

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

Lecture – 50
UNet and SegNet for Semantic Segmentation

Welcome. Today, we are going to do our exercise on semantic segmentation now the network which. So, we had basically worked out on two different networks one of them was called as U-Net, and the other one was called a SegNet. Now, while in U-Net you do recall that one of the major advantages was that you had some sort of a VGG like structure of down sampling. So, there were consecutive blocks of convolution, and then you had a max pooling block which was down sampling it and then this kept on repeating over the process.

And then you could transfer all of these last layers of features and appended on the decoder pathway over there. On the other side, you had SegNet in which there was a any such sort of appending of these feature layers, but then it was again similar to a vgg like structure I said for the fact that in the decoder side you were when trying to do a max unpooling equivalent of it.

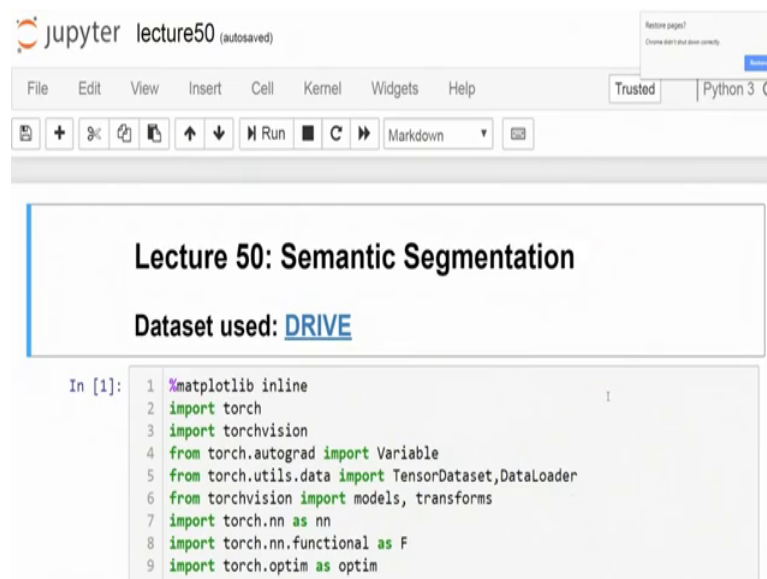
So, it was not just a simple interpolation which was has in the case of a U-Net, but here you actually could do a index passing based interpolation or which was the location from where exactly you are pooling was happening on the encoder and that was passed down onto your decoder side over there in order to get down a much higher resolution version of the image so that on one side it helped you do actually we create out high frequency components in a much better way, then you would have otherwise done when trying to use a simple nearest neighbor or a bilinear kind of an interpolation.

Now, here today what we are going to do is take up an example problem, and we are not taking up any of these key t data set kind of problems where you have hand annotated labels over camera images or there are videos over which say roads, cars, trees, footpaths, these things are annotated, but we are taking down the problem from something which we had solved out in the earlier lectures, and this is basically on vessel detection.

So, there are class of images called as retinal images and these are when you use an ophthalmoscope. So, these are predominantly medical grade images. So, an ophthalmoscope is used by a retinal physician in order to look into their retina. So, which is the photo sensory layer and then the rear part of your eye. Now, over there are multiple blood vessels which actually circulate blood through your retina and that that is the only source of your nutritioning of the retina going down.

Now, in a certain class of problems, there are new kind of diseases or a lesions or some sort of car kind of things which appear start appearing on the retina, and this can appear typically if you are diabetic or you are old age, and some of these are really age related. Now, the distance of this location of this lesion from the retina is something which is very crucial and it is typically expected that it is really far off from the retina. Now, for that you would need these vessels to be discriminated out quite perfectly. And then this whole problem which we are solving over here today is of vessel segmentation.

(Refer Slide Time: 03:08)

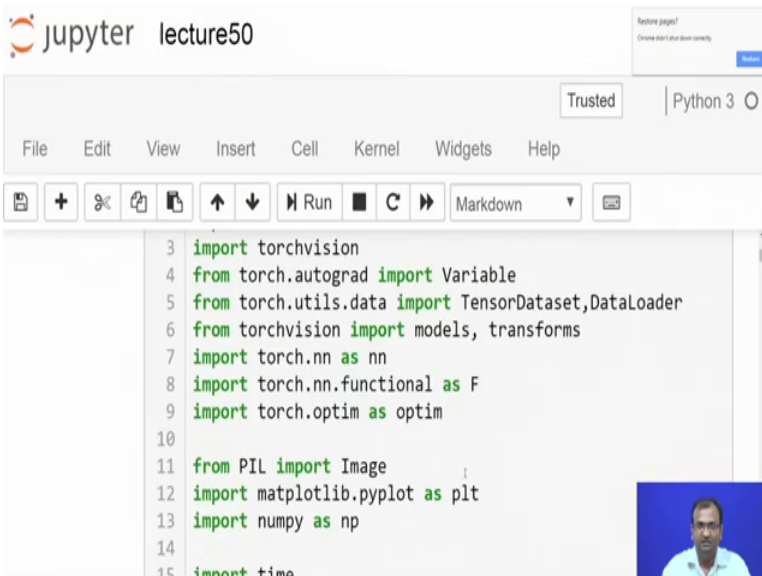


```
Jupyter lecture50 (autosaved)
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3
+ %matplotlib inline
import torch
import torchvision
from torch.autograd import Variable
from torch.utils.data import TensorDataset, DataLoader
from torchvision import models, transforms
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
```

So, we are solving out vessel segmentation in these retinal images as semantic segmentation problem. The particular data set which we are using is called as drive or the digital retinal images for vessel extraction. Now, this is an openly available data set you can just click go over there and register sign up and you would get down the data. Now, what it has is there are two different sets over there; one is called as the training set and other is called as a testing set.

In the training set, you have 20 images you have 20 images in the testing set itself. Now all the images be it on training set or on the testing set and annotated by two different radiologists; so two different ophthalmologist ophthalmic physicians, who have actually marked down where these vessels are present down and then that acts as a ground truth. Now, our purpose is that we are trying to solve it as a semantic segmentation problem. So, given RGB image using a convolutional segmentation mechanism over there, we would like to find out the different classes which belong to so each pixel belongs to which of these classes. So, there is a pix there is a class annotation coming down for each of these pixels over there.

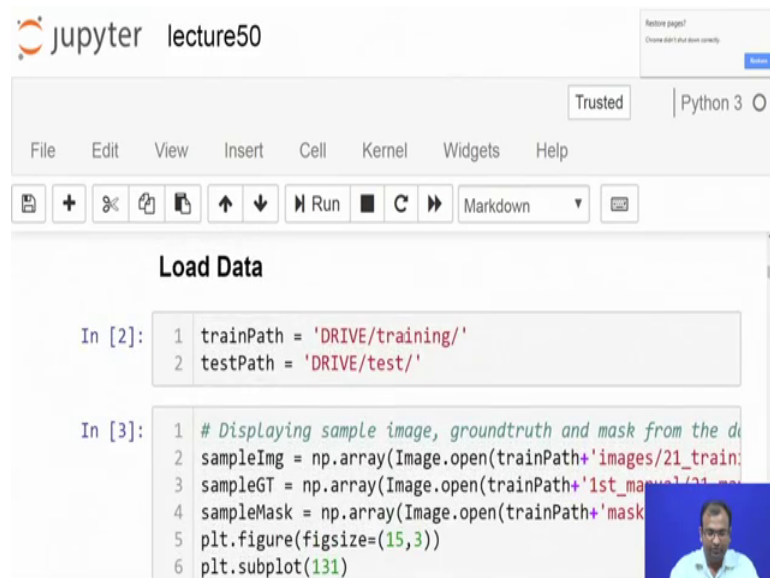
(Refer Slide Time: 04:19)



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

So, let us start and then go through what is done over there. So, the initial part is quite simple, you have your standard header which is similar to what we have been using; otherwise and then there is no extra additions which comes down over there.

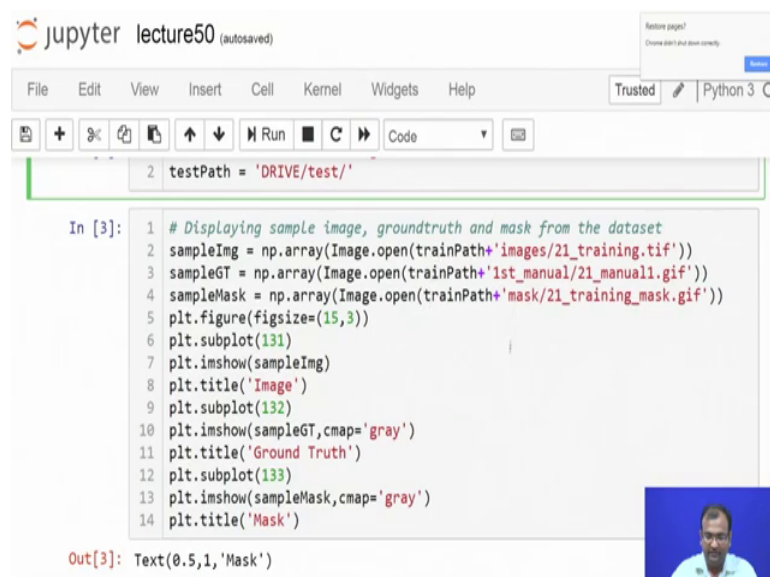
(Refer Slide Time: 04:26)



```
jupyter lecture50
Trusted Python 3
File Edit View Insert Cell Kernel Widgets Help
Load Data
In [2]: 1 trainPath = 'DRIVE/training/'
        2 testPath = 'DRIVE/test/'
In [3]: 1 # Displaying sample image, groundtruth and mask from the dataset
        2 sampleImg = np.array(Image.open(trainPath+'images/21_train.tif'))
        3 sampleGT = np.array(Image.open(trainPath+'1st_manual/21_manual1.gif'))
        4 sampleMask = np.array(Image.open(trainPath+'mask/21_training_mask.gif'))
        5 plt.figure(figsize=(15,3))
        6 plt.subplot(131)
```

Now, for your data part over there, so since we have two different data sets one is called as the training dataset and other is a test dataset. So, we make two different path allocations over there. One is for your train directory when you have to all your training images; and the other one is for your testing directly when you are testing images are present.

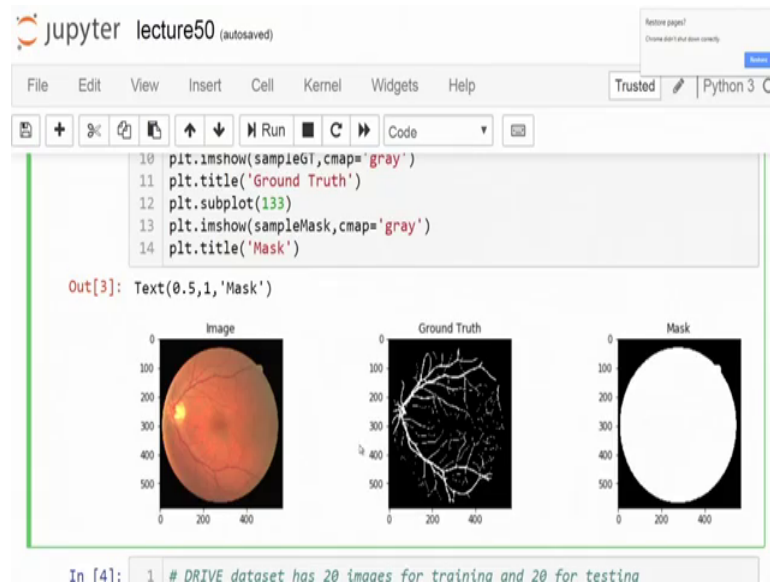
(Refer Slide Time: 04:51)



```
jupyter lecture50 (autosaved)
Trusted Python 3
File Edit View Insert Cell Kernel Widgets Help
Code
2 testPath = 'DRIVE/test/'
In [3]: 1 # Displaying sample image, groundtruth and mask from the dataset
        2 sampleImg = np.array(Image.open(trainPath+'images/21_training.tif'))
        3 sampleGT = np.array(Image.open(trainPath+'1st_manual/21_manual1.gif'))
        4 sampleMask = np.array(Image.open(trainPath+'mask/21_training_mask.gif'))
        5 plt.figure(figsize=(15,3))
        6 plt.subplot(131)
        7 plt.imshow(sampleImg)
        8 plt.title('Image')
        9 plt.subplot(132)
       10 plt.imshow(sampleGT, cmap='gray')
       11 plt.title('Ground Truth')
       12 plt.subplot(133)
       13 plt.imshow(sampleMask, cmap='gray')
       14 plt.title('Mask')
Out[3]: Text(0.5,1, 'Mask')
```

Now, once you have that one next what we end up doing is something like this that I am going to take these images from my train path and then so one simple way over here is just to show it to you.

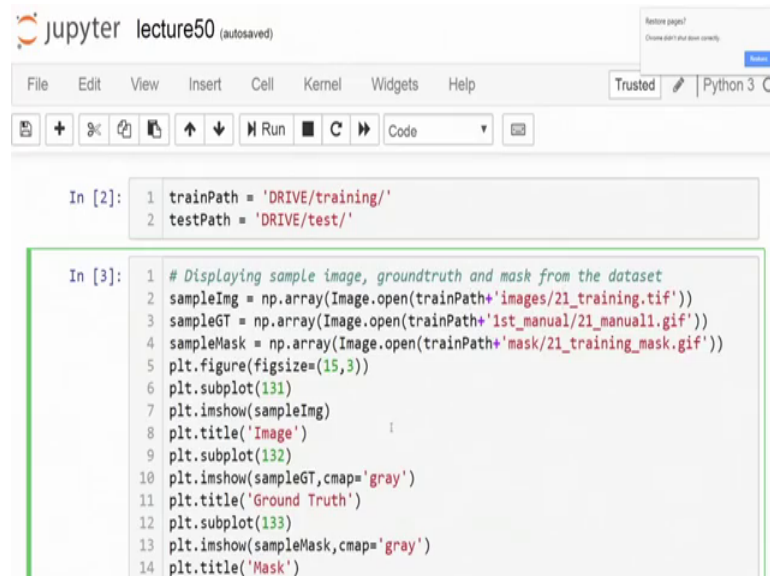
(Refer Slide Time: 04:58)



So, here what we do is say this is one of the sample images which you have on your training site. So, the RGB image standard which is given down then you have your ground truth. So, whatever is present in white is basically the vessels over there. Now, if you carefully look into this image, what you would see down is that there is a part of this image which is black, and the whole reason is because your eye circular the retina is also circular.

So, whenever it images with a circular aperture, you are going to get down a circular zone over there. Whereas, your sensors over there they are typically rectangular in shape. So, this rest of these parts which is not part of the circle is what is masked out as a black mask over there, so that is what we see in this mask as well.

(Refer Slide Time: 05:46)



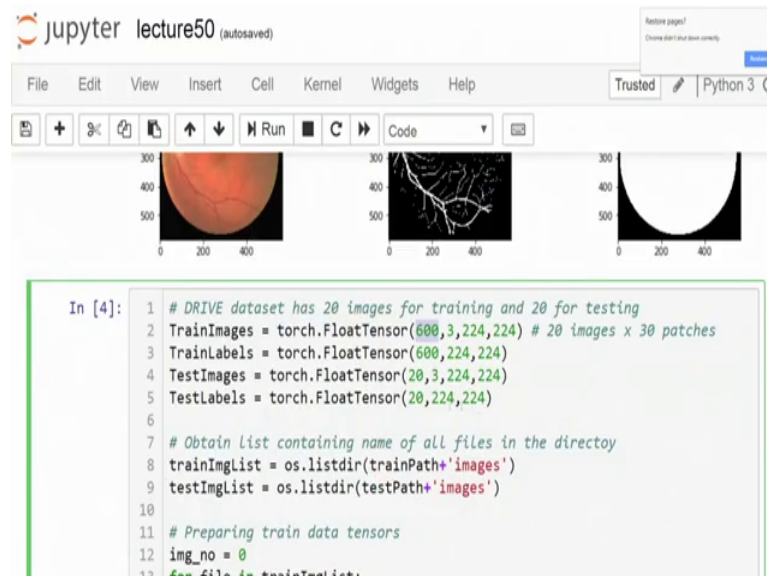
```
jupyter lecture50 (autosaved)
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3
In [2]: 1 trainPath = 'DRIVE/training/'
        2 testPath = 'DRIVE/test/'

In [3]: 1 # Displaying sample image, groundtruth and mask from the dataset
        2 sampleImg = np.array(Image.open(trainPath+'images/21_training.tif'))
        3 sampleGT = np.array(Image.open(trainPath+'1st_manual/21_manuall.gif'))
        4 sampleMask = np.array(Image.open(trainPath+'mask/21_training_mask.gif'))
        5 plt.figure(figsize=(15,3))
        6 plt.subplot(131)
        7 plt.imshow(sampleImg)
        8 plt.title('Image')
        9 plt.subplot(132)
       10 plt.imshow(sampleGT, cmap='gray')
       11 plt.title('Ground Truth')
       12 plt.subplot(133)
       13 plt.imshow(sampleMask, cmap='gray')
       14 plt.title('Mask')
```

So, what you typically get is this image plus these ground truth and the mask given down together. Now, this is from one of these images which is image number 21 from my training data set. Now, over here what we end up doing now is that we try to create 30 different manifestations of from each of these image, so that way I will be able to really polish up and get down a larger data size coming down as compared to just being limited to only 20 images because that is pretty small.

Now, for that the initial part is to define a blank tensor over there and the way we are defining it down as that since I am picking up 30 sample patches and these are not small size patches, but these are rather big size patches over there. And I am going to pick down 30 sample patches per image and there are 20 images.

(Refer Slide Time: 06:33)



```
In [4]: 1 # DRIVE dataset has 20 images for training and 20 for testing
2 TrainImages = torch.FloatTensor(600,3,224,224) # 20 images x 30 patches
3 TrainLabels = torch.FloatTensor(600,224,224)
4 TestImages = torch.FloatTensor(20,3,224,224)
5 TestLabels = torch.FloatTensor(20,224,224)
6
7 # Obtain list containing name of all files in the directory
8 trainImgList = os.listdir(trainPath+'images')
9 testImgList = os.listdir(testPath+'images')
10
11 # Preparing train data tensors
12 img_no = 0
13 for file in trainImgList:
```

So, in total I will be getting down 600 such sample patches coming to me each is an RGB image. So, you have three channels over here. And then in order to be conformal to image net like architectures, so it is not mandatory that you need to be conformal at least for semantic segmentation purposes over here. So, we are just taking down it 224 cross 224 and then sticking down over there ok.

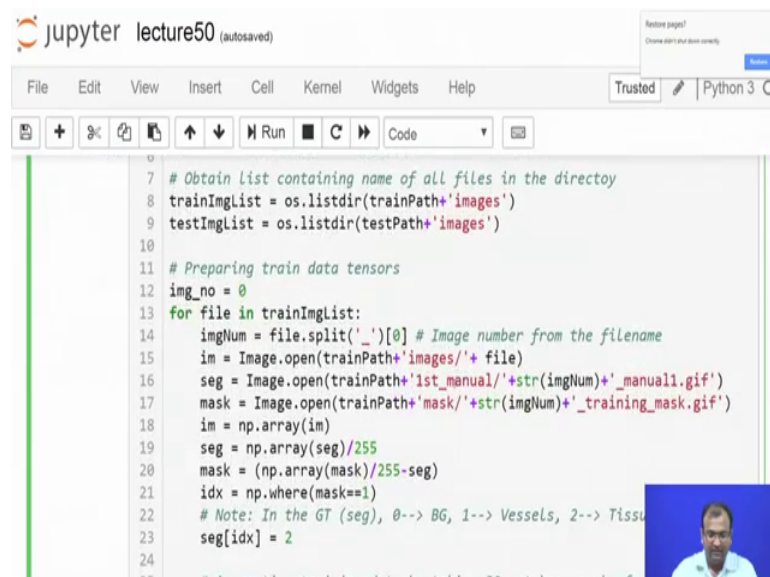
Now, on my training labels, what I have I have similar number of labels. So, I will have 600 label maps generated down with each for each of these training images which I am taking down. Now, if you look down I do not actu actually have a channel coefficient over here. Now, looking back into when we were learning down about semantic segmentation, so the whole point was that given an RGB image over there on the output side you are going to get down one channel which is corresponding to each class over there.

However, the mechanism of training this thing is either with say negative log likelihood loss or with a binary cross entropy kind of a criterias. In those cases while your network is still going to generate out these number of slices or the number of channels which correspond to each of these classes. But in your training in the data ground truth data which you are passing down for training over there that data over there is just going to have these particular channel numbers or the class number given down. So, if it is class number zero that corresponds to channel zero and that will be of high; if it is class

number one then it corresponds to channel number one which will have the highest probability over there. So, for that reason, we just pass down 3d matrix 3d tensor over there where each element of the tensile is a matrix of 224 cross 224.

Now, on my testing side over there what I choose to do is we do not take any more patching around the whole image, but we pass down the whole image as it is in a size of 224 cross 224. So, there are 20 images on my testing side and corresponding to each of them I have my testing labels also defined. This helps me in finding out my accuracy as well as my loss calculation on the validation part of my code.

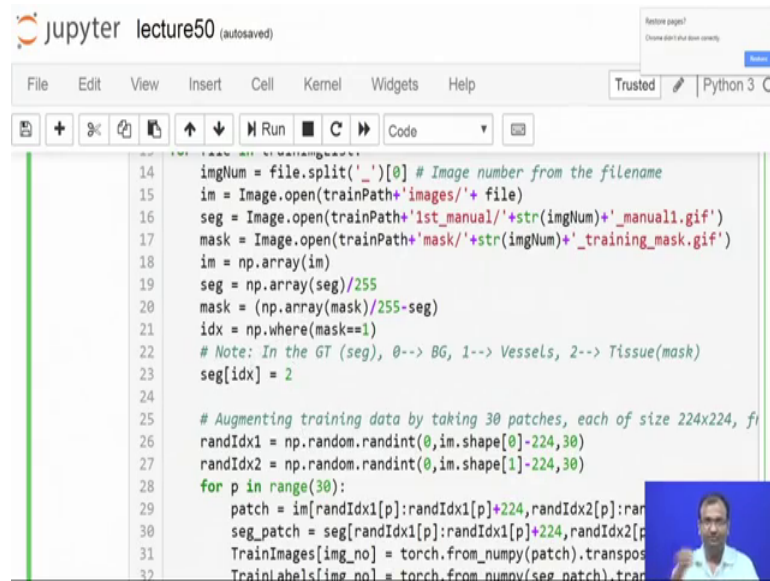
(Refer Slide Time: 08:37)



```
7 # Obtain List containing name of all files in the directoy
8 trainImgList = os.listdir(trainPath+'images')
9 testImgList = os.listdir(testPath+'images')
10
11 # Preparing train data tensors
12 img_no = 0
13 for file in trainImgList:
14     imgNum = file.split('_')[0] # Image number from the filename
15     im = Image.open(trainPath+'images/'+ file)
16     seg = Image.open(trainPath+'1st_manual/'+str(imgNum)+'_manual1.gif')
17     mask = Image.open(trainPath+'mask/'+str(imgNum)+'_training_mask.gif')
18     im = np.array(im)
19     seg = np.array(seg)/255
20     mask = (np.array(mask)/255-seg)
21     idx = np.where(mask==1)
22     # Note: In the GT (seg), 0--> BG, 1--> Vessels, 2--> Tissu
23     seg[idx] = 2
24
25 # Augmenting training data by taking 20 patches each of
```

Now, that we end up doing this one the first part is basically take open up one of these images from my training data set. Now, once I have this a image taken down over there the next part is to keep on generating these labels over there.

(Refer Slide Time: 08:53)



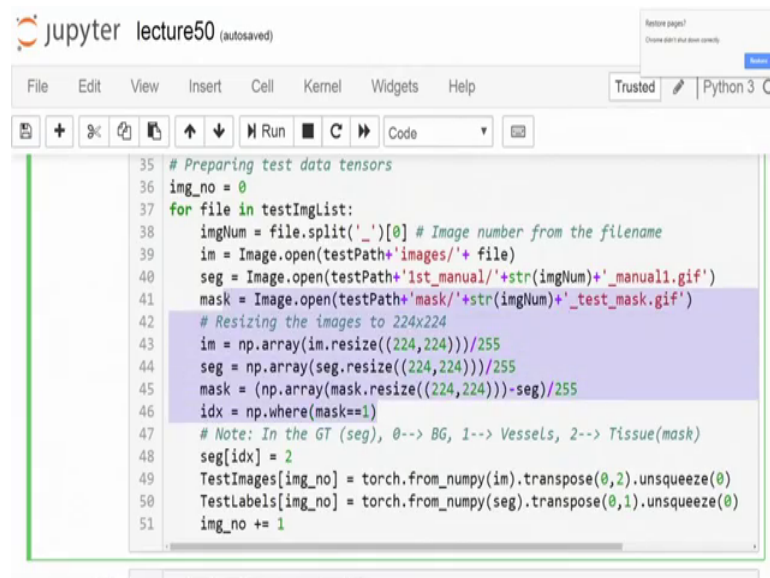
```
14 imgNum = file.split('_')[0] # Image number from the filename
15 im = Image.open(trainPath+'images/'+ file)
16 seg = Image.open(trainPath+'1st_manual/'+str(imgNum)+'_manual1.gif')
17 mask = Image.open(trainPath+'mask/'+str(imgNum)+'_training_mask.gif')
18 im = np.array(im)
19 seg = np.array(seg)/255
20 mask = (np.array(mask)/255-seg)
21 idx = np.where(mask==1)
22 # Note: In the GT (seg), 0--> BG, 1--> Vessels, 2--> Tissue(mask)
23 seg[idx] = 2
24
25 # Augmenting training data by taking 30 patches, each of size 224x224, fr
26 randIdx1 = np.random.randint(0,im.shape[0]-224,30)
27 randIdx2 = np.random.randint(0,im.shape[1]-224,30)
28 for p in range(30):
29     patch = im[randIdx1[p]:randIdx1[p]+224,randIdx2[p]:ran
30     seg_patch = seg[randIdx1[p]:randIdx1[p]+224,randIdx2[p]
31     TrainImages[img_no] = torch.from_numpy(patch).transpos
32     TrainLabels[img_no] = torch.from_numpy(seg_patch).tran
```

Now, what we end up doing is if it is it belongs to the background which is over here. So, if it is in the mask zone and if this is black, then we just retain that as class 0. If I have any of these regions present over here which are my blood vessels, then I put them as class one and if I have this background over here then I make that as class 2. So, it is 0, 1, 2 or 3 class classification problem which I have and that label is what is associated with each pixel coming down over there. So, now that is what is effectively done over here. So, you have your annotation per pixel for class levels also given now.

Now, what we do is we randomly pull out certain chunks over there to get down some random cropped out regions ok. So, this is what we initiate over here. And we choose such thirty random locations. So, what its effe essentially doing is you have a large size image, but you want to chunk out only 224 cross 224 sized blocks over there.

Now, what we are doing is we are randomly generating some x and y indices size that I can have these as random positions over there, and I can chunk out 224 cross 224 sized blocks out of it. And this just helps me augment out my data set without incorporating any without undergoing any changes sort of a bias towards a particular region or location over there.

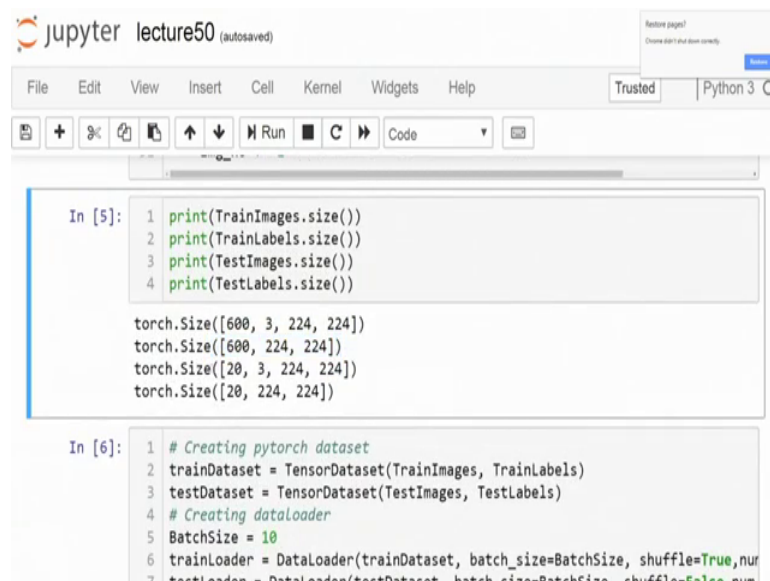
(Refer Slide Time: 10:31)



```
35 # Preparing test data tensors
36 img_no = 0
37 for file in testImgList:
38     imgNum = file.split('.')[0] # Image number from the filename
39     im = Image.open(testPath+'images/'+ file)
40     seg = Image.open(testPath+'1st_manual/'+str(imgNum)+'_manuall.gif')
41     mask = Image.open(testPath+'mask/'+str(imgNum)+'_test_mask.gif')
42     # Resizing the images to 224x224
43     im = np.array(im.resize((224,224)))/255
44     seg = np.array(seg.resize((224,224)))/255
45     mask = (np.array(mask.resize((224,224)))-seg)/255
46     idx = np.where(mask==1)
47     # Note: In the GT (seg), 0--> BG, 1--> Vessels, 2--> Tissue(mask)
48     seg[idx] = 2
49     TestImages[img_no] = torch.from_numpy(im).transpose(0,2).unsqueeze(0)
50     TestLabels[img_no] = torch.from_numpy(seg).transpose(0,1).unsqueeze(0)
51     img_no += 1
```

Now, once that is done we have it packed up in our training data set format and then. So, we take up these training data over there, and then this is the packing which keeps on going down over there. Now, once this whole packing is done, we pack down both the training data which is the image as well as the training labels over there. So, each has to be in the same format packed up.

(Refer Slide Time: 10:44)



```
In [5]: 1 print(TrainImages.size())
        2 print(TrainLabels.size())
        3 print(TestImages.size())
        4 print(TestLabels.size())

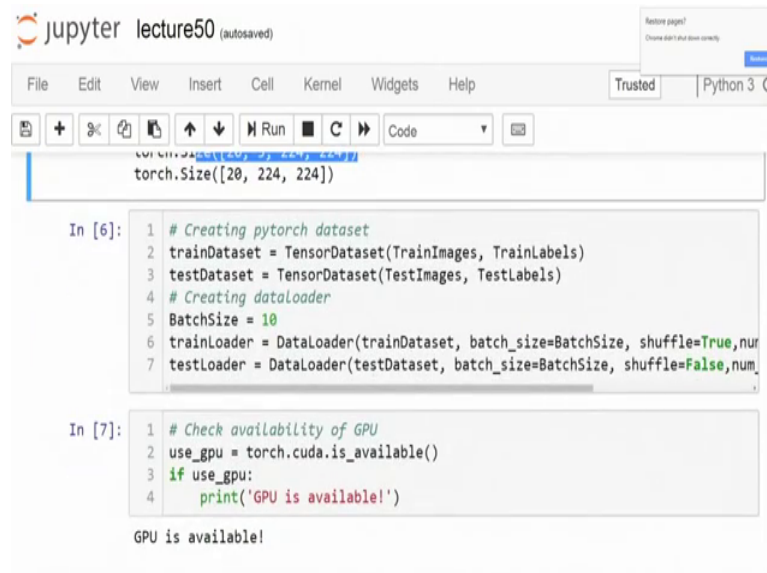
torch.Size([600, 3, 224, 224])
torch.Size([600, 224, 224])
torch.Size([20, 3, 224, 224])
torch.Size([20, 224, 224])

In [6]: 1 # Creating pytorch dataset
        2 trainDataset = TensorDataset(TrainImages, TrainLabels)
        3 testDataset = TensorDataset(TestImages, TestLabels)
        4 # Creating dataLoader
        5 batchSize = 10
        6 trainLoader = DataLoader(trainDataset, batch_size=batchSize, shuffle=True, num_workers=0)
        7 testLoader = DataLoader(testDataset, batch_size=batchSize, shuffle=False, num_workers=0)
```

Now, once that packing is done, so you can have a look into what it looks like. So, you have your 600 cross 3 cross 224 cross 224 which is your input data tensor size you have

600 cross 224 cross 224 which is your class tensor size, and then you have your test data over there as well.

(Refer Slide Time: 11:02)



```
lecture50 (autosaved)
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3
torch.Size([20, 224, 224])

In [6]:
1 # Creating pytorch dataset
2 trainDataset = TensorDataset(TrainImages, TrainLabels)
3 testDataset = TensorDataset(TestImages, TestLabels)
4 # Creating dataLoader
5 BatchSize = 10
6 trainLoader = DataLoader(trainDataset, batch_size=BatchSize, shuffle=True, num_workers=4)
7 testLoader = DataLoader(testDataset, batch_size=BatchSize, shuffle=False, num_workers=4)

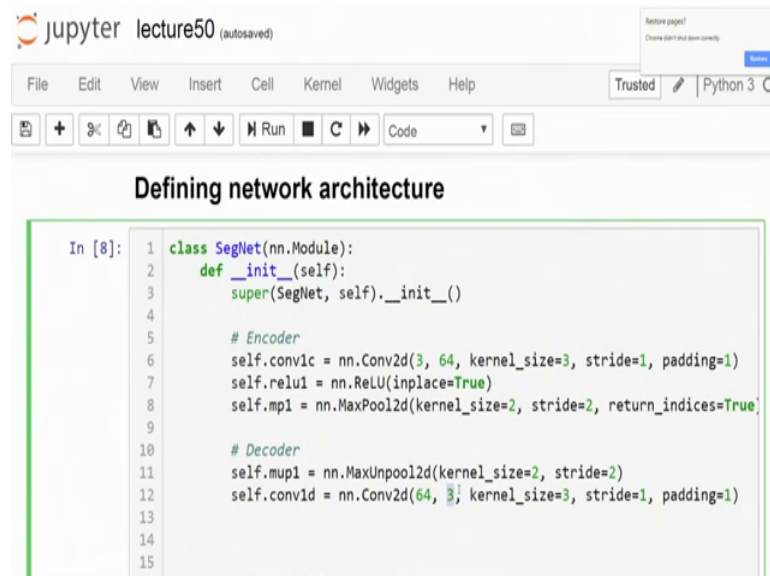
In [7]:
1 # Check availability of GPU
2 use_gpu = torch.cuda.is_available()
3 if use_gpu:
4     print('GPU is available!')

GPU is available!
```

Now, once we have this we just invoke our training and testing data loaders over there, and we choose to have a batch size of ten. Now, keep one thing in mind that here this process is obviously, memory intensive because you are not doing you do not have a neural network which is just taking an image it comes down to some linear neuron representation over there.

So, in this case because it is narrowing down in its representation form, so it is going to use lesser amount of operational memory. However, here you are almost going to continue in the same way or sometimes it goes down and then in the decoder part it again starts moving up in that way these kind of networks do require more amount of operational memory, they are more data intensive as such going down. So, the batch size is typically need to be brought down to a lesser number. Next, we check out our same old trick over there to find out if GPU is available, and if cuda is ready then we can just export out onto the GPU for faster execution.

(Refer Slide Time: 12:05)



```
In [8]: 1 class SegNet(nn.Module):
2         def __init__(self):
3             super(SegNet, self).__init__()
4
5             # Encoder
6             self.conv1c = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1)
7             self.relu1 = nn.ReLU(inplace=True)
8             self.mp1 = nn.MaxPool2d(kernel_size=2, stride=2, return_indices=True)
9
10            # Decoder
11            self.mup1 = nn.MaxUnpool2d(kernel_size=2, stride=2)
12            self.conv1d = nn.Conv2d(64, 3, kernel_size=3, stride=1, padding=1)
13
14
15
```

Now, here we start by defining our network. Now, this network which we use is not exactly the SegNet, but there is a model which is loosely inspired by SegNet and it is a much simpler model and it is a trivial model. Now, one of the reasons why we chose to go with a trivial model is that in order to one point is in order to avoid overfitting. If you have large number of parameters and lesser amount of training data, you are prone to overfit over the model.

The next important aspect which comes out over here is that the total number of classes which I am taking on the output is 3 it is a really less number of classes coming down. So, there might possibly not be too many different kind of manifestations which it has to learn up and that is one of the reasons why we are just taking down to this kind of a very simple baseline model of a segmentation convolutional semantic segmentation network. So, what we have in this network is pretty simple, I have a convolution layer where the kernel sizes are 3 cross 3, and I have 64 such kernels taken down for my work.

Now, I do convolution with the stride of one and a padding of one which necessarily means that whatever was the size of my x y dimension of my input image. After this convolution, I am going to retain the same x y dimension over there, because I am not changing down either the stride or the padding given over here. The next point is the we put in place a non-linearity, and this is a rectified linear unit which is put in place for a non-linearity to come down.

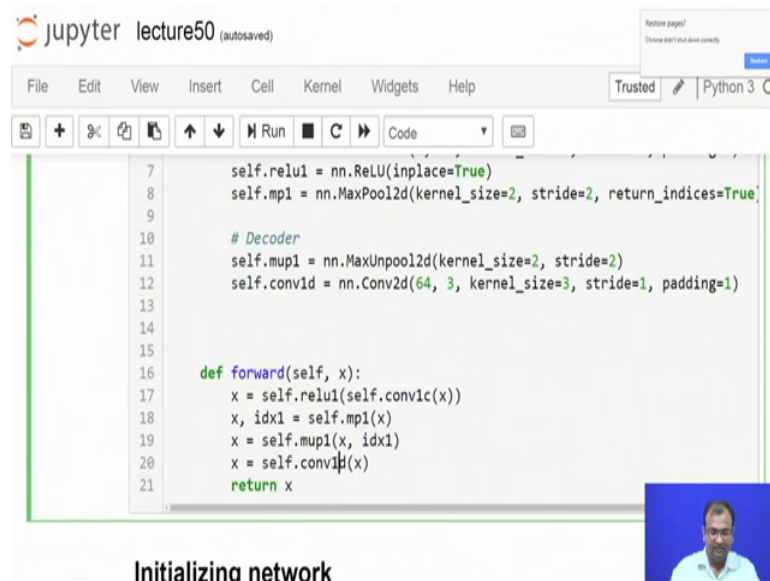
Following this, we have a max pooling of 2 cross 2 kernel size with a stride of two. So, this is a 2 cross 2 max pooling which means that the x and y dimensions are going to get reduced to half of it. Now, this part serves as the encoder part over there. Now, one thing which we do extra is that we set in this extra argument over here which is called as return indices equal to true. And one of the reasons for doing these return indices is equal to true is that we need to have a trace of where all these indices were present in order to do our unpooling. It was the logic for a SegNet like behavior was that you are not going to just do a bilinear interpolation or a nearest neighbor interpolation in order to scale it up during the unpooling there, but you will do a guided sort of a unpooling, so which meant that you needed to have an index transfer from your encoder to your decoder side as well.

Now, in your decoder side what you have is, first you have a unpooling over there which skills it up, now once you have that unpooling after that you have your convolution a 2D convolution coming down. So, this 2D convolutions job is to convert down 64 channels onto 3 channels. Now, this is three channels is not corresponding to the image channels over there, but this is because you have 3 different classes.

Now, if you took down 6 classes then you would have 6 channels, if you take down 10 classes, then you have 10 channels over here the convolutions are still with the kernel size of 3 plus 3 with a stride of one and a padding of one which meant that the same size xy size is preserve. Now, if we look into this kind of a network, so if my input is 3 cross 224 cross 224, then after this before this max pooling just after the ReLu, it is going to be 64 cross 224 cross 224.

After this max pooling, it is going to be 64 cross 112 cross 112. Now, from here when I do an max unpooling over there, so that is going to take in 64 cross 112 cross 112, and make it up to 64 cross 224 cross 224. Then I have this convolution running down and this is going to return me a 3 cross 224 cross 224 where each of these channel corresponds to one of the classes to which a pixel can belong to.

(Refer Slide Time: 15:36)



```
7 self.relu1 = nn.ReLU(inplace=True)
8 self.mp1 = nn.MaxPool2d(kernel_size=2, stride=2, return_indices=True)
9
10 # Decoder
11 self.mup1 = nn.MaxUnpool2d(kernel_size=2, stride=2)
12 self.conv1d = nn.Conv2d(64, 3, kernel_size=3, stride=1, padding=1)
13
14
15
16 def forward(self, x):
17     x = self.relu1(self.conv1d(x))
18     x, idx1 = self.mp1(x)
19     x = self.mup1(x, idx1)
20     x = self.conv1d(x)
21     return x
```

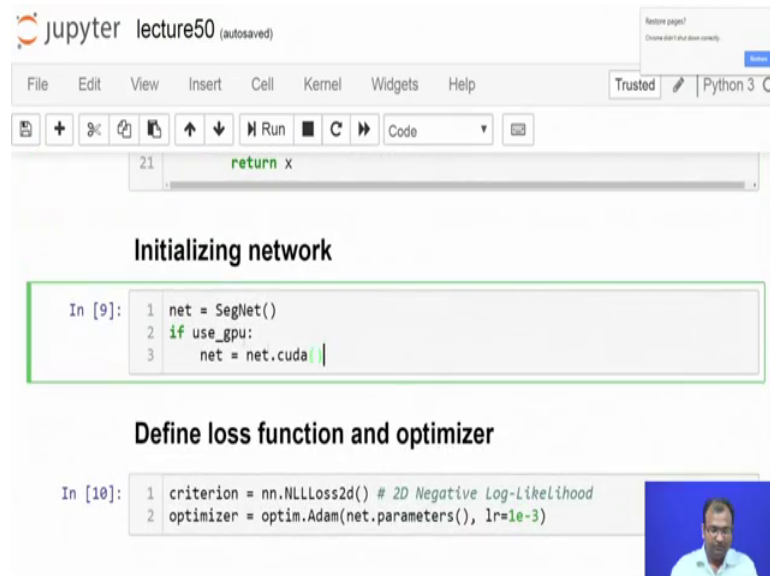
Initializing network

Next, we have the forward definition for this network return term which is plain and straight. So, what you do is you take an input over there, you pass it through the first convolution layer, then pass it through the relu after you have passed it through the relu you are going to pass it through the max pooling. On the max pooling side, we are supposed to take down these indices for the pooling indices as well, so that we can pass it down to the unpooling layer. So, that is this id x 1 which is taken up

Next when the max unpooling layer comes into play, you pass down the output from the previous layer which is x as well as these indices as well. Now, you have the unpooled version coming up over here and then you do your second convolution over here which is with the block called as conv 1 d so that is straightforward a very simple SegNet like architecture which has just one convolution layer one max pooling, one max unpooling another convolution.

Now, you can definitely make these instead of one single convolution layer you can have a band of convolution and that depends on what is the granularity of the problem you are trying to look at, and what are the different classes you are trying to solve. If it is just a plain simple three class which is like apparently visually even very distinct from each other, then it does not make sense to even go down for more number of kernels or even depth encoding of these kernels you can pretty much do it with a very shallow network as we are doing over here.

(Refer Slide Time: 16:55)



The screenshot shows a Jupyter Notebook window titled "lecture50 (autosaved)". 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 code is organized into sections:

- A code cell at the top contains the line: `return x`.
- A section titled "Initializing network" contains the following code:

```
In [9]: 1 net = SegNet()
        2 if use_gpu:
        3     net = net.cuda()
```
- A section titled "Define loss function and optimizer" contains the following code:

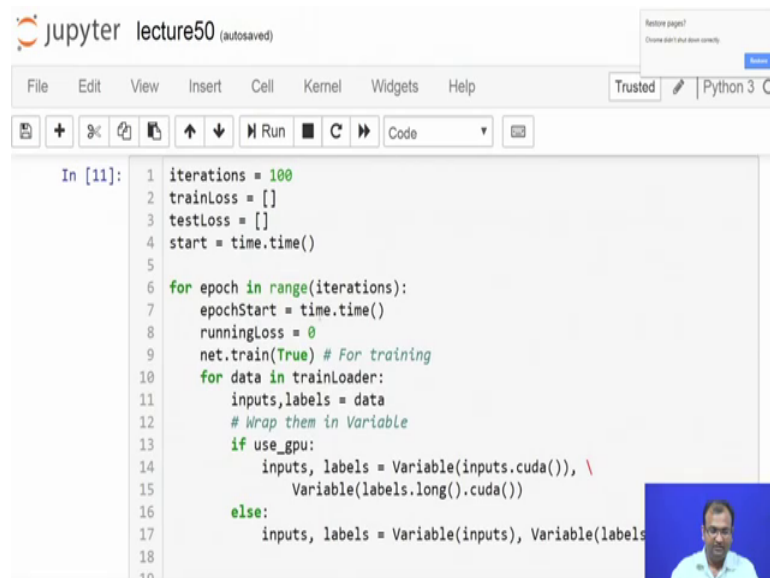
```
In [10]: 1 criterion = nn.NLLLoss2d() # 2D Negative Log-Likelihood
         2 optimizer = optim.Adam(net.parameters(), lr=1e-3)
```

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

The next part is to work on the initialization of this model and that is pretty straightforward. So, you just define a variable over here called as net which is equal to SegNet, and SegNet is the model which is defined by this function created over here. And then if there is a GPU then can just convert it onto cuda. Now, the next part is to define these two important parts; one is your criterion function, another or the loss function another is your optimizer. So, for loss we are going to use classification loss for negative log likelihood.

Now, one thing you need to remember is that the output which it is producing is no more a 1d tensor, but it is a 2d tensor. So, we need to use a negative log likelihood loss which is going to calculate itself on a per pixel basis or a per neuron output over this 2d matrix. And for that there is a separate kind of a function which is called as a nll loss 2d. So, this is just to give down that whatever comes out on your output side you are just going to operate it on the 2d space. And it is 2d tensor and not a d tensor which comes out if you were using a classification model over there ok. Now, you have your optimizer and we choose to use Adam as our optimizer over here.

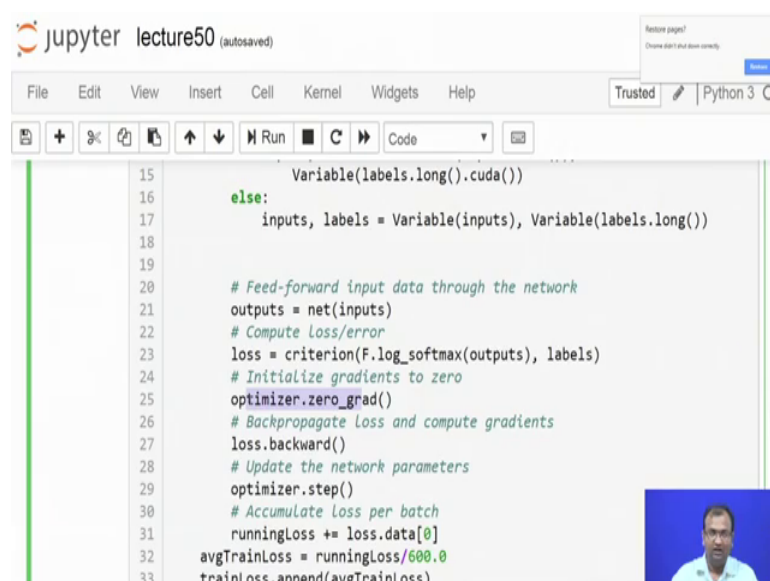
(Refer Slide Time: 18:04)



```
jupyter lecture50 (autosaved)
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3
In [11]: 1 iterations = 100
2 trainLoss = []
3 testLoss = []
4 start = time.time()
5
6 for epoch in range(iterations):
7     epochStart = time.time()
8     runningLoss = 0
9     net.train(True) # For training
10    for data in trainLoader:
11        inputs, labels = data
12        # Wrap them in Variable
13        if use_gpu:
14            inputs, labels = Variable(inputs.cuda()), \
15                               Variable(labels.long().cuda())
16        else:
17            inputs, labels = Variable(inputs), Variable(labels)
```

Now, next comes down my training part over there. Now, we decided to train it just for 100 epochs as such and then we have our stand up routine written down for training. Now, what we do is if I have my GPU available then inputs and labels both are converted on to variables and type casted as cuda. So, these labels are basically the outputs over there on the training data and the input is this image which goes on the input side over there. Now, if I do not have a GPU available then the typecasting is not done, but you still need to convert it onto an auto grade variable for the auto grade solver to come into play.

(Refer Slide Time: 18:31)

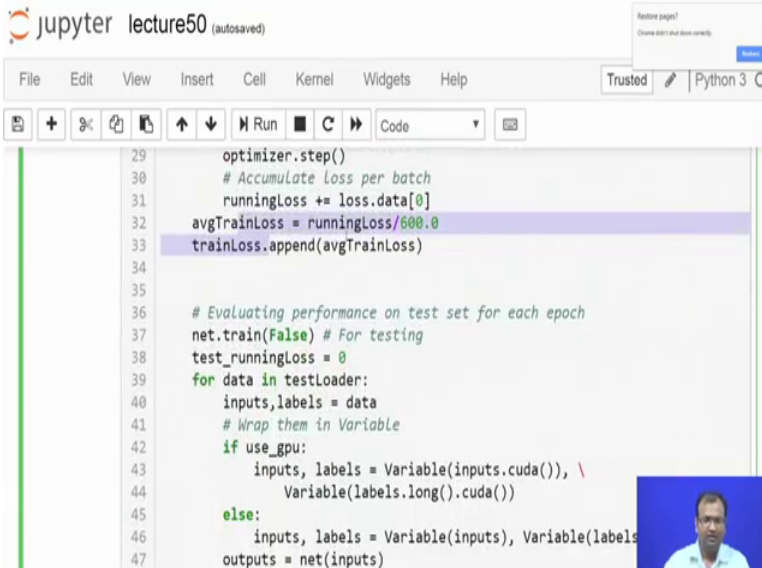


```
15         Variable(labels.long().cuda())
16     else:
17         inputs, labels = Variable(inputs), Variable(labels.long())
18
19
20     # Feed-forward input data through the network
21     outputs = net(inputs)
22     # Compute Loss/error
23     loss = criterion(F.log_softmax(outputs), labels)
24     # Initialize gradients to zero
25     optimizer.zero_grad()
26     # Backpropagate loss and compute gradients
27     loss.backward()
28     # Update the network parameters
29     optimizer.step()
30     # Accumulate Loss per batch
31     runningLoss += loss.data[0]
32     avgTrainLoss = runningLoss/600.0
33     trainLoss.append(avgTrainLoss)
```


Next you do a feed forward and get down your output. So, this output is a three channel output over there of the same size as that of the input image. So, here it is going to stay down to 224 cross 224. Then you have your loss being calculated over here. Now, in case of your loss what you need to do is the outputs over there since I am using a negative log likelihood lost function. So, I need to have a log softmax calculate it over the output so that is what is taken care and you also pass down your labels so which particular pixel is or which particular class is denoting a particular pixel over there.

Next, we zero down our gradients in the optimizer and then you do a lost dot backward or the back propagation operation over the loss which is nabla of loss or del del w or gw is what is calculated over there. Now, this part is the first derivative of the loss function, then you have your rest of the network whose weights are updated, following the update rule and that is with this step optimizer dot step. Then you can find out your running loss or accumulate out loss over all the samples in a given epoch and then you have the same thing stored down over there.

(Refer Slide Time: 19:43)

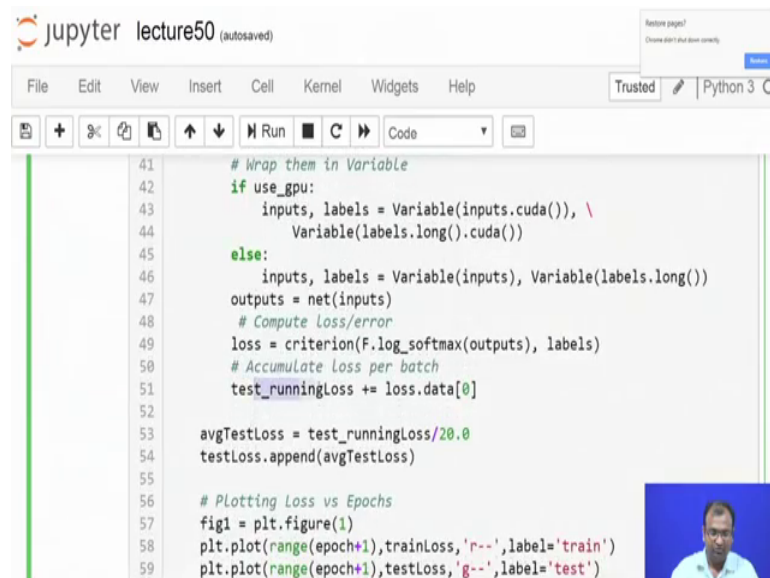


```
29     optimizer.step()
30     # Accumulate loss per batch
31     runningLoss += loss.data[0]
32     avgTrainLoss = runningLoss/600.0
33     trainLoss.append(avgTrainLoss)
34
35
36 # Evaluating performance on test set for each epoch
37 net.train(False) # For testing
38 test_runningLoss = 0
39 for data in testLoader:
40     inputs, labels = data
41     # Wrap them in Variable
42     if use_gpu:
43         inputs, labels = Variable(inputs.cuda()), \
44             Variable(labels.long().cuda())
45     else:
46         inputs, labels = Variable(inputs), Variable(labels)
47     outputs = net(inputs)
```

Now, these are pretty straightforward and simple. The only important change which comes with semantic segmentation or a pixel wise classification is that your loss function changes on to a 2d loss function such that now you can use these not just one single tensor of an output, but now you have a two 3d tensor which is coming out in the earlier case you would have just a 1d tensor coming out as your output

So, after that this training is over, then what we do is we try to look into the performance over there. Now, for performance what we do is we just pass over the data on my testing data available to me. So, there were 20 images which I was using for my testing. Now, I do a feed forward over here then find out my loss using my criteria function and then I keep on accumulating this loss.

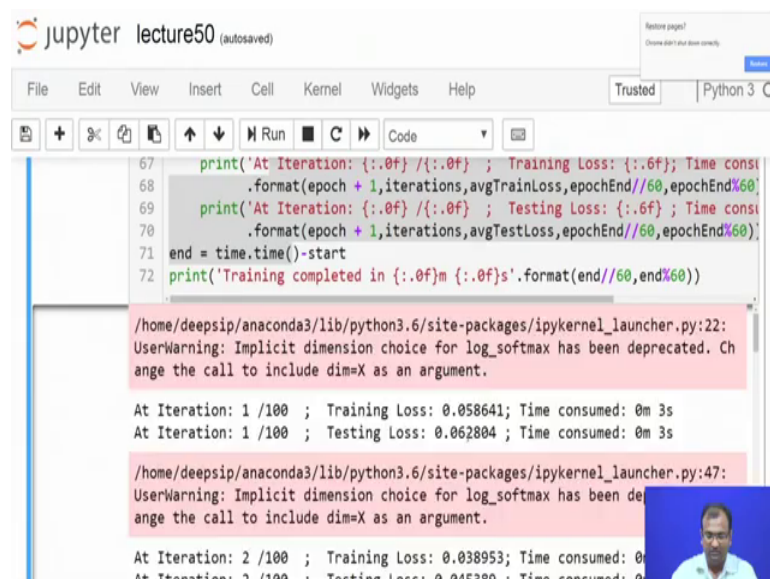
(Refer Slide Time: 20:38)



```
41 # Wrap them in Variable
42 if use_gpu:
43     inputs, labels = Variable(inputs.cuda()), \
44         Variable(labels.long().cuda())
45 else:
46     inputs, labels = Variable(inputs), Variable(labels.long())
47 outputs = net(inputs)
48 # Compute Loss/error
49 loss = criterion(F.log_softmax(outputs), labels)
50 # Accumulate Loss per batch
51 test_runningLoss += loss.data[0]
52
53 avgTestLoss = test_runningLoss/20.0
54 testLoss.append(avgTestLoss)
55
56 # Plotting Loss vs Epochs
57 fig1 = plt.figure(1)
58 plt.plot(range(epoch+1),trainLoss,'r--',label='train')
59 plt.plot(range(epoch+1),testLoss,'g--',label='test')
```

So, this is my validation loss which is getting accumulated over there. Then I have my plotting routines and then my timekeeper over here.

(Refer Slide Time: 20:48)



```
67 print('At Iteration: {:.0f} / {:.0f} ; Training Loss: {:.6f}; Time consumed: {:.0f} s'.format(epoch + 1, iterations, avgTrainLoss, epochEnd//60, epochEnd%60))
68
69 print('At Iteration: {:.0f} / {:.0f} ; Testing Loss: {:.6f}; Time consumed: {:.0f} s'.format(epoch + 1, iterations, avgTestLoss, epochEnd//60, epochEnd%60))
70
71 end = time.time()-start
72 print('Training completed in {:.0f}m {:.0f}s'.format(end//60, end%60))
```

/home/deepsip/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:22:
UserWarning: Implicit dimension choice for log_softmax has been deprecated. Change the call to include dim=X as an argument.

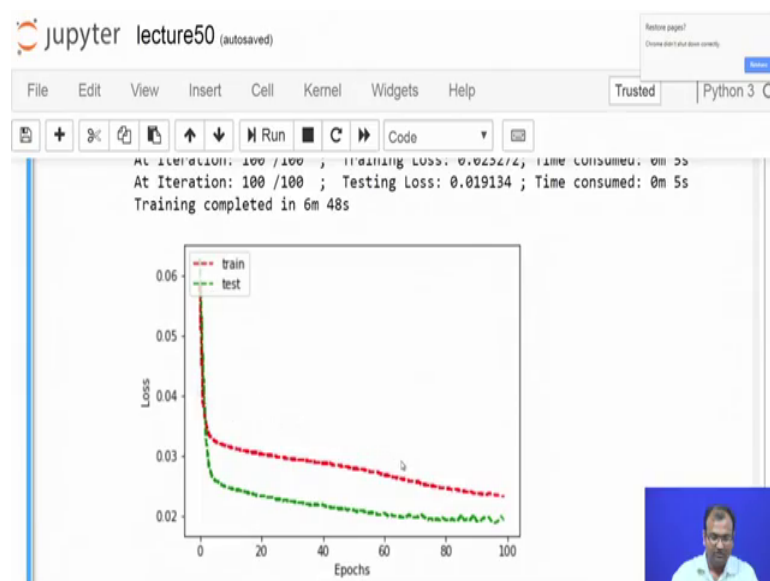
At Iteration: 1 /100 ; Training Loss: 0.058641; Time consumed: 0m 3s
At Iteration: 1 /100 ; Testing Loss: 0.062804 ; Time consumed: 0m 3s

/home/deepsip/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:47:
UserWarning: Implicit dimension choice for log_softmax has been deprecated. Change the call to include dim=X as an argument.

At Iteration: 2 /100 ; Training Loss: 0.038953; Time consumed: 0m 3s
At Iteration: 2 /100 ; Testing Loss: 0.045389 ; Time consumed: 0m 3s

Now, if we go through this one we can pretty much see that it keeps on running over there and then we can find out that per iteration we calculated out separately. So, once you have your training loss reported, and then you are testing losses. So, it starts somewhere around a training loss of 0.05 and then it keeps on decreasing where somewhere around the third one hundred epoch this training loss is something which goes down about 0.02.

(Refer Slide Time: 21:18)



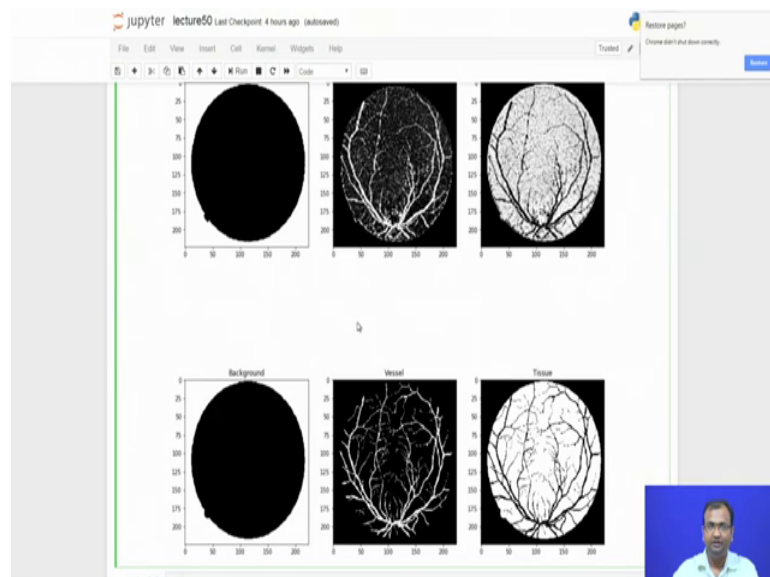
So, this is my funk plot for this training loss as well as the testing loss now. What you can see is that while you are training loss was still higher, but you are testing loss keeps on going low which definitely is a good sign because this means that your model is now generalizing much faster over a larger corpus of images then. So, typically if your test loss is lower than your training loss it means that the generalization ability for your network is definitely good. Whereas, if it is the other way around where your test loss is above the training loss then its prone to over fitting and then that is not a scenario which you would prefer in any cases.

(Refer Slide Time: 21:56)

```
jupyter lecture50 (autosaved)
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3
2 randIdx = np.random.randint(20)
3 testImg = TestImages[randIdx]
4 testLab = TestLabels[randIdx].numpy()
5
6 # Feed-forward
7 segImg = net(Variable(testImg).unsqueeze(0).cuda())
8 # Applying softmax to get class probabilities
9 segImg_np = F.softmax(segImg.data.cpu()).squeeze(0).numpy()
10
11 # Displaying segmented output and ground truth
12 plt.figure(figsize=(15,15))
13 plt.subplot(231)
14 plt.imshow(segImg_np[0,:,:], cmap='gray')
15 plt.title('Channel 1')
16 plt.subplot(232)
17 plt.imshow(segImg_np[1,:,:], cmap='gray')
18 plt.title('Channel 2')
19 plt.subplot(233)
20 plt.imshow(segImg_np[2,:,:], cmap='gray')
```

Now, comes down our visualization of what this model has learnt out. So, what we end up doing over here is pretty simple. I am going to pick up one of these images from my testing set over there. And then on the only one of these images I am going to actually show you what comes out ok.

(Refer Slide Time: 22:07)



So, let us look into what comes out over here. So, we had picked up one of those images on which for my channel one which was basically my background class. So, white is the regions which have the highest probability. So, since the blank out region over there

master out region was belonging to class zero, so that is the region which gets the highest probability over here and that is become one. Channel two was what belong to the vessel, so I can see my vessel probabilities coming up over here. Channel three is the probability of getting down my background tissue over here off of the retina, so that is what I end up getting in this map coming up.

Now, if we compare that with our ground truth then that is also pretty much standard. So, this is just the pixels which belong to the background class, so that is marked in white. So, if a particular pixel belongs to a class, we are just going to show it up in white. When it belongs to my vessels and where it belongs to my background tissue over there. So, this let us zoom out a bit so that you can actually see them.

Now, if you now since you can compare all of these together; so you can see pretty much how accurately the semantic segmentation with such a lightweight network really comes up in segmenting out all of these blood vessels much more accurately then you could have otherwise thought of. And it does not actually take much of amount of time. So, if we get back into our training time consumed over there, so in order to finish off 100 epochs of training it just took me 6 minutes and 48 seconds on our machine so that means, that this kind of a network is actually pretty fast to train and does not have much of a overhead incurred as such.

So, this is one of these examples of trying to have a very simple semantic segmentation using a network of your own side. It is not always necessary that you will have to stick down to very costly networks and much deeper networks with complicated calculations. If the problem demands that you can actually finish it off with simpler networks over here. Then be it so, so this is a clear example of where a simpler network beats any of these more complicated networks in order to solve it out.

So, that is where we come to an end on semantic segmentation and this week lecturer series. So, in the next week we are going to enter into another interesting aspect about deep neural networks and that is called as generative modeling or is there some way that given some random numbers can we generate an image in a very simple sense. So, just stay tuned next week when we get back onto this interesting aspect of genetic models as well.

Till then, thanks.