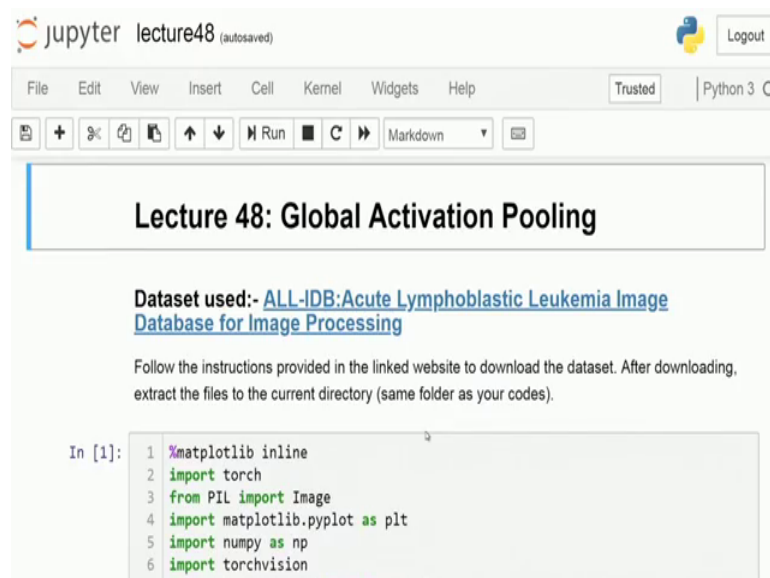


Deep Learning for Visual Computing
Prof. Debdoot Sheet
Department of Electrical Engineering
Indian Institute of Technology, Kharagpur

Lecture – 48
GAP + rCNN

Welcome, today we are going to look into one of these lab exercises, associated with activation pooling. And while in the last two lectures, you have studied about global activation pooling as one of them, and the other one for finding out regions was using the region proposal network, so that was an rCNN.

(Refer Slide Time: 00:21)



```
jupyter lecture48 (autosaved)
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3
+ % Run C Markdown
Lecture 48: Global Activation Pooling
Dataset used:- ALL-IDB: Acute Lymphoblastic Leukemia Image Database for Image Processing
Follow the instructions provided in the linked website to download the dataset. After downloading, extract the files to the current directory (same folder as your codes).
In [1]: 1 %matplotlib inline
2 import torch
3 from PIL import Image
4 import matplotlib.pyplot as plt
5 import numpy as np
6 import torchvision
```

And, we had compared out the pros and cons of each of them; while in global activation pooling the advantage is quite distinct, as it comes out that, you do not need to train a network explicitly just for finding out where the object is located. So, that means that even if you do not have object localization data present with you, you can still train a network. And then using the activation maps coming out of this network, you can of this classification network, you can actually find out and localize by expressing. And that gives you more of a hotspot, kind of a behavior and where this object is present in the image.

Now, on the other side of it, you also had region proposal networks in which you needed to have some sort of region proposals given to you while training. And based on these

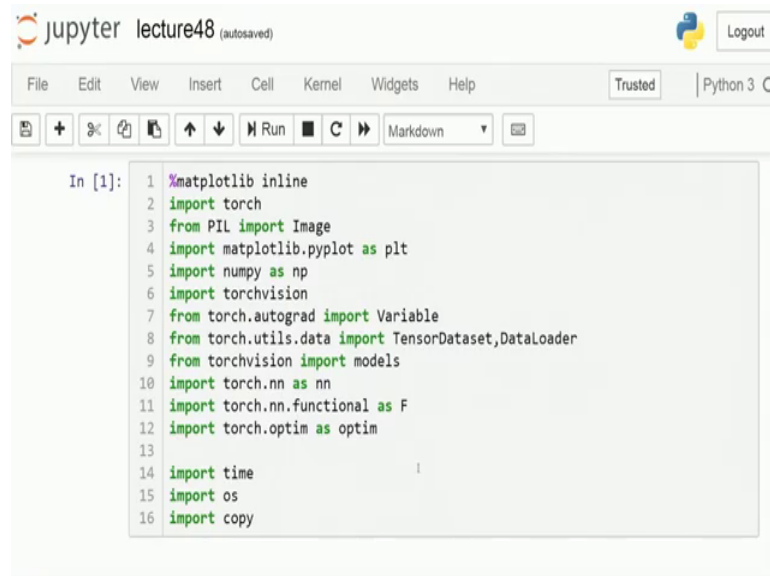
region proposals, you will be able to predict out where exactly the object is located. And now the advantage with the region proposal network was that, you can have partially occluded objects over there and still it can find out what is the total exhaustive span over there. So, if there is one person behind the other person, then the person who is in behind the bounding box, which comes up is pretty much distinct and confirm, met to the total physical appearance of the person over there.

Now, we will be taking up one of these examples in our lab session and that is just with activation pooling, now that is that is easier to implement and has a significant amount of use cases in lot of practical problems. And for this purpose, what we are doing is, we are revising one of these data sets, which we had earlier used in our auto encoder exercises and that was about this, microscopy image classification of white blood cells ok.

So, we are going to make use of that ALL-IDB data set once again, so refreshing back from your auto encoder days. And, what we are going to do is, if you remember then when we have downloaded the dataset, there were two distinct folders over there in the data set. One was just for your classification, and it was given in terms of smaller patches; and the other one which you had was the whole image over there on which you could find out WBCS and RBCS scattered down together, and the whole idea was like where is the WBC located.

Now, we are going to take this first one, which is trying to create a network which is just going to classify WBC is in two benign or malignant kind of a behavior over there. And then use this network and its activations subsequently, in order to find out where is that WBC exactly located on the image. So, this is overall what we are going to do.

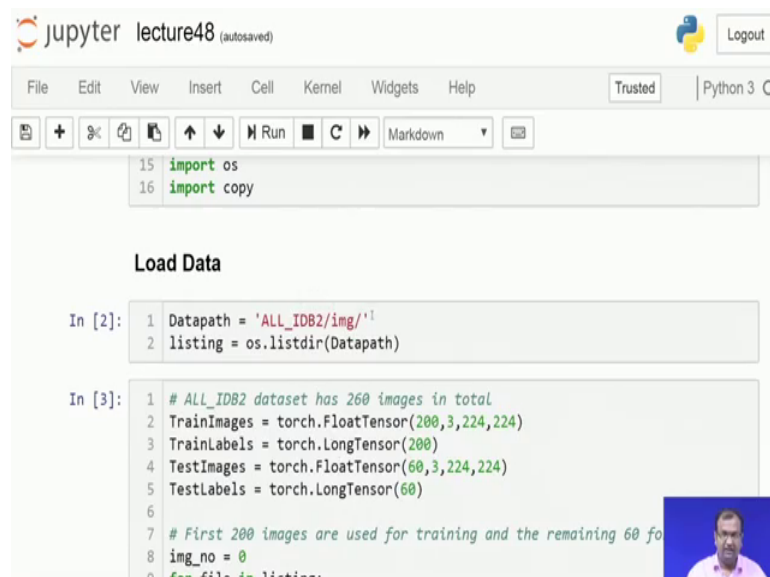
(Refer Slide Time: 02:57)



```
1 %matplotlib inline
2 import torch
3 from PIL import Image
4 import matplotlib.pyplot as plt
5 import numpy as np
6 import torchvision
7 from torch.autograd import Variable
8 from torch.utils.data import TensorDataset, DataLoader
9 from torchvision import models
10 import torch.nn as nn
11 import torch.nn.functional as F
12 import torch.optim as optim
13
14 import time
15 import os
16 import copy
```

Now, let us get into what we are trying to do over here, now the first part of the network is still quite common and conformal to what we have done. So, these are your standard header files, which we would just be needing. Now, once you have all of that done, and once you have this data downloaded from this location, so you get done basically two folders.

(Refer Slide Time: 03:13)



```
15 import os
16 import copy

Load Data

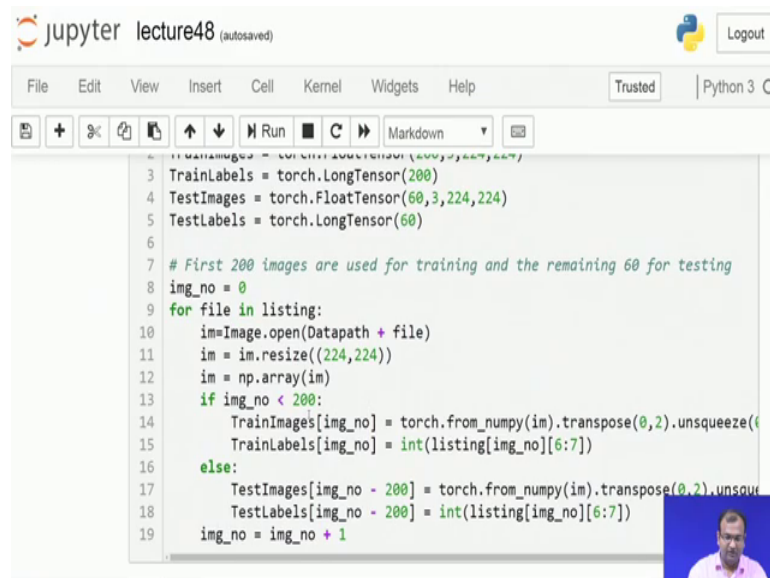
In [2]: 1 Datapath = 'ALL_IDB2/img/'
        2 listing = os.listdir(Datapath)

In [3]: 1 # ALL_IDB2 dataset has 260 images in total
        2 TrainImages = torch.FloatTensor(200,3,224,224)
        3 TrainLabels = torch.LongTensor(200)
        4 TestImages = torch.FloatTensor(60,3,224,224)
        5 TestLabels = torch.LongTensor(60)
        6
        7 # First 200 images are used for training and the remaining 60 fo
        8 img_no = 0
        9 for file in listing:
```

One of them is ALL IDB2, the other one is ALL IDB2. Now, IDB2 folder is the one where you have this smaller patches of images stored over there. And IDB1 is the one

where you have the whole image, and there are WBCs present at a certain location. So, since the whole concept of training these kind of activation maps was to train a classification network, and then use this classification network subsequently to do a localization problem. So, we are going to use the data from ALL IDB2. Now, the first point which we try to do over here is, create down just a basic scratch tensor over there.

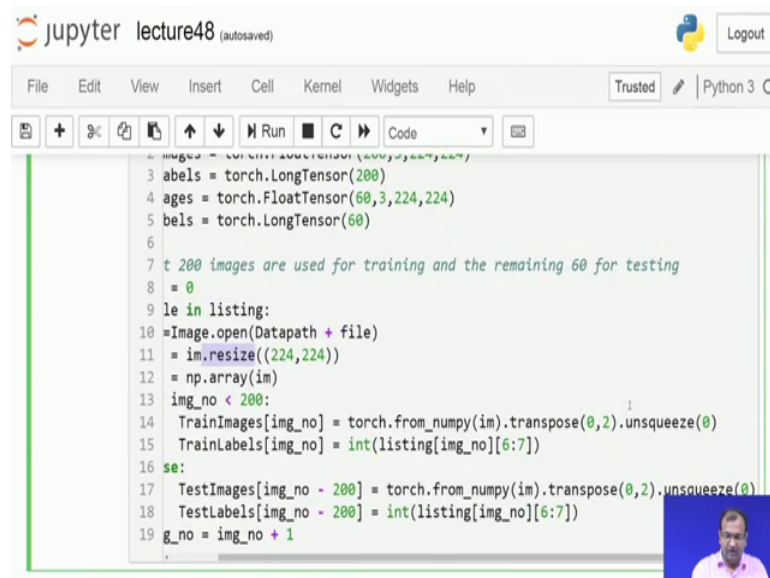
(Refer Slide Time: 03:51)



```
1 TrainImages = torch.FloatTensor(200,3,224,224)
2 TrainLabels = torch.LongTensor(200)
3 TrainImages = torch.FloatTensor(200)
4 TestImages = torch.FloatTensor(60,3,224,224)
5 TestLabels = torch.LongTensor(60)
6
7 # First 200 images are used for training and the remaining 60 for testing
8 img_no = 0
9 for file in listing:
10     im=Image.open(Datapath + file)
11     im = im.resize((224,224))
12     im = np.array(im)
13     if img_no < 200:
14         TrainImages[img_no] = torch.from_numpy(im).transpose(0,2).unsqueeze(0)
15         TrainLabels[img_no] = int(listing[img_no][6:7])
16     else:
17         TestImages[img_no - 200] = torch.from_numpy(im).transpose(0,2).unsqueeze(0)
18         TestLabels[img_no - 200] = int(listing[img_no][6:7])
19     img_no = img_no + 1
```

And now once that tensor is created out. Next is you read down one image at a time. And now once this image is read down, now since the images over there are not of a fixed size, so they keep on varying and typically they are about 256 cross 256 pixels in size or sometimes, 257 cross 257 sometimes 240 cross 240 pixels. So, they were varying out, but are the networks which we typically tend to use are the ones which would be taking down only a fixed size of image. Now, we are going to use one of the networks from our image net challenge over there, and for that reason we need to resize, all of these images on to 224 cross 224 pixels, so that is the purpose of doing a resize over here.

(Refer Slide Time: 04:36)

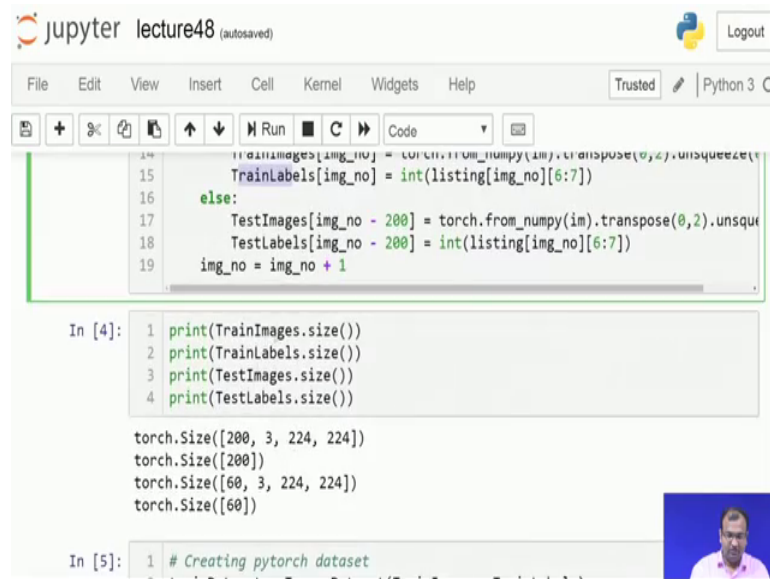


```
1 images = torch.FloatTensor(200, 3, 224, 224)
2
3 labels = torch.LongTensor(200)
4 ages = torch.FloatTensor(60, 3, 224, 224)
5 bels = torch.LongTensor(60)
6
7 # 200 images are used for training and the remaining 60 for testing
8 = 0
9
10 # in listing:
11 = Image.open(Datapath + file)
12 = im.resize((224,224))
13 = np.array(im)
14
15 img_no < 200:
16 TrainImages[img_no] = torch.from_numpy(im).transpose(0,2).unsqueeze(0)
17 TrainLabels[img_no] = int(listing[img_no][6:7])
18
19 se:
20 TestImages[img_no - 200] = torch.from_numpy(im).transpose(0,2).unsqueeze(0)
21 TestLabels[img_no - 200] = int(listing[img_no][6:7])
22
23 g_no = img_no + 1
24
25 .
```

And then what we do is, we try to just change the dimensions of the data set, so which meant that the color channel which otherwise is supposed to be on the third axis, should be coming down on the first axis, so that was our convention for using in (Refer Time: 04:48). So, that is the whole purpose of transposing 0 and 2, these two axis over there. And then you just need to convert it onto a torch array from numpy format. And this gives you, basically a 3 comma 224 comma 224 sized tensor, which can be fed down into one of these tensor locations.

Now, what we have for our case is we are going to take down, 200 images for training and 60 images for our testing and validation. So, the input tensor over there has a number of images, cross number of channels, cross x cross y dimension, so that is what you see 200 comma 3 comma 224 comma 224. Now, here we take one image at a time, resize it to 224 do the transformation get down this in 3 comma 224 comma 224 format. And then rewrite that into one of these tensile locations over there. And as you keep on changing this tensile location, you have everything filled up on the all the images over there. So, you do the same thing for your labels as well, so this is just for your training data set.

(Refer Slide Time: 05:54)



```
14 trainImages[img_no] = torch.from_numpy(im).transpose(0,2).unsqueeze(1)
15 TrainLabels[img_no] = int(listing[img_no][6:7])
16 else:
17     TestImages[img_no - 200] = torch.from_numpy(im).transpose(0,2).unsqueeze(1)
18     TestLabels[img_no - 200] = int(listing[img_no][6:7])
19     img_no = img_no + 1

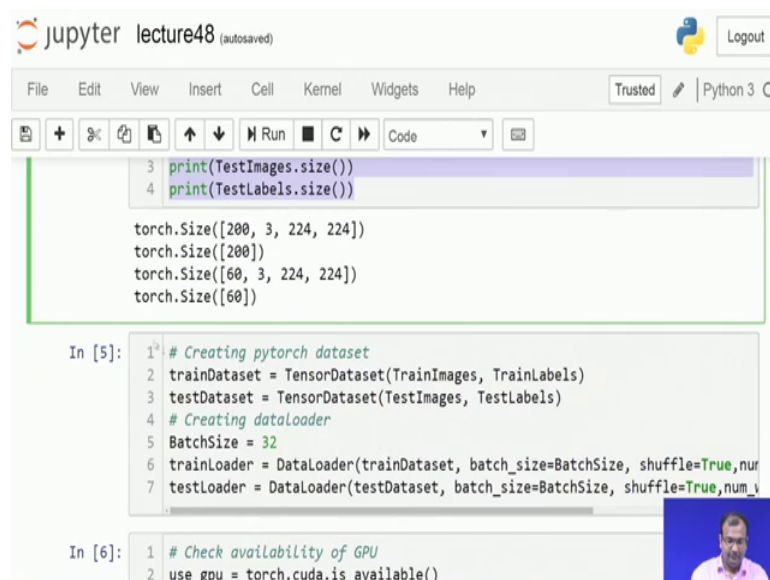
In [4]: 1 print(TrainImages.size())
2 print(TrainLabels.size())
3 print(TestImages.size())
4 print(TestLabels.size())

torch.Size([200, 3, 224, 224])
torch.Size([200])
torch.Size([60, 3, 224, 224])
torch.Size([60])

In [5]: 1 # Creating pytorch dataset
```

Now, once it is done, so let us let us try looking into you can print out the complete thing. And then you can see, what is the size of your training and testing data sets over here.

(Refer Slide Time: 06:03)



```
3 print(TestImages.size())
4 print(TestLabels.size())

torch.Size([200, 3, 224, 224])
torch.Size([200])
torch.Size([60, 3, 224, 224])
torch.Size([60])

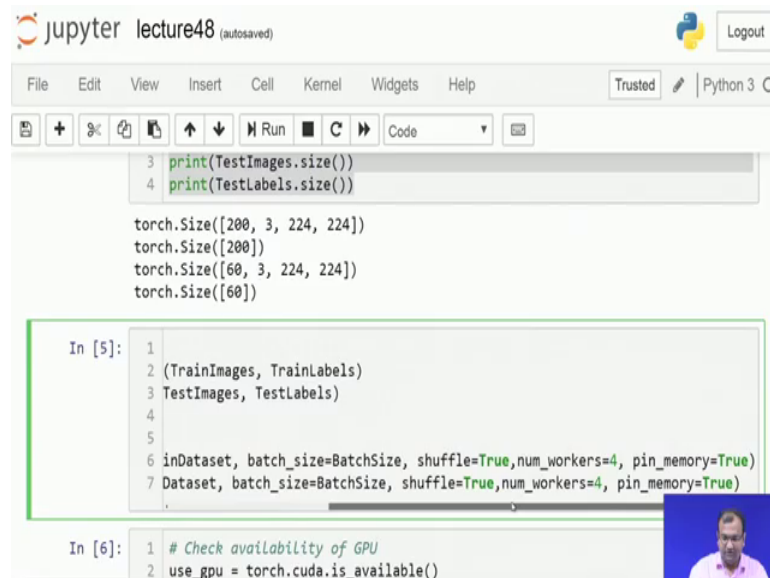
In [5]: 1 # Creating pytorch dataset
2 trainDataset = TensorDataset(TrainImages, TrainLabels)
3 testDataset = TensorDataset(TestImages, TestLabels)
4 # Creating dataLoader
5 BatchSize = 32
6 trainLoader = DataLoader(trainDataset, batch_size=BatchSize, shuffle=True, num_workers=4)
7 testLoader = DataLoader(testDataset, batch_size=BatchSize, shuffle=True, num_workers=4)

In [6]: 1 # Check availability of GPU
2 use_gpu = torch.cuda.is_available()
```

Now, that your data set is created over there. So, you can actually transfer all of this onto pi torch equivalent data set, and what that helps you is that now you can use your data loader functions in order to load it down in terms of your batches. And we define our batch size as just 32 images on our batch. So, just to stay conformal to what we had done

in the earlier cases. And then you have your data set given down, whether you are going to shuffle it across over there, so whether your stochastic nature for gradient descent to come into play the number of parallel workers over there, and these are the stuff which you keep on set.

(Refer Slide Time: 06:29)



```
jupyter lecture48 (autosaved)
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3
3 print(TestImages.size())
4 print(TestLabels.size())

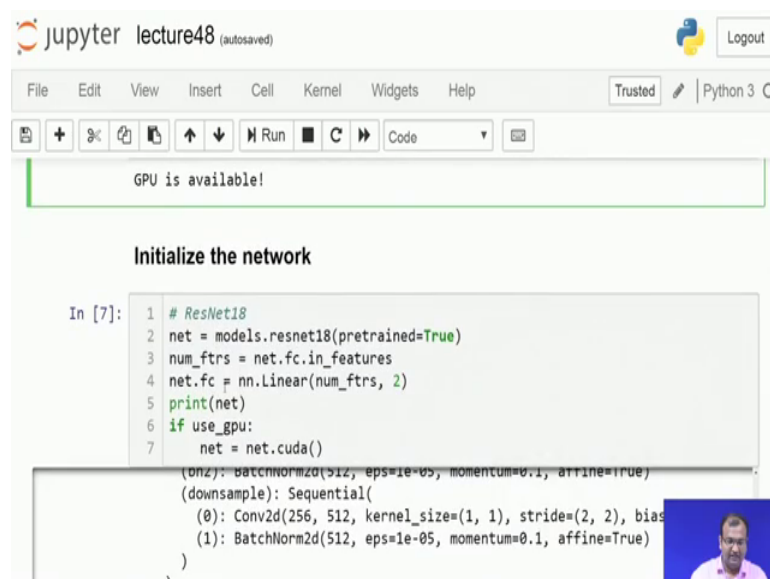
torch.Size([200, 3, 224, 224])
torch.Size([200])
torch.Size([60, 3, 224, 224])
torch.Size([60])

In [5]:
1
2 (TrainImages, TrainLabels)
3 TestImages, TestLabels)
4
5
6 inDataset, batch_size=BatchSize, shuffle=True,num_workers=4, pin_memory=True)
7 Dataset, batch_size=BatchSize, shuffle=True,num_workers=4, pin_memory=True)

In [6]:
1 # Check availability of GPU
2 use_gpu = torch.cuda.is_available()
```

Now, next what we do is we look into whether a cuda base GPU is available over there. So, since your GPU is there, so you just tag off that flag.

(Refer Slide Time: 06:48)



```
jupyter lecture48 (autosaved)
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3
GPU is available!

Initialize the network

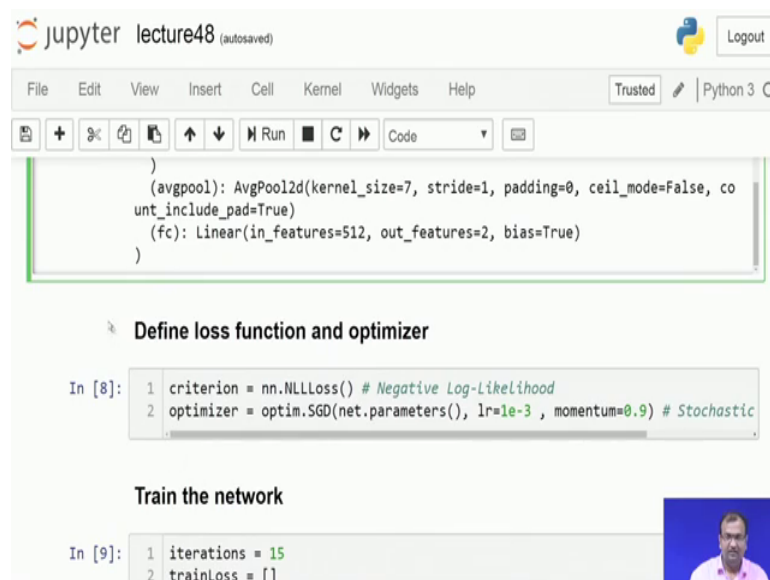
In [7]:
1 # ResNet18
2 net = models.resnet18(pretrained=True)
3 num_fters = net.fc.in_features
4 net.fc = nn.Linear(num_fters, 2)
5 print(net)
6 if use_gpu:
7     net = net.cuda()

(torch.nn.modules.batchnorm._BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True)
(downsample): Sequential(
  (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
  (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True)
)
```

And then you start with your network. Now, we start with resnet18 and we are choosing a pre-trained model; so the model which is already pre-trained on image net. Now, where we need to make a change is that image net classifying model is something which has to classify 10,000 object categories. So, you will have 10,000 neurons over there or 1,000 object categories 1,000 neurons. So, it is it is more over there.

Now, what we need to effectively changes we just have two classes of classification benign versus malignant and nothing else over there. So, we change that and make it down just as 2 neurons. So, this is the first part of the architectural adaptation, which we need to do; then get it converted onto your cuda. So, this is the only part with changes within a pre trained network and the last classification part over there.

(Refer Slide Time: 07:44)



```
jupyter lecture48 (autosaved)
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3
+ %< > Run C Code
)
(avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0, ceil_mode=False, co
unt_include_pad=True)
(fc): Linear(in_features=512, out_features=2, bias=True)
)

Define loss function and optimizer

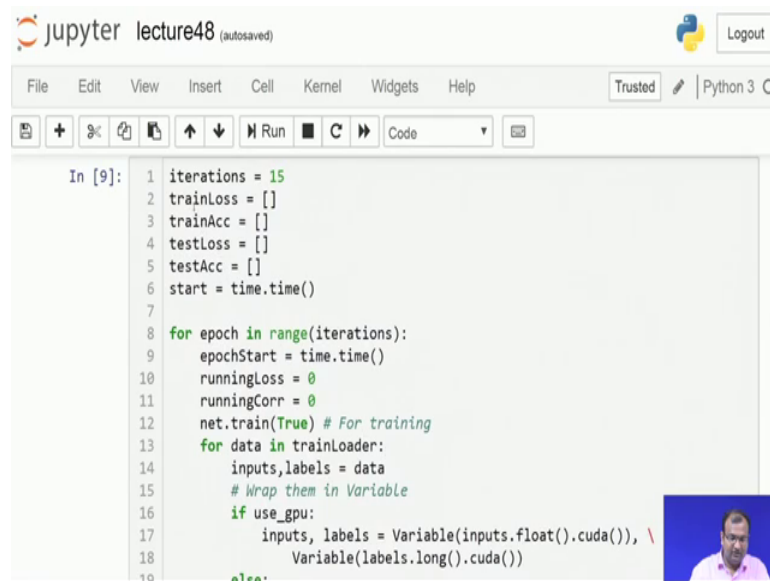
In [8]: 1 criterion = nn.NLLLoss() # Negative Log-Likelihood
2 optimizer = optim.SGD(net.parameters(), lr=1e-3, momentum=0.9) # Stochastic

Train the network

In [9]: 1 iterations = 15
2 trainLoss = []
```

Now, till now we have not yet come down into any aspect of activation pooling. And this was just to print the network, and you can pretty much see what is present over there. So, this is your resnet 18 the standard resnet 18 network without any kind of a difference coming down. Next is since this network is first trained for a classification problem. So, we need to take a cost function, which is conformal to classification problem solving; and for that purpose we choose a negative log likelihood cost function over there. And the optimizer which we choose is stochastic gradient descent.

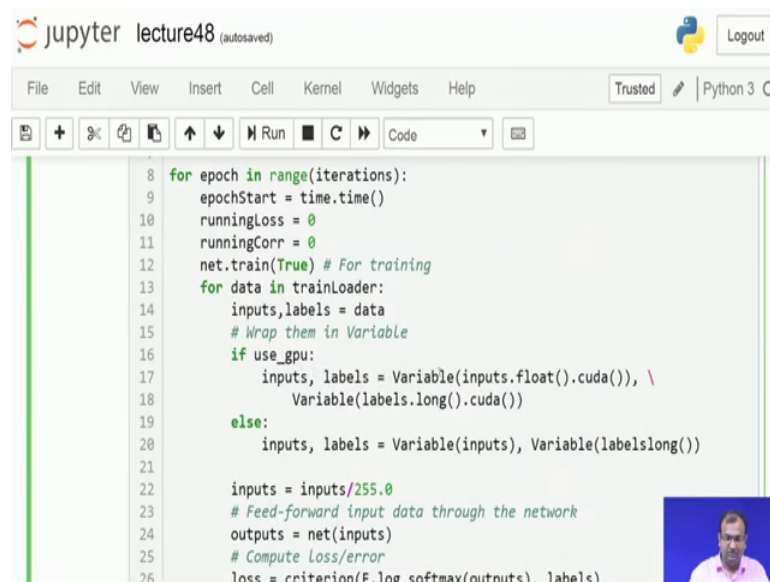
(Refer Slide Time: 08:08)



```
jupyter lecture48 (autosaved) Python 3 O
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 O
In [9]: 1 iterations = 15
2 trainLoss = []
3 trainAcc = []
4 testLoss = []
5 testAcc = []
6 start = time.time()
7
8 for epoch in range(iterations):
9     epochStart = time.time()
10    runningLoss = 0
11    runningCorr = 0
12    net.train(True) # For training
13    for data in trainLoader:
14        inputs, labels = data
15        # Wrap them in Variable
16        if use_gpu:
17            inputs, labels = Variable(inputs.float().cuda()), \
18                Variable(labels.long().cuda())
19    else:
```

Now, once everything is set over there, we decide to run this one for 15 iterations or epochs over that, now within each epoch it is it looks the same.

(Refer Slide Time: 08:16)

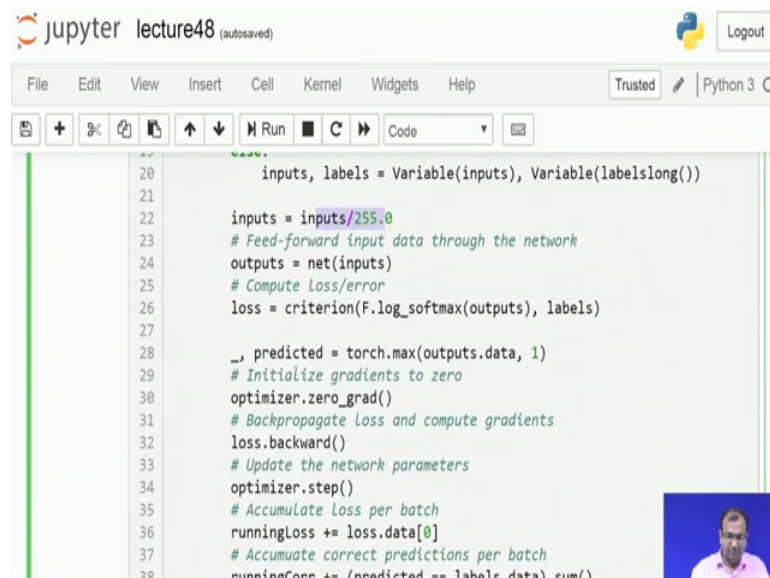


```
jupyter lecture48 (autosaved) Python 3 O
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 O
8 for epoch in range(iterations):
9     epochStart = time.time()
10    runningLoss = 0
11    runningCorr = 0
12    net.train(True) # For training
13    for data in trainLoader:
14        inputs, labels = data
15        # Wrap them in Variable
16        if use_gpu:
17            inputs, labels = Variable(inputs.float().cuda()), \
18                Variable(labels.long().cuda())
19    else:
20        inputs, labels = Variable(inputs), Variable(labels.long())
21
22    inputs = inputs/255.0
23    # Feed-forward input data through the network
24    outputs = net(inputs)
25    # Compute Loss/error
26    loss = criterion(F.log_softmax(outputs), labels)
```

So, there is not much of a difference which we are doing over here in terms of code. So, first is you need to convert your inputs to variables the variable container for use within auto grad features. And then once you have that one converted, next this is a normalization, because your inputs were basically in unsigned 8 bit integer. So, the dynamic range over there was 0 to 255 whereas, whatever inputs we are supposed to get

down to this network is supposed to be in the range of 0 to 1. So, this normalization comes off a huge help over there.

(Refer Slide Time: 08:45)

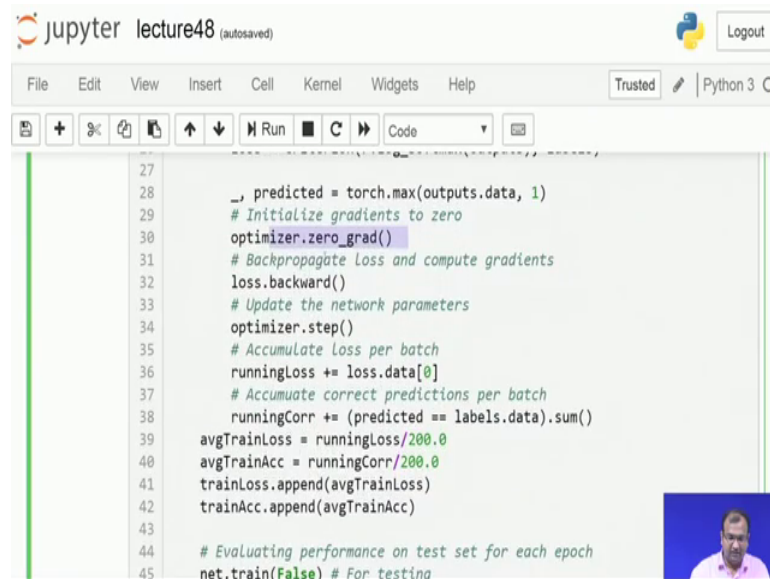


```
jupyter lecture48 (autosaved) Python 3 O
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 O
Run Code
20 inputs, labels = Variable(inputs), Variable(labelslong())
21
22 inputs = inputs/255.0
23 # Feed-forward input data through the network
24 outputs = net(inputs)
25 # Compute Loss/error
26 loss = criterion(F.log_softmax(outputs), labels)
27
28 _, predicted = torch.max(outputs.data, 1)
29 # Initialize gradients to zero
30 optimizer.zero_grad()
31 # Backpropagate loss and compute gradients
32 loss.backward()
33 # Update the network parameters
34 optimizer.step()
35 # Accumulate loss per batch
36 runningLoss += loss.data[0]
37 # Accumulate correct predictions per batch
38 runningCorr += (predicted == labels.data).sum()
```

Next you; so next what you are going to do is basically you do a feed forward over the network. Then find out your loss and that would basically be via forward propagation over the criterion. And now since you are going to use a negative log likelihood as a cost function. So, you need to have a log softmax on your output side as well. So, that is this extra transformation which comes down, for matching down the for matching down the input parameter space, conformal to your criterion function.

Now, that you get your loss coming out over there. So, you can find out which is your maximum, which is your predicted class coming out over there. And based on that, you have your optimizer set down ok.

(Refer Slide Time: 09:25)

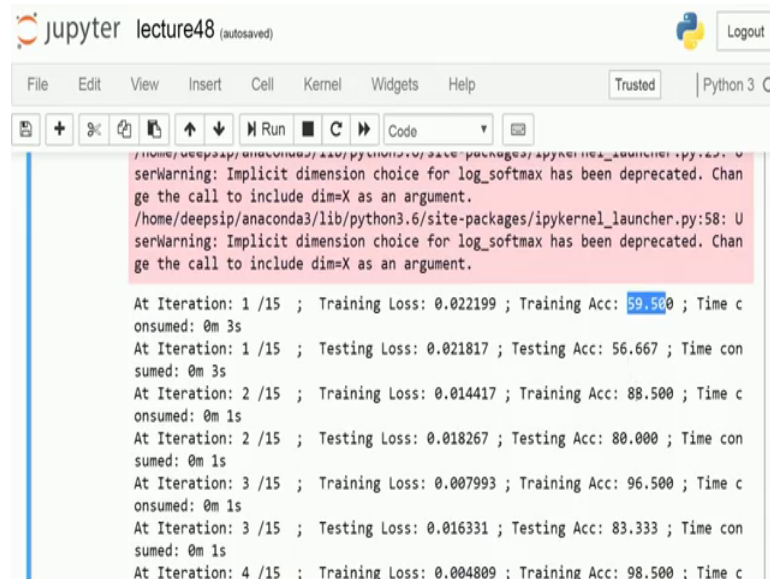


```
27
28     _, predicted = torch.max(outputs.data, 1)
29     # Initialize gradients to zero
30     optimizer.zero_grad()
31     # Backpropagate Loss and compute gradients
32     loss.backward()
33     # Update the network parameters
34     optimizer.step()
35     # Accumulate Loss per batch
36     runningLoss += loss.data[0]
37     # Accumulate correct predictions per batch
38     runningCorr += (predicted == labels.data).sum()
39     avgTrainLoss = runningLoss/200.0
40     avgTrainAcc = runningCorr/200.0
41     trainLoss.append(avgTrainLoss)
42     trainAcc.append(avgTrainAcc)
43
44     # Evaluating performance on test set for each epoch
45     net.train(False) # For testing
```

Now, once your optimizer is starting down with the zeroing down on gradient. So, the first part is you need to find out your nabla of loss or the derivative of loss that solved out. Then, you update your parameters over there.

Now, once that is done you keep on accumulating your losses over, as it keeps on performing as well as your accuracy of correct predictions as well, and then you store this data over there. Now, the next is what we need to find out is our testing. So, this was one part of my training epoch in which, I do a feed forward then my losses and then I back propagate it over there, and then find out that what is the networks current state, after this first update which has happened out. So, that is what we do with this testing data set over here, and that is also pretty simple except for the fact that, I do not have any further back propagation operation going down on my testing dataset. It is just the forward prop which takes place and you have your errors and predictions coming out over there.

(Refer Slide Time: 10:32)

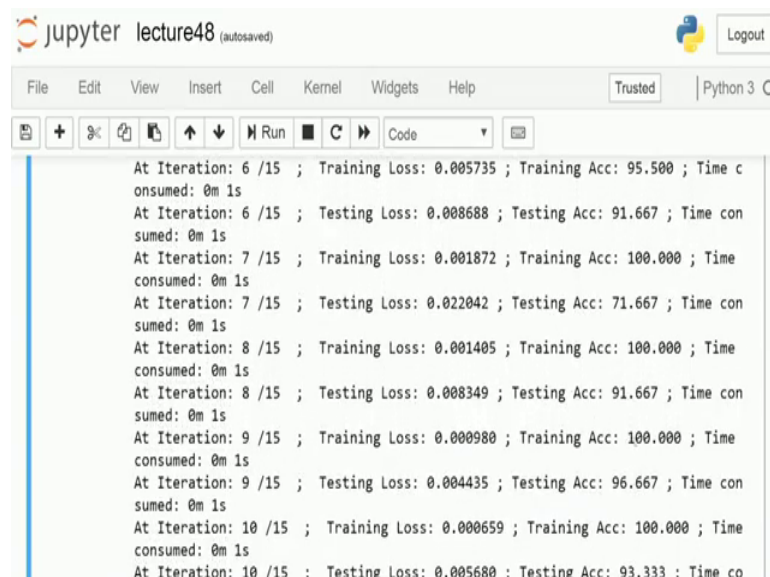


```
Warning: Implicit dimension choice for log_softmax has been deprecated. Change the call to include dim=X as an argument.
Warning: Implicit dimension choice for log_softmax has been deprecated. Change the call to include dim=X as an argument.

At Iteration: 1 /15 ; Training Loss: 0.022199 ; Training Acc: 59.560 ; Time consumed: 0m 3s
At Iteration: 1 /15 ; Testing Loss: 0.021817 ; Testing Acc: 56.667 ; Time consumed: 0m 3s
At Iteration: 2 /15 ; Training Loss: 0.014417 ; Training Acc: 88.500 ; Time consumed: 0m 1s
At Iteration: 2 /15 ; Testing Loss: 0.018267 ; Testing Acc: 80.000 ; Time consumed: 0m 1s
At Iteration: 3 /15 ; Training Loss: 0.007993 ; Training Acc: 96.500 ; Time consumed: 0m 1s
At Iteration: 3 /15 ; Testing Loss: 0.016331 ; Testing Acc: 83.333 ; Time consumed: 0m 1s
At Iteration: 4 /15 ; Training Loss: 0.004809 ; Training Acc: 98.500 ; Time c
```

Now, from there we enter into this is just a simple plotting, routine which we had followed now. So, let us look at what it comes down to, so you start down with your first iteration, and the training loss initially is about 0.22; and then you see this loss keeps on going down. And your testing accuracy over there initially starts with about 59.5 percent and then it keeps on increasing. So, around the second epoch it is already at 88.5 percent.

(Refer Slide Time: 10:54)

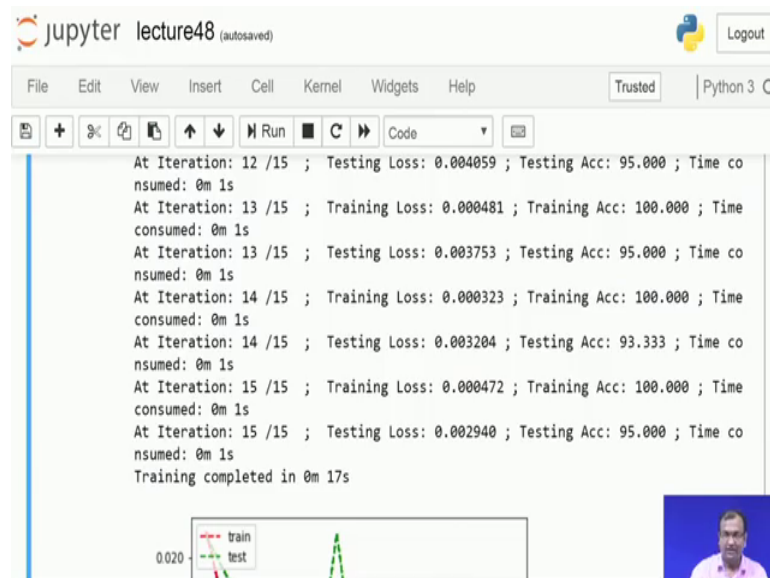


```
At Iteration: 6 /15 ; Training Loss: 0.005735 ; Training Acc: 95.500 ; Time consumed: 0m 1s
At Iteration: 6 /15 ; Testing Loss: 0.008688 ; Testing Acc: 91.667 ; Time consumed: 0m 1s
At Iteration: 7 /15 ; Training Loss: 0.001872 ; Training Acc: 100.000 ; Time consumed: 0m 1s
At Iteration: 7 /15 ; Testing Loss: 0.022042 ; Testing Acc: 71.667 ; Time consumed: 0m 1s
At Iteration: 8 /15 ; Training Loss: 0.001405 ; Training Acc: 100.000 ; Time consumed: 0m 1s
At Iteration: 8 /15 ; Testing Loss: 0.008349 ; Testing Acc: 91.667 ; Time consumed: 0m 1s
At Iteration: 9 /15 ; Training Loss: 0.000980 ; Training Acc: 100.000 ; Time consumed: 0m 1s
At Iteration: 9 /15 ; Testing Loss: 0.004435 ; Testing Acc: 96.667 ; Time consumed: 0m 1s
At Iteration: 10 /15 ; Training Loss: 0.000659 ; Training Acc: 100.000 ; Time consumed: 0m 1s
At Iteration: 10 /15 ; Testing Loss: 0.005680 ; Testing Acc: 93.333 ; Time co
```

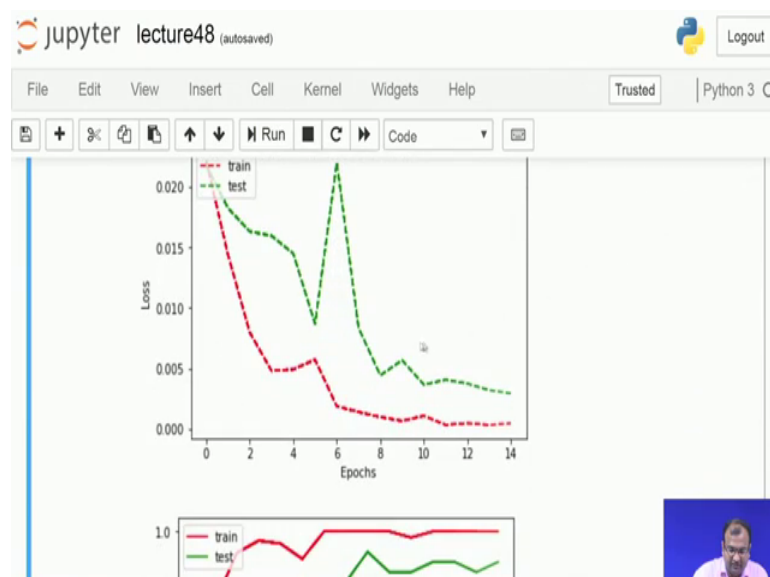
Then there is a bit of fluctuations, and then it keeps on going down to a steady point; where sometimes it even touches down 100 percent accuracy in classification. However,

you need to keep in mind that the training data is said is small; your testing data set is even smaller. it just has third 60 images present over there. So, there are chances of it getting over fit, but nonetheless this is something, which comes down between your 95, 200 percent of border and since you are using stochastic gradient decent. So, for that is one of the reasons why it is it is obviously, fluctuating a lot.

(Refer Slide Time: 11:12)



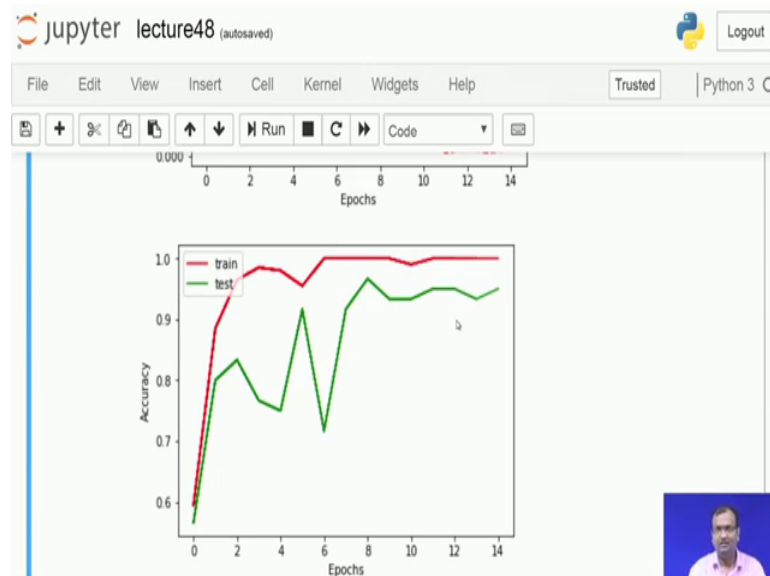
(Refer Slide Time: 11:38)



Now, a better practice is basically you can pull down your learning rates over there. And shift over from a stochastic gradient to and adam and do it. However, this was just done

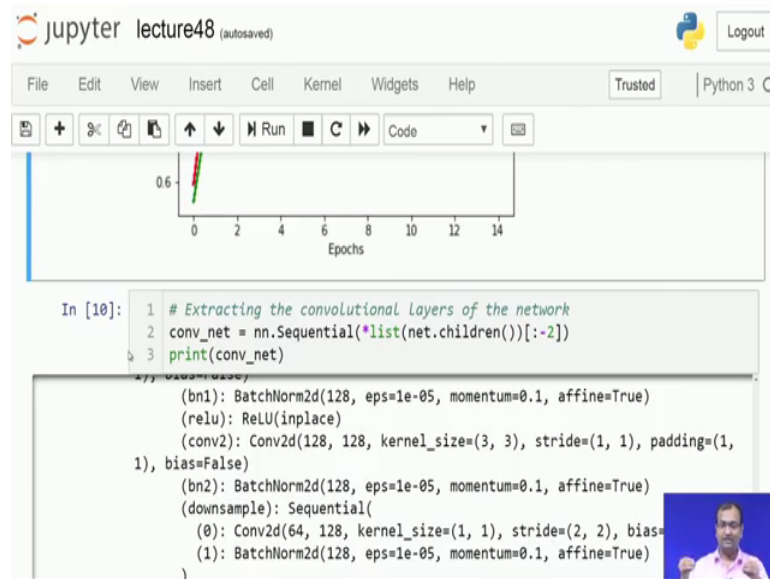
for the purpose of that stochastic gradient descent computes much faster than an adam as such.

(Refer Slide Time: 11:42)



Now, once you have that you can see your losses and then you are accuracy on training and testing, and this is our standard way of looking at it. But the interesting part comes after this, because once I have trained this network over there. Now I will have to actually create out these weights for associating each channel. And then what will be the weight associated for classifying a particular output or localizing a particular category of object.

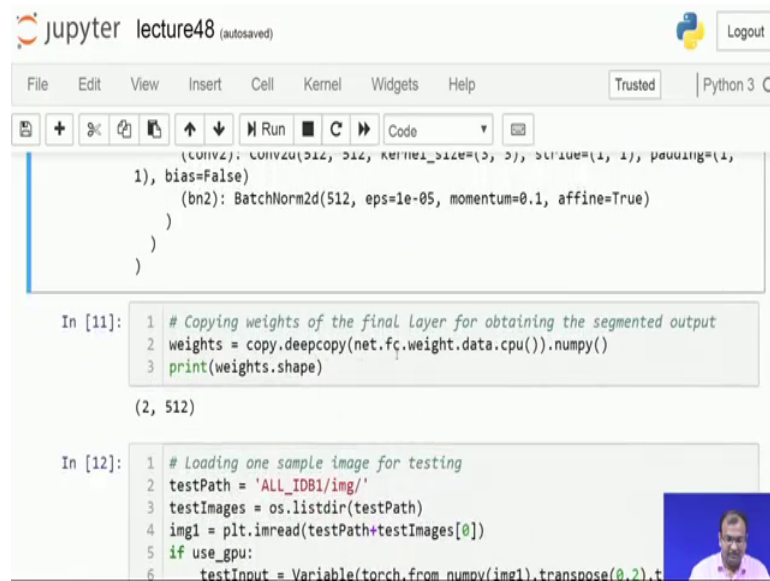
(Refer Slide Time: 12:04)



So, for that purpose what we had done in activation maps was quite simple, that we take out the last activation map over there. And then you do a global average pooling on top of that map, and this global average pooling is basically going to reduce, the last channel over there the x, y dimensions into just one single one. So, if you had I have 512 channels over there. So, in essence what I get down after this global activation pooling is 512 cross 1 size tensor over there, for each image. And then you have a fully connected layer which connects out and does the classification training once again

So, that is essentially what you would run down over there and then finally, you can see your network comes out over here.

(Refer Slide Time: 12:50)



```
lecture48 (autosaved) Python 3 O
File Edit View Insert Cell Kernel Widgets Help Trusted
(Conv2d): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
(bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True)
)
)
)

In [11]: 1 # Copying weights of the final layer for obtaining the segmented output
2 weights = copy.deepcopy(net.fc.weight.data.cpu()).numpy()
3 print(weights.shape)

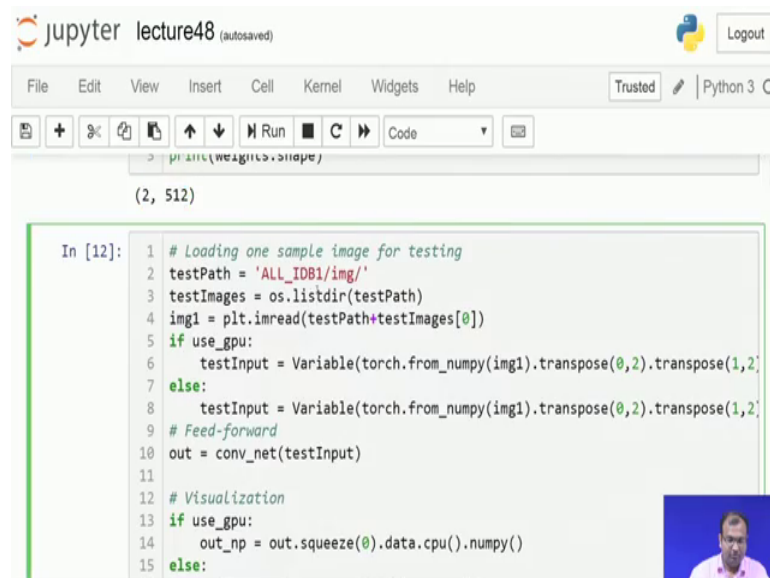
(2, 512)

In [12]: 1 # Loading one sample image for testing
2 testPath = 'ALL_IDB1/img/'
3 testImages = os.listdir(testPath)
4 img1 = plt.imread(testPath+testImages[0])
5 if use_gpu:
6 testInput = Variable(torch.from_numpy(img1).transpose(0,2), t
```

Now, once you have trained it out. The next part is basically to copy down all of these bits and just keep it for your purpose, because what you have is we had sort it down with just a two class classification problem. So, I have 512 neurons which are connected down to 2 neurons over there. So, this is going to give me a 2 cross 512 sized tensor coming out over here.

Now, there will obviously, be another bias tensor over there, but we are not interested in the bias at the current point, because that would just be giving a dc offset, and that was not part of the original formulation of using down global activation pooling for creating these kind of activation maps.

(Refer Slide Time: 13:33)



```
print(weights.shape)
(2, 512)

In [12]: 1 # Loading one sample image for testing
2 testPath = 'ALL_IDB1/img/'
3 testImages = os.listdir(testPath)
4 img1 = plt.imread(testPath+testImages[0])
5 if use_gpu:
6     testInput = Variable(torch.from_numpy(img1).transpose(0,2).transpose(1,2))
7 else:
8     testInput = Variable(torch.from_numpy(img1).transpose(0,2).transpose(1,2))
9 # Feed-forward
10 out = conv_net(testInput)
11
12 # Visualization
13 if use_gpu:
14     out_np = out.squeeze(0).data.cpu().numpy()
15 else:
```

Now, the next part is where we need to look into our, testing data from the other data set; where you have whole images. Because, here the objective is that I have my WBCs present at certain locations in my image, and I would like to find out where exactly is this WBC located. So, it is not know more about classification in any way. So, I am not going to need those smaller patches of images, but now I can give down a larger patch of image.

And on the other side over as well, since I do not have a fully connected layer going down anywhere, and they are just convolutional connections. So, I can put down basically any sized input over there, and still it can give me an output coming out. So, I do not need to restrict myself to 224 plus 224. I do not even need to restrict myself to say a square, like aspect ratio over there. I can pretty much give down a rectangular one, say 3,000 cross 2,000 pixel images has input over there.

However, one thing which we need to keep in mind is that, you cannot go much lesser than 224 cross 224, because over the subsequent convolutions which were there, say in your VGG in 19 architecture. You were doing a max pooling as well, and that is going to reduce it down. So, if you put down a 100 cross 100, somewhere in between it might just vanish out, so this is a fact which you really have to keep in mind while doing it.

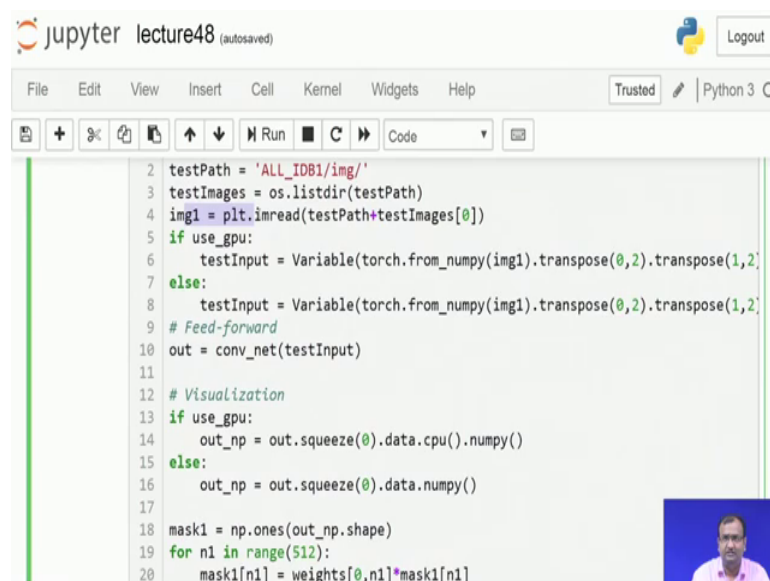
And, the other fact which you need to also keep in mind is that, if the span of my object in the original image; which I was using for classifying, if that is say 100 pixels, then on

the target one also it should be about 100 pixels. Now, suddenly on the target image if it becomes down about say 500 pixels cross 500 pixels, then you will not be getting down exact features, which will translate across the network. So, this is becomes a scale problem, and we are not training out a scale agnostic network in anywhere.

So, these are few intricate points which you need to keep in mind, otherwise while trying to do a action localization over there. And in fact, based this is one of the reasons, why activation maps do not fare that great, as compared to a region proposal network; because in a region proposal network, you do not have this kind of restriction. You can actually get down objects of different sizes and you can train with that and the region proposals, can also be generated. You do not need to stick to this fixed aspect ratio or fixed attribute sized of objects, while you are trying to train it down.

Now, here what we do is we start reading from this other database which is IDB1 which is a whole slide image, it has WBCS RBCS everything spread over there.

(Refer Slide Time: 16:02)

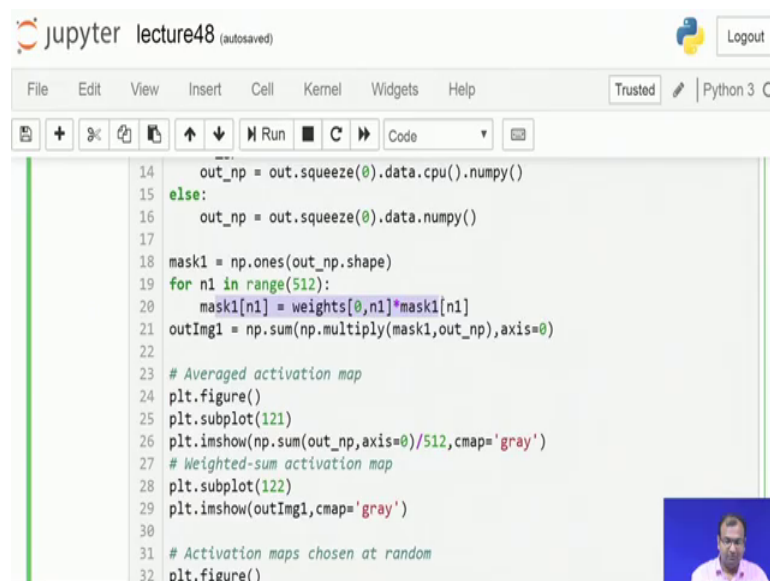


```
2 testPath = 'ALL_IDB1/img/'
3 testImages = os.listdir(testPath)
4 img1 = plt.imread(testPath+testImages[0])
5 if use_gpu:
6     testInput = Variable(torch.from_numpy(img1).transpose(0,2).transpose(1,2))
7 else:
8     testInput = Variable(torch.from_numpy(img1).transpose(0,2).transpose(1,2))
9 # Feed-forward
10 out = conv_net(testInput)
11
12 # Visualization
13 if use_gpu:
14     out_np = out.squeeze(0).data.cpu().numpy()
15 else:
16     out_np = out.squeeze(0).data.numpy()
17
18 mask1 = np.ones(out_np.shape)
19 for n1 in range(512):
20     mask1[n1] = weights[0,n1]*mask1[n1]
```

And, then we just chunk out one of these images in order to demonstrate and show it to you ok. So, this is location 0 at the first image which comes up on that tensile location over there. Now, if you have a GPU, which in our case is what is available, so we are just going to convert it into an auto grad variable. Once this typecasting is present, and you have your transpose operations appropriately done over there as well.

Next, for the visualization part over there what we need to do is, so you can do a feed forward over here. So, you get your output coming up completely. Now, if you have GPU over there, then everything is present onto a cuda kind of a construct, you need to convert it back onto a CPU, and then a retype (Refer Time: 16:44) numpy, so that is this other part of the network which you see. Whereas, if you are running it on a CPU, then you do not have this part in working out in any ways, because everything is still residing on your CPU RAM and is in nampy format Ah.

(Refer Slide Time: 17:00)



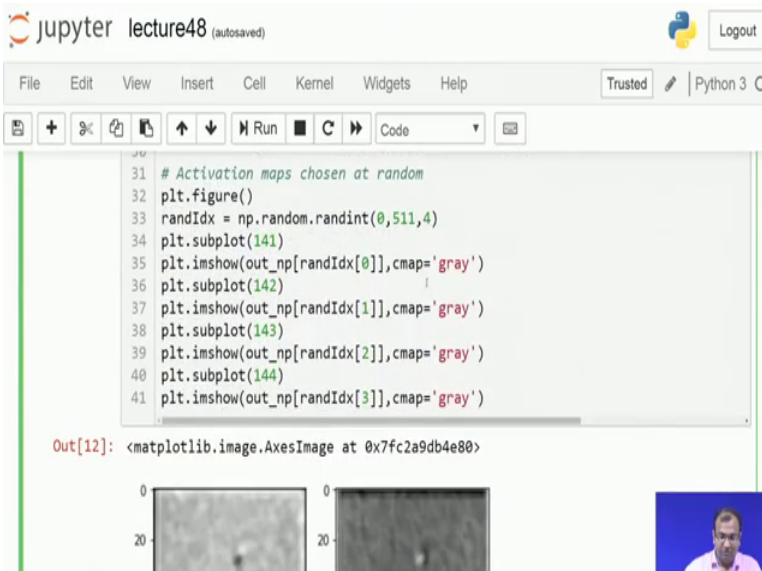
```
14 out_np = out.squeeze(0).data.cpu().numpy()
15 else:
16 out_np = out.squeeze(0).data.numpy()
17
18 mask1 = np.ones(out_np.shape)
19 for n1 in range(512):
20 mask1[n1] = weights[0,n1]*mask1[n1]
21 outImg1 = np.sum(np.multiply(mask1,out_np),axis=0)
22
23 # Averaged activation map
24 plt.figure()
25 plt.subplot(121)
26 plt.imshow(np.sum(out_np,axis=0)/512,cmap='gray')
27 # Weighted-sum activation map
28 plt.subplot(122)
29 plt.imshow(outImg1,cmap='gray')
30
31 # Activation maps chosen at random
32 plt.figure()
```

So, over here what we end up doing is create something called as a mask. So, what this mask essentially does is you are going to take a weighted summation over all the channel outputs which come down, so if I have 512 channels; then I have 512 weights associated with them. So, each channel is going to get weighed by the weights over there, and these weights are something which I get in my earlier case from here. Which is the weights which were connecting down these, 512 neurons on to the two classification neurons over there. And this 512 were, coming down as the global average pooling of the activation maps over there.

Now, so this is the second part of it, which you see over here. Now, once you get down this weighted masking coming up over there, the next part is just to take down an average along the axis along each of these channels over there or the zeroth axis or the first dimension which comes out, and then you can pretty much see it out.

Now, this first one which we see is in terms of trying to see, if we do not have a weighted averaging over there across the pixels, but we took down just a plain simple averaging which is you take all the activation maps over there sum up all the activation maps and divide by 512 or the total number of channels which comes down over there, so that is going to be a plain simple. But then I did say that you would figure out that if different activation masks will be showing out different objects in a varying proportion and not everything is going to be not every activation is something which is related down to count your particular objects which you are trying to look for, and that is one of the reasons why this mechanism will not be working out. So, instead of that here this weighted combination is something which is preferred. So, we take this weighted one and that is also shown down.

(Refer Slide Time: 18:42)



```
jupyter lecture48 (autosaved)
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3
+ %<  Run C Code
31 # Activation maps chosen at random
32 plt.figure()
33 randIdx = np.random.randint(0,511,4)
34 plt.subplot(141)
35 plt.imshow(out_np[randIdx[0]], cmap='gray')
36 plt.subplot(142)
37 plt.imshow(out_np[randIdx[1]], cmap='gray')
38 plt.subplot(143)
39 plt.imshow(out_np[randIdx[2]], cmap='gray')
40 plt.subplot(144)
41 plt.imshow(out_np[randIdx[3]], cmap='gray')
Out[12]: <matplotlib.image.AxesImage at 0x7fc2a9db4e80>
```

The screenshot shows a Jupyter Notebook window titled 'lecture48 (autosaved)'. The code cell contains Python code that randomly selects four indices from 0 to 511 and displays the corresponding activation maps as grayscale images in a 2x2 grid. The output shows two of these images, with axes labeled from 0 to 20. A small video feed of a person is visible in the bottom right corner of the notebook interface.

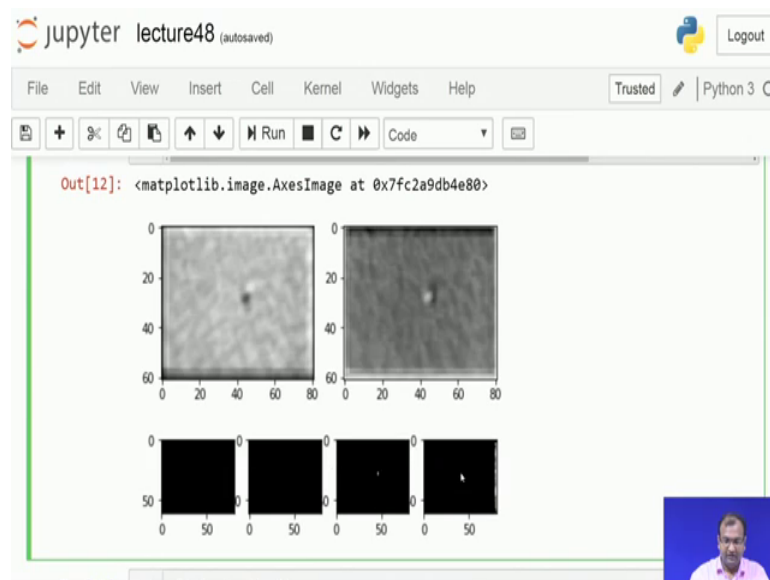
Next, what we do is we randomly pull out a few activation maps, just four activation maps over there to see exactly what comes out in them. Now, if you look over here, this first one and the second one, the first one is if you go back onto my code over here, then I can see that my first one is basically a plain simple averaging. It does not have the weighted averaging taken place.

(Refer Slide Time: 19:04)

```
jupyter lecture48 (autosaved) Python 3 O
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 O
Run Code
19 for n1 in range(512):
20     mask1[n1] = weights[0,n1]*mask1[n1]
21     outImg1 = np.sum(np.multiply(mask1,out_np),axis=0)
22
23     # Averaged activation map
24     plt.figure()
25     plt.subplot(121)
26     plt.imshow(np.sum(out_np,axis=0)/512,cmap='gray')
27     # Weighted-sum activation map
28     plt.subplot(122)
29     plt.imshow(outImg1,cmap='gray')
30
31     # Activation maps chosen at random
32     plt.figure()
33     randIdx = np.random.randint(0,511,4)
34     plt.subplot(141)
35     plt.imshow(out_np[randIdx[0]],cmap='gray')
36     plt.subplot(142)
37     plt.imshow(out_np[randIdx[1]],cmap='gray')
```

But, the second one is out Img1 is something which comes from here, and this is my weighted averaging which takes place.

(Refer Slide Time: 19:11)

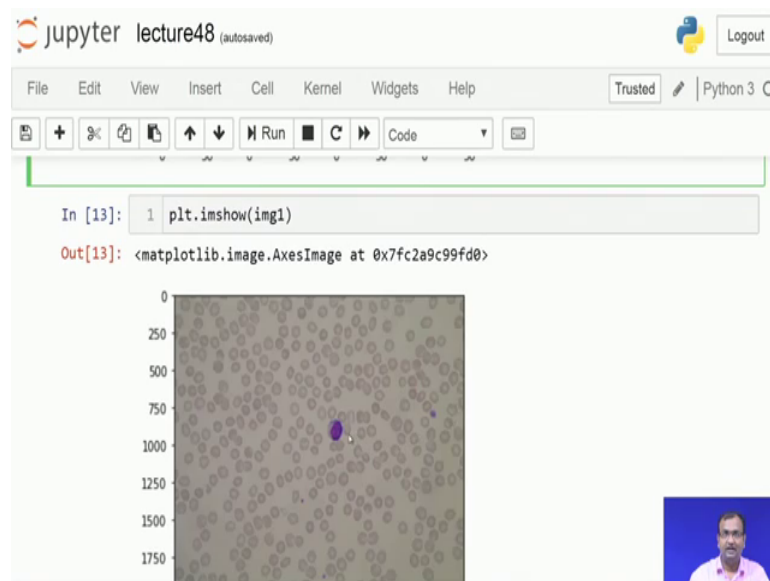


Now, on my weighted averaging, I do see a hotspot coming up over here, whereas over here, it is somewhat in the negatives, like wherever there is a high on this case I see a low over here, but that is not a very distinct case as such to come down. And our definition of what we had discussed in the last lecture, clearly said that wherever the

object is present that is which is going to show you a higher probability and we will have a higher response.

Now, if we pull out four different chunks from there, you would see in that in one of these activation channels, you have something high present somewhere near where the object was. Whereas, others were pretty much low and even this one had an activation which was almost at the periphery. So, it does not help out in any way.

(Refer Slide Time: 19:53)



Now, let us look have a look at the actual image which was present. So, this was the actual image which we were trying to look at, and this is the WBC which is present over there. And the whole objective was just to localize what where this particular WBC is present. Now, if you go back and look into our activation map, it does do a very good amount of localization. What interestingly also comes out is, that the other one which kind of negates.

Incidentally for this case I just created a negative of this map, so if you do a inversion of this map, you might end up getting the WBC itself, but then this is not always the case. In some of these cases you might find these, averaged out maps to be really erratic in the nature of their behaviors over that. They might not be having much consistent behavior whereas, if you look down onto this other network, which is a weighted combination of all the activations which come out of the channels, then you can pretty much see where

wherever this WBC is present you have a pretty distinct high probability coming out over there.

Now, this was a principle example of how to use, activation maps from any kind of a classification network; in order to initialize entry in your object location tracker over there. Now, you have many more examples which you can pretty much take up at this point of time, you can train down just a classification network on smaller patches of images, and then go down and replicate it on bigger images.

So, examples are you can train something like; alphabet detectors, or character detectors over there. And then we use this kind of a linear architecture for handwrite character detection run it over full scale images, and you can find out there is, something handwritten present on the images. Or you can create a number plate detector and then run it over images and you can find out, isolate out and really get where is the number plate present on that whole image.

So, there can be many more examples of practical problems, which we can solve it out. So, just make yourself comfortable to go around with more data, which is available out there open to use in the world, and then get it going and solved. So, that is where we come to an end for these kind of problems on activation pooling and trying to localize our objects.

In the next lecture, we are going to start with process called a semantic segmentation or simple segmentation, which is when instead of classifying one image, we are going to look into classifying each single pixel on an image and how it solved out. So, till then stay tuned and get up.

Thanks.