Deep Learning for Visual Computing

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

Lecture - 05
Classification with Perceptron Model

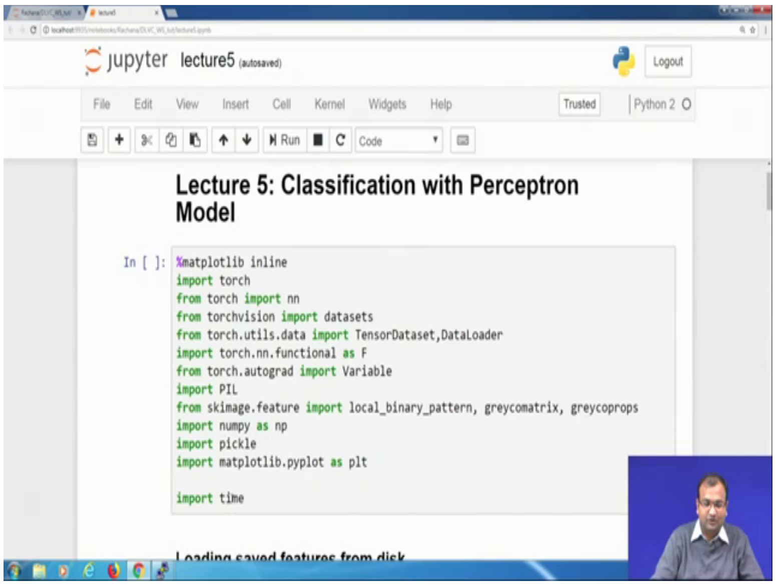
So, welcome to today's lecture and as from we will continue with what we have done on the last class. So, today is a practical lab based session, in which we would be doing coding exercise. So, if you have followed down the last four lectures which we have finished off, so the earlier lecture was on actually getting you introduced to a very linear perceptron model and that is one of the most simplest model of what is called as a neural network and one of the foundational ones. So, today what we are going to do is basically code implementation of a perceptron to be used as a classification model using the features which we had extracted in the earlier classes.

So, if you remember from the third lecture which was basically a coding exercise, and there we had implemented one particular code to extract out features of different kinds and these included LDPS Gabor wave let us co-occurrence matrices. And summary of those features which we had extracted from the complete training set as well as on the test set of c p, and then that was also stored down as a pickle file, which is your local file used within python in order to store down your recorded matrices.

Now, today what we are going to do is build up on top of that. So, in the last class and I had concluded on the point where all of those files were saved, and get down for your offline usage. So, that every time you do not have to reload your images and keep on calculating.

So, today I will be taking up from that point onwards where you have all the files which are saved down, and then appropriately from those I am going to take it forward and create a new perceptron model, and use these features which were already stored in a pickle file by loading it from the pickle file and subsequently go down to run a classification.

(Refer Slide Time: 01:57)


jupyter lecture5 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

Lecture 5: Classification with Perceptron Model

```
In [ ]: %matplotlib inline
        import torch
        from torch import nn
        from torchvision import datasets
        from torch.utils.data import TensorDataset,DataLoader
        import torch.nn.functional as F
        from torch.autograd import Variable
        import PIL
        from skimage.feature import local_binary_pattern, greycomatrix, greycoprops
        import numpy as np
        import pickle
        import matplotlib.pyplot as plt

        import time
```

Loading saved features from disk

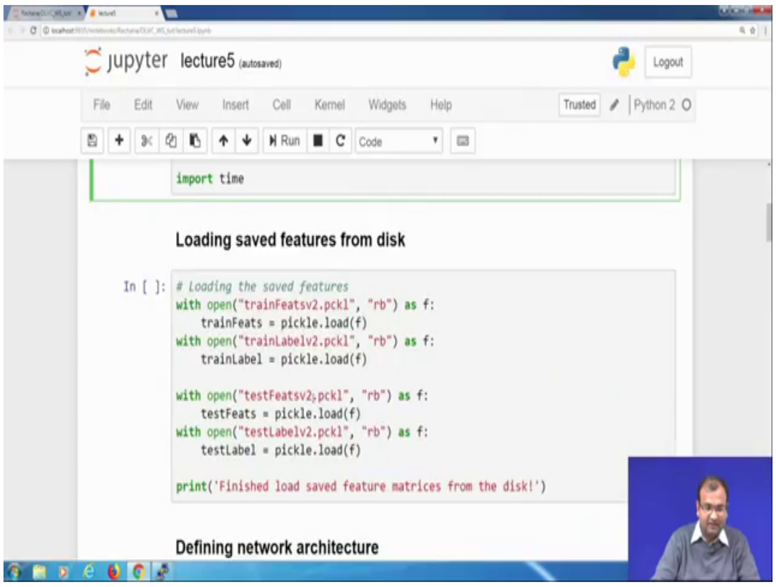
So, the location is still the same. So, you will have to get it down from the GitHub link, where we have been keeping all our codes posted, and then if you invoke the Ipython notebook over there. So, you would be getting on your lecture 5, which is your classification with perceptron models. So, it is a lecture 5 if dot ip ynb and that is the file with which we start.

So, the initial part of the same file is the same header which is over there with an extra point over there also going to use this extra library called as time. And this is just. So, it does not have a major impact as such on your training in anywhere, but this plays a role in order to show you as to that we are accelerating using GPUs and how much time it takes down in order to get the computation done for f e epoch.

So, it is just to show you how much of is the time difference over there, now we have these parts of the header already preserved and all though they are not actually required, but the reason we preserved it if you would like to have all the earlier computation also pasted along with the this part of the code which is you want to calculate the features.

And once you have calculated out the features you would go into it, then this header actually serves as a very generic header to keep down and import all the functions which you need, all though this particular one on cycads feature that is definitely not something which is needed. As well as you will know more be requiring direct access to the torch vision data set because all the required information have been stored down in your pickle file it is elf. So, these are optional but always a good guide to keep them going on.

(Refer Slide Time: 03:36)


jupyter lecture5 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

```
import time
```

Loading saved features from disk

```
In [ ]: # Loading the saved features
        with open("trainFeatsv2.pckl", "rb") as f:
            trainFeats = pickle.load(f)
        with open("trainLabelv2.pckl", "rb") as f:
            trainLabel = pickle.load(f)

        with open("testFeatsv2.pckl", "rb") as f:
            testFeats = pickle.load(f)
        with open("testLabelv2.pckl", "rb") as f:
            testLabel = pickle.load(f)

        print('Finished load saved feature matrices from the disk!')
```

Defining network architecture

Now, in the last class what we had done was we had trained them and we had stored them into a different pickle file. So, these file names may be different from varying to varying, but what you need to keep in mind is that you have one pickle file which is stored down with your features from the training data.

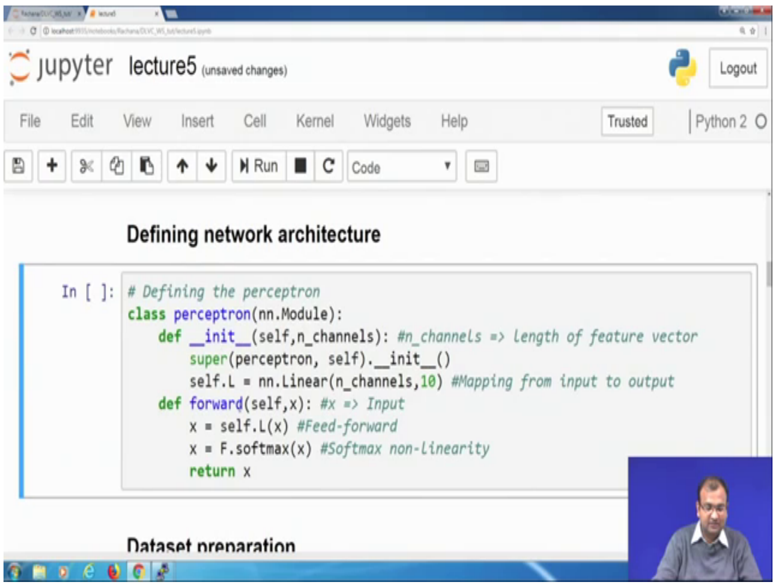
There is another pickle file which stores down all the labels from your training data, and then we need to load basically these two pickle files. So, just remember your location and the file name which you have created, as such if I am not giving down a very explicit location this means that all these files reside in the same directory structure where my base code for running this one is residing.

So, wherever your main code is there it needs to be over there, otherwise if you are from some other location then you can just append down the extra directory location from where these files are to be picked up. Now once you have opened it up the next part is to load this one. So, this is where you just define a file pointer as f, and then you use that file pointer in order to load your pickle, and you get you are basically your arrays of training features and train labels.

Similarly you get your test set features and your test set labels which come down over here, and then like once this is done there is a interactive command to just display whether your matrices have been fetched out. So, let us keep on running one by one, so this is the first model which has to be executed. So, once that is done. So, it works out you get your first instance which has been executed over there.

Now, we go from the comment on to the next one which is about loading. So, it shows that the feature matrices which were saved on the disk they have been loaded good. So, we can go down to the next one which is to define my network architecture ok.

(Refer Slide Time: 05:22)



Defining network architecture

```
In [ ]: # Defining the perceptron
        class perceptron(nn.Module):
            def __init__(self,n_channels): #n_channels => length of feature vector
                super(perceptron, self).__init__()
                self.L = nn.Linear(n_channels,10) #Mapping from input to output
            def forward(self,x): #x => Input
                x = self.L(x) #Feed-forward
                x = F.softmax(x) #Softmax non-linearity
                return x
```

Dataset preparation

So, within my network architecture how it goes down is if you remember from our simple neural network. So, what we had was you have a set of inputs, and you are just connecting it to an output which is your classification.

Now, for classification over here we are doing a ten class classification. So, it means that there are 10 neurons on which the output will be coming down, and these are a one hot neuron kind of a thing. So, what that technically means is that whichever class is present that particular neuron will be one everything else is going to have a 0 value over there ideally, that is what is present on my training data set whenever you are predicting you will get one of the neurons, which will have a very high value which is close to one all the other neurons are going to have a value which is more or less closer to 0.

And that is how we are going to define over there. So, the first part is to define an initialization part, so first we will start by defining a class. So, this class definition is basically trying to define your container within python of saying, that this is a particular 1 which I am using, so what I am going to do is that in order to define my perceptron, I define a class which is called as perceptron and that has a data structure of the type called as a n n dot module.

And these n n dot module is a native structure which has been called on from your torch library. So, we are using the torch in order to implement our pytorch environment over here and within thoughts we are going to import down my n n library. So, within n n I have a particular data type which is defined as n n dot module. Now that I need to start my initialization and this initialization is trying to say down that what is the number of inputs which goes down into my neural network over here.

That is the on the input side of it, and that is equal to my number of channels which I call over here as an underscore channels. So, this is exactly if you look over here, so this n underscore channels it is supposed to be equal to the length of the vector. The next part is that you need to identify what goes sort of like within the perceptron model over here, which is the starting point of my pointer on the constructor, and which is my end point of the constructor.

So, the starting point of the constructor is what gets defined by this command of super or which is just a superior constructor over there, so over here the first part is that I will be defining my perceptron which has this, and then my next part over there is that the perceptron is linked with it is elf and this has to start it is own initialization and how it would be starting it is own initialization is what is defined over here, and this keeps on giving the point that my input to it is basically a linear combination which means that all the neurons which I have on my input side of it they will be densely connected to all the neurons which I have on my output side. And how this is connected is that I have n number of channels in my input set which equals to the length of the feature vector, and they connect down to all the ten neurons on my output side over there.

Now typically for this initial part there would not be much of a modification subsequently when you keep on defining your network, you will have to define this part of your input output relationship as you keep on adding subsequent more layers. So, these we will be taking down into the multi-layer perceptron examples, which we take next week up while doing the theory and then subsequently to the lab.

So, here for our simple perceptron model well you have a set of inputs which are just connected down to 1 what 10 cross 1 output vector. So, this is what initializes and defines my perceptron model.

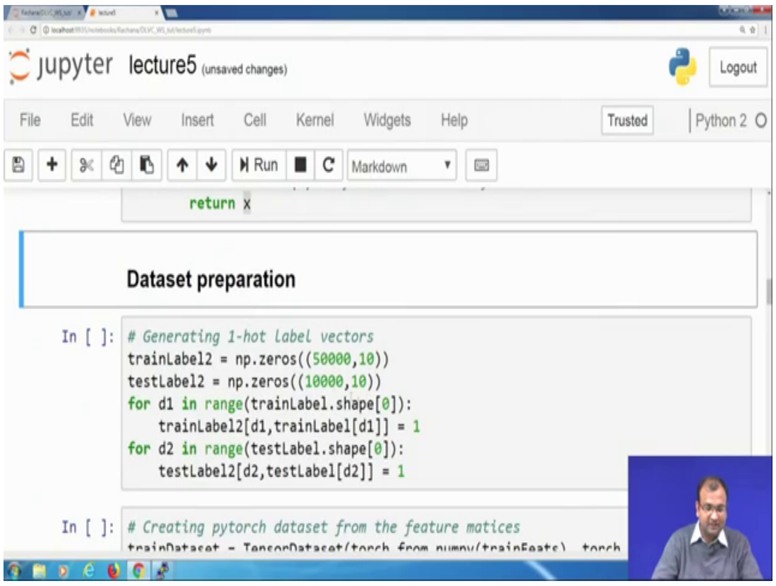
The next part is that I need to define, what is my forward pass over this model or forward pass means, that if I am giving an input I get an output. So, what is the relationship between this inputs to my output, now you remember from our perceptron model over there, so my input output relationship is that all connect switch come down from my input side over there had to be connected down everything to my output. And once I have this feed forward going down the next part is that there will be a summation which happens over there.

So, all the j number of neurons from my input, they will be connecting down to each of these k neurons on my output over there. Now once this summation is achieved you always have a bias vector which is also a trainable parameter within the library it is self, then the next part is that I need to have another non-linearity.

So, this non-linearity is what gets added over here. So, what this means over here is that whatever comes down as my input that gets a linear transformation, so this linear transformation is what is already defined over here, after this linear transformation has been there then I take this same output over here feed it as input to this network, and then I apply a soft max criteria over there.

Now, as I have this soft max non-linearity over there whatever comes as my output that is what I returned from this function, so whenever I do a call for this particular function name called as perceptron, whatever I give as input it will have a linear transformation, then a non-linearity of soft max imposed on top of it and gives me my final output from there. So, this is how I define my perceptron model. So, let us run that part so, I have my function defined over here for my neural network. So, my neural network is not just a function and that is what how we were treating it down throughout.

(Refer Slide Time: 10:40)


jupyter lecture5 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Markdown

```
        return x
```

Dataset preparation

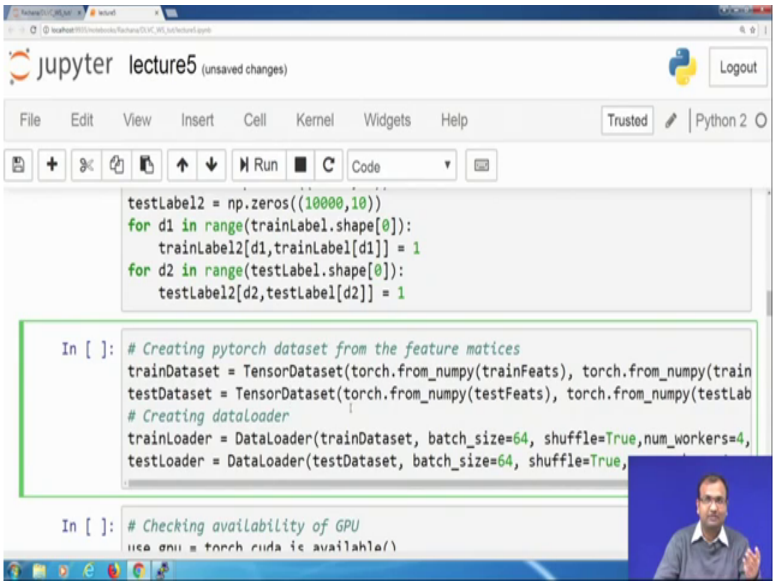
```
In [ ]: # Generating 1-hot label vectors
        trainLabel2 = np.zeros((50000,10))
        testLabel2 = np.zeros((10000,10))
        for d1 in range(trainLabel.shape[0]):
            trainLabel2[d1,trainLabel[d1]] = 1
        for d2 in range(testLabel.shape[0]):
            testLabel2[d2,testLabel[d2]] = 1

In [ ]: # Creating pytorch dataset from the feature matices
        trainDataset = TensorDataset(torch.from_numpy(trainFeats), torch
```

Now, from there the next part which we would like to do is actually generate a data set labels are accordingly. So, here if you remember, so I was telling that in this classification problem you just have a one hot vector remaining in your representation.

So, in that one hot vector what comes out is that as whichever class is present over that that particular element is going to be one everything else is going to be 0, but then in the labels which we had stored. So, they had just numbers from 1 2 3 4 5 6 7 8 9 10. So, it is it is not exactly 10 cross 1 vector of which 1 is 0. So, that is what is done over here by running down another simple for loop.

So, what it would do is that it would define an array of size of 50000 and 10 columns. So, 50000 is the number of elements in your training sample, and 10 is the number of output classes, which can be present over there. And now based on whichever is the class indicated in your train label over there so this part, if that matches down to a particular column index on a given row, then you are just going to set that as one and everything else is otherwise going to remain as zero based on this initialization given down over there.

(Refer Slide Time: 11:54)



```
testLabel2 = np.zeros((10000,10))
for d1 in range(trainLabel.shape[0]):
    trainLabel2[d1,trainLabel[d1]] = 1
for d2 in range(testLabel.shape[0]):
    testLabel2[d2,testLabel[d2]] = 1
```

```
In [ ]: # Creating pytorch dataset from the feature matices
        trainDataset = TensorDataset(torch.from_numpy(trainFeats), torch.from_numpy(train
        testDataset = TensorDataset(torch.from_numpy(testFeats), torch.from_numpy(testLab
        # Creating dataloader
        trainLoader = DataLoader(trainDataset, batch_size=64, shuffle=True,num_workers=4,
        testLoader = DataLoader(testDataset, batch_size=64, shuffle=True,
```

```
In [ ]: # Checking availability of GPU
        use_gpu = torch.cuda.is_available()
```

So, we can just run that part and this generates your one hot label vectors, now once that is done. The next part is that you can actually create a pytorch data set from your feature matrix, and know where this comes down from the fact is that your earlier data structures and whatever videos you are using.

So, they were all in numpy format and numpy is a library which so, all numpy format libraries are used within numpy constructor it is self. So, whenever you are dealing with torch as another library it has it is own data type definitions to go there. So, you need some sort of a wrapper in order to wrap anything any variable which is given as an umpire constructor on to our torch equivalent constructor, and for that the functions which are needed are something which is called as a torch from numpy.

So, what this effectively does is that given any array in numpy you are going to convert it into and torch equivalent 1. And then pack it down into a tensor data set it is self. So, these are all touch tensors which get created. So, we need to create that for the training and testing for both of them.

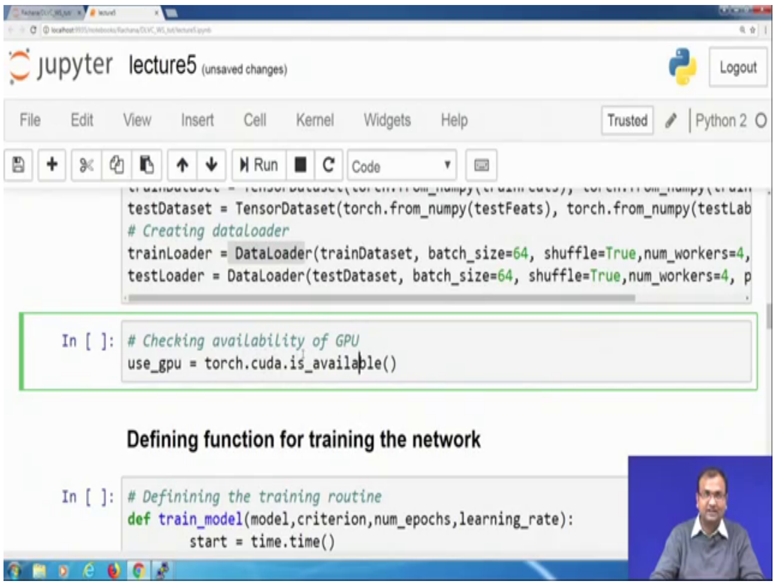
So, just running this one creates it for the training and testing, and then after that for a for the sake of the library and how it works out is that you also have something called as a data loader. So, this data loaders main efficiency is that when you are trying to load data on to say a GPU device over there. So, you have a lot of memory transfers going down between your ram to the GPU.

If you remember from your other related classes on computer organization so, you remember that these those happen down in fetch cycles over there, Now these fetch cycles would typically be when I fetch down some part of my memory from my ram on to from my CPU ram or system ram onto my GPU ram, and then in the GPU the rest of the code keeps on working. Now while this code keeps on running in the GPU my CPU is not doing anything, and I am not even making use of the CPU ram.

So, I can now pre fetch the next set of data which will be needed for operating on the GPU in this duration and that will actually pipeline and speed up my whole operation, instead of having to wait for the GPU operation to get over, and then write it back to the ram and then fetch the next one.

So, this data loader is basically and as is a wise function which has been provided within the library, which can fetched on 64 such units of samples from your CPU ram onto the GPU ram, at any given point of time and. In fact, this does also work down within while you are even trying to fetch down from your hard drive onto your ram as well. So, we will be covering that in to as we go into much more detail classes as well. So, as of now this data loader is now made down to optimize, and then fetch down it pre fetch everything before it keeps on working.

(Refer Slide Time: 14:43)


jupyter lecture5 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

```
testDataset = TensorDataset(torch.from_numpy(testFeats), torch.from_numpy(testLab
# Creating dataloader
trainLoader = DataLoader(trainDataset, batch_size=64, shuffle=True,num_workers=4,
testLoader = DataLoader(testDataset, batch_size=64, shuffle=True,num_workers=4, p
```

```
In [ ]: # Checking availability of GPU
        use_gpu = torch.cuda.is_available()
```

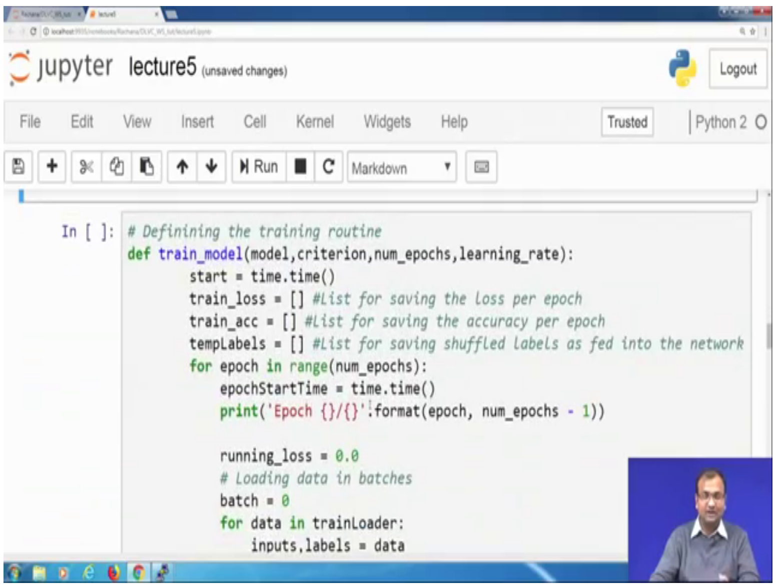
Defining function for training the network

```
In [ ]: # Definining the training routine
        def train_model(model,criterion,num_epochs,learning_rate):
                start = time.time()
```

Now, the next part which we are going to look at is actually check down whether GPU is available on the system. In case a GPU is not available, then we do not use a GPU, so the typecasting and everything nothing is going to get done. So, you are GPU in order for it to work on the data. So, the data has to reside on the ram present, in the GPU you cannot directly operate on the ram which is present on the CPU there are on two different data buses over there. So, you will fetch and that fetch happens through your data loader, once you have fetched over there then within your GPU memory itself you can keep on operating everything.

So, if this is not that then, but the point is whenever you are fetching and putting it on the GPU it means that there is another kind of a typecasting which goes on. So, if the device is not found then we will not be doing that typecasting, because the rest of the codes on CPU they cannot make use of the GPU type cluster it is self. So, here my typecasting gets over, and then I start by defining the function for training the network.

(Refer Slide Time: 15:41)



```
In [ ]: # Definining the training routine
        def train_model(model,criterion,num_epochs,learning_rate):
                start = time.time()
                train_loss = [] #List for saving the loss per epoch
                train_acc = [] #List for saving the accuracy per epoch
                tempLabels = [] #List for saving shuffled labels as fed into the network
                for epoch in range(num_epochs):
                    epochStartTime = time.time()
                    print('Epoch {}/{}'.format(epoch, num_epochs - 1))

                    running_loss = 0.0
                    # Loading data in batches
                    batch = 0
                    for data in trainLoader:
                        inputs,labels = data
```

So, once you had your network defined the next part is that you need to define the trainer. So, this trainer is basically a part where you are defining your forward paths which means that given an x you get an output y. Now what you are getting down was a y hat which was a predicted part of y, you have your original y which is the ground truth available.

Now, based on that you are going to calculate a difference between them which is your cost function, we had used the MSE or l 2 norm as a cost function over there in order to find out our j w, which was the cost associated with the weights. So, this part of the algebra which gets transformed on to a code is what comes down within your training routine. So, as a simple part what we define is a training model function over here. So, the input to this one is a pointer which is the model or the neural network given down over there.

Next is the criteria which is basically the cost function which I am going to use that is also referred to within these libraries as criterion function the number of epochs over which it is going to learn. So, number of epochs basically is the number of iterations over which I would do. So, remember from my last theory class explanation that you start with some arbitrary value, you put your input data you get some predicted output y hat, you take a difference between them. So, that is your cost.

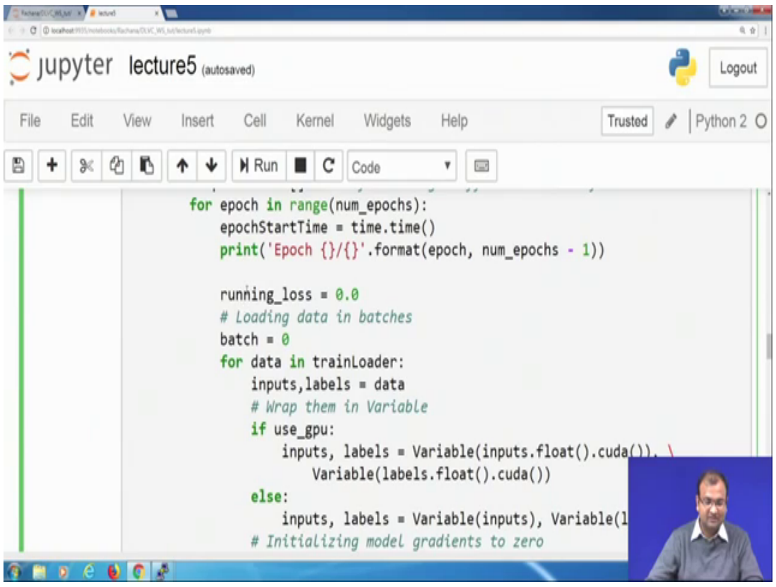
Now, based on that cost you will be taking a derivative of the cost and then back propagating it over the weights back. Now once you have done this one this is over the first epoch, then with these revised weights you are again going to put down your input x over there, and get down another predicted value y hat, get down a new value of j w which is at the end of the second epoch, and then again back propagate and this keeps on going. So, that is what is called as epoch these iteration counter over there.

The next part is learning it which is your eta factor which comes down into your weight update rule. So, your w nu was piece w of n plus 1 was equal to w of n minus eta times Del w of j w. So, this eta factor over there was my learning rate which was helping me scale down and put down this gradient in the order of the magnitude of the weights it is self. So, that they can all scaled on in a similar way and there are no erratic variations. So, what we start initially is we put down a pointer which is called as a starting time point.

Then. So, this is just to show you how much long it takes to traverse down per epoch then, we create a few lists so one of them is for the training loss, one is for training accuracy, another is just for accumulating out temporary labels or whatever is the prediction which comes down at the end of epoch. Now we start down by running a counter which runs over the number of epochs over here. So, epoch is my loop counter variable over here, and then this exists in the range of one to the number of epochs ok.

So, it is basically equal to the counter goes up to the total number of epochs which is present over there. So, let us define a small timer variable over here which is my epochs start time. Now this is just a small printer for printing down like how many epochs out of how many epochs have elapsed out as of now.

(Refer Slide Time: 18:50)


jupyter lecture5 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

```
    for epoch in range(num_epochs):
        epochStartTime = time.time()
        print('Epoch {}/{}'.format(epoch, num_epochs - 1))

        running_loss = 0.0
        # Loading data in batches
        batch = 0
        for data in trainLoader:
            inputs,labels = data
            # Wrap them in Variable
            if use_gpu:
                inputs, labels = Variable(inputs.float().cuda()), \
                    Variable(labels.float().cuda())
            else:
                inputs, labels = Variable(inputs), Variable(l
            # Initializing model gradients to zero
```

And then running loss is a basically the criterion of the cost function evaluated within that particular epoch, and that is 0 down at the start of every epoch, and keeps on accumulating as we keep on going across the epoch. Now, the point is that we will start with loading data. So, my initial data loader will go down into a back size of 0 or the minimum path size which it takes.

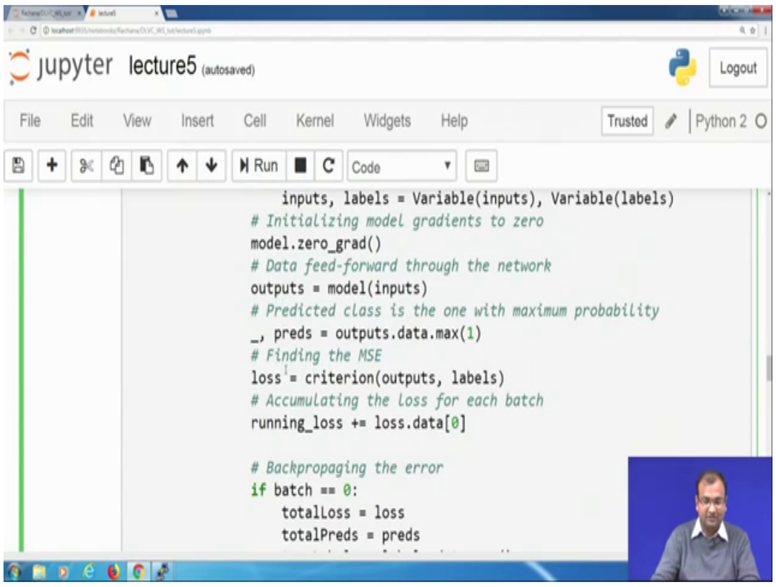
So, then what this definitely means is that I will load, so I will basically traverse 1 data point at a time and go through it. So, within my data over here which exists in my train loader. So, here what we are basically defining is that another loop one which we are looking at the total number of data points existing. So, if I have 50000 once, so within each epoch for every single sample I am going to do a feed forward, then find it is error and then I will be back propagating it over there. So, this is per sample based over there.

So, going from there you remember that we had done this flag of use GPU. So, if this is set to true; that means, that the GPU exists, and whenever there is a GPU existing we need to typecast the variable and set it as cuda. So, if the GPU does not exist then anyways it comes down as else over here, and then I do not need to typecast any of these. Now the way of converting and writing inputs as variable inputs is of the fact that this particular library of pytorch is based a dynamic graph build up constructor, and there they have a separate data type which is called as a variable and if you are not defining certain thing as an explicit variable within the library.

So, they will be taken as constants across and there will be no modifications over there. So, we in order so that these can be used within your update rules, we are putting them into another type caster which is called as variable and this is pretty much intrinsic to the library it is self, that there is no algebraic restriction as to why you need to do that this is from reasoning left best left down to the library builders it is self.

Now, from there the first thing which you need to do is you remember that we had a Del Del w of jw to be calculated, and this value has to be set down initially to 0, and that is what we do with the zero grad factor over there. Now once that is done I will get my set of output. So, this set of outputs is basically if model is which is my function, which is that neural net for which I had created, so whatever is my input I put down to that model I get a set of outputs, and that output is what is the output of the network which I get.

(Refer Slide Time: 21:34)


jupyter lecture5 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

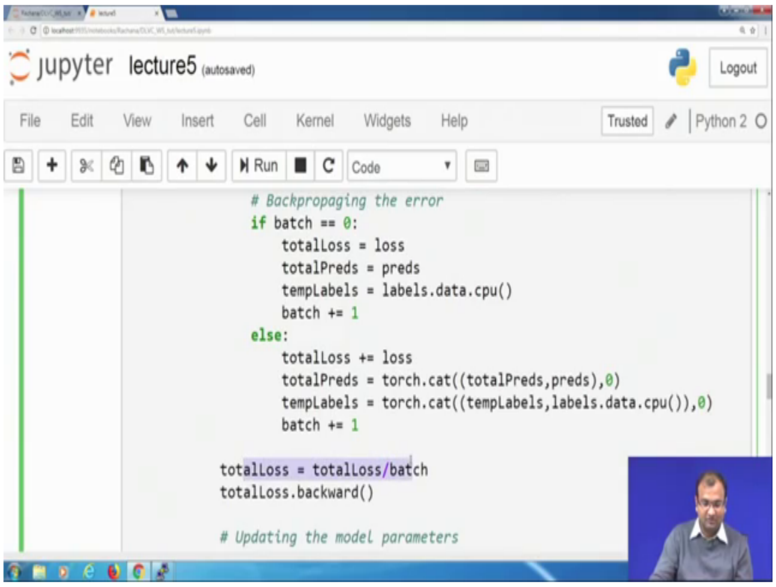
```
        inputs, labels = Variable(inputs), Variable(labels)
    # Initializing model gradients to zero
    model.zero_grad()
    # Data feed-forward through the network
    outputs = model(inputs)
    # Predicted class is the one with maximum probability
    _, preds = outputs.data.max(1)
    # Finding the MSE
    loss = criterion(outputs, labels)
    # Accumulating the loss for each batch
    running_loss += loss.data[0]

    # Backpropaging the error
    if batch == 0:
        totalLoss = loss
        totalPreds = preds
```

Now, once I have this output of the network coming down over there, then what I would like to find out is basically that which particular class over there has the maximum probability of occurring. So, once I get that whichever of class has the maximum probability of occurring. So, I will be just be putting that particular class as a 1 hot everything else is going to subside down towards you now once that is placed down into my prediction.

So, these are basically predictions of whichever is the maximum probability of that coming down. Next I defined on my criteria function as the so, this is my loss and that loss is defined in terms of my criteria function. Now here the input to my loss function or my jw is basically y hat and y. So, y hat is my predicted output which is given over here, and y is the exact ground truth label which is supposed to be present and that is my labels over here.

Now once that is done I can start accumulating my losses within every single epoch. So, in any epoch it well take all the samples and then keep on calculating losses for each sample, and then we will add down losses for each sample together it is self. So, and that would give me the total loss at the end of passing down all the samples through an epoch. Now once I have that part done, then I can actually find out what is my total loss in that complete epoch and that is what is found out over here.

(Refer Slide Time: 23:00)

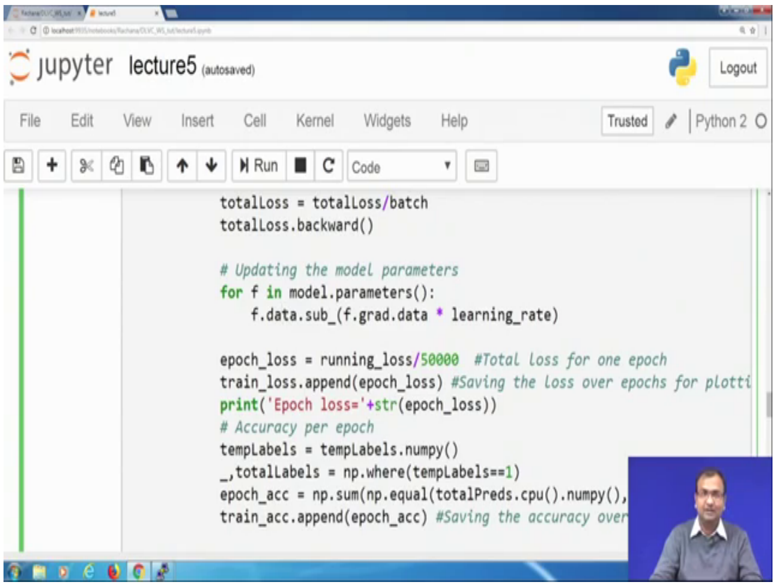

jupyter lecture5 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

```
        # Backpropaging the error
        if batch == 0:
            totalLoss = loss
            totalPreds = preds
            tempLabels = labels.data.cpu()
            batch += 1
        else:
            totalLoss += loss
            totalPreds = torch.cat((totalPreds,preds),0)
            tempLabels = torch.cat((tempLabels,labels.data.cpu()),0)
            batch += 1

    totalLoss = totalLoss/batch
    totalLoss.backward()

    # Updating the model parameters
```

So, I take my running loss and from there I get my so, basically I get my running loss over here, and I have the loss given down from one particular data, and then when I start my back propagating over there first is find out by what is my total loss or the total error which occurs over there, and then I can divide it with the batch size which is equal to basically the total number of samples which were given down in one particularly for batching concepts you will come down a bit later on.

It may so, happen that you might not need to push down all the 50000 data points into one single epoch in one single shot and calculate for each of them, but you can push it down into say 100 samples out of this 50000 go in together, their error is calculated and that is back propagated. And next you have another 100. So, it would mean that there can be multiple back propagation within the same epoch as well whereas; here we are looking into one single back propagation which can happen.

(Refer Slide Time: 23:58)


jupyter lecture5 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

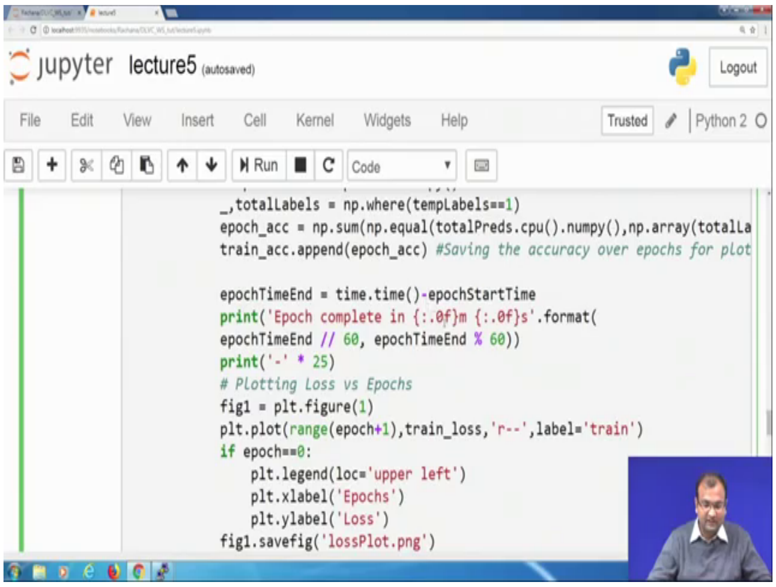
```
            totalLoss = totalLoss/batch
            totalLoss.backward()

            # Updating the model parameters
            for f in model.parameters():
                f.data.sub_(f.grad.data * learning_rate)

        epoch_loss = running_loss/50000  #Total loss for one epoch
        train_loss.append(epoch_loss) #Saving the loss over epochs for plotti
        print('Epoch loss='+str(epoch_loss))
        # Accuracy per epoch
        tempLabels = tempLabels.numpy()
        _,totalLabels = np.where(tempLabels==1)
        epoch_acc = np.sum(np.equal(totalPreds.cpu().numpy(),
        train_acc.append(epoch_acc) #Saving the accuracy over
```

Now, whenever we write down this total loss dot backward that is the derivative of my loss function which is calculated. So, my Del Del w of jw in order for this to be calculated, I need a like basically j first derivative of my cost function if you remember so, that first derivative of my cost function is what is also called as a backward operator over here. Now once that is done the next part is that I need to get down my network parameters and keep on updating my network parameters as well.

So, in my network parameters I write down another loop over here which will basically be counting down all my model parameters. So, model parameters is a function which gives me all the free power in a model or basically all the weights which are present within my neural network which can be updated. So, within your neural network the only thing which can be updated are basically the weights, you cannot update anything related to which is this non-linearity present over there in any way.

So, since you do not have any updation on to be done with the non-linearity, so the only trainable parameter left down for you is just the weights. Now what we need to get is once you get down pointers to each of these weights coming down, next is you need to find out what is your too so we have calculated our training loss and everything going down over there this is just a small script to print that at any given point of time.

(Refer Slide Time: 25:38)

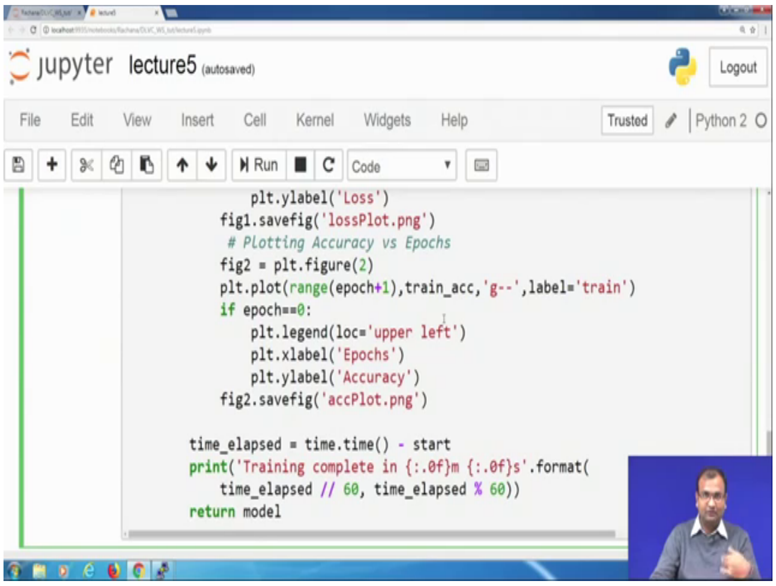

jupyter lecture5 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

```
_,totalLabels = np.where(tempLabels==1)
epoch_acc = np.sum(np.equal(totalPreds.cpu().numpy(),np.array(totalLa
train_acc.append(epoch_acc) #Saving the accuracy over epochs for plot

epochTimeEnd = time.time()-epochStartTime
print('Epoch complete in {:.0f}m {:.0f}s'.format(
epochTimeEnd // 60, epochTimeEnd % 60))
print('-' * 25)
# Plotting Loss vs Epochs
fig1 = plt.figure(1)
plt.plot(range(epoch+1),train_loss,'r--',label='train')
if epoch==0:
    plt.legend(loc='upper left')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
fig1.savefig('lossPlot.png')
```

The next is that you will have to basically update each of them. So, for updating you will be running down through the rest of it, and then as you keep on going so, what we definitely find out over here is that there is so you have your feed forward happening down, and then you get your error calculated from there you will be getting your gradients calculated, you have the gradient for the loss function you have your gradient for the network everything done.

With these you can now update all of your weights which goes within your update routine over there. Once the update is over the next part which remains for us is just to keep on looking into the errors. So, initially we will start with a very naive approach of just trying to look into the errors and see whether the error is coming down and maybe just plot it down.

(Refer Slide Time: 26:21)

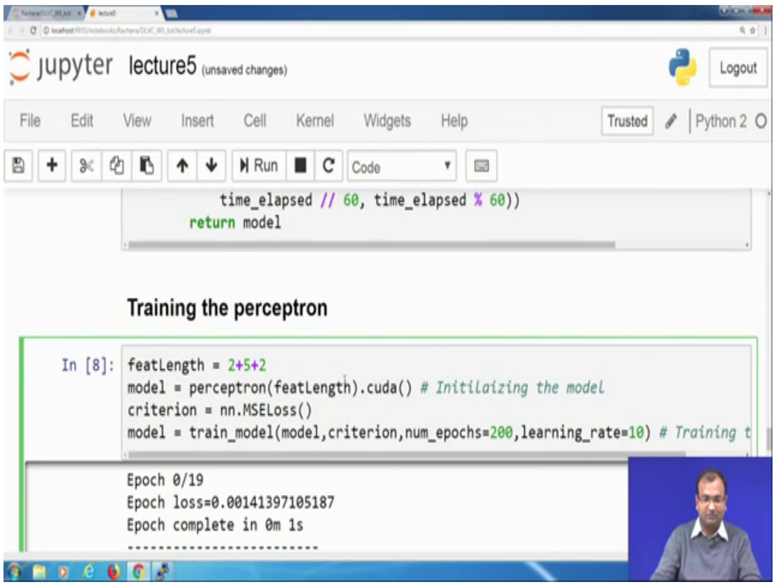


```
            plt.ylabel('Loss')
        fig1.savefig('lossPlot.png')
         # Plotting Accuracy vs Epochs
        fig2 = plt.figure(2)
        plt.plot(range(epoch+1),train_acc,'g--',label='train')
        if epoch==0:
            plt.legend(loc='upper left')
            plt.xlabel('Epochs')
            plt.ylabel('Accuracy')
        fig2.savefig('accPlot.png')

    time_elapsed = time.time() - start
    print('Training complete in {:.0f}m {:.0f}s'.format(
        time_elapsed // 60, time_elapsed % 60))
    return model
```

But as we keep on going I will be coming toward to a point for very deep deeper networks well you take on for a longer duration of time. As to what our strategies and then how do you come down with some way where just by looking into errors you can come down to a point where, you can just converge and stop and not take them any further.

So, let us just run it down. So, what I have done is so, this is the function which gets defined and once that is done we need to start with the training.

(Refer Slide Time: 26:48)



```
        time_elapsed // 60, time_elapsed % 60))
    return model
```

Training the perceptron

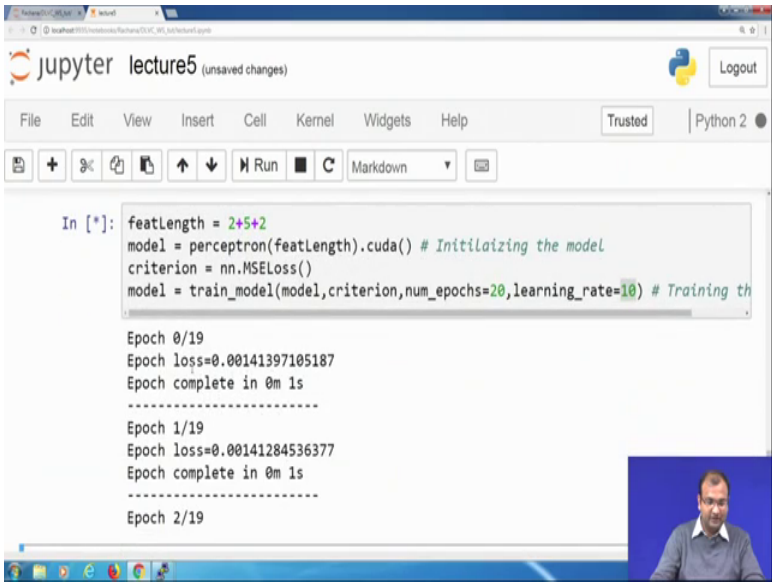
```
In [8]: featLength = 2+5+2
        model = perceptron(featLength).cuda() # Initilaizing the model
        criterion = nn.MSELoss()
        model = train_model(model,criterion,num_epochs=200,learning_rate=10) # Training t
```

```
Epoch 0/19
Epoch loss=0.00141397105187
Epoch complete in 0m 1s
------------------------
```

Now, in the training part over there. So, this is your train model function which we had defined over here so, now that the whole complex procedure is now just a function call for me. So, over here I need to put down my model which comes down from a perceptron which is defined over here, that exists on my cuda because I have a GPU running with me, then I have my criterion which is defined over here which is my mse loss or rho mean square error loss, the number of epochs I am using this is for 20 epochs and the learning rate over here is kept at 10.

And so, this has been empirically optimized you can definitely play around with on 100 or 10 power minus 3 and just check around you will find like real fun around over there. So, just let us run this part.

(Refer Slide Time: 27:40)

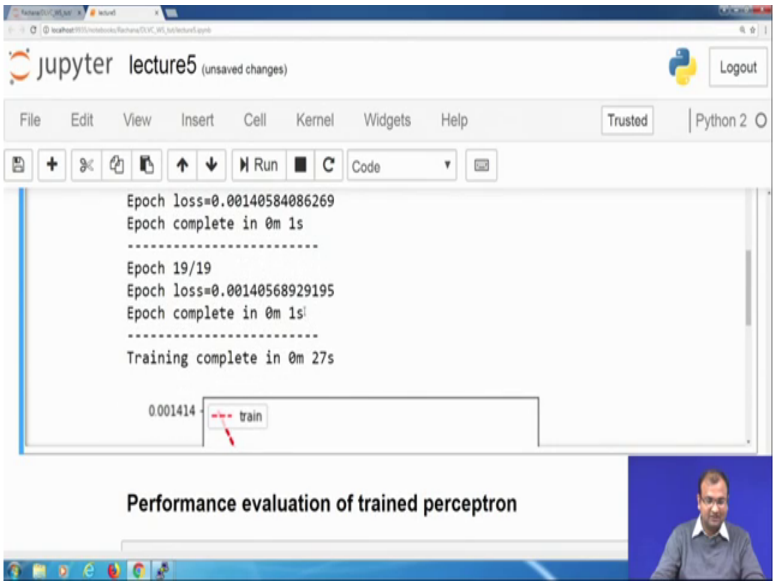

jupyter lecture5 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Markdown

```
In [*]: featLength = 2+5+2
        model = perceptron(featLength).cuda() # Initilaizing the model
        criterion = nn.MSELoss()
        model = train_model(model,criterion,num_epochs=20,learning_rate=10) # Training th

        Epoch 0/19
        Epoch loss=0.00141397105187
        Epoch complete in 0m 1s
        ------------------------
        Epoch 1/19
        Epoch loss=0.00141284536377
        Epoch complete in 0m 1s
        ------------------------
        Epoch 2/19
```

So, you can see these epochs going down, so 0 out of 19 which means the 0th epoch which has run down. So, this barely takes down about a second, and if you look into these errors over there, somewhere around this point is where the error is changing. So, it is changing at a much slower rate really slow and tardy rate.

But you see you can clearly see that these errors do keep on changing, and so, going at the rate of 1 second it is it is somewhere around 20 second that.

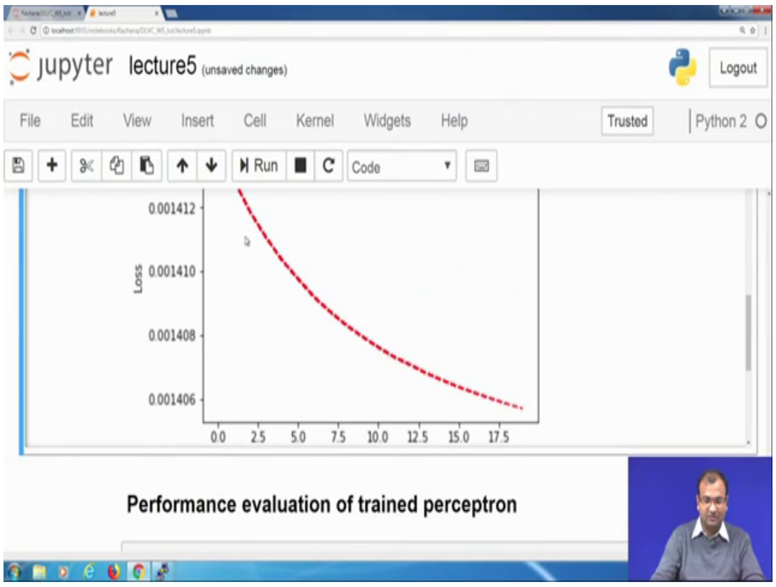
(Refer Slide Time: 28:07)


```
Epoch loss=0.00140584086269
Epoch complete in 0m 1s
------------------------
Epoch 19/19
Epoch loss=0.00140568929195
Epoch complete in 0m 1s
------------------------
Training complete in 0m 27s
```
jupyter lecture5 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code
0.001414
train

Performance evaluation of trained perceptron

So, the total thing takes maximum of 27 seconds in which and this is how the rate is going down.

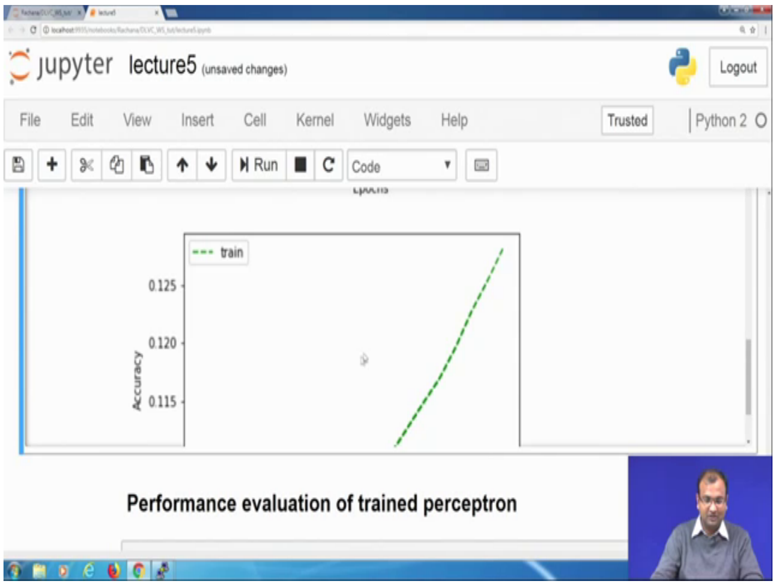
(Refer Slide Time: 28:12)



Performance evaluation of trained perceptron

Now, clearly going down by this I do know that this will keep on continuing further and go down. So, please keep on putting your errors as towards a large number of epochs.

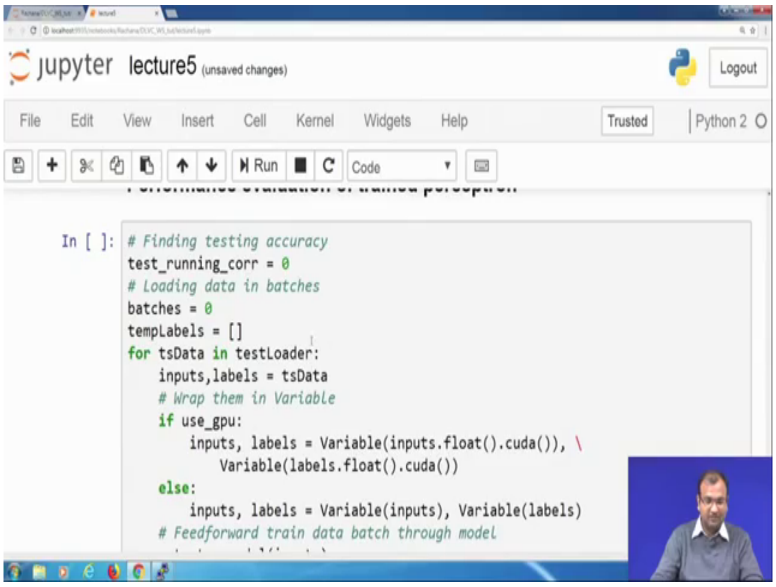
(Refer Slide Time: 28:28)


jupyter lecture5 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code
train
0.125
0.120
0.115
Accuracy

Performance evaluation of trained perceptron

Now, this is the other side of it where we are looking into the accuracy plot, and if you clearly see as the error is decreasing your accuracy is definitely increasing over there, but then this accuracy is not that high. So, it is just a really it will 0.5 percent on your accuracy side over there.

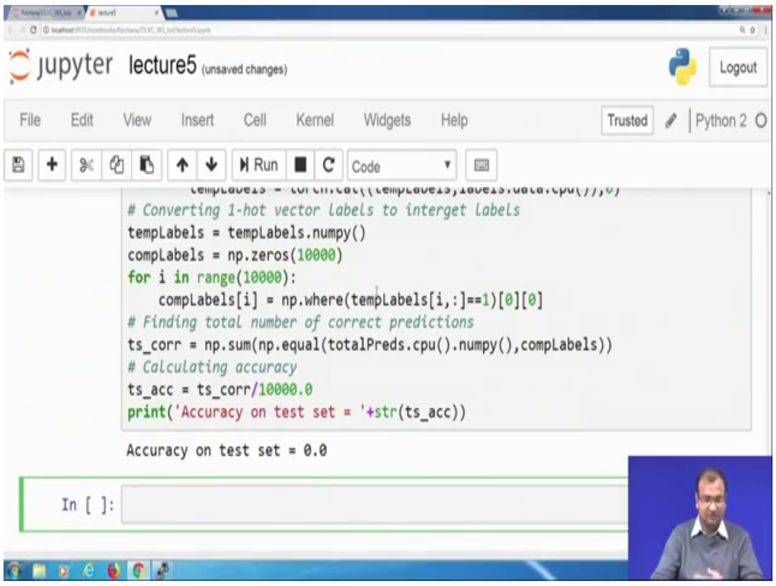
(Refer Slide Time: 28:40)


jupyter lecture5 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

```
In [ ]: # Finding testing accuracy
        test_running_corr = 0
        # Loading data in batches
        batches = 0
        tempLabels = []
        for tsData in testLoader:
            inputs,labels = tsData
            # Wrap them in Variable
            if use_gpu:
                inputs, labels = Variable(inputs.float().cuda()), \
                    Variable(labels.float().cuda())
            else:
                inputs, labels = Variable(inputs), Variable(labels)
            # Feedforward train data batch through model
```

So, once that is done that is on your training part of it the next part is to look down into your accuracy over your test set unfortunately with all though it had our 12.5 percent accuracy on the training set on the test set it just unfortunately came down as 0.

(Refer Slide Time: 28:50)



```
# Converting 1-hot vector labels to interget labels
templabels = templabels.numpy()
compLabels = np.zeros(10000)
for i in range(10000):
    compLabels[i] = np.where(tempLabels[i,:]==1)[0][0]
# Finding total number of correct predictions
ts_corr = np.sum(np.equal(totalPreds.cpu().numpy(),compLabels))
# Calculating accuracy
ts_acc = ts_corr/10000.0
print('Accuracy on test set = '+str(ts_acc))
```

```
Accuracy on test set = 0.0
```

So, if you keep on running this for a longer time. So, we Know that from our experiences if we were training this for say about 200 epochs or so, you do come to a point where you can have about 10 12 percent of accuracy very easily coming down. So, I will leave these exercises more off to you to explore, and have fun along with that and we then next week we will be continuing with multi-layer perceptron, and having much more exercises in to understanding multi-layer perceptron, as well so with that stay tuned.

Thanks.