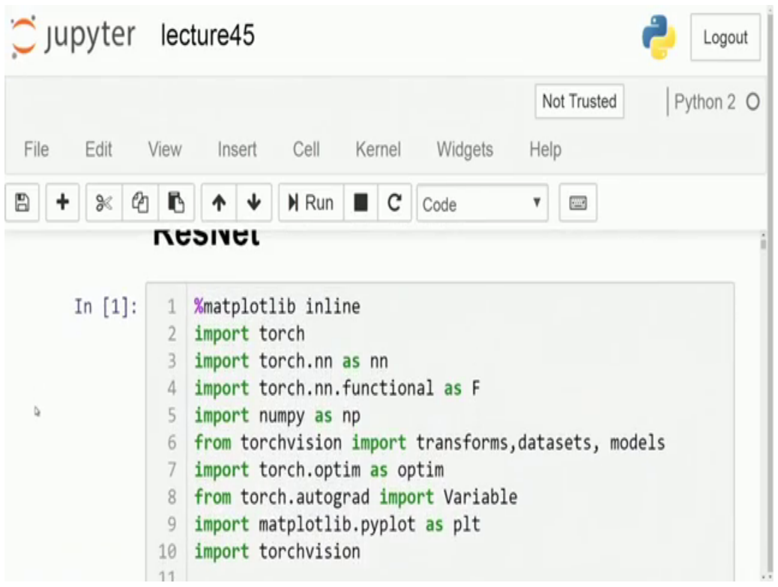
Deep Learning for Visual Computing

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

Lecture – 45
Transfer Learning a ResNet

(Refer Slide Time: 00:20)



```
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
```

Welcome. So, in the last lecture which we had covered down it was more often understanding transfer learning and for domain adaptation purposes and that is where we were doing it with one of these first versions of a very deep network which was called as GoogLENet. Now, in this particular lecture, we are going to do it on a residual network which is the successor for let us let us say in someone says the successor and the subsequent model which comes down on the image net challenge after that.

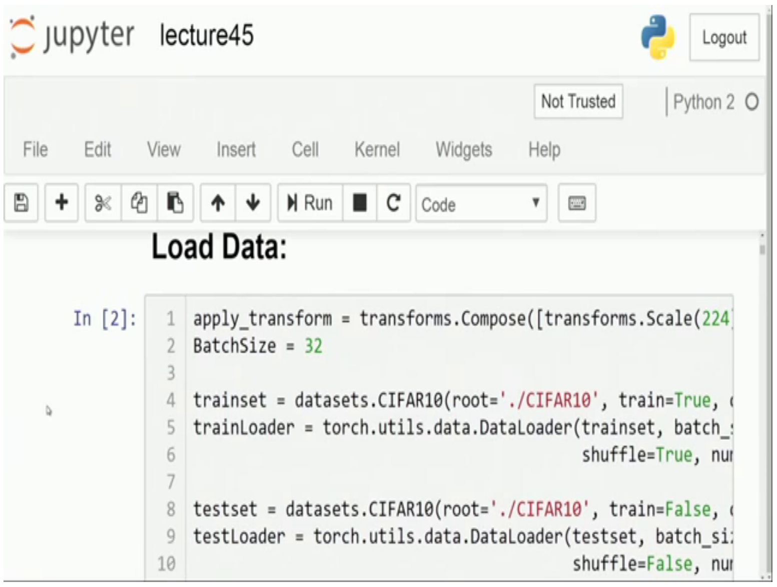
Now, you had look into different aspects over there, one was that GoogLENet is something which in order to have done real deeper networks over there which can work down in order to get trained. So, one major aspect within a GoogLENet was you needed to have an auxiliary arm for doing an auxiliary (Refer Time: 00:59).

So, whatever you are modifying, you also had to modify in the auxiliary arm and then whether you are going to update down only the final layers or everything end to end that was actually something which was influencing the total performance. And what we had seen is that over there in fact, like not being able to update everything and just trying to update only the terminal nodes was producing a lesser accuracy at the end of five epochs as compared to the one where I was trying to train it from scratch which is take a completely untrained network with randomly initialized widths and I am trying to train it out.

So, now obviously, there was almost a 20 percent of accuracy difference if you compare down taking your pre trained model versus our non pre trained model. Or even a pre trained model where you are doing whole end-to-end update versus taking a model where you are not doing an end-to-end update.

So, based on that let us look into what we have today. So, on the residual network what we are going to do is something like this. So, the initial part is just my header which is quite similar to what we have been taking down for all of our other earlier exercises has but not much of a difference which comes down.

(Refer Slide Time: 02:08)



Load Data:

```
In [2]:
apply_transform = transforms.Compose([transforms.Scale(224
BatchSize = 32

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

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

Next I get into my data. Now, for residual networks as we had seen in the earlier case, we are just taking down to two two four cross two two four sized images. Whereas, in GoogLENet you had take a; you have to take down 299 plus 299 sized images. And also in the earlier experiment where we were just trying to get you updated on what your residual network is you did not find out that the total number of parameters is low. And based on our computer complexity calculations you could figure out that the operational space is also lesser for a residual network as compared to a GoogLENet.

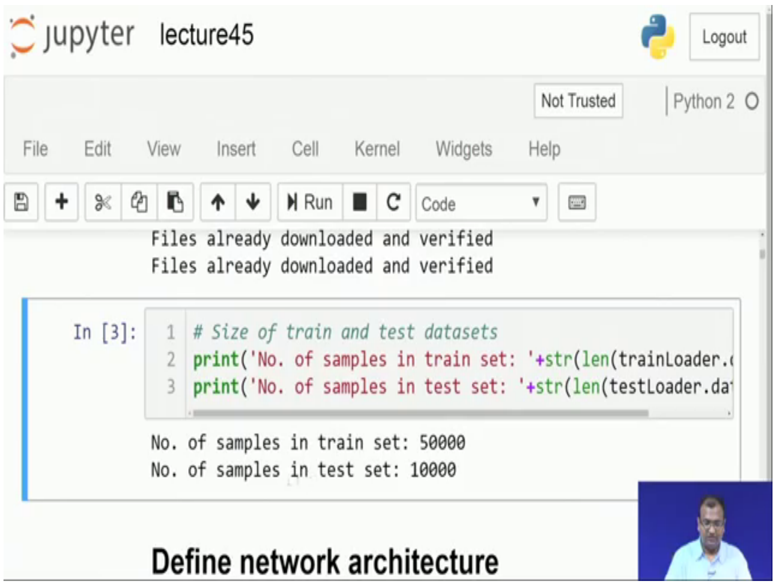
So, this is what I had left out as open-ended problems which you could solve out on your space. And based on this operational space complexity, you always get to decide what will be your batch size, because finally the idea is that your operational space four batch should be something which fits into the GPU memory or CPU memory whatever you are going to use over there. So, over here for residual networks, typically your back size was more than the batch size which we could take down for GoogLENet and we are sticking down to a batch size of 32 which fits perfectly for running down even three different configurations.

So, here also we have the same kind of a logic as we had used in a GoogLENet, where we had three different networks. So, the first network is which was just download it and it was without any of the trained weights over there. So, everything was randomly initialized bits.

The second network and the third network was what was taken down from a image net pre trained model and then based on this pre trained model we had just updated the weights over there. In the second network net two is where we did an end-to-end update which is over all the layers and weights over there. In net three is where we had done an update only for the last layer over there; we did not update anything in the previous layers in anywhere.

Now, we start from there and then get down onto say our understanding of the data set. So, what we do is we create our data set loaders over there, the train and test loader, and then they are pretty decently created.

(Refer Slide Time: 04:14)



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

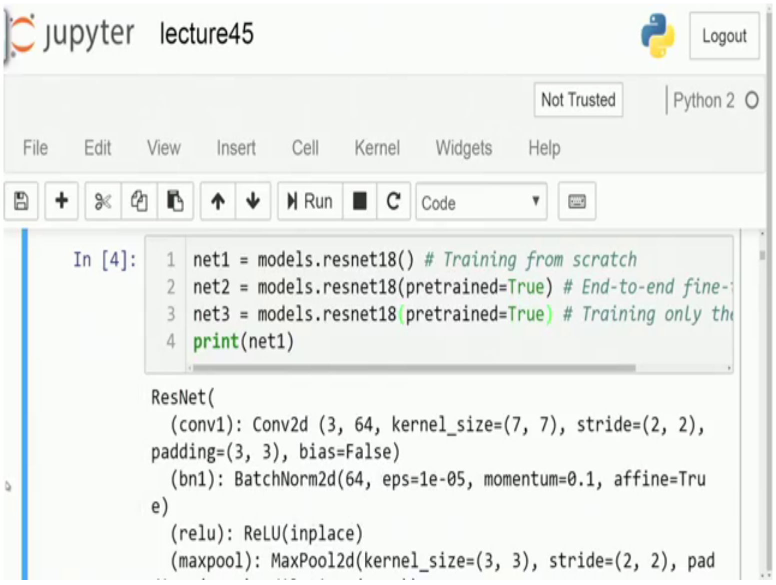
```
In [3]: # Size of train and test datasets
        print('No. of samples in train set: '+str(len(trainLoader.
        print('No. of samples in test set: '+str(len(testLoader.da
```

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

Define network architecture

Now, once that is done let us look into the size of the data, now, now, this is also pretty standard. The reason why we put it down every time is just in order to check that the correct size of the data set has been loaded. So, you have the correct data set and everything available over there, but say in case they have how was a collapsing or within this particular session when it was loading, it could not load it down perfectly, then it creates a messy issue with you. So, this is just a sort of a debugging and safe side state which we typically put down for all of our exercises as you have seen till now.

(Refer Slide Time: 04:47)



```
In [4]: net1 = models.resnet18() # Training from scratch
        net2 = models.resnet18(pretrained=True) # End-to-end fine-
        net3 = models.resnet18(pretrained=True) # Training only th
        print(net1)
```

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

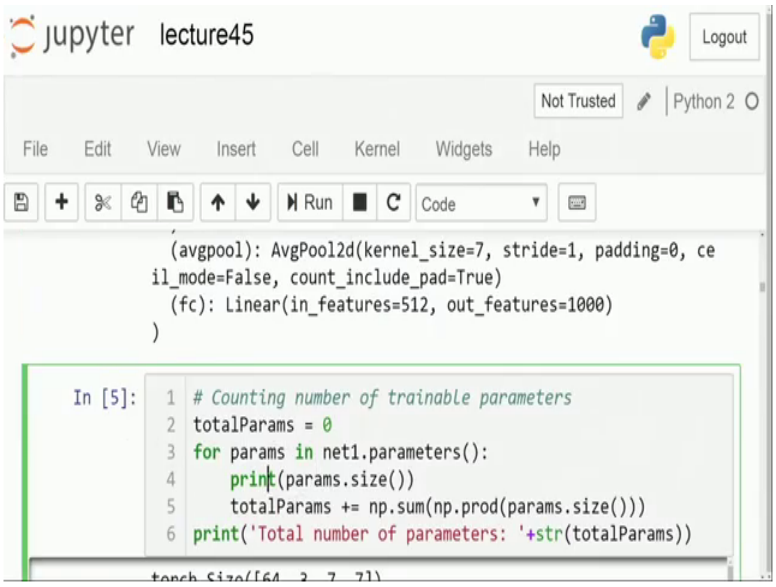
Now, comes down to our model part of it. So, that first network which is just the network architecture taken down from our taught fusion models. The second net is where it is taken down as a pre trained model. So, this one pre trained equal to true is what downloads the image net pre trained model for us, so the weights are also downloaded. So, if you are downloading it on the real time value you are running these exercises and watching this video.

Then you would see that, it consumes a lot of space on your it does consume a lot of your bandwidth; and then the total space also consumed on the hard drive is also large. Now, one of the reasons why it would be consuming more bandwidth is, because it needs to download the weights, and the size of the parameter space. In terms of the space complexity in bytes of the parameter space is what we had already discussed in the earlier lecture.

So, from there you can actually do a pen paper calculation based on your total number of parameters and multiply it with the precision and find out the total size of the file, so that that is easy and straight for you.

The next network over here is net three. And net three is also a pre trained model. So, the only thing which we do in net three is we change the terminal load and just update the terminal node so anyways we will have to download the whole model as such. Now, this is where we print only the first model over there to look down whether the model is correct or not. So, this was the if you even if you print down the next two models net 2 and net 3 they will come down as the same. So, it is just the same model as far as architecture definition goes down over there.

(Refer Slide Time: 06:16)



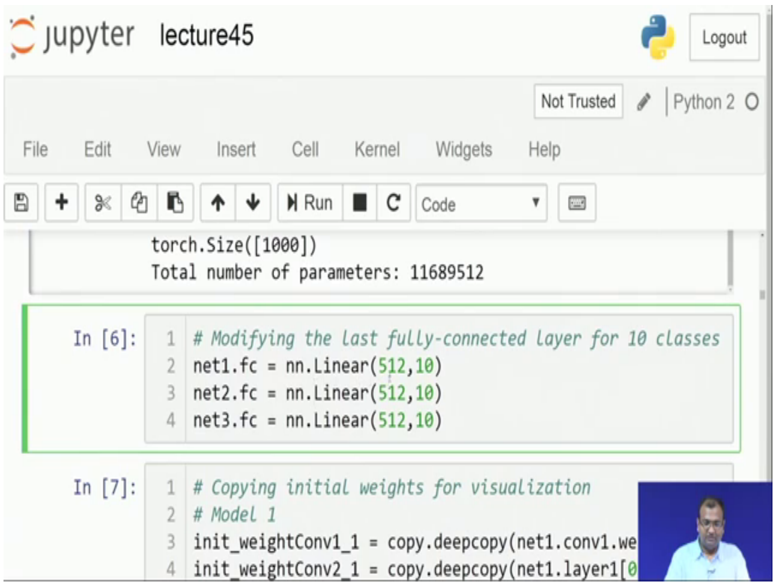
```
  (avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0, ce
il_mode=False, count_include_pad=True)
  (fc): Linear(in_features=512, out_features=1000)
)
```

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

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

Now, the next part is to count down the total number of parameters and whether we counted down from network 1 or network 2 or network 3 it makes the same sense they did. There would not be much of a change and this comes down to about 11 million parameters, so that is roughly say 2.5 times lesser than your GoogLENet the number of parameters within GoogLENet over there ok. Now, we go on to our modifications.

(Refer Slide Time: 06:41)



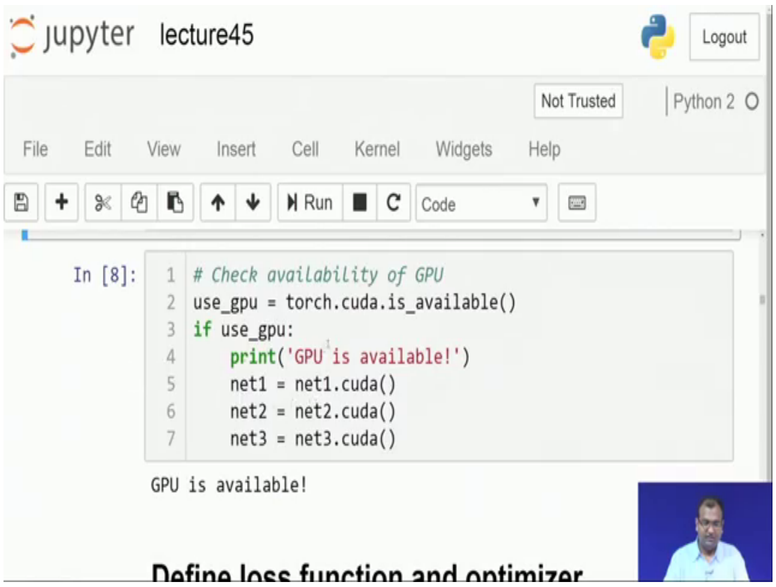
```
torch.Size([1000])
Total number of parameters: 11689512
```

In [6]:
```
# Modifying the last fully-connected layer for 10 classes
net1.fc = nn.Linear(512,10)
net2.fc = nn.Linear(512,10)
net3.fc = nn.Linear(512,10)
```

In [7]:
```
# Copying initial weights for visualization
# Model 1
init_weightConv1_1 = copy.deepcopy(net1.conv1.we
init_weightConv2_1 = copy.deepcopy(net1.layer1[0
```

So, the modification which we need to do over here is instead of a 1,000 class classification problem, this is now becomes a 10 class classification problem. So, for each of these networks we are just going to change the final node which was 512 nodes to 1,000 nodes instead of that, we make it 512 nodes to 10 nodes. And then there is the only changes which comes down in net 1, net 2, and net 3 the f c layer is where the change comes down ok. Now, having done that next what we do is we just copy down all the weights and keep it for our use.

(Refer Slide Time: 07:11)



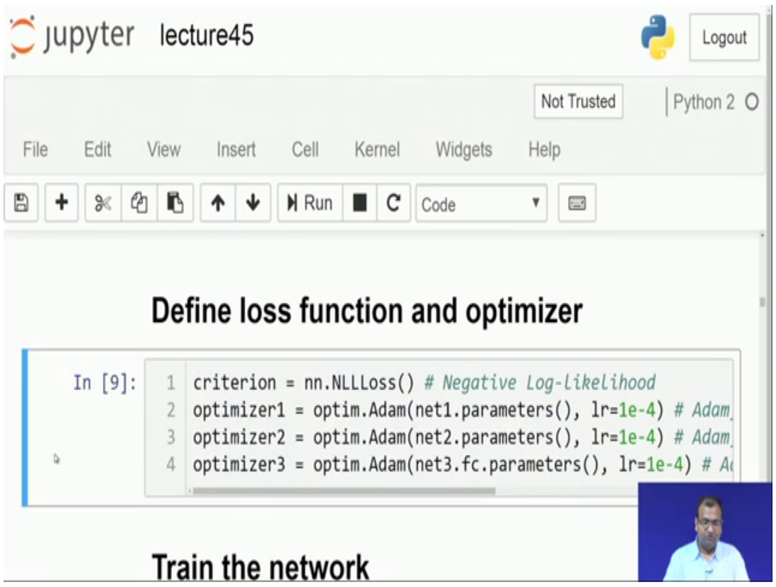
```
In [8]:
# Check availability of GPU
use_gpu = torch.cuda.is_available()
if use_gpu:
    print('GPU is available!')
    net1 = net1.cuda()
    net2 = net2.cuda()
    net3 = net3.cuda()
```

```
GPU is available!
```

Define loss function and optimizer

And then we check out if a GPU is available then the models are converted onto cuda. Now, till this part it is straightforward what we have done in the earlier cases.

(Refer Slide Time: 07:23)



Define loss function and optimizer

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

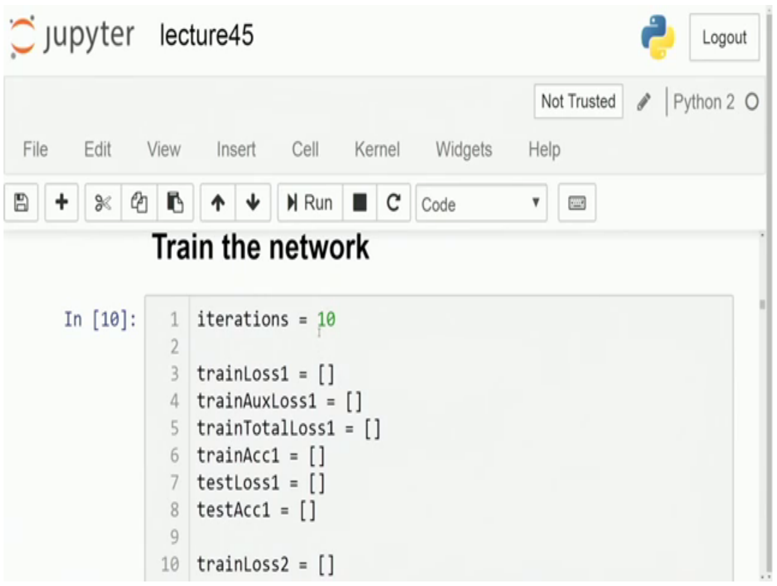
Train the network

Now, coming down to our loss functions we still stick down to negative log likelihood loss which is one of the losses which you are doing for any of our classification problems. For optimizers is where the change comes in. So, in case of net 1, we need all the parameters to be taken down and the optimizer which we have is Adam.

In net 2 also, we are going to take down all the parameters. So, net 1 is where you had judge the model and they were randomly initialized bits over there. So, all the parameters will be updated end to end. Net 2 is where you had the model downloaded and the weights where something which belong to the image net pre trained model and still although you have updated only the last layer, but you will be updating weights everywhere over there.

And net three is where you have your image net pre trained model, you have replaced only the last layer from 512 to 10 neurons that is so only replacement you have done. And you would like to update only the last layer over there. So, if you get back onto the code then you can see that the parameters which you are taking down and just fc parameters over the net three dot fc so that is the only parameter which we take down for net three. And this is by the same logic as we had done in case of GoogLENet as far.

(Refer Slide Time: 08:36)



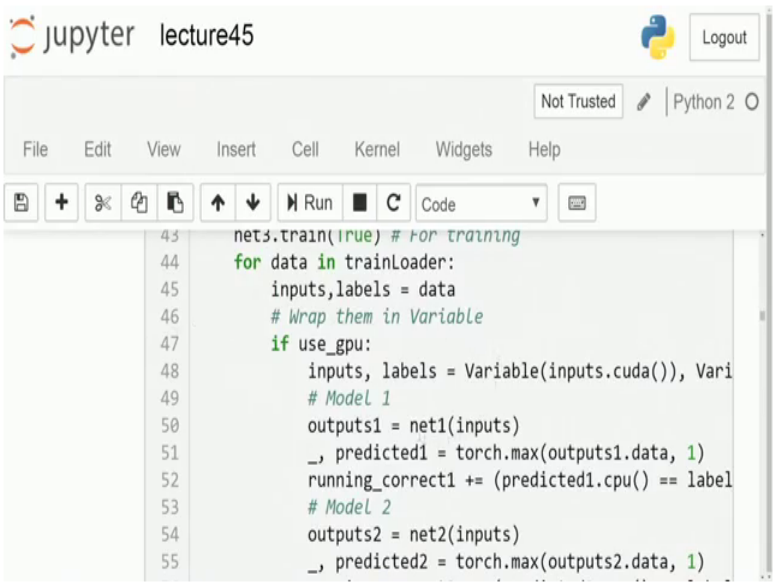
Train the network

In [10]:
```
iterations = 10

trainLoss1 = []
trainAuxLoss1 = []
trainTotalLoss1 = []
trainAcc1 = []
testLoss1 = []
testAcc1 = []

trainLoss2 = []
```

Now, going down to train this network since the total number of floating point operations which you do for a residual network is much lesser than the number of floating point operations which you do for the GoogLENet. And for that reason it takes much lesser time actually do train it out.

So, what we do over here is we train it down for 10 epochs in order to look down at a long term performance. What comes down? So in the GoogLENet case we had just trained it down with 5 epochs and here we are just going to train it down with 10 epochs to look into the change over there. Now, we just have these buffers created down and then we keep on running over the epoch pointers over here.

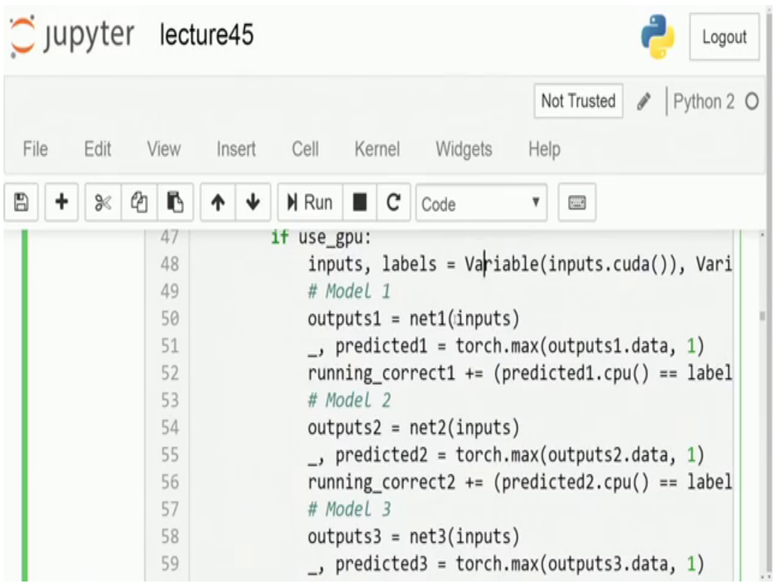
(Refer Slide Time: 09:09)



```
    net3.train(True) # For training
    for data in trainLoader:
        inputs,labels = data
        # Wrap them in Variable
        if use_gpu:
            inputs, labels = Variable(inputs.cuda()), Vari
            # Model 1
            outputs1 = net1(inputs)
            _, predicted1 = torch.max(outputs1.data, 1)
            running_correct1 += (predicted1.cpu() == label
            # Model 2
            outputs2 = net2(inputs)
            _, predicted2 = torch.max(outputs2.data, 1)
```

Now, the first thing which you do is within an epoch which is going to load down one batch of the data. And if you see that there is a GPU available with you then you convert it onto cuda which is your typecasting or in terms of a hardware or computer organization is perspective this was a DMA transfer which you were initiating. And this DMA transfer is what transfers all of these inputs which were otherwise deciding only on the CPU RAM onto your GPU RAM. And if the model is present on the GPU RAM the data is present on the GPU RAM then only you will be able to make use of the GPUs processing capabilities in order to solve it out.

(Refer Slide Time: 09:52)

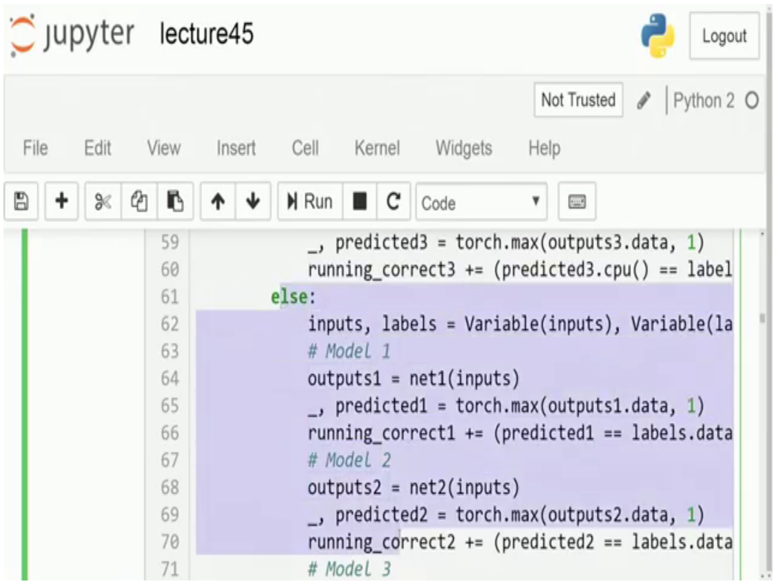


```
        if use_gpu:
            inputs, labels = Variable(inputs.cuda()), Vari
            # Model 1
            outputs1 = net1(inputs)
            _, predicted1 = torch.max(outputs1.data, 1)
            running_correct1 += (predicted1.cpu() == label
            # Model 2
            outputs2 = net2(inputs)
            _, predicted2 = torch.max(outputs2.data, 1)
            running_correct2 += (predicted2.cpu() == label
            # Model 3
            outputs3 = net3(inputs)
            _, predicted3 = torch.max(outputs3.data, 1)
```

So, that goes down with the variables over there. Next, what I need to do is I will have to run down my inputs through my network. And this is just network one and get down my outputs. Now, for once I have this one I just find out what is my predictions coming down. Now, keep in mind one thing that we do not have an auxiliary classification node in anyway present within this particular network. This is a just as plain simple residual network. So, you just have the end part over there and that is why you are not aggregating or finding out auxiliary losses in any way as well ok.

Now, comes down the second module and in the second model what you see is that we no more make any type castings available, because my inputs are still over there. It is a same input which will be passed down through my second network, but then outputs have to be collated and collected down in a separate variable container that is output 2 which corresponds to network two over there.

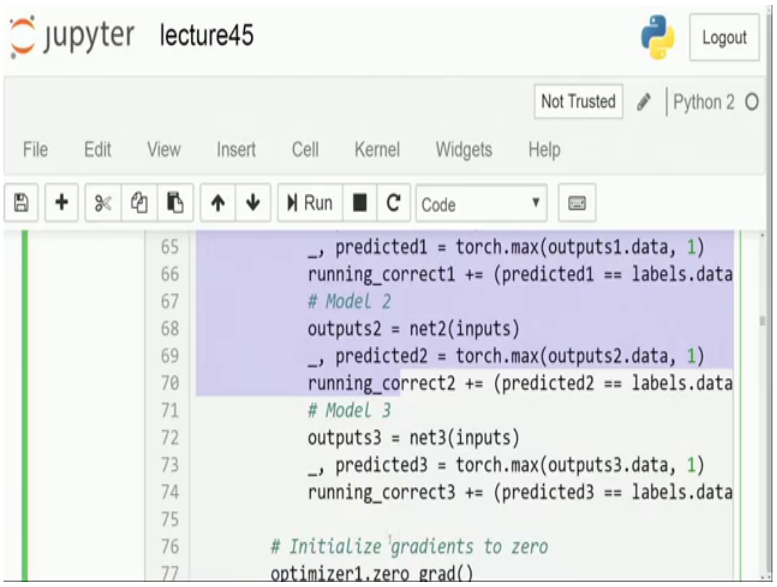
(Refer Slide Time: 10:49)



```
            _, predicted3 = torch.max(outputs3.data, 1)
            running_correct3 += (predicted3.cpu() == label
        else:
            inputs, labels = Variable(inputs), Variable(la
            # Model 1
            outputs1 = net1(inputs)
            _, predicted1 = torch.max(outputs1.data, 1)
            running_correct1 += (predicted1 == labels.data
            # Model 2
            outputs2 = net2(inputs)
            _, predicted2 = torch.max(outputs2.data, 1)
            running_correct2 += (predicted2 == labels.data
            # Model 3
```

So, I run the same sort of predictors and then find out what is my number of correct classification is done. So, it is the same thing over here within the else routine which is when you do not have access to a GPU on your system, but you are running it down purely on a CPU.

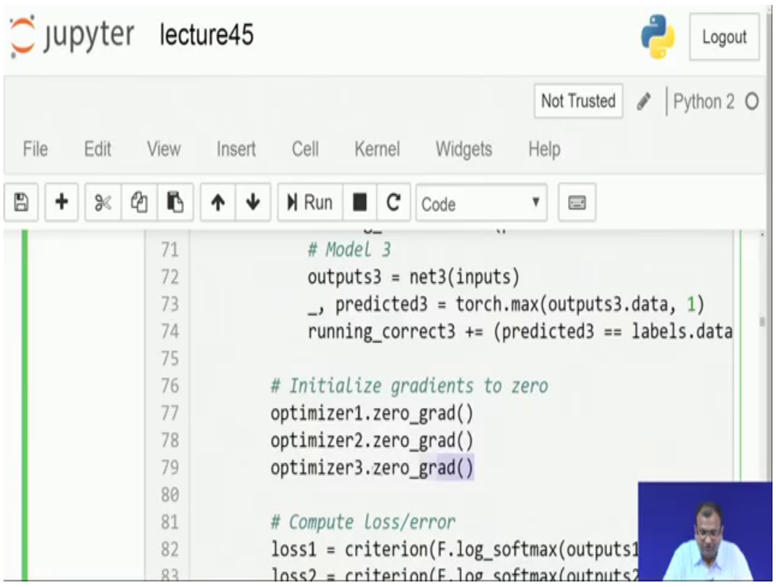
(Refer Slide Time: 10:51)



```
            _, predicted1 = torch.max(outputs1.data, 1)
            running_correct1 += (predicted1 == labels.data
            # Model 2
            outputs2 = net2(inputs)
            _, predicted2 = torch.max(outputs2.data, 1)
            running_correct2 += (predicted2 == labels.data
            # Model 3
            outputs3 = net3(inputs)
            _, predicted3 = torch.max(outputs3.data, 1)
            running_correct3 += (predicted3 == labels.data

        # Initialize gradients to zero
        optimizer1.zero_grad()
```

Now the next thing what you do is you zero down the gradients for all the optimizers.

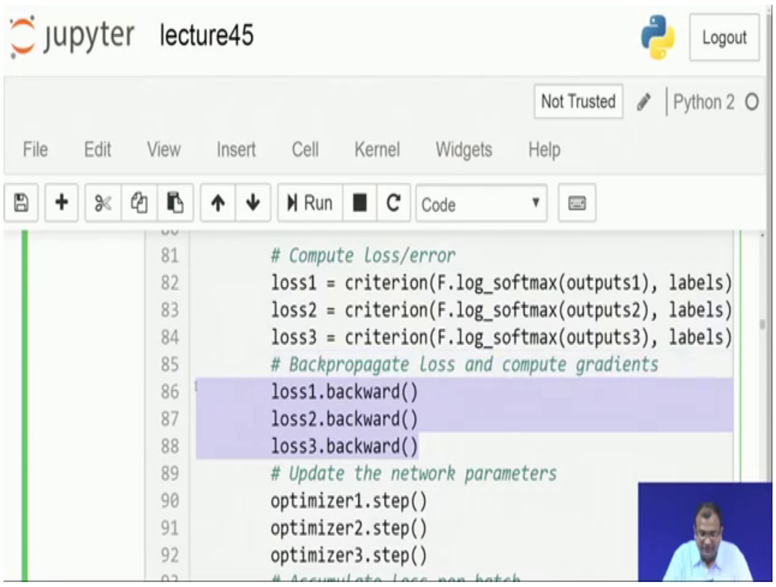
(Refer Slide Time: 10:54)



```
        # Model 3
        outputs3 = net3(inputs)
        _, predicted3 = torch.max(outputs3.data, 1)
        running_correct3 += (predicted3 == labels.data

    # Initialize gradients to zero
    optimizer1.zero_grad()
    optimizer2.zero_grad()
    optimizer3.zero_grad()

    # Compute loss/error
    loss1 = criterion(F.log_softmax(outputs1
    loss2 = criterion(F.log_softmax(outputs2
```

And then you start calculating the losses. Now, it is the same criteria which is being used for each of them.

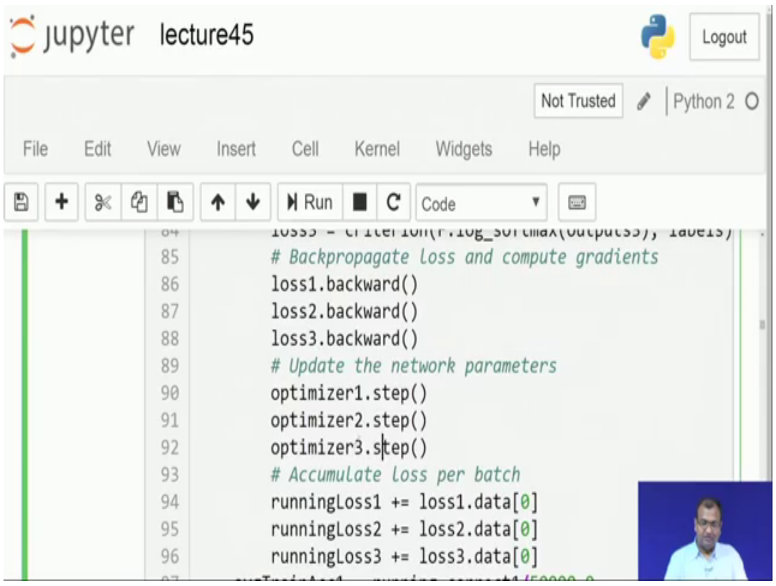
(Refer Slide Time: 11:09)



```
    # Compute loss/error
    loss1 = criterion(F.log_softmax(outputs1), labels)
    loss2 = criterion(F.log_softmax(outputs2), labels)
    loss3 = criterion(F.log_softmax(outputs3), labels)
    # Backpropagate loss and compute gradients
    loss1.backward()
    loss2.backward()
    loss3.backward()
    # Update the network parameters
    optimizer1.step()
    optimizer2.step()
    optimizer3.step()
    # Accumulate loss per batch
```

And you find out your losses, and then you find out what is the gradient over all of these losses.

(Refer Slide Time: 11:12)

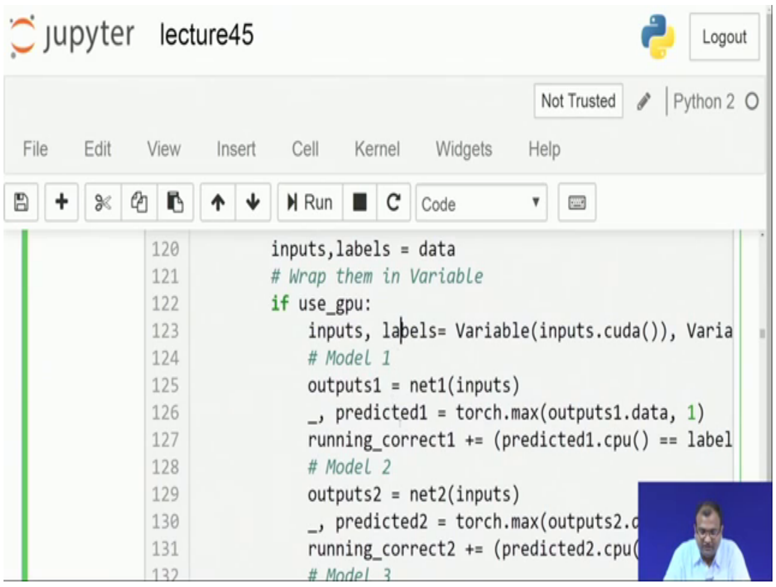


```
        # Backpropagate loss and compute gradients
        loss1.backward()
        loss2.backward()
        loss3.backward()
        # Update the network parameters
        optimizer1.step()
        optimizer2.step()
        optimizer3.step()
        # Accumulate loss per batch
        runningLoss1 += loss1.data[0]
        runningLoss2 += loss2.data[0]
        runningLoss3 += loss3.data[0]
```

And once you have the gradient or the del del w of j w calculated via are this backward operator on each of these classes is when you get your optimizer running. Now, when I do an optimizer dot step you need to keep one thing in mind that optimize a three which corresponds to actually network 3. So, if we get up over here, you would find that within your optimizer definition for your network 3 is what optimizer 3 is defined. And optimizer 3 is defined parameters over which it works out or it just the fully connected layers parameter on network 3.

So, whenever if there is a update or update rule in terms of an optimizer dot step. So, in case of optimizer 1 and 2, it is going to update all the parameters or all the weights of the network whereas for optimizer 3 it is going to update only the last terminal node over there. Then we calculate out our losses and just have them stored down in your container.

(Refer Slide Time: 12:07)

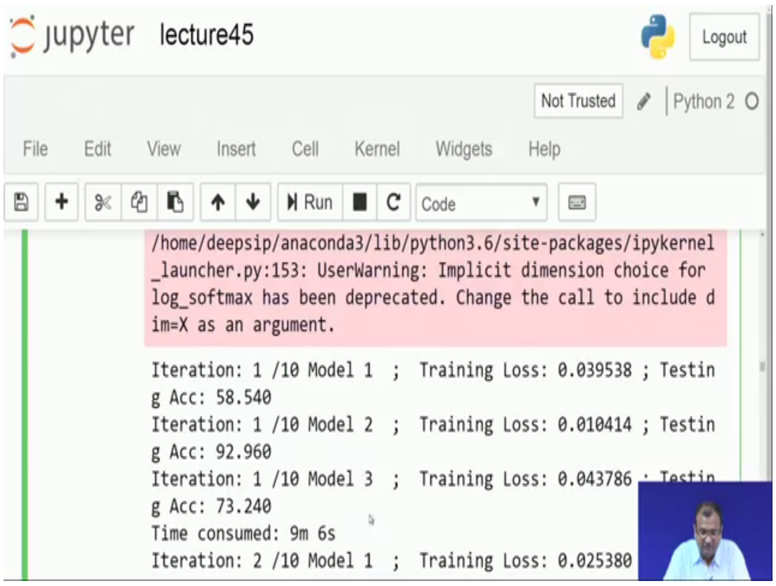


```
        inputs,labels = data
        # Wrap them in Variable
        if use_gpu:
            inputs, labels= Variable(inputs.cuda()), Varia
            # Model 1
            outputs1 = net1(inputs)
            _, predicted1 = torch.max(outputs1.data, 1)
            running_correct1 += (predicted1.cpu() == label
            # Model 2
            outputs2 = net2(inputs)
            _, predicted2 = torch.max(outputs2.d
            running_correct2 += (predicted2.cpu(
            # Model 3
```

Now, once having done that the next is at the end of each epoch what is the networks performance in terms of the testing data over there or the validation performance. So, we do the same thing. So, I just see if GPU is available then type casted typecast my inputs and labels onto a GPU variable. And then I run down through my network 1, similarly I through a network 2, and there is no retyped casting being done over there.

Now, this runs from for the CPU version over there as well and then we find out what are the losses which comes down, and then take down my running losses for my testing part over there. And then these were just some plots which we needed to do and now let us look into what it starts down.

(Refer Slide Time: 12:43)



```
/home/deepsip/anaconda3/lib/python3.6/site-packages/ipykernel
_launcher.py:153: UserWarning: Implicit dimension choice for
log_softmax has been deprecated. Change the call to include d
im=X as an argument.

Iteration: 1 /10 Model 1  ;  Training Loss: 0.039538 ; Testin
g Acc: 58.540
Iteration: 1 /10 Model 2  ;  Training Loss: 0.010414 ; Testin
g Acc: 92.960
Iteration: 1 /10 Model 3  ;  Training Loss: 0.043786 ; Testin
g Acc: 73.240
Time consumed: 9m 6s
Iteration: 2 /10 Model 1  ;  Training Loss: 0.025380
```

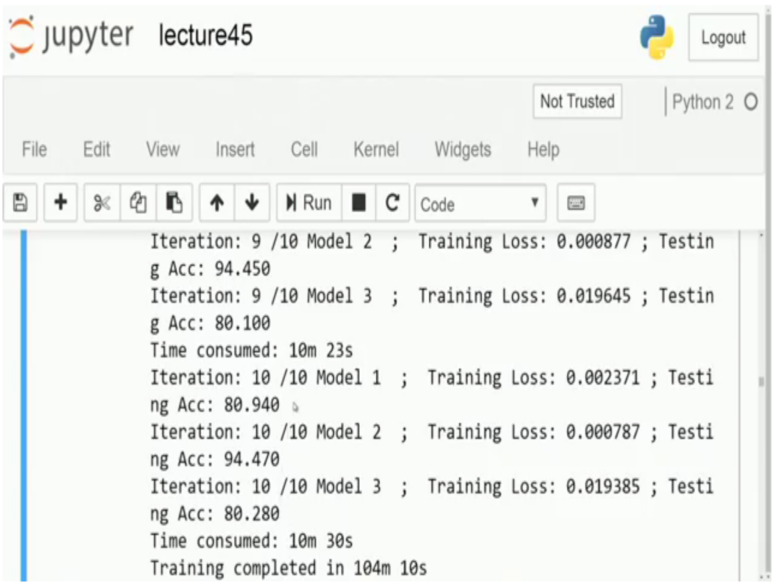
So, if you look at the first iteration itself so at the end of first iteration you see that the training loss for network 1 which is where it was randomly initialized. And there was no modifications being so everything had to be updated. So, it was randomly initialized and no previous input was taken out. So, it starts at the training loss about 0.03.

The second model net 2 is where you had it pre initialized for an image net problem, and then you are updating end-to-end the whole model, whereas model 3 or the network 3 is where you had updated only the last terminal nodes over there and not the other nodes. So, this was having our performance which so the error over there is definitely something which is larger than the error of my model 2 on network 2 over there and that is that is for sure.

However, if you look into the training error over there, the training error is still larger than the training error in the earlier case just by our few factors. But then if you look into the generalizability you would see that the testing accuracy for network 1 is lower than the testing accuracy for network 3. And one of the reasons why this was happening down has because network 3 is already pre-trained. So, some of these early layered features are quite tuned onto capture these natural object like behaviours and their features.

And for that reason you saw that although it was not fitting out properly on the train set where it says model conformity to the training data set is not so great. But then overall capably to adjust to a larger corpus and a set of data is really good. So, and for that reason the generalizability is much better for my testing accuracy case for model 3. Now, it keeps on going like this and let us look what happens at the end of 10 epochs.

(Refer Slide Time: 14:33)



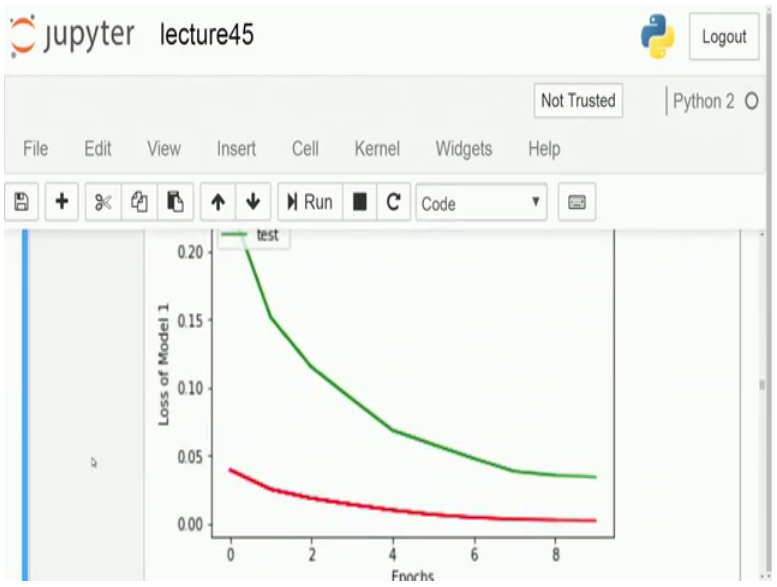
```
Iteration: 9 /10 Model 2  ;  Training Loss: 0.000877 ; Testin
g Acc: 94.450
Iteration: 9 /10 Model 3  ;  Training Loss: 0.019645 ; Testin
g Acc: 80.100
Time consumed: 10m 23s
Iteration: 10 /10 Model 1  ;  Training Loss: 0.002371 ; Testi
ng Acc: 80.940
Iteration: 10 /10 Model 2  ;  Training Loss: 0.000787 ; Testi
ng Acc: 94.470
Iteration: 10 /10 Model 3  ;  Training Loss: 0.019385 ; Testi
ng Acc: 80.280
Time consumed: 10m 30s
Training completed in 104m 10s
```

So, at the end of ten epochs what you see for this particular model is that your model number 1 and model number 3. So, model number 1 is where do you started with some random initialization and you were updating all the weights over there. Model number 3 is where do you started with an image net initialization, but you are updating only the terminal nodes over there.

Now, they give you almost the same kind of a accuracy 80 percent of an accuracy for both of them. A tad bit say almost like 60.64 is the difference in the accuracy which comes down. But then 0.164 is not much of a difference 0.58 is the difference 0.56 is the difference, but then that is not much of a difference to be attributed over there. Both of them are almost at 80 percent behaviour.

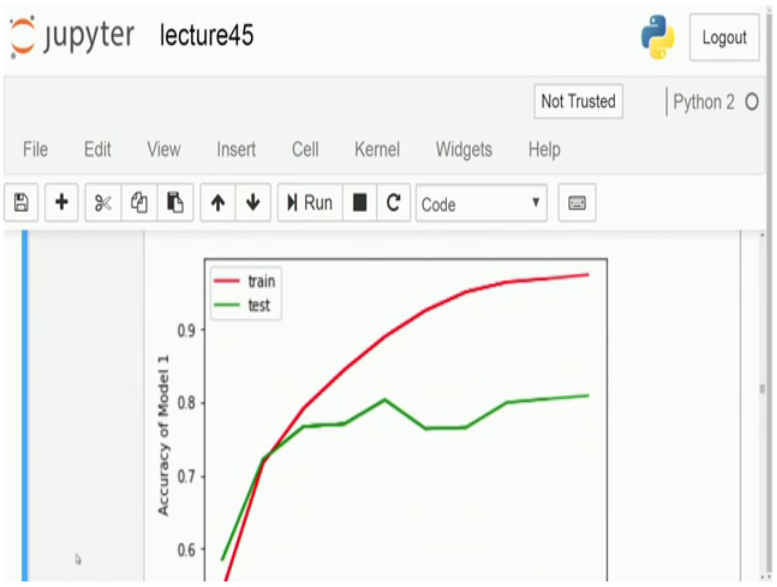
Whereas, model 2 is where you took a pre trained model and then using this pre trained model what you had done is you had updated only the terminal so you had updated all the layers in to end over there. And because of that reason model 2 is where you get down a much better performance accuracy coming down ok. Now, let us get into the performance curves as we see over there.

(Refer Slide Time: 15:41)



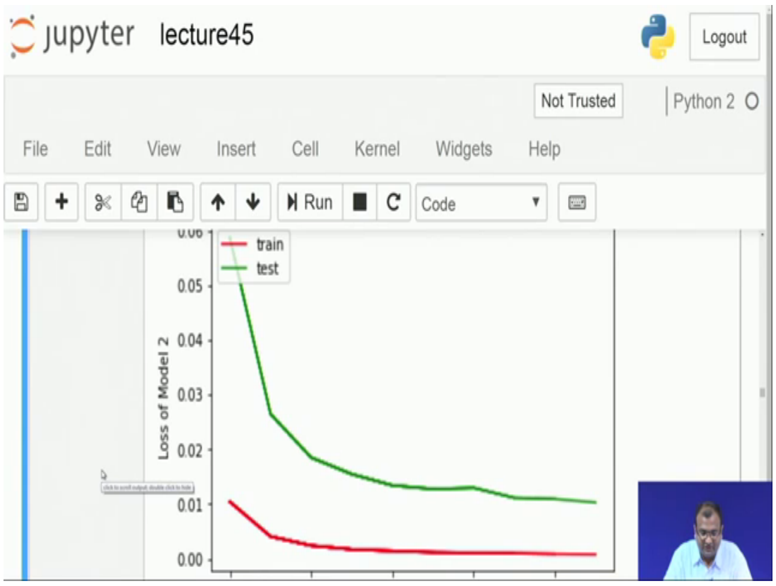
So, this is for a model 1 where you have your training curve and you are testing curve over there. So, definitely the losses fall down.

(Refer Slide Time: 15:47)



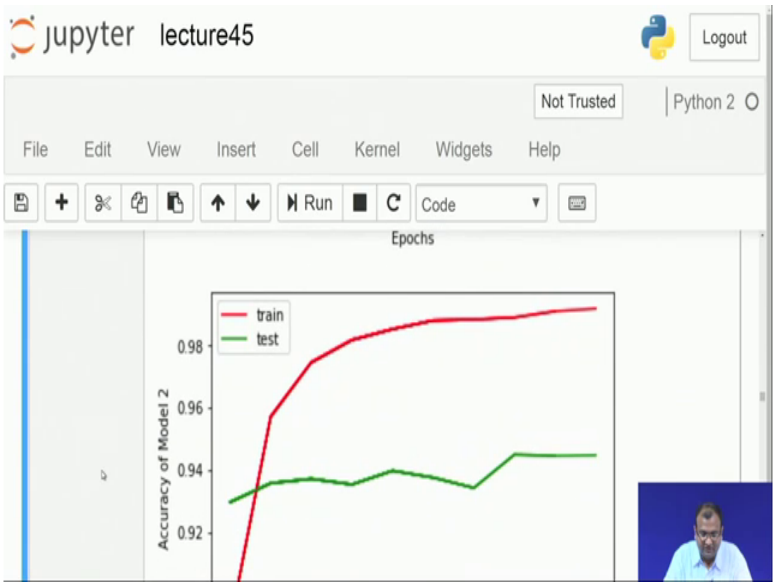
And if you look down into the model one you see that there is some sort of an over fitting which comes on quite early on. So, you can see that there are training accuracy is increasing, but then your test accuracy does not increase starts sort of like wavering over there at around 80 percent.

(Refer Slide Time: 16:08)



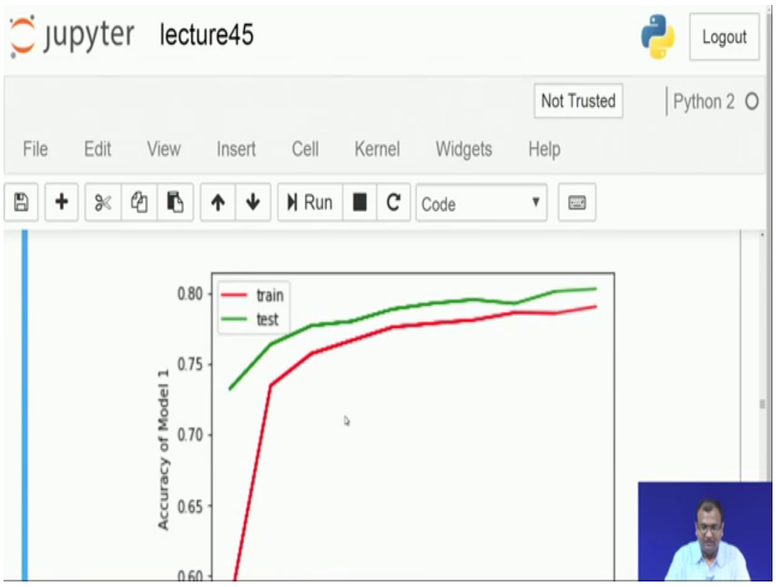
Now, if you look down for a model 2, where you started with a very good prediction and then did a end-to-end pre training over there.

(Refer Slide Time: 16:11)



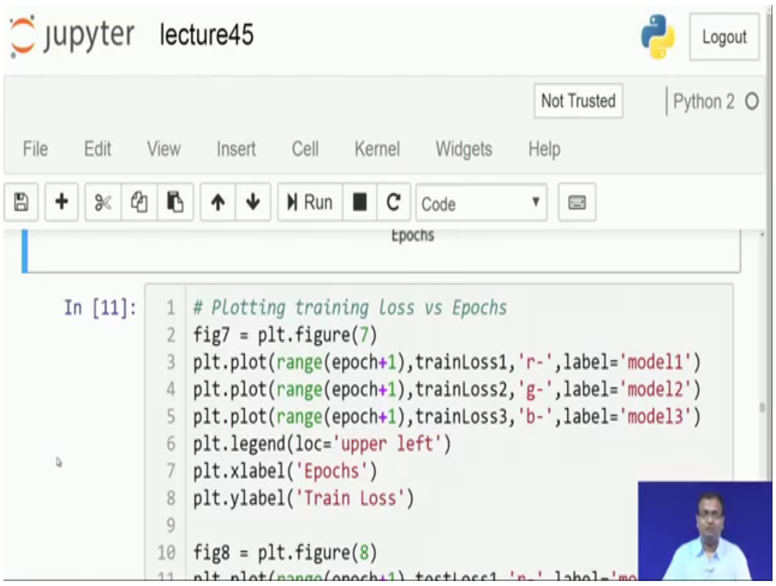
So, your starting initial accuracy is almost at 93 percent and you go up to a 94 percent. And most likely you would just saturate over here. So, even if you keep on increasing on the training data set, it is just an over fitting, but then over fitting does not have much of an effect on the generalizability over here.

(Refer Slide Time: 16:28)



Now, comes down the last model where I was just updating only my terminal nodes over there. So, when I have updating that you see the accuracy still my testing accuracy is still going to go up, but then it is much below the 80 percent mark. Whereas is in the earlier case where I was updating end-to-end of the whole model, I could very easily come down to 94 percent. So, that is a major improvement which we have with the case where we try to update all the neurons over there all the layers from an end to end perspective.

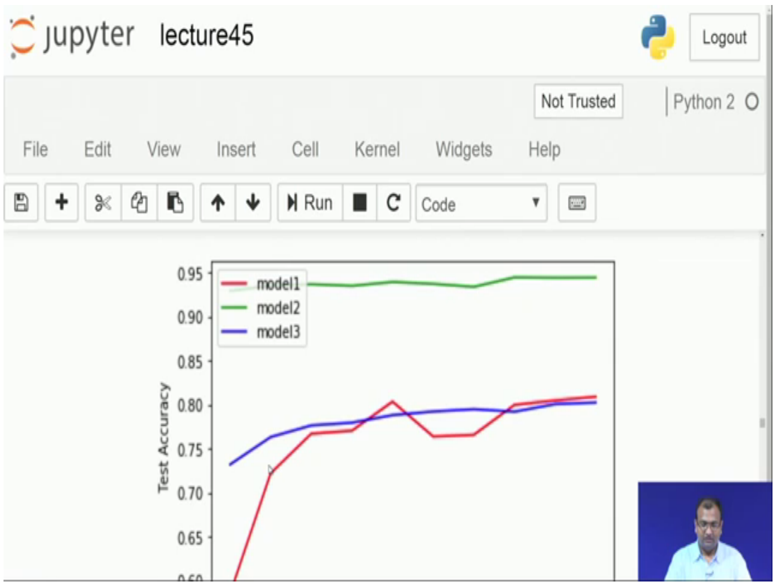
(Refer Slide Time: 16:48)



```
In [11]:  # Plotting training Loss vs Epochs
          fig7 = plt.figure(7)
          plt.plot(range(epoch+1),trainLoss1,'r-',label='model1')
          plt.plot(range(epoch+1),trainLoss2,'g-',label='model2')
          plt.plot(range(epoch+1),trainLoss3,'b-',label='model3')
          plt.legend(loc='upper left')
          plt.xlabel('Epochs')
          plt.ylabel('Train Loss')

          fig8 = plt.figure(8)
```

Now, what we also did plot is the training losses the test losses and the test accuracy. So, this was a very good comparison which we have for us actually.

(Refer Slide Time: 17:02)



So, if you look into model 2 is something which is more conformal in fact, like if you were stopping down even at one epoch or two epoch you still see that there is a same sum. So, it is almost a flat line which goes around at 94 percent not much of a change which keeps on happening even if you are doing it for more number of epochs. So, that is the generalizability which you get done.

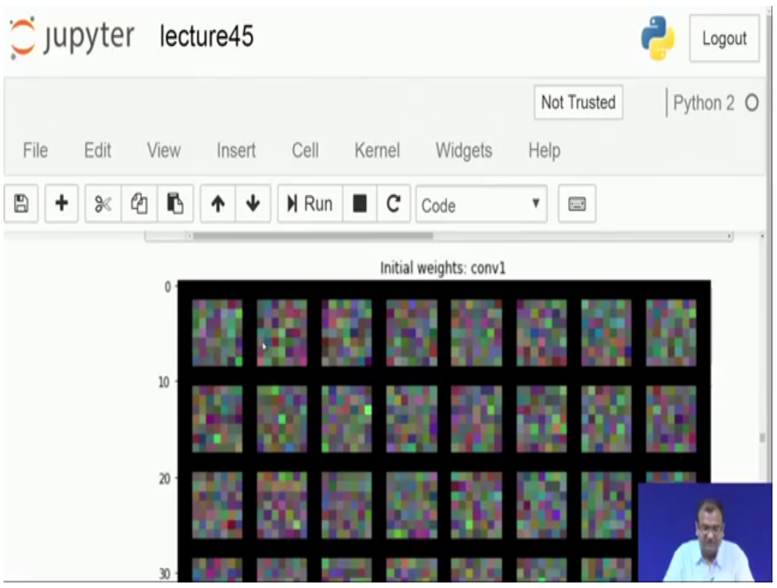
And typically if you if you compare down in the earlier case of a GoogLENet, where you have to wait for at least three or four epochs by the time you saw a good generalizability performance coming down for a image net pre trained model which was trained end-to-end. Over here, you see that already at the end of first epoch its refined in such a way that the generalizability is already captured over there; and there is no much of a difference which you get even if you keep on continuing over more number of epochs.

Now, one of the reasons this attribute set out because in a residual network, you are at the end of it you are just learning down residual connections or what are the what is that extra information which you need to recover from the signal in order to be able to classify it out.

Now, this excess information which you recover because anyways you have your residual transfers which is which transfers a part of the input to the output itself in order for classification, now that is a good part of it. Now, intermediate what you are learning down within this convolution corners are just the difference between these information which has to be transferred. So, is there some extra kind of a feature which has to be learnt up which is a difference feature between the input and output over there.

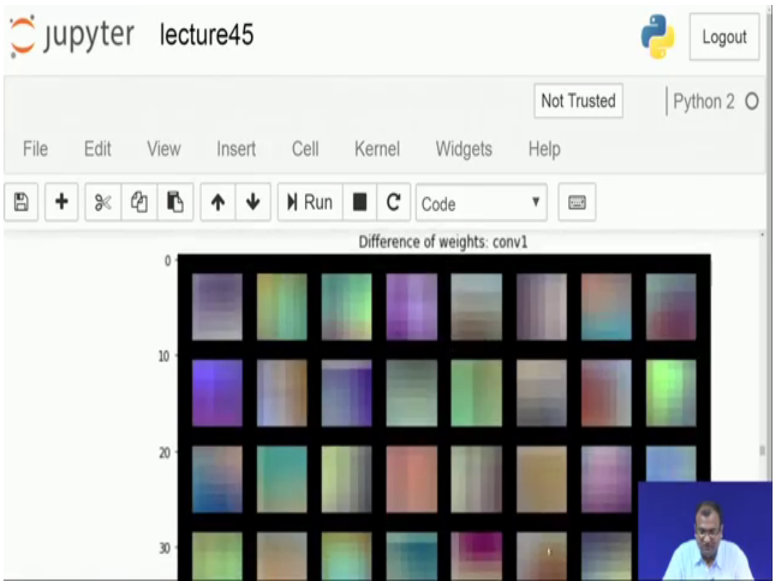
Now, there are not much of difference features which anyways would have to be learned down. And for that particular reason when you take down a residual network and the start itself it is it is so good. So, the generalizability of residual network is there already in place; and for that that is the thing which you see also substantiated through this particular experiment over here.

(Refer Slide Time: 18:58)



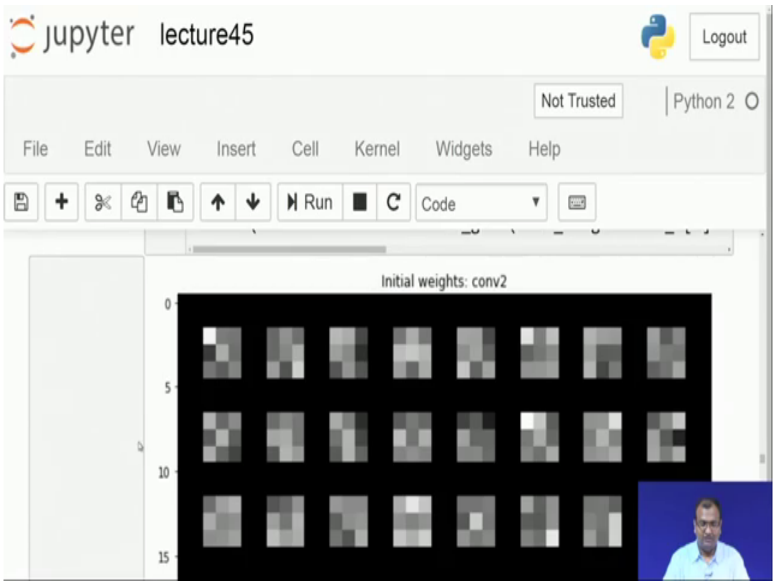
Now, let us look into the weights over there. So, this is for my network one which is randomly initialized and then I look down what happens at the end of my training process. So, each of these is a 7 cross 7 kernel, and then I have 64 of such kernels in my first layer itself.

(Refer Slide Time: 19:12)


jupyter lecture45
Logout
Not Trusted
Python 2
File Edit View Insert Cell Kernel Widgets Help
Run
Code
Difference of weights: conv1
0
10
20
30

Now, this was at the end of the training. And if you look into the difference over here this is what we get down. So, this kind of difference visualizations is the same like we had done for our residual network case when we were doing around the regular one. Now, if you look into the second layer weights over there, so we just take down one of these convolution colonel over there or 3 cross 3 convolution colonel. And we take all the 64 channels which this 3 cross 3 convolution colonel had. So, this was before training, this is after training and then these were the differences which got in the whole process of training.

(Refer Slide Time: 19:25)



Now, let us look into the weights of a pre trained network. So, this is what other weights which you look down after pre trained image net pre trained model for a several network. You can make out much distinct features over here from these bits as compared to the earlier case. So, these look almost like some sort of gradient operators which come down over here.

Now, if you take a pair of weight, and try to look into the difference between pre trained so just a pre trained model and a refined model, there is not much of a difference comes out. So, that is what we see in these weights over there. So, there are slight differences majority of them are near zero differences. And there may be some accentuations around certain colours. And this is the domain adaptation point which takes place. So, there will be a certain colours or certain features the blurred coronel sizes and everything which is changing and that is the minor amount of difference which comes in.

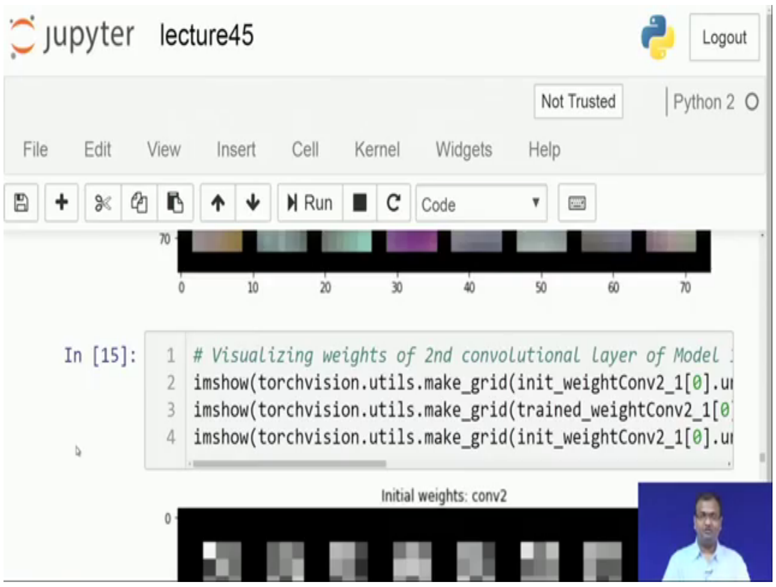
However, keep one thing in mind that it is not necessary that none of these weights will be updated this is what you see down over here. So, even if you had a pre trained and you are retraining out over there, so certain neurons which are not positively impacting the classification performance or the ones which will get which will have to be updated in order, so that they influence their whole classification process in a good way, so that is the fact which you see over here.

Now, in the second layer also you would see this is the kind of matrix which you get for a pre trained network. And this is after you have trained it over 10 epochs and these are the different bits. So, you see that not just the first layer but also the second layer second residual block layer is also where there is an update of weights which goes down.

Now, let us look into the third configuration where we were just updating the final layers over there. So, definitely that means, that since I was just updating the parameters from the final layer, so none of these initial layers we are getting updated. So, there will not be any change of weights as such coming down. In either the first layer or the second layer or anything which goes down before that fully convolutional layer fully connected layer. So, none of these convolutional layers will have any of my weight updates in the third configuration. So, this is what I had to show down for transfer learning case.

So, as such transfer learning you have very simple process to do down. And for most of our practical applications is where we would preferred going down with the transfer learning problem.

(Refer Slide Time: 21:55)

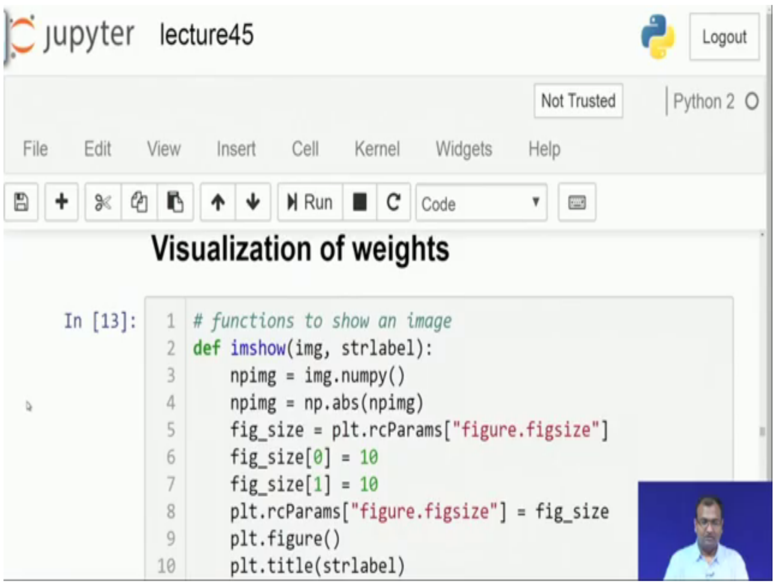


In [15]:

```
# Visualizing weights of 2nd convolutional layer of Model
imshow(torchvision.utils.make_grid(init_weightConv2_1[0].u
imshow(torchvision.utils.make_grid(trained_weightConv2_1[0
imshow(torchvision.utils.make_grid(init_weightConv2_1[0].u
```

And one thing which you need to keep in mind is; obviously, when you are updating and modifying the network architectures over there. So, what are the parameters which you will be learning and that that definitely decides a significant in a significant way it decides on what will be your batch sizes and everything. So, you can in some of these cases say for this kind of an image net kind of a problem over here and for these two cases over there.

(Refer Slide Time: 22:22)



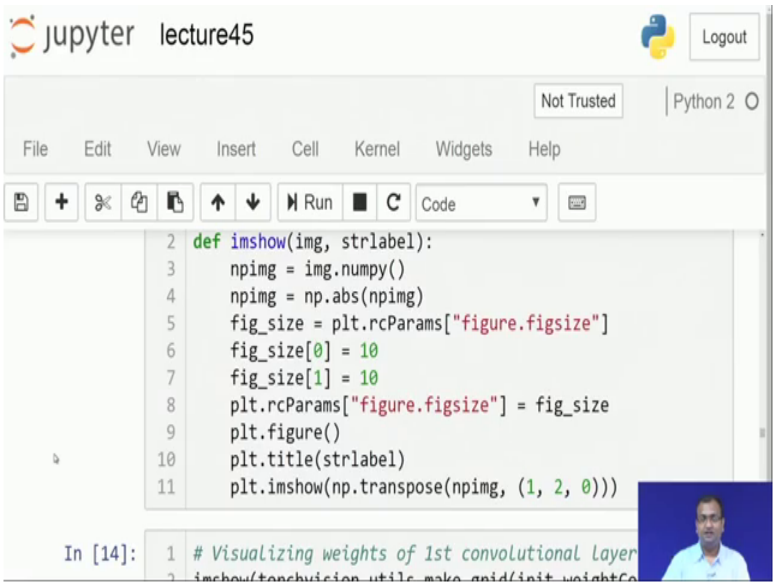
Visualization of weights

In [13]:
```python
# 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)
```

You would find out that obviously, training down end to end is something which is performing good, but then even if you are not able to so it. So, this is the trade off which you have. So, if you have a memory and GPU availability where you can choose to train the network end-to-end then it is well and good.

But then if you have a limited availability over there then you can choose to train only the final layers over there may be over a longer number of epochs, but it will still come down to a convergence at some point of time. So, this is a trade off which we do from a practical design perspective; so some of the aspects which we have discussed earlier in terms of memory issues.

(Refer Slide Time: 12:52)



```
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)))
```

```
In [14]: # Visualizing weights of 1st convolutional layer
```

And my device computer issues are what we handle down in this way using a transfer learning problem. So, with this we come down to an end for most of our work with just classification based convolutional neural network, image input and a class level output. The next week what we are going to do is something where you have an image net input, you have a matrix output or something of the same size of an image.

So, in terms of solving a semantic segmentation problem which is more of something within the regression framework pipeline; you also have a other issues of our region proposals and an average pooling for object localization which we will be doing over there.

So, then stay tuned for the advanced topics which comes out for the next few weeks over there, then going on to generating models and then subsequently onto video analytics. So, still then stay tuned.

Thanks.