train Error : 0.330821 ;Test Accuracy : 91.2300
00
At Iteration : 9 / 10 ; Train Error : 0.319464 ;Test Accuracy : 91.5000
00
At Iteration : 10 / 10 ; Train Error : 0.309653 ;Test Accuracy : 91.710
000
Finished Training
```

Below the code cell, there is a text box with the heading 'Performance of different Classes:'.

However, the final accuracy is almost of the same plot. It does not change, because in the earlier case, you had about 92 percent, we trained it over and over for 20 epochs or you went down to 92 and here, you might not necessarily get done, but you are starting estimate, in this case is a much better estimate than you have in the earlier case without having the sparsity.

(Refer Slide Time: 24:48).



The screenshot shows a Jupyter Notebook interface with the title 'lecture19 (unsaved changes)'. The notebook contains a code cell with the following output:

```
At Iteration : 9 / 10 ; Train Error : 0.319464 ;Test Accuracy : 91.5000
00
At Iteration : 10 / 10 ; Train Error : 0.309653 ;Test Accuracy : 91.710
000
Finished Training
```

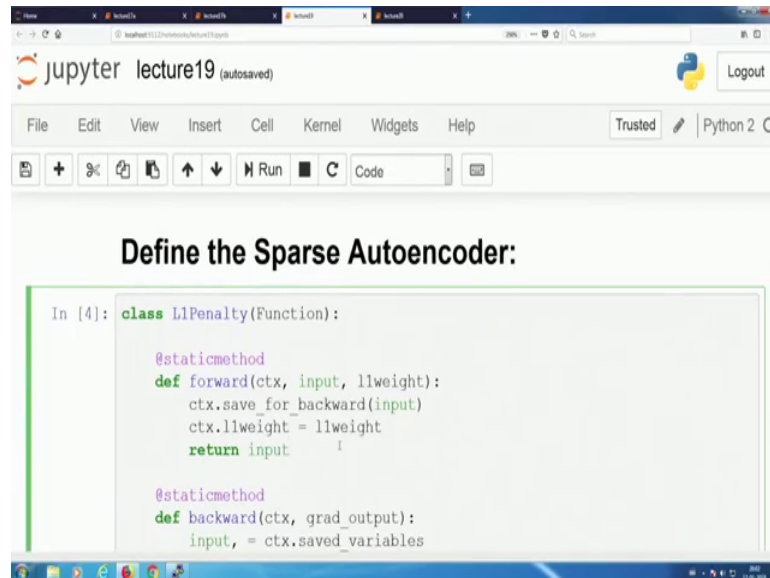
Below the code cell, there is a text box with the heading 'Performance of different Classes:'. Below this, there is a code cell with the following code:

```
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
```

So, if we look into each of these classes and how much they come down you still see that for class 5, in the earlier case it was about 55 percent, accurate 85 percent accurate and

here we are barely 84.7, which is almost the same amount of accuracy there has not been much of a change; however, the features which it has been learning over here are something which are more congruent to be representative of the numbers which are handwritten. So, with this we come to an end of lecture nineteen on our whole aspect of trying to train down an actual autoencoder with sparse sparsity included over there.

(Refer Slide Time: 25:38)



```
In [4]: class L1Penalty(Function):  
  
    @staticmethod  
    def forward(ctx, input, llweight):  
        ctx.save_for_backward(input)  
        ctx.llweight = llweight  
        return input  
  
    @staticmethod  
    def backward(ctx, grad_output):  
        input, = ctx.saved_variables
```

And with this concept we have shown you for the first time about how to actually start defining your own kind of layers as well in including, if you want to just have some customized cost functions coming down. So, in the next lecture we would be covering down on top of this one which is to come down with the next aspect of a denoising autoencoder and how does a denoising autoencoder help you to learn better weights as well as really clean down the noise from any of these results. So, for then stay tuned and.

Thanks.