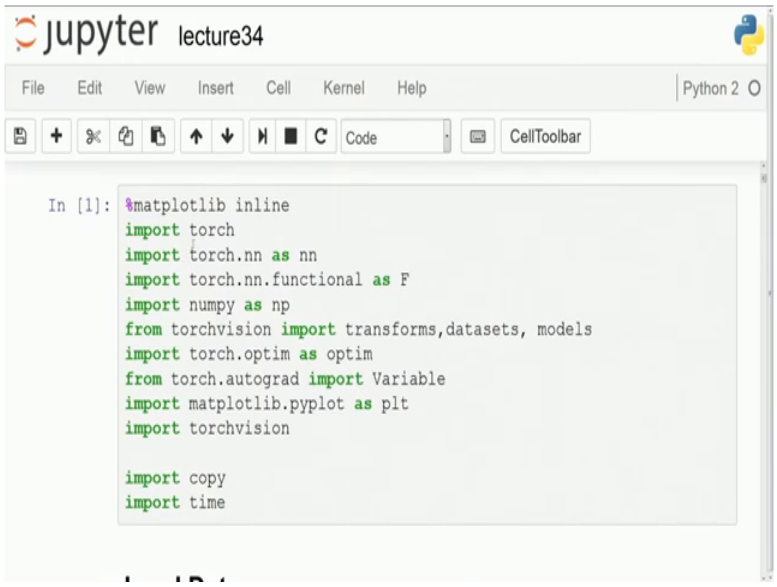
Deep Learning For Visual Computing
Prof. Debdoot Sheet
Department of Electrical Engineering
Indian Institute of Technology, Kharagpur

Lecture-34
VGGNet

So, welcome to today's lecture. And while we have done in the earlier classes on some of these introductory versions of convolutional neural networks, and trying to get down into what is an abstraction of trying to understand a deep convolutional neural network.

(Refer Slide Time: 00:31)



```
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
```

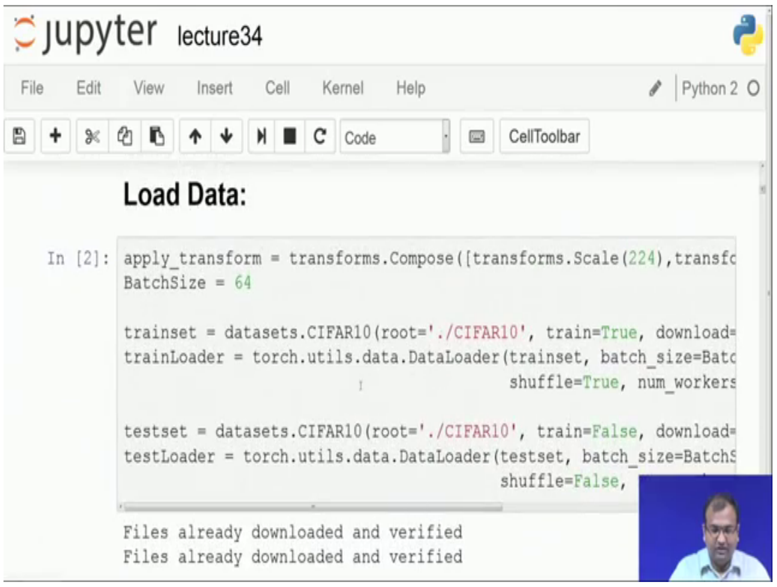
Here, today we would be starting with something called as a VGG net that is a short acronym for the group from where originate out. So, this originates out from the visual geometry group, and that is where the name comes down as VGG. So, while in the earlier lectures we had already studied about how the different characteristics of the network as well as the, architecture and how it is linked out of the data flow happens from. Today what I am going to do predominantly is actually to go through this whole set of codes over here which will be showing you how this whole architecture is implemented, and then the learning as progresses through it ok.

So, let us let us just go through how it works up. So, the first part of here is just a set of your header files which we are going to use for all the purpose. I am just it is the same set of headers which are present on in all the earlier course, which we are going to reuse over here as well.

Now, if you see that this is a run version which I am using and one of the reasons is that this network being quite deep. It takes us substantial amount of time in order to compute if we are just going to run through the epochs that is runs in a few minutes in order of a few minutes actually typically about 15 to 20 minutes over there. And in order to keep it just precise and within the time limit of our recordings I am just using a network which is already trained out and the states of it which is preserved over here.

So, that that does not change anything from your side because you will be provided with an untrained network over there, which you will have to actually train it start training it from the scratch, and then you would see how the network gets strained and its performance increases over there.

(Refer Slide Time: 02:10)

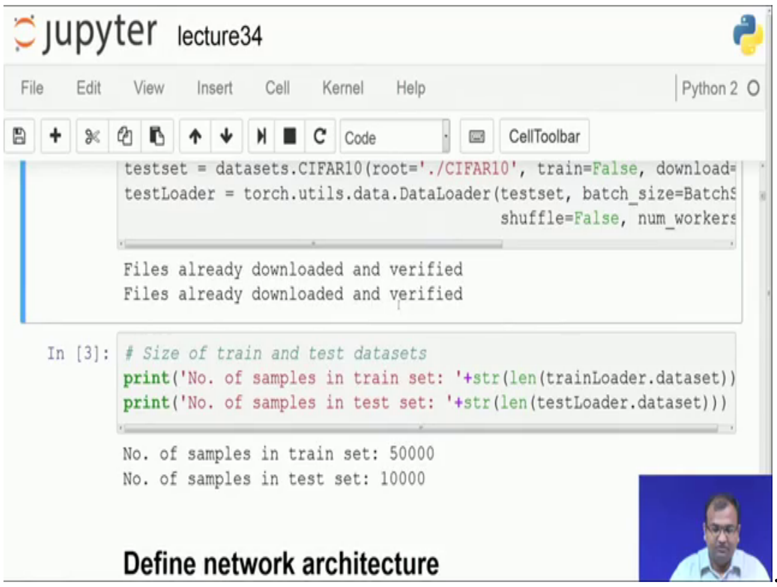


Load Data:

```
In [2]: apply_transform = transforms.Compose([transforms.Scale(224),transfc
        BatchSize = 64

        trainset = datasets.CIFAR10(root='./CIFAR10', train=True, download=
        trainLoader = torch.utils.data.DataLoader(trainset, batch_size=Batc
                                                  shuffle=True, num_workers

        testset = datasets.CIFAR10(root='./CIFAR10', train=False, download=
        testLoader = torch.utils.data.DataLoader(testset, batch_size=BatchS
                                                 shuffle=False,

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

So, this is the first part of it which is just my header files over there, which I need to copy down for my own library calls whichever I want to do. The next part is to go around and do with the data. So, on my data what I use is the standard c file. So, as in you had seen in the earlier lecture on Alex net where we had all the Alex net was trained down for a image net kind of classification problems which has one thousand categories in which you are to classify and all the images are of size 224 plus 224. So, here we are going to use down CIFAR. And since the images are of size 32 plus 32, so we need to rescale it out. And I am using the same kind of our transformation scale factor over here which will be applying their transformations on the datas which is getting loaded over here. So, that is just to keep it aligned with what we have done in the earlier one as well.

(Refer Slide Time: 02:53)



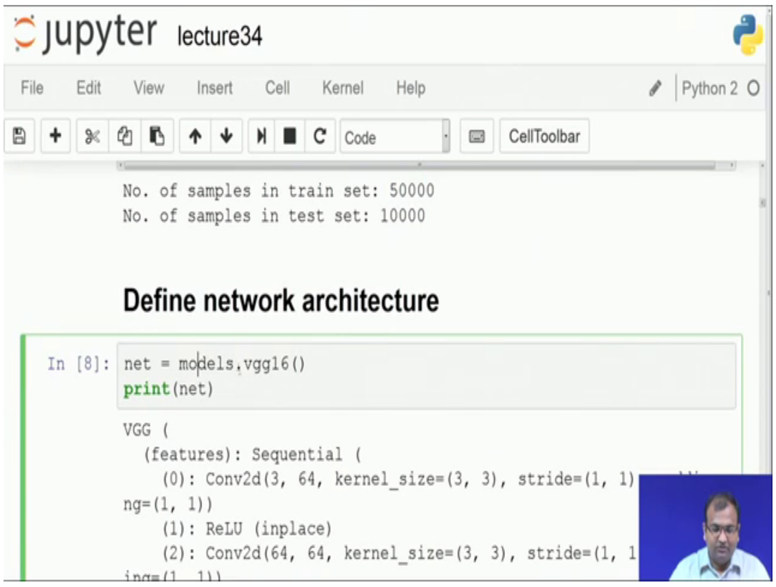
```
testset = datasets.CIFAR10(root='./CIFAR10', train=False, download=
testLoader = torch.utils.data.DataLoader(testset, batch_size=BatchS
                                         shuffle=False, num_workers

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

And now that is just my train loader and my test loader, so both my data files get loaded over there and then you see that the files are present, so it is verified.

(Refer Slide Time: 03:02)



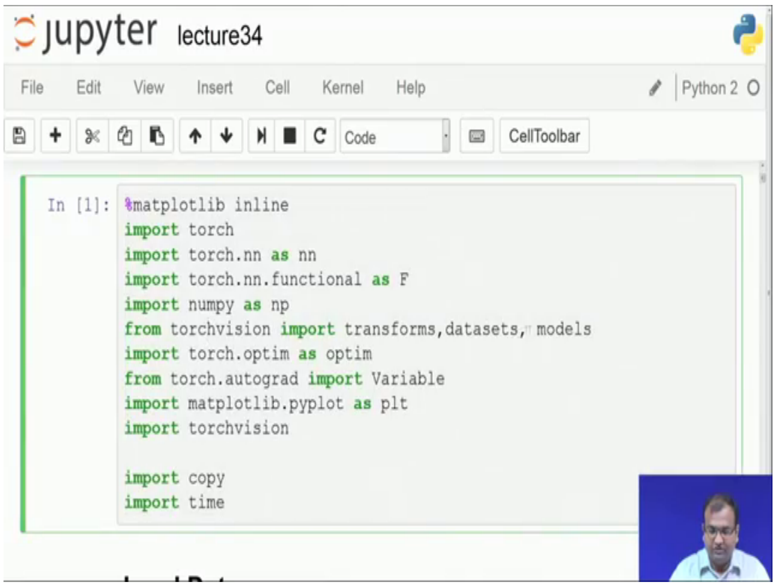
```
No. of samples in train set: 50000
No. of samples in test set: 10000
```

Define network architecture

```
In [8]: net = models.vgg16()
        print(net)

        VGG (
          (features): Sequential (
            (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1)
        ng=(1, 1))
            (1): ReLU (inplace)
            (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1
```

And finally, we run down and check down what is the total number of samples, so that is quite same as we had for our CIFAR10. So, 50,000 images on the training sample over there, and 10,000 images for my testing and that is what I am going to start working on. Now, you had seen down in Alex net it was quite easy because you could actually call down your models over here and that is a library within torchvision.

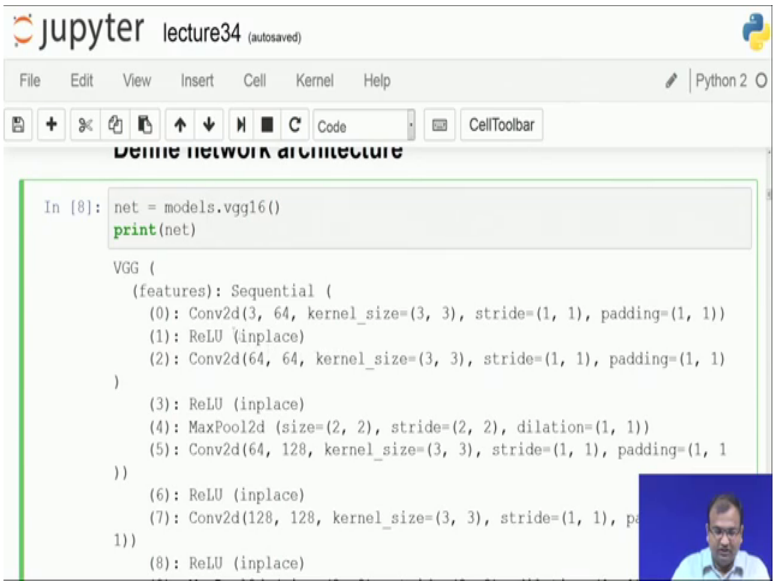
(Refer Slide Time: 03:25)



```
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
```

So, within torch vision you have a definition architecture over there available within the function called as models. So, here I just need to invoke my call to VGG 16. Now, VGG16 is basically that layer which has sixteen learnable parametric layers in total, and then that is the one which we are going to choose.

(Refer Slide Time: 03:31)


jupyter lecture34 (autosaved)
File Edit View Insert Cell Kernel Help
Python 2
Code
CellToolbar

```
In [8]: net = models.vgg16()
        print(net)

        VGG (
          (features): Sequential (
            (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (1): ReLU (inplace)
            (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)
        )
            (3): ReLU (inplace)
            (4): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
            (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1
        ))
            (6): ReLU (inplace)
            (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), pa
        1))
            (8): ReLU (inplace)
```

So, once we have this model loaded down and then I can print and look through it. So, if you see through it you would see your it is sort of like one by one blocks which are connected down as in your VGG. So, the first block was a battery of two subsequent 2D convolutions and finally after that you have max pooling operation coming down and then again 2D convolution.

So, the first one is what converts down from 3 channels onto 64 channels, so that is the battery of filters which you have within the first convolution layer. Each filter is of size three cross three with a stride of one and a padding of one which means that whatever is the size of the input which goes over here, the same is the size of the output which comes out from this 2D.

This is what happens in essence over here. Now, then you have a non-linear function which is ReLU and subsequent to that you again have a 2D convolution which maps down 64 layers which come as an output from the earlier 2D convolution onto another 64 layer. So, there are 64 such unique kernels which get defined over here. And your kernels are again three cross three in the specials size, it has a stride of one and a padding of one as goes down good.

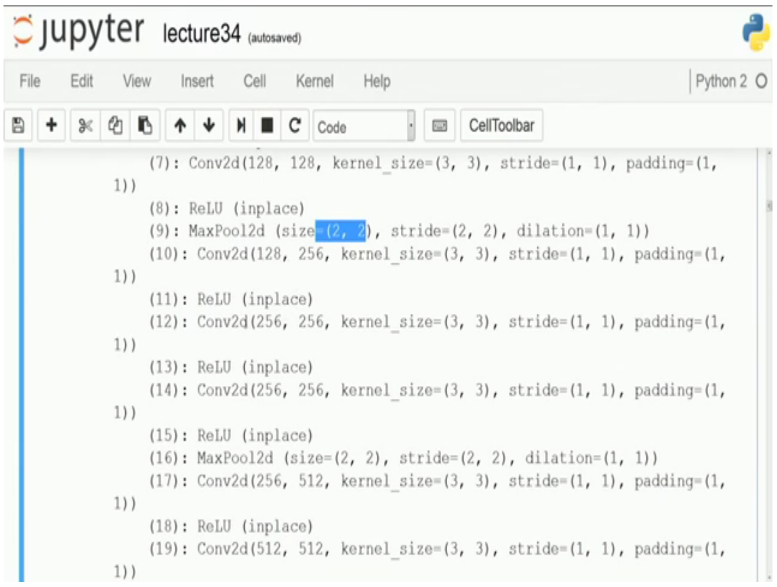
So, in the next version what I do is we have ReLu again put down as a non-linearity, and then you have a max pooling operation coming down over here which is the two cross two max pudding. And then subsequent to that we have 2D convolution and then again a

ReLu and then again a 2D convolution and then a ReLu. So, this is the second battery which comes down whereas, this max pooling has now reduced on the size by doing a so it uses a max full kernel of two cross two and with the stride of two; and for that reason it just reduces to half of the size.

So, from 224 across 224 which was the spatial size and the output from here you get it reduce to 112 plus 112 at the result of this particular block which is being called down and then that 120, so 112 cross 112 sized image is or that block over there. So, there is it is 64 number of channels and 112 cross 122 with the spatial span over there.

Now, that is again convolved with 128 unique kernels in order to get 128 channels or slices coming down on the output. Each kernel is of size three cross three with stride of one comma one, and a padding of one in place and then you have your ReLu coming down as a non-linear function.

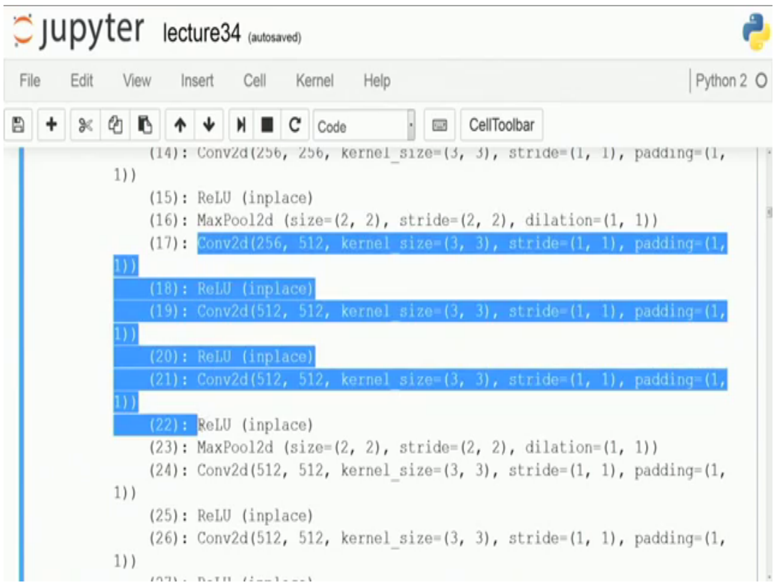
(Refer Slide Time: 06:14)


jupyter lecture34 (autosaved)
File Edit View Insert Cell Kernel Help
Python 2
Code
CellToolbar

```
        (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
        (8): ReLU (inplace)
        (9): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
        (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
        (11): ReLU (inplace)
        (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
        (13): ReLU (inplace)
        (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
        (15): ReLU (inplace)
        (16): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
        (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
        (18): ReLU (inplace)
        (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
```

Now, we do not have any further max pooling over here, but that again succeeds by another battery of 128 convolution kernels, each of size three cross three, and stride of one comma one and padding of one comma one. So, that would mean that the resultant of this one is also of a special size of 112 cross 112, subsequent to that you have your ReLu and then a max pooling which brings it down to half of the size. So, that makes it 56 cross 56 because now the size and style. And then you keep on doing that.

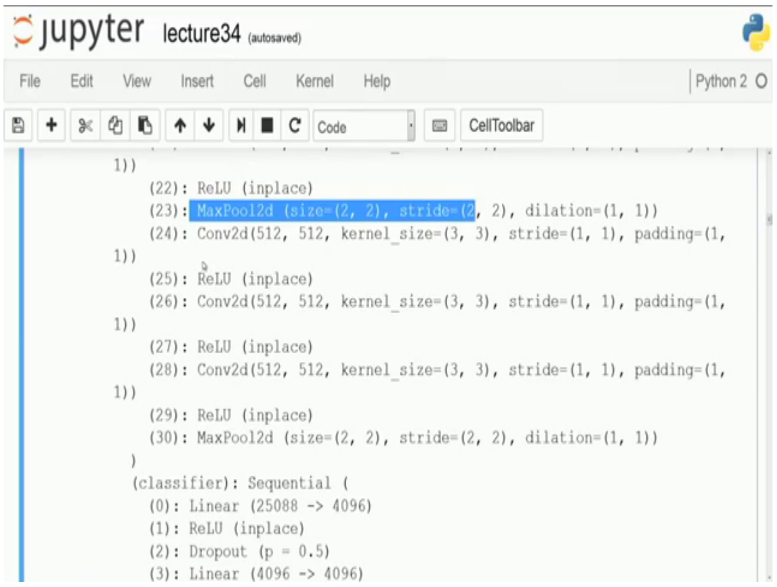
(Refer Slide Time: 06:55)


jupyter lecture34 (autosaved)
File Edit View Insert Cell Kernel Help
Python 2
Code
CellToolbar

```
        (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
        (15): ReLU (inplace)
        (16): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
        (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
        (18): ReLU (inplace)
        (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
        (20): ReLU (inplace)
        (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
        (22): ReLU (inplace)
        (23): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
        (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
        (25): ReLU (inplace)
        (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
```

So, over here you again have in the next battery, you have three convolution kernels which would come down, and subsequent to that you have your next max pooling.

(Refer Slide Time: 07:00)

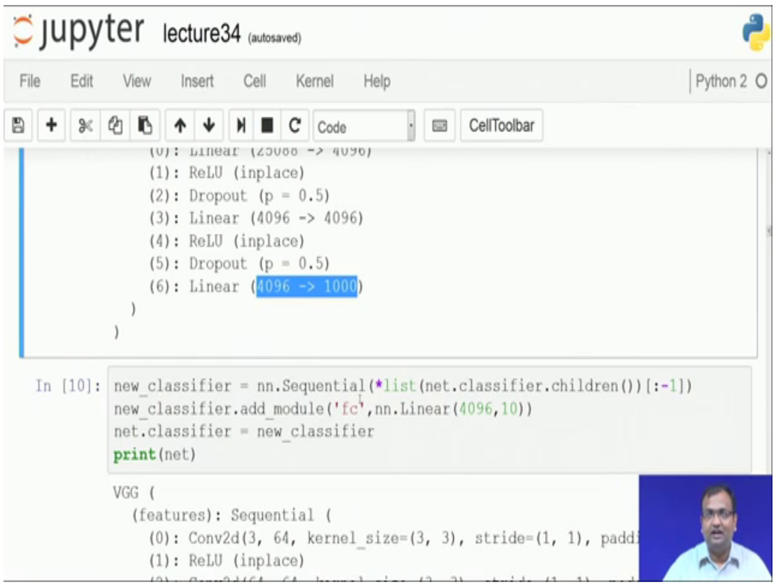

jupyter lecture34 (autosaved)
File Edit View Insert Cell Kernel Help
Python 2
Code
CellToolbar

```
1))
        (22): ReLU (inplace)
        (23): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
        (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
        (25): ReLU (inplace)
        (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
        (27): ReLU (inplace)
        (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
        (29): ReLU (inplace)
        (30): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
      )
      (classifier): Sequential (
        (0): Linear (25088 -> 4096)
        (1): ReLU (inplace)
        (2): Dropout (p = 0.5)
        (3): Linear (4096 -> 4096)
```

Now, following that you again have three convolution kernels coming down over here, and then again a max pulling. Subsequent to that you again have three convolution kernels and then max pooling ok. Now, over here as a resultant which comes down this has 512 such channel channels across the volume which is coming out over there. And then in total if you look into the total number of neurons present over there, so that total number of neurons after linearization is 25088, 25088 such neutrons. So, this in the first fully connected layer is what connects it down to 4096 neurons.

Then we have a dropout which uses a 50 percent dropout ratio over here. These 4096 neurons are again connected down to 4096 neurons, and then you have a 50 percent dropout coming down over here. And finally, these 4096 neutrons are again connected down to 1000 neurons. Now, this is your VGG net in the classical style.

(Refer Slide Time: 07:44)


jupyter lecture34 (autosaved)
File Edit View Insert Cell Kernel Help
Python 2
Code
CellToolbar

```
      (0): Linear (25088 -> 4096)
      (1): ReLU (inplace)
      (2): Dropout (p = 0.5)
      (3): Linear (4096 -> 4096)
      (4): ReLU (inplace)
      (5): Dropout (p = 0.5)
      (6): Linear (4096 -> 1000)
    )
  )
```

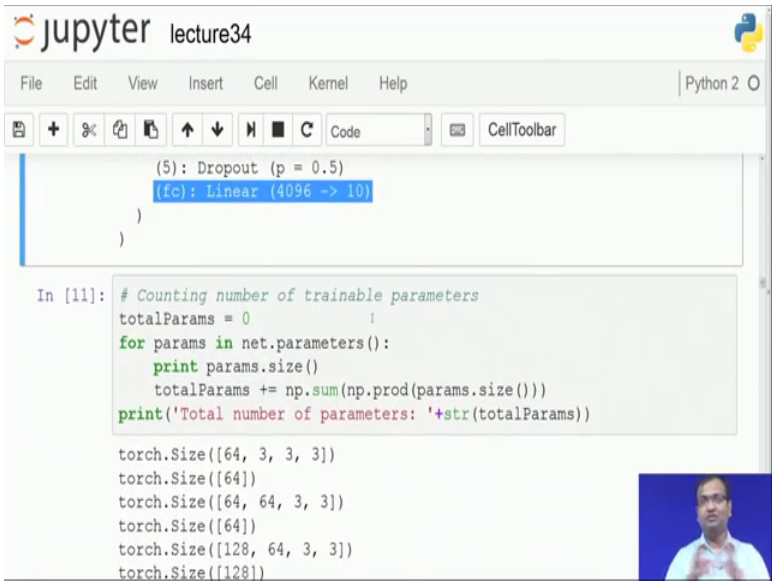
```
In [10]: new_classifier = nn.Sequential(*list(net.classifier.children())[:-1])
         new_classifier.add_module('fc',nn.Linear(4096,10))
         net.classifier = new_classifier
         print(net)
```

```
VGG (
  (features): Sequential (
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), paddi
    (1): ReLU (inplace)
```

However, the kind of data which we are using in CIFAR10 that is a 10 class problem. So, if you just have 1000 neurons over there, it does not make sense because it will not be learning for 990 neurons it will not have any data to train it done because those classes are just missing over there. So, in order to do that what we are going to do is just a tag bit of modification this is still untrained. So, it does not create any sort of a problem inside over there as well. And for that what we do is we keep on modifying this one.

So, first point is just delete out this last layer over here, and then introduce a fully connected connection between 4096 neurons to 10 neurons and that is what will solve it out. So, that is what is exactly introduced over here. And now if you print your whole network, so you would be able to see that the last layer is now which connects on 4096 neurons onto 10 neurons and quite simple and sweet solution to this problem ok.

(Refer Slide Time: 08:46)

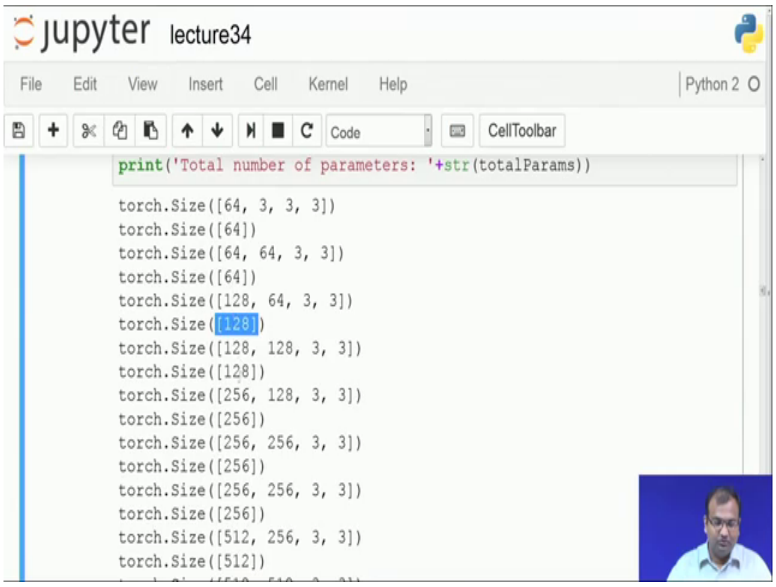


```
        (5): Dropout (p = 0.5)
        (fc): Linear (4096 -> 10)
      )
    )
```

```
In [11]: # 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, 3, 3])
         torch.Size([64])
         torch.Size([64, 64, 3, 3])
         torch.Size([64])
         torch.Size([128, 64, 3, 3])
         torch.Size([128])
```

So, now we have our VGG net which is modified onto take down and classify up to 10 neurons. Now, with this modified architecture which just classifies it onto 10 classes and not 1000 classes as far as you know originally measure problem, we need to find out what is the total number of parameters which goes down. So, for that I will be looking into my first layer. So, my first layer basically has three cross three kernels, and the Z direction or the number of channels in that kernel is also three because that matches down with the number of channels present down in my input image ok.

(Refer Slide Time: 09:14)

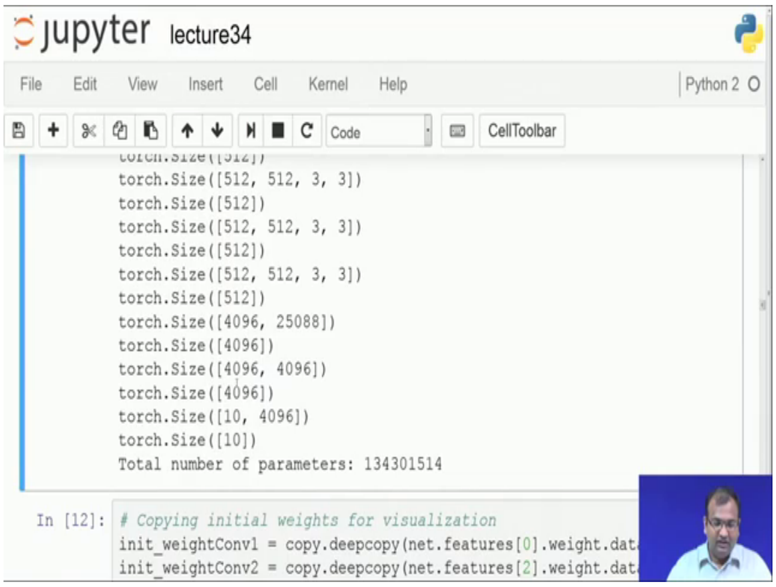


```
print('Total number of parameters: '+str(totalParams))

torch.Size([64, 3, 3, 3])
torch.Size([64])
torch.Size([64, 64, 3, 3])
torch.Size([64])
torch.Size([128, 64, 3, 3])
torch.Size([128])
torch.Size([128, 128, 3, 3])
torch.Size([128])
torch.Size([256, 128, 3, 3])
torch.Size([256])
torch.Size([256, 256, 3, 3])
torch.Size([256])
torch.Size([256, 256, 3, 3])
torch.Size([256])
torch.Size([512, 256, 3, 3])
torch.Size([512])
```

And there are 64 such unique number of corners, which are present over there, so that makes it 64 into 3 into 3 ok. Now, with each of these 64 conducts, you also have a bias associated with it and that is this 64 number of biases which comes down. So, this is for my first layer ok. My second convolution layer will have the input over there has a depth or the number of channels of the input data is 64, the spatial span is three and three still.

So, this becomes 64 into 3 into 3. And I have 64 such number of unique kernels coming down, so that makes it 64 into 64 into 3 into 3, and I have one bias associated with each of these kernels over there, so that makes it 64 additional biases. And then this is what is added down. Subsequent to that, I connect down 64 channels on 228 channels with the 3 cross 3 convolution kernel and for each channel I have 128 biases and so on and so forth I can keep on going till I reach down a point over here which is my end of convolution layers.

(Refer Slide Time: 10:13)

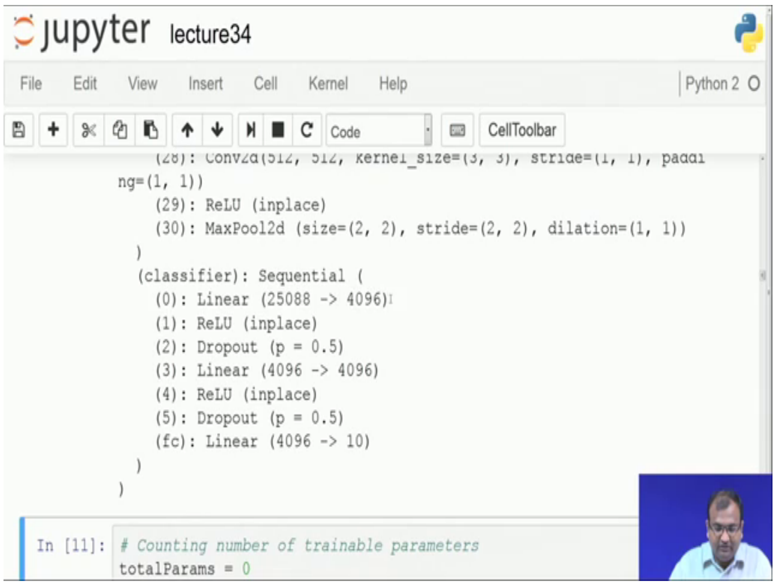


```
torch.Size([512])
torch.Size([512, 512, 3, 3])
torch.Size([512])
torch.Size([512, 512, 3, 3])
torch.Size([512])
torch.Size([512, 512, 3, 3])
torch.Size([512])
torch.Size([4096, 25088])
torch.Size([4096])
torch.Size([4096, 4096])
torch.Size([4096])
torch.Size([10, 4096])
torch.Size([10])
Total number of parameters: 134301514
```

```
In [12]: # Copying initial weights for visualization
         init_weightConv1 = copy.deepcopy(net.features[0].weight.dat
         init_weightConv2 = copy.deepcopy(net.features[2].weight.dat
```

So, beyond that what I have is a connection of 25088 neurons 24096 neurons and for each of in the fully connected layer, so that is the first fully connected layer which comes down.

(Refer Slide Time: 10:27)



```
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), paddi
ng=(1, 1))
    (29): ReLU (inplace)
    (30): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
  )
  (classifier): Sequential (
    (0): Linear (25088 -> 4096)
    (1): ReLU (inplace)
    (2): Dropout (p = 0.5)
    (3): Linear (4096 -> 4096)
    (4): ReLU (inplace)
    (5): Dropout (p = 0.5)
    (fc): Linear (4096 -> 10)
  )
)
```

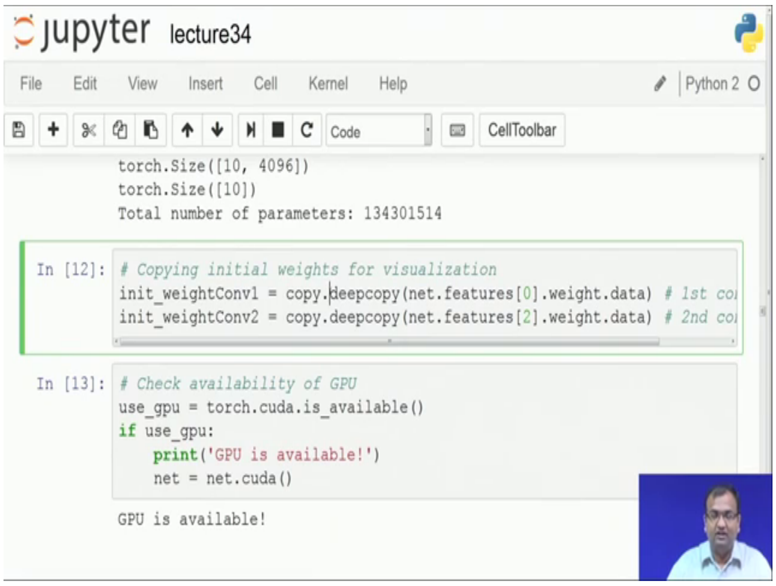
```
In [11]: # Counting number of trainable parameters
         totalParams = 0
```

So, you have 25088 neurons connected down to 4096 neuron. And each of these 4096 neuron also has one-one bias coming down over there and that is this additional 4096 which gets added. Now, 4096 neurons are again connected to 4096 neurons, and then you have an additional bias of 4096 number of elements. The final one is which connects on 4096 neurons to 10 neurons. And for each of the neuron you have an additional bias which comes down over here.

Now, if you take down a total of all of these parameters, this is what it comes down. So, that is closely about 134 million parameters. So, this is number so somewhere over here is your dot if you would like to put that. So, this becomes 134 dot 301514 into 10 power of 6 or 130 roughly about 134 million parameters over there.

If it was an original VGG net which has 1000 neurons to which it was mapped, so obviously that would be a larger value. Then instead of this n over here, this becomes 1000, the number of biases over here also become 1000, so that is the change which would happen if we move it over to a standard VGG net for the image net kind of a problem. Now, once this is done, my network is completely established. Now, what I would like to do is that just copy down my random bits and keep it for my use if I want to do anything.

(Refer Slide Time: 11:46)



```
torch.Size([10, 4096])
torch.Size([10])
Total number of parameters: 134301514
```

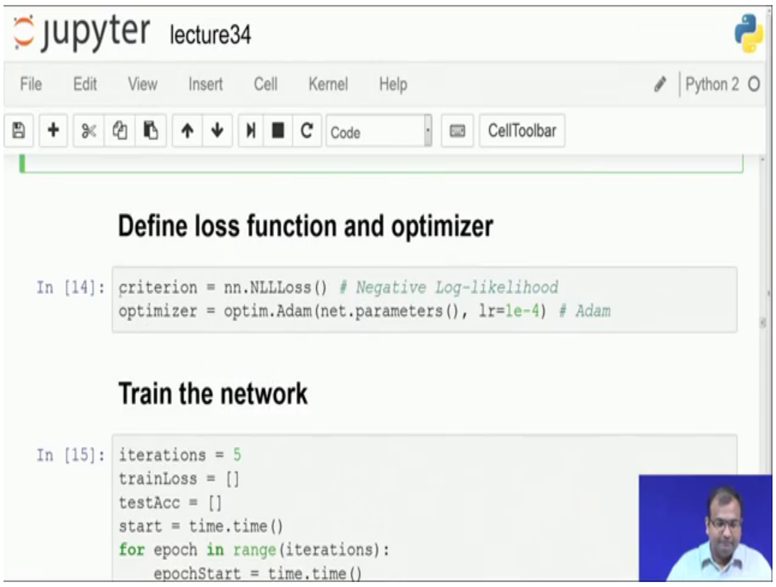
In [12]:
```
# Copying initial weights for visualization
init_weightConv1 = copy.deepcopy(net.features[0].weight.data) # 1st co
init_weightConv2 = copy.deepcopy(net.features[2].weight.data) # 2nd co
```

In [13]:
```
# Check availability of GPU
use_gpu = torch.cuda.is_available()
if use_gpu:
    print('GPU is available!')
    net = net.cuda()
```

```
GPU is available!
```

And then typically as with the earlier examples I was actually visualizing a dot and trying to show you how these things are changing over there ok. Now, following this what I do is actually check down if my GPU is available. Now, if my GPU is available then I can get it running and started on the GPU itself.

(Refer Slide Time: 12:03)



Define loss function and optimizer

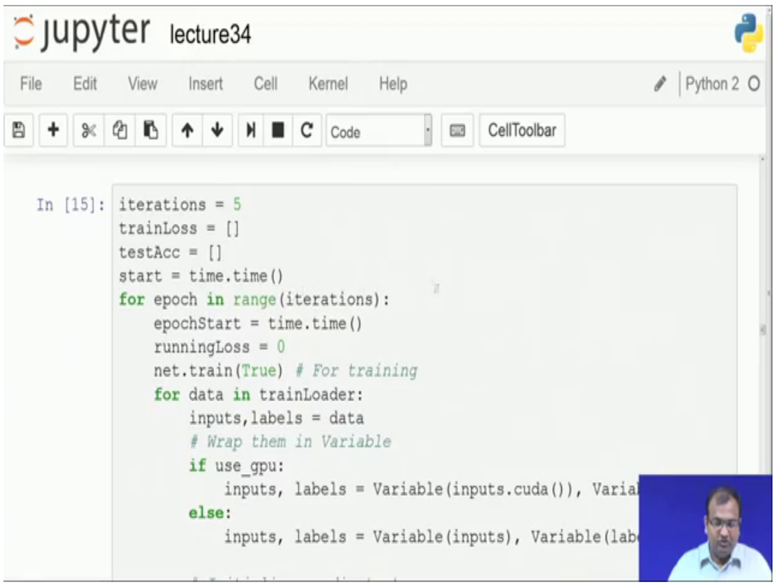
In [14]:
```
criterion = nn.NLLLoss() # Negative Log-likelihood
optimizer = optim.Adam(net.parameters(), lr=1e-4) # Adam
```

Train the network

In [15]:
```
iterations = 5
trainLoss = []
testAcc = []
start = time.time()
for epoch in range(iterations):
    epochStart = time.time()
```

So, once having done that the next is the classical thing which is to look down into what kind of a problem I am trying to solve. So, here the kind of a problem is a classification problem. So, the cost function which I am going to use is going to be a classification cost function. And for that purpose we make use of the negative log likelihood cost function over here. And then finally, I have my optimizer which is to be used and this optimizer which I make use of over here is the Adam optimizer plain simple Adam adaptive momentum optimization over here. Now, this solves my part of the loss function and the optimizes which I am supposed to use.

(Refer Slide Time: 12:40)

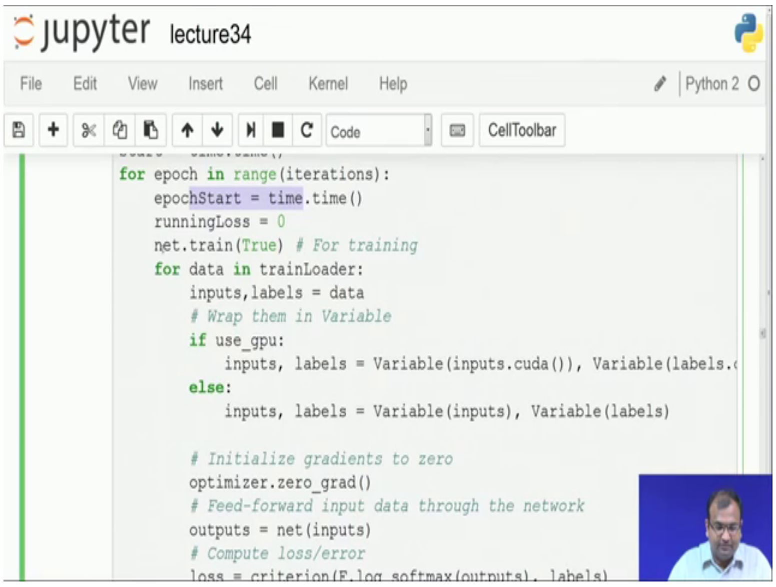


```
In [15]: iterations = 5
         trainLoss = []
         testAcc = []
         start = time.time()
         for epoch in range(iterations):
             epochStart = time.time()
             runningLoss = 0
             net.train(True) # For training
             for data in trainLoader:
                 inputs,labels = data
                 # Wrap them in Variable
                 if use_gpu:
                     inputs, labels = Variable(inputs.cuda()), Varia
                 else:
                     inputs, labels = Variable(inputs), Variable(lab
```

The next part is to look through my network ok. So, what I do with in my training part over that. So, till here you had your network defined the total number of parameters taken down your GPU availability check, and then you our training and loss function is defined. And the next part is quite straightforward that you are going to look down through the whole training process.

Now, the training process is in no way different. And by now you should have been getting used to an accustomed that the best part of all of this is that it is too modular in structure. So, you can just have all of these definitions taken down and kept down over there. And based on whatever is your network definition in your net just changing that definition is what solves the rest of your problems over here so not nothing major to change as such. So, over here what I do is I start my iterations or my number of epochs over there within each epoch I just initialize my timer.

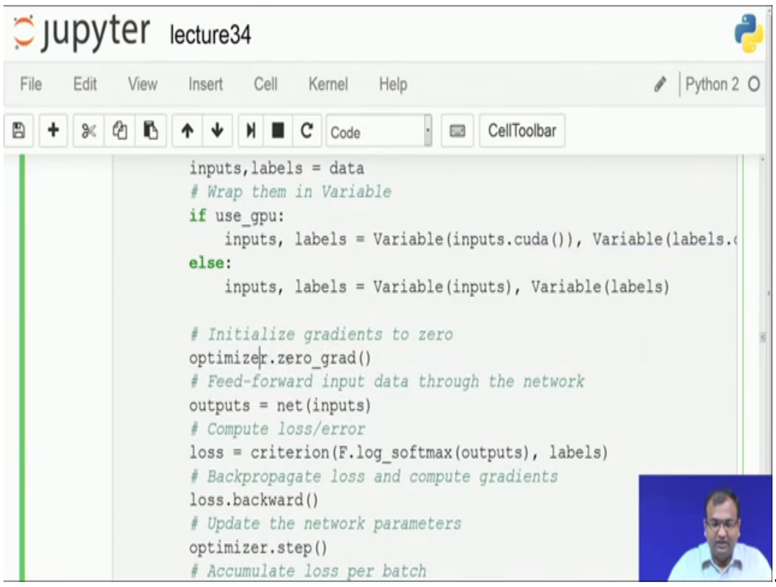
(Refer Slide Time: 13:31)



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

And then what I do is I start my training data loaders. Now, if my GPU is available then its type casted as a cuda variable, so that it is available on the GPU as well.

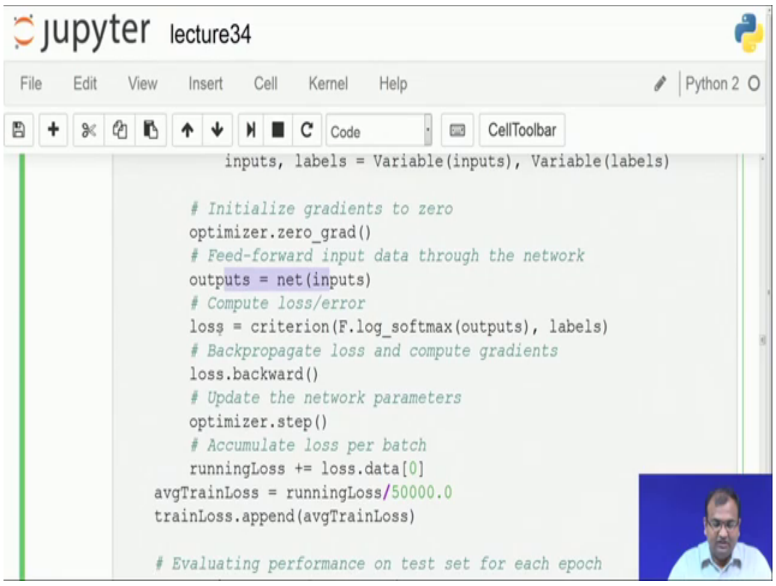
(Refer Slide Time: 13:45)



```
        inputs,labels = data
        # Wrap them in Variable
        if use_gpu:
            inputs, labels = Variable(inputs.cuda()), Variable(labels.
        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
        optimizer.step()
        # Accumulate loss per batch
```

And then after that what I do is I zero down all the gradients within my optimizer, the first step of just zeroing down gradient so that we do not have any residual gradients pumping through the network over there. The next part is doing a feed forward over the network which comes down over here.

(Refer Slide Time: 13:56)

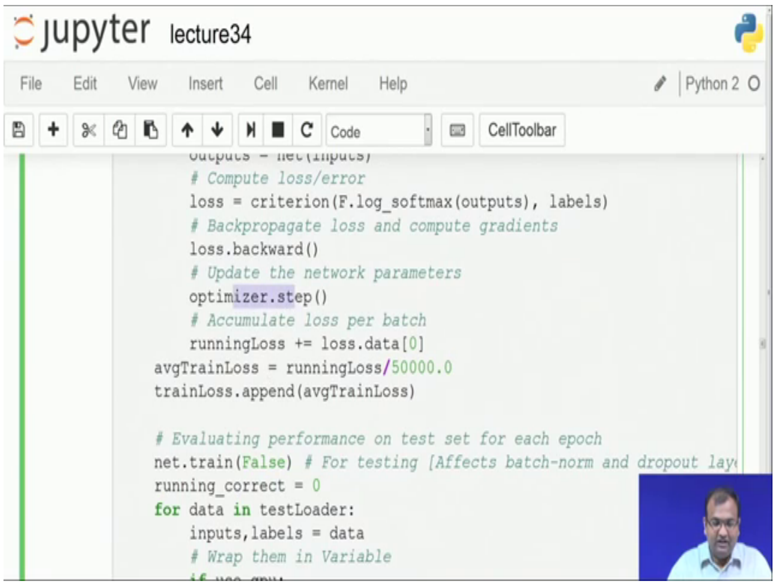


```
            inputs, labels = Variable(inputs), Variable(labels)

        # 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
        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
```

Once your feed forward is done, you have these outputs available over there you can find out your loss function. Now, your criterion over here being a negative log likelihood. So, we needed to take a logs of max transfer function over there as well and then you can convert and then you can actually find out what is the computed value of the loss between your output and the labels.

So, whatever thing was predicted over there with whether it was matching down your labels or not ok. So, once your loss is there, next we need to find out what is nabla of the loss from nabla of g or the first derivative of the cost function and that is what we need and that is what is computed using the backward operator over here.

(Refer Slide Time: 14:34)

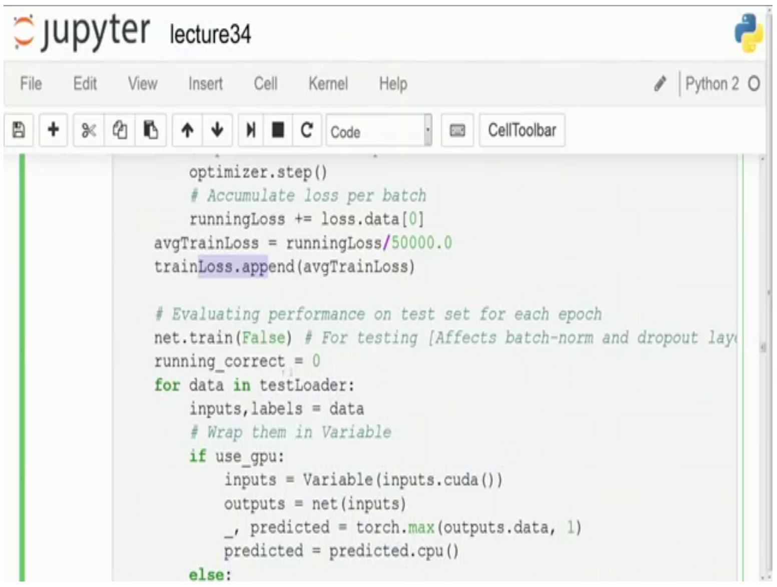


```
            # Compute loss/error
            loss = criterion(F.log_softmax(outputs), labels)
            # Backpropagate loss and compute gradients
            loss.backward()
            # Update the network parameters
            optimizer.step()
            # Accumulate loss per batch
            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 lay
        running_correct = 0
        for data in testLoader:
            inputs,labels = data
            # Wrap them in Variable
```

And finally, once the loss is computed, you need to do your back propagation, so that is with the optimizer dot step running around over there. And then over each batch, we are supposed to just sum up or what is the total loss coming down ok. And then the average loss is what is divided by the total batch size which gives you the standard average loss coming down.

And then what you can do is create an array of training loss, so that at every epoch you know exactly how the loss was moving down, and what was the loss at the end of every epoch during an epoch of training.

(Refer Slide Time: 15:08)



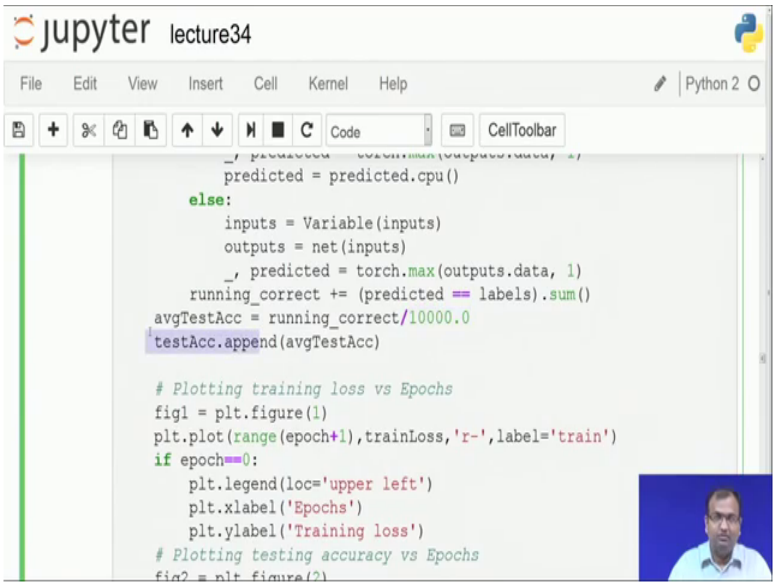
```
            optimizer.step()
            # Accumulate loss per batch
            runningLoss += loss.data[0]
        avgTrainLoss = runningLoss/50000.0
        trainLoss.append(avgTrainLoss)

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

Now, once that is done, the next part is to look into the validation. And what we make use of over here is the standard validation set which we have access to. So, this is the test data set which is available over there which has not been used during the training process. So, what we do is we just run down our data within the test loaded part over there. And then if my GPU is available then type casted onto my GPU things. And then once the type casted variable is available in terms of cuda or available on the GPU memory, I can do a forward pass over the network and then get done whatever is my predicted value of the class which comes down and that is the result of this max operator coming down over here.

Now, once it is predicted down, what I do is just convert it onto my CPU and bring it back over here such that I can do the rest of the calculations. The rest of the calculation is pretty simple that I need to compare whatever is my predicted value of the label whether that is the exact value of the label on the data which I am using over there. And if it is so then that is correct so just pick a sum over all of them and that is going to give me the total number of correct things I have done and that is out of one thou[sand] that is out of the 10,000 samples which are available in my testing data.

So, if I divide this by 10,000, I get down my average accuracy coming down over here and then I just decide to put down my average testing accuracy poll epoch in terms of an array which I can use it for subsequent inferencing over there.

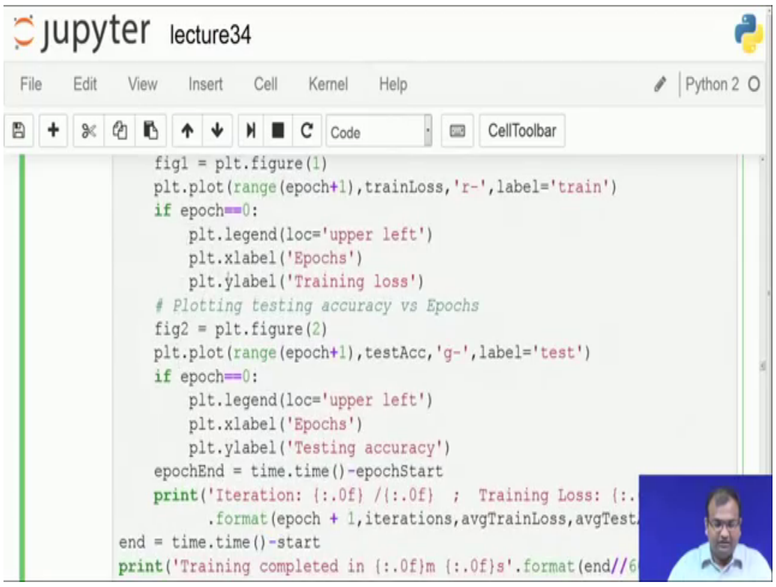
(Refer Slide Time: 16:22)



```
                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
```

Now, the next part is pretty simple. So, this is just to plot down your training errors as well as your validation accuracy over there. So, while your training error is expected to go down, your validation accuracy over time is expected to go up as it comes down, and then this just keeps on printing it out.

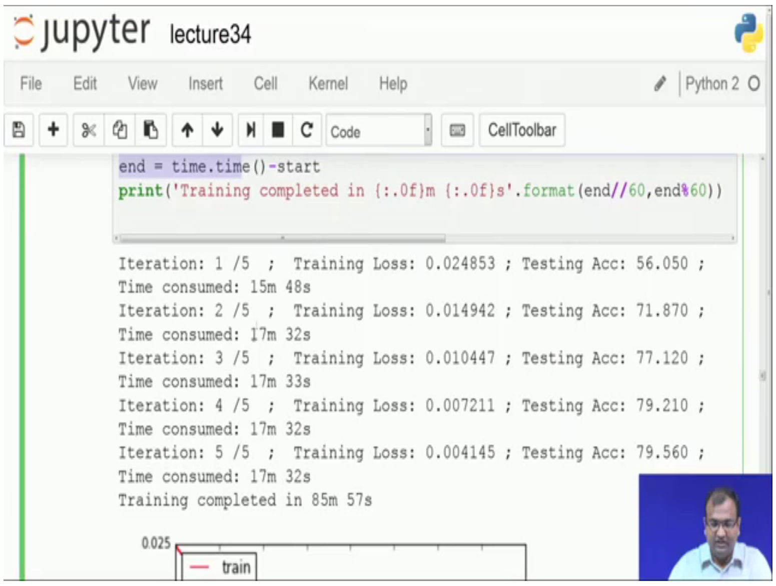
(Refer Slide Time: 16:31)



```
        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: {:.
              .format(epoch + 1,iterations,avgTrainLoss,avgTest
    end = time.time()-start
    print('Training completed in {:.0f}m {:.0f}s'.format(end//6
```

So, now once you train it down, you would see that it does take a significant amount of time.

(Refer Slide Time: 16:46)



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

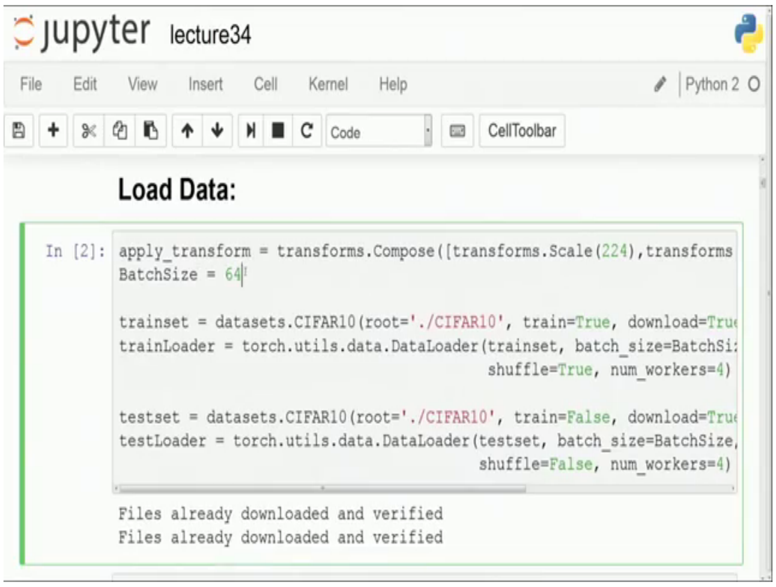
```
Iteration: 1 /5  ;  Training Loss: 0.024853 ; Testing Acc: 56.050 ;
Time consumed: 15m 48s
Iteration: 2 /5  ;  Training Loss: 0.014942 ; Testing Acc: 71.870 ;
Time consumed: 17m 32s
Iteration: 3 /5  ;  Training Loss: 0.010447 ; Testing Acc: 77.120 ;
Time consumed: 17m 33s
Iteration: 4 /5  ;  Training Loss: 0.007211 ; Testing Acc: 79.210 ;
Time consumed: 17m 32s
Iteration: 5 /5  ;  Training Loss: 0.004145 ; Testing Acc: 79.560 ;
Time consumed: 17m 32s
Training completed in 85m 57s
```

So, an epoch takes roughly like 15 minutes or 17 minutes of time going down over there and for that reason we had just chosen down only five epochs. Now, you need to keep in mind so networks like Alex net or LeNets, they are much smaller in size and then the number of compute which happened down is much smaller. Whereas, over here you have large number of computes, the total number of parameters which you are learning the total number of parameters which get updated over there is also large. And it says for this reason that you are going to take down more amount of time deeper networks are computationally expensive.

But then later on when we get into studying down even more deeper networks like residual networks and other soft like you will realize that it is it is not always necessary that you need to have a lot of parameters only if you have a deeper network it is just by a mathematical scaling of the width of the network which is also a critical factor in terms of determining the total number of parameters which you have to tune as you go across the network.

Now, if you look into it you start with the training loss initially of 0.02; within the end of first epoch it drops down almost to half of it and then it keeps on steadily the declining over them. The accuracy starts roughly at 56 percent, but you need to keep in mind that when the network was initially started down and then we had batch updates which were going down. Now, my batch size over here which I had defined within my loader somewhere over here is 64.

(Refer Slide Time: 18:11)



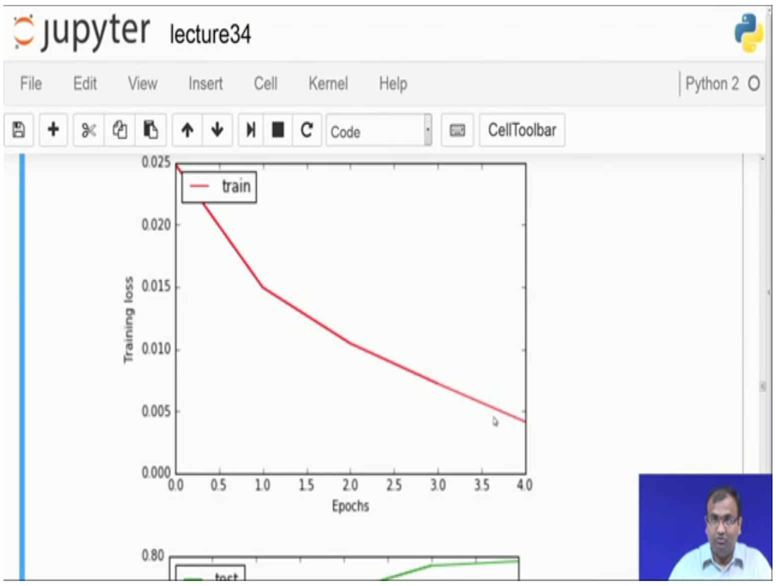
Load Data:

```
In [2]: apply_transform = transforms.Compose([transforms.Scale(224),transforms
        BatchSize = 64

        trainset = datasets.CIFAR10(root='./CIFAR10', train=True, download=Tru
        trainLoader = torch.utils.data.DataLoader(trainset, batch_size=BatchSi
                                                  shuffle=True, num_workers=4)

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

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

Now, I have 50,000 such samples available and I am loading those into number of patches, which is 64. So, I get down basically 50,000 divided by 64 number of updates happening within each epoch. So, after that many number of updates, so that is that is roughly 50,000 divided by 64 is about 8,000 of updates which go down over there and with that many updates happening down within each epoch, you already raised to or accuracy of 56 percent at the end of the first epoch when it stops training ok.

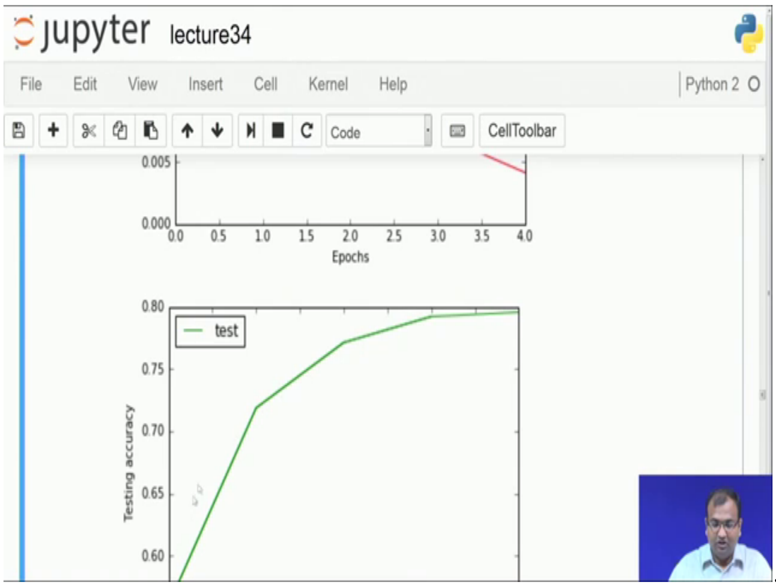
Then you go down to the next epoch and you see at the end of the next epoch, you have a jump which is almost 16 on almost 15 percent above the earlier case and goes down to about 71 percent. From there, it has a 6 percent jump going down to 77 percent from there a 2 percent jump to 79 percent. And then it somewhat comes down to a point of where it might be hitting down saturation. So, this is the point where you can still keep on training playing around with the learning rate over there, may be reduced to learning it and then you would see this increasing over there.

(Refer Slide Time: 19:21)



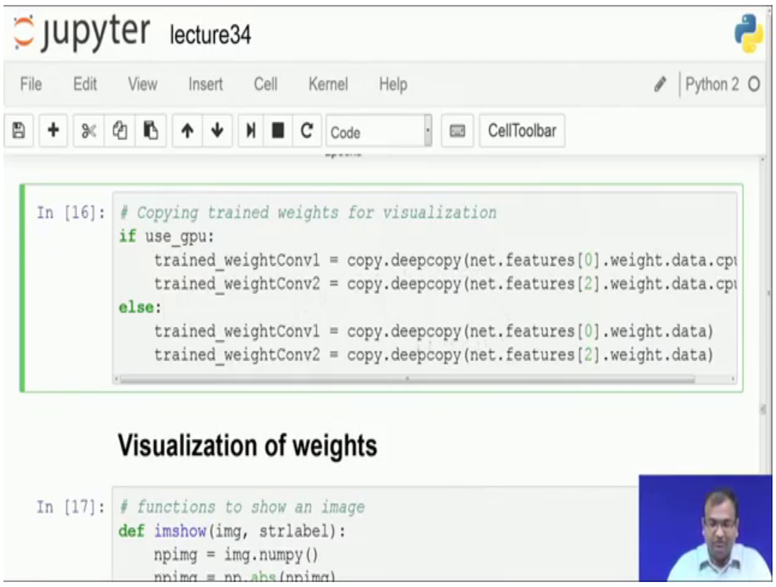
So, what happens down on my last curve is my last curve is still monotonically looking like it is going to decrease significantly.

(Refer Slide Time: 19:22)



Although my accuracy comes down as if it is going to saturate out; however, this is a whole trade off. So, my slope of accuracy increased does not become so high; whereas, my errors which are back avoiding and still tuning out my weights are significant. And then what I need to do over here is basically fix down the learning rate such that it keeps on coming down to its actual global minimize sort of getting stuck in the local minima over there so that is what we had already discussed earlier.

(Refer Slide Time: 19:52)



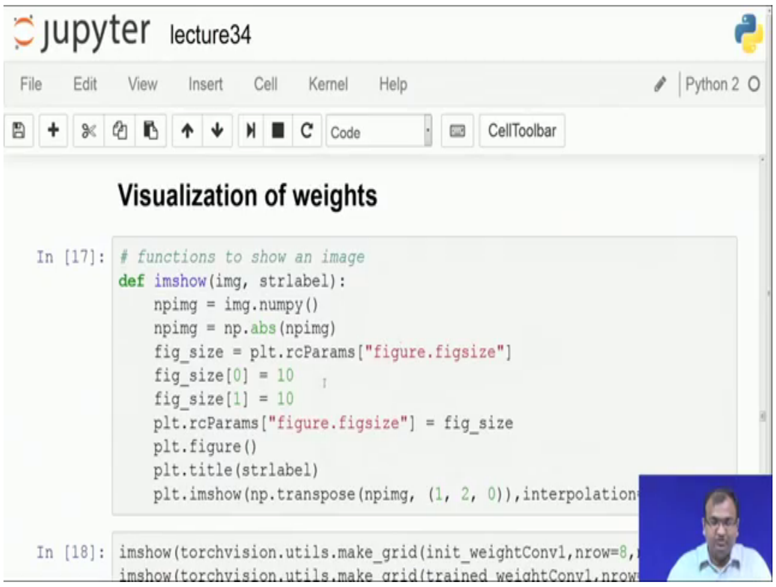
```
In [16]: # Copying trained weights for visualization
         if use_gpu:
             trained_weightConv1 = copy.deepcopy(net.features[0].weight.data.cpu
             trained_weightConv2 = copy.deepcopy(net.features[2].weight.data.cpu
         else:
             trained_weightConv1 = copy.deepcopy(net.features[0].weight.data)
             trained_weightConv2 = copy.deepcopy(net.features[2].weight.data)
```

Visualization of weights

```
In [17]: # functions to show an image
         def imshow(img, strlabel):
             npimg = img.numpy()
```

And then finally, what I do is on my model which has been trained over here, I just copy down my weights, and keep them and then try to visualize it out.

(Refer Slide Time: 20:00)

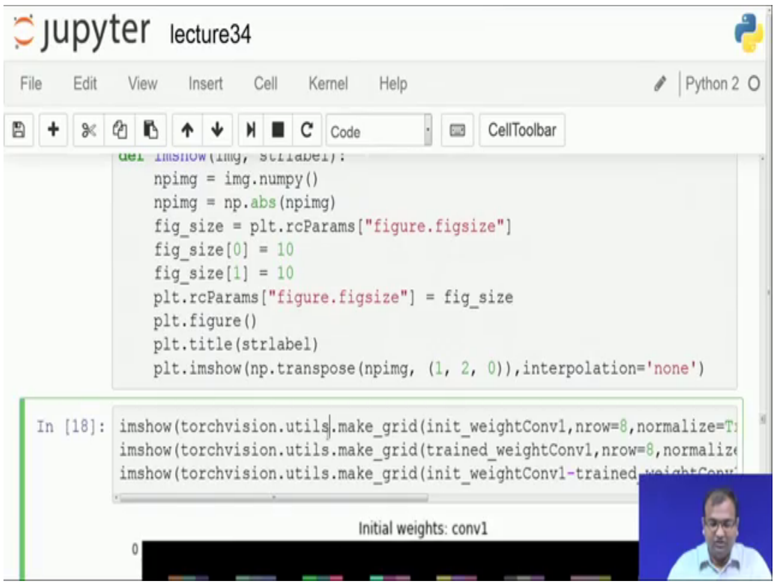


Visualization of weights

```
In [17]: # functions to show an image
         def imshow(img, strlabel):
             npimg = img.numpy()
             npimg = np.abs(npimg)
             fig_size = plt.rcParams["figure.figsize"]
             fig_size[0] = 10
             fig_size[1] = 10
             plt.rcParams["figure.figsize"] = fig_size
             plt.figure()
             plt.title(strlabel)
             plt.imshow(np.transpose(npimg, (1, 2, 0)),interpolation

In [18]: imshow(torchvision.utils.make_grid(init_weightConv1,nrow=8,
         imshow(torchvision.utils.make_grid(trained_weightConv1,nrow
```

So, my visualization routine is the same as we had done in the earlier case.

(Refer Slide Time: 20:05)

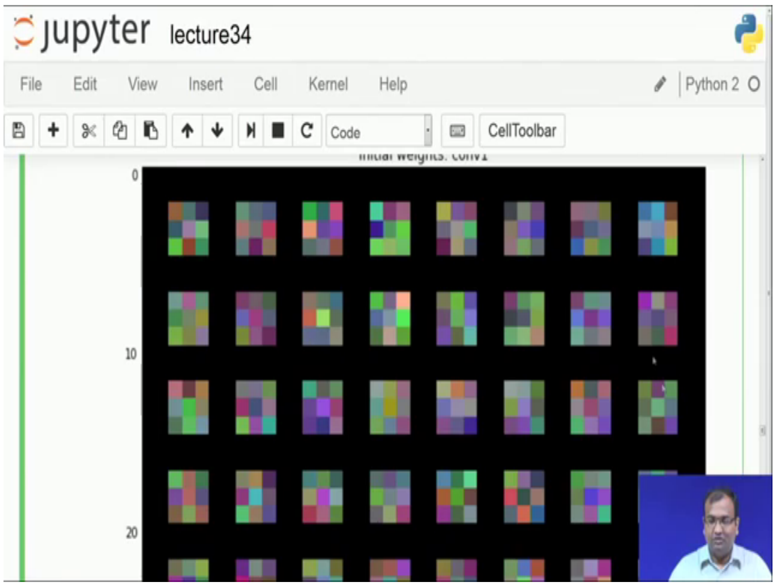


```
def imshow(img, strlabel):
    npimg = img.numpy()
    npimg = np.abs(npimg)
    fig_size = plt.rcParams["figure.figsize"]
    fig_size[0] = 10
    fig_size[1] = 10
    plt.rcParams["figure.figsize"] = fig_size
    plt.figure()
    plt.title(strlabel)
    plt.imshow(np.transpose(npimg, (1, 2, 0)),interpolation='none')
```

```
In [18]: imshow(torchvision.utils.make_grid(init_weightConv1,nrow=8,normalize=T
         imshow(torchvision.utils.make_grid(trained_weightConv1,nrow=8,normaliz
         imshow(torchvision.utils.make_grid(init_weightConv1-trained
```

And the first model is to look down into this first set of convolution layer; So, the first convolution which connects down your image space onto 64 channels over there.

(Refer Slide Time: 20:16)

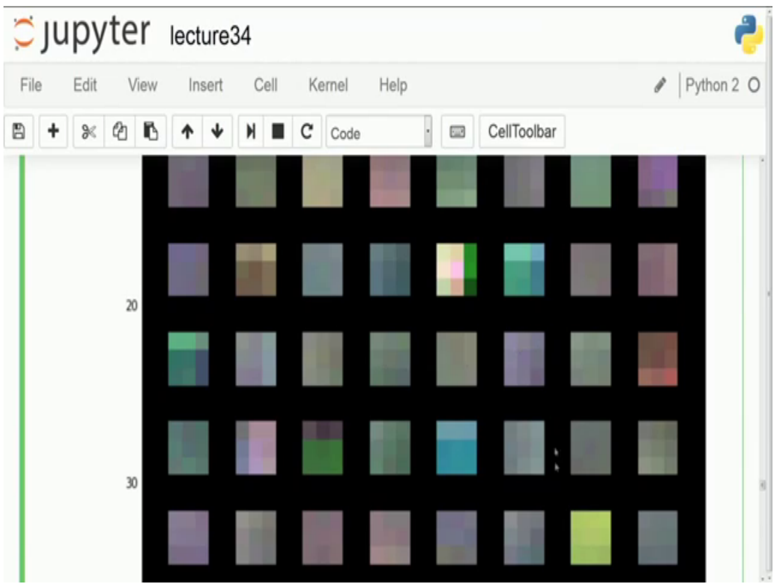


So, what I do technically is each is a 3 cross 3 kernel and that is why you have this 3 cross 3 patches. And then you have 1, 2, 3, 4, 5, 6, 7, 8 patches over here. And similarly you have 8 number of rows coming down as well and that makes a total number of 64 patches which corresponds to 64 unique kernels which has been learnt by the first convolution layer over there. And each takes in at gb or three channel inputs ok.

Now, from there these are the weights which were at the start of the training or what was randomly available down to everyone for use. Now, this is what happens at the end of training off all of these five epochs. By the time it has reached almost till 80 percent of an accuracy.

Now, it still does not quite make sense because you do not you are not able to see down figures, if you had larger size kernels, you might be able to make sense of what kind of wavelets these are. But these are some sort of feature extractors quite similar to convolutional extract feature extraction with wavelets, which are quite useful and these exist from the color space as well. So, but looking down into my weights, I can see that there have been decays there have been certain number of weights which had a different kind of a change.

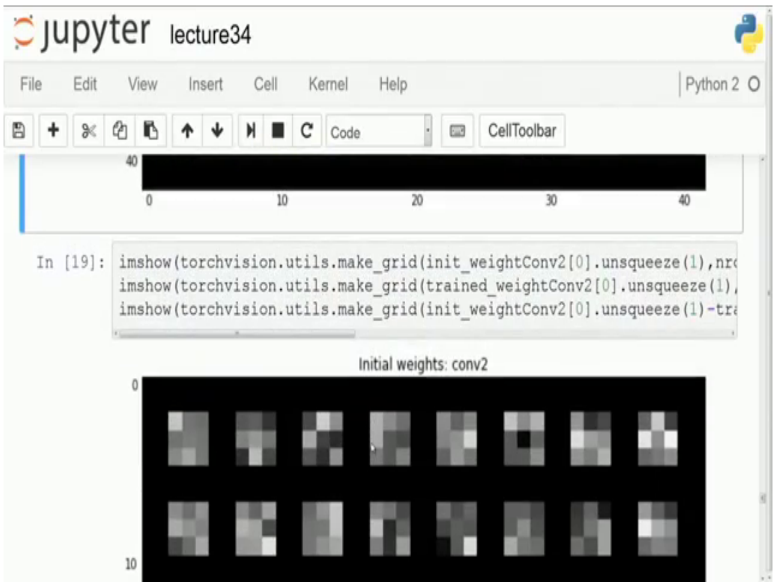
(Refer Slide Time: 21:26)



So, they were not shades of the same color which were changing, but they actually end up getting a different color as well so that is that is what we have over here in this one in this one, in this one, this one and then you see significant amount of change coming down. And then this in total makes down that the change which happens, although you do not see it evidently in terms of getting down very crisp or kind of a structure, but although the structure is not very crisp and geometrically what can be defined down, but what it learns is necessary in order to actually wire down all the neurons which are characteristic of a certain kind of an object. And that is the reason why this group of neurons would be helping you in classifying the whole process.

So, later on in the lectures, when we go down we will also be looking down how to track down on how these neuron activations are, and then what is the area from where a certain kind of an activation is coming down in order to find out what regions of an image are what are very significant of a particular kind of a class of object. And can we actually go down to understanding from a classification problem like which attributes of an image as they appear visually, and what is important and significant in order to classify it belonging to that class.

(Refer Slide Time: 22:33)

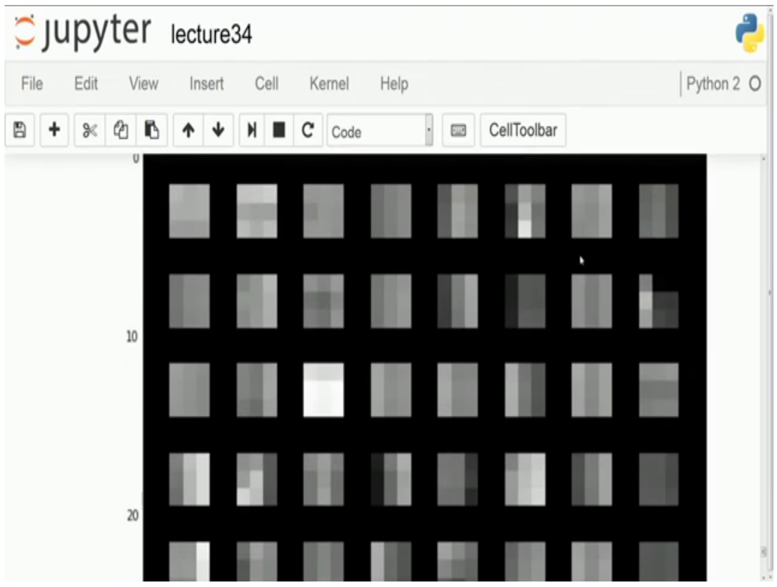


```
In [19]: imshow(torchvision.utils.make_grid(init_weightConv2[0].unsqueeze(1),nr
         imshow(torchvision.utils.make_grid(trained_weightConv2[0].unsqueeze(1),
         imshow(torchvision.utils.make_grid(init_weightConv2[0].unsqueeze(1)-tr
```

Now, this is for my first layer over here. Then I go into looking down into my second convolution layer and what I choose over here is to take down my first kernel of my second convolution layer ok. Now, on the my first kernel of my second convolution layer, I have 64 channel output which comes down from my earlier convolution layer. And all of these 64 are mapped down to a kernel over here.

So, there is technically this kernel is a 64 cross 3 cross 3 in size. And what I choose to do is I take down one of these kernels of the first cornel over here, and then I just display each of these channels as individual blocks over here. So, that is how you see them in grey scale shades over here. So, there are 1, 2, 3, 4, 5, 6, 7, 8 of them, and there are eight rows over there and each of these is a 3 cross 3 that is for the purpose of visualization.

(Refer Slide Time: 23:21)



So, this is my initially random light randomized things. This is what comes down after training over five epochs. And then if we look into our changes or the differences between them, then this is the kind of difference which comes down. So, all of them have been exposed to certain kind of a thing and in fact, for a lot of them you do see a gradient kind of a behavior. So, this is where there are almost horizontal gradients coming into play down a significant role in how they are changing this is where you have some sort of a vertical gradient as well.

So, these in total is what happens down when training it down with a VGG in it for the first few layers. And although it takes a lot of time, and then it does have a lot of parameters and subsequently the number of computations which you do, but it is it is always nonetheless fine. And one of these models is what was a ruling model for a longer period of time you have other options of going down to a VGG 19 as well, so that is up to your exploration.

So, I leave it up to you what you need to basically change is going to the documentation of torch vision and look into your models part over there. And then once you have it, so what you can change is here where you are just referencing it out. So, if you want to go down with the VGG 19, you can just write it as 19. You go down with any other models available over there you can just pull and do. So, in the subsequent ones where we do some of these very standard models and trying to understand their implement issues while we have also already covered down their theory aspects over there. They will be getting you into details of how to recall over there and how to modify and these are all models remember which are not trained initially, these were all randomly initialized and then available from scratch.

Whereas, on the models over there on the standard release library you also have trained models which means that they have been already been trained by imagenet for a large number of epochs by the time they reach down the saturation accuracy of somewhere more than 90 percent.

So, somewhere about 95, 97 percent over there and those models with the train weights are also kept down for use. And that is in a later point of time, when we go into understanding domain adaptation and transfer learning is where we will be making use of them; So, till then stay tuned and while we resume back for more deeper networks as well.