

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

Lecture - 33
AlexNet

Welcome so, today we are going to discuss about one of these are deep convolutional neural networks called as AlexNet which actually brought in the biggest amount of change and brought in deep learning on to the password. So, it is roughly about a decade or close to a decade hold and this network as it goes down. This was one of the first versions of a convolutional neural network which was brought it to solve practical problems and practical image problem.

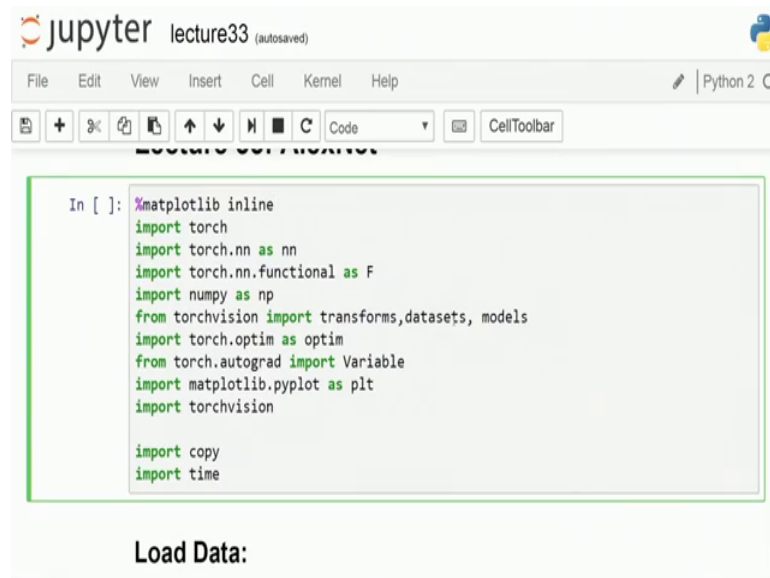
So, one of the challenges which was faced on by the community and which sort of decade in a standardization contest across all of them in computer vision was to do a large scale image object recognition and that is also called as the image net challenge.

So, consists of 1000 object classes and this images which are used on the image net I know more of small pack size of say 32 cross 32. But, these images and now of a larger size and that is 224 hours, 244 and 2 and that is called as images so, there are 3 channels over there. But, anyways if you look down between your standard cifar like images which were 32 cross 32 and then you compare it with 2 these images over here which are 224 cross 224.

So, that is a 8 fold increase across each axis itself and that is would map down to almost like 64 times or 70 times or even more of the area increase over there. So, the amount of information which you have contained in this image is much larger than the amount of information which you had in the images which are solved in the earlier case. And this 224 cross 224 sized images or what have proven down and have taped the way for these deep convolutional neural networks to actually come and be part of the world.

So, we are going to start with that first model and that is called as AlexNet, we have already discussed about it in the lecture theory over there.

(Refer Slide Time: 01:55)



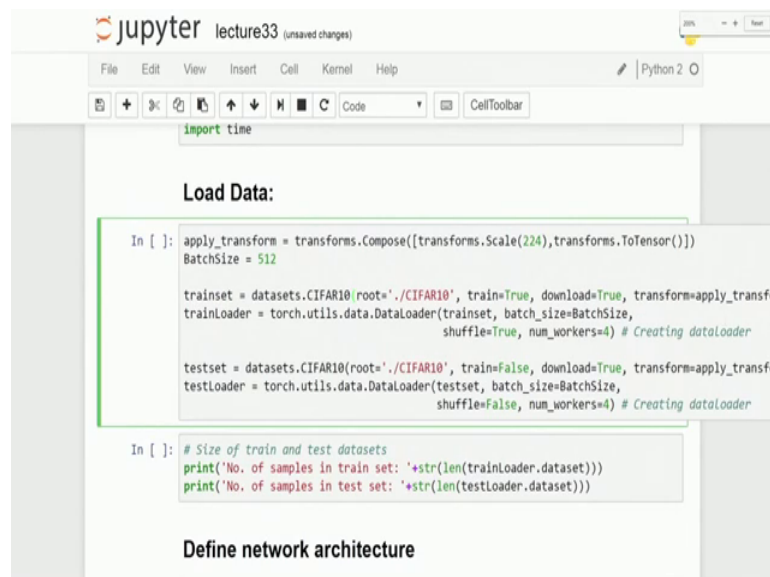
```
In [ ]: %matplotlib inline
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
from torchvision import transforms, datasets, models
import torch.optim as optim
from torch.autograd import Variable
import matplotlib.pyplot as plt
import torchvision

import copy
import time
```

Load Data:

So, today what I am going to do is just walk you through the code and then get it started and running, ok. So, the first part of it is just to load up all the libraries which we need over here.

(Refer Slide Time: 02:08)



```
import time
```

Load Data:

```
In [ ]: apply_transform = transforms.Compose([transforms.Scale(224), transforms.ToTensor()])
BatchSize = 512

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

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

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

Define network architecture

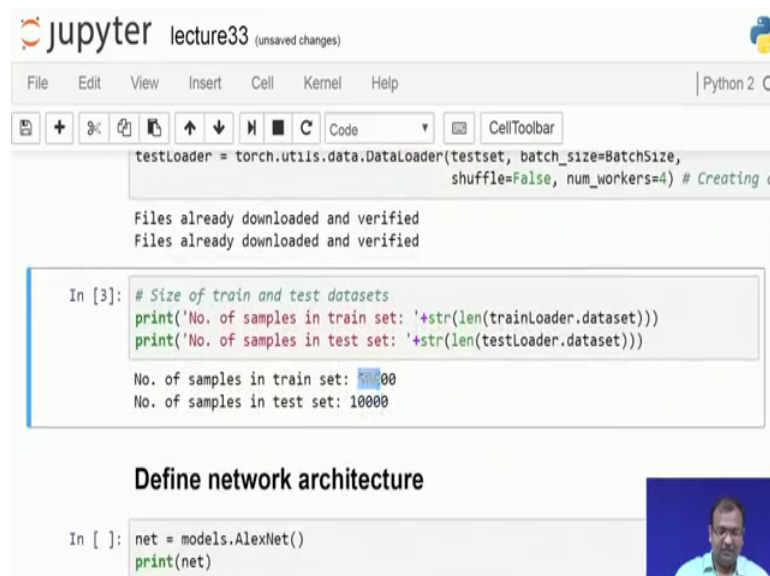
And the next part is to get into my data. Now, for the purpose of just for the sake of similarity and keeping it easier, I am just going to stick down to using my cifar data. However, you need to keep one thing in mind that AlexNet over there is going to take largest size images and not 32 cross 32. So, we need some way of converting all of these

32 cross 32 into larger size images of 224 cross 224 and for that purpose we are just going to use an interpolation later down over here.

Now, what we do for that is we also need to define down whether we are going to put down some sort of a transfer over there. And this transform which we take down within our test loader is going to help me actually get down an image which is offer design size, whatever size I have designed over there. And the size which I designed over the here is to get down a 224 cross 224 sized image coming down over there. Now, I load down batches in back size of 512 and that is for our similarity and inconsistency purposes. So, let us load down all my training and the test data in terms of using just my data loader as I have done.

So, since I have my CIFAR10 data set already downloaded over there so, I can do it. However, there is another interesting point which comes down because you have just 10 classes of objects in CIFAR, where as image that was typically what is defined on for 1000 classes so, but what do we do is quite interesting ok. Next, let us just look into what is the size of these. So, I have 50,000 images on my training set and I have 10,000 images on my testing set from cifar, great ok.

(Refer Slide Time: 03:35)



The screenshot shows a Jupyter Notebook window titled "lecture33 (unsaved changes)". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Help) and a toolbar with icons for file operations and code execution. The code in the notebook is as follows:

```
testLoader = torch.utils.data.DataLoader(testset, batch_size=BatchSize,
                                         shuffle=False, num_workers=4) # Creating a

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)))

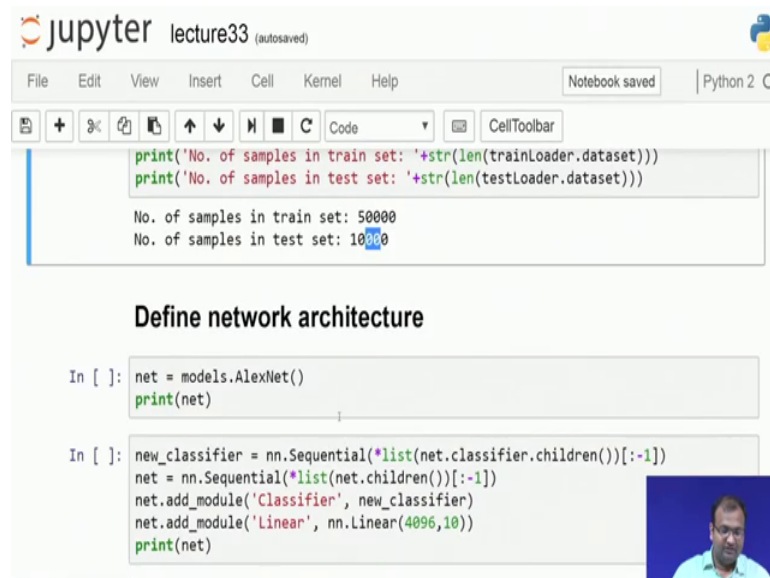
No. of samples in train set: 50000
No. of samples in test set: 10000

Define network architecture

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

A small video thumbnail of a person is visible in the bottom right corner of the notebook interface.

(Refer Slide Time: 03:41)



The screenshot shows a Jupyter Notebook window titled "lecture33 (autosaved)". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Help), a toolbar with icons for file operations and code execution, and a status bar indicating "Notebook saved" and "Python 2".

```
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

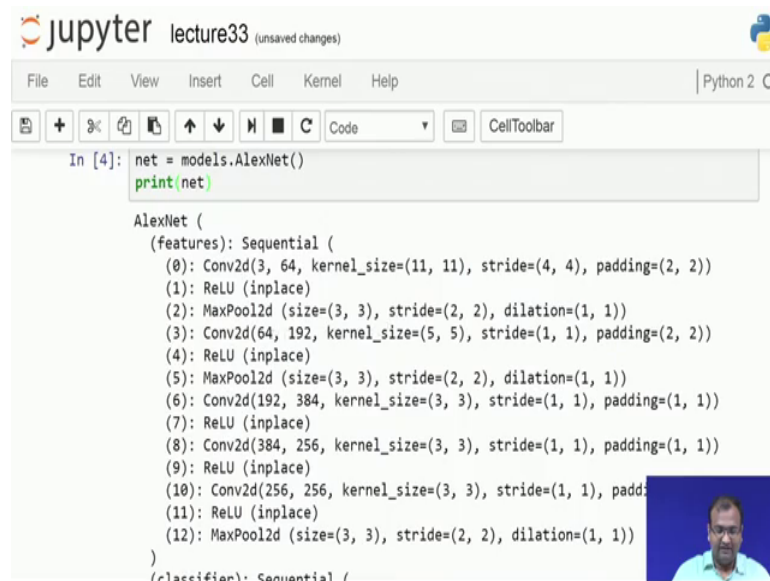
```
In [ ]: net = models.AlexNet()
        print(net)
```

```
In [ ]: new_classifier = nn.Sequential(*list(net.classifier.children())[:-1])
        net = nn.Sequential(*list(net.children())[:-1])
        net.add_module('Classifier', new_classifier)
        net.add_module('Linear', nn.Linear(4096,10))
        print(net)
```

Now, there is an interesting part over there that we might not need to actually defined on the whole network AlexNet, there are from scratch ok. What we can do is we can make use of the predefined models, which are already available within my loa within my (Refer Time: 03:55) libraries over there. So, once I get down over here you would see that I have actually use this particular library from my torch vision. So, I have called on models which I pretend models which are available down and that is within my library called as torch vision and this helps me to call down any architecture which is already defined over the web.

So, what we do is just we say that my network is actually AlexNet which is residing on my models it is there. So, I am just save down from the pain of writing down the architecture and the forward pass function over there. Now, what I do is I just take a print in order to see whether that is what exactly was looking down for.

(Refer Slide Time: 04:35)



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

AlexNet (
  (features): Sequential (
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU (inplace)
    (2): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU (inplace)
    (5): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU (inplace)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU (inplace)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU (inplace)
    (12): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))
  )
  (classifier): Sequential (
```

So, you remember from here AlexNet things that what you had was that the first convolution layer is what takes in a color image over there and has 64 channels to be produced on the output. The kernel size of the first convolution layer is 11 cross 11 you have a side of 4 comma 4. So, it hops by a factor of 4 and you have a padding 2 available over there.

Then the transfer function over there is a ReLU, following that you have a MaxPool operation which has a pooling size of 3 cross 3 in which it is going to pool and pool with the side of 2 comma 2. So, these are very different from any of the earlier networks or even LeNet which we had seen where you were just sliding down on the pooling by the same factor which was the size of the kernel for your max pooling operations.

So, that is quite different from the earlier one. Next, what we have is we connect down 64 channels on my output from the first convolution layer on to 192 channels over there. And then my kernel size changes it comes down to 5 comma 5, 5 cross 5 kernels. I take a lowest stride and actually a fully connected stride over there. So, I do not actually hop down while trying to do this convolution, I am just sticking down to 1 comma 1 stride over there with the padding of 2 comma 2.

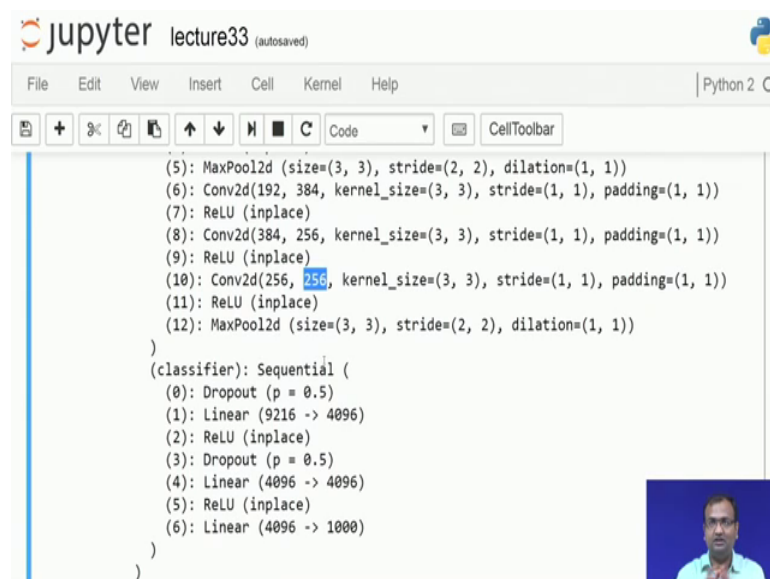
And this is just to ensure that the my boundaries and my sizes are quite consistent on what comes down. So, then I have my ReLU and then following that I have a MaxPool

and this MaxPool also replicates the same behavior of this max pooling which I had in the earlier case.

Now, following that I put on my convolution kernel which converts on 192 channels on to 384 channels and the kernel size is 3 cross 3. So, this reduces down from 5 cross 5 to 3 cross 3. So, you had seen that the first kernels which I had were 11 cross 11, then 5 cross 5 and 3 cross 3 and you can just refer back to the earlier lecture on AlexNet in order to check this consistency with what we had done over there. So, while we had also discussed like how the data would flow, what will be the size of kernels and what will be size of the data following operations. So, that is what is happening over here.

Now, a convolution thing ends over here which is also called as the feature learning other feature extracting convolutional feature extract or part of my network. Following that I just have my classifiers which are more of the sequential connections over there. So, in these sequential connections what I have is initially I have a dropout layer ok.

(Refer Slide Time: 06:42)



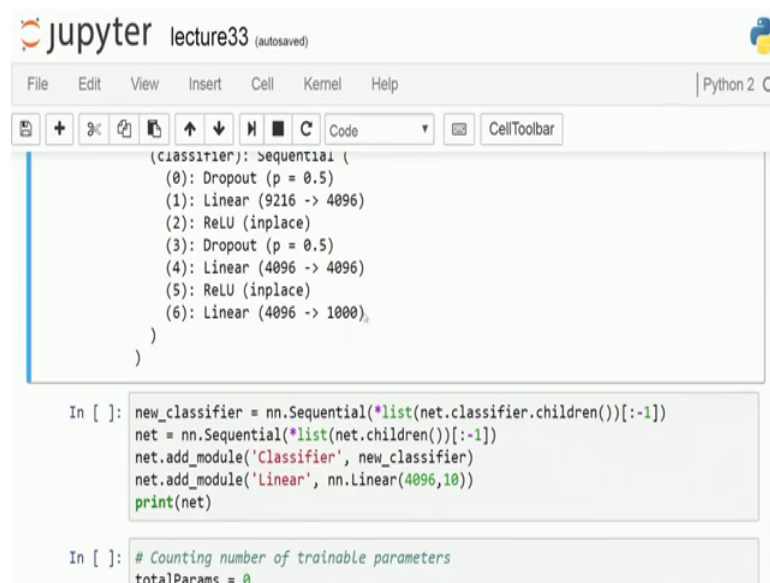
```
Jupyter lecture33 (autosaved) Python 2.0
File Edit View Insert Cell Kernel Help
(5): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))
(6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(7): ReLU (inplace)
(8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(9): ReLU (inplace)
(10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(11): ReLU (inplace)
(12): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))
)
(classifier): Sequential (
(0): Dropout (p = 0.5)
(1): Linear (9216 -> 4096)
(2): ReLU (inplace)
(3): Dropout (p = 0.5)
(4): Linear (4096 -> 4096)
(5): ReLU (inplace)
(6): Linear (4096 -> 1000)
)
)
```

Which has a drop out of 50 percent introduced over there and then that connects down 9216 neurons onto 400 4096 neurons. So, the resultant over here is basically something which is spent over 256 channels and in total if you look into your x, y spans over there and just some of the total number of neurons which will be uniquely available that some comes down to a number of 9216.

Following that I again have my ReLU as my transfer function and then I dropout of 50 percent following that I connect down 4096 neurons onto 4096 neurons and a ReLU, but no more dropout and it is just a connection from 4096 to 1000 ok. It is great looks quite nice and interesting because this is the standard AlexNet architecture which we had.

However, keep wanting in mind that we have cifar and not image net data available to us. So, the labels over that the categories, which we have with us is just 10, it is not 1000 categorical label. So, if we are trying to train a network it will never workout, your cross function will just blot out basically that is a problem. You know you do not have 1000; you have just 10 over there. So, 990 such neurons are just going to be erratically firing up and you do not have a control over those 990 neurons for which you do not have any labels available. So, you need to make a change over the network over here and I mean that that is what we do in the next part over there.

(Refer Slide Time: 08:17)



```
lecture33 (autosaved) Python 2.0
File Edit View Insert Cell Kernel Help
+ -> <-> ↺ ↻ ⌂ Code CellToolbar
(classifier): Sequential (
  (0): Dropout (p = 0.5)
  (1): Linear (9216 -> 4096)
  (2): ReLU (inplace)
  (3): Dropout (p = 0.5)
  (4): Linear (4096 -> 4096)
  (5): ReLU (inplace)
  (6): Linear (4096 -> 1000)
)

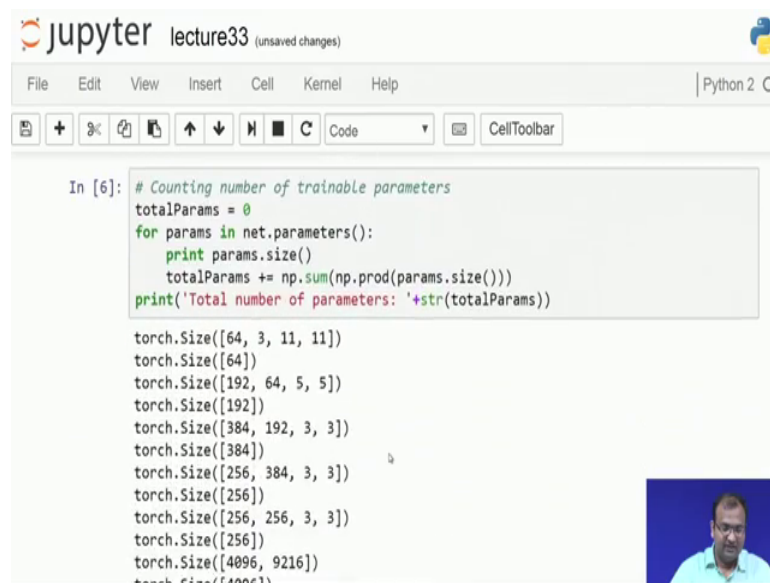
In [ ]: new_classifier = nn.Sequential(*list(net.classifier.children())[:-1])
net = nn.Sequential(*list(net.children())[:-1])
net.add_module('Classifier', new_classifier)
net.add_module('Linear', nn.Linear(4096,10))
print(net)

In [ ]: # Counting number of trainable parameters
totalParams = 0
```

So, what I try to do is basically I remove this last layer which connection 4096 neurons onto 1000 neurons. So, that is that is what has to be removed out and from there I just need to add down another 4 new connection of 4096 to 10 neurons over there. Now, since the network is not trained or anything, it is just randomly initialize network which is available to me. So, this change also does not bring any change on to the characteristics of the network in anyway.

So, let us just make this change and just looked through this one. So, you see that this convolutional part over there for feature extraction remains the same. You have your connections coming down over here and then I have removed out this 4096 to 1000 which was present over here. And then I have just introduced the connection of 4096 neurons onto 10 neurons and that pretty much closes my loop over here. I have my classified part defined and then my network is now consistent to operate down on my cifar 10 data sets.

(Refer Slide Time: 09:14)



The screenshot shows a Jupyter Notebook window titled "lecture33 (unsaved changes)". The code cell contains the following Python code:

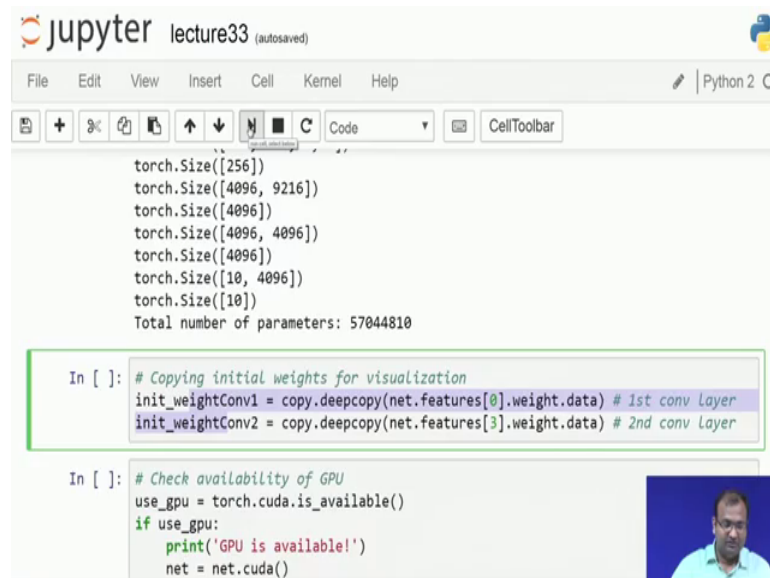
```
In [6]: # Counting number of trainable parameters
totalParams = 0
for params in net.parameters():
    print params.size()
    totalParams += np.sum(np.prod(params.size()))
print('Total number of parameters: '+str(totalParams))
```

The output of the code cell is:

```
torch.Size([64, 3, 11, 11])
torch.Size([64])
torch.Size([192, 64, 5, 5])
torch.Size([192])
torch.Size([384, 192, 3, 3])
torch.Size([384])
torch.Size([256, 384, 3, 3])
torch.Size([256])
torch.Size([256, 256, 3, 3])
torch.Size([256])
torch.Size([4096, 9216])
torch.Size([4096])
```

So, let us look into the total number of parameters which are present over there. So, this is what we had done.

(Refer Slide Time: 09:18)



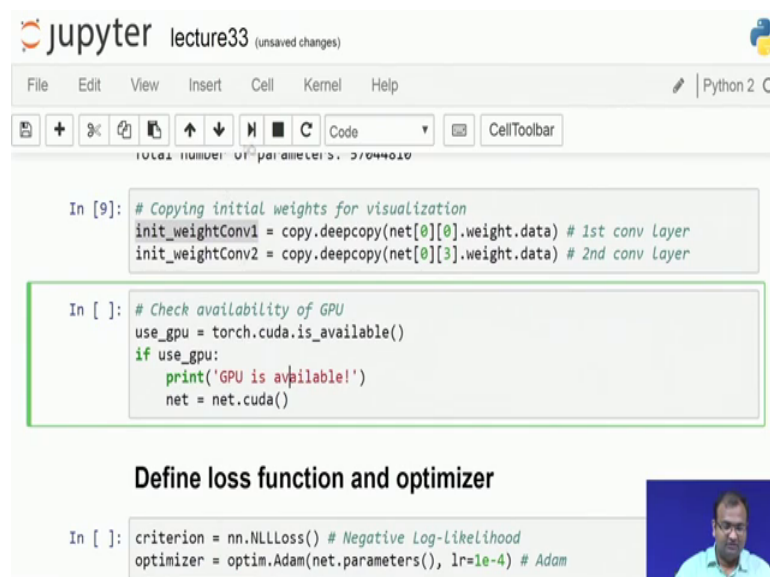
```
torch.Size([256])
torch.Size([4096, 9216])
torch.Size([4096])
torch.Size([4096, 4096])
torch.Size([4096])
torch.Size([10, 4096])
torch.Size([10])
Total number of parameters: 57044810

In [ ]: # Copying initial weights for visualization
init_weightConv1 = copy.deepcopy(net.features[0].weight.data) # 1st conv Layer
init_weightConv2 = copy.deepcopy(net.features[3].weight.data) # 2nd conv Layer

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

In the earlier case when looking at LeNet and trying to make a whole distinction of the weight space over there now, if you look into this kind of AlexNet over there you see that we have something like 5,70,44,810 or if you just go down by the metric system then that is about 57 million parameters. So, this is where your 1000 score down, this is where your million comes into play and then you have 57. So, it is 57 into 10 power of 6 so, 57 million parameters which you have to train over there. Now, the network does not look complicated to most of you; however, you see that the number of parameter is really large and then that is what happens.

(Refer Slide Time: 10:55)



```
Total number of parameters: 57044810

In [9]: # Copying initial weights for visualization
init_weightConv1 = copy.deepcopy(net[0][0].weight.data) # 1st conv Layer
init_weightConv2 = copy.deepcopy(net[0][3].weight.data) # 2nd conv Layer

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

Define loss function and optimizer

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

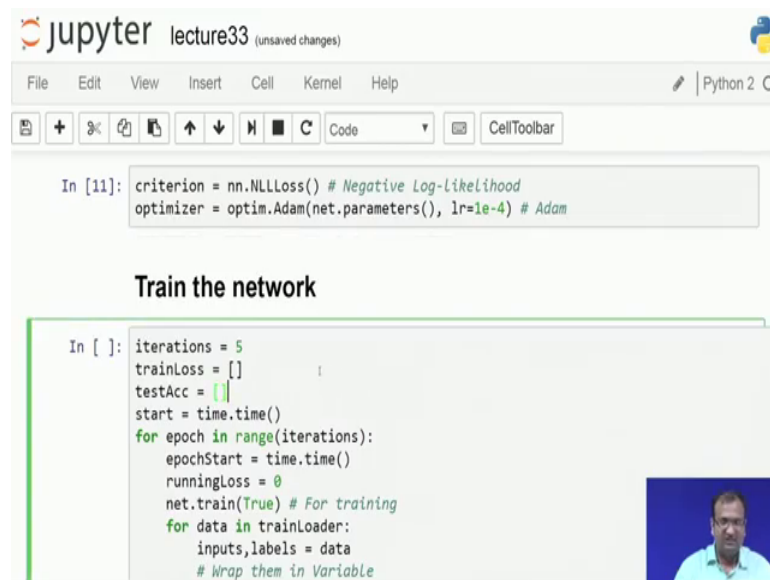
So, in the next subsequent lectures you will be doing even deeper networks which are called as a standard with VGGNet as an example over there, which place you will see even even more number of parameters coming down over there. So, the number of parameters is really high and thanks to one of the points that the number of convolution kernels you are taking down at every layer that being really large to the number of parameters have to be high. There is no other way of going around with that ok.

So, that goes around it and then what we do is we just end up copying down my weights which are the initial random initialization weights which are present over there. So, let us let us do that so, at this part we are just going to copy down all my randomly initialize weights, which I have over here such that I can reuse when I am trying to look down what happens at the end of my training process.

So, let us just run this part and that is where my weights to get copy down and kept down for me and the initial part of it. And then later on after the training process we are going to make use of them for visualization purposes. Now, the next part comes down where I just see if my GPU is available, since if GPU is available then I can just make use of my GPU coming down over there I just fits on for the pooling to happen down if it comes out yes. So, my GPU gets detected and it is available and quite set down to run on this one.

Now, for this to be a classification network what I am going to do is my loss over here is no more emissive or any of them, but it is just classification loss and I am going to make use of negative log likelihood cost function over there. the optimizer which I make use of Adam and that is based on a historical evidence and the earlier lectures, where we had actually pulled across the space of optimizers at to find out which of them is the best possible optimizer to work out. So, we run this part and that is done.

(Refer Slide Time: 11:39)



```
In [11]: criterion = nn.NLLLoss() # Negative Log-Likelihood
optimizer = optim.Adam(net.parameters(), lr=1e-4) # Adam

Train the network

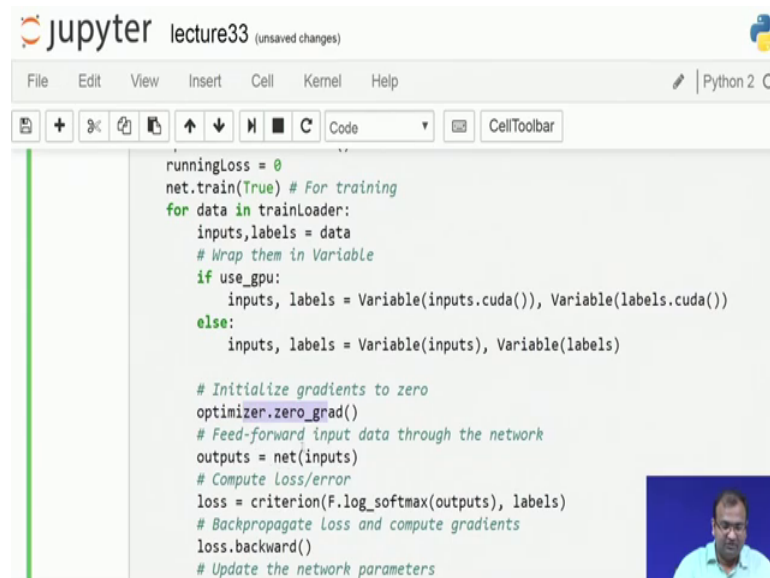
In [ ]: iterations = 5
trainLoss = []
testAcc = []
start = time.time()
for epoch in range(iterations):
    epochStart = time.time()
    runningLoss = 0
    net.train(True) # For training
    for data in trainLoader:
        inputs, labels = data
        # Wrap them in Variable
```

And then I get into my training, now since this network does take a bit of time to train I will just set this one running while we keep on discussing about what goes down inside the network ok.

So, now if you look through this training procedure , I am just using 5 iterations for it and that is just for the sake of sticking down to our time limits over there and not just over shoot it out. Now, if you have an of time and in fact, like when you are doing it on your own site, I would tell you that keep on training it now. And if you want to write really look into what an AlexNet? How long it would take? You are suggested that you actually trended down for more than 10,000 number of epoch and use an actual image net kind of data in order to do it and not just take down for cifar network coming down.

So, within my training system over here I have my epoch which goes down within the epoch, what I do is just find out if my GPU is available, then convert it on to a GPU availability and then have my inputs available over here. Now my input is both the in which comes from as well as the categorical label over therefore, whatever classes getting define and then since I have 10 classes over here. So, it is 10 classes which are present down in my label.

(Refer Slide Time: 12:49)



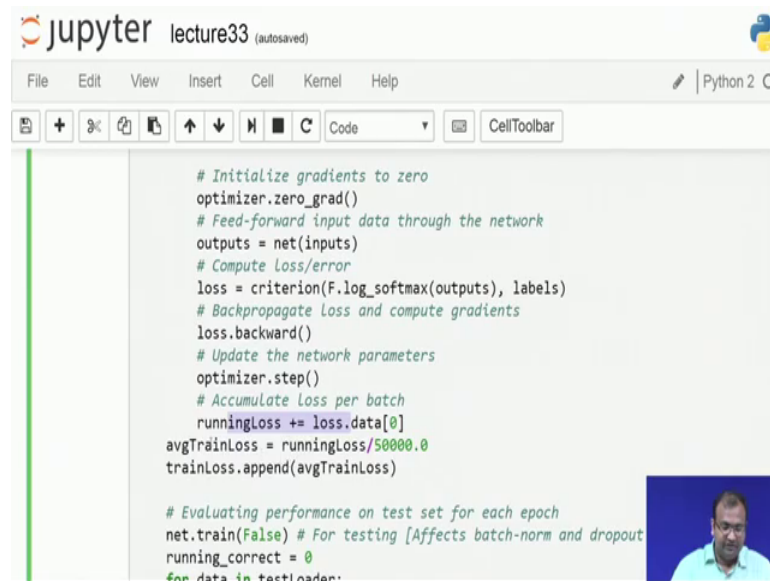
```
runningLoss = 0
net.train(True) # For training
for data in trainLoader:
    inputs, labels = data
    # Wrap them in Variable
    if use_gpu:
        inputs, labels = Variable(inputs.cuda()), Variable(labels.cuda())
    else:
        inputs, labels = Variable(inputs), Variable(labels)

    # Initialize gradients to zero
    optimizer.zero_grad()
    # Feed-forward input data through the network
    outputs = net(inputs)
    # Compute loss/error
    loss = criterion(F.log_softmax(outputs), labels)
    # Backpropagate loss and compute gradients
    loss.backward()
    # Update the network parameters
```

Now, what I do is 0 down my gradients whichever is present inside over there do a feed forward over the network and then find out my loss. And the loss over here is computed after logsoftmax over my whole transfer function and the output and converts it down to my labels which I pres have over here.

Now, once I get down my loss as in the loss of classification at the end of each epoch I am just going to find out what is the derivative of that loss on that side nabla of the loss function ok. Once that is done we run down our step function which is do to do a one step of a back propagation across the whole network over there. So, that is going to propagate across my fully connected and my convolutional connection layer. So, over that completely and then I accumulate of accumulate losses over the whole patch which has been coming down.

(Refer Slide Time: 13:31)

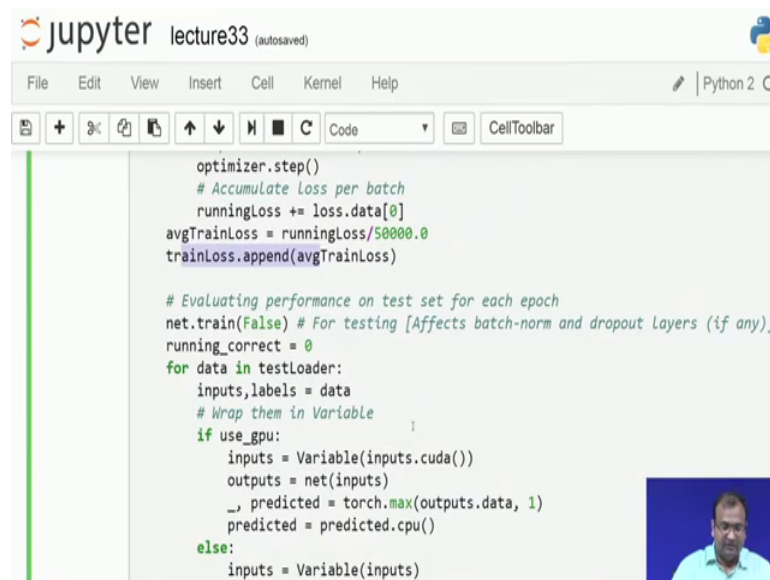


```
optimizer.zero_grad()
# Feed-forward input data through the network
outputs = net(inputs)
# Compute Loss/error
loss = criterion(F.log_softmax(outputs), labels)
# Backpropagate Loss and compute gradients
loss.backward()
# Update the network parameters
optimizer.step()
# Accumulate Loss per batch
runningLoss += loss.data[0]
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
running_correct = 0
for data in testLoader:
```

Finally, coming down to getting an average training loss for each epoch and then I just create an array of this training losses to look down what happens across all the 5 of them.

(Refer Slide Time: 13:44)



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

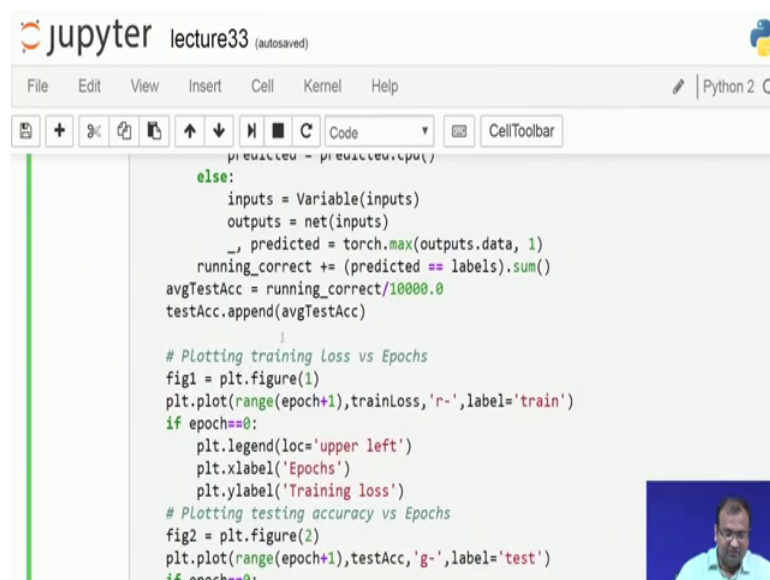
# Evaluating performance on test set for each epoch
net.train(False) # For testing [Affects batch-norm and dropout layers (if any)]
running_correct = 0
for data in testLoader:
    inputs,labels = data
    # Wrap them in Variable
    if use_gpu:
        inputs = Variable(inputs.cuda())
        outputs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

Now, once I have done my training over there using my training data, the next is to actually look down into the performers validation. Now for that I am going to make use of the earlier case which was just 10,000 images from my testing dataset and how good it performs on each of them.

So, what I do is initially low down my inputs over there from my test loader and once I have my inputs available, I do a feed forward pass over my network and then get done by predicted value coming down over here. Now, this prediction over here is not the value, but that is a index label over there of whichever has the maximum value stored over there on the maximum probability.

And then if your value matches down the exactly label over there, then it is correct otherwise it is wrong. And then since you had 10,000 such images or 10,000 samples in your test data sets. So, the average accuracy is the total number of corrects divided by 10,000 and that is gives you the total number of correct measures on the average which has been made down.

(Refer Slide Time: 14:39)



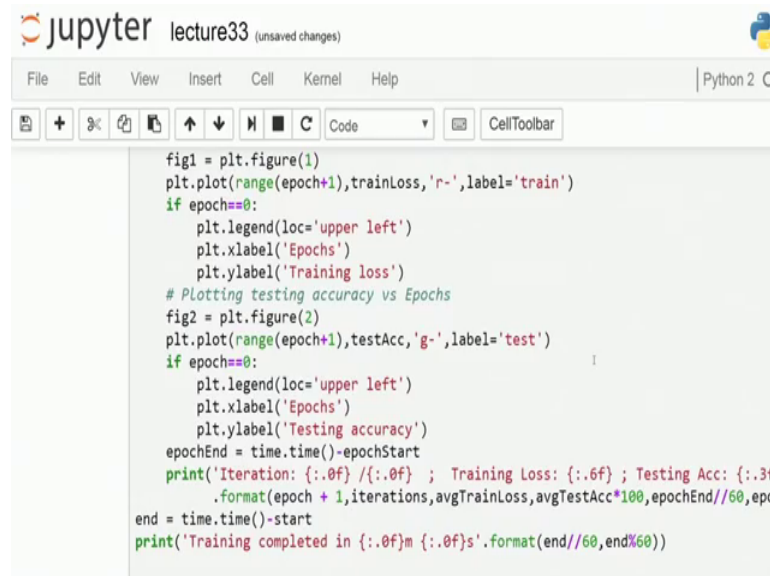
```
predicted = predicted.cpu()
else:
    inputs = Variable(inputs)
    outputs = net(inputs)
    _, predicted = torch.max(outputs.data, 1)
    running_correct += (predicted == labels).sum()
avgTestAcc = running_correct/10000.0
testAcc.append(avgTestAcc)

# Plotting training Loss vs Epochs
fig1 = plt.figure(1)
plt.plot(range(epoch+1),trainLoss,'r-',label='train')
if epoch==0:
    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:
```

So, once that is done I have my average accuracy for testing case taken down. And then over here I have my same old repetitive plotting functions, which can be used for plotting down my cos functions. While I am training on the training data set as well as looking down on my accuracy which keeps on increasing expectedly, which is supposed to increase down as I am training it across the number of epoch over there and finally, to print out what comes over here.

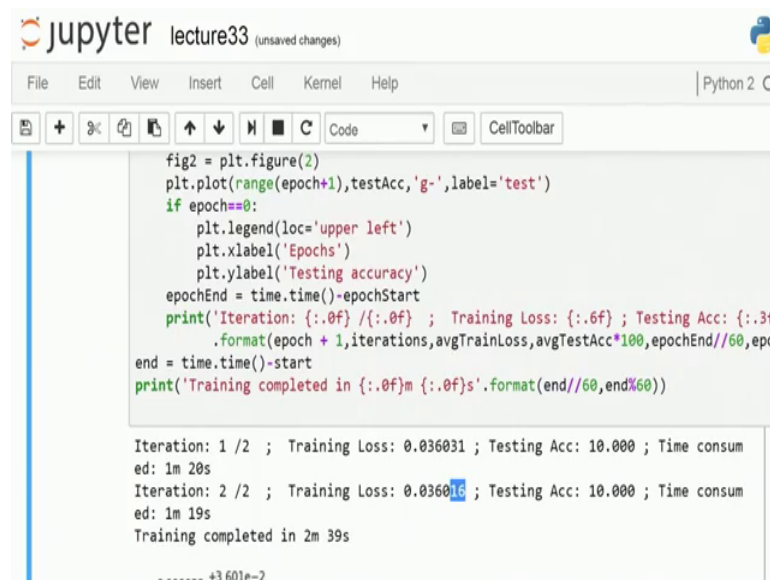
(Refer Slide Time: 15:06)



```
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} ; Testing Acc: {:.3f}
      .format(epoch + 1,iterations,avgTrainLoss,avgTestAcc*100,epochEnd//60,epochEnd//60)
end = time.time()-start
print('Training completed in {:.0f}m {:.0f}s'.format(end//60,end%60))
```

So, now that we have kept on running this part of the trainer and then we are seeing what the trainer consist of so, let us have a look into how it has trained out over there.

(Refer Slide Time: 15:13)



```
fig2 = plt.figure(2)
plt.plot(range(epoch+1),testAcc,'g-',label='test')
if epoch==0:
    plt.legend(loc='upper left')
    plt.xlabel('Epochs')
    plt.ylabel('Testing accuracy')
epochEnd = time.time()-epochStart
print('Iteration: {:.0f} / {:.0f} ; Training Loss: {:.6f} ; Testing Acc: {:.3f}
      .format(epoch + 1,iterations,avgTrainLoss,avgTestAcc*100,epochEnd//60,epochEnd//60)
end = time.time()-start
print('Training completed in {:.0f}m {:.0f}s'.format(end//60,end%60))

Iteration: 1 / 2 ; Training Loss: 0.036031 ; Testing Acc: 10.000 ; Time consumed: 1m 20s
Iteration: 2 / 2 ; Training Loss: 0.036018 ; Testing Acc: 10.000 ; Time consumed: 1m 19s
Training completed in 2m 39s

+3.601e-2
```

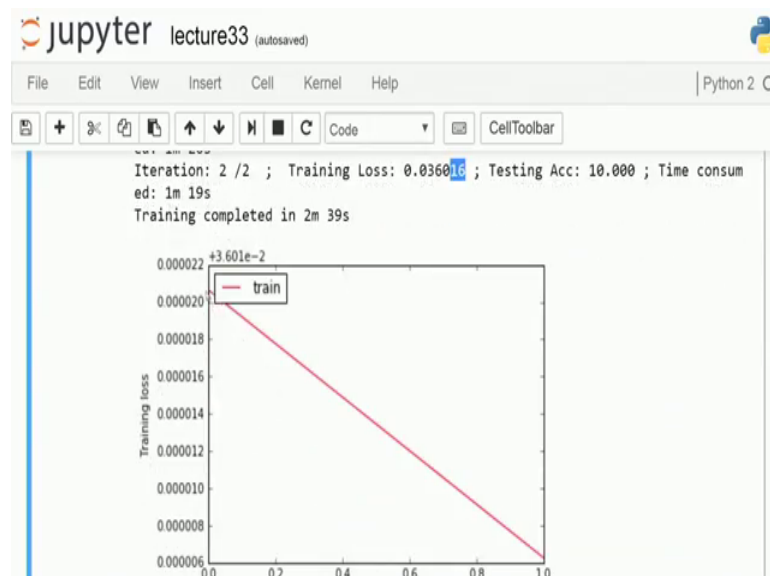
So, if you look over here we have just change it for this 2 iterations and for the sake of time it takes quite a long amount of time. And that given down that your AlexNet which you have over here, that has a significant number of convolutions which come down and then you are fully connected layer and that makes it really complicated to work it out.

So, here when I have this train down, you see that they has it been much of a change over the loss, but then and as well as the accuracy has not increase. But, given the fact that this is a very deep network which you are going to train, it will take out more amount of time in order to do that. But, the final convergence accuracy is are expected to be much better than what you had with lesser number of neurons along the depth. So, because when you have a network which is much (Refer Time: 15:59) not with so, many number of neurons over there.

So, the total number of trainable parameters is also less. The lesser the number of parameters it is much easier and faster to train it. However, the hierarchical complexity in which it can input data with lesser number of parameters is much lower than what it would need for encoding for a deeper number of network.

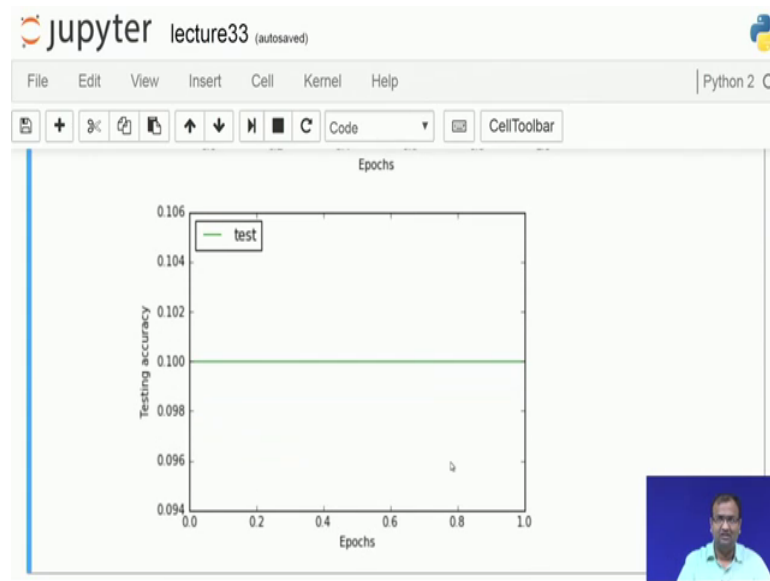
So, if we have a deeper network since you have more number of parameters over there. So, the effective space of features and then they have representations which you are learning is much higher and then that way the total saturation accuracy increases. So, you would need longer and longer to train over for a deeper network, but then the final performance always is something which is to be amazed and it is it is remarkable.

(Refer Slide Time: 16:42)



So, let us have a look into what happens on with the loss function.

(Refer Slide Time: 16:49)



So, the loss function as such is still decreasing, it is it has just operate down to epochs and it is just the start of it. So, in general I would suggest that keep on operating it over for at least 1000 epochs that might take time. And for the given the kind of a high power GPU which we have it already took us more than a minute ah. So, if you running it on CPU, then you have to sort a bear with it and keep on running for longer duration of time.

(Refer Slide Time: 17:13)

The screenshot shows a Jupyter Notebook interface with a code cell. The code is as follows:

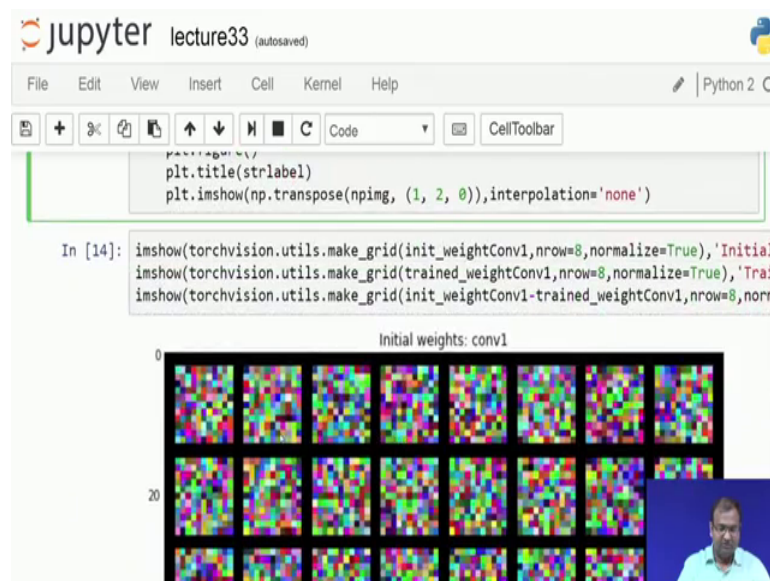
```
In [12]: # Copying trained weights for visualization
if use_gpu:
    trained_weightConv1 = copy.deepcopy(net.features[0].weight.data.cpu())
    trained_weightConv2 = copy.deepcopy(net.features[3].weight.data.cpu())
else:
    trained_weightConv1 = copy.deepcopy(net.features[0].weight.data)
    trained_weightConv2 = copy.deepcopy(net.features[3].weight.data)
```

A small video inset of a person is visible in the bottom right corner of the notebook window.

So, thought the accuracy has not changed as much over there and that is for a marginal change which has happened. Then let us look into what happens down with the weights over there.

So, what I have done over here is quite similar to as in the earlier case that after my training I just copy down my weights and then try to visualize out my weights over here.

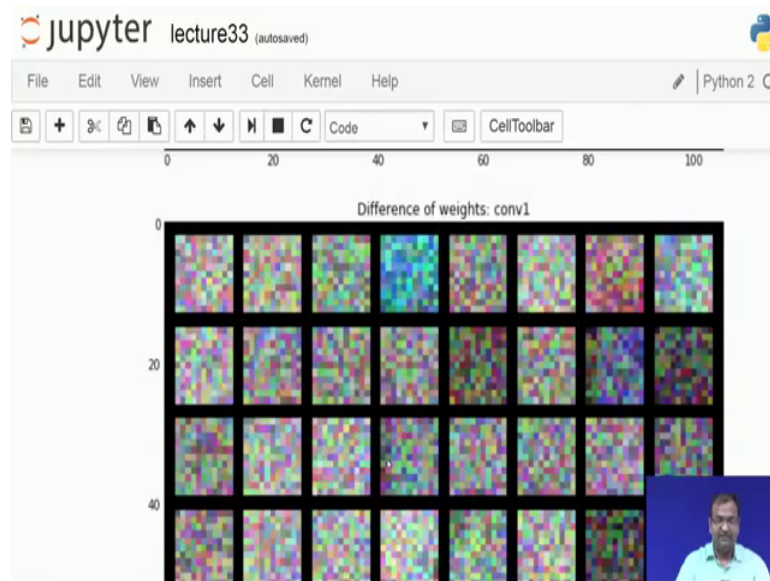
(Refer Slide Time: 17:25)



So, when I go down to my weights these are my first convolution layer. So, my first convolution layer basically has 1, 2, 3, 4, 5, 6, 7, 8 that is 64 number of kernels over there which are arranged in a stack of 8 rows and 8 number of columns for row over there. And each of these is an 11 cross 11 matrix so, let us count it out 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 and on this side you have 11. So, it is a 11 cross 11 and maps down a 3 channel input onto these weights and that is why you have a colors thing coming down over there.

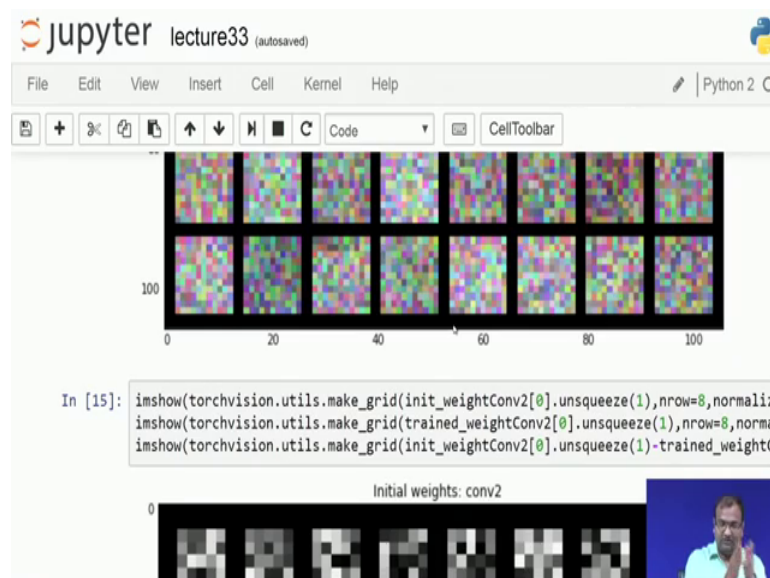
To going on the same logic this is what was the initialization the random initialization of a network which I had and this are the weights which comes down after my train. Now that there you do not see any features as such coming down and to be very very frank on that there has not been any possible large amount of learning which has happened over there. Is just been from my random to what is after 2 epochs over there.

(Refer Slide Time: 18:21)



So, if I look down into my difference you do see that some other weight are changing because there is a significant amount of differences in the weights which are coming down over there. And this is an indicator that the learning is happening and then you are not hitting down on a valley region or a certain out point. Through your accuracy gives an indicator as effects just start down at 10 percent, but then the change which is happening over it is much lower.

(Refer Slide Time: 18:41)

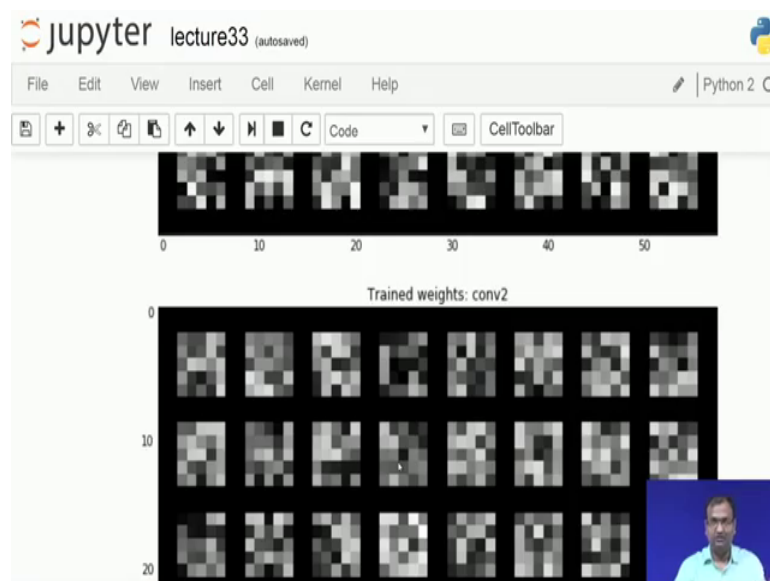


So, this was about the first layer which you had, then you have the second layer. In the second what you do connect down as that you take down this 64 channel input over there and connected down to 128 channels and then this 64. So, if we take any one of these kernels of the second convolutional layer so, it has 64 such cud 2D matrixes, which constitutes the whole kernel over there.

So, my second convolution kernel which is a 5 cross 5 in special spread has the z axis or the total number of channels over there which is equal to 64. So, what I do is I pull up one of these kernels and display all of these 64 such 5 cross 5 matrixes over there.

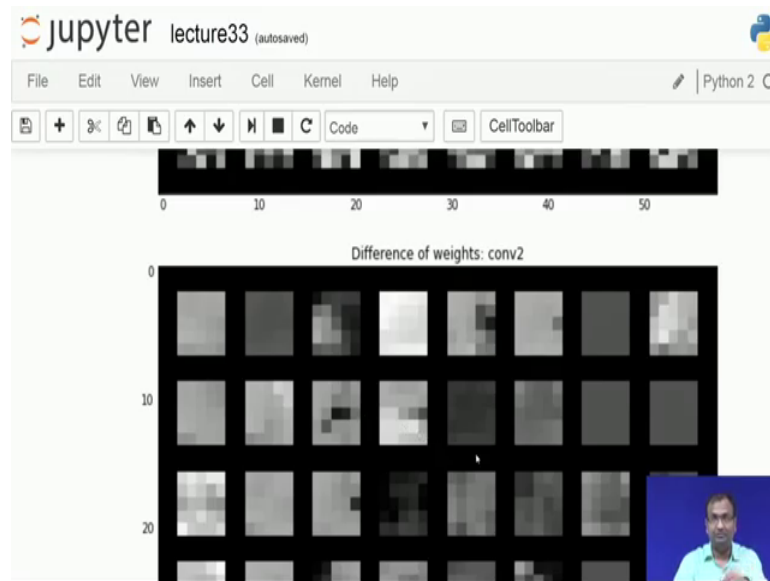
So, this is what constitutes for one of them this if you can look over here it is 1, 2, 3, 4, 5, 1, 2, 3, 4, 5 that is the 5 cross 5 matrix and there are 1, 2, 3, 4, 5, 6, 7, 8 of them and arranged 8 of those rows as well. So, that makes it a whole collection of 64 kernels coming down over there. Now, these are the kernels weights which were before the training, this is what happens after this training procedure of 2 epochs.

(Refer Slide Time: 19:42)



And if you look down and the updates which comes down over here.

(Refer Slide Time: 19:48)



You see that there has been some change, even in the second place though some of them have not changed at all. If you look down over here this is practically 0 these are the ones which have ideally like all the weights have made that might of a positive change. This is where all the weights have made significant amount of negative change as well as and some of them are randomly changing, some of them have directional changes as you see over here some of them are like this. So, this is just to give you an indication of how these changes are happening and that the whole network is training down over there as well.

So, this was one of the first kind of a deep convolutional neural network for a practical application using larger size images. So, then subsequent ones which we do will now be looking into those largest size images of 224 cross 224 and then how to work around with them and to give you word of advice over there.

So, if you have access to a higher performance computing over there or cloud axis in some sort and please try to use it, please try to use a GPU if you have some resource available. Or if you are trying to running on your laptop or standard desktop CPU's, then please bear on and I have give it some time thought it will not be blasting of your memory, but it would just be taking a that bit longer to work it out then.

And that is what you have to do given the number of calculations and the number of parameters you are training. So, with that keep on stay tuned and stay excited as we go on to the next version of even deeper networks.