

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

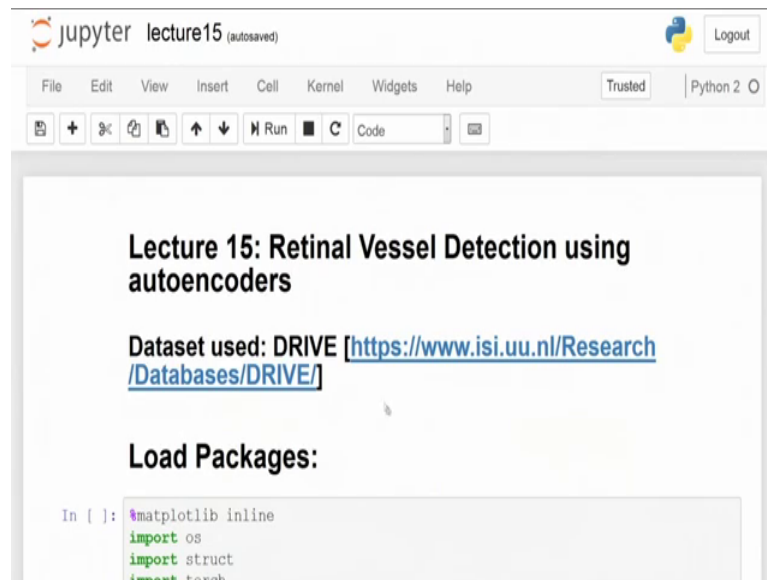
Lecture- 15
Retinal Vessel Detection using autoencoders

So, welcome. Today, in this lecture we would actually be working out on using auto encoders for pixel wise segmentation and which is what today we also call as semantic segmentation problems. And for this particular exercise, I am going to demonstrate it on medical images and the class of images, which we use over here are images of retinal scans. So, retina is predominantly the rear part of the anterior, the posterior part of your eye and where which has all of your photo sensory neurons present over there.

Now, since like it has those photo sensory neurons. So, you have your blood vessels also which go and carry down blood and over here the whole objective is that when there is an equipment called as ophthalmoscope, with which you can actually take an image of the rear part of your eye, then the whole objective is to segment out these blood vessels. And this goes into a very practical problem. Because say, like whenever there is some sort of a disease.

So, you have some damaged out tissues which are also called as lesions, and their location with respect to this blood vessel is quite critical for medical diagnosis and while there are some techniques which are called as angiography where you inject a dye which fluorosis within your eye and then, you can track out the blood vessel. These dyes are typically photo, like toxic to your body, and for that particular purpose there have been a lot of efforts on the field in order to get these blood vessels out, extracted out quite clearly without having to do this extra kind of a dye use over there.

(Refer Slide Time: 01:50)



So, there is a very famous data set. This problem has been there on the community for quite long and the data set which we use is what is called as a drive and the full form is digital retinal images for vessel extraction.

So, it is openly available and in fact, you can also download and have it for your use. So, when you just need to sign up over there with your email id and you would be sent out a link to download the whole data set. It has 2 different sets over there, one is called as a training set, and other is a testing set has in an any kind of a standard supervised machine learning problem. It has typically 3 bunch of images over there.

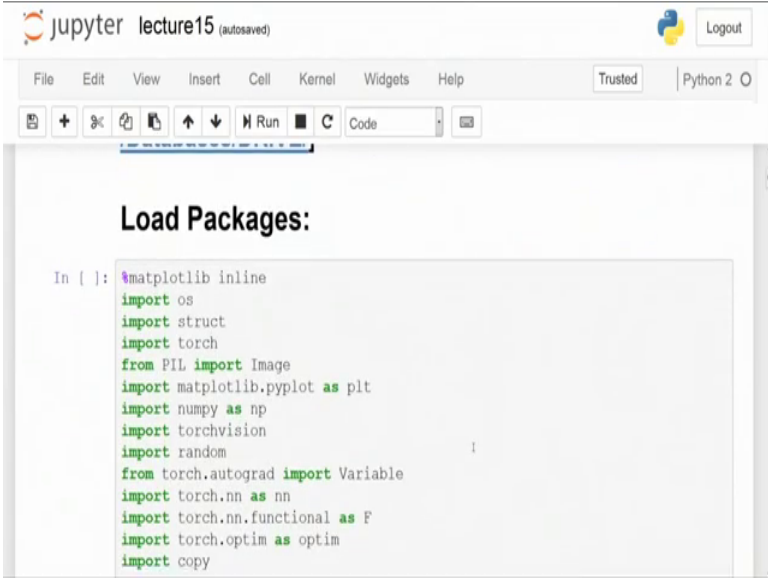
One image is the actual image of the retina, there is another image which has all the vessels marked out correctly like, whatever is a ground truth of the vessels, there is another image which is just called as a mask. So, what that does is, the valid regions where you have the retinal image taken is what is marked as white and rest everything is darkened out so that you do not have any issues. Now, for our purpose we will just be needing this, when we are training we will just be needing the retinal images and the vessel maps over there in order to train it and here the point is that it is no more trying to give a diagnosis out of one image.

So, it is not like you have one image as an input to the autoencoder and you have a class cable coming out, but here it is more of like, there are multiple pixels on the image and

you will have to label each and every single pixel over there, and that is the challenge which we are trying to solve over here.

So, this kind of a problem where you have a simple segmentation approach being solved to mark out and annotate and classify each and every pixel on the image that is what is also called as a semantic segmentation. So, the first step of this is what we are going to do with an auto encoder.

(Refer Slide Time: 03:26)

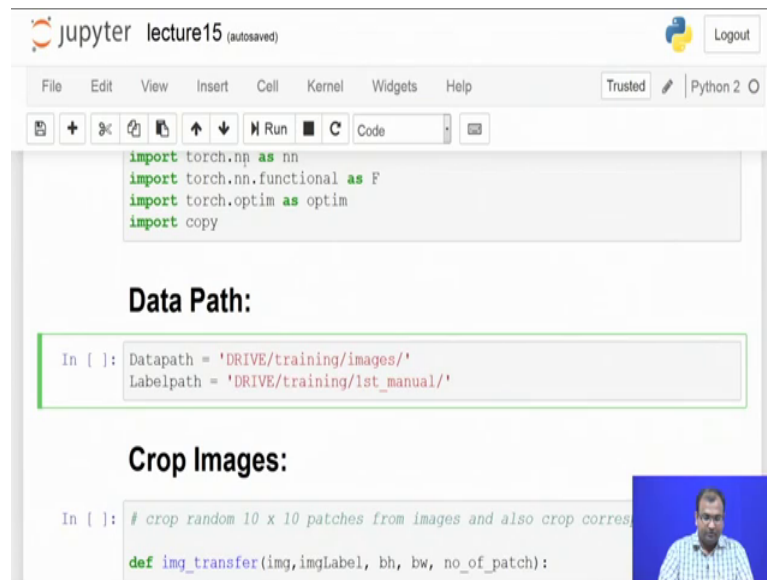


```
Load Packages:

In [ ]: %matplotlib inline
import os
import struct
import torch
from PIL import Image
import matplotlib.pyplot as plt
import numpy as np
import torchvision
import random
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import copy
```

Now, as in with any of our earlier ones, you have your standard header where you are just going to import down all the libraries and keep it ready for you. So, you just need to run this part and that is ready.

(Refer Slide Time: 03:38)



```
jupyter lecture15 (autosaved)
File Edit View Insert Cell Kernel Widgets Help Trusted Python 2
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import copy

Data Path:
In [ ]: Datapath = 'DRIVE/training/images/'
Labelpath = 'DRIVE/training/1st_manual/'

Crop Images:
In [ ]: # crop random 10 x 10 patches from images and also crop corres
def img_transfer(img,imgLabel, bh, bw, no_of_patch):
```

Now, on my data set what I do is, when you download the hold folder and then you have everything coming down onto a zip folder location. So, if you unzip it out, so you will see out this kind of a structure of the directory. So, within your drive if that is the main folder which is also the file name of the zip file itself, you will find down one folder called as training and within that you will have another folder called as images and inside that you will have multiple of those images available to you the labels over there, which is pixel to pixel labeling and then something where you would see all the vessels in white.

So, down the line when we go down I will show you one of those images. So, they are what is available in this folder called as first manual. There is also something called as the second manual and in these training sets and the whole rationale around with that second manual is that there were 2 annotators who were annotating. So, there were 2 ophthalmologists who sat down and annotated each and every single pixel over there.

So, the first ophthalmology annotation is given in one folder, the second is given in another folder. So, for our purpose we are just going to make use of one of these annotations not both of them, and we will just keep on it. Whereas you are quite free in fact, what you can do is, if you take down both the annotations. So, you are actually doubling up your training data set, because your class levels are coming down from 2 different sources and that would make you have build up much robust system. Later on,

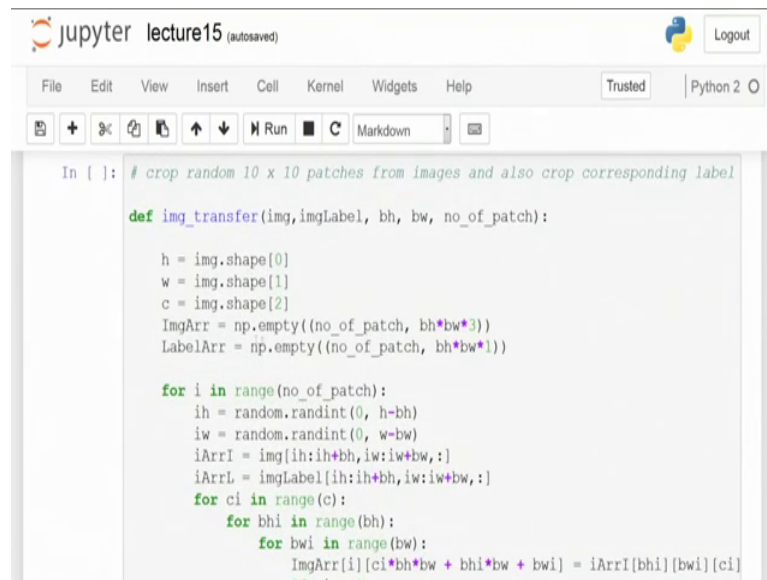
when we go towards advance once of doing multiple instance learning and also aggregate learning and then online learning and how to do some sort of a domain adaptation problem, you would be learning more about how to make use of multiple expert annotations in order to make a system more and more robust.

So, we run down this part and I have 2 strings of my data path and label path which come down to me. Now here what my objective is, I model the whole problem as something of this sort. So, as you had done in one of the lectures where I was also mentioning about this problem. So, one pixel can be considered as some sort of a dependent response based on it is neighbors over there.

And for me what I do is, let me take down say 10 cross 10 sized patches over there, and whatever is the central pixel over there, that central pixels label, is what is associated with these patches over there. So, now, as in with your classical traditional computer vision what you have, would have done is, every patch gets represented in terms of a feature which is derived out of that patch. And based on the features you are going to make down a decision over there.

Now what I consider is, instead of deriving out some features from the patch, I will take the whole patch as my input to the network and features around a particular pixel on that patch is what will be learnt within the network by the auto encoder itself. So, that is typically what we try to solve over here. So, initially for sampling out these random locations, our idea is just to write down a function over there.

(Refer Slide Time: 06:34)



```
In [ ]: # crop random 10 x 10 patches from images and also crop corresponding label

def img_transfer(img, imgLabel, bh, bw, no_of_patch):

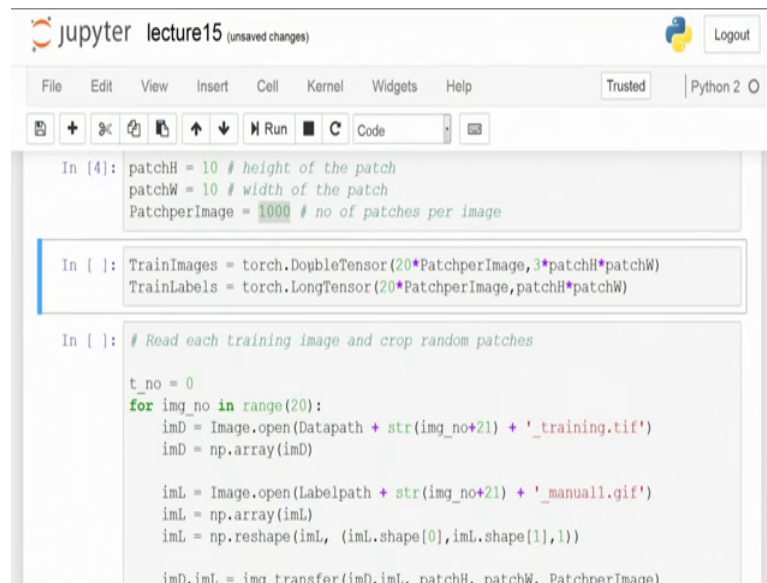
    h = img.shape[0]
    w = img.shape[1]
    c = img.shape[2]
    ImgArr = np.empty((no_of_patch, bh*bw*3))
    LabelArr = np.empty((no_of_patch, bh*bw*1))

    for i in range(no_of_patch):
        ih = random.randint(0, h-bh)
        iw = random.randint(0, w-bw)
        iArrI = img[ih:ih+bh, iw:iw+bw, :]
        iArrL = imgLabel[ih:ih+bh, iw:iw+bw, :]
        for ci in range(c):
            for bhi in range(bh):
                for bwi in range(bw):
                    ImgArr[i][ci*bh*bw + bhi*bw + bwi] = iArrI[bhi][bwi][ci]
```

So, what it does is, it defines out a small array over there, which is of the same size with an height as the normal image over there. Now you are, this point over, on the image annotation that is what is defined as a bh cross wh cross 3 which means that, it is 10 into 10 into 3 given that I have an RGB image over there, and my labels over there for this particular patch is what is bh cross wh cross 1. Because it is just either 0 or 1, it is one single plane over there and nothing beyond it.

So, this part of my code just randomly selects out locations of where I can select out these patches coming down from. So, let us initialize this part of my function for copying out patches. Now here what I define is, if you clearly see, then this function does not have 10 cross 10 explicitly mentioned anywhere because, I am just using some relative numbers being given down over here and some containers or some reference variable points in order to return down my in order to get down my actual bigger sized image of say 512 cross 512 pixels over there.

(Refer Slide Time: 07:46)



```
jupyter lecture15 (unsaved changes) Python 2 O
File Edit View Insert Cell Kernel Widgets Help Trusted Python 2 O
In [4]: patchH = 10 # height of the patch
        patchW = 10 # width of the patch
        PatchperImage = 1000 # no of patches per image

In [ ]: TrainImages = torch.DoubleTensor(20*PatchperImage, 3*patchH*patchW)
        TrainLabels = torch.LongTensor(20*PatchperImage, patchH*patchW)

In [ ]: # Read each training image and crop random patches

        t_no = 0
        for img_no in range(20):
            imD = Image.open(Datapath + str(img_no+21) + '_training.tif')
            imD = np.array(imD)

            imL = Image.open(Labelpath + str(img_no+21) + '_manuall.gif')
            imL = np.array(imL)
            imL = np.reshape(imL, (imL.shape[0], imL.shape[1], 1))

            imD, imL = img_transfer(imD, imL, patchH, patchW, PatchperImage)
```

So, here I start by defining my actual patch size in terms of patch height and patch width, as well as the number of patches per image. So, what I define is I would like to have down 1000 random patches selected per image. Great, so we just get that one down and then, we need some place to store all of them. So, you remember that we always typically had 1 tensor being created out.

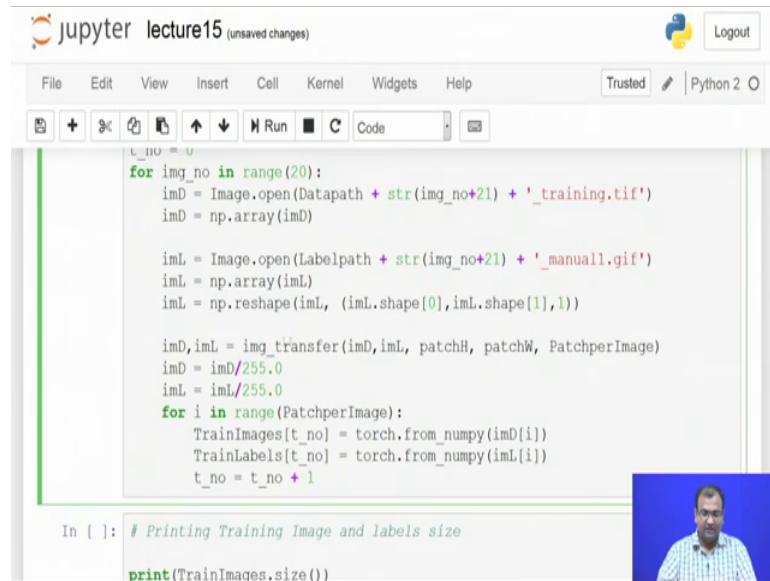
So, when you were having your data loader being used over there. So, the data loader had some parallel workers which were pulling out in each batch itself and then that is what was being used, but then every batch's size was equal to the batch length into 3 if it was a color image, into 1 if it was grayscale image, into height into width. So, for your MNIST you had 28 cross 28. So, it used to become say, if there are 10, 1000 images taken on the done back. So, it becomes 1000 cross 1 cross 28 cross 28.

Now, here I decided to take 1000 of these 1's, each of size 10 cross 10 and they are grayscale. So, typically then it would be basically 1000 into 3 into 10 into 10. However, my training set, I basically have 20 images available in my training. So, 1000 patches taken down from each image and there are 20 such images. So, you would be getting down 20000 of such training patches created out in 1 single batch and that is the size over here which I do for my training images, and similarly is a matrix which is defined for my training labels as well. So, now, once this part comes down to you then what we

do is, so let me run that part. Now here, I am just going to randomly crop and patch out each of these part.

So, it is quite simple. So, what do you need to do is, you will have to actually open up the image.

(Refer Slide Time: 09:39)



```
lecture15 (unsaved changes)
File Edit View Insert Cell Kernel Widgets Help Trusted Python 2
t_no = 0
for img_no in range(20):
    imD = Image.open(Datapath + str(img_no+21) + '_training.tif')
    imD = np.array(imD)

    imL = Image.open(Labelpath + str(img_no+21) + '_manuall.gif')
    imL = np.array(imL)
    imL = np.reshape(imL, (imL.shape[0],imL.shape[1],1))

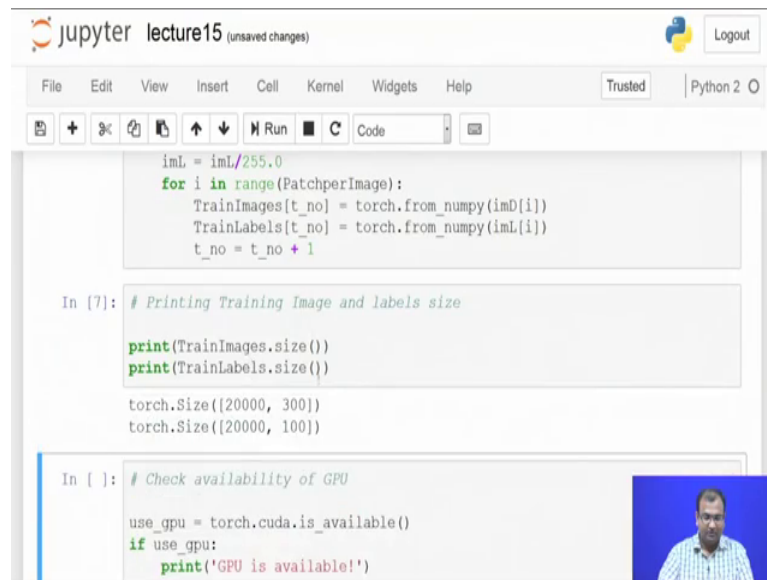
    imD,imL = img_transfer(imD,imL, patchH, patchW, PatchperImage)
    imD = imD/255.0
    imL = imL/255.0
    for i in range(PatchperImage):
        TrainImages[t_no] = torch.from_numpy(imD[i])
        TrainLabels[t_no] = torch.from_numpy(imL[i])
        t_no = t_no + 1

In [ ]: # Printing Training Image and labels size
print(TrainImages.size())
```

Once the image has been opened up, then you need to have some sort of randomized selection of locations from where you are going to patch it out, and once that is done you store it in your array over there.

So, that goes. So, we need to wait for some time for it to actually read and get that one down and by now it is ready. So, here let us look into the size of what we wanted to do.

(Refer Slide Time: 09:59)



```
imL = imL/255.0
for i in range(PatchperImage):
    TrainImages[t_no] = torch.from_numpy(imD[i])
    TrainLabels[t_no] = torch.from_numpy(imL[i])
    t_no = t_no + 1

In [7]: # Printing Training Image and labels size
print(TrainImages.size())
print(TrainLabels.size())

torch.Size([20000, 300])
torch.Size([20000, 100])

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

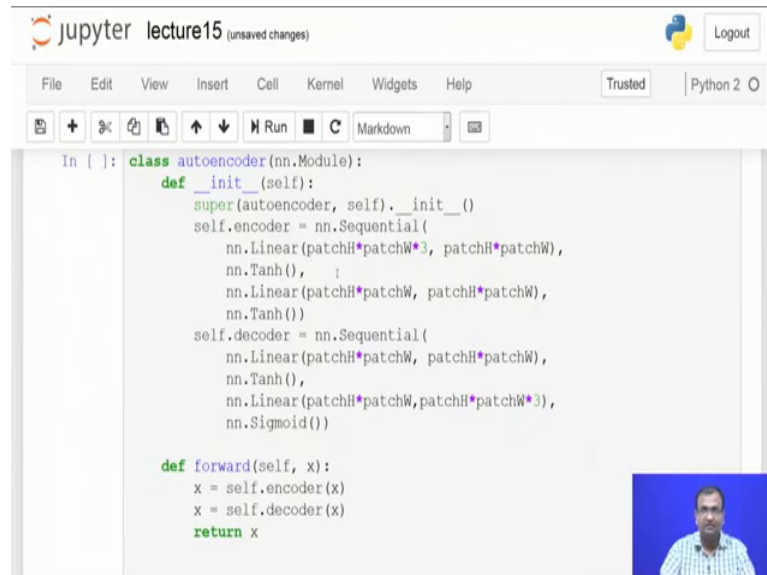
So, we to had 20 images and 1000 patches selected out from 20 images, you got down basically 20000. Now over here you see that there are 300 such neuron locations or it is just 20000 into 3. This comes from the fact that, I had a 10 cross 10 image, and there were 3 color channels. So, the total number of neurons on my input side becomes 300. So, it is 3 into 10 into 10. Similarly on my output side, I just had 1 channel, because it was either black or white, true or false, and us had a 10 cross 10 patch. So, it is 1 into 10 into 10 that makes it hundred neurons over there.

So, now my neural network will have an input over here, which goes down as 300 neurons, my output over there is 100 neurons and in between it has to train itself as an autoencoder. So, quite unlike where in the earlier cases you had, where you were associating only 1 label and 1 output coming down to 1 single patch or 1 single image over there. Here we are going to associate a label correspond to each single position over there.

Now one way you can treat all, you can always treat this as a regression problem, you can train this end to end, but since we are doing it with autoencoders or our first objective is actually to do an unsupervised pre training in order to learn down features, and then use these features in order to reconstruct back my actual space of all the pixel labels over there.

Now, that goes to the next point, which is check out on my gpu availability and I have my gpu created.

(Refer Slide Time: 11:29)



```
In [ ]: class autoencoder(nn.Module):
        def __init__(self):
            super(autoencoder, self).__init__()
            self.encoder = nn.Sequential(
                nn.Linear(patchH*patchW*3, patchH*patchW),
                nn.Tanh(),
                nn.Linear(patchH*patchW, patchH*patchW),
                nn.Tanh())
            self.decoder = nn.Sequential(
                nn.Linear(patchH*patchW, patchH*patchW),
                nn.Tanh(),
                nn.Linear(patchH*patchW, patchH*patchW*3),
                nn.Sigmoid())

        def forward(self, x):
            x = self.encoder(x)
            x = self.decoder(x)
            return x
```

Now this is what part I was telling you down. So, what I would like to do within this as an Autoencoder is that, I have 300 neurons which comes down to me. So, that is what we do is in a simple way since my patch height and patch width can change, so in fact, I can go up over there and just change the patch height and patch width, make it like 11 into 11 or make it 31 into 31 what whatever you choose to be like. So, my input is patch width into patch height into 3 that is the total number of neurons. That I just connected down to equal to patch height into patch width. So, it means that 300 neurons get connect it down to 100 neurons. Then I have a tan hyperbolic activation function, once that is done, then I connect down these 100 cross 100 to another 100 cross 100.

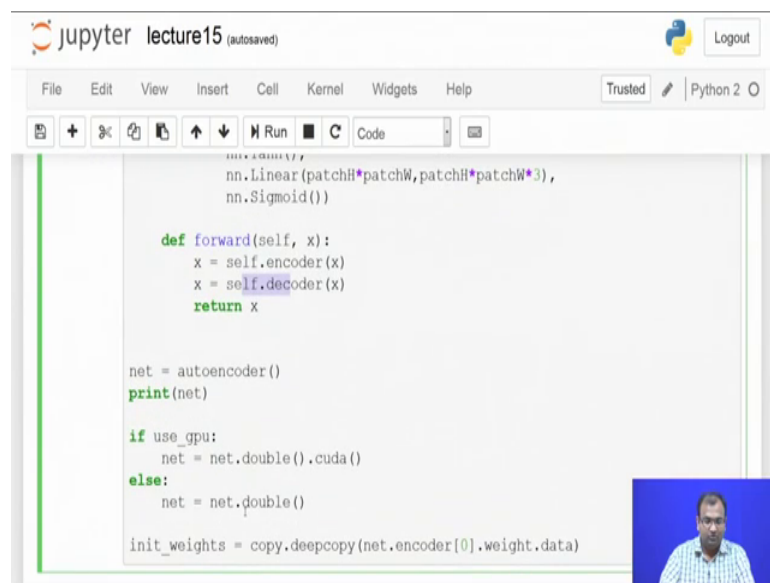
So, it is a very straightforward way of connecting it out, and then a tan hyperbolic. So, by now I have basically 1 hidden layer which connects on my input to this first one, then I have another hidden layer which connects down the output of the first hidden layer to the second hidden layer. So, this is where my, this is where typically my output of the second hidden layer comes out.

Now, that has to go into my decoder side over there, on my decoder, I have that 100 connected down to 100. Now those 100 over here will connect down to 300 because, I am training it just as an Autoencoder over there, and then I have my sigmoid activation

function on the last output over there so that, I have ranges which are in 0 to 1 and they then not necessarily go down in some negative values over there, and then I define my forward pass over my auto encoder.

So, it is quite straight forward that you have your encoder module and you have your decoder module.

(Refer Slide Time: 13:13)



```
nn.Linear(patchH*patchW, patchH*patchW*3),
nn.Sigmoid())

def forward(self, x):
    x = self.encoder(x)
    x = self.decoder(x)
    return x

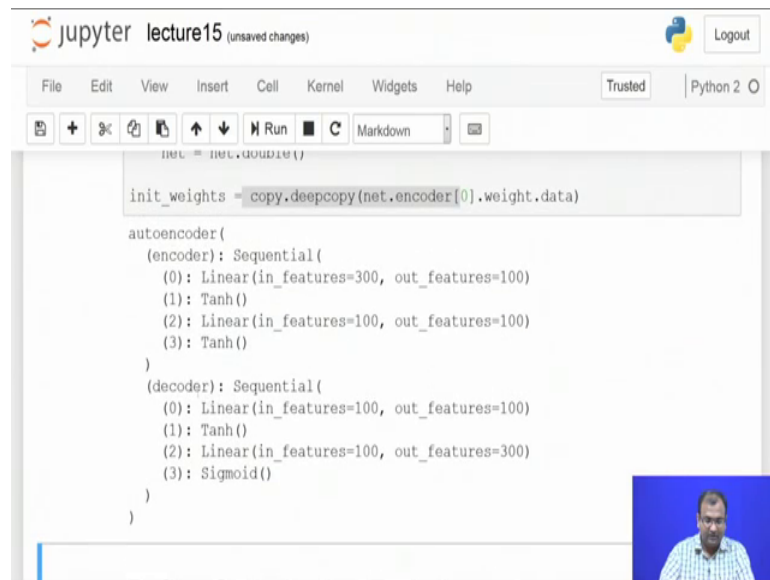
net = autoencoder()
print(net)

if use_gpu:
    net = net.double().cuda()
else:
    net = net.double()

init_weights = copy.deepcopy(net.encoder[0].weight.data)
```

So, you do a input to the encoder whatever comes out, you save that container and just do a feed forward through your decoder module as well, and that would define my whole network, and then I choose to just copy down all my weights and keep it for later on visualization purposes.

(Refer Slide Time: 13:26)



```
net = net.clone()

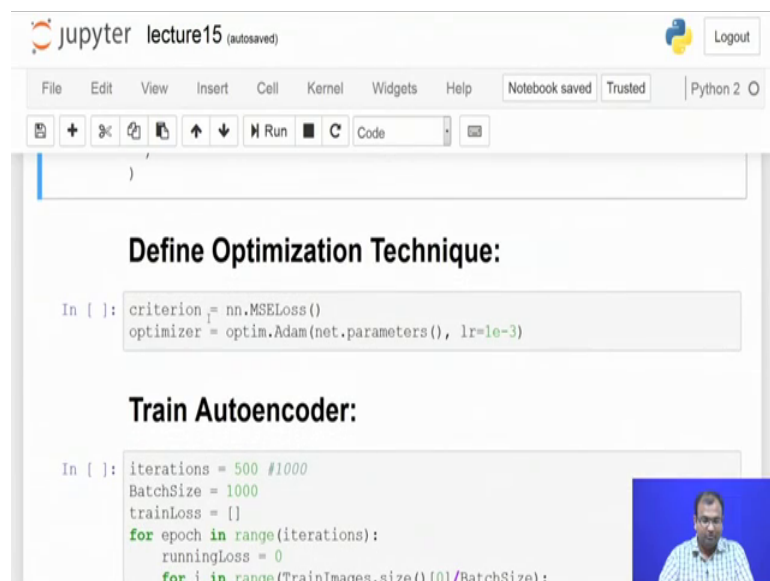
init_weights = copy.deepcopy(net.encoder[0].weight.data)

autoencoder(
  (encoder): Sequential(
    (0): Linear(in_features=300, out_features=100)
    (1): Tanh()
    (2): Linear(in_features=100, out_features=100)
    (3): Tanh()
  )
  (decoder): Sequential(
    (0): Linear(in_features=100, out_features=100)
    (1): Tanh()
    (2): Linear(in_features=100, out_features=300)
    (3): Sigmoid()
  )
)
```

So, let us run this one and then you see this is the network structure which is defined over here.

So, and from, based on whatever we had done in the earlier exercises, you can just use this part of the network, which is your Autoencoder or the encoder part of the network as a feature extractor and then you can use it for classification as well.

(Refer Slide Time: 13:45)



```
Define Optimization Technique:

In [ ]: criterion = nn.MSELoss()
        optimizer = optim.Adam(net.parameters(), lr=1e-3)

Train Autoencoder:

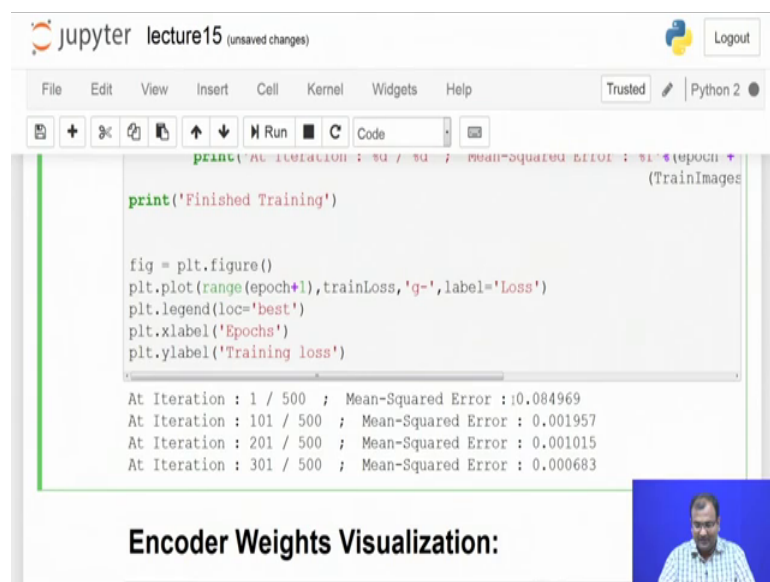
In [ ]: iterations = 500 #1000
        batchSize = 1000
        trainLoss = []
        for epoch in range(iterations):
            runningLoss = 0
            for i in range(TrainImages.size()[0]/BatchSize):
```

But before that we need to train it out. So, my training as of now, since I am using an Autoencoder over there and it is for representation learning, I use a MSELoss function or

instead of trying to print down at every single epoch as I was doing in earlier cases when we were running down with say some 10 epochs, 20 epochs, 15 epochs which was much lesser, but here I am running it down with 500 epochs, now if I keep on really printing it out every single epoch, it becomes a huge table which would be coming out over here.

So, what I choose to do is, just see if the epoch is a integral multiple of 100. So, like every 100 th epochs. So, 100 th epoch then 200 th epoch and so on and so forth it would keep on printing itself over here. And finally, what I choose to do is, that you have all of your losses which are present in this one, per epoch loss over all the 500 epochs is what is present within this array called as train loss.

(Refer Slide Time: 16:10)



The screenshot shows a Jupyter Notebook window titled "lecture15 (unsaved changes)". The code cell contains the following Python code:

```
print('At Iteration : %d / %d ; Mean-Squared Error : %f' % (epoch, 500, trainLoss))
print('Finished Training')
```

```
fig = plt.figure()
plt.plot(range(epoch+1), trainLoss, 'g-', label='Loss')
plt.legend(loc='best')
plt.xlabel('Epochs')
plt.ylabel('Training loss')
```

The output of the code is displayed below the code cell:

```
At Iteration : 1 / 500 ; Mean-Squared Error : 0.084969
At Iteration : 101 / 500 ; Mean-Squared Error : 0.001957
At Iteration : 201 / 500 ; Mean-Squared Error : 0.001015
At Iteration : 301 / 500 ; Mean-Squared Error : 0.000683
```

Below the code and output, there is a section titled "Encoder Weights Visualization:" with a small video thumbnail of a person speaking.

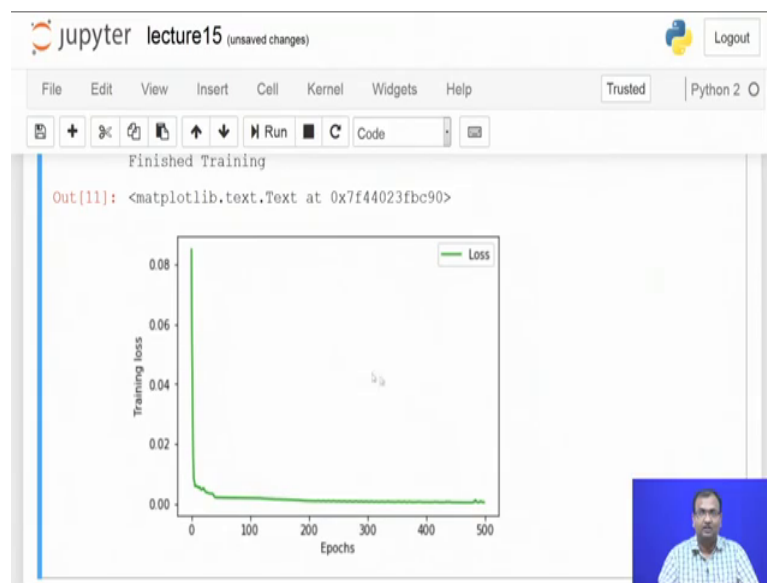
So, here I write down a small function in order to actually plot this one out and see. Because, looking at these text coming down over here, it is not so hard to actually understand how the loss would be going down. So, using this plot functions over here, we are able to actually create a visual outlay of how the loss was decreasing as it was training down an Autoencoder for representation learning. So, let us just wait for some more time because, you can see that there has been a significant decrease, it starts from somewhere around 0.08 and then after 100 it has gone down to 1 figure less and significantly.

So, you can pretty much see that around at 400 epochs it is already 1000 times not 1000 say 100 almost 100 times lesser, then what it was, when it had actually started down at

the first iteration. So, there has been a significant drop as you would see and that would mean that the dynamic range in which the errors are visible are also going to change down significantly and that is it is much dependent on the data and you do not practically have much of a say over there.

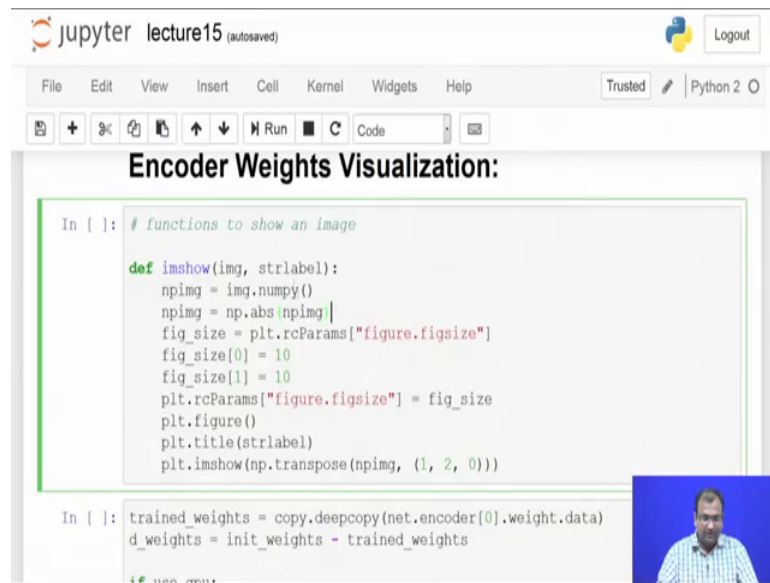
So, now my training is over. And then this is what the plot looks like.

(Refer Slide Time: 17:22)



So, you could see that there was a significant drop happening from this point to this point within even less than a 100 epochs, and that is what you see starting with the 0.08 and coming down at the end of 100 epoch to 0.00195 and then finally, it goes even lower. So, it looks as if it is quite close to 0 as in over here and that is a good number I mean for an MSELoss which is close to the point of 10 power of minus 3 is a 10 power of minus 4 is actually a great number .

(Refer Slide Time: 17:55)



The screenshot shows a Jupyter Notebook window titled "lecture15 (autosaved)". The code in the cell is as follows:

```
In [ ]: # functions to show an image

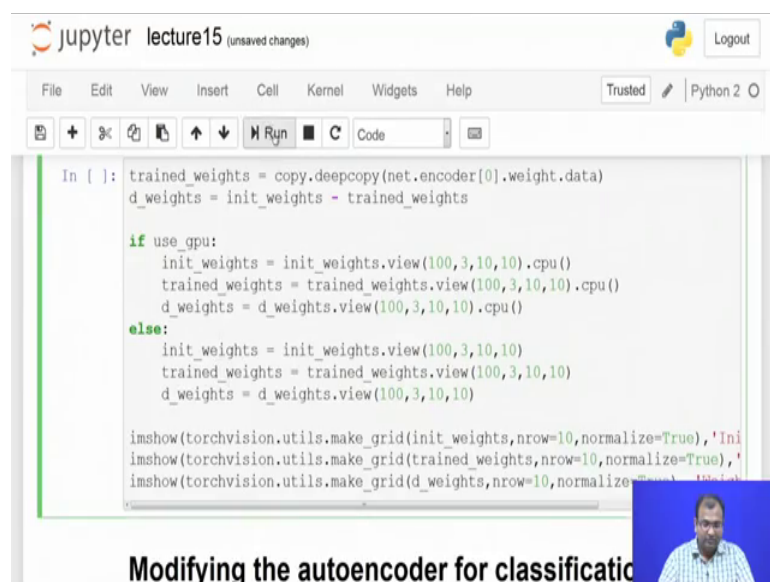
def imshow(img, strlabel):
    npimg = img.numpy()
    npimg = np.abs(npimg)
    fig_size = plt.rcParams["figure.figsize"]
    fig_size[0] = 10
    fig_size[1] = 10
    plt.rcParams["figure.figsize"] = fig_size
    plt.figure()
    plt.title(strlabel)
    plt.imshow(np.transpose(npimg, (1, 2, 0)))

In [ ]: trained_weights = copy.deepcopy(net.encoder[0].weight.data)
d_weights = init_weights - trained_weights

if use_gpu:
```

So, here we decide to actually look into the Visualization of this Weight. So, what I have done is I had initially copied all of these weights and kept it aside. So, if you look over here. So, once my network is trained, once my network is created and before the training starts, actually copy down all of these random weights which have been initialized for each of these weights over there.

(Refer Slide Time: 18:25)



The screenshot shows a Jupyter Notebook window titled "lecture15 (unsaved changes)". The code in the cell is as follows:

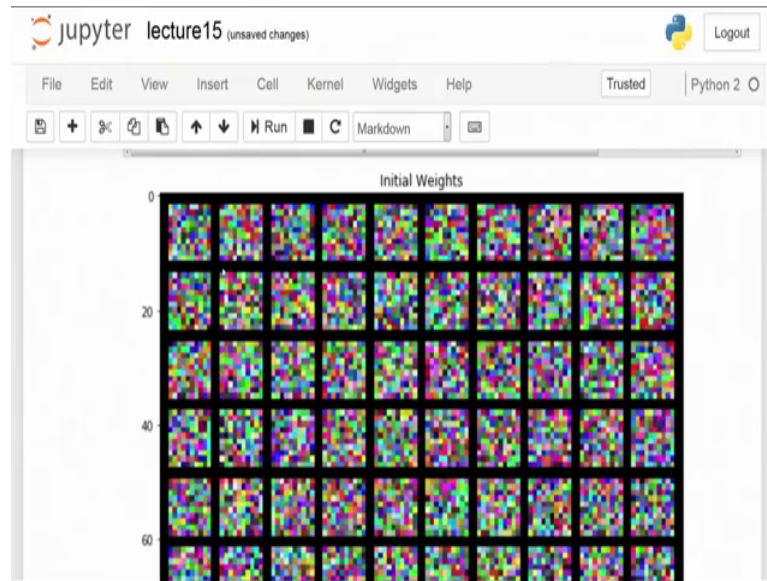
```
In [ ]: trained_weights = copy.deepcopy(net.encoder[0].weight.data)
d_weights = init_weights - trained_weights

if use_gpu:
    init_weights = init_weights.view(100,3,10,10).cpu()
    trained_weights = trained_weights.view(100,3,10,10).cpu()
    d_weights = d_weights.view(100,3,10,10).cpu()
else:
    init_weights = init_weights.view(100,3,10,10)
    trained_weights = trained_weights.view(100,3,10,10)
    d_weights = d_weights.view(100,3,10,10)

imshow(torchvision.utils.make_grid(init_weights,nrow=10,normalize=True),'Init')
imshow(torchvision.utils.make_grid(trained_weights,nrow=10,normalize=True),'Trained')
imshow(torchvision.utils.make_grid(d_weights,nrow=10,normalize=True),'Diff')
```

Now, here is a small function in order to display plot out these matrices in terms of an image and here I would be actually working out to display these ones.

(Refer Slide Time: 18:30)

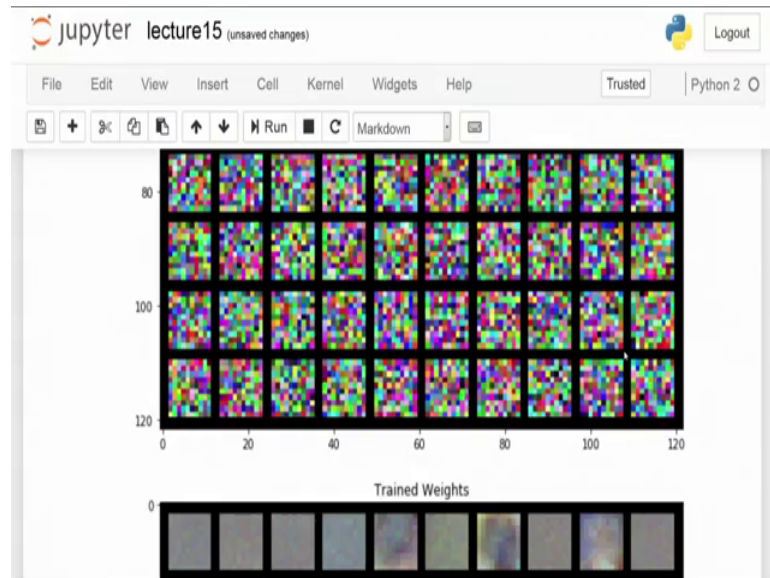


So now, this one is basically matrix of how these weights are associated. You see a lot of colors over there, the reason that you see these in color is because, your inputs there are. So, your input is basically 3 into 10 into 10.

So, you would be having, in some way like 3 channels each of size 10 cross 10 or 3 planes which are matrices each of times 10 cross 10 and you can associate that the pixels which were connecting down the red channel of your image are the ones which are visualized on the red channel in this Visualization, the green channel is on the green channel over here.

And the way it is connected with the blue channel are can present with the blue channel over here. And that is how you get down a colored matrix; however, if you count it down you would get down 10 pixels over here and 10 pixels over here. So, it is a 10 cross 10 matrix into 3 which is for each of the color channels now. So, that is that is technically 300 weights which are visualized in this color format. And if you count down over here you have, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 and similarly on this side you would be having 10.

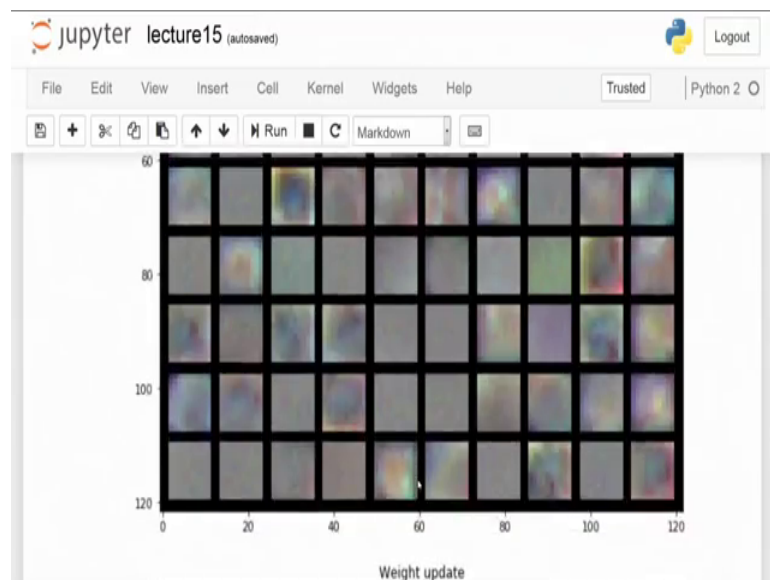
(Refer Slide Time: 19:32)



And that is because you have 300 neurons which are connected down to 100 neurons and we just choose to display it in a 10 cross 10 matrix.

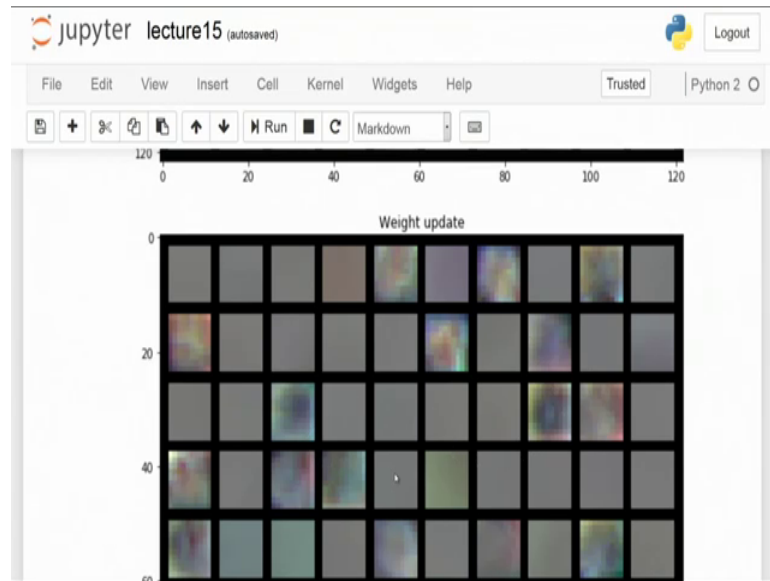
So, this was my random weights which were acquired at the start of the program.

(Refer Slide Time: 19:43)



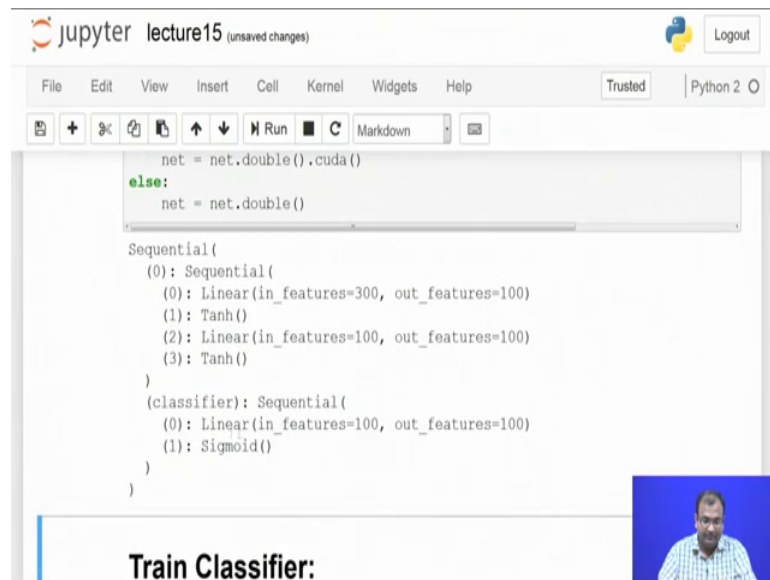
And then after the training, these are some of the visualizations which come down for the bits. Now they do not look that great to say honestly.

(Refer Slide Time: 19:54)



However, if you see that there has been a significant change which comes down on each of these weights though most of them are still 0 and that brings us to the point that this might have learned an over complete representation. So, later on in the next week when we start doing into stacking of auto encoders and sparsity within auto encoders, you would be learning more about details on how this lot of zeros and how similarity in appearance between these weights are something exploited even further. So, as of now the next part comes down that, I have trained on my autoencoder. So, my representation learning is done. Now, I would need to actually do a classification. So, I will have to modify my network, I have to throw down everything on my decoder and replace that with just a sequential connection.

(Refer Slide Time: 20:31)



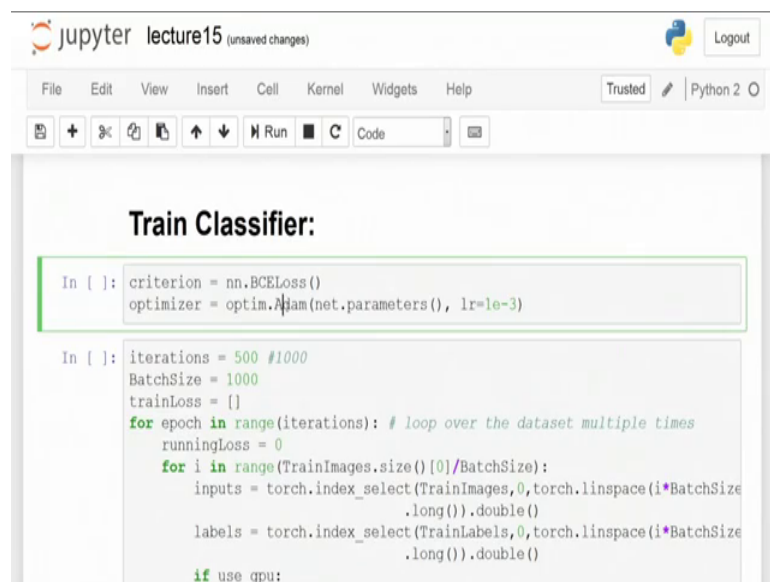
```
net = net.double().cuda()
else:
    net = net.double()

Sequential(
  (0): Sequential(
    (0): Linear(in_features=300, out_features=100)
    (1): Tanh()
    (2): Linear(in_features=100, out_features=100)
    (3): Tanh()
  )
  (classifier): Sequential(
    (0): Linear(in_features=100, out_features=100)
    (1): Sigmoid()
  )
)
```

Train Classifier:

So, I have my second hidden layers output which is just 100 neurons, those have to be connected down to 100 such features because on my output side, I have 10 cross 10 map of all the pixels ok.

(Refer Slide Time: 20:48)



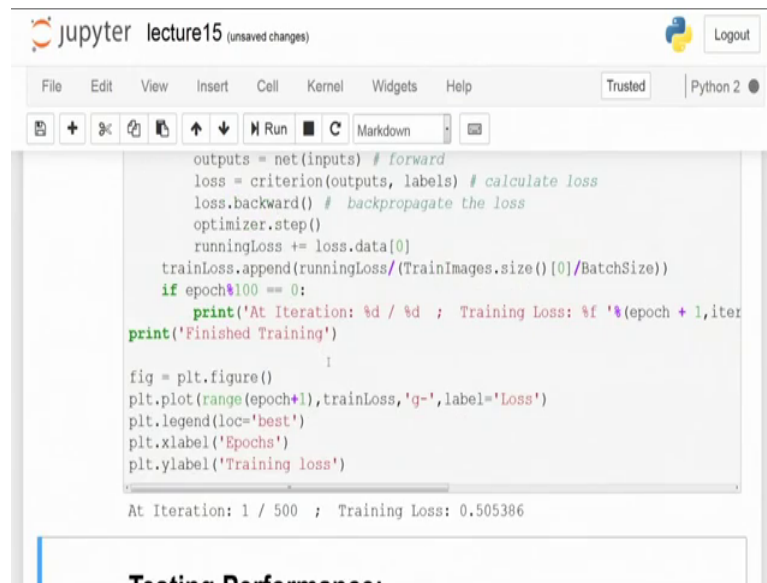
```
Train Classifier:
```

```
In [ ]: criterion = nn.BCELoss()
optimizer = optim.Adam(net.parameters(), lr=1e-3)

In [ ]: iterations = 500 #1000
BatchSize = 1000
trainLoss = []
for epoch in range(iterations): # loop over the dataset multiple times
    runningLoss = 0
    for i in range(TrainImages.size()[0]/BatchSize):
        inputs = torch.index_select(TrainImages,0,torch.linspace(i*BatchSize
            .long()).double())
        labels = torch.index_select(TrainLabels,0,torch.linspace(i*BatchSize
            .long()).double())
        if use_gpu:
```

Now, my next point is actually to train it out as a classifier. So, here I had changed out my criterion function or loss function to a BCELoss as with any kind of a classifier. And I still go on to use my Adam optimizer itself. Now here I start my training for the network.

(Refer Slide Time: 21:04)



```
outputs = net(inputs) # forward
loss = criterion(outputs, labels) # calculate loss
loss.backward() # backpropagate the loss
optimizer.step()
runningLoss += loss.data[0]
trainLoss.append(runningLoss/(TrainImages.size()[0]/BatchSize))
if epoch%100 == 0:
    print('At Iteration: %d / %d ; Training Loss: %f' % (epoch + 1, iter
print('Finished Training')

fig = plt.figure()
plt.plot(range(epoch+1), trainLoss, 'g-', label='Loss')
plt.legend(loc='best')
plt.xlabel('Epochs')
plt.ylabel('Training loss')

At Iteration: 1 / 500 ; Training Loss: 0.505386
```

Now, this would also be taking some amount of time in order to train down; however, this loss is no more than MSELoss, but this becomes a BCELoss and it is really hard to again start directly commenting. What you need to keep in mind is that, while in your earlier examples where you had the classification with MNIST or you had to classify just those white blood cells into whether they are leukemic or non leukemic, you were going to have 1 label associated with 1 single patch of an image.

Here it is sort of a series of labels associated with every single pixel within the image and that is what is changing over there. Now you might even come up with the question that if we are doing going to do it on small patches of 10 cross 10, then why not create the whole image and solve it out yes. In fact, that is possible that is pretty much possible, you can have all of your 512 into 512 into 3 neurons connected densely through some sort of an autoencoder and giving it down.

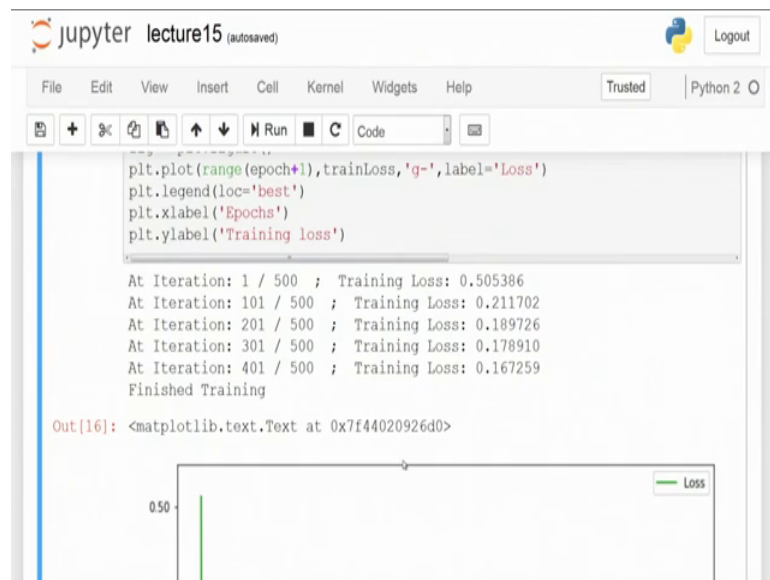
You need to keep something in mind that, the number of weights which you will get down over here will be significantly large. The compute complexity of that problem will also be large and where there would be another significant problem is that, the moment you change down the size of the image, your network is no more of any use over there.

Whereas if I am taking down these smaller patches of 10 cross 10 and then I keep on doing some sort of a non overlapping window and stride it over there for inferencing, then it is quite easy to actually get down on any arbitrary sized image and that would

work out pretty good. So, that is one of the reasons why we choose to go and do it with these smaller sized images. Now if you look into this part of the training, you see that there has been a decrease, however, the rate at which the values are decreasing they are not so significant.

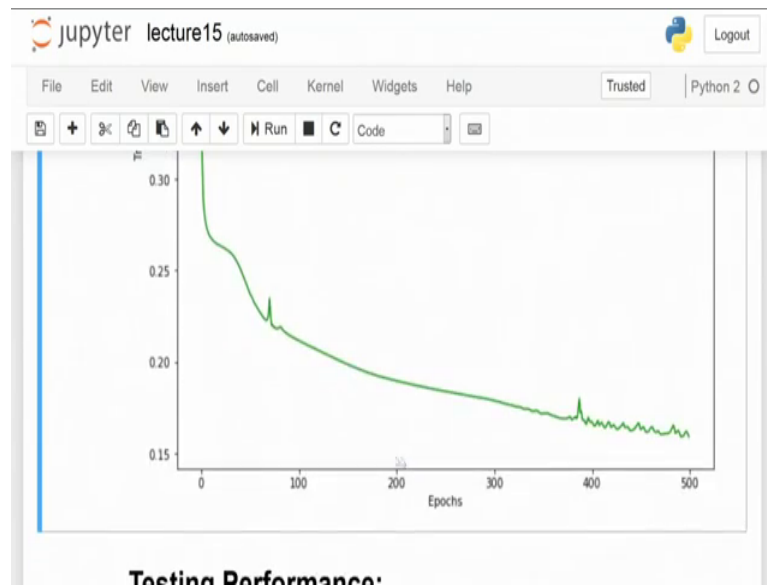
And again saying from our earlier experiences, how the learning rate dynamics would behave across epoch is not something which you can predispose at the start of training over there. And it is very much depends on the data and the dynamics your batch size the kind of optimizers you are using as well as the cost function and that what is goes down over here. So, now, my training is over for 500 epochs and if we look into my error plot over there. So, that desktop screen [FL] desktop screen yeah.

(Refer Slide Time: 23:50)



So, now my training for 500 epochs is over and now if I get back to actually looking into how it works out, so you would see that there has been a significant drop.

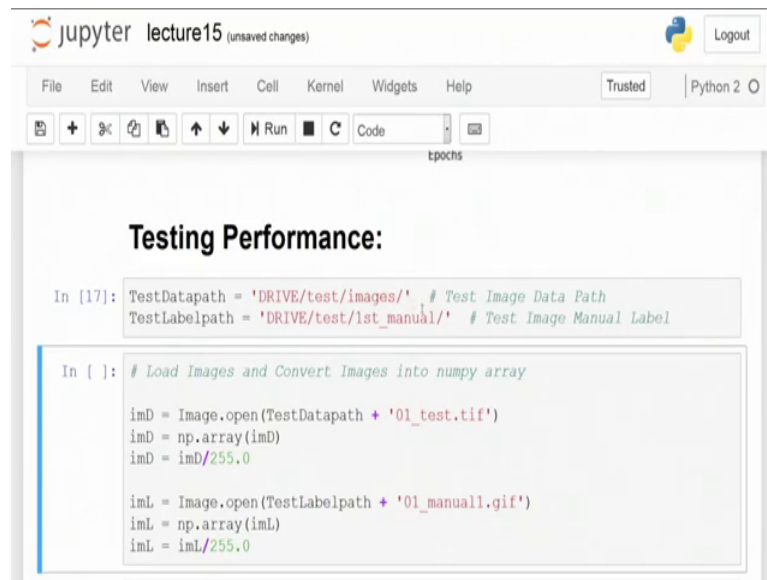
(Refer Slide Time: 23:58)



And while it started down with a BCE loss of somewhere around 0.5, it gradually keeps on going down and somewhere around 100 epochs you have a significant decrease coming down to about close 2 point and then significantly it keeps on doing. You see these jitters over there and that one of the reasons why these small jitters come down over there is, you have your local minima position and then you are going to just keep on saddling around that local minima for some amount of time. And that is the reason why this might be oscillating over there.

Now one way of going around by getting a much smoother curve is to actually, by the point you hit this position on your error. You can actually keep on reducing your learning rates and that would actually help you come down too much further. So, later on when we do learning rate dynamics, we will be covering about those in more details. Now that is about training my whole network. And now, I would be interested in order to look down into how it performs on my testing.

(Refer Slide Time: 24:58)



The screenshot shows a Jupyter Notebook window titled "lecture15 (unsaved changes)". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help), a toolbar with icons for file operations and execution, and a code editor. The code in the notebook is as follows:

```
epochs

Testing Performance:

In [17]: TestDatapath = 'DRIVE/test/images/' # Test Image Data Path
TestLabelpath = 'DRIVE/test/1st_manual/' # Test Image Manual Label

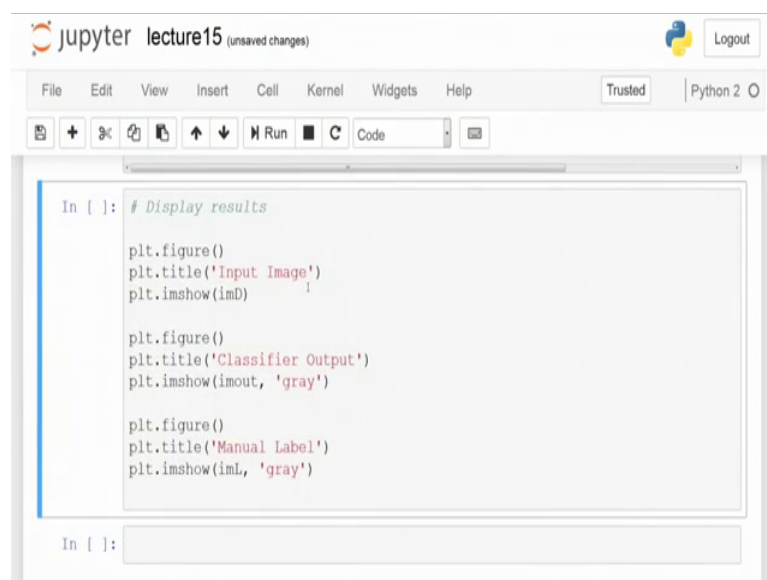
In [ ]: # Load Images and Convert Images into numpy array

imD = Image.open(TestDatapath + '01_test.tif')
imD = np.array(imD)
imD = imD/255.0

imL = Image.open(TestLabelpath + '01_manuall.gif')
imL = np.array(imL)
imL = imL/255.0
```

So, for my testing I have would be taking down images from my test data set and I just take down the ground truth labels for it is evaluation from this thing which is called as first manual. Now here, all of these images are loaded and they are converted on to numpy array and here it is going to break it down into those small 10 cross 10 patches as we had done in the earlier case.

(Refer Slide Time: 25:18)



The screenshot shows a Jupyter Notebook window titled "lecture15 (unsaved changes)". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help), a toolbar with icons for file operations and execution, and a code editor. The code in the notebook is as follows:

```
In [ ]: # Display results

plt.figure()
plt.title('Input Image')
plt.imshow(imD)

plt.figure()
plt.title('Classifier Output')
plt.imshow(imout, 'gray')

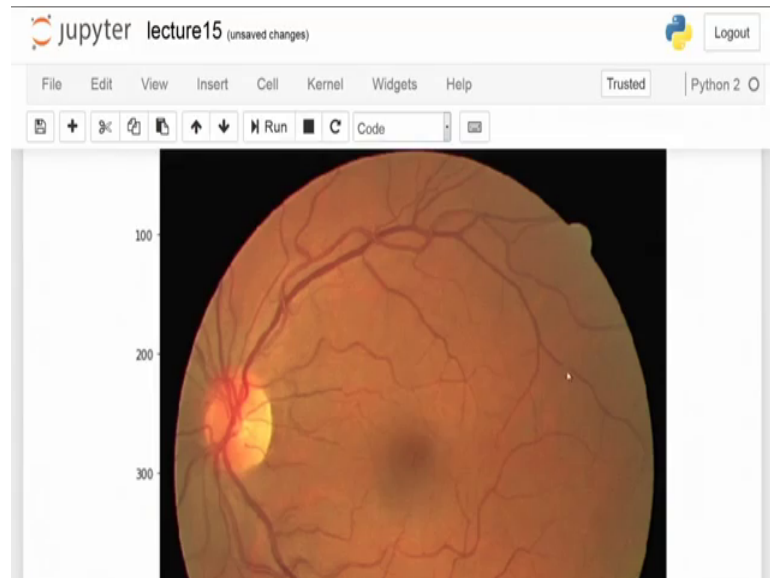
plt.figure()
plt.title('Manual Label')
plt.imshow(imL, 'gray')

In [ ]:
```

Now here, what we are doing is basically 1 single image which is large in size has been broken down into multiple number of non overlapping patches such that it covers down

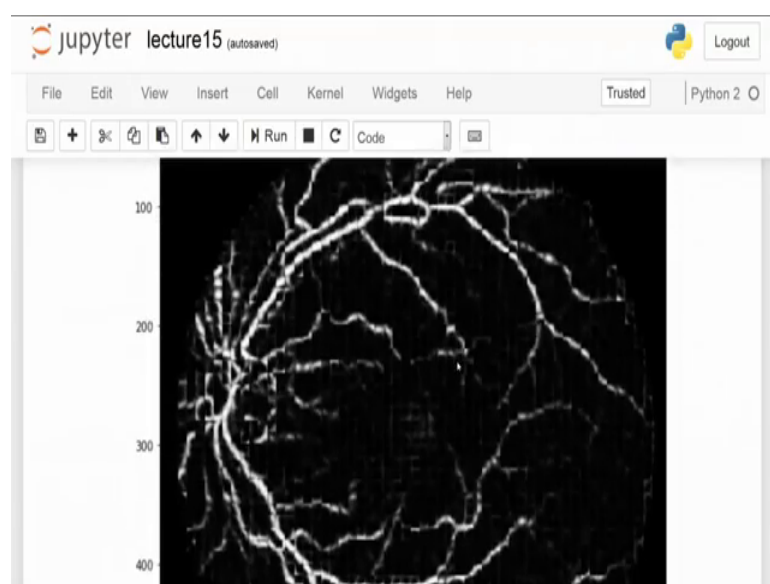
everything. And then for each of the patch, it is going to infer out and then reconstruct that by placing each of the patch back onto the matrix.

(Refer Slide Time: 25:38)



So, this was my input image which I have over here and these are these vessels which you see the dark ones. Now there are even very fine vessels, the ones which you see over here or the ones which you see over here and they are the ones which is which are really hard to predict at all.

(Refer Slide Time: 25:50)



This is the sort of performance which the network as of now performs. And given the fact that, we are just training a very simple autoencoder to solve the purpose, this is not a bad performance at all. I mean, you get a pretty decent vessel map coming down when you know that this is the kind of ground truth label. So, based on our experience of training this network at earlier instances, you can keep on training this for a larger number of epochs and keep on gradually reducing down the learning rates over there.

So, say for after every 500 epochs, you make the learning rate half of what it was in the previous instance. And then you would see it gradually saddle down to much lower error bounds on the binary cross entropy, you will have to train the autoencoder itself for larger epoch say some 5000 epochs or 10000 epochs.

And the classifier on that pre initialized auto encoder for also an equal number of epochs and then you can see down a pretty standard result coming down in terms of these results replicating what we had over here. So, I would leave down those kinds of exploration on to you that would be too much time consuming beyond what we have in one single lecture. So, have those and really enjoy and play around with it and stay tuned for the next weeks where we discuss advance topics on autoencoders including how to do stacking and denoising as well as sparsity criteria till then stay tuned and.

Thanks.