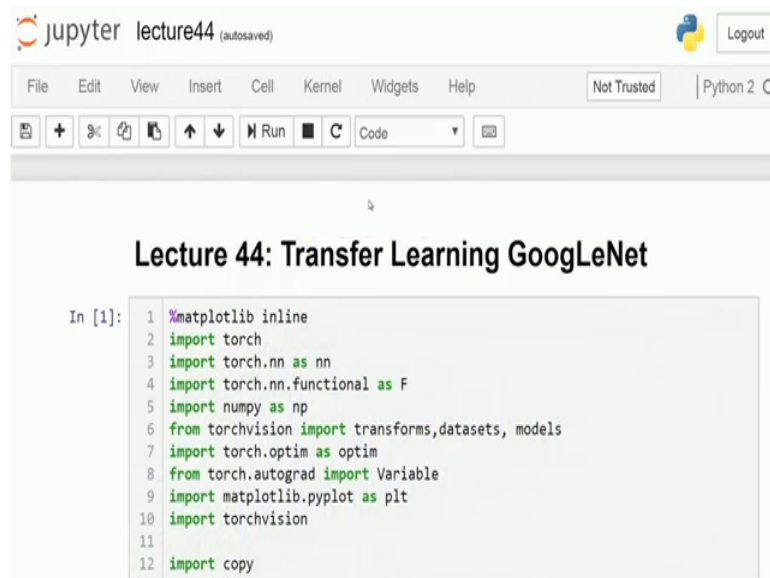


Deep Learning for Visual Computing
Prof. Debdoot Sheet
Department of Electrical Engineering
Indian Institute of Technology, Kharagpur

Lecture – 44
Transfer Learning a GoogLeNet

(Refer Slide Time: 00:19)



```
In [1]: 1 %matplotlib inline
2 import torch
3 import torch.nn as nn
4 import torch.nn.functional as F
5 import numpy as np
6 from torchvision import transforms, datasets, models
7 import torch.optim as optim
8 from torch.autograd import Variable
9 import matplotlib.pyplot as plt
10 import torchvision
11
12 import copy
```

Welcome. So, today we will be doing our exercises with transfer learning. So, in the last lecture we had discussed about one of these aspects which is called as duman adaptation. And duman adaptation as it goes down is to an experiment I had through one of the results from one of our earlier publication papers and through a lot of other aspects I will shown you that if you have a classifier of some sort; So, not necessarily only a deep neural network, but then you have a classifier which has been trained on one particular data set to do a particular task. And then can you use all of this in order to solve another related kind of a problem.

So, one of those examples was what we did with retinal images in which it was (Refer Time: 00:49) on one data set called as drive which was more of from healthy people and was from a different camera a different hospital and everything. And then when we wanted to use the same model which was trained to do vessel segmentation on that particular data set on a different data set, and then you saw this problem that initially it was not giving out good results if directly deployed.

And then what we had an idea was can we use some examples at least a minimum number of examples over there in order to make this come down to a convergent point and will that be better than trying to train a network from scratch, which is using only the data from that other data set on diabetic retinopathy which is called a stair ok.

Now, standing on top of that on those experiments what I said is that for deep neural networks. We had done our experiments still now using some of our pre trained models, but then we never use the weights over there. We were just importing the model architecture and then we trying to do it. So, we had done it on LeNet we had done it on vgg net on GoogLeNet on ResNet on densnet and over there what we had done is just take down the architecture, and trained it now for our own data set, and for us most of these experiments we were doing it on see for dataset which is just a ten class classification problem and the images are pretty small at 32 cross 32 pixels and colored images.

So, the only common denominator which was binding all of them was that these were natural images and GoogLeNet kind of networks which were also done where for natural images, but then all of these networks were for a thousand class classification problem on image net. And here was just a ten class classification problem, and we took this ten class example over there from the aspect that, it is easier to train when you have lesser number of classes, you will be requiring lesser number of data. So, it was just a good toy example to get started with around.

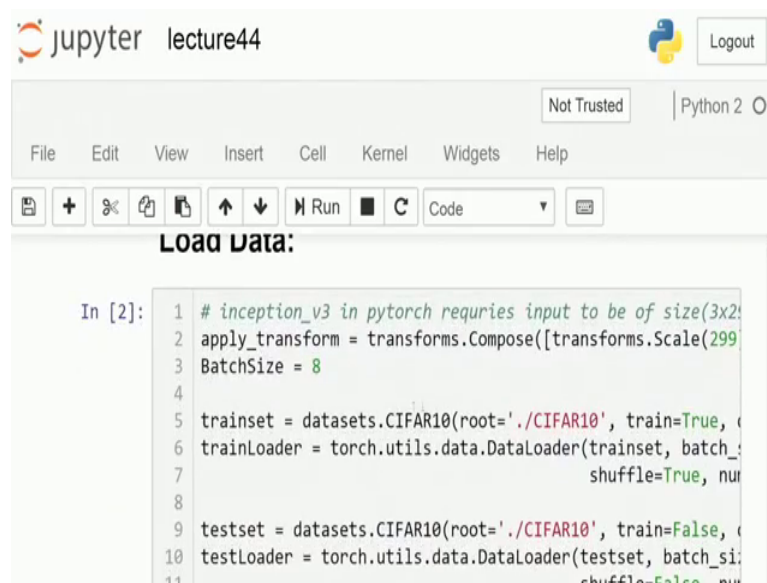
Now, today what I am going to discuss is if we take a network which was already trained to solve the image net problem to a great extent. So, say these GoogLeNets and vgg net then dens net, ResNet which made up its real soaring appearance in cvpr on virtue of winning the image net challenge. So, can we use all these pre trained models and would that give us an leverage.

Now, definitely looking down at the aspect of what we had learnt from the earlier class if we have a trained model and then we just try to refine it out it. It would be much easier to do ; however, then there can be multiple ways of refining. So, one of the ways of refining maybe where I update all the weights over that for the whole network, but that is a costly process.

Now, the other way you can refine it that you just update that final terminal layer, anyways you will have to change the architecture for the terminal layer. Since you are now not doing a thousand class classification problem, but just a ten class classification problem. So, let us let us look into what happens over there, and today's experiment will have something where I do not use a pre trained model. I just train everything from the start versus if we use a pre trained model and then either modified partly or modify end-to-end and then what comes out ok.

So, the over here as it goes through if we look into the codes over here, then you can see down that, the first part is pretty simple and standard that is the header which we have been using as of now for all of our experiments. We do not make any change over there. Next comes down our data. So, for the data, we are using CIFAR10, and now since this particular experiment deals with GoogLeNet.

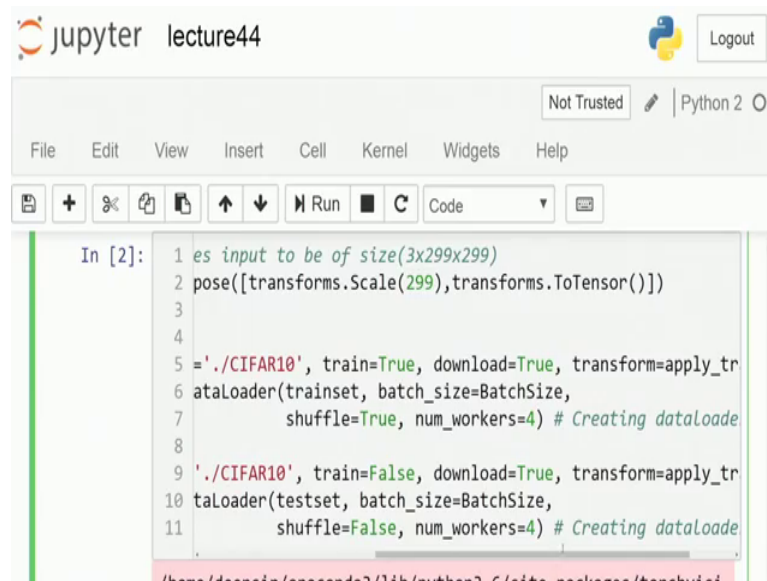
(Refer Slide Time: 04:05)



The image shows a Jupyter Notebook window titled "lecture44". The interface includes a top bar with the Jupyter logo, the title "lecture44", a Python logo, and a "Logout" button. Below the top bar is a menu bar with "File", "Edit", "View", "Insert", "Cell", "Kernel", "Widgets", and "Help". A toolbar contains icons for file operations, a "Run" button, and a "Code" dropdown menu. The main area is a code cell labeled "In [2]:" with the following code:

```
Load Data:
In [2]: 1 # inception_v3 in pytorch requiries input to be of size(3x2
2 apply_transform = transforms.Compose([transforms.Scale(299
3 batchSize = 8
4
5 trainset = datasets.CIFAR10(root='./CIFAR10', train=True,
6 trainLoader = torch.utils.data.DataLoader(trainset, batch_
7 shuffle=True, num
8
9 testset = datasets.CIFAR10(root='./CIFAR10', train=False,
10 testLoader = torch.utils.data.DataLoader(testset, batch_si
11 shuffle=False, num
```

(Refer Slide Time: 04:13)

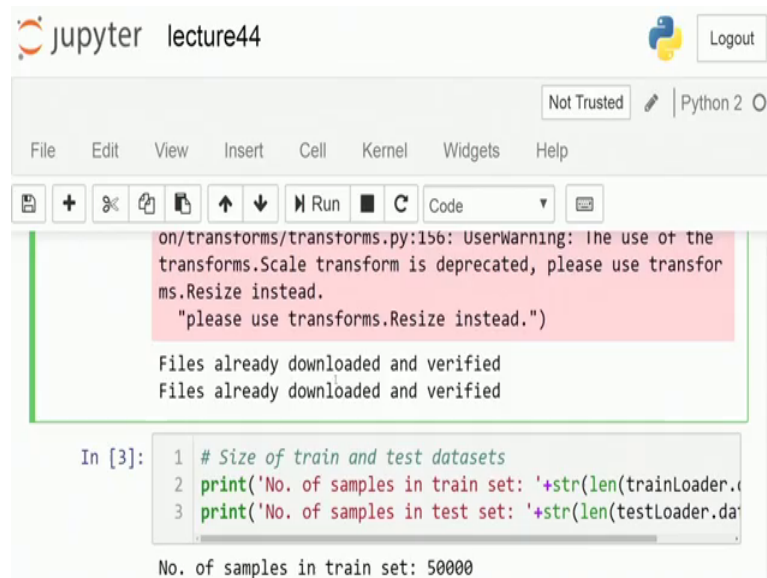


```
jupyter lecture44
Not Trusted Python 2
File Edit View Insert Cell Kernel Widgets Help
In [2]:
1 es input to be of size(3x299x299)
2 pose([transforms.Scale(299),transforms.ToTensor()])
3
4
5 = './CIFAR10', train=True, download=True, transform=apply_tr
6 ataLoader(trainset, batch_size=BatchSize,
7           shuffle=True, num_workers=4) # Creating dataलो
8
9 './CIFAR10', train=False, download=True, transform=apply_tr
10 taLoader(testset, batch_size=BatchSize,
11          shuffle=False, num_workers=4) # Creating dataलो
```

And what we had seen is that GoogLeNet by its virtue core virtue over there ; is something which was requiring 299 cross 299 pixel image is not two four cross two two four sized images over there. And for that reason we had to scale and transform these images over there.

And once you have the scale version of images at 3 cross 2 2 3 cross 299 cross 299 available then you can work it out. And we also choose down a smaller batch size. In fact, here we have even a lesser batch size of 8 for the fact that we would be training actually three networks; and that is going to consume a lot of memory on our side as well.

(Refer Slide Time: 04:58)



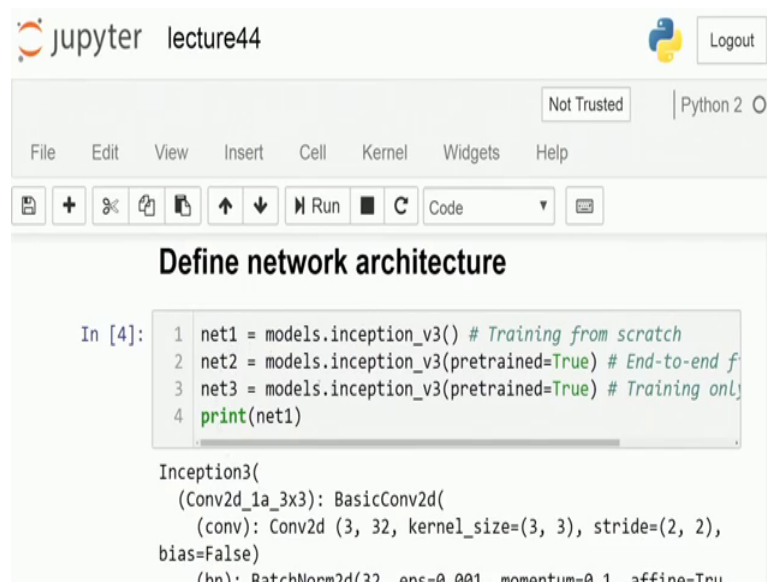
The screenshot shows a Jupyter Notebook window titled "lecture44". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for file operations and execution. The main area displays a warning message in a pink box: "UserWarning: The use of the transforms.Scale transform is deprecated, please use transforms.Resize instead." Below the warning, the text "Files already downloaded and verified" appears twice. The code cell shows the following Python code:

```
In [3]: 1 # Size of train and test datasets
2 print('No. of samples in train set: '+str(len(trainLoader.d
3 print('No. of samples in test set: '+str(len(testLoader.da
```

The output of the code cell is: "No. of samples in train set: 50000".

So, let us get down to this one, so there is the train loader and the test loader, the train set and the test set which are completely created out. And we are just going to you make use of that. Now the, so since we have the data and file already there it is it is confirmed and it works it out.

(Refer Slide Time: 05:13)



The screenshot shows a Jupyter Notebook window titled "lecture44". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for file operations and execution. The main area displays the title "Define network architecture" and the following Python code:

```
In [4]: 1 net1 = models.inception_v3() # Training from scratch
2 net2 = models.inception_v3(pretrained=True) # End-to-end f
3 net3 = models.inception_v3(pretrained=True) # Training only
4 print(net1)
```

The output of the code cell is a detailed representation of the Inception3 model structure, including layers like Conv2d_1a_3x3, Conv2d, and BatchNorm2d.

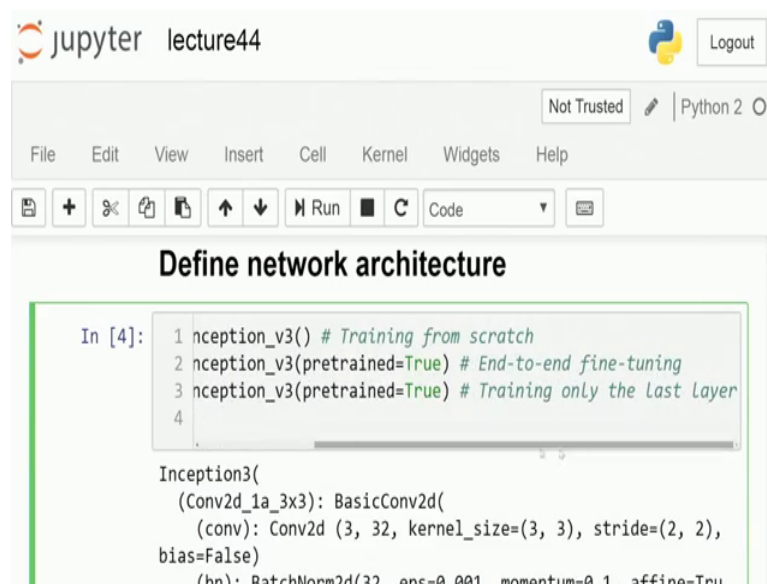
So, the CIFAR data set which we have is of 50000 examples in the train set; and 10,000 examples on the test set. And then here we start with defining our network. Now, if you look into the first network um, this is the one which is similar to the definition which we

had created in the earlier case example, which was just pull it down from our taught fusion library of models. And this is inception v3 which we were using.

Now, I create the next network which is the same as over here, except for the fact that I used this pre trained equal to true argument over there. And what this does is it does not just pull in the architecture from the web archive source, but it also pulls in the weights over there. So, although pre trained weights we will now get important and I am going to make use of this pre trained model. So, this is something which is already trained for solving the imaginary problem, and now we are going to look down whether it is able to solve this CIFAR10 classification problem as well.

So, then we define another one which is called as net 3 and next we also has the same kind of a mechanism over there. The only change comes down is that in net 2. So, this is what we have commented over here.

(Refer Slide Time: 06:12)



```
In [4]: 1 inception_v3() # Training from scratch
        2 inception_v3(pretrained=True) # End-to-end fine-tuning
        3 inception_v3(pretrained=True) # Training only the last layer
        4

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

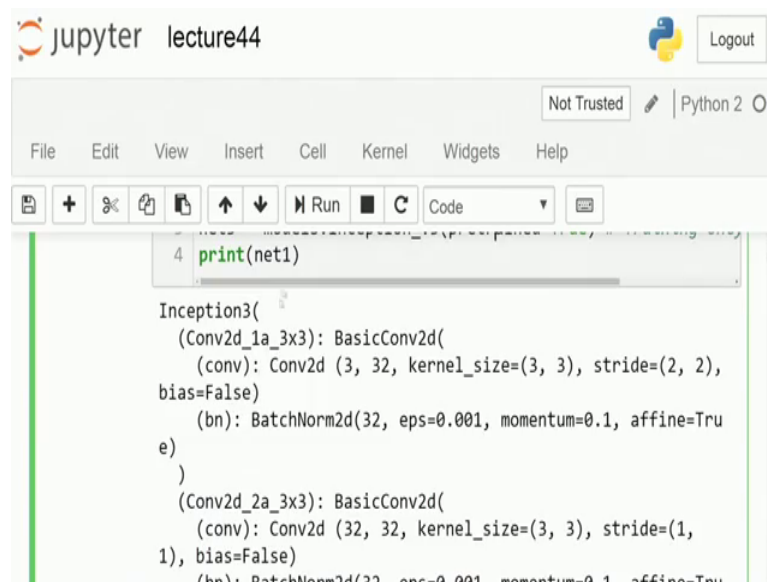
So, net 2 is the network which we are going to do end to end fine tuning which is you replace the last layer of 10,000 nodes of 1,000 nodes, and then replace it by just 10 nodes over there. In an when you are training you are going to do the whole error back propagation over the whole network.

In the other one, what we are going to do is, we are just going to update the last layer weights we are not going to update the complete network over there. So, this is the

change which we will look down and we will look into, so one aspect is that when you are just updating the last layer weights over there. So, your total compute complexity is really coming down, because you do not have to update the whole network at a go. Ah

On the other side, we need really need to look at that at this reduced compute complexity which we are offering, so that is a gain for us definitely in terms of the compute potential for the network. The last is in terms of decrease in accuracy; so is accuracy significantly decreasing because of this reason or can it really, keep on still working out. So, these this is the actual motivation of the whole work over there.

(Refer Slide Time: 07:15)

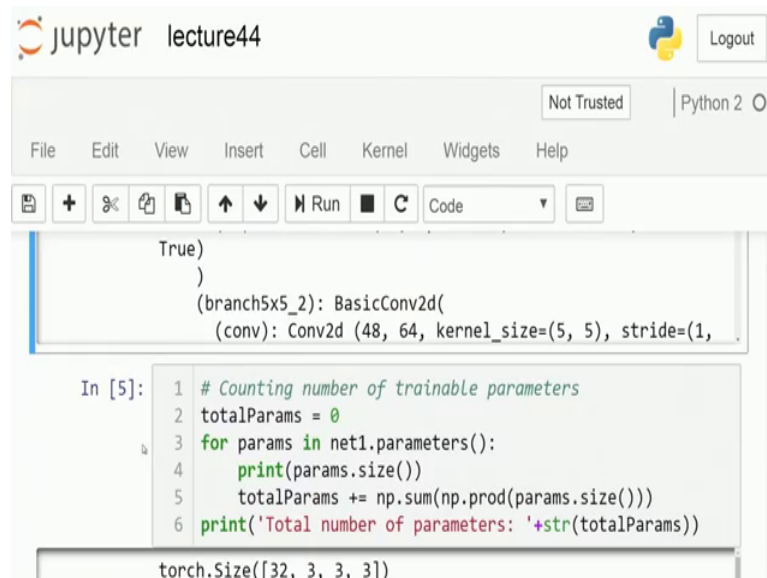


```
4 print(net1)

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

So, we just print down net 1 architecture, because as such it does not print weights if we are just printing down the other network over there. So, net 2 and net 3 we will have the same architecture which looks out.

(Refer Slide Time: 07:25)



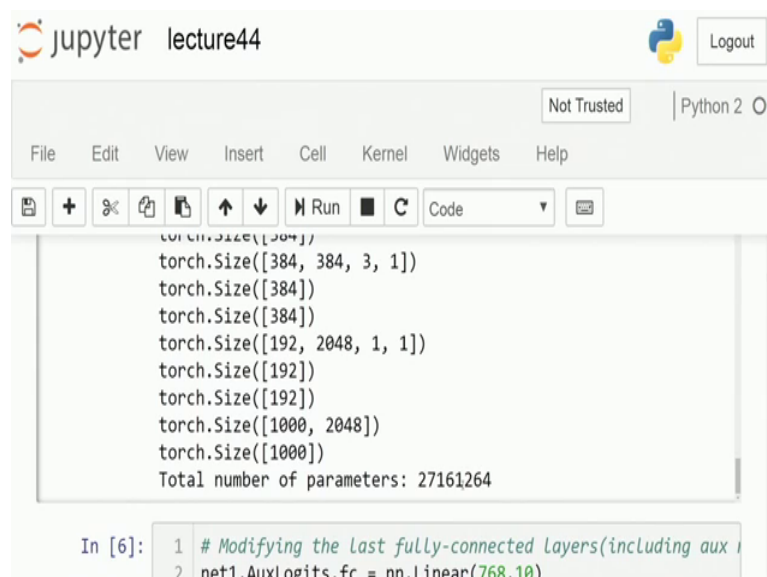
```
True)
)
(branch5x5_2): BasicConv2d(
  (conv): Conv2d (48, 64, kernel_size=(5, 5), stride=(1,

In [5]: 1 # Counting number of trainable parameters
2 totalParams = 0
3 for params in net1.parameters():
4     print(params.size())
5     totalParams += np.sum(np.prod(params.size()))
6     print('Total number of parameters: '+str(totalParams))

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

So, next the straightforward thing which we had for inception and we just have that, now what we do is. So, this one was looking into the total number of parameters over there, and then to count it out, so this was another part which we had done earlier this week which was look into the theory; Now, actually finding out the total number of computational parameters.

(Refer Slide Time: 07:48)

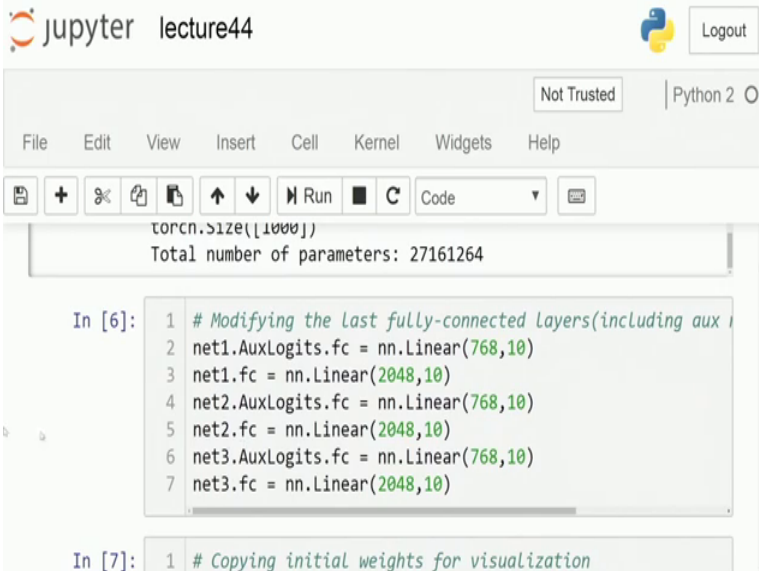


```
torch.Size([384])
torch.Size([384, 384, 3, 1])
torch.Size([384])
torch.Size([384])
torch.Size([192, 2048, 1, 1])
torch.Size([192])
torch.Size([192])
torch.Size([1000, 2048])
torch.Size([1000])
Total number of parameters: 27161264

In [6]: 1 # Modifying the Last fully-connected layers(including aux
2 net1.AuxLogits.fc = nn.Linear(768,10)
```


Now, if you go down through this part, you would be able to find out the total compute parameters as we had done; in the earlier experiment on GoogLeNet and that was something around 27 million parameters over there. Now once that gets over.

(Refer Slide Time: 07:53)



```
torcn.size([1000])
Total number of parameters: 27161264

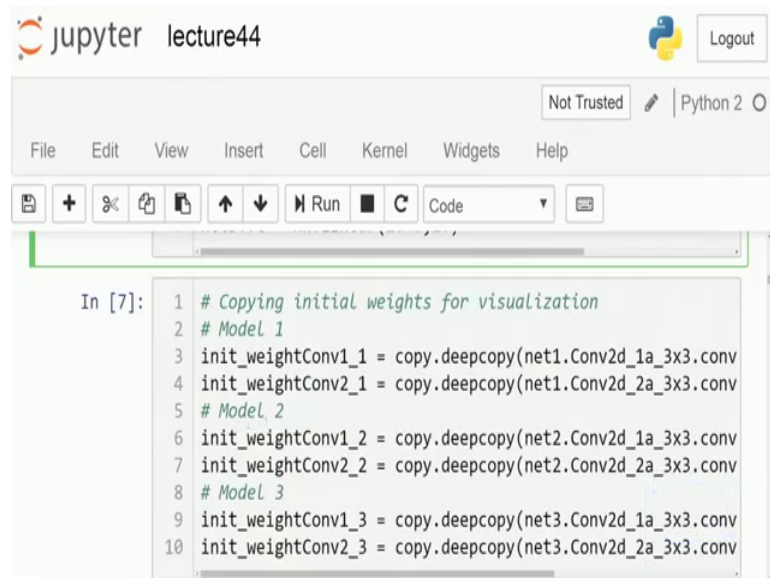
In [6]: 1 # Modifying the Last fully-connected layers(including aux
2 net1.AuxLogits.fc = nn.Linear(768,10)
3 net1.fc = nn.Linear(2048,10)
4 net2.AuxLogits.fc = nn.Linear(768,10)
5 net2.fc = nn.Linear(2048,10)
6 net3.AuxLogits.fc = nn.Linear(768,10)
7 net3.fc = nn.Linear(2048,10)

In [7]: 1 # Copying initial weights for visualization
```

The next part is to do this architecture modification. So, as you remember, that in all of the cases what we were doing is that the last layer over there was getting replaced from 2048 to 1,000 neuron connection instead of that; we are just going to replace it with 2048 to 10 neuron connection. As well as the same kind of a thing happens for the auxiliary layer as well. Because in each of these, things you again have an auxiliary layer. So, the aux auxiliary layer also gets replaced from 768 neuron to 1,000 neuron connection instead of that, we are going to put down 768 to 10 neuron connections over there. So, these are the two modifications which we do for all the networks.

And again recalling back from the GoogLeNet lecture over there; So, the particular model which we are using in torch in pi torch over here. Is the one it just has only one auxiliary classification arm it does not have two auxiliary classifications and for that reason, we just take down to changing only one of these arms over there; because the there is not the other one in anyway.

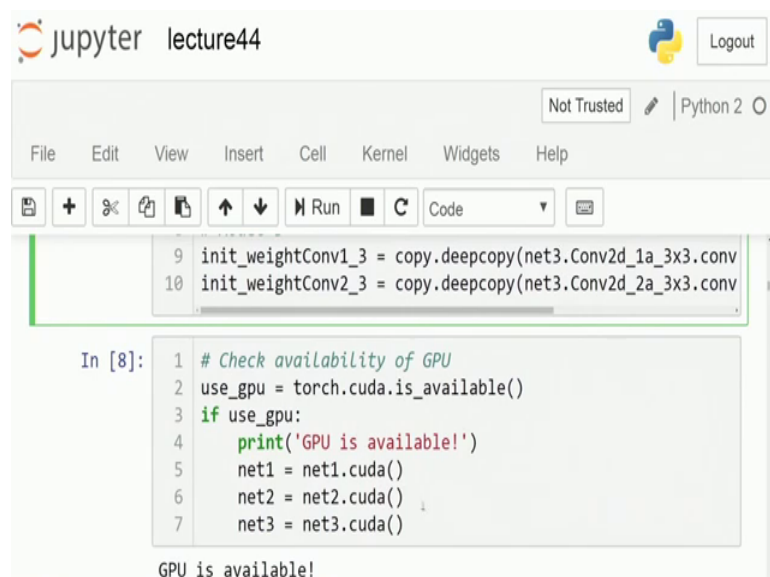
(Refer Slide Time: 08:55)



```
1 # Copying initial weights for visualization
2 # Model 1
3 init_weightConv1_1 = copy.deepcopy(net1.Conv2d_1a_3x3.conv
4 init_weightConv2_1 = copy.deepcopy(net1.Conv2d_2a_3x3.conv
5 # Model 2
6 init_weightConv1_2 = copy.deepcopy(net2.Conv2d_1a_3x3.conv
7 init_weightConv2_2 = copy.deepcopy(net2.Conv2d_2a_3x3.conv
8 # Model 3
9 init_weightConv1_3 = copy.deepcopy(net3.Conv2d_1a_3x3.conv
10 init_weightConv2_3 = copy.deepcopy(net3.Conv2d_2a_3x3.conv
```

So, now what we do is we just copy down our weights, the initial weights over there. Now keep in mind one thing that for your net 1, this is which is randomly pre trained this is this randomly initialized this is not at all pre trained; net 2 and net 3 are the pre trained model. So, net 2 and net 3 is initial state weights are the same in both the cases. They do not mix up in any of them and net 1 weight is very different from net 2 and net 3.

(Refer Slide Time: 08:20)



```
9 init_weightConv1_3 = copy.deepcopy(net3.Conv2d_1a_3x3.conv
10 init_weightConv2_3 = copy.deepcopy(net3.Conv2d_2a_3x3.conv

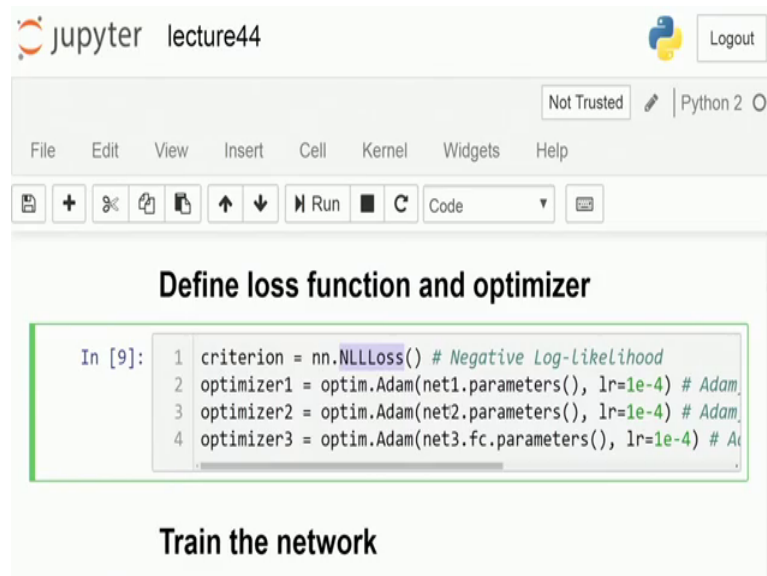
In [8]: 1 # Check availability of GPU
2 use_gpu = torch.cuda.is_available()
3 if use_gpu:
4     print('GPU is available!')
5     net1 = net1.cuda()
6     net2 = net2.cuda()
7     net3 = net3.cuda()

GPU is available!
```

Now, we get into the pattern GPU availability, now it is if it is available then there is a typecast operator. And we are done in the previous lectures on what happens during

typecasting, and how there is a dma transferred and the whole model as well as the weights and the data itself gets transferred over there. So, this part is also quite clear with you guys on what we were doing with the typecasting and what was the compute equivalent on the computer hardware side of it, what would was it initiating on a hardware aspect as well.

(Refer Slide Time: 09:50)



The screenshot shows a Jupyter Notebook window titled "lecture44". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for file operations and execution. The main content area is titled "Define loss function and optimizer" and contains a code cell with the following Python code:

```
In [9]: 1 criterion = nn.NLLLoss() # Negative Log-likelihood
2 optimizer1 = optim.Adam(net1.parameters(), lr=1e-4) # Adam
3 optimizer2 = optim.Adam(net2.parameters(), lr=1e-4) # Adam
4 optimizer3 = optim.Adam(net3.fc.parameters(), lr=1e-4) # Adam
```

Below the code cell, the text "Train the network" is visible.

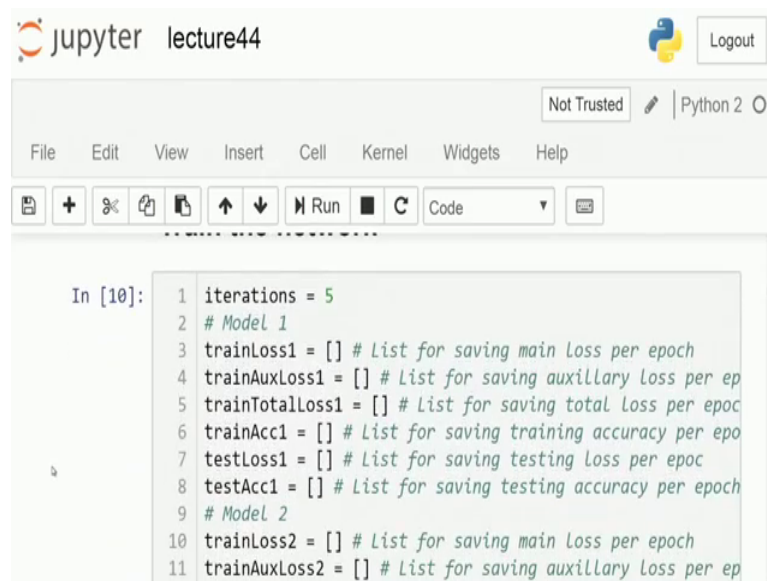
Now, from there coming down to the loss criteria and they are pretty simple. So, since we are solving the classification problem, so it is the negative log likelihood loss which we are taking down and then the optimizer is Adam for all of them and the learning rate over there is also kept constant ok.

Now, having said that there is one critical aspect which we need to look into it; So, for network 1 and network 2 is where we are going to do end to end. So, network 1 is which was randomly initialized and then you had trained it from the final node to the initial layers over there. Network 2 is which is already pre initialized with the weights from solving the image net problem and we are still going to update the total weights of the all the layers from the last layer till the first layer.

Network 3 is where we are going to freeze in the weights of the layers from the first till the terminal one. So, only the last layer which is for connecting down 2048 neurons to 10 neurons that is the one which is getting updated everything else does not get updated.

Now, as a result when I am not updating anything else over there, so I do not even need the auxiliary losses, because there is no back propagation happening off for all of them. So, they are they are the ones which are going to remain preserved. Now, that is the change which we have in the code. So, in the code, if you look down into the parameters what we do for network three is we just use this fully connected parameters over there of the fc layer. So, this was this fc layer which we had changed over there for each of them.

(Refer Slide Time: 11:34)



The screenshot shows a Jupyter Notebook window titled "lecture44". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help), a toolbar with icons for file operations and execution, and a code editor. The code in the editor is as follows:

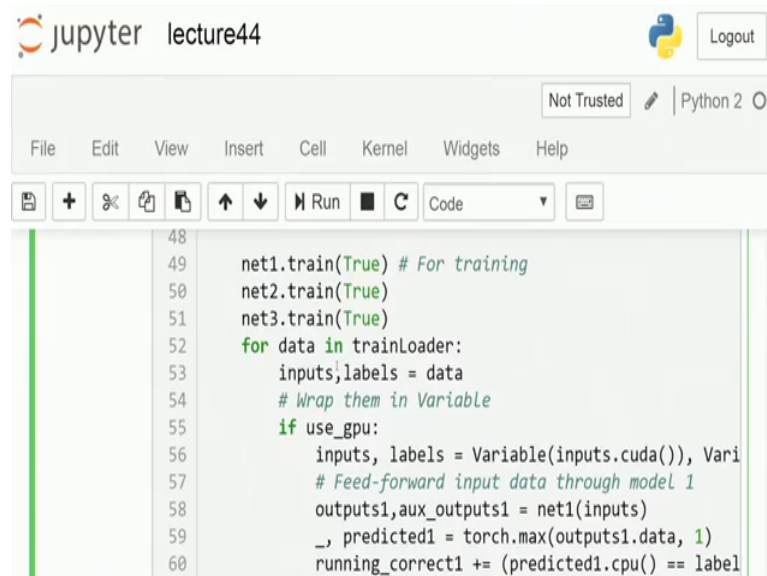
```
In [10]: 1 iterations = 5
2 # Model 1
3 trainLoss1 = [] # List for saving main Loss per epoch
4 trainAuxLoss1 = [] # List for saving auxillary loss per ep
5 trainTotalLoss1 = [] # List for saving total loss per epoc
6 trainAcc1 = [] # List for saving training accuracy per epo
7 testLoss1 = [] # List for saving testing loss per epoc
8 testAcc1 = [] # List for saving testing accuracy per epoch
9 # Model 2
10 trainLoss2 = [] # List for saving main Loss per epoch
11 trainAuxLoss2 = [] # List for saving auxillary loss per ep
```

Now, we are going to just use the parameters over here in order to optimize. And what that would mean is that this is these are the only set of parameters on which the update rules are going to run on the other parameters the update rules are not going to run, and you will end up updating only the last layer for this network ok. So, now, once that goes down, we train this model just for 5 iterations because some part of it is already trained. So, it is it is easier to show it to you. The other downside is that it takes really long say GoogLeNet per epoch was taking down like what 8 to 9 minutes of it to train down.

And now since we have three networks to train down in one single shot. So, whether it takes about 24 minutes or roughly half an hour of time to train it. So, going it down for more epochs is going to take more amount of time; so for us we have just done it for five epochs. In fact, within five epochs, it does come down to the saturation point what we wanted to show on the performance characteristics ok. Now, there goes down the losses

and an accuracy tensors which we needed for each of the three models and then we start for each of the model over there.

(Refer Slide Time: 12:27)

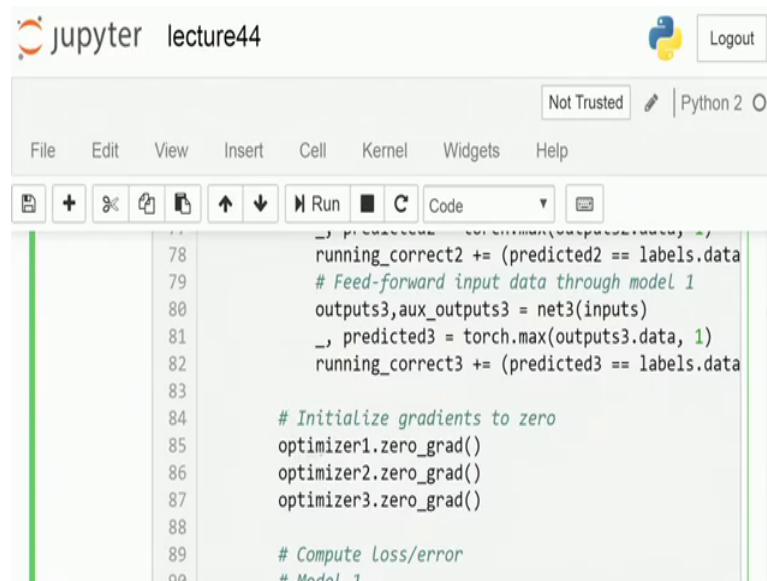


```
48
49 net1.train(True) # For training
50 net2.train(True)
51 net3.train(True)
52 for data in trainLoader:
53     inputs,labels = data
54     # Wrap them in Variable
55     if use_gpu:
56         inputs, labels = Variable(inputs.cuda()), Vari
57         # Feed-forward input data through model 1
58         outputs1,aux_outputs1 = net1(inputs)
59         _, predicted1 = torch.max(outputs1.data, 1)
60         running_correct1 += (predicted1.cpu() == label
```

Now, within an epoch, I am going to load down one batch of data. Now, for that batch of data I just see if it is if there is a gpu available, then we just do a cuda typecasting which is to get the dma transfer of the batch of data, onto my gpu ram. So, that it can work over there.

And then my first part is, do a feed forward over it, collect down the outputs from the auxiliary arm as well as from the actual terminal output over there. And then do a final classification accuracy compute over there. Now, once this accuracy is computed over there.

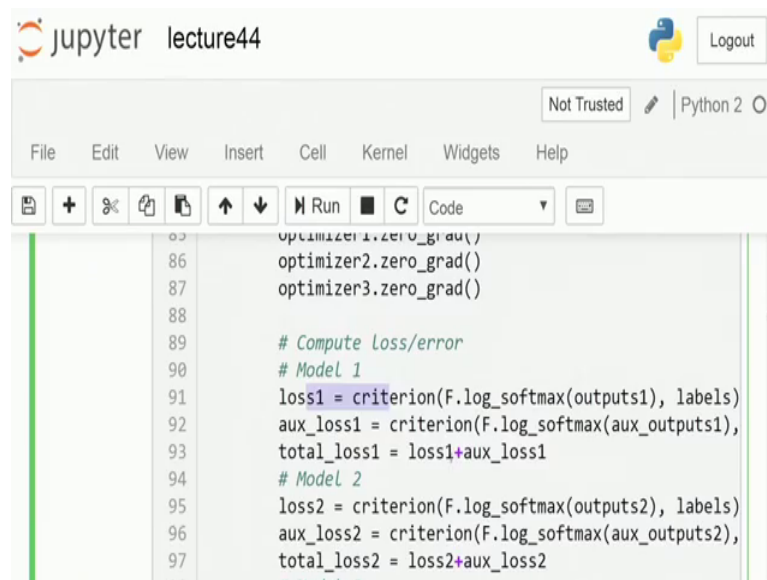
(Refer Slide Time: 13:01)



```
jupyter lecture44
Not Trusted Python 2
File Edit View Insert Cell Kernel Widgets Help
Run Code
78 running_correct2 += (predicted2 == labels.data)
79 # Feed-forward input data through model 1
80 outputs3,aux_outputs3 = net3(inputs)
81 _, predicted3 = torch.max(outputs3.data, 1)
82 running_correct3 += (predicted3 == labels.data)
83
84 # Initialize gradients to zero
85 optimizer1.zero_grad()
86 optimizer2.zero_grad()
87 optimizer3.zero_grad()
88
89 # Compute Loss/error
90 # Model 1
```

Now, what we need to do is you will come down to your optimizers and then actually create down, zero down all the gradients intermediate over that.

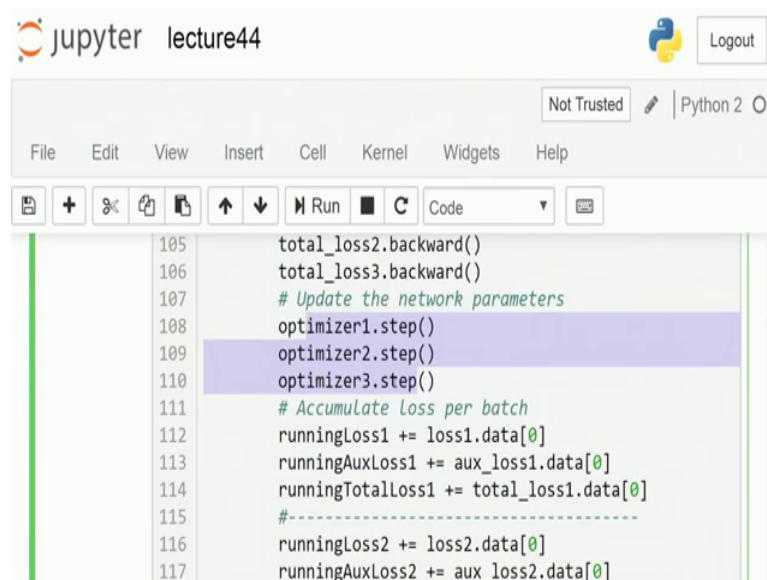
(Refer Slide Time: 13:10)



```
jupyter lecture44
Not Trusted Python 2
File Edit View Insert Cell Kernel Widgets Help
Run Code
85 optimizer1.zero_grad()
86 optimizer2.zero_grad()
87 optimizer3.zero_grad()
88
89 # Compute Loss/error
90 # Model 1
91 loss1 = criterion(F.log_softmax(outputs1), labels)
92 aux_loss1 = criterion(F.log_softmax(aux_outputs1),
93 total_loss1 = loss1+aux_loss1
94 # Model 2
95 loss2 = criterion(F.log_softmax(outputs2), labels)
96 aux_loss2 = criterion(F.log_softmax(aux_outputs2),
97 total_loss2 = loss2+aux_loss2
98 # Model 3
```

Then do a criterion based loss function compute for the auxiliary node; as well as for the main terminal node over there. And then find out what is your total loss for the model. And this is what you keep on storing, as well as within your optimizer you are going to make use of all of this. And do an optimizer dot step which just once the update rule.

(Refer Slide Time: 13:27)



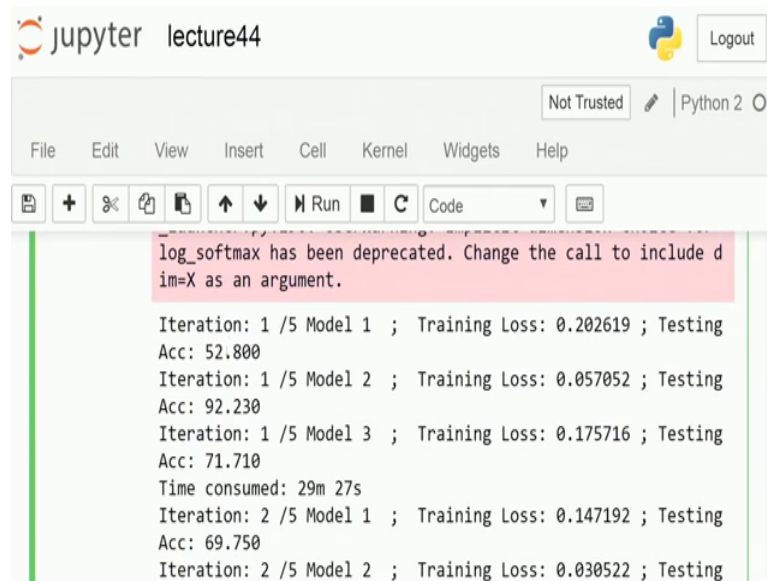
```
105     total_loss2.backward()
106     total_loss3.backward()
107     # Update the network parameters
108     optimizer1.step()
109     optimizer2.step()
110     optimizer3.step()
111     # Accumulate loss per batch
112     runningLoss1 += loss1.data[0]
113     runningAuxLoss1 += aux_loss1.data[0]
114     runningTotalLoss1 += total_loss1.data[0]
115     #-----
116     runningLoss2 += loss2.data[0]
117     runningAuxLoss2 += aux_loss2.data[0]
```

Now, since in your first two networks network and network one and network two, net 1 and net 2. You had collected down all the parameters to be used within atom. So, all the parameters are getting updated in net 3, we had just used the last fully connected layer. And, so the updates which happened down are only in the last fully connected layer over there and nothing else gets updated or modified in any way ok.

Now, once that is done. We have our losses accuracies and everything taken down and averaged out over it. And then within each epoch I am also calculating out my validation scores. So, this is on the test data set which is left over there. So, at the end of update of one of these epochs, how much is the change which happens, during the test performance for the module.

So, this is what I compute out and then it pretty much goes on in the same way. The only difference over here is that instead of using one network, we have three networks which we are using. And then I decide to plot all of them. So, we will let us go down and look into the models and how they are training and then come down to it ok.

(Refer Slide Time: 14:25)

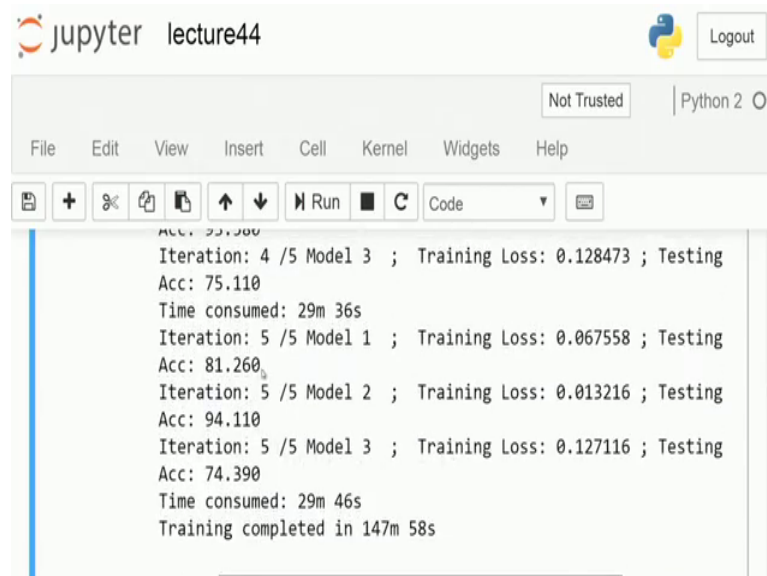


```
jupyter lecture44 Python 2  
File Edit View Insert Cell Kernel Widgets Help  
log_softmax has been deprecated. Change the call to include dim=X as an argument.  
Iteration: 1 /5 Model 1 ; Training Loss: 0.202619 ; Testing Acc: 52.800  
Iteration: 1 /5 Model 2 ; Training Loss: 0.057052 ; Testing Acc: 92.230  
Iteration: 1 /5 Model 3 ; Training Loss: 0.175716 ; Testing Acc: 71.710  
Time consumed: 29m 27s  
Iteration: 2 /5 Model 1 ; Training Loss: 0.147192 ; Testing Acc: 69.750  
Iteration: 2 /5 Model 2 ; Training Loss: 0.030522 ; Testing
```

So, within each iteration there are three models which are training down over there. So, if you look down at the training loss; So, the first model which was a randomly initialized model it starts with a much higher loss whereas, the second model where you had done a you had just modified only the terminal node over there. That is the one which has a lower loss; while the other one is where you had modified the terminal layer, but not updated all the weights you had just modified only the terminal layer weights.

So, in net 2 is where you do an end-to-end update. In net 3 is where you do only the terminal layer update. So, it trains out definitely much faster, but then the loss is not so low. So, the loss is relatively high over there.

(Refer Slide Time: 15:09)



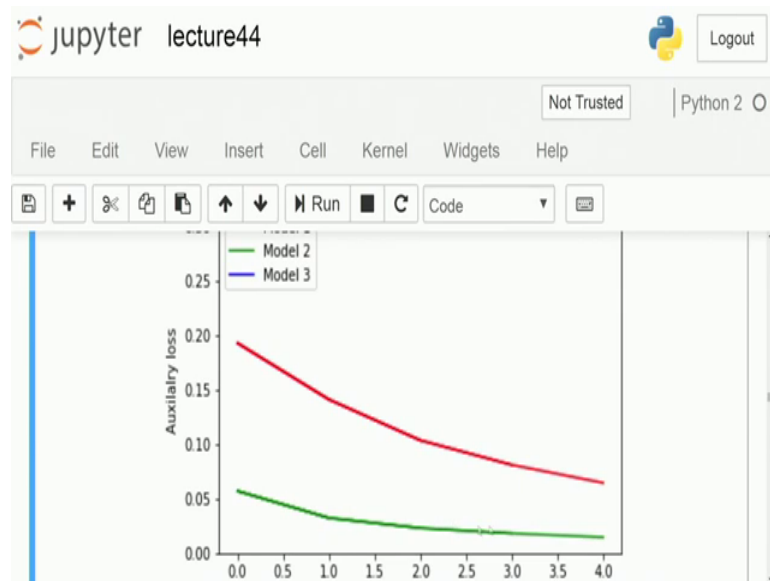
```
Acc: 75.110
Iteration: 4 / 5 Model 3 ; Training Loss: 0.128473 ; Testing
Acc: 75.110
Time consumed: 29m 36s
Iteration: 5 / 5 Model 1 ; Training Loss: 0.067558 ; Testing
Acc: 81.260
Iteration: 5 / 5 Model 2 ; Training Loss: 0.013216 ; Testing
Acc: 94.110
Iteration: 5 / 5 Model 3 ; Training Loss: 0.127116 ; Testing
Acc: 74.390
Time consumed: 29m 46s
Training completed in 147m 58s
```

Now, if we keep on going through the network, you would what you would find out is that at the end of five epochs over there. So, for model 1 the loss is somewhere around point 06; for model 2 on net 2 is, where you had updated the whole network end-to-end for CIFAR problem is around point 1. And for the last one, where you are just updating the final layers is, where the loss is a bit higher.

Now, if you look even at accuracies what you see is that the first model where you had started with a random initialization, a random guess over the weights and then updated all the weights together; that has a higher accuracy on the test data set; As compared to the other model where you were updating only the last terminal nodes.

So, this definitely means that, one point which I was emphasizing on the lectures on, domain adaptation and transfer learning was that you need to really update, all the weights and features, and everything over there. Not just the classifier layer over there. So, it may be that your features over there, are not quite specific to the actual problem which you are dealing with and they might also need to get an update. So, this is a clear example where you can see this discrimination coming down.

(Refer Slide Time: 16:17)

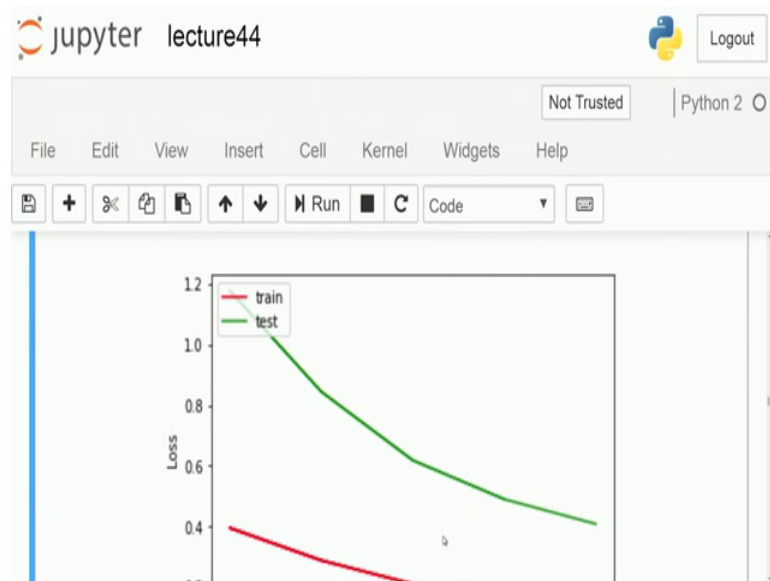


Where in the first case, when you are updating all the weights over there; So, you see an accuracy come down to at 81 percent; whereas, in the last case where you are updating only the last layer weights over there. So, the accuracy does not go up that high, it just stays at 74 percent ok. So, we look into this part over there.

So, this is the auxiliary loss. Now, look into one thing in model 3 is, where we were updating only the final layer. So, we all not connecting the auxiliary arm, parameters and updating that. Though we were collecting the losses, and there was this arm kept over that. So, this remains fixed this does not change in any way.

And for the other two models, you would see that the auxiliary loss also keeps on decreasing substantial. Now, model 1 is where it was a random guess. So, it starts with a much higher one; and then keeps on going down.

(Refer Slide Time: 16:59)



Now, in the next case is where you see for the first network. network one, where I was training it from scratch and doing it. So, this is how the my train and test, losses are decreasing and this is how my accuracies are going down.

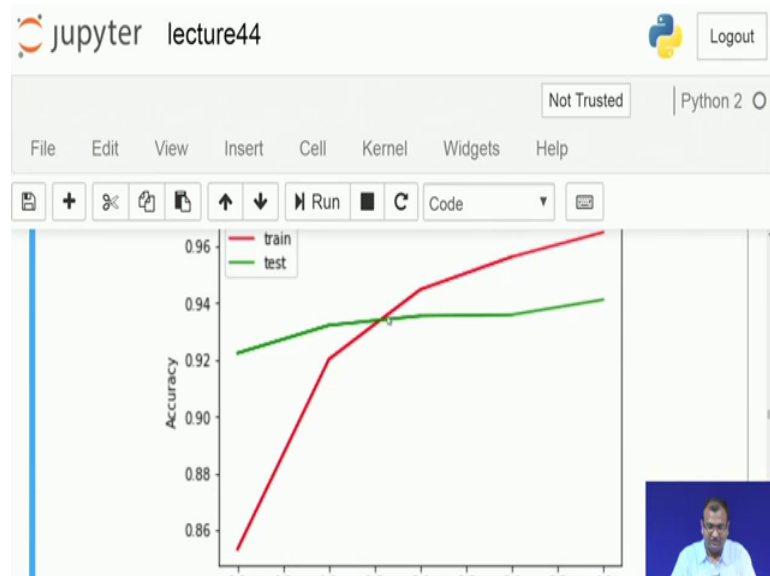
(Refer Slide Time: 17:06)



So, you see crossover point at about the fifth epoch itself. Is when my train accuracy is increasing, but my test accuracy is, now either saturating or trying to decrease out. So, this is the first crossover point which comes down over there. Now, this is for my second model where I was doing an end-to-end update based on the training over there. Now,

definitely it comes down much faster, towards the convergence as compared to the earlier model.

(Refer Slide Time: 17:33)



Where it was not initialized based on some lever guess, but it was a random initialized one. You see that, the crossover also takes place much earlier. So, this is already at the at this point. So, which is almost like this is the first epoch, second epoch, so this is my third epoch. So, in between the second and the third epoch is where the crossover is already taking place over there.

So, maybe just after 2 epochs you can just stop it, or a maximum you can go out to 3 epochs; and then you see that the whole network is updated. So, this is much faster. So, if you look into the earlier case, you do not get this much of an accuracy of say, 94 percent in the in single shot over there.

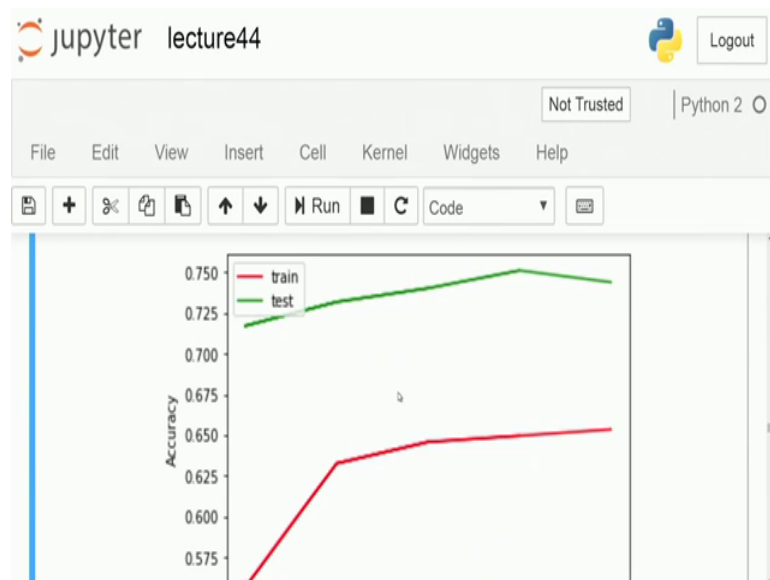
So, if you if we look into the first case which is this model. So, at the end of third epoch which is somewhere over here, I have an accuracy of 75 percent, but then if I start down with my weights which are taken down from the image net problem. And then at the end of third epoch, I am already at 94 percent which is really good and that is the advantage which you get down with the transfer learning of using a model which was trained on a similar or a rel or a on a related kind of a data; for the related kind of a problem and then just modify it. So, you have a much higher accuracy coming down with just, lesser number of epochs of training.

(Refer Slide Time: 18:45)



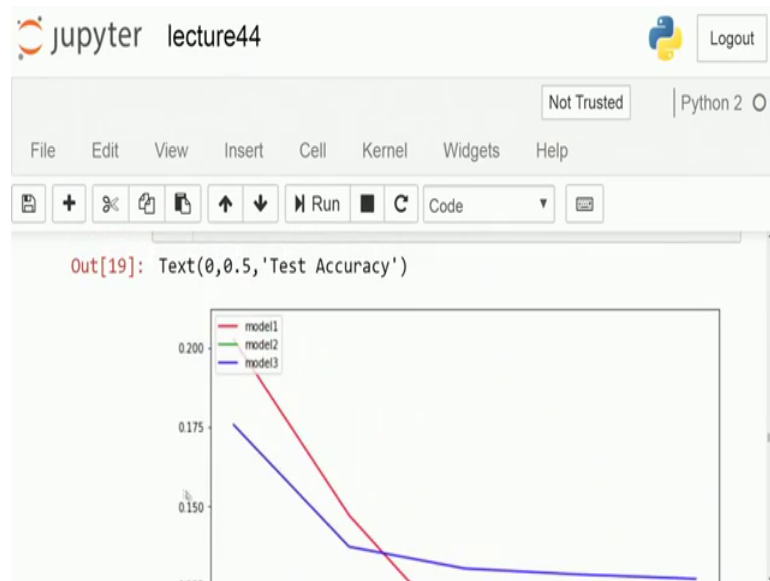
And then if you look into the last one, where I was looking into just modifying the final nodes over there.

(Refer Slide Time: 18:49)



Now, that will not always be helpful, because you still see that you are limited at the below 75 percent accuracy in either of these cases and the. So, the trained accuracy is also lowered, though the test accuracy or the generalizability is much better, but then one that your test accuracy is still limited below 75 percent.

(Refer Slide Time: 19:22)



So, this is another critical fact which you need to keep in mind that always updating, just the final terminal nodes might not always be the best possible solution to go around with.

(Refer Slide Time: 19:34)

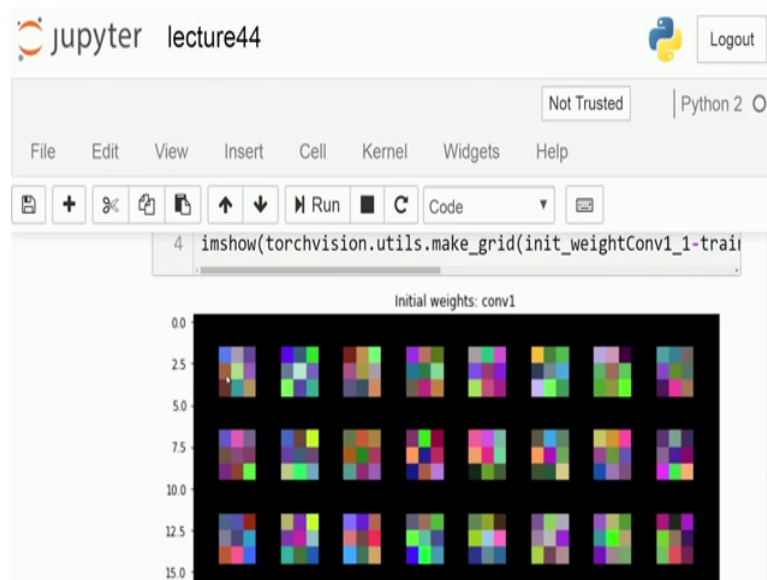


So, once we have that the next idea was to. So, this is where it says just the same kind of plot, but taken down into the same thing together. So, this is my train loss, and my test losses, I also have my test accuracy plotted down over there for all three of them.

So, if you look into it your model 2 which was a pre initialized model, and then just being updated over there. So, your test accuracy is really keep on increasing, and it is

definitely a few folds higher than your other models over there; model 1 and model 2, model 1 and model 3. Model one which was just randomly initialized and trained. Model 3 is where were only the final nodes, were getting updated.

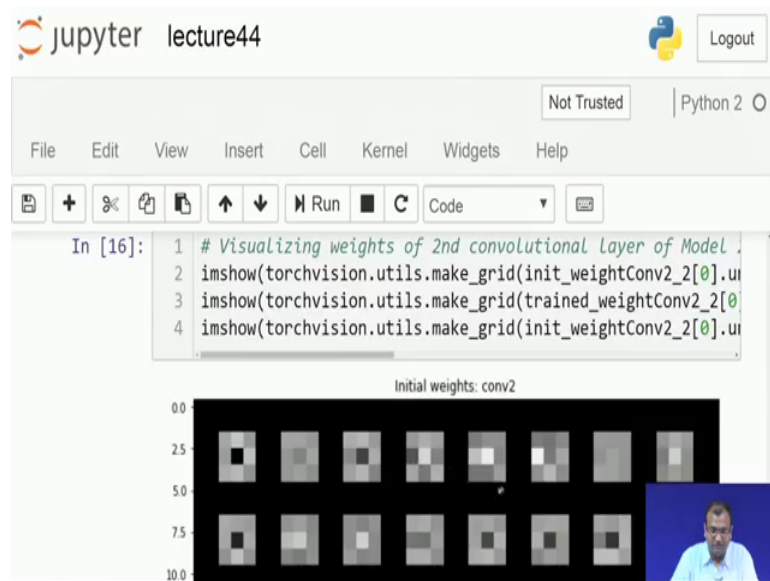
(Refer Slide Time: 20:08)



So, now, let us look into the weights over there. So, this is for my model 1 where I had my random initialization. So, this was the weights for the first convolutional layer. So, you have 3 cross 3 kernels over there and 1, 2, 3, 4, 5, 6, 7, 8 and 8 so that was sorry 4 and 1, 2, 3, 4 1, 2, 3, 4. So, there were 32. So, you have 32 channels over there and 3 cross three convolutional kernels over there, which you had for the first layer.

Now, what you see is these were the random initializations, which had been done. This was the update after one after at the end of five epochs over there, and these were the change of the weights which you see. Now, for your second layer over there, this is what you observe over there. Now, in the second layer definitely, because the input was 32 channels over there, so we just display for one of the kernels all the channels coming down over there.

(Refer Slide Time: 21:10)



Now, if you look into the second model where you had taken a pre trained model. So, this were the weights at the start of the training process, and these were the weights at the end of the training process over there, and these were the difference of the weights which were coming now. Now, if you look into the second convolutional layer over there, so do you see these were the initial final states as well as the difference coming down.

Now, here comes the interesting part which was the model 3, in which what I was doing is updating only the final nodes and I was not updating anything intermediate. So, these were my starting states. And since I was not updating anything on the previous layers over there, so this becomes my ending state; then my start state and end state is the same. And for that reason, you see that your weight difference matrix over there is completely 0. So, there is there is no update which is happening over there.

Similarly, is the case for the second convolution layer for the network three as well. So, this is what we have as a clever and clear exercise, for doing your transfer learning on any kind of a deep network. So, this was one example just using GoogLeNet in which you have, the aspect of arm auxiliary arm as well. And then how do you do this either a full end-to-end training, or you are just concerned with the last final edge of final classification terminal nodes over there.

So, now, one thing which is clear from these graphs is that, definitely it is always advantageous to take down a model which is pre trained. And for you it is going to

consume lesser amount of your CPU time yes it does take more amount of time to download the model, because the model is not just the architecture definition. It also has the weights over there. So, that is the additional bytes which will be downloading over there. On the other side of it your CPU time which is going to get consumed is lesser.

Now, the next factors will you be updating only the terminal layer. Now, this is case to case dependent, we cannot directly comment over here. But typical practice is that, only updating the terminal layer might not be a very good idea, because your features might not be optimum for your particular data set which you are looking. So, if you look into image data sets from image net, which are 224 cross 224 sized over there. The granularity and resolution is much higher. Whereas, for CIFAR it is smaller sized image, they were just thumbnails of 32 cross 32.

So, when once you scale it up, in fact there is a lot of blurring which comes down around over that. Now, the features associated with trying to identify an object which is blurred versus the features associated with trying to identify an object, where the image is very crisp and of a high resolution are going to be very different. And for that reason what you would find out, is that this particular model, does not work out good if you are just updating the final layer.

Whereas, if you go down for newer kind of problems, where it is a activity net is a new challenge which comes up. A lot of people have a good way of just trying to take a pre trained GoogLeNet were, this GoogLeNet was trained on for the image net problem. And then you update only the final layers, you do not update everything, because on activity net also your videos are almost of the same size of 224 cross 224. And then natural looking images, they are of high resolution; there is this mismatch of early level or low level feature descriptors is, not that.

So, that is what we have for the transfer learning. As of now we have, the next lecture where we are going to cover down, transfer learning with receiver networks as well; so what happens within residual connections, and what is an advantage which you gain and if there is some disadvantages as well. So, we will be doing the same thing and repeating it, once over again to check down what goes on over there as well. So, with that we come to an end to today's one.

And thank you.