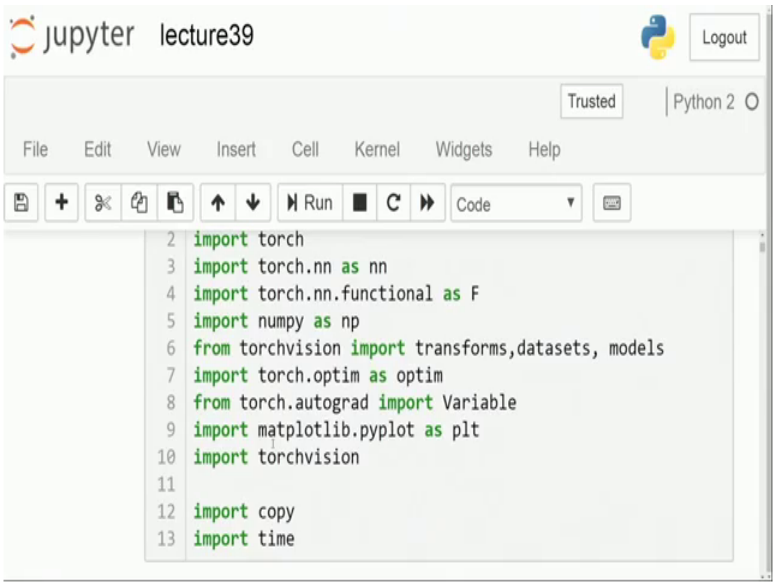
Deep Learning for Visual Computing

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

Lecture - 39
ResNet

Welcome in the last lecture. We had studied about two interesting architectures and they were much deeper and then you could go up to 100s or 200s of layers over there. One of them was residual network the other one is what is called as a densenet or a densely connected dense residually connected networks over there. So, today we are going to do hands on for the first perient of it which is called as densenet or very deep network with the residual connection over there.

(Refer Slide Time: 00:43)

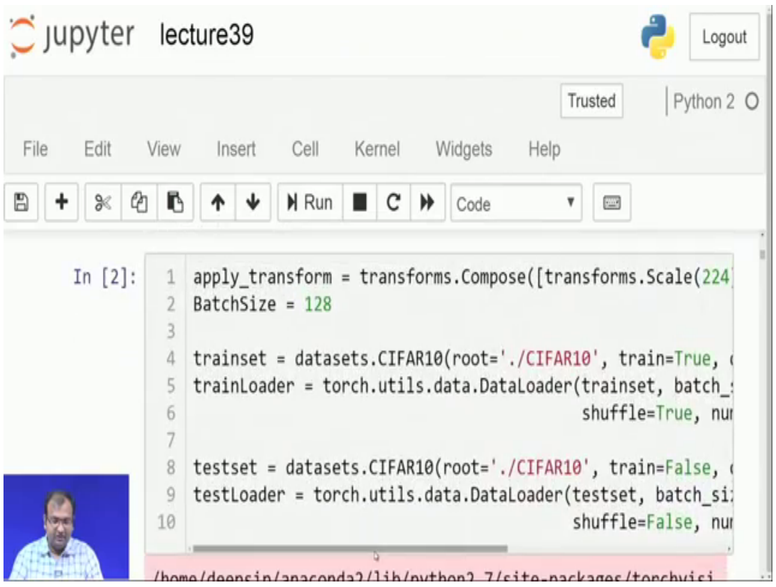


```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
from torchvision import transforms,datasets, models
import torch.optim as optim
from torch.autograd import Variable
import matplotlib.pyplot as plt
import torchvision

import copy
import time
```

Now its not much of a change as far as the first part of it is concerned and these are also some things which take good amount of time. So, I am just going to show down a notebook, which has most of the things which are re compiled over there and we do a simple walkthrough of it.

Now, over here the first part is pretty simple, which is your just the header files or the initial ones which you need to take in, and then next comes down your data over there.

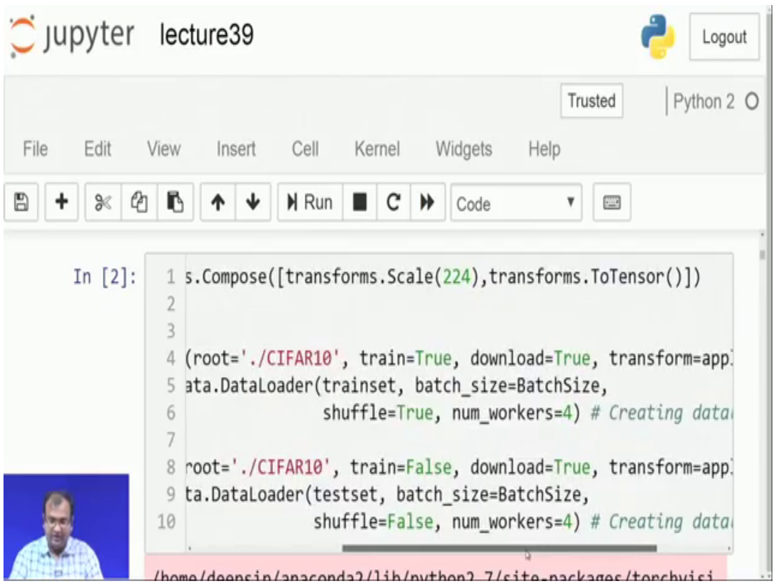
(Refer Slide Time: 01:05)



```
In [2]:
apply_transform = transforms.Compose([transforms.Scale(224
BatchSize = 128

trainset = datasets.CIFAR10(root='./CIFAR10', train=True, (
trainLoader = torch.utils.data.DataLoader(trainset, batch_:
                                          shuffle=True, nur

testset = datasets.CIFAR10(root='./CIFAR10', train=False, (
testLoader = torch.utils.data.DataLoader(testset, batch_si:
                                         shuffle=False, nur
/home/deepsin/anaconda2/lib/python2.7/site-packages/torchvisi
```

Now, this particular network quite different from your unlike your googlenet.

(Refer Slide Time: 01:09)

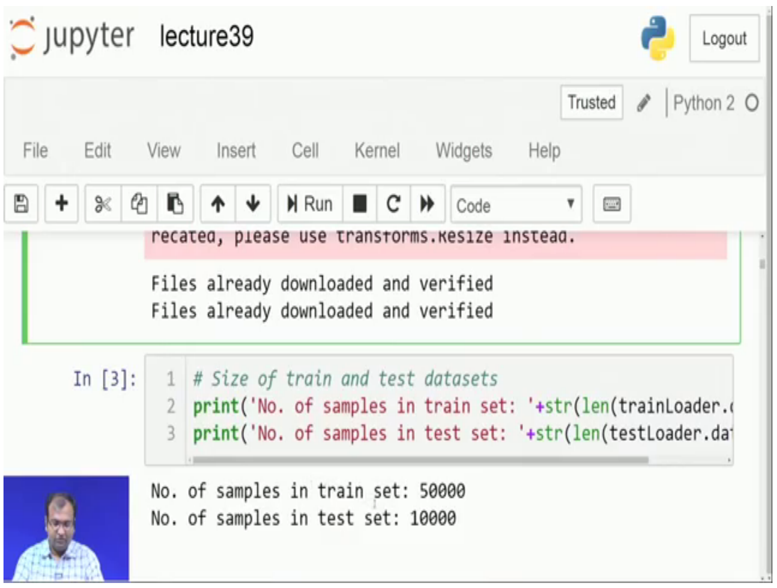


```
In [2]:
s.Compose([transforms.Scale(224),transforms.ToTensor()])


(root='./CIFAR10', train=True, download=True, transform=app]
ata.DataLoader(trainset, batch_size=BatchSize,
               shuffle=True, num_workers=4) # Creating data

root='./CIFAR10', train=False, download=True, transform=app]
ta.DataLoader(testset, batch_size=BatchSize,
              shuffle=False, num_workers=4) # Creating data
/home/deepsin/anaconda2/lib/python2.7/site-packages/torchvisi
```

Which would have needed you to take down 200 and then 229 cross 229 sized images over there; so here we take down smaller ones which are 224 cross 224 and this is a network, which has been trained to stick down exactly two my image that kind of models and imagine it kind of sizes ok. Now here fortunately we could take down a larger batch size and that is all dependent on how much of intermediate data my model is handling. So, based on and then that is a pretty easy calculation based on what we had done in the earlier lecture itself. Now you need to work it out around over there based on how much your available ram is there. So, while in googlenet we had to stick down to a much smaller batch size of 32 based on the particular card I was using over here.

So, for me with this given with the same card and the same kind of a system architecture, I am able to get down more number of images. And here I am pretty convenient taking down 128 images into one single batch. Now the rest part is quite simple. So, instead of doing it with image net, we are doing with a smaller dataset called a c 5 and this is just a 10 class classification problem. Now based on this 10 class classification problem, accordingly we will also be modifying the network architecture which we download from our model zoo available with us. Now that is this for this first part of it which is quite straightforward and clear there is nothing much to do.

(Refer Slide Time: 02:39)



```
recated, please use transforms.Resize instead.

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

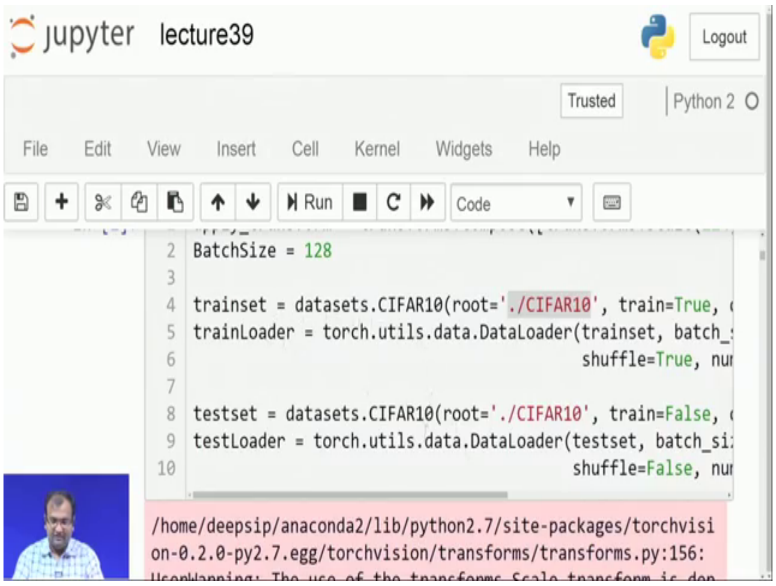
In [3]:

```
# Size of train and test datasets
print('No. of samples in train set: '+str(len(trainLoader.
print('No. of samples in test set: '+str(len(testLoader.da
```

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

Now, the next is to look into whether the data has been loaded down and yes the same way that we had divided 50,000 for our training set.

(Refer Slide Time: 02:46)

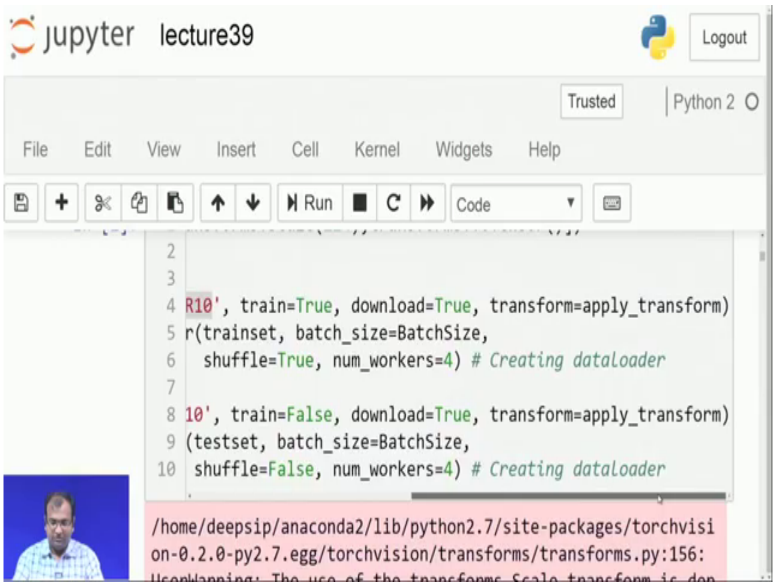


```
BatchSize = 128

trainset = datasets.CIFAR10(root='./CIFAR10', train=True, 
trainLoader = torch.utils.data.DataLoader(trainset, batch_
                                          shuffle=True, nu

testset = datasets.CIFAR10(root='./CIFAR10', train=False, 
testLoader = torch.utils.data.DataLoader(testset, batch_si
                                         shuffle=False, nu
```

```
/home/deepsip/anaconda2/lib/python2.7/site-packages/torchvisi
on-0.2.0-py2.7.egg/torchvision/transforms/transforms.py:156:
```

And then 10,000 for our testing said the same thing comes out. So, so that is pretty straightforward.

(Refer Slide Time: 03:49)

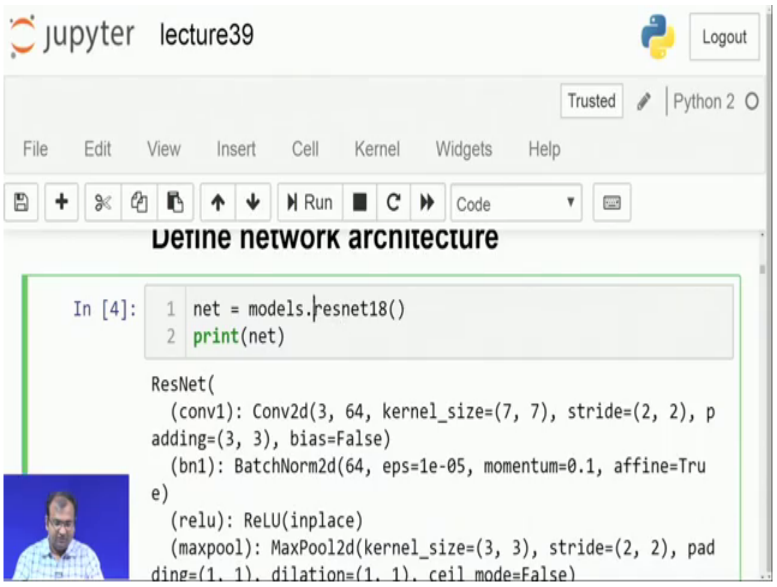


```
R10', train=True, download=True, transform=apply_transform)
r(trainset, batch_size=BatchSize,
  shuffle=True, num_workers=4) # Creating dataloader

10', train=False, download=True, transform=apply_transform)
(testset, batch_size=BatchSize,
 shuffle=False, num_workers=4) # Creating dataloader
```

```
/home/deepsip/anaconda2/lib/python2.7/site-packages/torchvisi
on-0.2.0-py2.7.egg/torchvision/transforms/transforms.py:156:
```

Now, here what we do after that is we just get down our models and this is my resnet 18.

(Refer Slide Time: 02:55)



Define network architecture

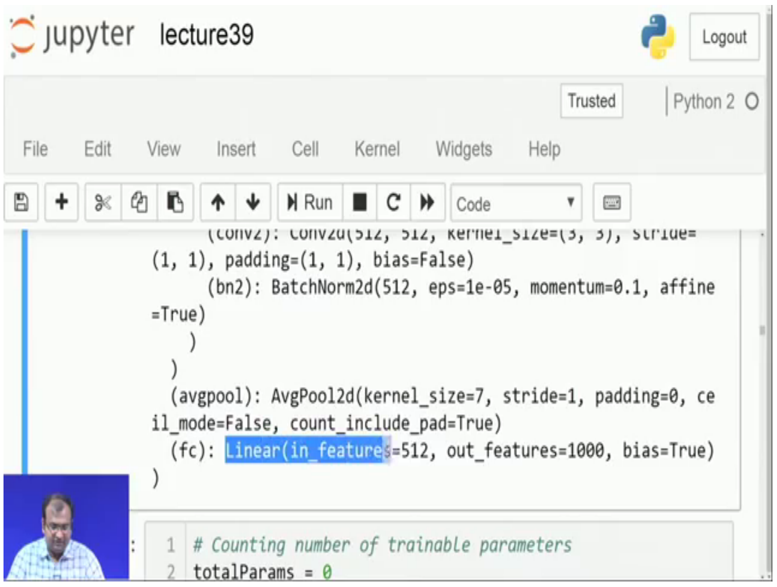
```
In [4]: net = models.resnet18()
        print(net)
```

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), p
adding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=Tru
e)
  (relu): ReLU(inplace)
  (maxpool): MaxPool2d(kernel_size=(3, 3), stride=(2, 2), pad
ding=(1, 1), dilation=(1, 1), ceil_mode=False)
```

So, this residual network is just 18 layers deep and I am choosing down only a very smaller one. You can go down 251 layers as well pretty much, but then you need to keep one thing in mind that your batch size will reduce and we will take much longer time to a finish off each of these epochs. Now in order to avoid any of those problems over there, I am just using a smaller model over here and this is just residual network 18. Now there is also another fact that I am not trying to discriminate between multiple number of classes. So, an earlier residual network which was really deep there is something which was done to actually classify between 1000 classes of objects over there. Now I am choosing down a very smaller one which is just 10 classes of objects and now just because of using 10 classes of objects I can have this liberty they are actually good and choose do not know much smaller size network.

Now, that is what I am doing with choosing down resnet 18 and this is what you can look down on the whole thing. Now over here we do not make actually much of a change to the whole network and you do not have any issues of an auxilary arm or something coming down over there. Now what does definitely come to play is that all of your basic blocks and everything being put down together.

(Refer Slide Time: 04:12)



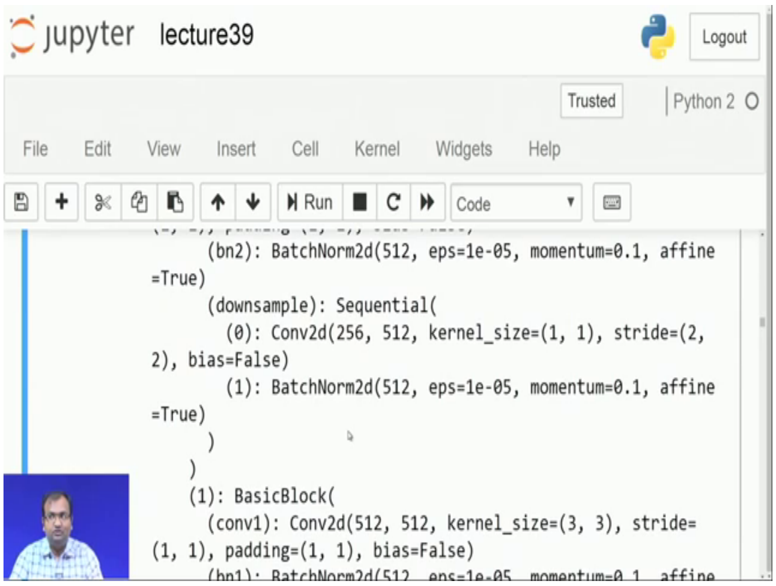
```
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=
(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine
=True)
      )
    )
    (avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0, ce
il_mode=False, count_include_pad=True)
    (fc): Linear(in_features=512, out_features=1000, bias=True)
  )
```

```
# Counting number of trainable parameters
totalParams = 0
```

The next interesting part is that your final one is something which gets down 512 as a linearized feature vectors and the terminal lane. And then this 512 are what are connected down as a fully connected network on 2000 years. You now this is the part which we need to modify because we did choose that we would be going down with just a 10 class classification problem and not a 1000 class classification problem. So, this is the only change which comes down and quite unlike the googlenet which we had done in the earlier session is where you had these auxiliary arm.

So, you also had to modify the auxilary arm, you do not have this issue over here because there is nothing of an auxilary arm. But given that the main point which comes down is that you have these residual connections as well now and just by virtue of these residual connections which are present over there.

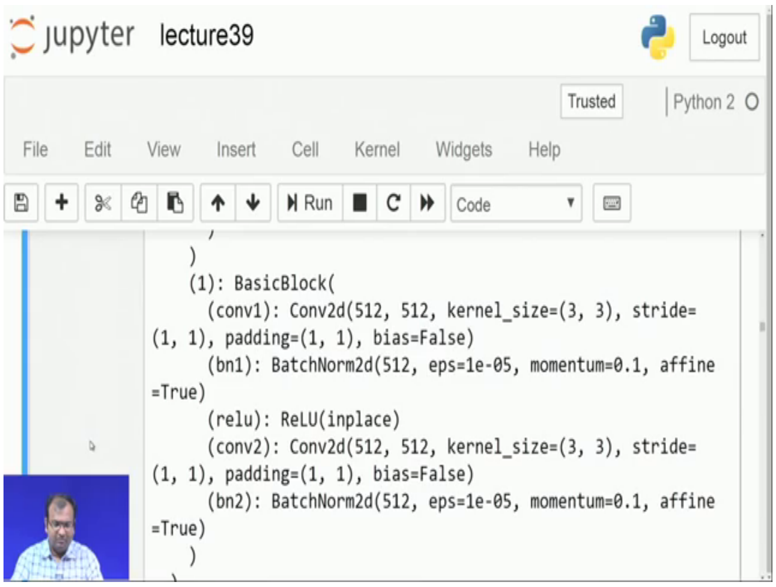
(Refer Slide Time: 04:54)



```
            (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine
=True)
            (downsample): Sequential(
              (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2,
2), bias=False)
              (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine
=True)
            )
          )
          (1): BasicBlock(
            (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=
(1, 1), padding=(1, 1), bias=False)
```

So, your gradient is actually back propagating much easily and because of this easy back propagation of the gradient, we do not need to worry about whether there would be a vanishing gradient at some point of time as we are traversing down the depth over there.

(Refer Slide Time: 05:12)

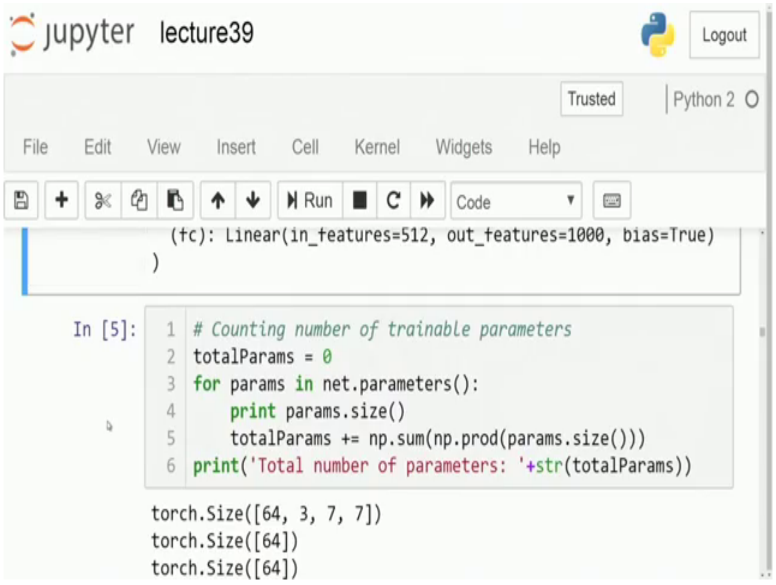


```
          )
          (1): BasicBlock(
            (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=
(1, 1), padding=(1, 1), bias=False)
            (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine
=True)
            (relu): ReLU(inplace)
            (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=
(1, 1), padding=(1, 1), bias=False)
            (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine
=True)
          )
```

Now, that is that is pretty much simple and easily done, what you need to keep in mind is that because of these residual connections, its again made down in the same kind of a tabular form of connecting.

So, you first build up the small blocks over there for each of them, and then you drop down residual connection and keep on doing a parallel connection and build it up. So, this is quite similar to how you were doing it down for your googlenets now since we are not doing much on the data structure of these neural networks as such to do. So, and then that is pretty much well documented within pie torch communities documentation. So, I will not be spending much of time over there, but my most important factors to discuss are more of related to what happens within the learning dynamics, what you need to make a change from adapting a model from an existing one to your particular one and further on as we keep on going over there, and what are the properties which does come to impact.

(Refer Slide Time: 06:02)



```
    (fc): Linear(in_features=512, out_features=1000, bias=True)
)
```

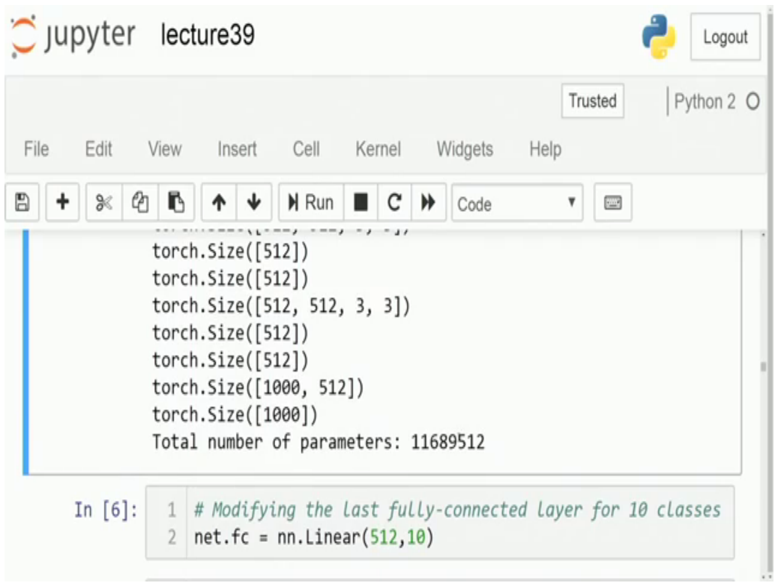
In [5]:

```
# Counting number of trainable parameters
totalParams = 0
for params in net.parameters():
    print params.size()
    totalParams += np.sum(np.prod(params.size()))
print('Total number of parameters: '+str(totalParams))
```

```
torch.Size([64, 3, 7, 7])
torch.Size([64])
torch.Size([64])
```

Now, once that is done, we look into the total number of parameters and then if I go through them you would see that as a total calculation, this particular model is something which has eleven million parameters over there.

(Refer Slide Time: 06:11)



```
torch.Size([512])
torch.Size([512])
torch.Size([512, 512, 3, 3])
torch.Size([512])
torch.Size([512])
torch.Size([1000, 512])
torch.Size([1000])
Total number of parameters: 11689512
```

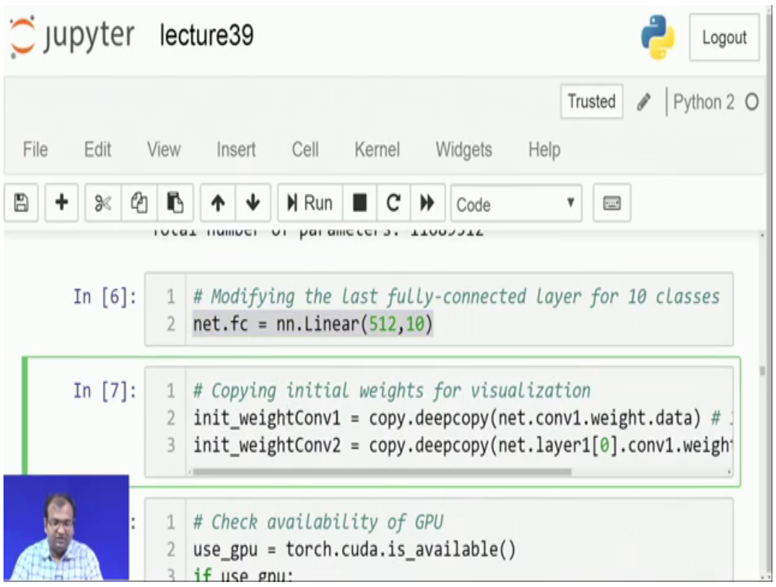
```
In [6]: # Modifying the last fully-connected layer for 10 classes
        net.fc = nn.Linear(512,10)
```

Now, this is definitely a model which is almost the same size as in a googlenet, but it does have much lesser number of parameters and by virtue of a lesser number of parameters you have a model which takes less space to load. By virtue of lesser number of parameters you are intermediate operating points also consume lesser amount of memory that is that is something which we had done for one of our networks earlier. Now just because of this sure interesting aspect and attribute present down within the model, you can now actually feed down much larger batch sizes. Now one thing you know is that if you are feeding on much larger batch sizes and the total number of back propagations over there is; obviously, decreasing. And once you keep on decreasing the number of back propagation. So, your compute is roughly definitely going to become faster and faster over there.

So, one is the compute which is steady which is for the forward pass and that is dependent on the order of the sample size within your training set over there. Well then your back prop is always dependent on your batch size. So, larger the batch size is the lesser number of larger is the batch size the lesser is the number of batches which fix in over there. So, the lesser is the number of back propagation you are going to do and that is something which does the definitely come to play a very important role now here this is what we have gone down and looked down. So, this particular network has about 11.7 million parameters. So, when we had and in the earlier theory, when we were comparing down our resnet versus densenet and there was one of these plots which was plotting down the total number of parameters versus what is the accuracy saturation accuracy achieved over there. So, you did see that densenets for the same kind of a depth have almost half the number of parameters to come down to the same kind of an accuracy over there.

So, this is something which we will do in the next class when we compare down with densenet and do a practical hands on with densenet is when we get down the parametric calculation over there as well. Now the thing which you need to modify for this network is your last layer, which is the f c. Now if we go over here this is my f c layer and this f c layer has to be changed from this kind of a linear structure to something which connects 512 to just 10 nodes over there and nothing more than that. So, that is the change which we bring over here. So, its a linear connection from 512 to just 10 nodes and that is the modification, which we do for a residual network to be used for a c power 10 class classification problem.

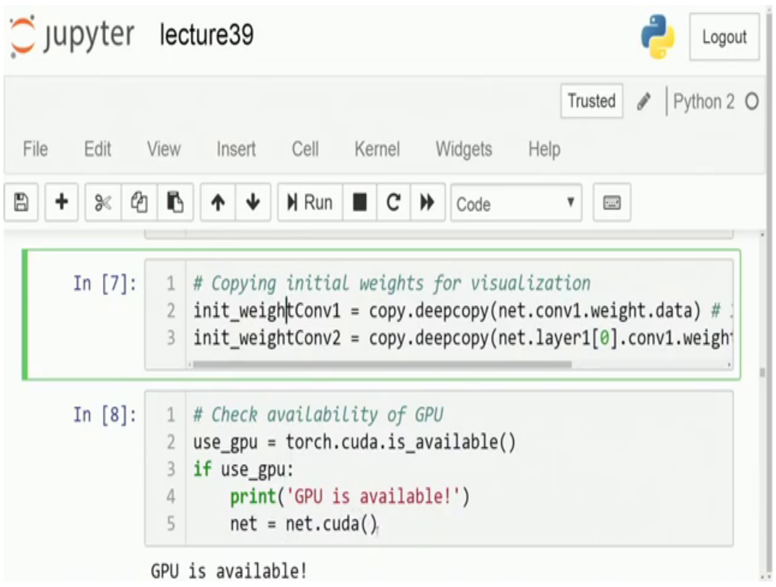
(Refer Slide Time: 08:36)



```
In [6]:  # Modifying the last fully-connected layer for 10 classes
         net.fc = nn.Linear(512,10)

In [7]:  # Copying initial weights for visualization
         init_weightConv1 = copy.deepcopy(net.conv1.weight.data) #
         init_weightConv2 = copy.deepcopy(net.layer1[0].conv1.weigh

         # Check availability of GPU
         use_gpu = torch.cuda.is_available()
         if use gpu:
```

Now once having done that the next part is just to copy down all of my weights and keep it. So, these are my initial weights now I will look down what happens after my whole training.

(Refer Slide Time: 08:43)



```
In [7]:
# Copying initial weights for visualization
init_weightConv1 = copy.deepcopy(net.conv1.weight.data) #
init_weightConv2 = copy.deepcopy(net.layer1[0].conv1.weigh
```

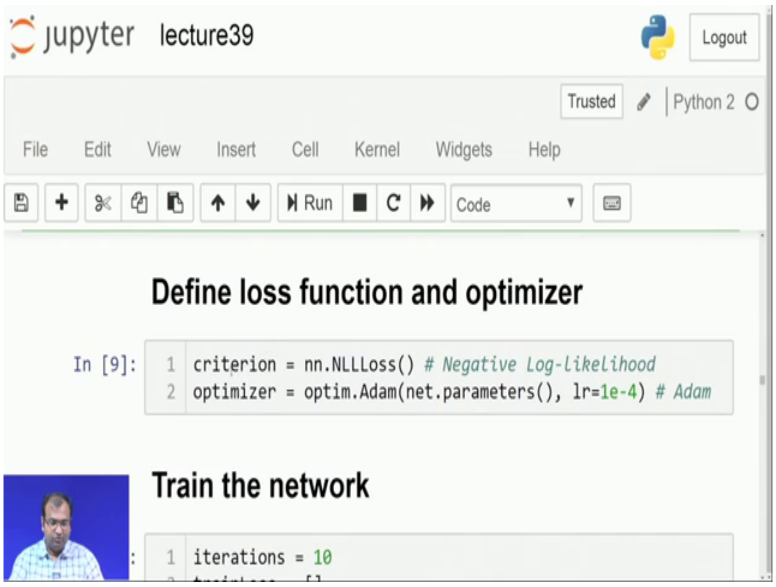
```
In [8]:
# Check availability of GPU
use_gpu = torch.cuda.is_available()
if use_gpu:
    print('GPU is available!')
    net = net.cuda()
```

```
GPU is available!
```

The next is pretty straightforward and simple as we had done in the earlier case is to check down whether I have a g p u available with me and whether cuda libraries and resources are all set and running.

Now, if I have that then I can just keep on using my cuda for acceleration.

(Refer Slide Time: 08:59)



Define loss function and optimizer

```
In [9]:
criterion = nn.NLLLoss() # Negative Log-Likelihood
optimizer = optim.Adam(net.parameters(), lr=1e-4) # Adam
```

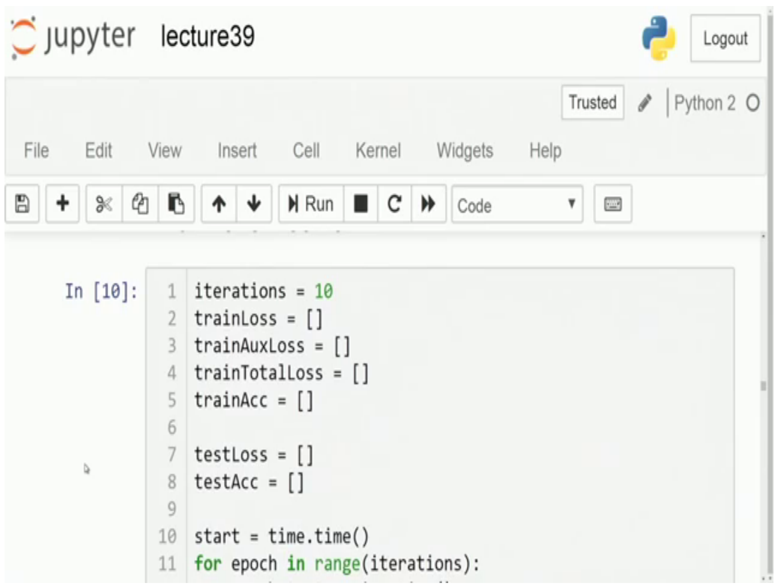
Train the network

```
iterations = 10
```

Now, once that is done, next is to look into my criteria it stays the same as in for classification, we choose to stick down with negative log likelihood classification criteria and then with adam as my optimizer. Now in the earlier classes we had discussed about different kinds of optimizers and their features and the different attributes, which they play down and what we did figure out that adaptive momentum and also by a experiment, we have seen down that adaptive momentum and a general operating conditions is something which would take a tad bit longer to actually optimize and give me the results. But then its something which would definitely guarantee convergence at much lesser number of epochs as well as if you take the total time consumed.

So, your pour epoch time is increasing because your per batch time is increasing on view of the adam optimizer over there. However, this net product of total number of epochs into the time taken per epoch that is much lesser when it comes down to adam in order to come down to a saturation point. So, that is that is really interesting because in terms of how many minutes or hours or normal human time and c p u time it consumes that is being brought down significantly by use of adam. Now once this part is done next is to look into the network and we start training the network.

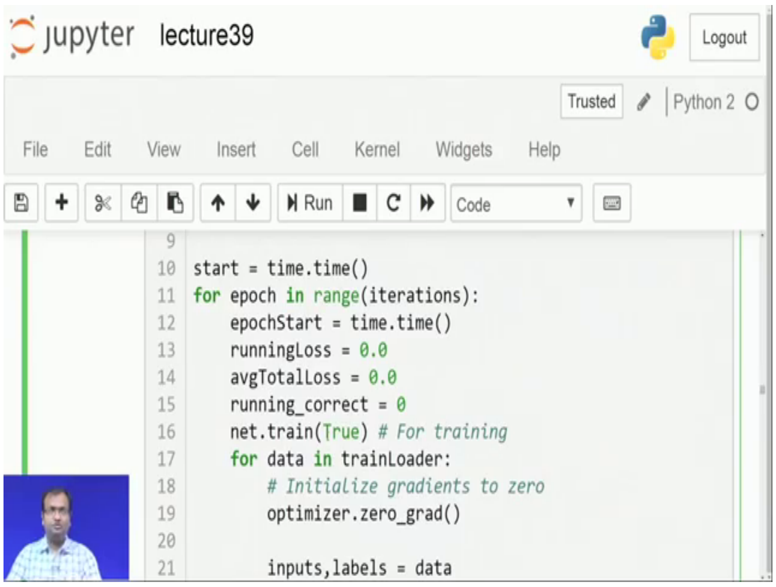
(Refer Slide Time: 10:09)



In [10]:
```
iterations = 10
trainLoss = []
trainAuxLoss = []
trainTotalLoss = []
trainAcc = []

testLoss = []
testAcc = []

start = time.time()
for epoch in range(iterations):
```

So, here what we do is go down with the same plain old rule of training it down just for 10 epochs, I do not drink it more than 10 epochs it does not play a role over here and then although this does not take much amount of time, this network does train much faster than googlenet which otherwise would have taken down about 8 minutes close to 8 minutes to do it. So, here it goes wrong about belly between something between 1 and 2 minutes over there. Now within each of my epoch now the difference which comes in that here this is just one single tapped out network which means that the classification and the losses, which you get down is only at the end part of the network its no more in between.

(Refer Slide Time: 10:43)

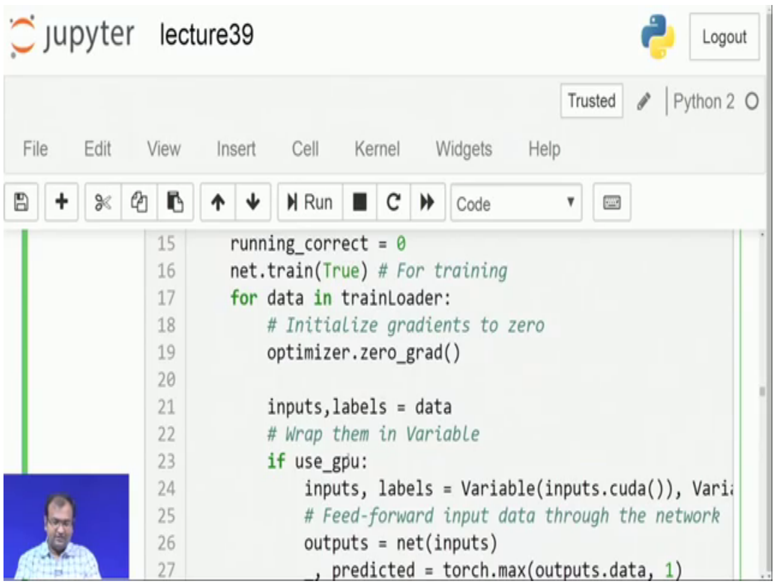


```

start = time.time()
for epoch in range(iterations):
    epochStart = time.time()
    runningLoss = 0.0
    avgTotalLoss = 0.0
    running_correct = 0
    net.train(True) # For training
    for data in trainLoader:
        # Initialize gradients to zero
        optimizer.zero_grad()

        inputs,labels = data
```

So, in googlenet you had these auxiliary ones auxiliary arms coming down and for that reason you also had to get down the losses, computed out of your auxiliary arm. And you also had needed to a back propagation with the losses being fed down over there.

Now, we do not need to do any of those for our case over here, now here what you are just going to do is the plain simple calculate out the final loss over there and then do the back pope.

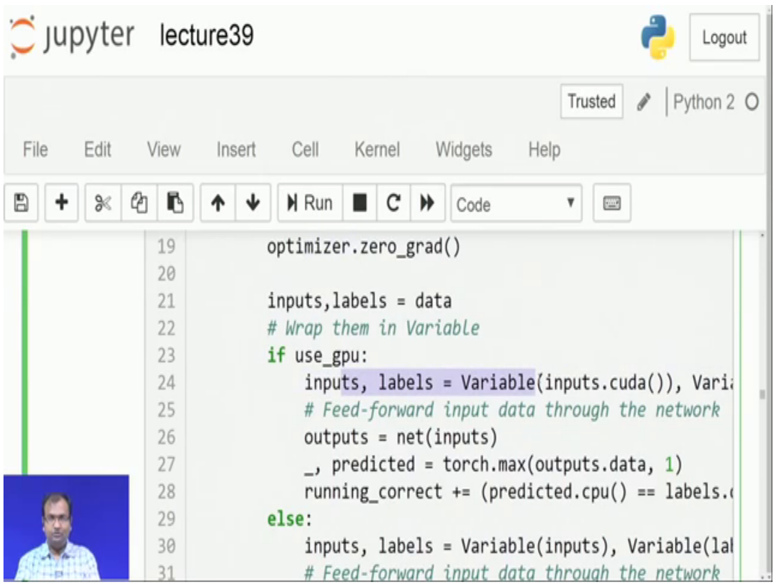
(Refer Slide Time: 11:13)



```
    running_correct = 0
    net.train(True) # For training
    for data in trainLoader:
        # Initialize gradients to zero
        optimizer.zero_grad()

        inputs,labels = data
        # Wrap them in Variable
        if use_gpu:
            inputs, labels = Variable(inputs.cuda()), Vari
            # Feed-forward input data through the network
            outputs = net(inputs)
            _, predicted = torch.max(outputs.data, 1)
```

Now, within each of these batches which comes down over there, you for zero down your optimizes gradient and everything.

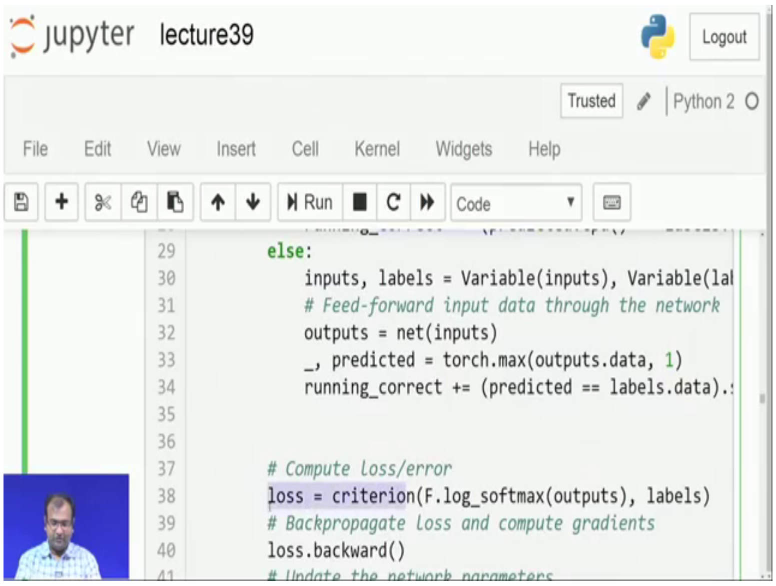
(Refer Slide Time: 11:20)



```
        optimizer.zero_grad()

        inputs,labels = data
        # Wrap them in Variable
        if use_gpu:
            inputs, labels = Variable(inputs.cuda()), Vari
            # Feed-forward input data through the network
            outputs = net(inputs)
            _, predicted = torch.max(outputs.data, 1)
            running_correct += (predicted.cpu() == labels.
        else:
            inputs, labels = Variable(inputs), Variable(la
            # Feed-forward input data through the network
```

And then you convert your if there is a g p u or available then you convert your variables into g p u and then type casted as a variable. So, that you can you can do your back propagation operations over there. Next is you find your feed forward and get your output then you find out what is your predicted and whether your predicted is correct or not, and then this would help you in getting down your error over here ok.

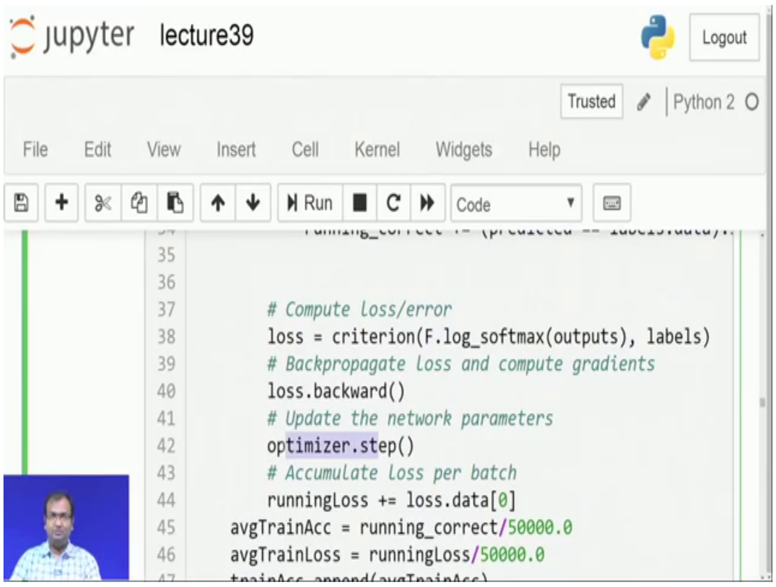
(Refer Slide Time: 11:40)



```
        else:
            inputs, labels = Variable(inputs), Variable(la
            # Feed-forward input data through the network
            outputs = net(inputs)
            _, predicted = torch.max(outputs.data, 1)
            running_correct += (predicted == labels.data).


        # Compute Loss/error
        loss = criterion(F.log_softmax(outputs), labels)
        # Backpropagate loss and compute gradients
        loss.backward()
```

Now, once I have all of this done and my errors computed, and then done then what I do is I do a back propagation of my loss over there and the optimizer comes into play in steps.

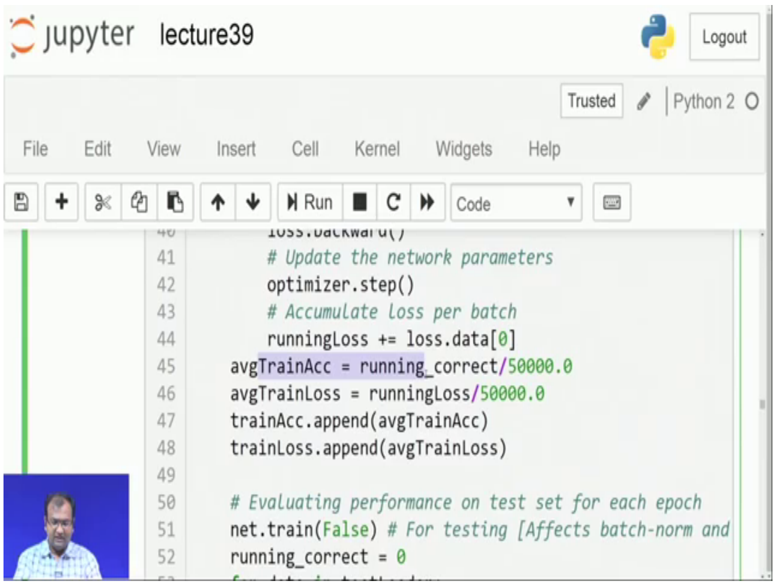
(Refer Slide Time: 11:53)



```


        # Compute Loss/error
        loss = criterion(F.log_softmax(outputs), labels)
        # Backpropagate loss and compute gradients
        loss.backward()
        # Update the network parameters
        optimizer.step()
        # Accumulate loss per batch
        runningLoss += loss.data[0]
    avgTrainAcc = running_correct/50000.0
    avgTrainLoss = runningLoss/50000.0
```

So, whenever I do a step it means that the optimizer is being solved over there and my weights are getting updated. So, this is where my weight update happens and then I have my running losses which I just calculated and stored.

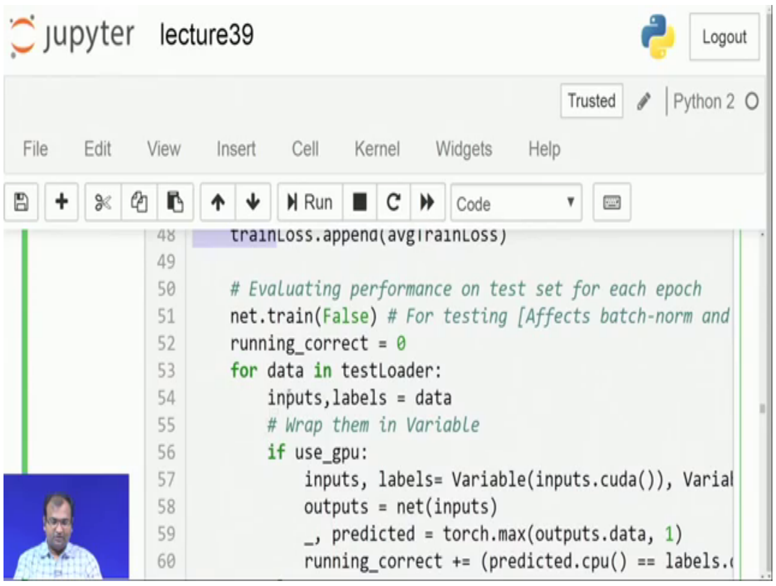
(Refer Slide Time: 12:04)



```
            loss.backward()
            # Update the network parameters
            optimizer.step()
            # Accumulate loss per batch
            runningLoss += loss.data[0]
        avgTrainAcc = running_correct/50000.0
        avgTrainLoss = runningLoss/50000.0
        trainAcc.append(avgTrainAcc)
        trainLoss.append(avgTrainLoss)

        # Evaluating performance on test set for each epoch
        net.train(False) # For testing [Affects batch-norm and
        running_correct = 0
```

Now, within each like each epoch over there I am passing down all my training samples, and I have in total 50,000 training samples over there. So, if I need to find out my accuracy or my loss, then I will have to take some sort of a average over there and that is what I am actually doing over there. Now once that is done, then train a c and train loss these are the two different arrays which I just being created dynamically. So, one array has an entry for each epoch over there.

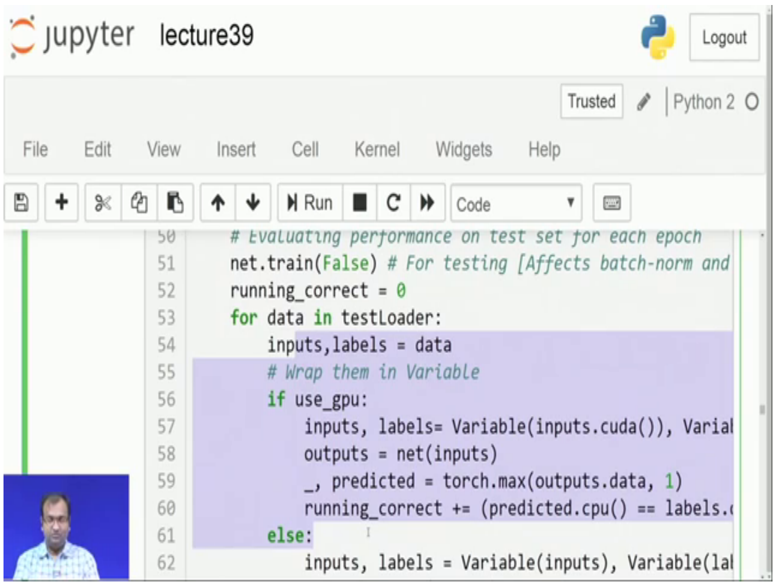
(Refer Slide Time: 12:36)



```
        trainLoss.append(avgTrainLoss)

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

Now, once that is done the next part is where we need to get down my next part of it is running, which is my validation over there.

(Refer Slide Time: 12:42)

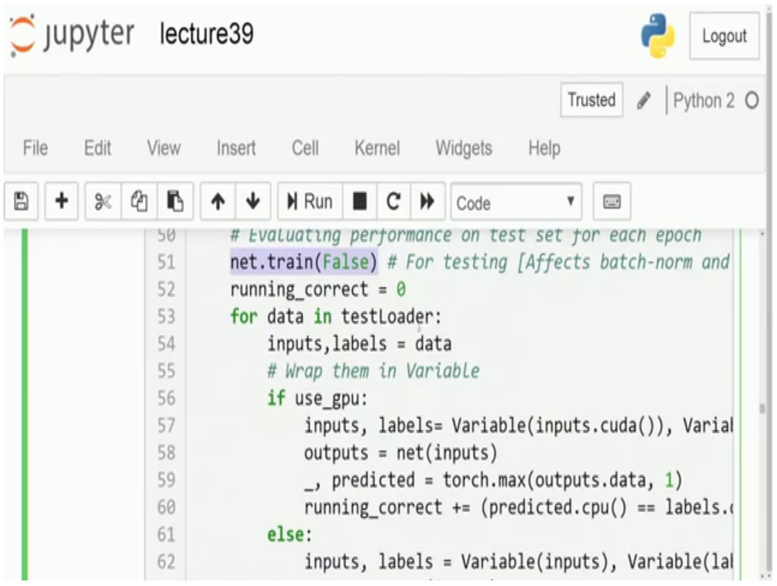


```
    # Evaluating performance on test set for each epoch
    net.train(False) # For testing [Affects batch-norm and
    running_correct = 0
    for data in testLoader:
        inputs,labels = data
        # Wrap them in Variable
        if use_gpu:
            inputs, labels= Variable(inputs.cuda()),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

Now, in validation what we need to keep down in mind is that one point which we said down over here is false, we just give an identifier that we are no more training the network and this is for the reason. So, is that your batch normalization does not come into play. So, typically when you are training a network you will always be trying to normalize in batch.

So, whenever you have a batch norm coming to place. So, there is a batch normalization taking place, but whenever you are doing a feed forward and just an inference in over there your batch like this may be different and based on that it will have a different dynamics coming out. So, the same sample if it is located at different batches based on what all other samples are located in the batches; if the other samples in the batch are changing, then the whole thing gets normalized along a batch and the response has a very skewed or behavior. So, in typically during infarencing we do not use batch normalization and that is just switched off. So, this is the very simple way of actually switching off all of my batch normalization issues ok.

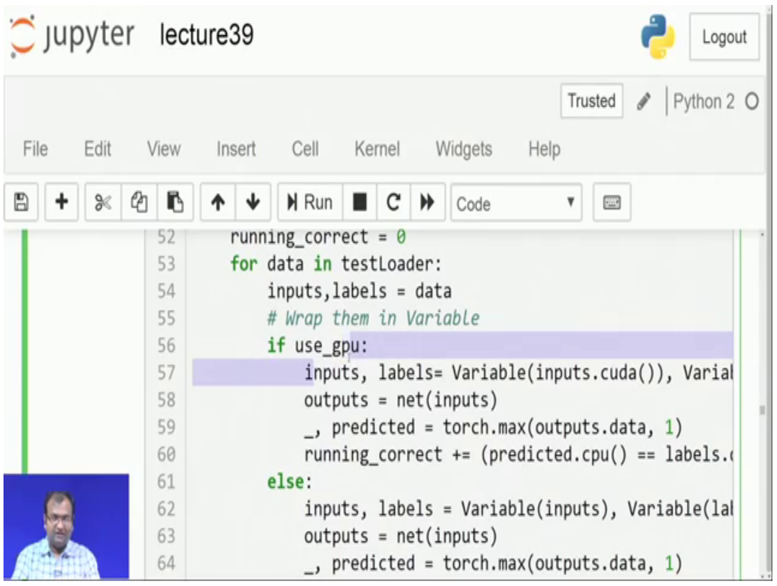
(Refer Slide Time: 13:31)


jupyter lecture39

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

Next what I do is just find out if my g p u is available.

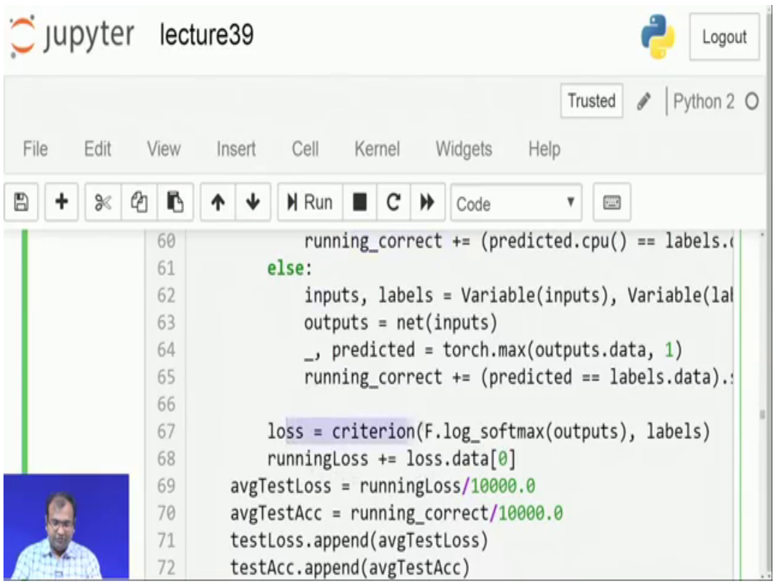
(Refer Slide Time: 13:39)


jupyter lecture39

```
running_correct = 0
for data in testLoader:
    inputs,labels = data
    # Wrap them in Variable
    if use_gpu:
        inputs, labels= Variable(inputs.cuda()), Varial
        outputs = net(inputs)
        _, predicted = torch.max(outputs.data, 1)
        running_correct += (predicted.cpu() == labels.(
    else:
        inputs, labels = Variable(inputs), Variable(lal
        outputs = net(inputs)
        _, predicted = torch.max(outputs.data, 1)
```

Then I typecast my data on to a cuda array, such that it resides on my g p u memory and my network can also work on my g p u and then I find out my outputs do a prediction over there and then find out whether its correct or not.

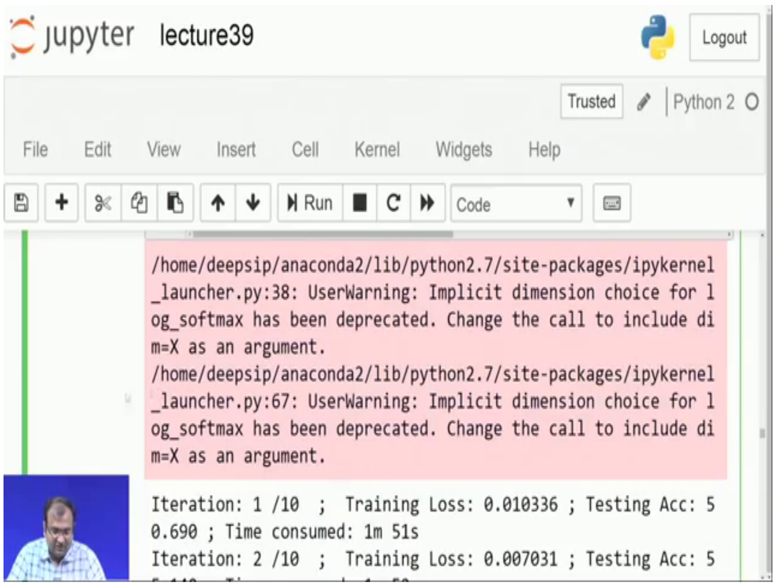
(Refer Slide Time: 13:55)



```
            running_correct += (predicted.cpu() == labels.(
        else:
            inputs, labels = Variable(inputs), Variable(lal
            outputs = net(inputs)
            _, predicted = torch.max(outputs.data, 1)
            running_correct += (predicted == labels.data).:

        loss = criterion(F.log_softmax(outputs), labels)
        runningLoss += loss.data[0]
    avgTestLoss = runningLoss/10000.0
    avgTestAcc = running_correct/10000.0
    testLoss.append(avgTestLoss)
    testAcc.append(avgTestAcc)
```

Now, that is what you would be doing down for your feed forward on the validation side of it, and then since there are 10 thousand samples present in my validation which get evaluated every epoch.

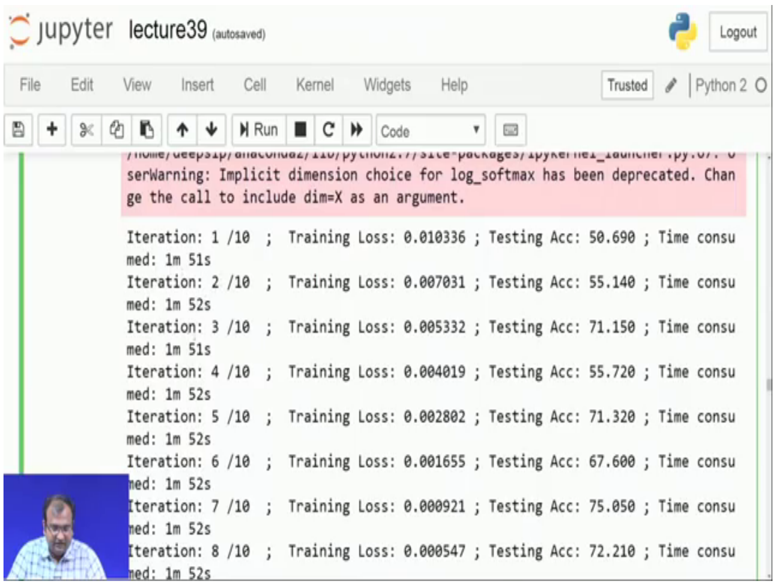
(Refer Slide Time: 14:09)



```
/home/deepsip/anaconda2/lib/python2.7/site-packages/ipykernel
_launcher.py:38: UserWarning: Implicit dimension choice for l
og_softmax has been deprecated. Change the call to include di
m=X as an argument.
/home/deepsip/anaconda2/lib/python2.7/site-packages/ipykernel
_launcher.py:67: UserWarning: Implicit dimension choice for l
og_softmax has been deprecated. Change the call to include di
m=X as an argument.

Iteration: 1 /10  ;  Training Loss: 0.010336 ; Testing Acc: 5
0.690 ; Time consumed: 1m 51s
Iteration: 2 /10  ;  Training Loss: 0.007031 ; Testing Acc: 5
```

So, I take an average over all the 10,000 samples the next part is pretty simple to plot it down and there is nothing much of a change.

(Refer Slide Time: 14:17)

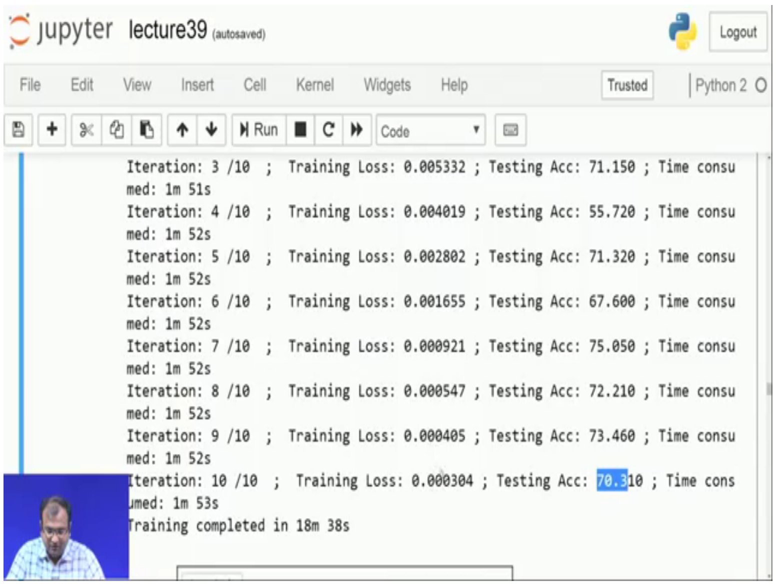

jupyter lecture39 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

```
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: 0.010336 ; Testing Acc: 50.690 ; Time consu
med: 1m 51s
Iteration: 2 /10  ;  Training Loss: 0.007031 ; Testing Acc: 55.140 ; Time consu
med: 1m 52s
Iteration: 3 /10  ;  Training Loss: 0.005332 ; Testing Acc: 71.150 ; Time consu
med: 1m 51s
Iteration: 4 /10  ;  Training Loss: 0.004019 ; Testing Acc: 55.720 ; Time consu
med: 1m 52s
Iteration: 5 /10  ;  Training Loss: 0.002802 ; Testing Acc: 71.320 ; Time consu
med: 1m 52s
Iteration: 6 /10  ;  Training Loss: 0.001655 ; Testing Acc: 67.600 ; Time consu
med: 1m 52s
Iteration: 7 /10  ;  Training Loss: 0.000921 ; Testing Acc: 75.050 ; Time consu
med: 1m 52s
Iteration: 8 /10  ;  Training Loss: 0.000547 ; Testing Acc: 72.210 ; Time consu
med: 1m 52s
```

Now, if you look down at how the model was working out. So, we trained it over at any epochs it takes about 1 minute 51 seconds. So, that is about 8 to 9 seconds short of 2 minutes is what it takes down to train it down.

Now, it starts with an average accuracy during training for testing somewhere at 50 percent and then my test accuracy goes up to something around 70 percent. Now in between it was at 75 percent 73 percent as well.

(Refer Slide Time: 14:47)

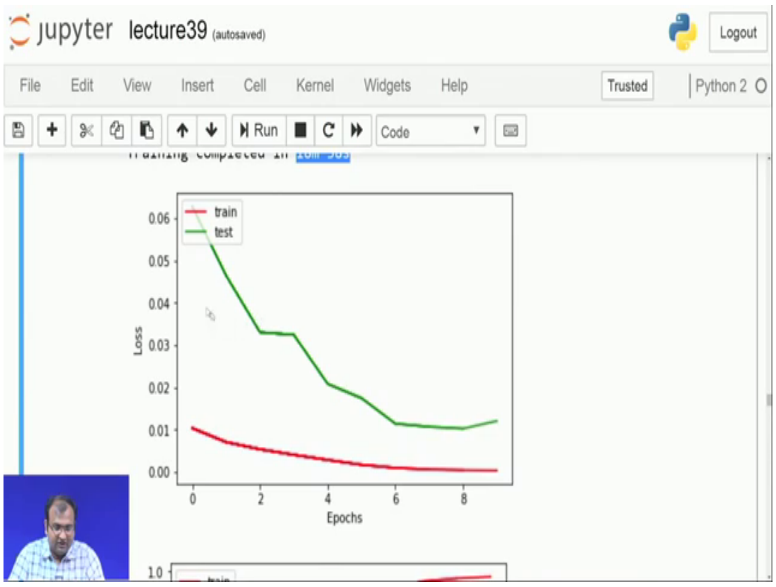

jupyter lecture39 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

```
Iteration: 3 /10  ;  Training Loss: 0.005332 ; Testing Acc: 71.150 ; Time consu
med: 1m 51s
Iteration: 4 /10  ;  Training Loss: 0.004019 ; Testing Acc: 55.720 ; Time consu
med: 1m 52s
Iteration: 5 /10  ;  Training Loss: 0.002802 ; Testing Acc: 71.320 ; Time consu
med: 1m 52s
Iteration: 6 /10  ;  Training Loss: 0.001655 ; Testing Acc: 67.600 ; Time consu
med: 1m 52s
Iteration: 7 /10  ;  Training Loss: 0.000921 ; Testing Acc: 75.050 ; Time consu
med: 1m 52s
Iteration: 8 /10  ;  Training Loss: 0.000547 ; Testing Acc: 72.210 ; Time consu
med: 1m 52s
Iteration: 9 /10  ;  Training Loss: 0.000405 ; Testing Acc: 73.460 ; Time consu
med: 1m 52s
Iteration: 10 /10  ;  Training Loss: 0.000304 ; Testing Acc: 70.310 ; Time cons
umed: 1m 53s
Training completed in 18m 38s
```

Now, you would see that my training loss is what is starting at point 1 and then it keeps on going down and down. Now while my accuracy had grown gone high on my testing side. So, you see that it goes out goes up to 71 percent then falls down to 55 percent, then again goes to 71 percent, then to 67 percent. Then again to 75 percent, but on the other side of it you would see my training loss is constantly decreasing. Now one thing is there are few people who would argue over here that possibly I am over fitting on my training loss and that is my testing one is going down, but nonetheless you need to keep one thing in mind that this kind of a jitter behavior over here is something which is happening, because its possibly hopping from one minimum point to the other minimum point.

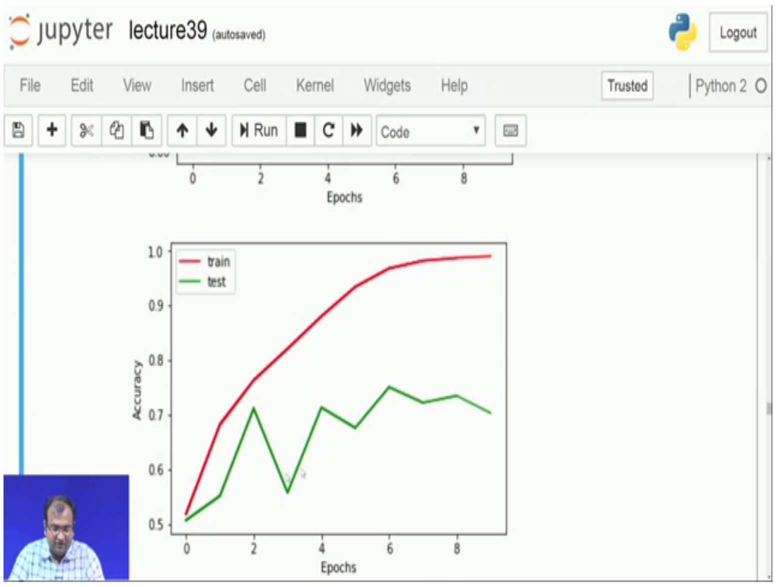
Now, that is a beauty of adam which comes to play over here, that you are not getting locked into local minimum points, but then you can hop and you are going down towards the global minimum and that is that is the dynamic change which you are observing over here. So, 10 epochs get in total of 18 minutes and 38 seconds. So, that is roughly about 1.8 minute poor epoch and that is not so bad actually given the fact that when we were trying to do it with googlenet. It was actually taking me about 8 epochs 8 minutes per epoch and here and something about 2 minutes per epoch. Now I have a model which is roughly one third of the model size, but much deeper and it takes me about one fourth of the time to run it over here, now that is that is a total cumulative thing which comes down. So, if you can really play around with your architectures you do save a lot of your resources over there.

(Refer Slide Time: 16:17)



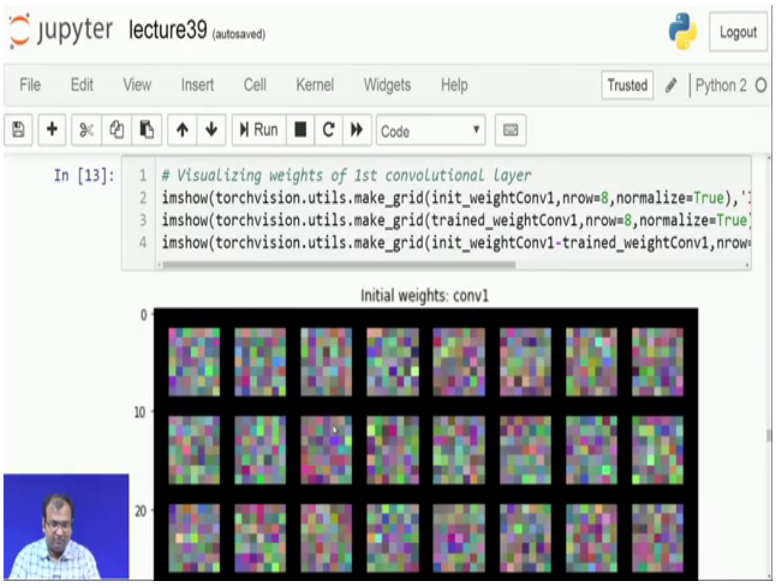
Now, we have our train and test losses over this you would see that your training losses is something which starts lower and keeps on going and your test loss is still going lower, but it does have a jittery behaviour.

(Refer Slide Time: 16:27)



On the accuracy side of it you see this whole jittery behavior between 51 percent and 71 percent and keeps on going. So, somewhere around at the end of tenth epoch is where it sticks down to 70 percent and that is more or less where it will be seeking down because you can see that its almost over there its roughly at 70 percent where it keeps on going. So, you can keep on training over a longer number of iterations, which I would typically suggest you to do and find out where it keeps on saturating. The next part after training is to copy down my weights and then try to look into the visualization of the weights over there that is what we do over here.

(Refer Slide Time: 17:01)

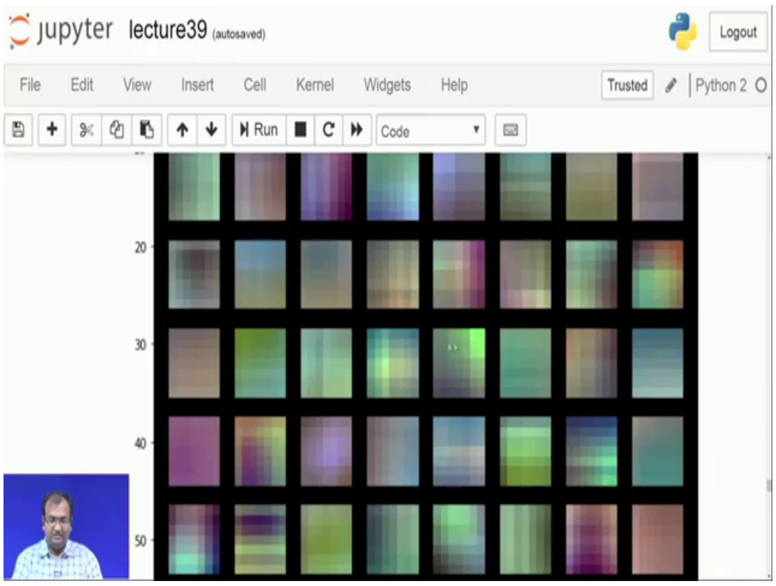

jupyter lecture39 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code
In [13]:
Initial weights: conv1
0
10
20

```
# Visualizing weights of 1st convolutional layer
imshow(torchvision.utils.make_grid(init_weightConv1,nrow=8,normalize=True),'
imshow(torchvision.utils.make_grid(trained_weightConv1,nrow=8,normalize=True
imshow(torchvision.utils.make_grid(init_weightConv1-trained_weightConv1,nrow
```

Now, if you recall from the discussions which we had on the architecture. So, the first layer over there is something which has a special kernels which are off span of 7 cross 7 and that is that is what we plot down over here and there are 1, 2, 3, 4, 5, 6, 7, 8 and 1, 2, 3, 4, 5, 6, 7, 8. So, it is a 64 cross 64 and then if you remember from your earlier lecture you can you can actually flip back onto that earlier one. So, you had this connections from 3 cross 2, 2 4 cross 224 and then you had 7 cross 7 kernels and you had 64 of those. So, each is one of these kernels of 3 cross 7 cross 7.3 is for the input number of channels and then you have 64 such channels which are visualized on a 8 cross 8 grid of these kernels coming down. Now these are the weights at the initial which is at a randomized start over there and then these are the weights which come down after 10 iterations of training.

Now, if you look down at the difference yes there is a significant amount of difference you see and then these are mole of like gradient based changes, which come from although look looking down at the kernel its really hard to understand like what is changing over there because they are they still appear as random pixels of colour.

(Refer Slide Time: 18:14)

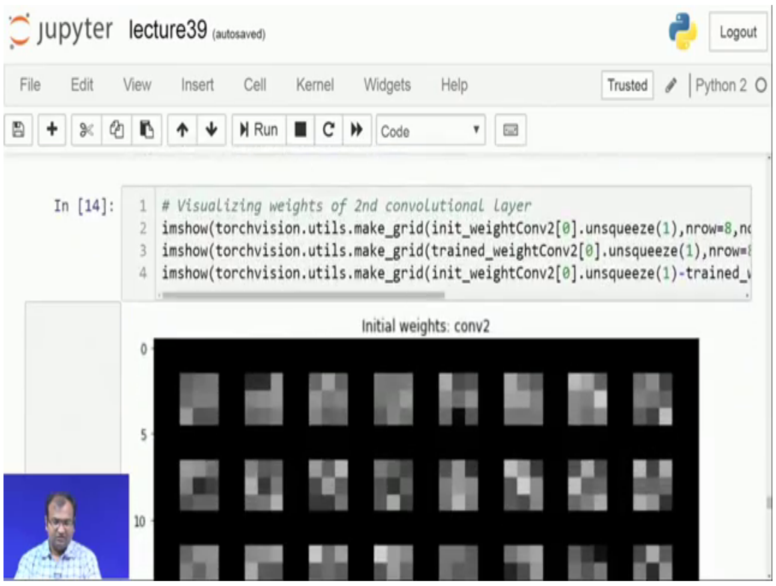

jupyter lecture39 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code
20
30
40
50

Now, here you would be seeing that there are certain kind of directional gradients which start setting in over there, and then these are more of colour based detection of gradients. Now that is an interesting factor which does come into play, but then given the fact that we are not putting down any imposition on what is the nature of change which should take place and how it should be taking place. The only thing which we are doing is back propagating my gradient across the network over there. So, that is what happens and it keeps on changing. Nonetheless to say that since this is a residual network which with a much wider receptive field on the first layer over there.

So, you are able to encapsulate much wider aspects of the images and the objects present in those images and that is what does play a significant role. Now see far which was a much smaller 32 plus 32 or 64 thing which were scaled up to almost 8 times of that to 224 cross 224. Now instead of that if you are taking down very detailed images of 224 cross 224 and on which I have a level on your image straight. So, that is something which we will leave down to all of you guys to do its a large data set you need to need time to download you would also incur more time to actually train down per epoch as well. Now if you are able to download that and do over that, you would see that there would be much finer granules or changes which would come down despite it would also be kneading down more number of epochs. So, that is something which I leave up to you are free to do so.

You can in fact, look into any of our other datasets which we are done with auto encoders and stuff as well, you would be getting on a very different behavior for each of them and that is something interesting to really keep on looking down over there. So, that is what we have for the first convolution kernel over there.

(Refer Slide Time: 19:48)


jupyter lecture39 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code
Initial weights: conv2
0
5
10

In [14]:
```
# Visualizing weights of 2nd convolutional layer
imshow(torchvision.utils.make_grid(init_weightConv2[0].unsqueeze(1),nrow=8,nc
imshow(torchvision.utils.make_grid(trained_weightConv2[0].unsqueeze(1),nrow=
imshow(torchvision.utils.make_grid(init_weightConv2[0].unsqueeze(1)-trained_
```

Next week we look into the next kernel which the next layers are now 3 cross 3 the first layer over there and then this is one of these kernels and now since it connects all the 64 over there. So, you still have one layer which corresponds to one of these convolution. So, for one kernel you have 64 such channels which are over there and each has a spatial span of 3 cross 3 which connects on the previous layer and since the previous layer is giving you an output which has 64 channels.

So, here also you need to have those many 64 channels and that is what you get down. So, these are the initial weights before training these are the ones after 10 epochs of training and this is the kind of changes which happens. And now these changes are really massive these are not zero changes, these are all nonzero changes now this is another thing you need to keep in mind.

Now a very higher difference comes down when trying to compare it with googlenet over there was not much of a change in majority of them and the second layer coming down, but here you do see a change coming down over there now that is that is interesting to explore out as you keep on going down. Now that makes me come to an end for residual networks and then stay tuned for the next lecture, where we would be doing and discussing about what happens with densely connected residual networks or densenets until then.

Thanks.