# Deep Learning for Visual Computing

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

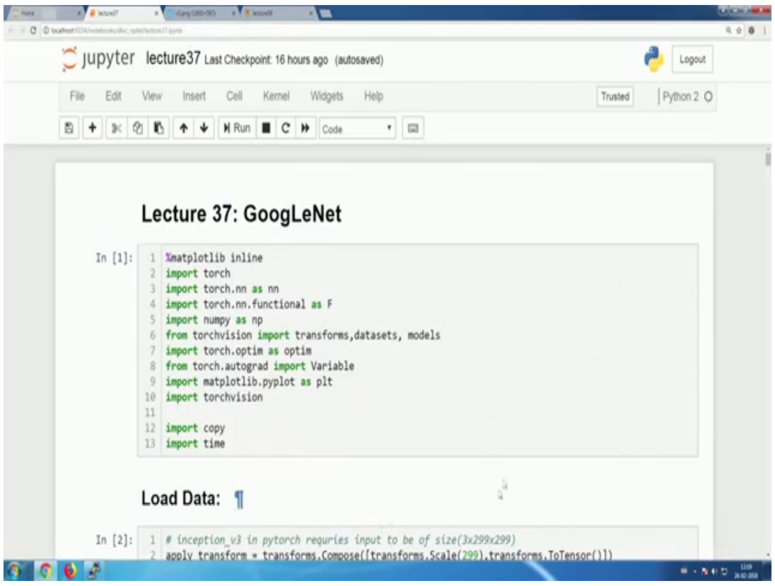
## Lecture - 37
## GoogLeNet

Hello, so in the last lecture we had discussed about one of these very deep neural networks and this was the GoogLeNet or this was the first time that you saw that while counting over v g unit one of the major advantages which we got. So, there were there were two important aspects in GoogLeNet which we had done, one was that this had different kinds of receptor field in terms of your inception models, where you could have 3 cross 3 kernels 5 cross 5 kernels and 7 cross 7 kernels.

So, you were in essence what you are doing is you are going on over larger spatial spans. And also, you had 1 cross 1 kernels which were trying to do some sort of a combination across all the channels of one given pixel over there. Now, what comes out over there was you were able to get down like different receptor fields and then which goes on passing and then eventually, like there was this whole hierarchical combination along the depth over there.

On top of it next major thing was these auxiliary classification arms which allowed you to actually do some sort of a gradient boosting or feedback extra amount of error onto it, whenever there was a chance that the error keeps on going down such that you can train it.

(Refer Slide Time: 01:29)



Lecture 37: GoogLeNet

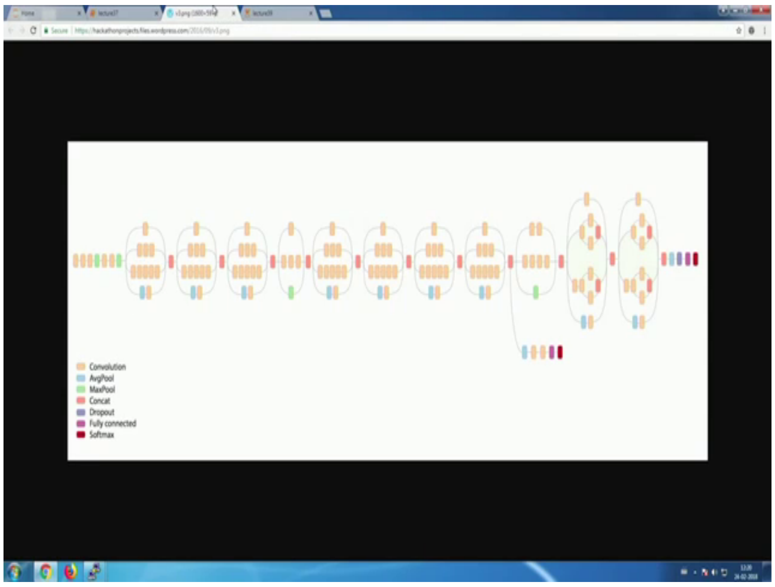
```
In [1]:
%matplotlib inline
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
from torchvision import transforms,datasets, models
import torch.optim as optim
from torch.autograd import Variable
import matplotlib.pyplot as plt
import torchvision

import copy
import time
```

Load Data: ¶

```
In [2]:
# inception_v3 in pytorch requries input to be of size(3x299x299)
apply_transform = transforms.Compose([transforms.Scale(299),transforms.ToTensor()])
```

Now, the model which we had studied was the vanilla model which was directly reported down on the GoogLeNet paper for the first, but the one which we are going to use today is a bit different.

(Refer Slide Time: 01:36)



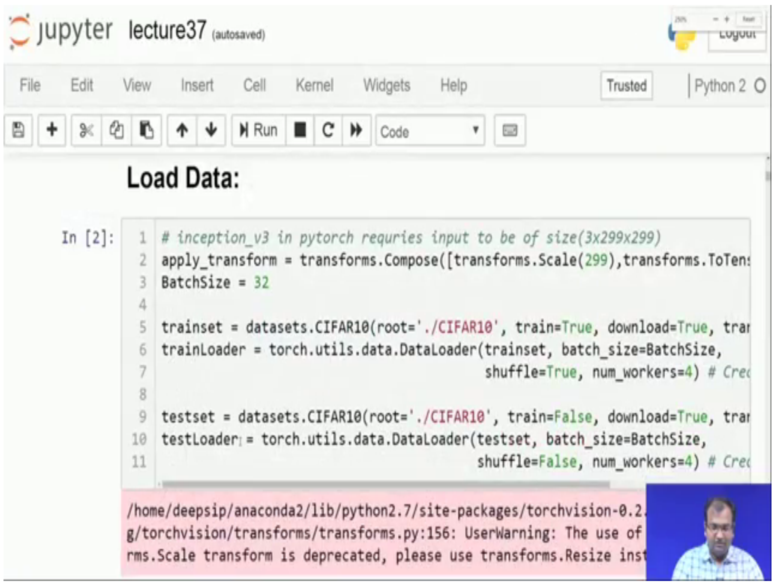
So, this is the model which we are going to use and this is something which is based out of 5 dot. So, you the link for this one is provided directly over there as well and this is from the pytorch documentation. Now, the difference over here is that it does not have two auxiliary arm.

So, there was one auxiliary arm somewhere located over here and there is another auxiliary arm over here. Now, this early auxiliary arm which was their present in the classical model over there is something which is missing in this particular model it just has one auxiliary arm and the main classification arm over there take it down. So, this is the implementation which we are going to make use of today. And this is what is directly available on pie torch for download.

Now, for the beginner part of it is pretty simple, now since it takes a bit longer to run these ones I have actually executed them ahead of time and we are just going to do a plain walkthrough. So, for you guys can actually clear because the one which you get it on your (Refer Time: 02:33) depository that is the one which is pipe cleaned off all the internal variables and anything.

So, anyways you will have to run the whole thing over there. Now, it would take a bit longer the earlier ones which were quite easily running down within seconds and maybe close to a minute this one takes roughly about 8 minutes per epoch 2 get over. Now, that is that is the only difference which comes from. So, what we have is in the initial part is our first part of the header which is just to look into whether everything is present over there or not, now once our header part is over next comes from the data loader over here.

(Refer Slide Time: 03:07)


jupyter lecture37 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

Load Data:

In [2]:

```
# inception_v3 in pytorch requries input to be of size(3x299x299)
apply_transform = transforms.Compose([transforms.Scale(299),transforms.ToTens
BatchSize = 32

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

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

```
/home/deepsip/anaconda2/lib/python2.7/site-packages/torchvision-0.2.
g/torchvision/transforms/transforms.py:156: UserWarning: The use of
rms.Scale transform is deprecated, please use transforms.Resize inst
```

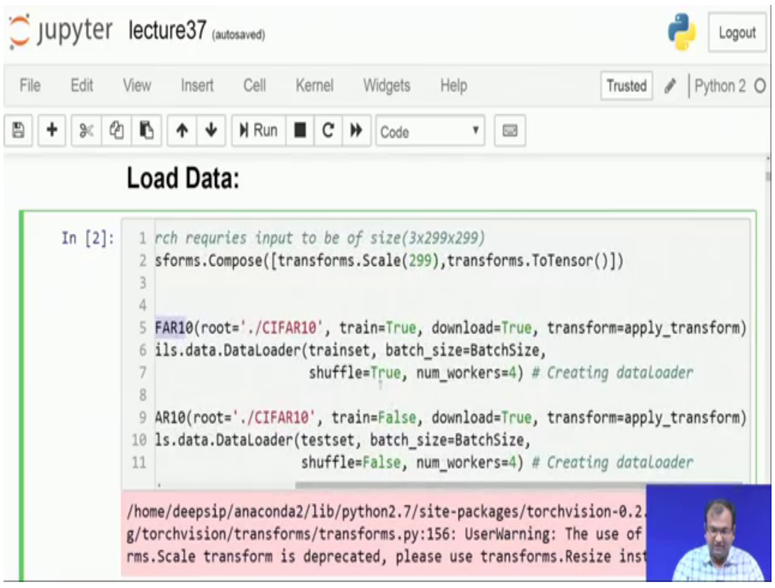
Now, there is something you need to be a bit careful on the data loaded now while we were using 2 to 4 cross 2 to 4 sized images for our VGG net. And that is what the image net itself comes down with, but then for GoogLeNet they had to make a bit of change it was more of with the architecture. And so, that the data does not get into the vanishing point over there.

So, for that the data which this particular model takes down is 229 cross 229 looks a bit weird, but then you have to deal with it is always intrinsic property of the network which you are using. Now, what you need to keep in mind is that, we had earlier as well looked into this data scaling factor and instead of using imaginary data set we are using the CIFAR10 data set for our purpose over here.

Now, what we do is all of these CIFAR10 data sets or something which comes down in smaller sizes of 32 cross 32 sorry 64 cross 64. And then we did it to always enhance it out. So, today what we are doing for this one quite unlike the VGG net value were scaling it up to 224 this is getting scaled up to 299 cross 299.

So, this is the pixel span for each of these images, now this part of the code is pretty simple it just so this is my transform definition over there and then when my train data sets over there what it does is it just looks into whether the data is available otherwise it downloads it out.

(Refer Slide Time: 04:30)


jupyter lecture37 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

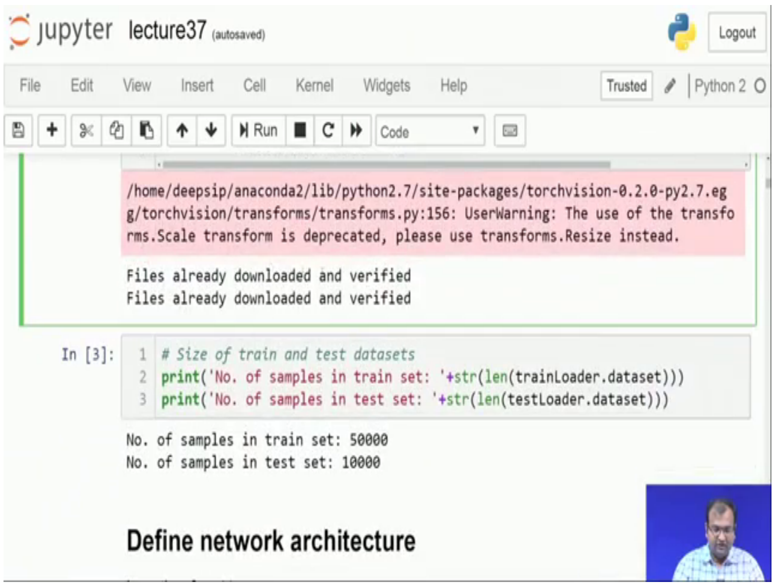
Load Data:

In [2]:
```
rch requries input to be of size(3x299x299)
sforms.Compose([transforms.Scale(299),transforms.ToTensor()])


FAR10(root='./CIFAR10', train=True, download=True, transform=apply_transform)
ils.data.DataLoader(trainset, batch_size=BatchSize,
                        shuffle=True, num_workers=4) # Creating dataloader

AR10(root='./CIFAR10', train=False, download=True, transform=apply_transform)
ls.data.DataLoader(testset, batch_size=BatchSize,
                       shuffle=False, num_workers=4) # Creating dataloader
```

```
/home/deepsip/anaconda2/lib/python2.7/site-packages/torchvision-0.2.
g/torchvision/transforms/transforms.py:156: UserWarning: The use of
rms.Scale transform is deprecated, please use transforms.Resize inst
```

And then you apply this transformation such that; this train set over here is something which is which has images of the size of 3 for the 3 channels cross 2 to 9 cross 2 to 9 over there. Similarly, you do it for the test dataset as well and then your data loader is appropriately defined. So, that you can load it down from your train set or test set whichever you need it.

(Refer Slide Time: 04:56)


jupyter lecture37 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

```
/home/deepsip/anaconda2/lib/python2.7/site-packages/torchvision-0.2.0-py2.7.eg
g/torchvision/transforms/transforms.py:156: UserWarning: The use of the transfo
rms.Scale transform is deprecated, 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.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
```

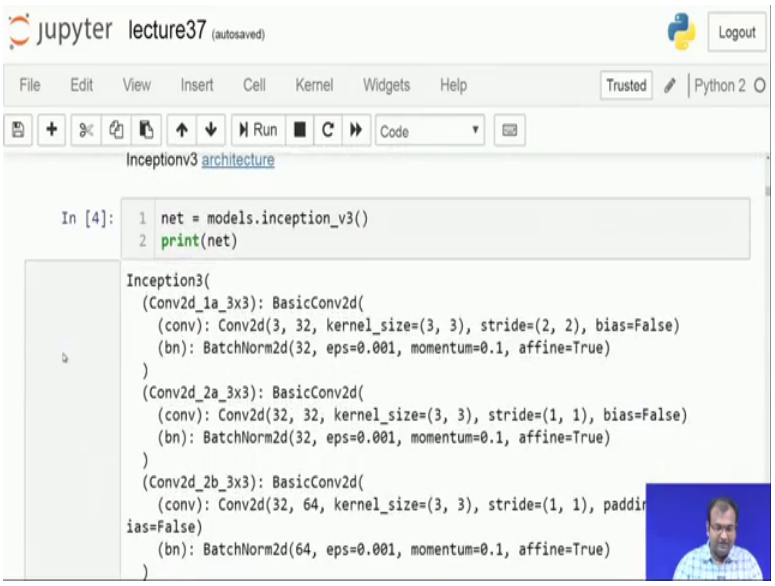
Define network architecture

Now, once you run this one so if your data is not downloaded so apparently, we try to flush it out every time we do a new run just to see one facility.

So, you see this one, but then you might if you are running down on a preloaded dataset which you already have with you might not be getting down these lines over there as well. Now, the next part is to look into the total number of samples so yes CIFAR10 was with 60000 samples and what we had done is we took down the first 50000 samples for training and the 10000 samples for validation.

And then this is quite the way similar as we had done with Aleks net and subsequently with VGG net itself. So, there is nothing which changes over there. Now, for the reference architecture the picture which I had shown you earlier for the model is what is hyperlinked already over here. So, you can as well click and go into that hyperlink and looked into the network in a graphical form. Now, on the other side of it this is an unrolled version of the complete network.

(Refer Slide Time: 05:47)


jupyter lecture37 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

Inceptionv3 architecture

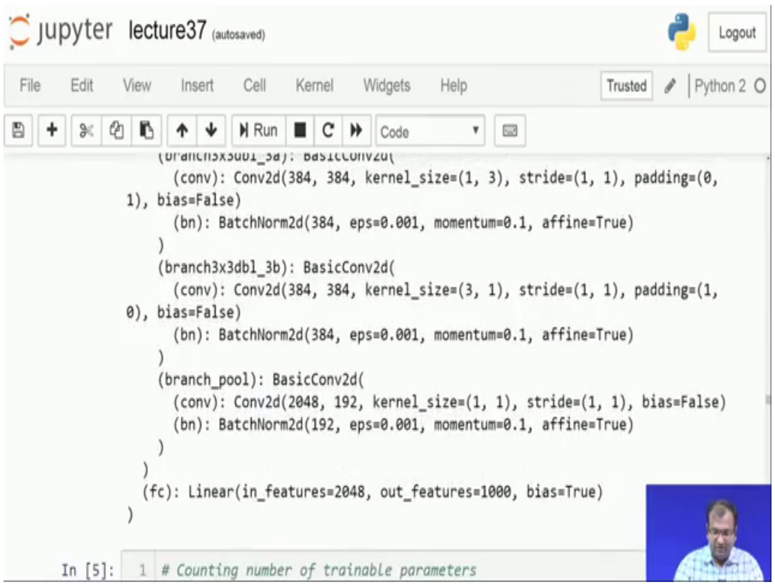
```
In [4]: net = models.inception_v3()
        print(net)
```

```
Inception3(
  (Conv2d_1a_3x3): BasicConv2d(
    (conv): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), bias=False)
    (bn): BatchNorm2d(32, eps=0.001, momentum=0.1, affine=True)
  )
  (Conv2d_2a_3x3): BasicConv2d(
    (conv): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), bias=False)
    (bn): BatchNorm2d(32, eps=0.001, momentum=0.1, affine=True)
  )
  (Conv2d_2b_3x3): BasicConv2d(
    (conv): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), paddi
ias=False)
    (bn): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True)
  )
```

Now, what typically they do is that it tries defining inception modules itself, and then happens up the inception module with the main network and that is that is how you do it. In between you also have these extra auxiliary arms I am just trying to yeah.

So, conception blocks and it keeps on going to each of those a b c for each of these layers is what you had on the table when which we had done in the earlier class and you find the same thing written down over here, now that is a quite a big network which goes down.

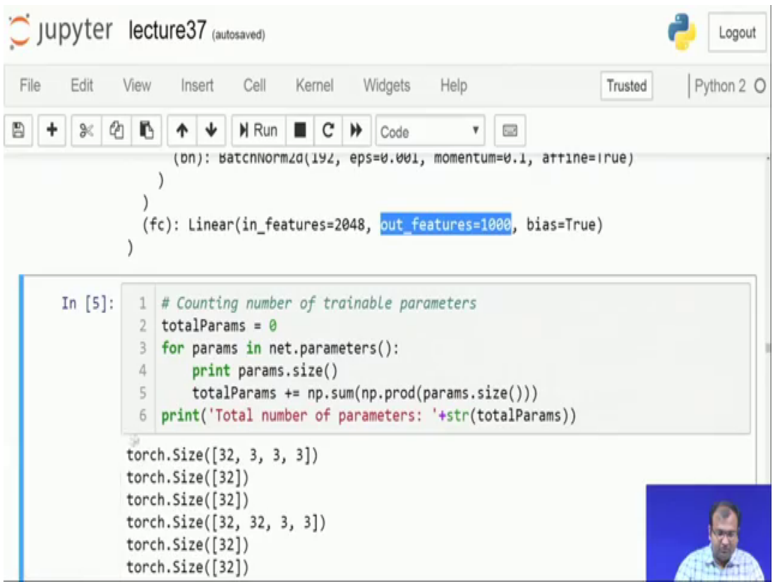
(Refer Slide Time: 06:32)


jupyter lecture37 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

```
      (branch3x3dbl_3a): BasicConv2d(
        (conv): Conv2d(384, 384, kernel_size=(1, 3), stride=(1, 1), padding=(0,
1), bias=False)
        (bn): BatchNorm2d(384, eps=0.001, momentum=0.1, affine=True)
      )
      (branch3x3dbl_3b): BasicConv2d(
        (conv): Conv2d(384, 384, kernel_size=(3, 1), stride=(1, 1), padding=(1,
0), bias=False)
        (bn): BatchNorm2d(384, eps=0.001, momentum=0.1, affine=True)
      )
      (branch_pool): BasicConv2d(
        (conv): Conv2d(2048, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True)
      )
    )
  (fc): Linear(in_features=2048, out_features=1000, bias=True)
)

In [5]: # Counting number of trainable parameters
```

And then this one is the final part of it which is what we needed to look into it ok. So, the final part is where you have 2000 and 48 features, which are linearized and coming out and since it was done originally for the image net challenge. So, you had 1000 classes over there to classify and that is why your output features from this fully connected layer is 1000 and then you had your bias over there.

(Refer Slide Time: 06:56)


jupyter lecture37 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

```
        (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True)
      )
    )
    (fc): Linear(in_features=2048, out_features=1000, bias=True)
  )
```

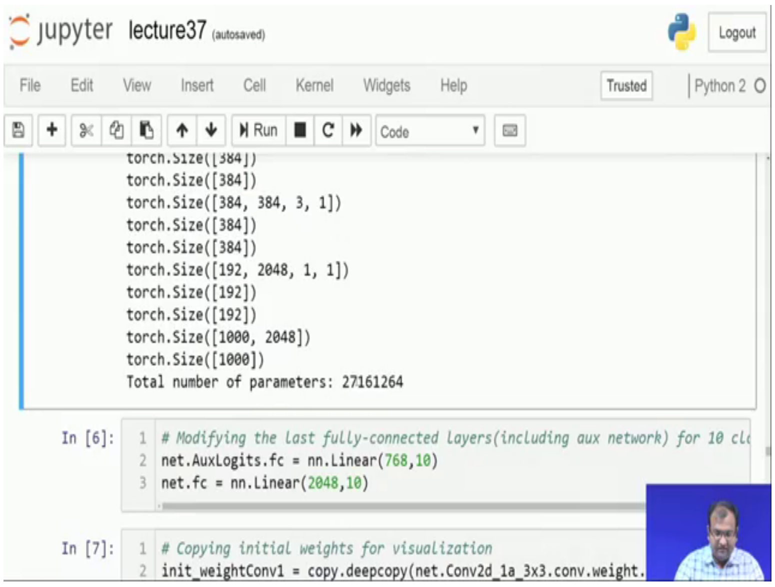
In [5]:

```
# 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([32, 3, 3, 3])
torch.Size([32])
torch.Size([32])
torch.Size([32, 32, 3, 3])
torch.Size([32])
torch.Size([32])
```

Now, the point over here is to look into what is the total number of parameters which you will be learning. So, what we end up doing is for each of these layers we try to put down; for each of the inception blocks and everything coming down what is the total number of parameters and then this is a total detailing for each of them.

(Refer Slide Time: 07:17)


jupyter lecture37 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

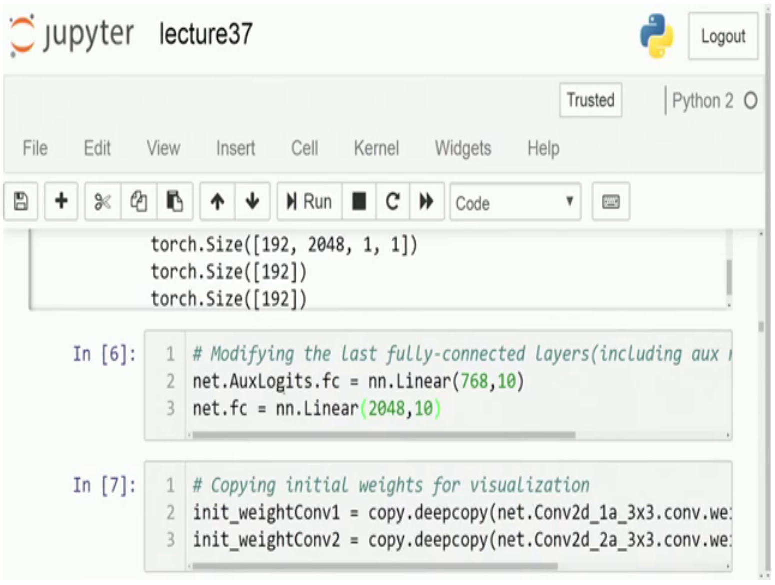
```
torch.Size([384])
torch.Size([384])
torch.Size([384, 384, 3, 1])
torch.Size([384])
torch.Size([384])
torch.Size([192, 2048, 1, 1])
torch.Size([192])
torch.Size([192])
torch.Size([1000, 2048])
torch.Size([1000])
Total number of parameters: 27161264
```

```
In [6]:
# Modifying the last fully-connected layers(including aux network) for 10 cl
net.AuxLogits.fc = nn.Linear(768,10)
net.fc = nn.Linear(2048,10)
```

```
In [7]:
# Copying initial weights for visualization
init_weightConv1 = copy.deepcopy(net.Conv2d_1a_3x3.conv.weight.
```

And once you finish it off you see that you are somewhere over here which is with 27 million parameters. So, it is 27 million 161264 parameters which are there which you need to learn over here.

(Refer Slide Time: 07:32)



```
torch.Size([192, 2048, 1, 1])
torch.Size([192])
torch.Size([192])
```

```
In [6]:
# Modifying the last fully-connected layers(including aux
net.AuxLogits.fc = nn.Linear(768,10)
net.fc = nn.Linear(2048,10)
```

```
In [7]:
# Copying initial weights for visualization
init_weightConv1 = copy.deepcopy(net.Conv2d_1a_3x3.conv.we
init_weightConv2 = copy.deepcopy(net.Conv2d_2a_3x3.conv.we
```

So now what we need to understand is that; since this network was originally defined for 1000 class classification, but then we are changing it down to just a 10-class classification so there has to be a modification. Now, one of the modification is what you would do at the end of the fully connected layer over there, so at the end when you had just 1000 classes to classify over there.

So, instead of thousand classes you are now left with just 10 classes. What you need to also keep in mind is that somewhere in between you also had this auxiliary arm, now this auxiliary arm is also something which will have to modify because you cannot have 1000 classes coming out of the auxiliary am because it still gets down only for 10 classes.

Now, that is where you need to modify the whole code for two different parts. So, one is that this part which is my auxiliary arms classification output now, let us get back into my actual network over here. So, on my network so one is this end part over there; So, if I draw it somewhere close, yeah one is this last part which is my linear now this in this linear one is where I will be making this 1000 to 10.

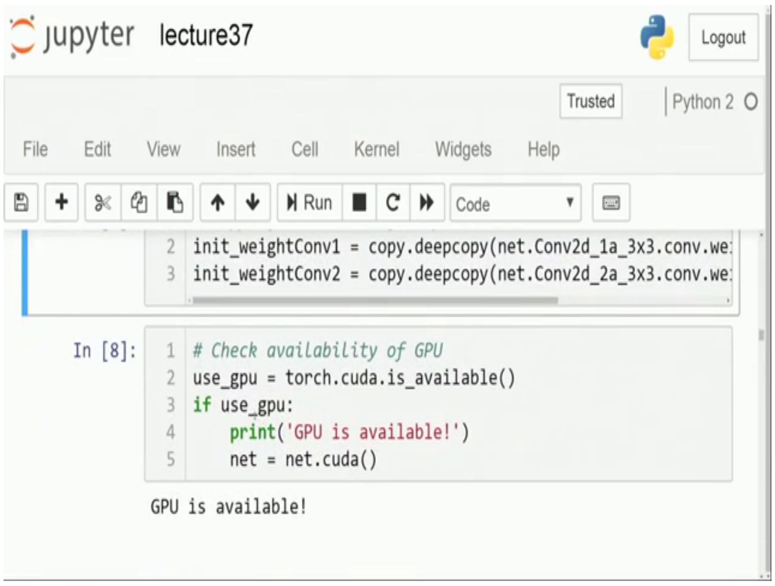
So, that is the second part of my change which I am making. So, I just make this connected from 1024 on to 10. And then the other part which I need to change is somewhere up where I have my auxiliary classifications. So, let us keep on going and I have my auxiliary inception auxiliary over here my, my auxiliary classification that is been my ox logits.

Now, over there I need to change this linear which takes in inputs of 768 features and connects it to just 1000 layers for classification. So, I need to convert this one from 786 which maps down to 1000 to 786 which maps it down to 10 and that is the change which comes over here. Now, what we are essentially doing is we are just replacing out the structures, over there essentially with these structures.

So, it is a pointer remapping which goes on and the rest of the transfer function keeps on remaining the same. Now, this is what we have been doing for VGG nets and over there it was just one arm which produces your output that makes it easier, but here then since it is two arms which produce the output.

So, you have to make down two changes over there. Now, after that we go back with our original form and which is just to do copy down my weights and keep them for our later on visualization at later on point of time.

(Refer Slide Time: 10:04)



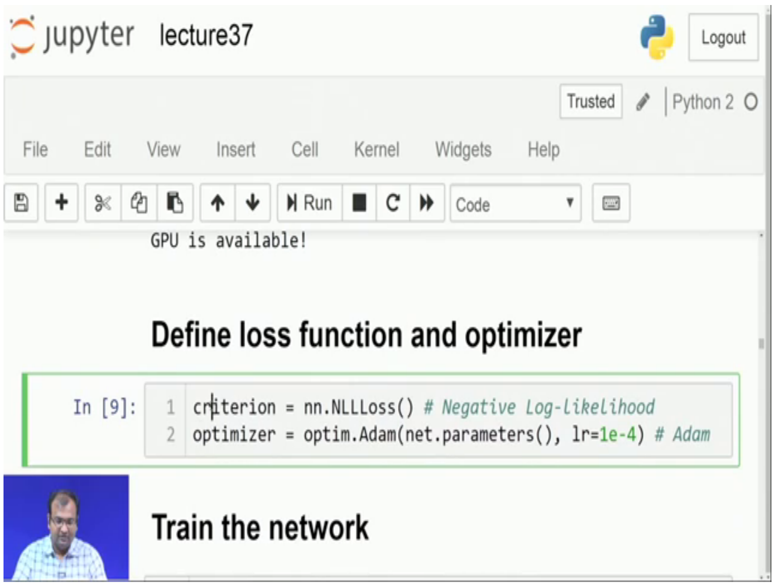
```
init_weightConv1 = copy.deepcopy(net.Conv2d_1a_3x3.conv.we
init_weightConv2 = copy.deepcopy(net.Conv2d_2a_3x3.conv.we
```

```
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!
```

Now, once they are done then I checked on whether my GPU is available and then I define my criterion.

(Refer Slide Time: 10:07)



```
GPU is available!
```

Define loss function and optimizer

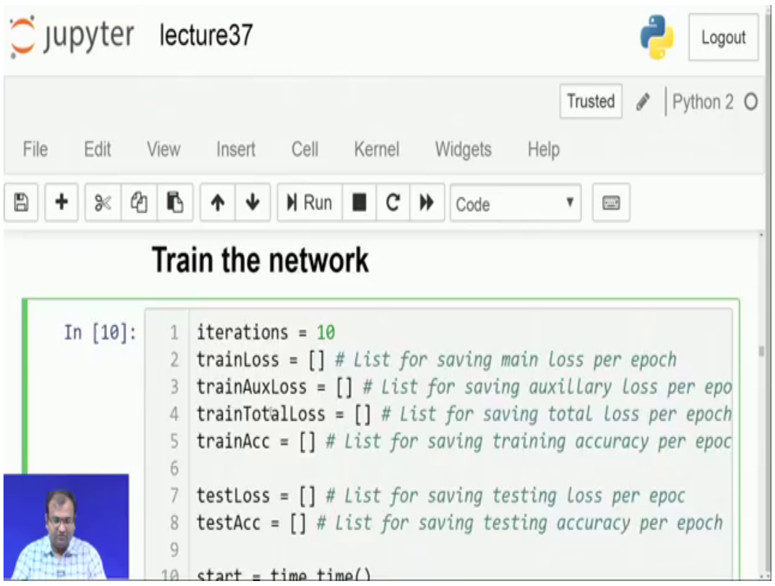
```
In [9]:
criterion = nn.NLLLoss() # Negative Log-likelihood
optimizer = optim.Adam(net.parameters(), lr=1e-4) # Adam
```

Train the network

So now, here we are just going to train it down following the classification pipeline and for that reason we are just using negative log likelihood as my criteria over there. And my optimizer over here is Adam which I am just going to follow them.

So, these are plain vanilla dependent and then taken down in the same way as we had done in the earlier ones.

(Refer Slide Time: 10:34)



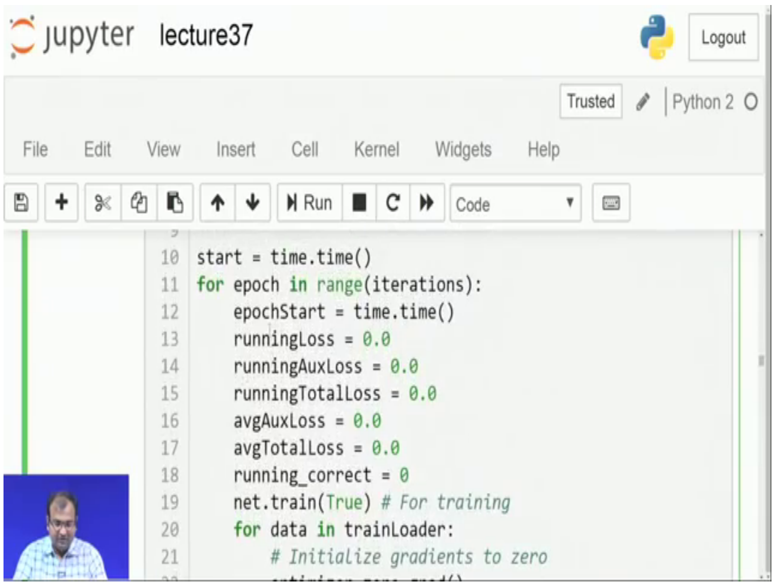
Train the network

In [10]:
```
iterations = 10
trainLoss = [] # List for saving main loss per epoch
trainAuxLoss = [] # List for saving auxillary loss per epo
trainTotalLoss = [] # List for saving total loss per epoch
trainAcc = [] # List for saving training accuracy per epoc

testLoss = [] # List for saving testing loss per epoc
testAcc = [] # List for saving testing accuracy per epoch

start = time.time()
```

Now, from there we need to start training down this network. So, what we do is, you know to keep two things in mind one is this no more has just one loss function, but it has two loss functions which you need to evaluate.

So, whenever you are taking it down you need to have these independently evaluated and independently either you can plot it down for your convenience, but then you need to evaluate each of them because the backward would require each of them to be done independently. Now, from there we start down and this is a training which we do just for 10 iterations 10 epochs over.

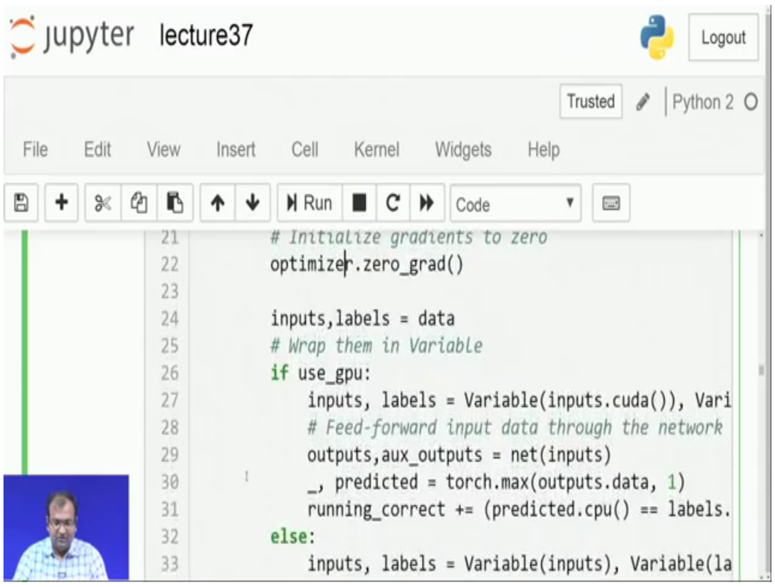
(Refer Slide Time: 11:03)



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

There now the first part is where you 0 down all of these buffers over there and then you start your epoch counter, now within your epoch counter we just do my train data loader.

(Refer Slide Time: 11:12)



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

Now, my data is getting loaded in two batches and each of my batches is of 32. Now, it depends on the kind of a machine on which you are running whether you what you would decide as your optimal batch size, now you can keep batches as small as even 4.

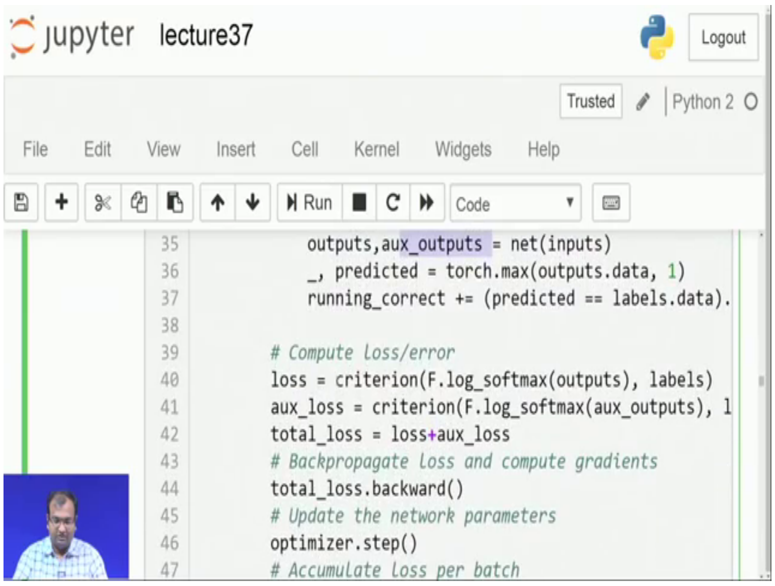
This pretty much you can bring it down to even one based on what we had done in the earlier examples on batch learning versus sequential learning versus whole epoch learning. So, it is just differences which crop up over there and how you are going to make use of that one. Now, other than that it is generally suggested that you keep down your batch in a convenient way which occupies a decent portion of your half level ramp space over there.

So, if your GPU ram on which you are running or maybe your CPU ram on which you are running if you can take down batches of the size of 64 then please put 64. If you are limited by 32 or something then put down smaller numbers like 32 or 16 any of them. So, that is totally up to you we over here have a convenient option of doing it down with the batch size of 32. Now over here what you do is if I have my GPUs available which for me is fortunately there and I just need to typecast all of my variables over here.

So, once the typecasting is done for the variables then I need to typecast my network over there and then once everything is done, now you do a feed forward for the network and then you find out what are your predictions, and then you find out how many of them are correct whether or not ok.

Otherwise your data is not converted any further to cuda it is it is just type casted into a variable and then you have your forward and your prediction and these computes coming out over there.

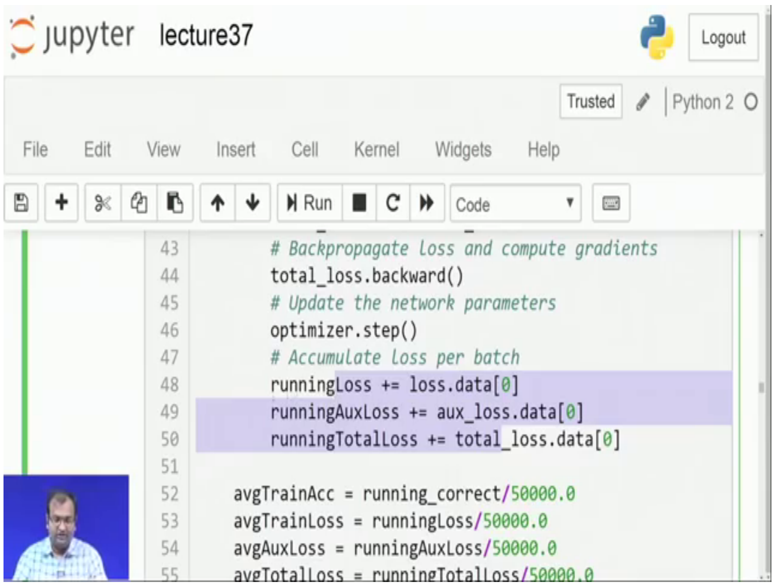
(Refer Slide Time: 12:42)



```
            outputs,aux_outputs = net(inputs)
            _, predicted = torch.max(outputs.data, 1)
            running_correct += (predicted == labels.data).

        # Compute loss/error
        loss = criterion(F.log_softmax(outputs), labels)
        aux_loss = criterion(F.log_softmax(aux_outputs), l
        total_loss = loss+aux_loss
        # Backpropagate loss and compute gradients
        total_loss.backward()
        # Update the network parameters
        optimizer.step()
        # Accumulate loss per batch
```

Now, the next part which comes out is you need to find out what is your loss and now based on that you will be starting your back propagation over the ok. Now, once your back propagation is done we just have our optimizer coming into play and this is Adam for us. So, without much of an issue and then what finally, you have is something like you are just going to take down your losses and then keep on accumulating.

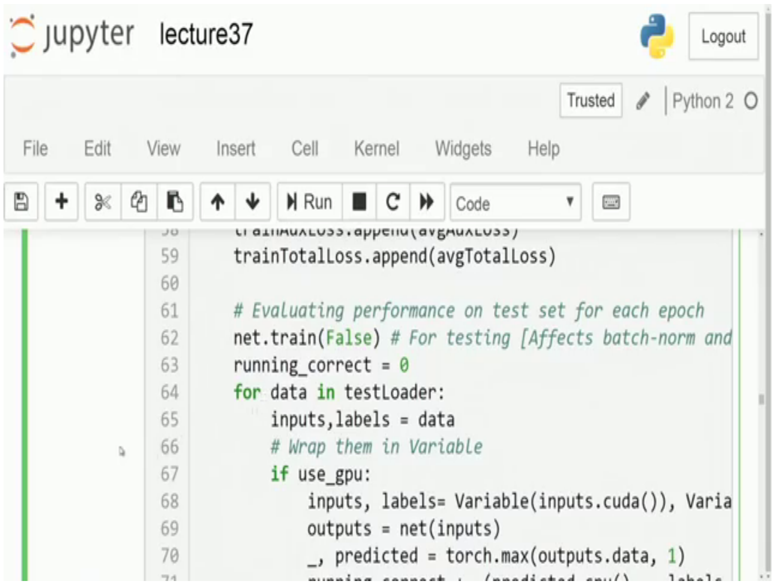
(Refer Slide Time: 13:10)



```
        # Backpropagate loss and compute gradients
        total_loss.backward()
        # Update the network parameters
        optimizer.step()
        # Accumulate loss per batch
        runningLoss += loss.data[0]
        runningAuxLoss += aux_loss.data[0]
        runningTotalLoss += total_loss.data[0]

    avgTrainAcc = running_correct/50000.0
    avgTrainLoss = runningLoss/50000.0
    avgAuxLoss = runningAuxLoss/50000.0
    avgTotalLoss = runningTotalLoss/50000.0
```

And so that you can plot and actually see what is the amount of changes which come down now.

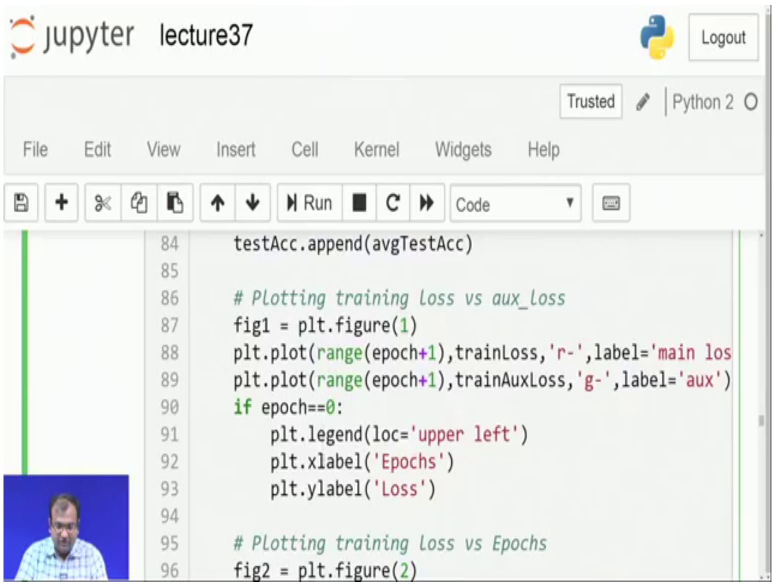
(Refer Slide Time: 13:18)



```
    trainAuxLoss.append(avgAuxLoss)
    trainTotalLoss.append(avgTotalLoss)

    # 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()), Varia
            outputs = net(inputs)
            _, predicted = torch.max(outputs.data, 1)
```

Once that is done for epoch now within one epoch once you finished on all the batches and everything so that is that that part is done; Now, the next part which comes into play is that you are going to actually do a feed forward and then deploy and see down what is the kind of a performance you get down at the end of training it features one epoch and then with the next epoch next go. So, that is a validation part of the code which we are running over here.

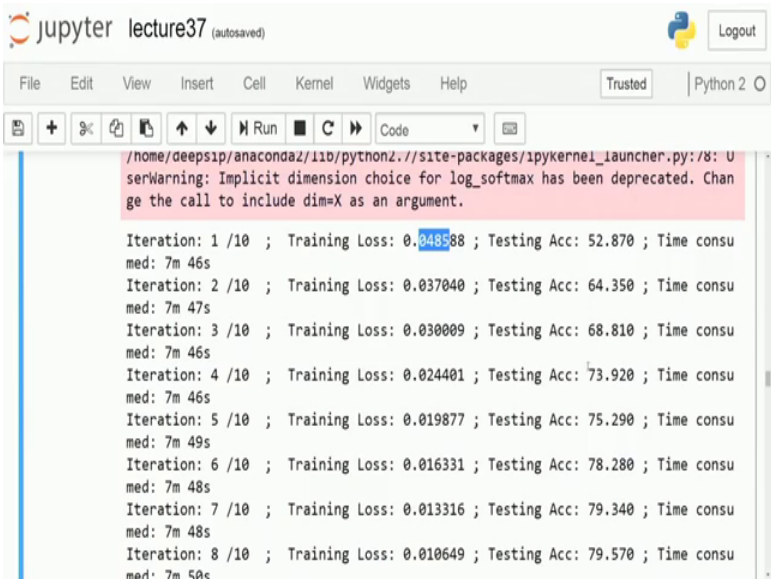
(Refer Slide Time: 13:42)



```
testAcc.append(avgTestAcc)

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

# Plotting training loss vs Epochs
fig2 = plt.figure(2)
```

So, once that is done this is your plot function over there and just to come out over there. So now, since it takes a good amount of time so typically it takes about 8 minutes per yeah so this will yeah.

(Refer Slide Time: 14:00)

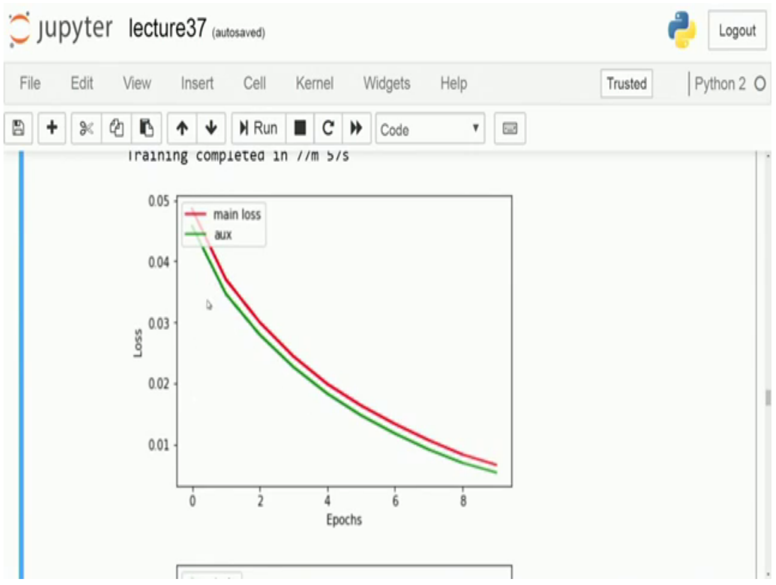

jupyter lecture37 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

```
/home/deepsip/anaconda2/lib/python2.7/site-packages/ipykernel_launcher.py:78: 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: 0.048588 ; Testing Acc: 52.870 ; Time consu
med: 7m 46s
Iteration: 2 /10  ;  Training Loss: 0.037040 ; Testing Acc: 64.350 ; Time consu
med: 7m 47s
Iteration: 3 /10  ;  Training Loss: 0.030009 ; Testing Acc: 68.810 ; Time consu
med: 7m 46s
Iteration: 4 /10  ;  Training Loss: 0.024401 ; Testing Acc: 73.920 ; Time consu
med: 7m 46s
Iteration: 5 /10  ;  Training Loss: 0.019877 ; Testing Acc: 75.290 ; Time consu
med: 7m 49s
Iteration: 6 /10  ;  Training Loss: 0.016331 ; Testing Acc: 78.280 ; Time consu
med: 7m 48s
Iteration: 7 /10  ;  Training Loss: 0.013316 ; Testing Acc: 79.340 ; Time consu
med: 7m 48s
Iteration: 8 /10  ;  Training Loss: 0.010649 ; Testing Acc: 79.570 ; Time consu
med: 7m 50s
```

So, if you look over here it is about 7 minutes 46 seconds 47 seconds. So, that is roughly close to like put 15 seconds short of a minute sort of 8 minute, that that is what it starts down with now initially it starts with the training loss of something about 0.4 then goes down to 0.37 and 0.3 n n you keep this one you see that this one keeps on decreasing, it starts with the accuracy let us do it.

So, it starts with the testing accuracy of 52 then goes to 64 68 73 and it keeps on increasing. And then somewhere around 10 epoch it is already at an accuracy of 80 percent.

(Refer Slide Time: 14:35)


jupyter lecture37 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code
main loss
aux
Loss
Epochs

```
Training completed in 77m 5/s
```

So, if you look down at your main loss and auxiliary loss you would see that they are steadily decreasing over there.

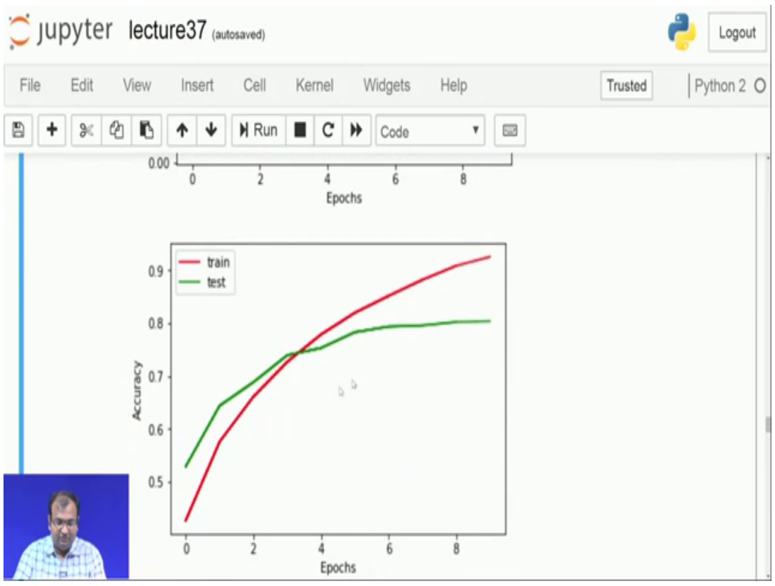
Now, one question does definitely come to your mind that auxiliary arm is something which is not so deep over there it is it is from a shallower region and that is showing a lesser error as compared to your main arm over there. Now, in that will definitely mean that your auxiliary classifier is able to classify better than the main classifier.

Now, you need to keep one thing in mind that this is already a pre-trained model this was not a model which was taken down from scratch this is something which got imported from a model which was trained to solve the image net problem, and they were all modified for your actual natural images and how to understand.

So, the features are possibly something which are not changing, now the only change which was happening within your main arm and auxiliary arm was the new modification of it which came down, although we did not put a hard imposition that we are just going to modify only this layer at a time, but then since everything else is not going to have get down much of an error and that error on the rest of the layers is going to be 0.

So, the only change which will happen is only at the random bits which are there in the auxiliary arm and then newly incorporated main arm for rest of the cases. Now this is the kind of losses which you would be seen down for your train and test and this is the accuracy which would you would be seeing down for your train and test these two.

(Refer Slide Time: 18:48)

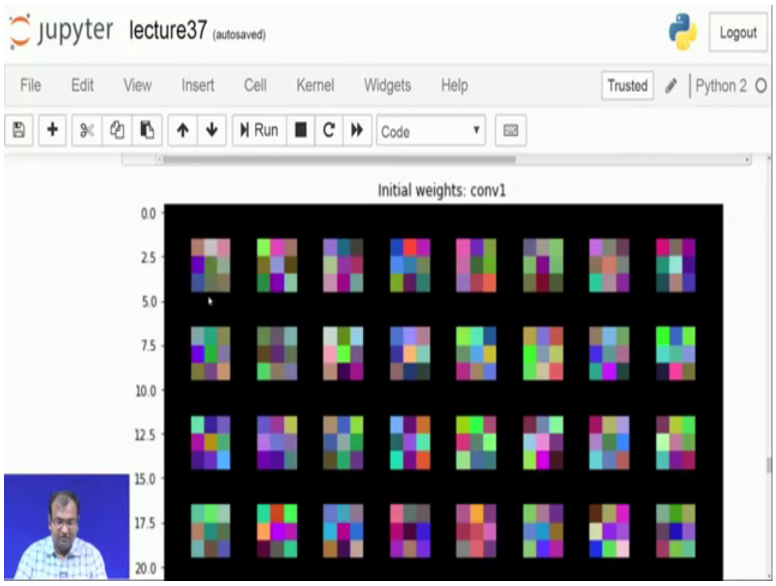


Now obviously, one thing you need to see is that your training accuracy still keeps on increasing while the test accuracy is increasing, but then the rate at which it keeps on increasing there is some sort of an influx and then it keeps on slowing down over there. Now, my testing case accuracy is now already something which is very close to 80 percent, while my training is something which has already overshot 90 percent over there.

So, that is that is just a initial level observation because we have not yet saturated it out. So, this has just been trained for 10 epochs and if you keep on training for longer you would definitely come down to a much more convergent point over there. The next part which we look into is we copy down the weights after training such that we can actually visualize it out.

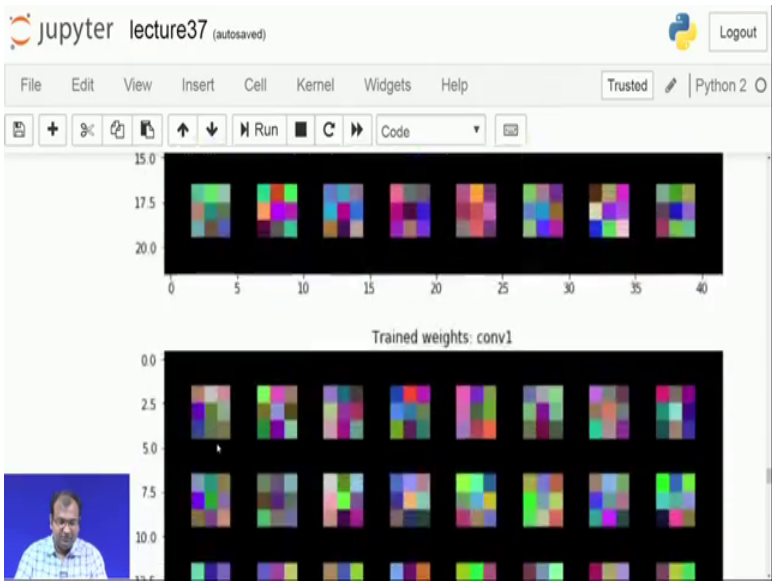
So, on my visualization part over there this is what we look into the first convolution layer over then.

(Refer Slide Time: 16:36)



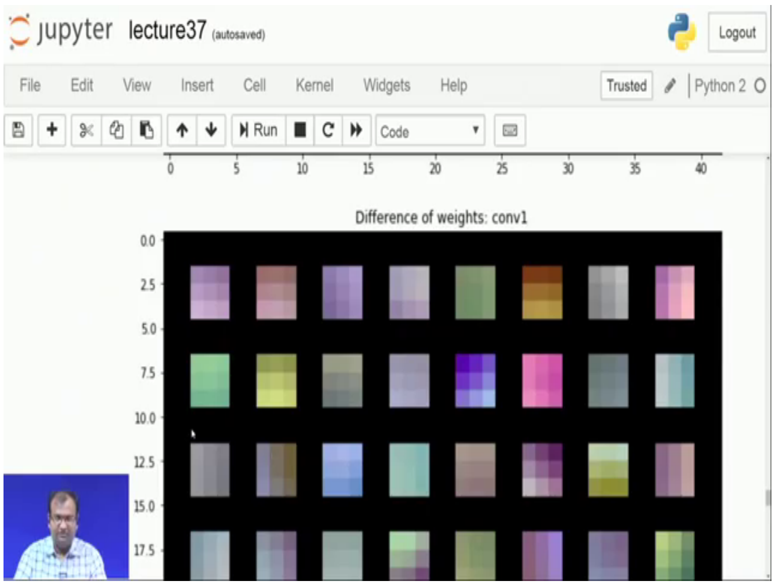
The first convolution layer which is just a 3 cross 3, convolution layers and then you had 1 2 3 4 5 6 7 8 8 and 4 that is 32. So, you had 32 channels coming out from the first convolution layer and this is what they look like. Now these were the initial weights which had been given over there.

(Refer Slide Time: 16:59)



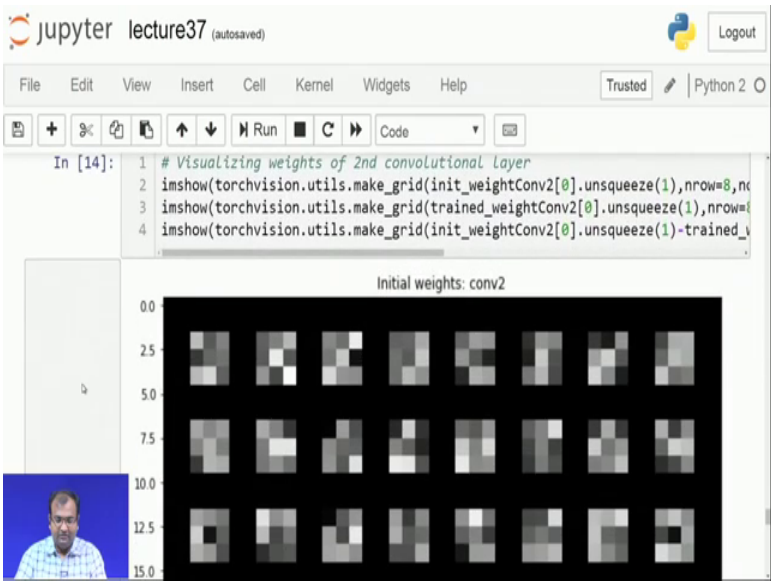
Now, after you had trained it out with this particular data set on C 4 this is the change, but then if you look into the differences which come down they are not much of a differences.

(Refer Slide Time: 17:06)


jupyter lecture37 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code
Difference of weights: conv1

So, there have been a slight changes in the shades or one particular colour channel over there which is just changing, not something which changes majorly. And one of the reasons why the changes are really low in terms of the intensity is because the earlier trade model was already taken down for natural images over there, and you are also dealing with the same kind of a natural image problem over here, now this is to look down into the convolution of one of the kernels for your second layer now.

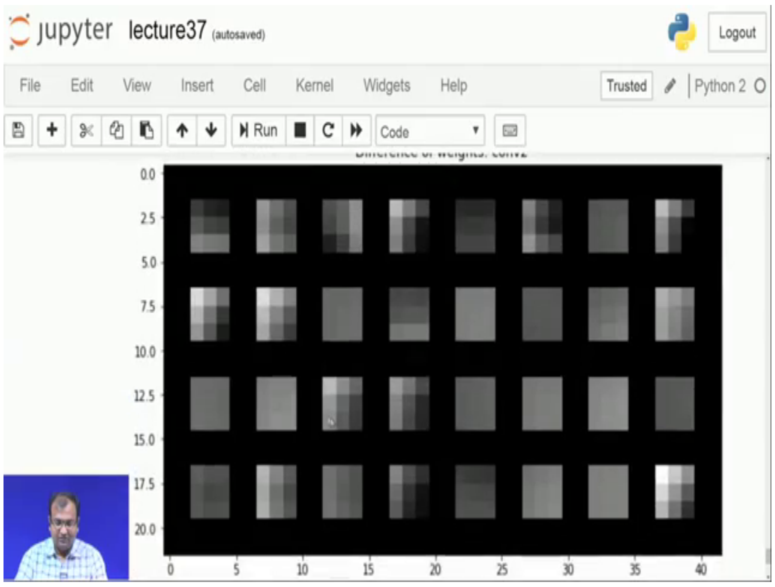
(Refer Slide Time: 17:31)


jupyter lecture37 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code
Initial weights: conv2

```
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_v
```

What do you need to keep in mind is that; since your first layer had 32 channels which are coming out now we just had 32 which is each of these convolution layers, but this is only for one of these kernels. Now, if you look over there for the differences as well.

(Refer Slide Time: 17:48)



You see not much of a change and a majority of them are still flat which is just 0 value changes or near 0 value changes which comes from over there.

So, that is all what we had to do with the GoogLeNet now there is not much of things to say around with it these days, but then the main point which you need to keep in mind is that given that there are two different cost functions which you had to evaluate over here, that was one of the main reasons why you did take a lot more off time.

But then given the other fact also that despite all of this being over there you are able to come down to converges so much faster, because you see that you had started somewhere around roughly a rough guessing estimate of 50 percent and then within 10 epochs you are already at 80 percent.

So, that is a very rapid rise, now what definitely does come to play is that epochs are not always a consideration of it because you also have more amount of time taken per epoch. So, if you take a consideration that the product of time taken per epoch into the total number of epochs then that that comes into an interesting place.

So, somewhere later down when we are going into more of practical buildings we will be seeing down like, what is a more preferable network to take at a given place, given all of these aspects and also the influencing aspect taken on over there.

So, this was about going very deeper with convolutions with GoogLeNet and then in the subsequent lecture, I would be telling you about two more networks which are quite recent from 2016 and 2017 c v p s and they are based on something called as a residual network and residual connections and networks with very dense residual connections and subsequent to that we will be doing their labs as well.

So, with that stay tuned and thanks.