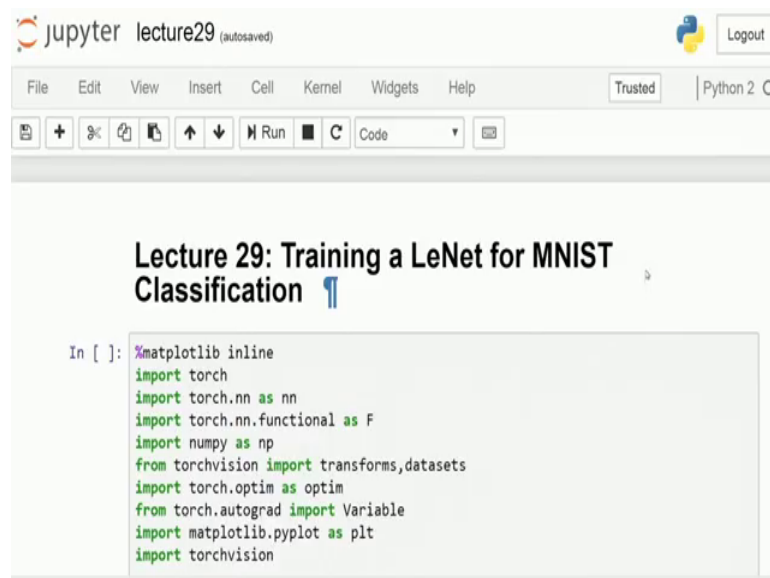


Deep Learning for Visual Computing
Prof. Debdoot Sheet
Department of Electrical Engineering
Indian Institute of Technology, Kharagpur

Lecture – 29
Training a LeNet for MNIST Classification

Welcome. So, in the last lecture we had done on how to actually define a LeNet the very basic model.

(Refer Slide Time: 00:25)



```
lecture29 (autosaved)
File Edit View Insert Cell Kernel Widgets Help Trusted Python 2
+ %< > Run C Code
Lecture 29: Training a LeNet for MNIST Classification
In [ ]: %matplotlib inline
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
from torchvision import transforms, datasets
import torch.optim as optim
from torch.autograd import Variable
import matplotlib.pyplot as plt
import torchvision
```

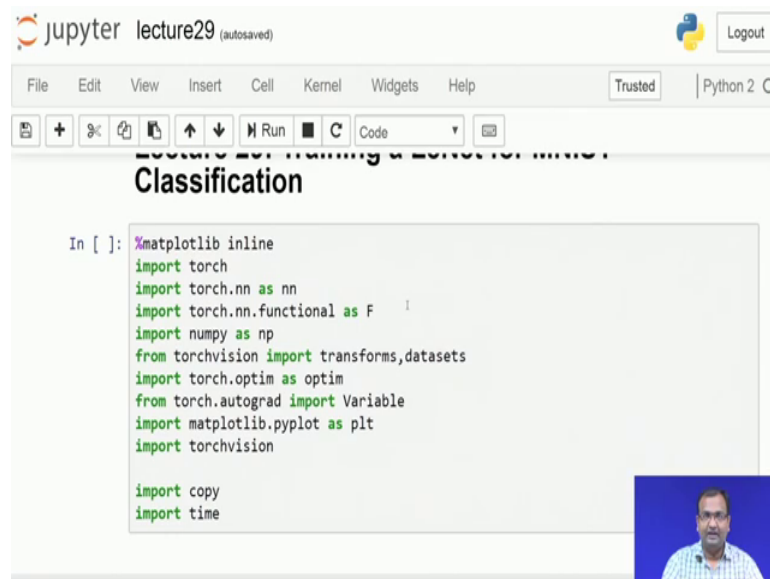
And then we had seen the different aspects of defining that LeNet and trying to use that for your MNIST here. So, the, to give you a recap of what we had done over there was primarily we defined the network and then, we over there went on to what really the kind of data which gets processed. So, as the images keep on getting processed over there since the size of the images and the size of the data keeps on changing and, varying across the network. So, we also looked into how it was very plus on top of that we had also seen. What is the total space of the weights? So, what is the total number of bits or parameters which you can keep on tuning over there.

And that was one major difference which came down from your fully connected layers because, there the output of a particular layer is the same as the total number of neurons in that layer whereas, when you are looking into convolutional you did see that the output is not dependent on the total number of neurons over there because, that the that

size of the neuron layer which is just within your weights it is much smaller and, it just operates by convolution and that is that is where the change comes in.

Now, here what we are going to do is actually start training down with the LeNet for the MNIST part.

(Refer Slide Time: 01:33)



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

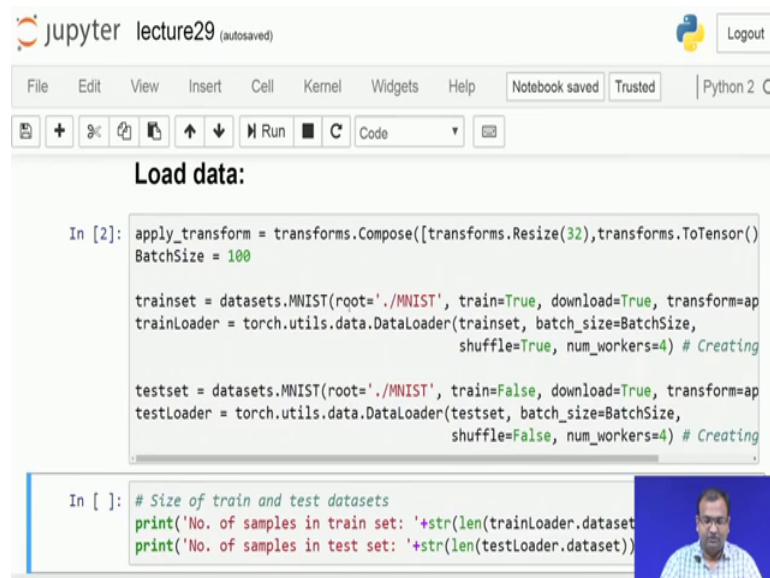
import copy
import time
```

So, in the earlier case whatever we had used down in our headers over, there was much lesser versions, we did not have any of these things from your optimizations to your so, we did define this as a variable because we want to do a feed forward, but then the optimization toolbox and the data sets and plot functions, were not present over there, we had not an a yet actually trained it out. So, this is where we start with the first training and this is to get you excited because it is your for cnn which will get trained.

So, till now you had trained down your fully connected networks and everything, in the last class you had just learnt how to write on your first cnn do those basic calculations, as to what is the total number of parameters which it has it has to learn what is the size of the output coming down from in. So, on what will be the dependencies between the input to the output over there and how that keeps on influencing it out.

So, without much delay let us get started with this one. So, the first part is just getting your environment variables and everything started down.

(Refer Slide Time: 02:30)



```
Load data:

In [2]: apply_transform = transforms.Compose([transforms.Resize(32),transforms.ToTensor())
BatchSize = 100

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

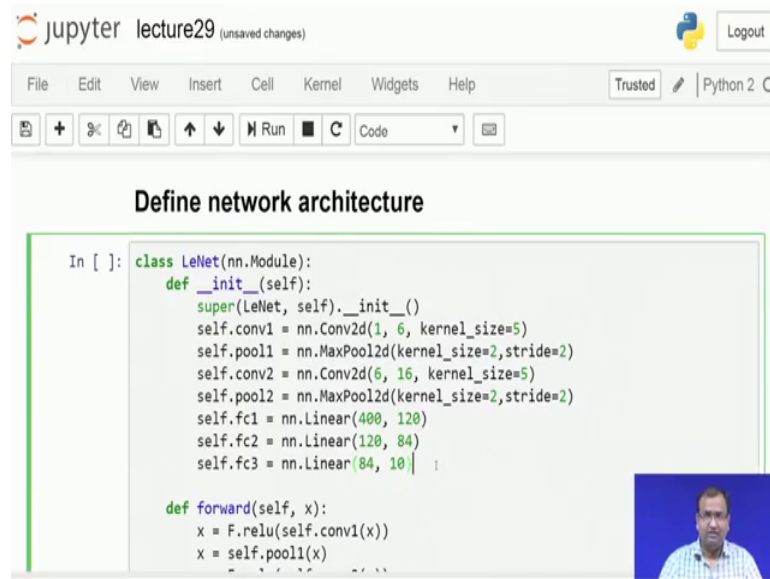
testset = datasets.MNIST(root='./MNIST', train=False, download=True, transform=ap
testLoader = torch.utils.data.DataLoader(testset, batch_size=BatchSize,
shuffle=False, num_workers=4) # Creating

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

The next part is to actually load your data and, we stick down to the same classical example of using an MNIST and what we are doing over here, is we load it on a batch size of 100 and as in the last lecture, we had seen that your batch size quite does not influence your number of tunable parameters over there the only relation your batch size has is to the size of the output which it keeps on produces. And the first index on any of this is your batch size and that does not change by the way throughout as it keeps on going. So, that that is how it works out.

So, let us do our data loader part over here, which is pretty similar to what we had done in our fully connected network. So, you just have train set and a train loader and, you also have a test set and a test load for both of them. So, for MNIST it is sixty thousand on the training and 1000 on the testing which is pretty standard and present ok.

(Refer Slide Time: 03:19)



The image shows a Jupyter Notebook window titled "lecture29 (unsaved changes)". The notebook is in "Code" view and displays the following Python code for defining a LeNet network architecture:

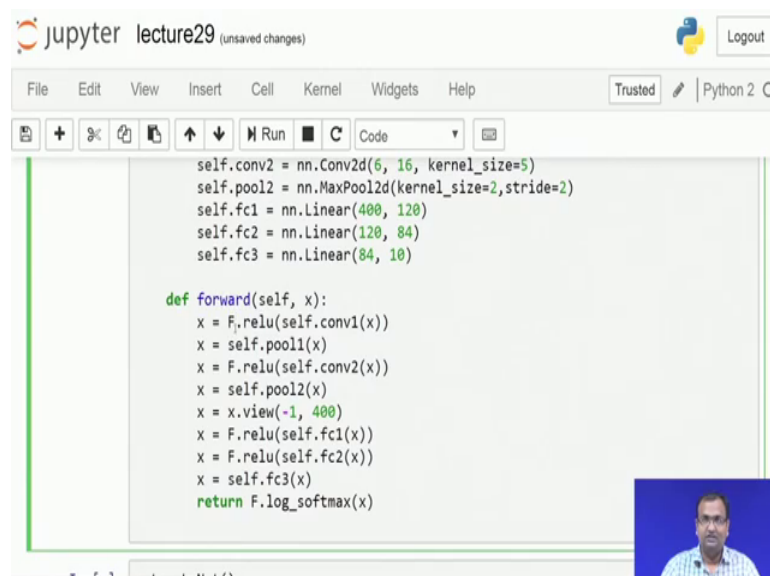
```
In [ ]: class LeNet(nn.Module):
def __init__(self):
super(LeNet, self).__init__()
self.conv1 = nn.Conv2d(1, 6, kernel_size=5)
self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
self.conv2 = nn.Conv2d(6, 16, kernel_size=5)
self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
self.fc1 = nn.Linear(400, 120)
self.fc2 = nn.Linear(120, 84)
self.fc3 = nn.Linear(84, 10)

def forward(self, x):
x = F.relu(self.conv1(x))
x = self.pool1(x)
```

Now comes down this part which is just a curious part of looking into it. So, I have all my training data set, all the 60000 of them and all the 10000 of my testing data set wrote it down and available to me.

Now, we come down to defining our network, now there is nothing new as such on the LeNet this is what we had done in the last lecture and, and a significant half an hour just discussing on all of the attributes. So, here it is it is still says the same the only difference which is brought down is in the forward function.

(Refer Slide Time: 03:47)



The image shows a Jupyter Notebook window titled "lecture29 (unsaved changes)". The notebook is in "Code" view and displays the following Python code for the forward function of a LeNet network architecture:

```
self.conv2 = nn.Conv2d(6, 16, kernel_size=5)
self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
self.fc1 = nn.Linear(400, 120)
self.fc2 = nn.Linear(120, 84)
self.fc3 = nn.Linear(84, 10)

def forward(self, x):
x = F.relu(self.conv1(x))
x = self.pool1(x)
x = F.relu(self.conv2(x))
x = self.pool2(x)
x = x.view(-1, 400)
x = F.relu(self.fc1(x))
x = F.relu(self.fc2(x))
x = self.fc3(x)
return F.log_softmax(x)
```

In the earlier case you did see that we had put down a lot of print statements to print down the actual size, of the output which is getting produced over there whereas, here we are just done away with all of those print statement it does not make sense, to keep on within every epoch as every single batch keeps on passing to it that you print it out, it will not play any role in down other than just for your satisfaction of seeing down, what is the size of output coming down over them.

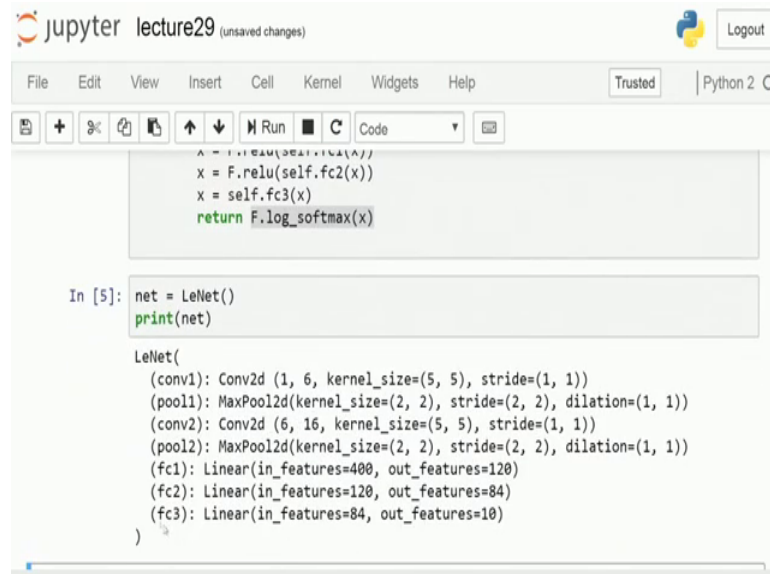
So, in the earlier case we had just done it for a debug purpose, you can consider this as more of a debug practice to really loop down and to what your pen and paper calculations lead to and whether, your network given down the similar kind of a data is actually producing a similar sized output. So, one is to always look into the size matching and the size match mismatch, which can possibly occur over there and try to minimize as much as this is possible.

In fact, there should not be any size mismatch otherwise you will segmentation fault over there, or a data mismatch fault. So, is that say if there are more number of data points coming down than the number of neurons are available, then it will always throw you an error. So, that is that is an issue. So, we just printed it over that here, you do not need to do any of those things and 1 extra thing which we add down is in the earlier cases we where in yet training it out. So, we did not have the, this basic non-linear transformation on the final output layer over there.

So, the final output is what has 10 different classes present over there and, then there is supposed to be a non-linearity over there. In the earlier case we did not mention this non-linearity because for the purpose of just data transformation through the network you do not need it.

So, here we go on to actually define this non-linearity and that is a logs of maximal linearity which we use and, in the same way as a LeNet for definition. So, once this definition is done. So, let us run it down. So, this is just to give you keep one thing clean to you that, if you are using that same module whatever is defined in the earlier lecture, then you would need to add down this extra line over here. So, that you have a functionally correct representation of the LeNet with this non-linearity in the last layer, which was not given mentioned in the last exercise which we had done over there. So, now let us go and bring down the whole network.

(Refer Slide Time: 06:00)



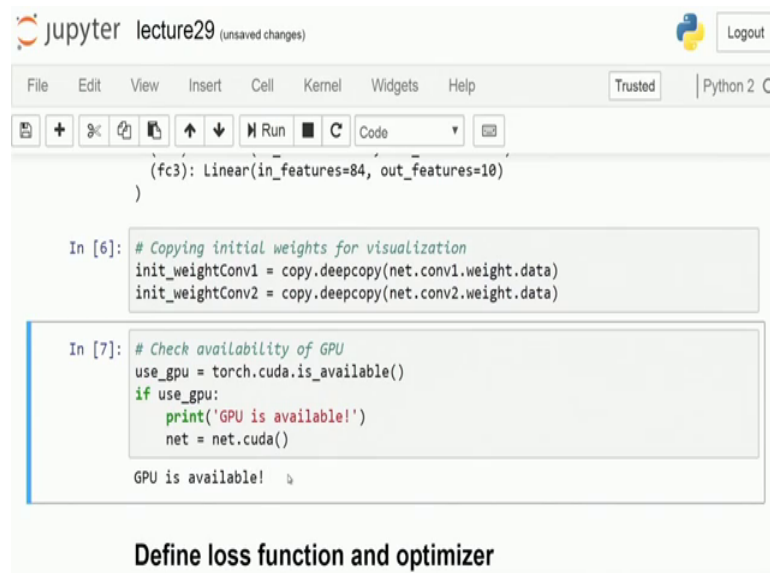
```
def forward(self, x):
    x = F.relu(self.fc2(x))
    x = self.fc3(x)
    return F.log_softmax(x)

In [5]: net = LeNet()
print(net)

LeNet(
  (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
  (pool1): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (pool2): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1))
  (fc1): Linear(in_features=400, out_features=120)
  (fc2): Linear(in_features=120, out_features=84)
  (fc3): Linear(in_features=84, out_features=10)
)
```

So, this is the network which comes down to me and as we had seen it looks pretty similar to what was there on the earlier case, though this extra non-linearity of a log softmax has come down over here, but then that is no change in the data architecture in any way and that does not make any change, or introduction coming down over here as well ok.

(Refer Slide Time: 06:17)



```
(fc3): Linear(in_features=84, out_features=10)
)

In [6]: # Copying initial weights for visualization
init_weightConv1 = copy.deepcopy(net.conv1.weight.data)
init_weightConv2 = copy.deepcopy(net.conv2.weight.data)

In [7]: # Check availability of GPU
use_gpu = torch.cuda.is_available()
if use_gpu:
    print('GPU is available!')
    net = net.cuda()

GPU is available!
```

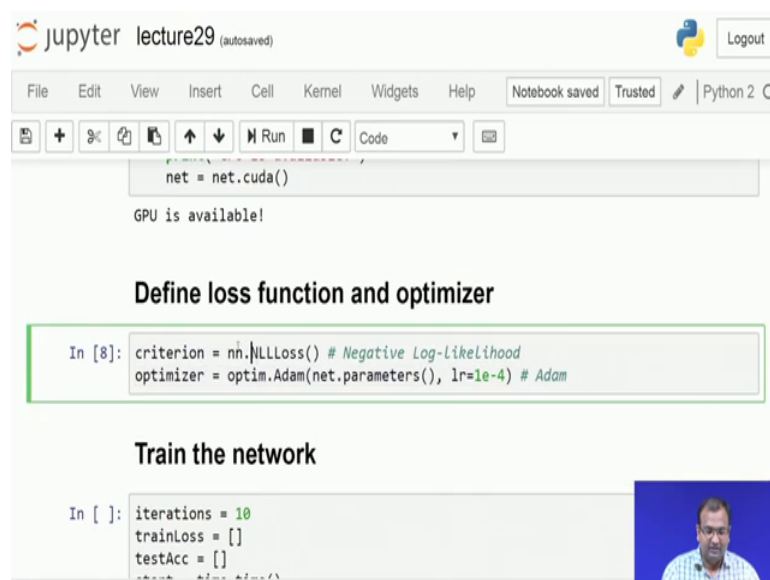
Define loss function and optimizer

Now, as in with the earlier case of where we were trying to visualize down weights and see down what is the amount of change which comes down in the weight. So, we try to

do a similar thing over here and, for that purpose we end up copying down all the weights in the first 2 convolution layers over there and, then just preserve it. So, 1 is you can copy down weights across the whole network and just keep it down, but instead of copying down all the weights and trying to visualize that im more of interested into just looking down what happens between these 2 convolution layers and, then what do they end up learning over there. So, they just to initialize and then copy down these two weights coming down over them.

The next part is you check out on your cuda and, then if your GPU is available. So, it it just shows that GPU available and, then you have your accelerations coming down on a standard GPU based system.

(Refer Slide Time: 07:05)



```
net = net.cuda()

GPU is available!

Define loss function and optimizer

In [8]: criterion = nn.NLLLoss() # Negative Log-Likelihood
optimizer = optim.Adam(net.parameters(), lr=1e-4) # Adam

Train the network

In [ ]: iterations = 10
trainLoss = []
testAcc = []
```

Now, comes your next part of it which is to get started with the training. So, you need to define your loss function or the criterion and over here since it is a classification one. So, we just make use of negative log likelihood cost function, or NLL criterion. And then you have to define your optimizer. So, for optimizations as we had seen in the earlier case, where we were trying to play around with the optimizers and different kinds of optimizers and how they come up.

So, you had the standard gradient descent, or the vanilla gradient descent, you had your stochastic gradient descent you had your stochastic gradient descent with different momentum factor added, then you had your Adam or adaptive momentum optimizer. So,

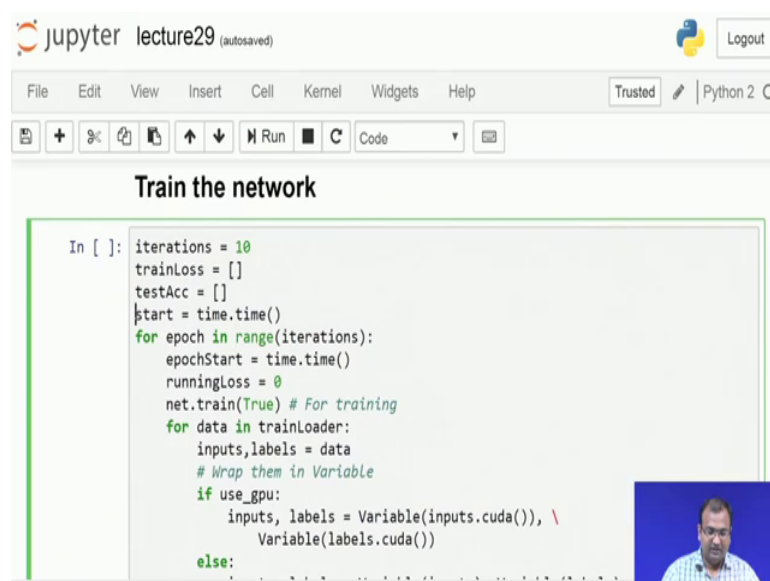
and we did find out that at any given circumstance your under whatever you were operating it down. So, while your Adam takes a tat bit longer than trying to do a standard vanilla gradient descent, but Adam does come down for epoch though that kind of time consumed is more, but it will take less in number of epochs actually to come down.

And now the amount of time in total it would take to come down to that error rate, as compared if you are comparing an Adam with a standard vanilla gradient descent, then an Adam is much faster and, then there is a reason. So, I do often use this term called as a vanilla gradient descent now, it is it has it is just a simple colloquial term used within the community and for the reason, then wherever it is a plain and simple as in a vanilla ice cream.

So, it is a plain and simple kind of a ice cream which you would typically be getting. So, whenever it is a plain and simple way of using, it we just use that extra term in front of it as vanilla so and so and, and it can be either a model or an optimizer or anything and it is just a common lingo within the community to use though not an official one. So, please do not stick down to writing it down onto your papers or any kind of a publications, which you are putting it down.

So, we have the optimizer set down as Adam based on our prior experiences that it does work out much faster to come down to a convergence.

(Refer Slide Time: 09:06)

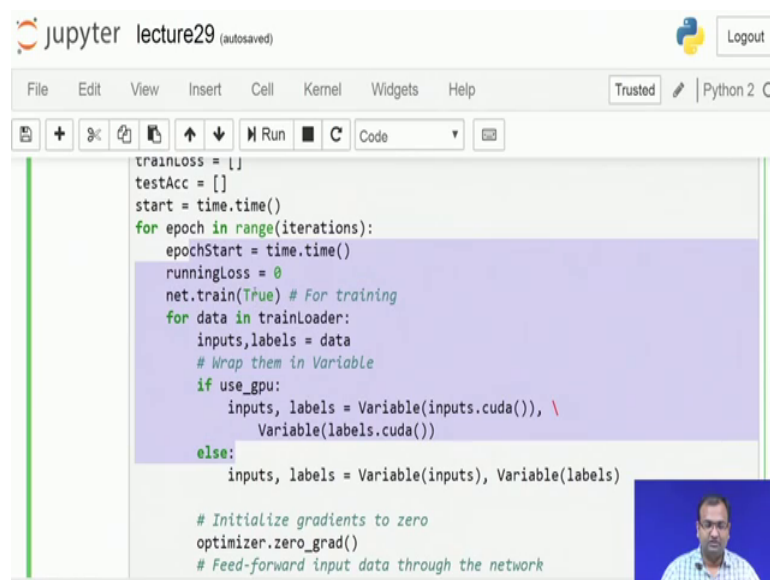


```
In [ ]: iterations = 10
trainLoss = []
testAcc = []
start = time.time()
for epoch in range(iterations):
    epochStart = time.time()
    runningLoss = 0
    net.train(True) # For training
    for data in trainLoader:
        inputs,labels = data
        # Wrap them in Variable
        if use_gpu:
            inputs, labels = Variable(inputs.cuda()), \
                Variable(labels.cuda())
        else:
```


And then we have our network written down over here ok. So, within my trainer function what I have is I set down my total number of iterations, then I have two scratchpad variables created down for my loss and accuracy while I am training. So, what I basically do is while I am training and looking at the error, or the total loss accumulated per epoch, as well as I put on my test dataset to use and at the end of training and epoch, I look at what is the accuracy achieved by this particular on the state test data set at the end of this particular epoch ok.

Then comes my timer over here just to look into the time consumed and the total process, over there to find out how fast it has actually operated up..

(Refer Slide Time: 09:48)



```
trainloss = []
testAcc = []
start = time.time()
for epoch in range(iterations):
    epochStart = time.time()
    runningLoss = 0
    net.train(True) # For training
    for data in trainloader:
        inputs, labels = data
        # Wrap them in Variable
        if use_gpu:
            inputs, labels = Variable(inputs.cuda()), \
                Variable(labels.cuda())
        else:
            inputs, labels = Variable(inputs), Variable(labels)

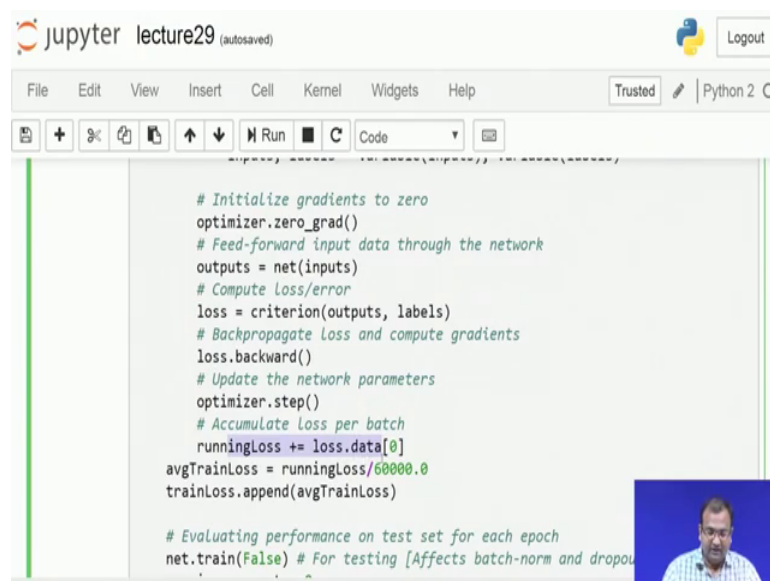
    # Initialize gradients to zero
    optimizer.zero_grad()
    # Feed-forward input data through the network
```

And then we start with our trainer function over there. So, within an epoch what I do over here is typically, I have a batch load of things going down. So, I will be doing within a batch, or within train loader which will just be pushing it down into one batch. Now within the batch I just find out if I have my GPU availability, then I have to convert and typecast all my variables on to a cuda array and, then I have my or otherwise, if it is plain on CPU then I do not do any kind of a typecasting and, then this was the option of casting it down into a variable, or something which can get propagated throughout the network and the memory optimization happens over there.

The next is to zero down all the gradients and, then you do your first part which is your feed forward or the forward pass function over there. So, we had done this forward pass

function in the earlier lecture as well, where the whole objective was just to look into what is the size of how the inputs are varying. So, as the data propagates across the network, then we just wanted to print and see it out so; however, here it is it is an actual forward pass, which happens over there and we are just not going to fool around by looking into the size of the data there is an actual output which comes from over there after you our last layer.

(Refer Slide Time: 11:00)



```

# Initialize gradients to zero
optimizer.zero_grad()
# Feed-forward input data through the network
outputs = net(inputs)
# Compute Loss/error
loss = criterion(outputs, labels)
# Backpropagate loss and compute gradients
loss.backward()
# Update the network parameters
optimizer.step()
# Accumulate loss per batch
runningLoss += loss.data[0]
avgTrainLoss = runningLoss/60000.0
trainLoss.append(avgTrainLoss)

# Evaluating performance on test set for each epoch
net.train(False) # For testing [Affects batch-norm and dropout]

```

Now from there we actually get into computing the loss function and, what is the total error which has been done at the end of this one by this network, which is trying to classify each image which comes down as to belonging to 1 of the 10 classes.

Now, once your loss is computed you need to find your (Refer Time: 11:20) of loss and that is your lost dot backward which is gradient of loss computed, once your gradient of loss is computed down and, then you can have an optimizer one step ahead or the whole update rule written down over here, when just a step. So, this is what we had done in the optimizations lectures and courses as well.

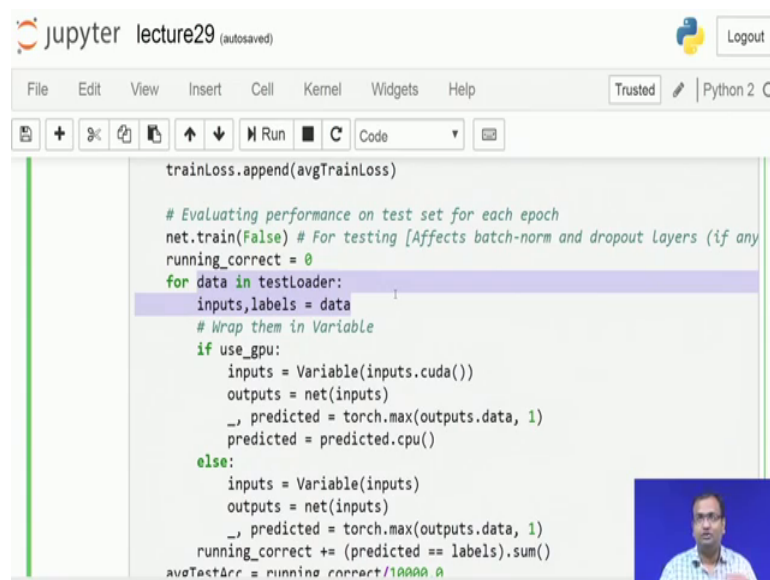
So, from there I get down my running loss or whatever is my loss per batch and, this update happens at the end of every batch. So, every batch keeps on getting an update. So, 100 samples in one batch which go through it you find out what is the error and then you update the weights. And then you keep on continuing this for 600 times. So, that so you

have 100 samples per batch and 600 such batches. So, that would make it 60000 and that is the total number of samples to be operated on.

So, at the end of every of this 600 batches so, now you had 60000 samples which have gone through it. So, what you have to do is basically you take the sum of all the loss and, then divide it by 6000. So, that gives you an average loss per sample which comes down and, then you just keep on happening this to the end of the string and you have an array of the length of the total number of epochs over which this learning has gone down.

Now, once this whole thing has happened down, what I do is actually find out how many of them are correct and, how many of the wrong at the end of each of them. So, now typically while this is running down within an epoch so, within each epoch I am also trying to find out that at the end of training of this network on an independent set which is my testing set how good does it behave because, this set has not been used for training the network. So, none of the weights have changed or anything based on any sample which it sees on the test data set. So, that is what I am doing over here within each epoch itself.

(Refer Slide Time: 13:10)

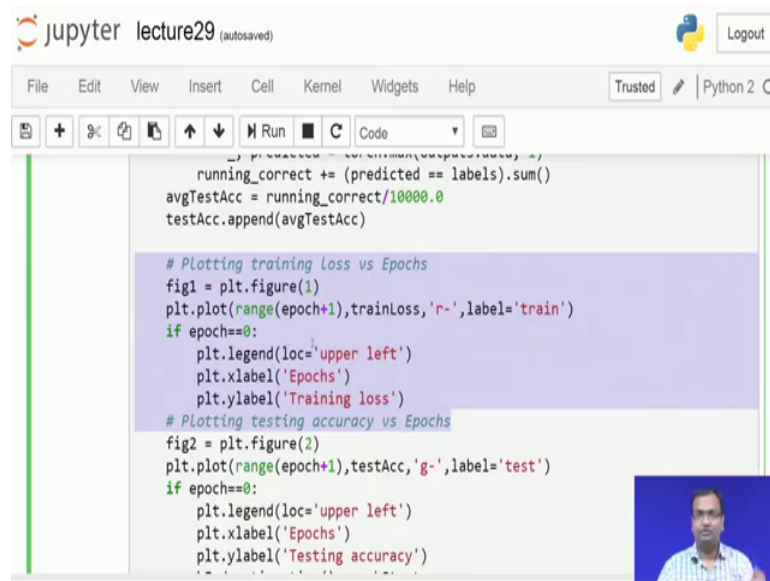


```
trainLoss.append(avgTrainLoss)

# Evaluating performance on test set for each epoch
net.train(False) # For testing [Affects batch-norm and dropout layers (if any
running_correct = 0
for data in testLoader:
    inputs, labels = data
    # Wrap them in Variable
    if use_gpu:
        inputs = Variable(inputs.cuda())
        outputs = net(inputs)
        _, predicted = torch.max(outputs.data, 1)
        predicted = predicted.cpu()
    else:
        inputs = Variable(inputs)
        outputs = net(inputs)
        _, predicted = torch.max(outputs.data, 1)
        running_correct += (predicted == labels).sum()
aveTestAcc = running_correct / 10000.0
```

So, here it is just to load down your variables and convert it down onto your cuda array, if it is available and the loader comes down from your test loader. So, your test loader loads again in batches of 100 and all the ten thousand samples present over there. So, there will typically be 100 batches which have to go through this one.

(Refer Slide Time: 13:27)



```
running_correct += (predicted == labels).sum()
avgTestAcc = running_correct/10000.0
testAcc.append(avgTestAcc)

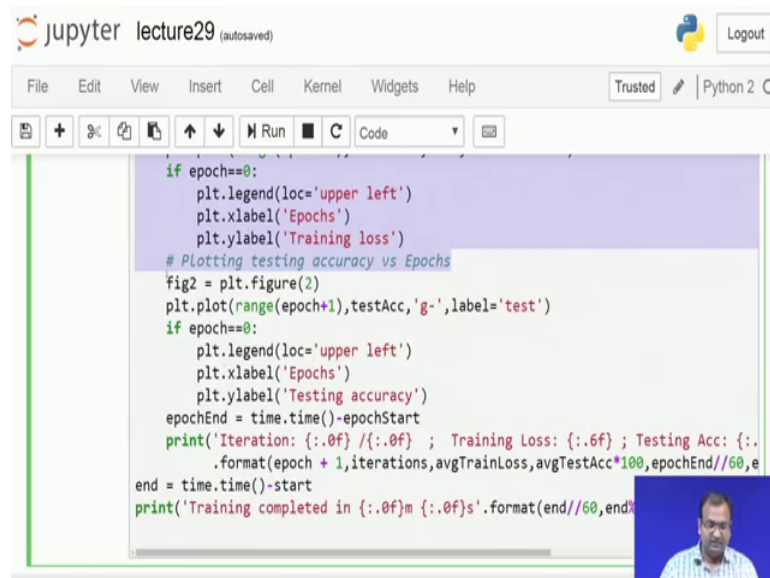
# Plotting training Loss vs Epochs
fig1 = plt.figure(1)
plt.plot(range(epoch+1),trainLoss, 'r-',label='train')
if epoch==0:
    plt.legend(loc='upper left')
    plt.xlabel('Epochs')
    plt.ylabel('Training loss')

# Plotting testing accuracy vs Epochs
fig2 = plt.figure(2)
plt.plot(range(epoch+1),testAcc, 'g-',label='test')
if epoch==0:
    plt.legend(loc='upper left')
    plt.xlabel('Epochs')
    plt.ylabel('Testing accuracy')
```

So, it is the same part what you do over there that you do a feed forward and, then just get down your output and, then your final part. So, you have your input which is converted down, then you see your outputs and then you find out what is the predicted value over there. Now the predicted value which comes down is is basically which index has the maximum value, or that will be whichever neuron gives you the highest probability coming out of it and that is the class to which it will be belonging if you are taking a decision.

Now, here the whole objective is to find out how many of them are correctly classified and that will give you an accuracy. So, that is what typically is done over here and, then you have an accuracy at the end of each epoch itself. Now from there we end up actually getting into the plot part. So, there are 2 plots which you are plotting the first part of the plot is actually to get into the loss functions plot and, that is what is a total error occurring when you are doing a feed forward and, then a back propagation which just the training data set.

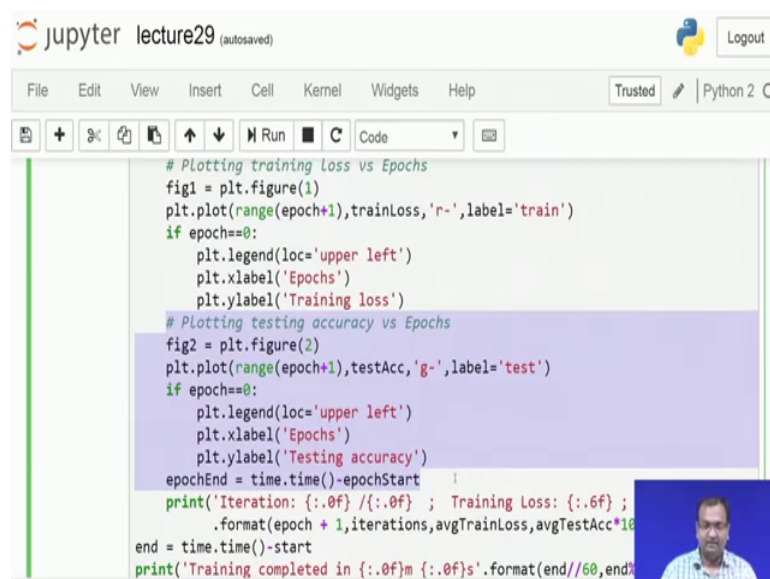
(Refer Slide Time: 14:20)



```
jupyter lecture29 (autosaved) Python 2.0
File Edit View Insert Cell Kernel Widgets Help Trusted Python 2.0
+ %< > Run C Code
if epoch==0:
    plt.legend(loc='upper left')
    plt.xlabel('Epochs')
    plt.ylabel('Training loss')
# Plotting testing accuracy vs Epochs
fig2 = plt.figure(2)
plt.plot(range(epoch+1),testAcc,'g-',label='test')
if epoch==0:
    plt.legend(loc='upper left')
    plt.xlabel('Epochs')
    plt.ylabel('Testing accuracy')
epochEnd = time.time()-epochStart
print('Iteration: {:.0f} / {:.0f} ; Training Loss: {:.6f} ; Testing Acc: {:.6f}
      .format(epoch + 1,iterations,avgTrainLoss,avgTestAcc*100,epochEnd//60,e
end = time.time()-start
print('Training completed in {:.0f}m {:.0f}s'.format(end//60,end
```

The second part of it is to do a accuracy plot with your independent test data set.

(Refer Slide Time: 14:26)



```
jupyter lecture29 (autosaved) Python 2.0
File Edit View Insert Cell Kernel Widgets Help Trusted Python 2.0
+ %< > Run C Code
# Plotting training Loss vs Epochs
fig1 = plt.figure(1)
plt.plot(range(epoch+1),trainLoss,'r-',label='train')
if epoch==0:
    plt.legend(loc='upper left')
    plt.xlabel('Epochs')
    plt.ylabel('Training loss')
# Plotting testing accuracy vs Epochs
fig2 = plt.figure(2)
plt.plot(range(epoch+1),testAcc,'g-',label='test')
if epoch==0:
    plt.legend(loc='upper left')
    plt.xlabel('Epochs')
    plt.ylabel('Testing accuracy')
epochEnd = time.time()-epochStart
print('Iteration: {:.0f} / {:.0f} ; Training Loss: {:.6f} ;
      .format(epoch + 1,iterations,avgTrainLoss,avgTestAcc*100,epochEnd//60,e
end = time.time()-start
print('Training completed in {:.0f}m {:.0f}s'.format(end//60,end
```

And, then look into how much of time it has taken, and then print them out. So this is what goes wrong within my trainer ok. So, let us just get this one running and once this see yeah. So, this has been set running over there. Now what I do at the end of it is let us see how much time it is taking. So, it is not taking actually much of our time. So, you can see that pretty much within four seconds.

(Refer Slide Time: 14:50)

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

/home/rachana/anaconda2/lib/python2.7/site-packages/ipykernel_launcher.py:21: UserWarning: Implicit dimension choice for log_softmax has been deprecated. Change the call to include dim=X as an argument.

Iteration: 1 /10 ; Training Loss: 0.010279 ; Testing Acc: 88.150 ; Time consumed: 0m 4s
Iteration: 2 /10 ; Training Loss: 0.003504 ; Testing Acc: 91.770 ; Time consumed: 0m 3s
Iteration: 3 /10 ; Training Loss: 0.002661 ; Testing Acc: 93.240 ; Time consumed: 0m 3s
Iteration: 4 /10 ; Training Loss: 0.002197 ; Testing Acc: 94.810 ; Time consumed: 0m 3s
Iteration: 5 /10 ; Training Loss: 0.001840 ; Testing Acc: 95.400 ; Time consumed: 0m 3s
```

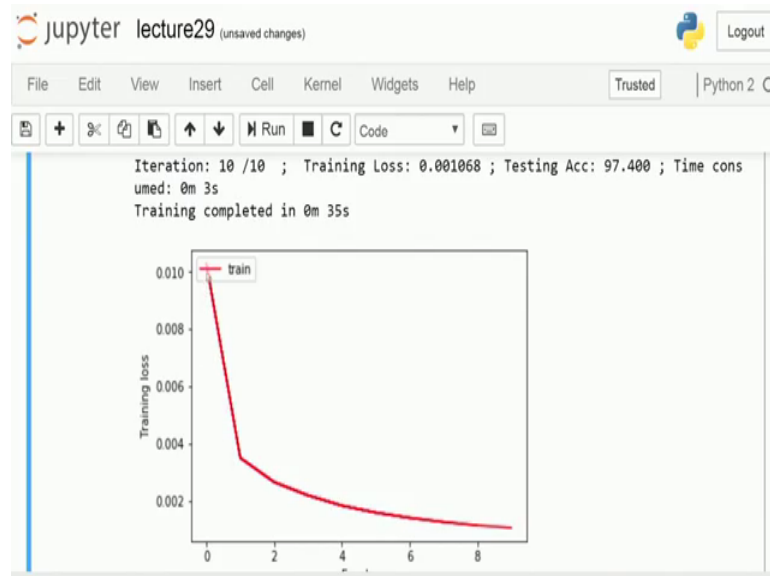
It is going to clean down each epoch and, then it starts with the initial accuracy or somewhere around 88.15 percent and, then it keeps on going.

(Refer Slide Time: 15:00)

```
med: 0m 4s
Iteration: 2 /10 ; Training Loss: 0.003504 ; Testing Acc: 91.770 ; Time consumed: 0m 3s
Iteration: 3 /10 ; Training Loss: 0.002661 ; Testing Acc: 93.240 ; Time consumed: 0m 3s
Iteration: 4 /10 ; Training Loss: 0.002197 ; Testing Acc: 94.810 ; Time consumed: 0m 3s
Iteration: 5 /10 ; Training Loss: 0.001840 ; Testing Acc: 95.400 ; Time consumed: 0m 3s
Iteration: 6 /10 ; Training Loss: 0.001596 ; Testing Acc: 96.030 ; Time consumed: 0m 3s
Iteration: 7 /10 ; Training Loss: 0.001413 ; Testing Acc: 96.560 ; Time consumed: 0m 3s
Iteration: 8 /10 ; Training Loss: 0.001268 ; Testing Acc: 96.720 ; Time consumed: 0m 3s
Iteration: 9 /10 ; Training Loss: 0.001151 ; Testing Acc: 97.050 ; Time consumed: 0m 3s
```

And then quite fast like around 8 the epoch is already at 96.7 percent and then at 10th epoch it is a 97.4 percent.

(Refer Slide Time: 15:11).

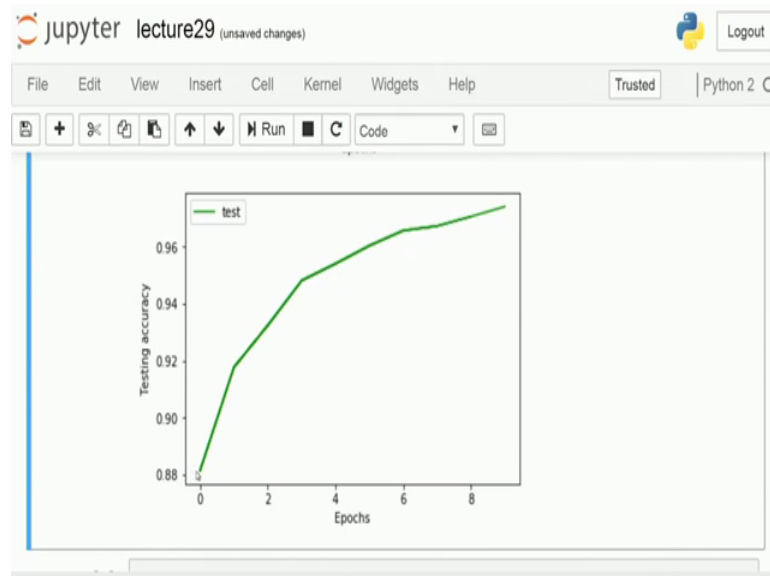


Now, given that if you remember closely that we had this autoencoder initialized neural networks, which were training down over there and the accuracy which they were reaching down. In the best of the cases was somewhere around 94 percent 94, 95 percent and then not more than that whereas, you get done with this kind of a network within a much lesser amount of time, almost a similar kind of a performance or even better right 97 percent though that is a 3 percent increase as compared down to just an autoencoder.

Though like whenever you cross down the 90 percent border this whole fight around 1 percent or 2 percent does not make much of a difference of you had an accurate of say 50 percent, and from there you went up to 53 percent, or even 60 percent that is a huge jump which you are doing. So, it is more of like relative to where you are standing, when you are at ninety percent. So, almost you are very close to coming down to a convergence point over there so; however, that is that is not much to discuss.

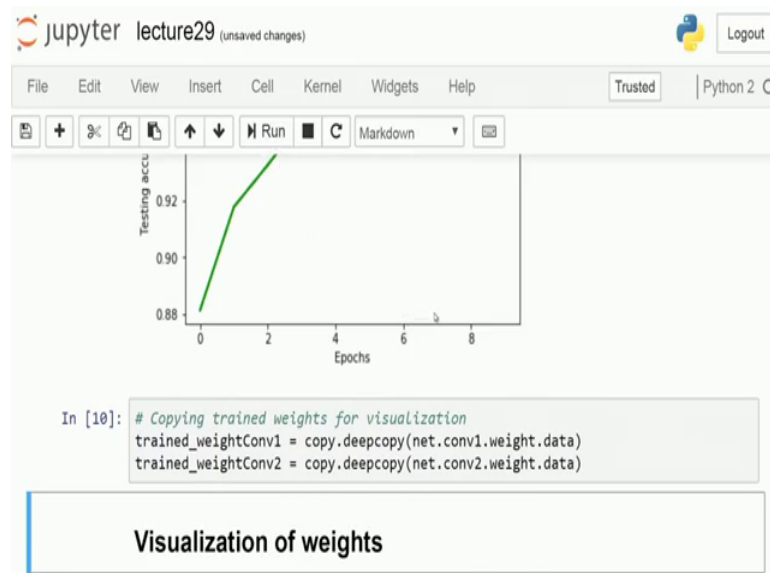
Now, one thing to really remember is that the starting point is 88.1 percent 15 percent accuracy, but that does not mean that it just happened out because, there were 600 times that the weight had actually updated within the network. So, by the end of first epoch that is the reason why you have a decent accuracy coming down at the start of it.

(Refer Slide Time: 16:23)



So, that is what you would be seeing. So, your accuracy starts over here and, then goes down goes up all the way up to 97 percent your loss also does fall down significantly.

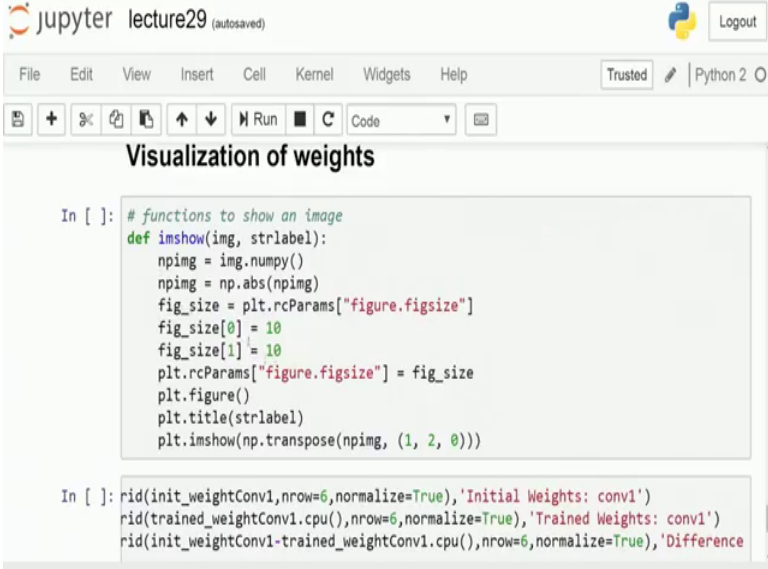
(Refer Slide Time: 16:36)



And actually within the first epoch itself it was falls down to a great point. Now having said all of that the next part is actually to look down into what happened with all of this training. So, what I try to do is I had already copied down all of my weights earlier. So, if you go up you would be seeing that over here, what we had done is once the network is initialized and, you have these randomized weights present. So, I have just copied and

kept all of these randomized weights over there. And then at the end of these ten epochs of training my whole objective is again to copy down these weights of the train network to see, what is the difference and how many of them have changed.

(Refer Slide Time: 17:10).



```
lecture29 (autosaved) Python 2.0
File Edit View Insert Cell Kernel Widgets Help Trusted Python 2.0
Visualization of weights
In [ ]: # functions to show an image
def imshow(img, strlabel):
    npimg = img.numpy()
    npimg = np.abs(npimg)
    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 [ ]: rid(init_weightConv1,nrow=6,normalize=True),'Initial Weights: conv1')
rid(trained_weightConv1.cpu(),nrow=6,normalize=True),'Trained Weights: conv1')
rid(init_weightConv1-trained_weightConv1.cpu(),nrow=6,normalize=True),'Difference
```

So, this is a small piece of code which has been written down just to visualize your code and, then this is just by the same kind of a definition as we had used in the earlier case for our auto encoder weight visualizations as well. Now while in an auto encoder the weights were basically the total number of connections, which were between one neurons in one layer to the neurons in the other layer and, then if you had say some 10 neurons in the first layer which some say for an example say a 28 cross 28 for new 784 neurons which are there in the first layer.

And they connect down to say some hundred neurons in the second layer. So, you had this mapping that there is a 28 cross 28 bit matrix which connects to each of these neurons and, there are such ten crossed 10 matrices created over there of these weights.

Now, here it is very different because, now you do not have all the neurons connecting down to each and every neuron, but they connect down by a convolutional operator over there.

(Refer Slide Time: 18:08)

```
jupyter lecture29 (unsaved changes) Python 2.0
File Edit View Insert Cell Kernel Widgets Help Trusted Python 2.0
Run Code
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)))

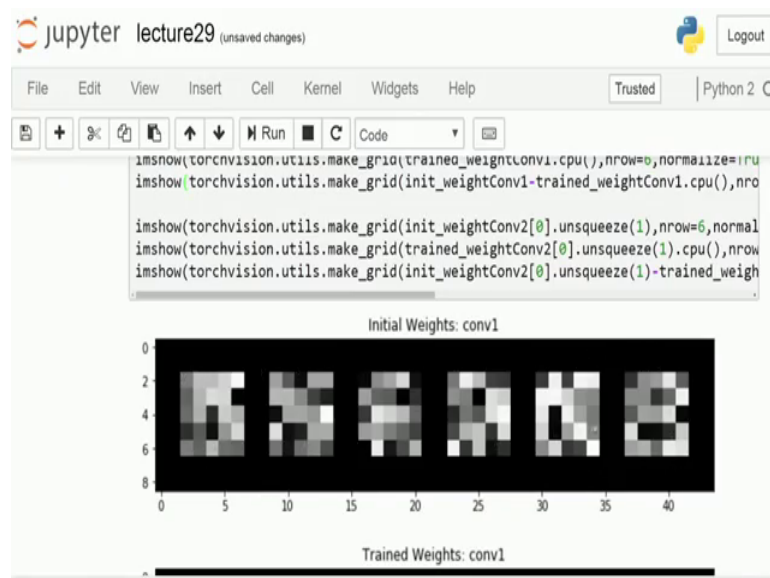
In [ ]: imshow(torchvision.utils.make_grid(init_weightConv1,nrow=6,normalize=True),'Initi
imshow(torchvision.utils.make_grid(trained_weightConv1.cpu(),nrow=6,normalize=Tru
imshow(torchvision.utils.make_grid(init_weightConv1-trained_weightConv1.cpu(),nro

imshow(torchvision.utils.make_grid(init_weightConv2[0].unsqueeze(1),nrow=6,normal
imshow(torchvision.utils.make_grid(trained_weightConv2[0].unsqueeze(1).cpu(),nrow
imshow(torchvision.utils.make_grid(init_weightConv2[0].unsqueeze(1)-trained_weigh

In [ ]:
```

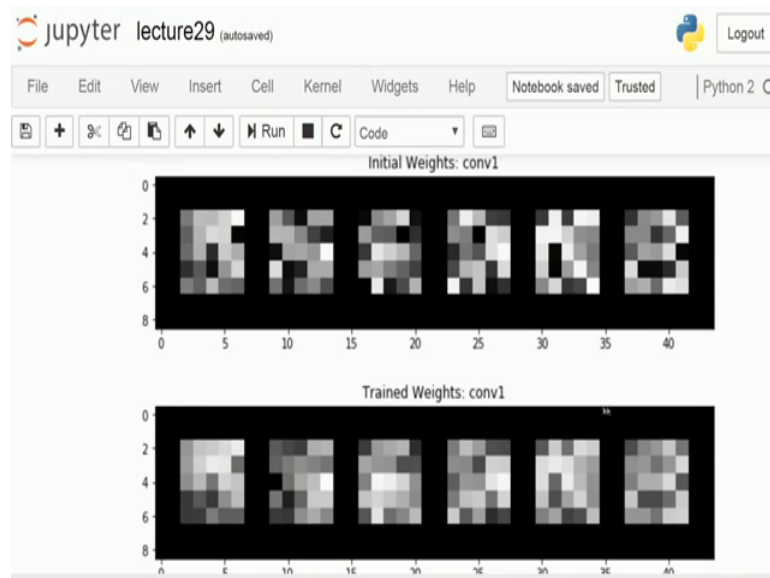
And the size of the convolution operator is basically number of channels on the input side into the kernel size. So, for me on the LeNet if I go up you would see, that my model which is over here. So, on my first layer I basically have 1 channel, there are 6 such kernels and each kernel is of size 5 cross 5. So, I can pretty much visualize it as a grayscale image of 5 cross 5 and there will be 6 of them. And that is where it comes down over here as well ok. So, if I run this one down you would be seeing what comes down over here.

(Refer Slide Time: 18:41)



So, this is for my first convolution layer, I have 6 of my weights which come down 1 2 3 4 5 6 each is a 5 cross 5. So, this is the first pixel 1 2 3 4 5, 1 2 3 4 5. So, there is 5 cross 5 and 6 of these kernels which are over there. Now this is at the start of the training and when nothing has trained and it is just randomly Initialized bits which you have over here.

(Refer Slide Time: 19:03)



Now this is at the end of the training, when I have done all of these 10 epochs of training and within each epoch that have been 600 batches which have updated each batch of size 100 and, then I get down my weights coming down something of this order.

So, now the interesting part is you do not typically tend to look into structures and then, or some visible kind of structures within these convolutions quite generally while, if you were comparing this with what had happened down in your auto encoders and, you had more distinct structures. And you need to keep one thing in mind over there the structure was much easily visible because, it was a 28 across 28 batch whereas, over here this is just a 5 cross 5 fact.

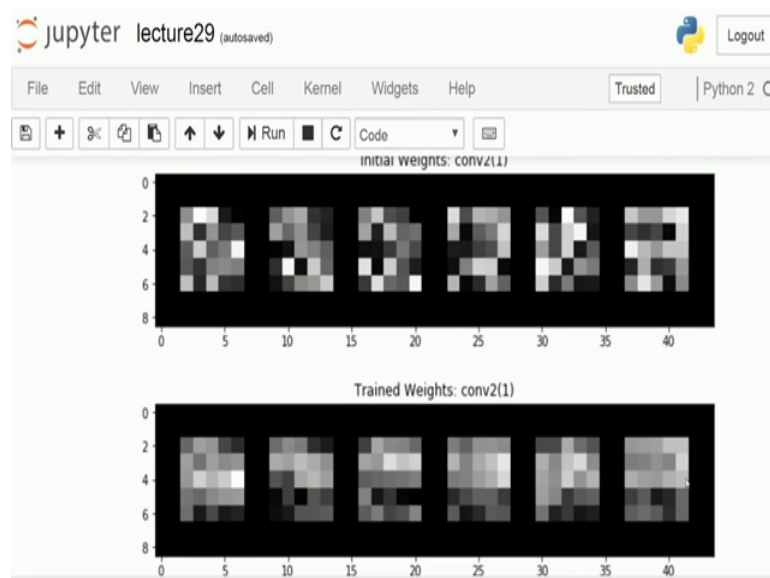
So, it is really hard to get done any kind of a visualization coming down over here, but then if you look into the differences you would see that there has been a gradient kind of a difference change over there. Now if this was training over longer and longer period of time because, if I look down into my curves over here, I see that they have not yet saturated actually and, then they are still monotonically so, my loss function is

monotonic is still decreasing my accuracy over here has still a monotonic tendency of increase over there so; that means, that these can still get keep on getting trained.

And come down to a conformal point. So, for general purpose what I would say is that you guys can actually train it down for about 100 or 500 epochs and then you would see much better structures getting visualized over here and, they would look very much similar to your gradient operators in general, now because that is what we have seen down.

Now, the next part is that if I go down on the second convolution layer. So, now, in the second convolution layer what comes down is you again have a 5 kernel over there, but then there are 6 channels which are present. So, it is basically a 6 into 5 into 5. Now it is hard to visualize a 3 d matrix over there and that too like if I was trying to do it in color, then I would need a RGB 3 different channels, but now there are 6 different channels and there are no. So, of standard visualizes. So, what we do over there is we visualize each channel independently, but for a given kernel.

(Refer Slide Time: 21:00)



So, what I do over here is that on the second convolution layer, I just pick up the first convolution kernel over there.

Now, the first convolution kernel or any of these random convolution values which have 6 channels and, each is 5 cross 5 kernel over there. So, that is what I see over here and

this is the randomized 1 initialized. Now at the end of the whole training process this is what the weight assumes; now you can still see some sort of directive structures coming down over here, though it makes it really hard and these are the weights which come down.

Now one thing in mind we have not yet put down any of our earlier concept of sparsity or denoising anything over here and still, this is what it works now, when if you try to really put all of these concepts of sparsity and denoising into these ones this would actually end up learning much better and better weights. So, somewhere down the line when we would be doing our other things, which will lead down any of these convolution operations to trying to create something similar to an autoencoder called as a convolution, or encoder we would actually bring you closer to doing all of those where, you would see whether convolution operations itself can also be defined can also be used in order to define auto encoder kind of a structure.

And can we used unsupervised feature learning in order to initialize a network to come down to a much conformal representation. So, till then it is it is interesting to go on with these ones. So, in the next lecture I am going to move over from just mere grayscale images onto color images and, then we look how these kind of kernels also perform over there. So, just stay tuned and stay excited for the next lecture.