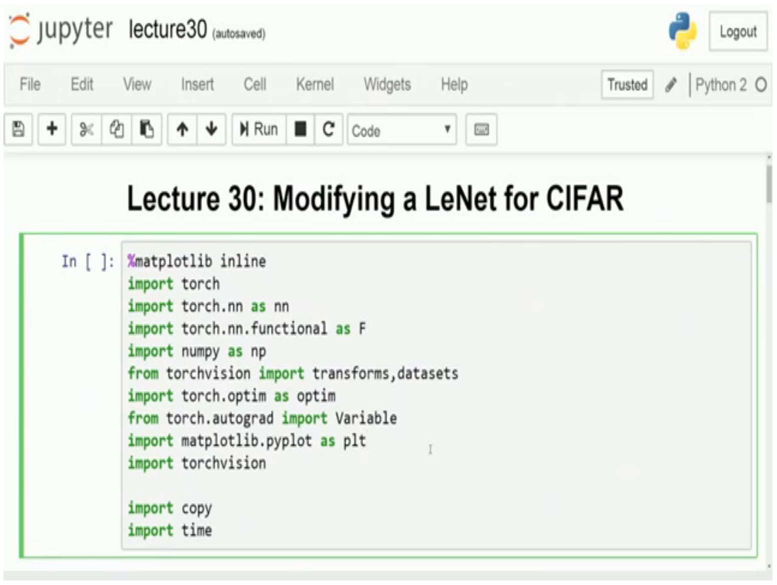
Deep Learning for Visual Computing
Prof. Debdoot Sheet
Department of Electrical Engineering
Indian Institute of Technology, Kharagpur

Lecture – 30
Modifying a LeNet for CIFAR

So, welcome this is the next lecture, where we are going to show you how to use a LeNet for your color images. And the difference where it comes down from standard LeNet as in for MNIST classification is that over there they were all grayscale images of 32 cross 32 and we were processing out. Now we are going to use another new data set which is called a CIFAR and on over here though your spatial span of the images are still rates it into 32 cross 32, but they are all color images. So, your input is no more one single channel, but it is 3 channels of input which comes down to you.

(Refer Slide Time: 00:50)

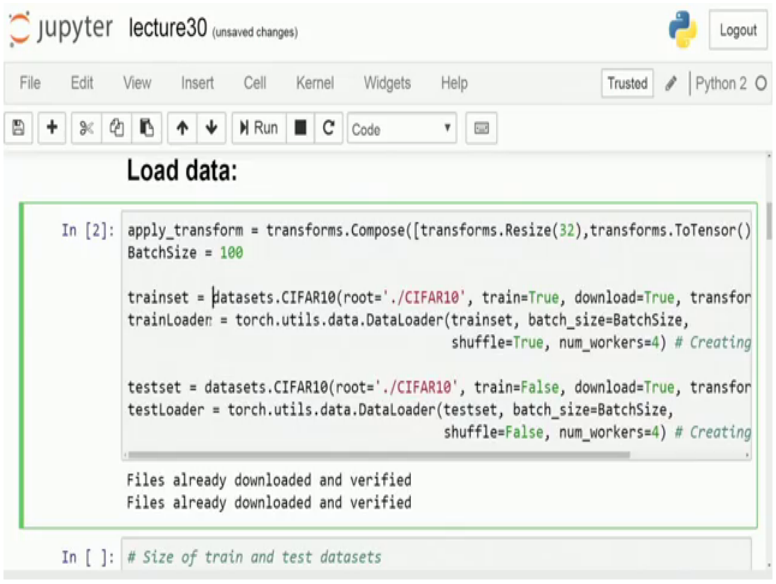

jupyter lecture30 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

Lecture 30: Modifying a LeNet for CIFAR

```
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, that is where it will have to modify because, your network cannot anymore be just taking on one single channel, you can put over there and the data complexity and the total number of tunable parameters will also be changing. So, as we go through it I will get down more and more into it.

(Refer Slide Time: 01:06)


jupyter lecture30 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

Load data:

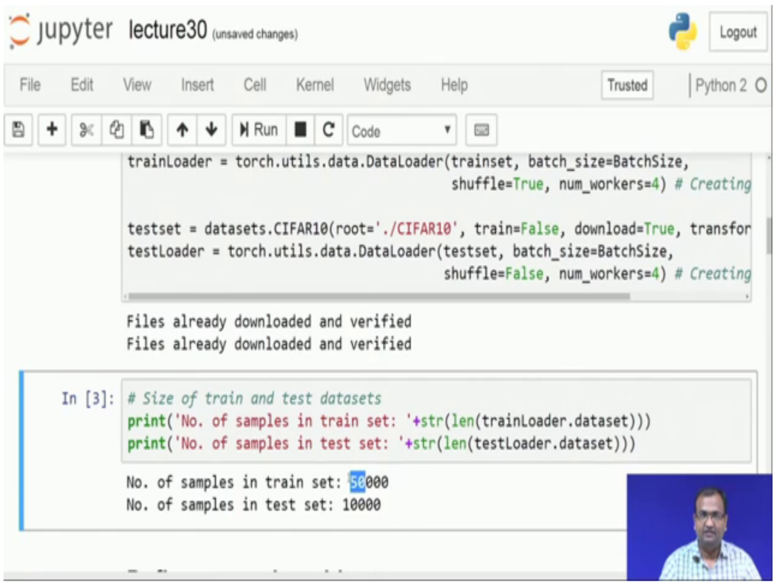
```
In [2]: apply_transform = transforms.Compose([transforms.Resize(32),transforms.ToTensor()
        BatchSize = 100

        trainset = datasets.CIFAR10(root='./CIFAR10', train=True, download=True, transfor
        trainLoader = torch.utils.data.DataLoader(trainset, batch_size=BatchSize,
                                                  shuffle=True, num_workers=4) # Creating

        testset = datasets.CIFAR10(root='./CIFAR10', train=False, download=True, transfor
        testLoader = torch.utils.data.DataLoader(testset, batch_size=BatchSize,
                                                 shuffle=False, num_workers=4) # Creating

        Files already downloaded and verified
        Files already downloaded and verified

In [ ]: # Size of train and test datasets
```

So, the first part is the same on just trying to write down your header and, your initial files which have to be called on and, then the next part is substantively to get your data loaded. Now what we do is we try to stick down to a similar kind of a conformal thing, as we had in our MNIST classification problems with LeNet and, that was to take down just one hundred images on my batch and, then subsequently do it.

So, here also I am trying to stick down to the same batch size of 100 images my training set is trying to load it down from the inbuilt touch data sets folder which also supports CIFAR. Now CIFAR is basically natural images available, or now small say there are cats, dogs animals bicycles kind of thing and there are ten categories and that is why it is also called as CIFAR 10.

Now, you can just load down the data set using the same train set and the train loader function over there and, this will just download if it is not already downloaded and then do it so. This is quite similar to using any of the standard data sets as we had been using for MNIST as well in the earlier examples. So, let us load this fight. So, so it just sees that files are already present and, then they are verified and it is ok. So, my 2 trainers are created over here.

(Refer Slide Time: 02:18)



```
trainLoader = torch.utils.data.DataLoader(trainset, batch_size=BatchSize,
                                          shuffle=True, num_workers=4) # Creating

testset = datasets.CIFAR10(root='./CIFAR10', train=False, download=True, transfor
testLoader = torch.utils.data.DataLoader(testset, batch_size=BatchSize,
                                         shuffle=False, num_workers=4) # Creating
```

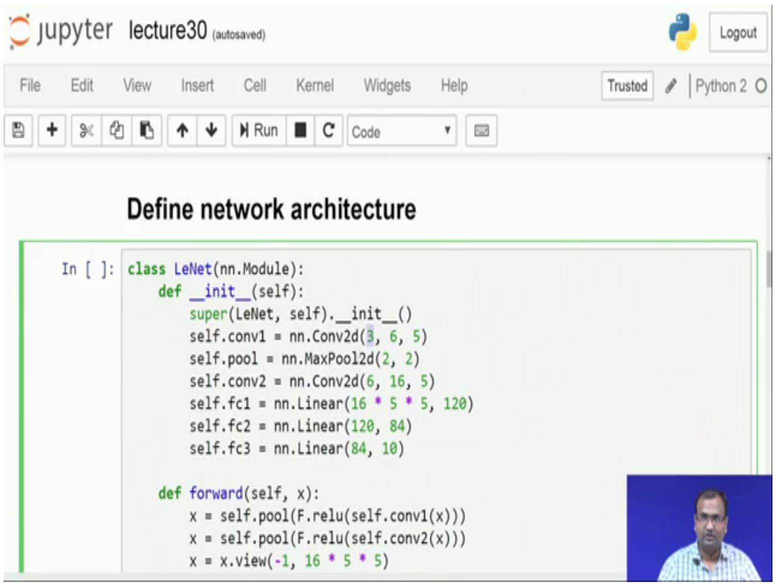
```
Files already downloaded and verified
Files already downloaded and verified
```

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

```
No. of samples in train set: 50000
No. of samples in test set: 10000
```

Now, the next part over here is to get down and actually print down the number of samples. So, let us print it out and quite on the on the different site is that while on MNIST you had 60000 examples for your training here, we have fifty thousand examples for training and, then 10000 examples form the testing over here. So, it is that is the only difference which comes down, but other than that the next difference is; obviously, they are not grayscale images and these are color images and, that will make a modification to my network definition.

(Refer Slide Time: 02:48)



Define network architecture

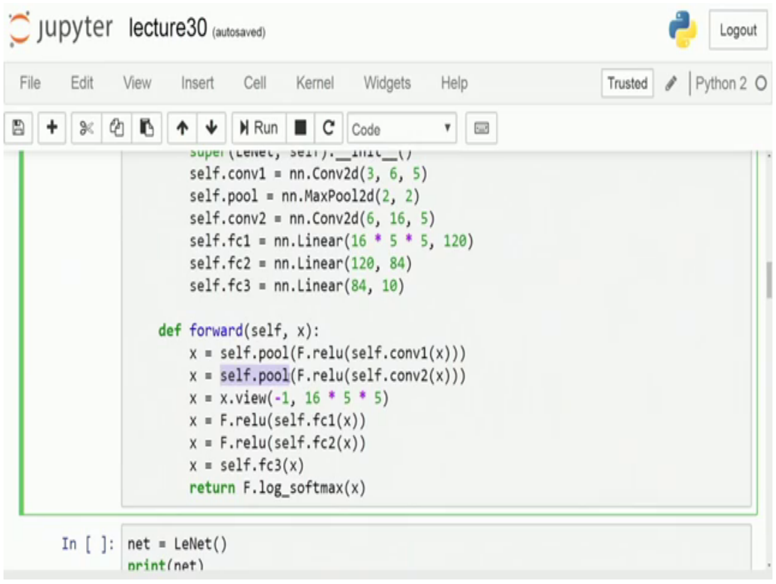
In [ ]:
```
class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
```

So, let us get into the network definition over here ok. So, here what I do is I try to get down a convolution, which takes in 3 channels on my input and, then generate 6 channels on my output. So, you remember the first layer of a LeNet and what it was doing is that my whatever was my input it will do if 5 cross 5 convolution over there. And, then generate that many number of outputs which is equal to the number of unique convolution kernels; however, over here what I have is that I have a 3 channel input over here.

It is no more single channel or a gray scale image which goes into it. So, it is a color image which comes in. So, from there I am mapping it down to 6 via 6 such convolution colors and each of this convolution kernel has a spatial spread of 5 comma 5 or 5 cross 5, and that is what is written over here we are not any more explicitly specifying what is the kernel size and what is the batch, but because the first argument which goes into Conv 2 d operator is actually the kernel size over there ok. So, this slide over here still says as one comma one and then there are no plannings which are introduced over there.

The next part is to do a max pooling and the max pooling, over here is to do a 2 d max pooling with 2 cross 2 kernel and with the stride of 2. So, this goes down the same way as we had in the earlier case. Now once this part is done the next part is actually to go down and get a look get. The next convolution being defined over there so, for the next convolution which we define over here, it is it is what maps down 6 channels 1 to 16 channels and with the kernel size of 5 cross 5 ok. Now that is done and an pretty much set over there. Now the next part is basically to map down all my outputs coming down over here and, then map it down to 120. Now, where we make a difference over here is over here taking down that, we are just taking it 1 1 single input coming.

(Refer Slide Time: 04:49)


jupyter lecture30 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

```
        super(LeNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return F.log_softmax(x)
```

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

Now, we know now instead of one single input you have 3 channel inputs coming down and, the rest of the network pretty much stays the same. Now what I do inside over here is I define my forward pass and for my forward pass, what I am doing is I have my convolutional the first convolution operator written down conv1 and I do relu or basically a non-linear transfer function over there and, then a pooling operator.

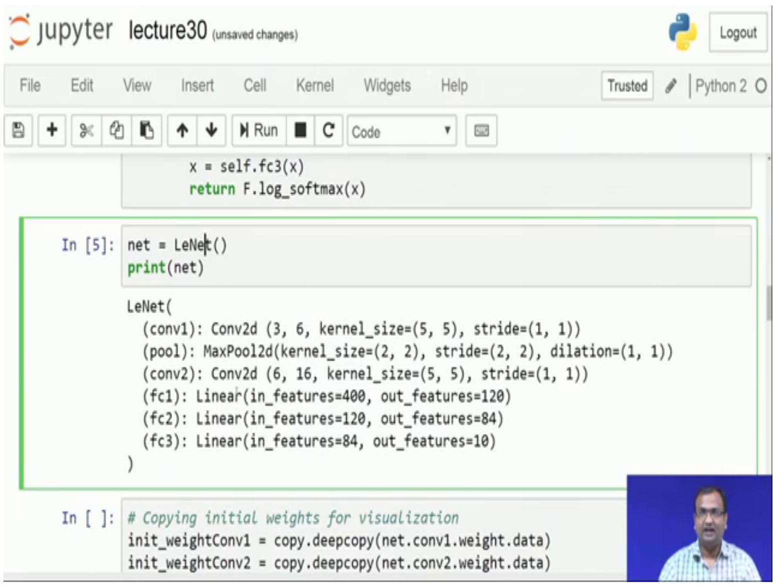
Now see one thing what I have is that since I am going to use the same kind of a pool in 2 instances, it is always a 2 cross 2 max pooling and doing it. So, I am no more defining it as 2 different pooling layers which are actually the same in their function and, I can actually do away by calling them one after the other. So, in case there were 2 different kinds of pooling operator then maybe defined 2 different pooling operators otherwise it does not because otherwise it does not make sense.

However, for convolution we had to define 2 different convolutions because, the number of input channels over there was quite different; in both the cases as well as the number of output channels are different. In case say the input output mapping and the kernel size everything is same you can just define it once as 1 module and, then within the forward pass you can keep on re recursively calling that module as many times as you want. So, latent down and then more deeper networks and more detailed networks is where we will get into reusing all of these, but the first kind of reuse which you see over here, for any kind of functional structure that is the pool layer which comes down.

So, once I have this output coming over here the next part is to actually flatten it out and get down a linear combination of 400 neurons coming and, that is what is connect down over here from 400 neurons 1 to 100 and 20 neurons that is that is what I do over here as my forward pass through fc 1 and, then doing a non-linear transformation as a relu.

The next one is to do a forward pass over fc 3 and then relu and, then the next one is to do a forward pass over fc 3 and that gives me 10 outputs over here and so, this is a classification network as in the earlier case we had a log softmax written no we write down the same blocks of max over here. So, this is about defining the network let us run it and then I have my network to be defined.

(Refer Slide Time: 06:57)


jupyter lecture30 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

```
        x = self.fc3(x)
        return F.log_softmax(x)
```

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

LeNet(
  (conv1): Conv2d (3, 6, kernel_size=(5, 5), stride=(1, 1))
  (pool): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1))
  (conv2): Conv2d (6, 16, kernel_size=(5, 5), stride=(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)
)
```

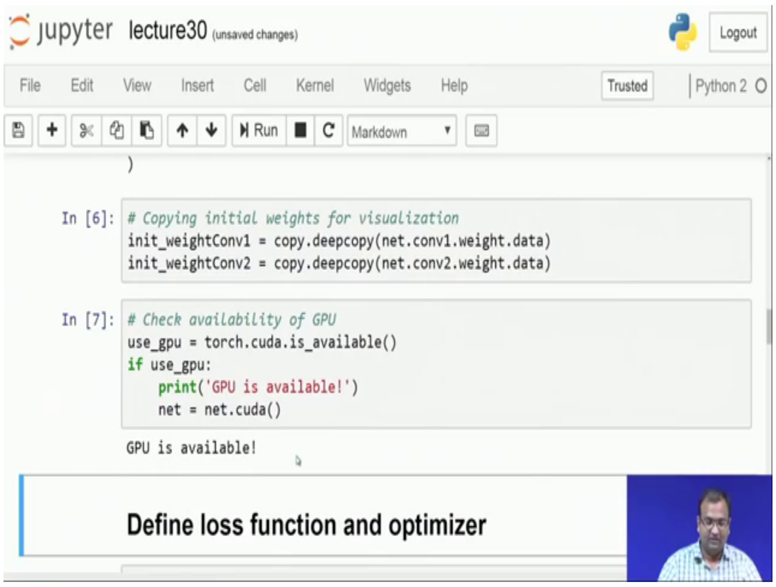
```
In [ ]: # Copying initial weights for visualization
        init_weightConv1 = copy.deepcopy(net.conv1.weight.data)
        init_weightConv2 = copy.deepcopy(net.conv2.weight.data)
```

So, that is what is defined as net at this point of time I have my network define and all the weights initialized and available to me for the usage. So, this is quite straightforward we did see that what I have is basically convolution which goes down over here, then a max pooling operator on the 2 d side of it and then again in a convolution and, then my linear combinations coming down.

And then this is a very straightforward way of looking down into whatever I wanted to define in terms of writing down just single arguments coming into, it is what has actually been defined on the system over there and, then there is nothing wrong and this pool operator over here is what is being reused. So, the output which comes over here again goes through pool and, then that output is what is fed down over here and you could pretty much do these sort of reuse of each of the models because, your forward function is how the data mapping is mapped on it is it is not in the order of exactly how these things come, you can have them quite juggled up and down as well, but then the only thing which defines how the data flows is just this forward function which is where it is written.

So, this was to give you a first hint for the first time, that it is it is not necessary that all your structures will have to be written down in the same order, it is just for convenience and for easy to debug, when you are trying to look into it at all the structures as they would come down they written in the same order as how the network gets built up, but then that is not a necessity from the perspective of handling the data the only necessity is that your forward function is properly defined which can take in the data in that particular format.

(Refer Slide Time: 08:35)

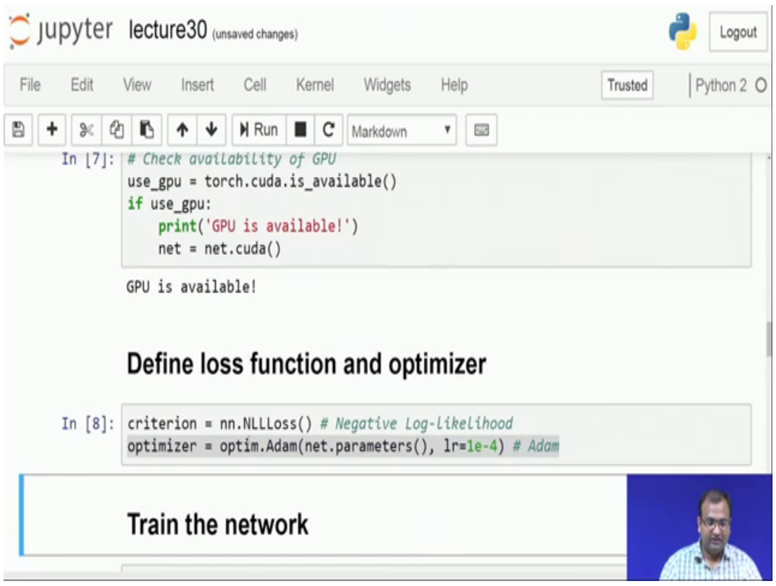

jupyter lecture30 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Markdown

```
)
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

And then what I do is once my network is defined, I try to just copy down my weights and keep them for my further use. So, this is in line with what we had done in the earlier lecture, where I was trying to show you that at the end of training how many of these bits have actually changed and what is the kind of a change which happens over there, next I check down for my availability of a GPU and yes my GPU is available. So, it is a great news and then I can get started with defining it.

(Refer Slide Time: 08:56)


jupyter lecture30 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Markdown

```
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!
```

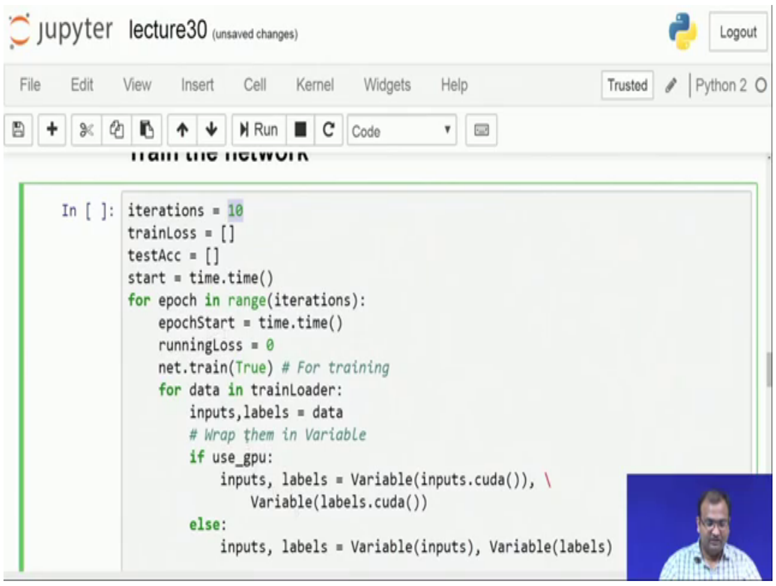
Define loss function and optimizer

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

Train the network

So, since it is a classification function, which I am using over here classification problem which I am using so, my cross function or the criterion, over here is still stays as negative log likelihood without much of an issue, the optimizer as in from our earlier expenses experiences coming down, we stick down to using and Adam for an adaptive momentum optimizer over here. So, that is what is defined and then let us get into my training part over there.

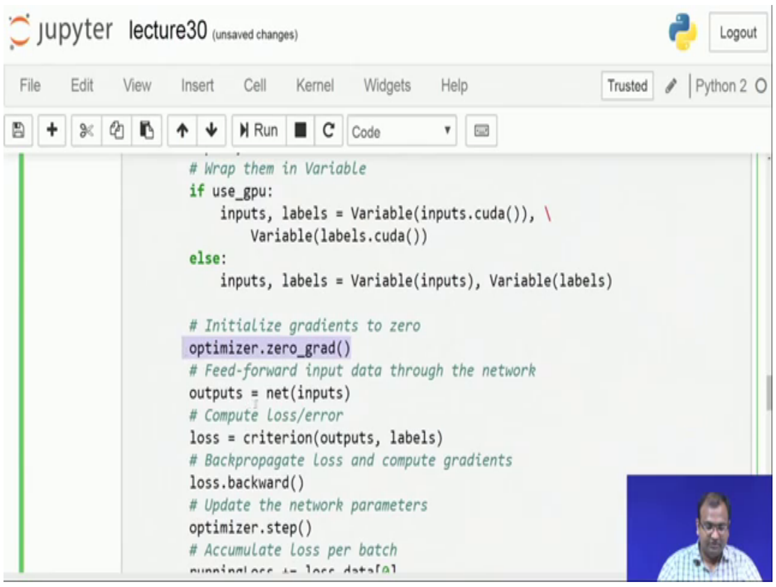
(Refer Slide Time: 09:20)


jupyter lecture30 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

```
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:
                    inputs, labels = Variable(inputs), Variable(labels)
```

Now, the trainer does not have much of a difference coming, in we just train it over ten iterations or 10 epochs over there, within each epoch what I am doing is I just load my data and, then if it is on the GPU then I just convert it onto my GPU.

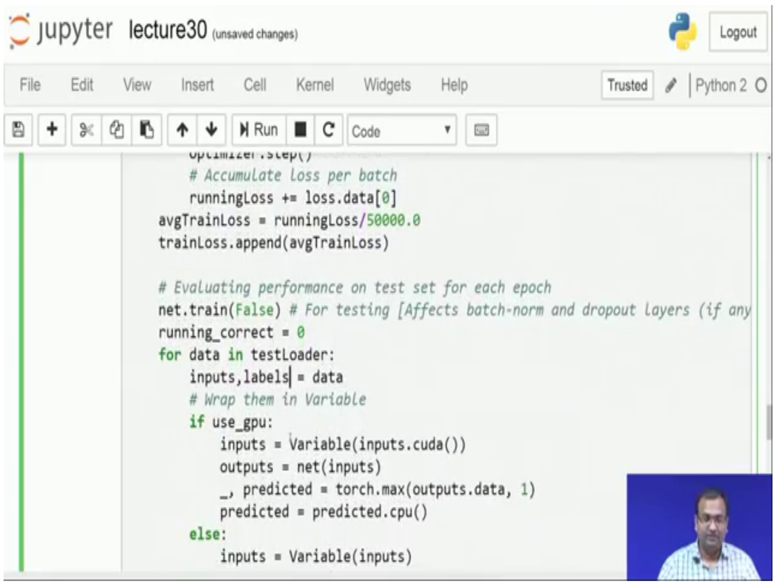
(Refer Slide Time: 09:38)


jupyter lecture30 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

```
        # 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
        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]
```

And, then just zero down on the gradients for the optimizer do a feed forward over the network and get down my output. So, whatever is this a batch size of the data which is over here. So, my outputs are also of the same batch size, and then I find out what my loss is or what is the number of errors wrong things, which it has made done while getting classified through this network and, then do what nabla of loss, or the gradient of the loss coming down over here and finally, get down to my optimizer and step on the optimizer over there.

(Refer Slide Time: 10:09)


jupyter lecture30 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

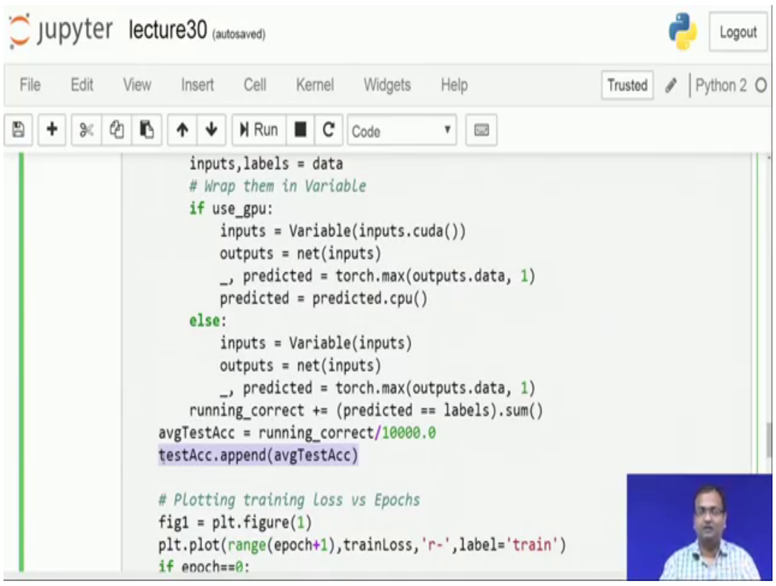
```
            optimizer.step()
            # Accumulate loss per batch
            runningLoss += loss.data[0]
        avgTrainLoss = runningLoss/50000.0
        trainLoss.append(avgTrainLoss)

        # 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)
```

Finally the point is to go down as running loss and to get down what is the total loss which comes down over here and, then from there come down to what is the average loss during trade and, then just keep on adding it down over the epochs ok. Now that is great and good part the next part is actually over within an epoch to actually look down into what is the way of how this whole thing is performed.

So, in the earlier case we just had evaluated our errors or the losses using our training data it, but here the point is independently over my test dataset, which has ten thousand examples how does it behave. So, here what I am trying to do is in case the GPUs available, then just convert all of my inputs into a variable which is cuda type casted available on the GPU, then do a feed forward over the network and then find out my predicted.

And over here whichever neuron is supposed to have the highest prediction value, or the highest probabilities what it will throw out, then I just find out which index over there, or that is the class which is at the highest probability of being predicted and, then from here what I do is just convert it onto a CPU number and that is because, the rest of the computations are what run on the CPU are not on the GPU. So, in case I am just I do not have an access to a GPU.

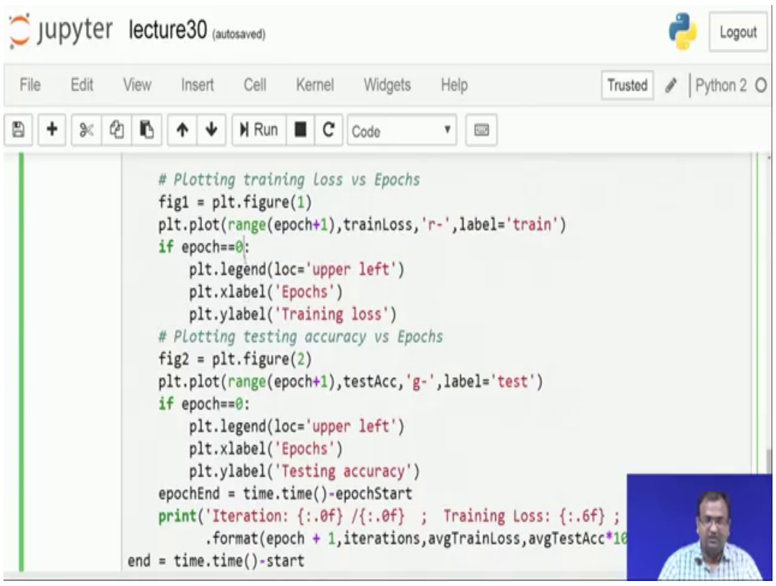
(Refer Slide Time: 11:26)


jupyter lecture30 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

```
        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()
    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:
```

So, this part will not be running and then I just run on this part, then it is its not an issue because my predicted is what is already available on the CPU and then I find out my total number of carats. And if whatever is predicted matches down the exact class level of what was present in my testing meter, then it is perfect it is correctly predicted. So, what I do is I have down my scores given down over 10000 examples. So, that is the total number of things which have to be classified. So, my average correct value or the accuracy is basically divided by 10000 on my test data. And then I just keep on appending my test accuracies across all the epochs which have come down over here.

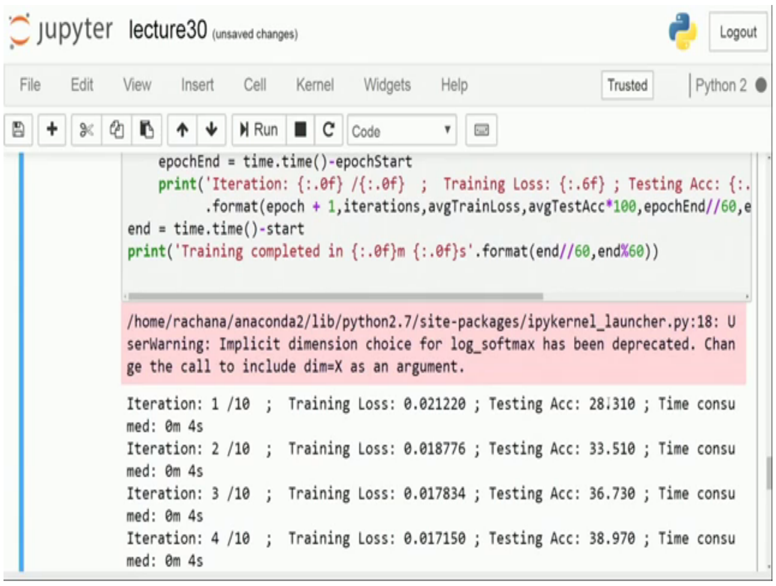
(Refer Slide Time: 12:07)


jupyter lecture30 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
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*10
end = time.time()-start
```

So, from there going on to how to actually start plotting this loss functions over here so, the first part is actually to look down and, then try to plot down the loss that the plot of all the losses, which comes from this next part is actually to get down what is the total accuracy and, then plot down the accuracy is coming up as well. So, once that is done we just look into this part which is to look down for iteration as it keeps on printing and, then finally, just prints out what is the final total amount of time taken down for this to happen.

(Refer Slide Time: 12:35)


jupyter lecture30 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

```
    epochEnd = time.time()-epochStart
    print('Iteration: {:.0f} /{:.0f}  ;  Training Loss: {:.6f} ; Testing Acc: {:.
          .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%60))
```

```
/home/rachana/anaconda2/lib/python2.7/site-packages/ipykernel_launcher.py:18: U
serWarning: Implicit dimension choice for log_softmax has been deprecated. Chan
ge the call to include dim=X as an argument.
```

```
Iteration: 1 /10  ;  Training Loss: 0.021220 ; Testing Acc: 28.310 ; Time consu
med: 0m 4s
Iteration: 2 /10  ;  Training Loss: 0.018776 ; Testing Acc: 33.510 ; Time consu
med: 0m 4s
Iteration: 3 /10  ;  Training Loss: 0.017834 ; Testing Acc: 36.730 ; Time consu
med: 0m 4s
Iteration: 4 /10  ;  Training Loss: 0.017150 ; Testing Acc: 38.970 ; Time consu
med: 0m 4s
```

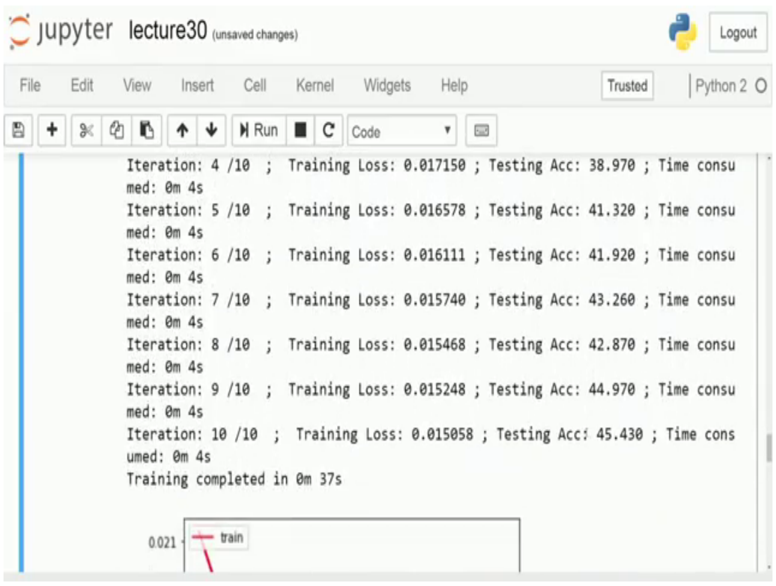
So, let us get this one running. Now once you would once you invoke it out and, then the training has started down you say see that it takes down about roughly 4 seconds to work done per epoch. And then it starts with an initial accuracy of 28 percent.

So, that is not something which is quite convincing to most of you because in the earlier example you saw that it itself started down with an initial accuracy of about 88 percent that was, but then you need to remember one thing in mind that that problem was rather a simpler problem because, you just had grayscale images, where the background was pretty much black and the foreground in which something is written down is just, in higher intensities or like full value of 1, if you are looking on in the floating 1 number system.

So, 0 and 1 and there is a very high amount of contrast between things and, it is just based on different strokes in my writing pattern that it is easy to discriminate, whereas here these are small natural images. So, there are cats, dogs apples cars bikes each of small sized thumbnails of size of 32 cross 32.

Now, given this point that say if you have a bus which is in the size of 32 cross 32 pixels versus, if you have a small minivan versus if you have a some sort of a truck, given down in small thumbnail sized images it is really even for humans it is a hard problem to understand. So, here it is also facing the same kind of a problem most likely like it is its just confusing around with these similar looking objects which are getting bound together. So, that is one of the reasons why you would not initially get to see that high accuracy coming down.

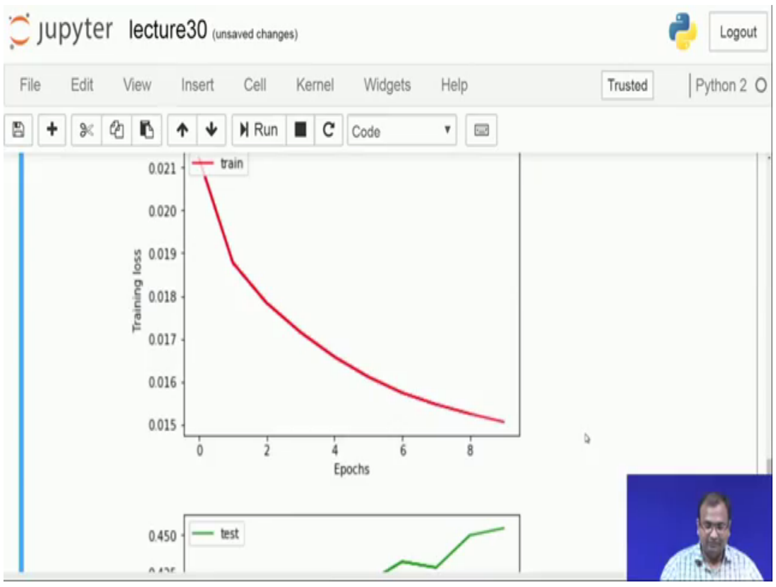
(Refer Slide Time: 14:00)


jupyter lecture30 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code
0.021
train

```
Iteration: 4 /10  ;  Training Loss: 0.017150 ; Testing Acc: 38.970 ; Time consu
med: 0m 4s
Iteration: 5 /10  ;  Training Loss: 0.016578 ; Testing Acc: 41.320 ; Time consu
med: 0m 4s
Iteration: 6 /10  ;  Training Loss: 0.016111 ; Testing Acc: 41.920 ; Time consu
med: 0m 4s
Iteration: 7 /10  ;  Training Loss: 0.015740 ; Testing Acc: 43.260 ; Time consu
med: 0m 4s
Iteration: 8 /10  ;  Training Loss: 0.015468 ; Testing Acc: 42.870 ; Time consu
med: 0m 4s
Iteration: 9 /10  ;  Training Loss: 0.015248 ; Testing Acc: 44.970 ; Time consu
med: 0m 4s
Iteration: 10 /10  ;  Training Loss: 0.015058 ; Testing Acc: 45.430 ; Time cons
umed: 0m 4s
Training completed in 0m 37s
```

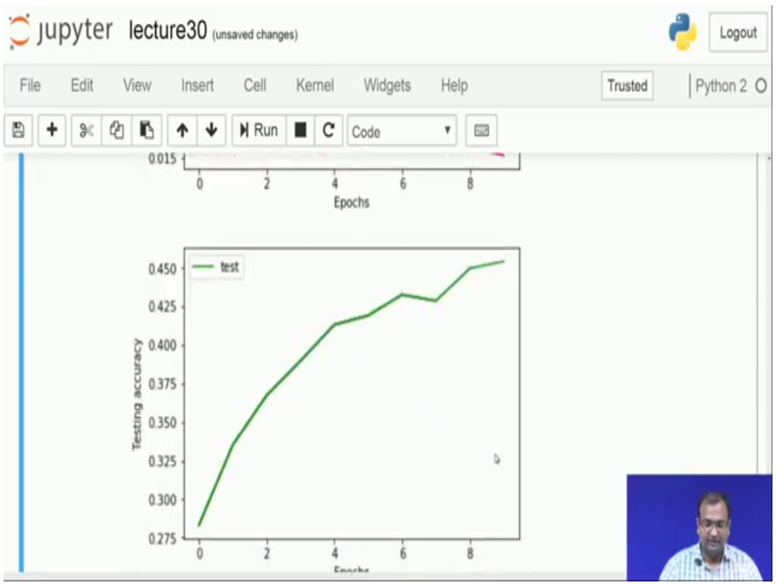
However; one major point is that and the end of the training.

(Refer Slide Time: 14:03)



When it took down 37 seconds you can still see that there is a monotonic decrease in the lost curve as well as a monotonic increase in the accuracy is coming down over here.

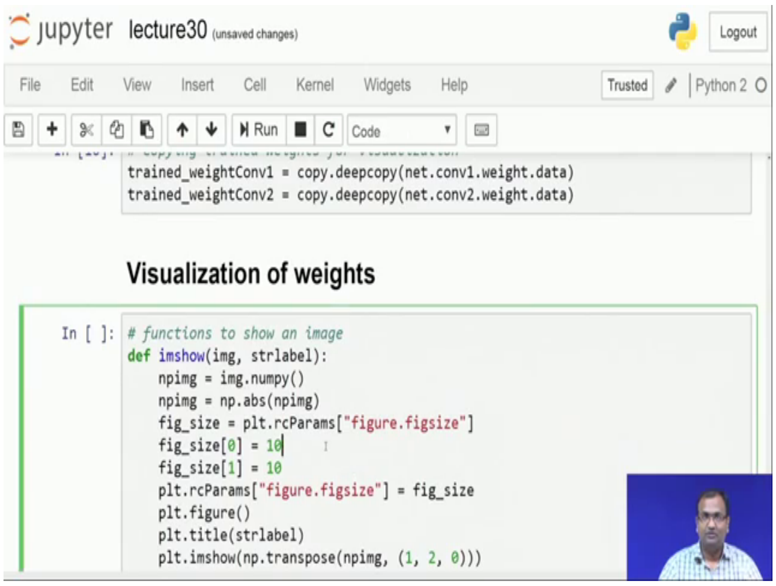
(Refer Slide Time: 14:14)



And that makes us to one of these points that, if we keep on trading it down for a larger number of epochs so, say put it down to train down for hundred epochs you would pretty easily see that this goes down much above the 80 percent benchmark.

Now, one more thing you need to keep in mind is also played on with the learning rates over there. So, change the learning rate iteratively as we had discussed in the earlier classes of how to come down to convergences. So, after every say 10 epochs you just reduce the learning rate by half of it and, then to let it keep on going down over there and, you would see that there will be a monotonic decrease and eventually it will come down to it is most saturation point over there.

(Refer Slide Time: 14:52)


jupyter lecture30 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

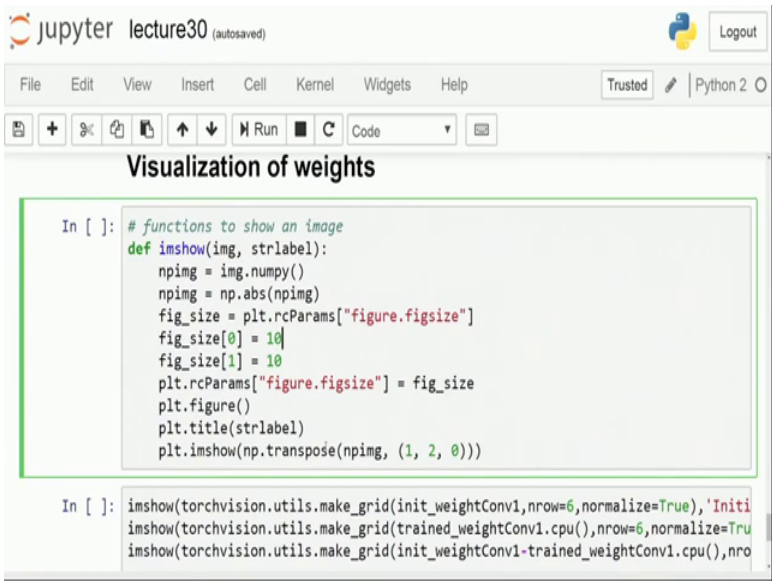
```
trained_weightConv1 = copy.deepcopy(net.conv1.weight.data)
trained_weightConv2 = copy.deepcopy(net.conv2.weight.data)
```

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)))
```

So, once that is done the next part is to actually look into my weights of what it was before my training and after my training. So, we make use of a similar kind of a argument and objective as we had in the earlier case, the only difference over here which would come down is in the earlier case, your first layer of weights was what is connecting a grayscale onto my number of channels and the number of it is over there is equal to the total number of channels which comes down over here.

(Refer Slide Time: 15:19)


jupyter lecture30 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

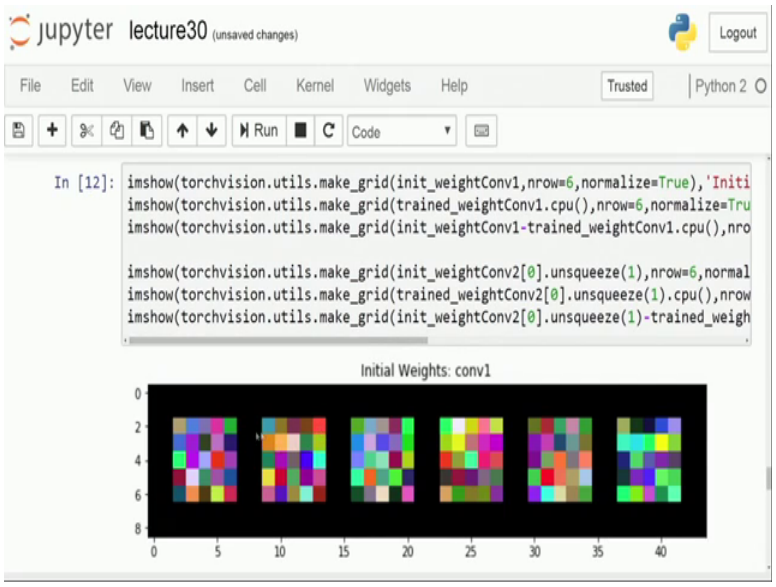
Visualization of weights

```
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 [ ]: 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
```

Now, here I have an rgb image 3 channel image coming down because, my total number of say planes on my convolution kernel is also equal to the total number of planes in my input image of the total number of channels over there and, that is why you would be seeing down that in your first convolution kernel in the earlier case while, you were just in grayscale bits here you are going to see.

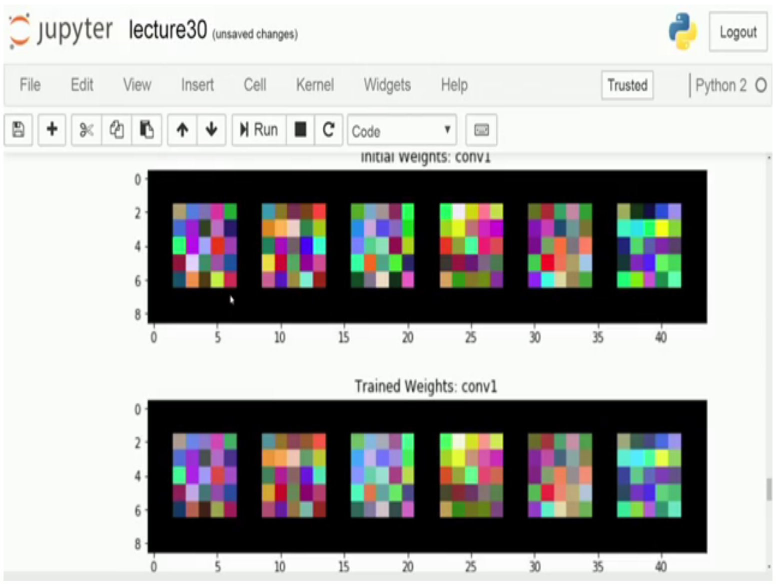
(Refer Slide Time: 15:35)


jupyter lecture30 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code
Initial Weights: conv1
0
2
4
6
8
0
5
10
15
20
25
30
35
40

```
In [12]: 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
```

Now, in color weights because each, there is a weight associated with the red channel, with the green channel, blue channel and this is a composite of each of them which comes up, there are 6 colonels in the first convolution layer. So, these correspond to the 6 colors each of a size of 5 cross 5 1 2 3 4 5, 1 2 3 4 5. So, it is a 5 cross 5 which comes down over here.

(Refer Slide Time: 15:59)

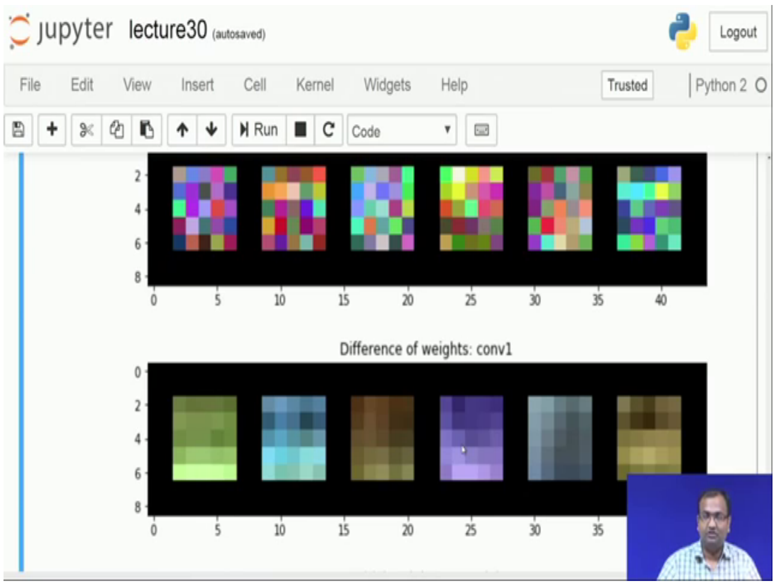


jupyter lecture30 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code
Initial Weights: conv1
Trained Weights: conv1

The next part is this is the weight basically these are all the weights which are there before the training and these are the weights which the kernel takes in after the training. And it looks more of like a chess board, or a very decorated piece of chess board and, then nothing much to convey down you do not see gradients, or anything coming down, but then you also need to keep in mind that the network as such has not come down to a great performance it is not even barely 50 percent on the accuracy side of it.

However, it is being much better than a random chance because, if you had to classify down ten different classes, then the random chance is one tenth or point one sorry sorry ten percent is the random chance and that would come down to an accuracy of 0.1 actually. So, instead of that this standing at 0.45, or more than that is actually a much better performance than just a random guess.

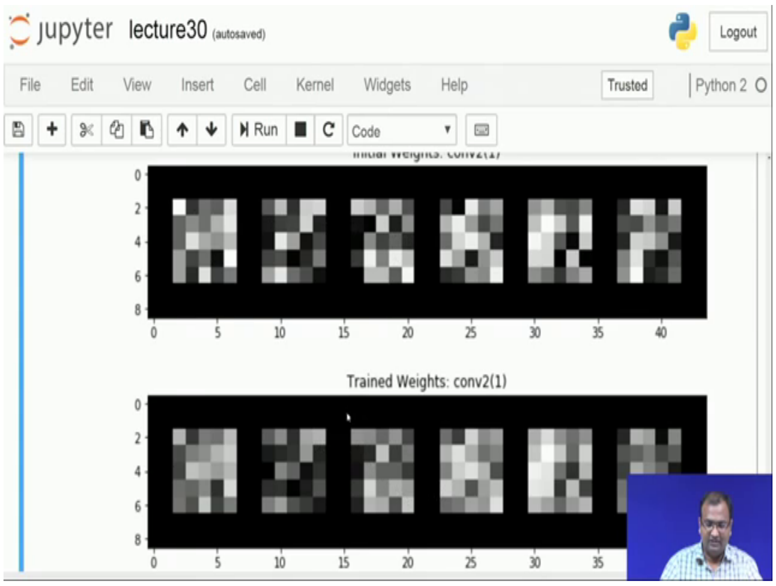
(Refer Slide Time: 16:52)


jupyter lecture30 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code
Difference of weights: conv1

So, anyways these weights are not yet trained, though if you look into the difference you would feel that there are gradients which are changing now. So, there are changes in the gradients in the green channel, there are changes ingredients in and then blue. So, which is basically one of the other channels other than green which is present, now here do you see that there is some sort of a change in the brown shade. So, it means that there has been changes in red, green and blue all of them taken down and most predominately it is basically a green and red where the change is coming down.

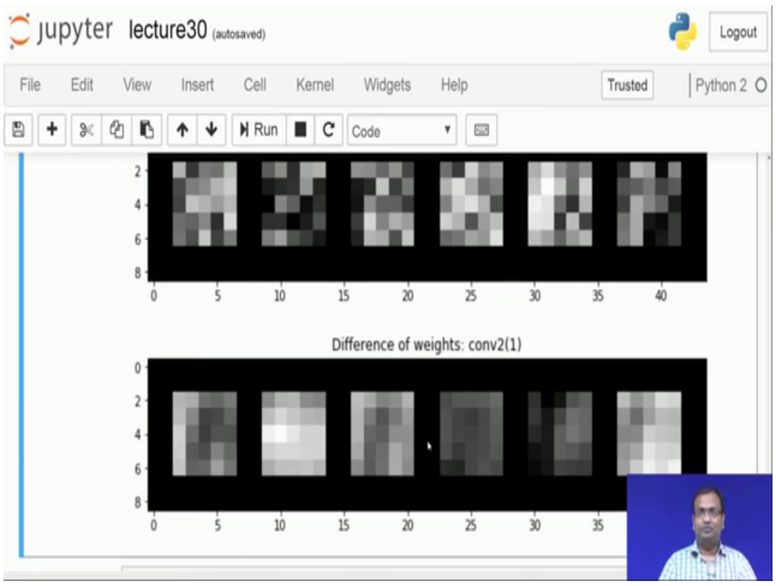
So, these look like shades of purple which is where the differences are in the plains of blue and red, this is where it is more of like a mix of green and blue, where the changes are coming down and again you have your brown. So, I am going back to your basic knowledge of colors within digital image processing, you can pretty much make impression of where which channels are getting changed over there and, you see the changes coming down in color because your images and objects on the side of in color.

(Refer Slide Time: 17:46)


jupyter lecture30 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code
Trained Weights: conv2(1)

However; when I am trying to look into my second layer over there so, I am just taking down my first kernel of convolution and then try to look into all the 6 channels over there. So, I see each of my 6 channels which are which connect down my output of the first convolution layers and look over there. So, each is a 5 cross 5 in the same logic and this is just my first convolution kernel on the second one, if I look between the weights of what is when it is not train versus, when it is end there a minor amount of changes which I would see and; however, on the difference I do see that there are gradients which are sitting down over here.

(Refer Slide Time: 18:20)


jupyter lecture30 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code
Difference of weights: conv2(1)

So, this is what comes down as we keep on moving towards color images. So, now that is as in with auto encoders, when I was explaining you for grayscale images to color images and there was one very immense important point which was made down is that, there is technically at this point of time no solution given down or there is no definite answer of when it will converge what kind of an architecture would come down as the best one, that is that is a still a challenge to be solved out over here.

And so, you see that in over here as well, it is the same kind of an architecture with the similar number of weights and everything which goes down the kind of performance it has on grayscale images were. So, the kind of performance it has one color images this is drastically different and one of the reasons is on your grayscale images of MNIST you had a much conformal way of representing the data, it was much easier to learn on very definitive patterns and that is why it was able to classify them in a much better way whereas, if you go down to your color images, then the way in which it would be learnt be able to learn down some discriminative patterns, which are of really importance and would make down, since is much low and for that reason it takes it longer and longer to learn it out.

So, if you keep on running this code over a larger number of iterations say just change down; go up over here and, where I have my number of iterations. If you just make and change this to hundred or maybe 1000, you would be seeing that this comes down to a much conformal way of decent amount of accuracy and a much lower loss. So, that is where we come to an end with writing non very plain and simple convolutional neural networks. So, stay tuned for the next ones where we start digging into deeper neural networks and, why they are called as deeper neural networks and more computational complexity issues around with them. So, till then.

Thanks and stay tuned.