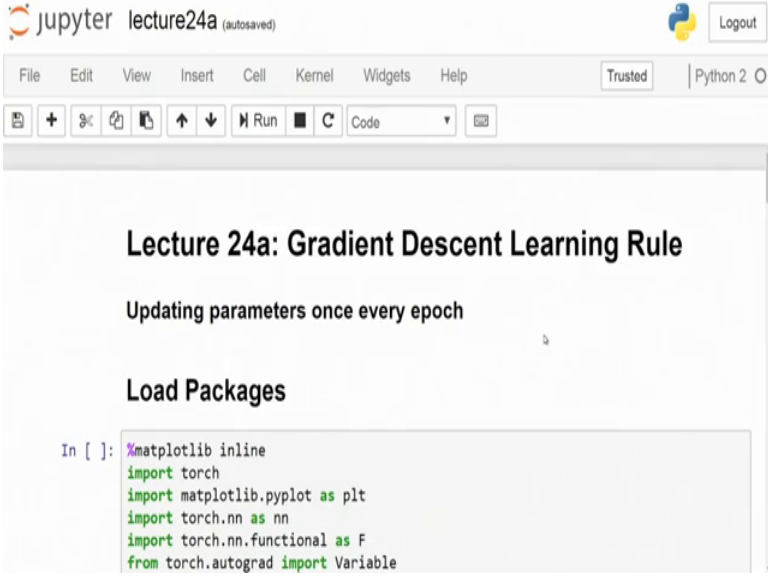


Deep Learning for Visual Computing
Prof. Debdoot Sheet
Department of Electrical Engineering
Indian Institute of Technology, Kharagpur

Lecture - 24
Gradient Descent Learning Rule

Welcome. So, in the last lecture, we have been going around with how to update down different networks.

(Refer Slide Time: 00:23)



```
jupyter lecture24a (autosaved) Logout
File Edit View Insert Cell Kernel Widgets Help Trusted Python 2 O
+ %< > Run C Code
Lecture 24a: Gradient Descent Learning Rule
Updating parameters once every epoch
Load Packages
In [ ]: %matplotlib inline
import torch
import matplotlib.pyplot as plt
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
```

And, there are 3 different kinds of updates which you can do is what I was speaking around. So, today I am going to actually demonstrate out on how these 3 different kinds of updates rule in a neural network would work. So, one of them is where you have all the gradients accumulated at the end of the network and then within each epoch. So, you keep on sending all the samples through it, all the samples whichever you have in your training data and then you sum up all the errors over there and then you take a gradient of the error and you back propagate throughout the network, that is the way of how I was explaining you the whole concept of back propagation in its own way.

There can be another way, where say you are not able to handle that much of data. So, say that your training data is something like 100 GB in size your system RAM is limited to some 8 GB or 4 GB. You will never be able to actually put down the whole thing and your model say it consumes about a few 100 MBs over there your system voice and

everything is consuming and out of your 8 GBs you are just left with a viable of 5 GB that is the maximum what you can do.

Now, if you load down your data, the maximum data you can load is 5 GB never 100 GB at a point of time and in that case we came up with something which is called as a batch learning rule in which in the idea was that you divide this whole set of your training examples into smaller subsets each of them is called as a batch, you feed within each epoch you are going to feed down all the batches. However, you feed one batch you get your error now you can back propagate that one through the network then you feed the next batch you get your error back provided through the network. And, the reason why we cannot just accumulate out everything and do a back propagation is because you have an response of the each layer on your output which is also multiplied down with the gradient of the error coming down. So, that was how our gradient descent was derived.

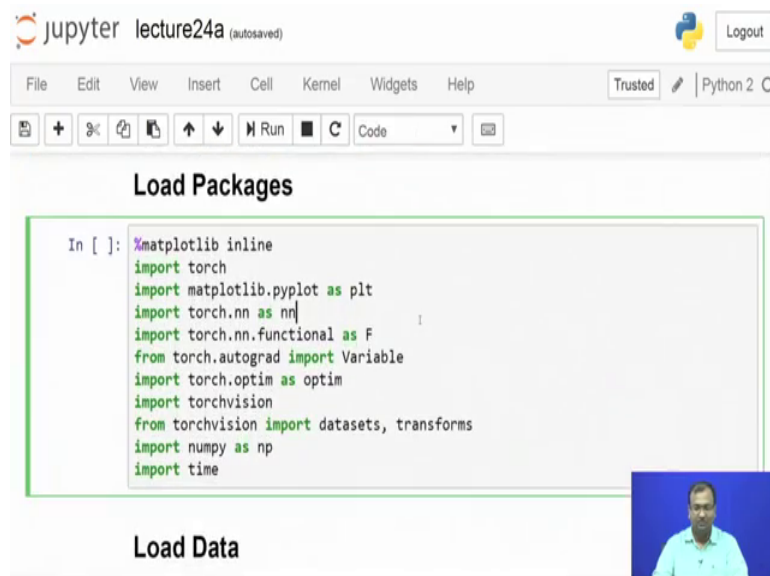
Now, since you need to preserve each and every instances output from the network for updating what is inside the network of the weights over there, so, you can use only just a finite number of samples and you cannot just somehow accumulate over the samples over there. So, this kind of a thing is what is called as a batch update rule and the other one is plain and simple vanilla update rules in which you put down you put a sample you get it is error you back propagate it out, then you put the next sample you get it is error back propagate and then what this would mean is that say you have some 10000 samples or 60000 samples as an MNIST in your training. So, within each epoch you will have sixty thousand times of an update of the network which is really large from a computational stand point. So, you will have to compute out gradient for every single sample that will be computed 60000 times per epoch and you will do it.

On the other side, if you compare it with the first one where I am just going to update the whole network at the end of every epoch it is just once then I am going to calculate out the gradient and it is once that I am going to do this back propagation and update over there. So, this is a plain difference which comes out and in fact, this has a very significant role when you look at the amount of time it is going to incur.

So, let us get down with each of them. So, how we have divided is we actually have 3 sub parts of this lecture number 24 which is kept on the jit release for us. So, 24a is to do

with updating parameters once every epoch. So, this is a scratch pad written down for your own understanding of how we change down, once we can do that every epoch.

(Refer Slide Time: 03:38)



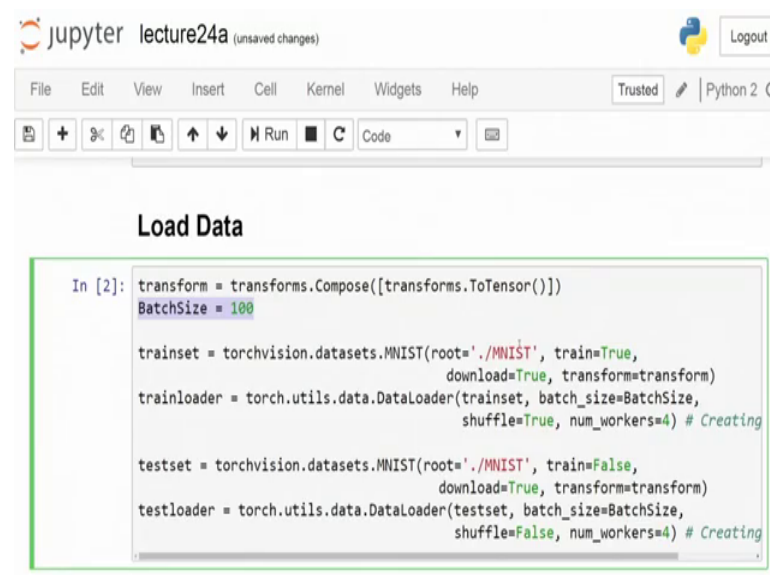
The screenshot shows a Jupyter Notebook window titled 'lecture24a (autosaved)'. The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for file operations and execution. The code cell is titled 'Load Packages' and contains the following Python code:

```
In [ ]: %matplotlib inline
import torch
import matplotlib.pyplot as plt
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
import torch.optim as optim
import torchvision
from torchvision import datasets, transforms
import numpy as np
import time
```

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

So, our customary first part of the ceremony of running down any code is just to get down your libraries which are there on the header.

(Refer Slide Time: 03:51)



The screenshot shows a Jupyter Notebook window titled 'lecture24a (unsaved changes)'. 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 cell is titled 'Load Data' and contains the following Python code:

```
In [2]: transform = transforms.Compose([transforms.ToTensor()])
BatchSize = 100

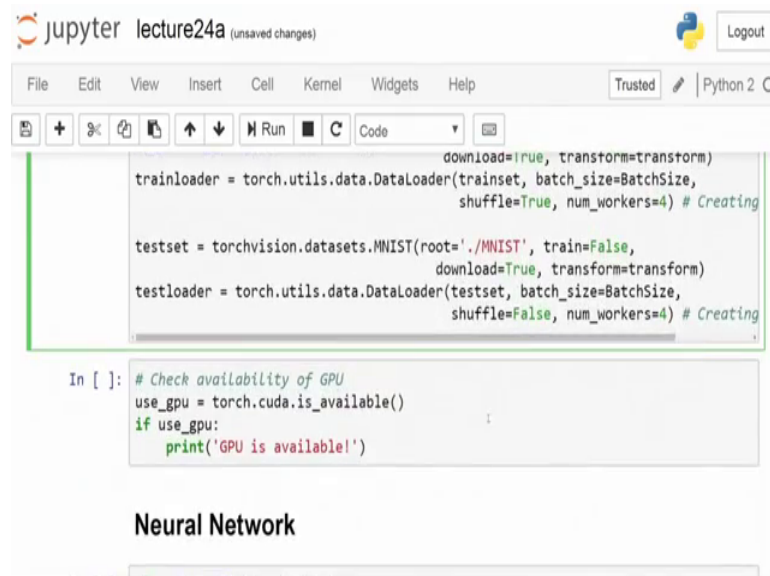
trainset = torchvision.datasets.MNIST(root='./MNIST', train=True,
                                     download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=BatchSize,
                                          shuffle=True, num_workers=4) # Creating

testset = torchvision.datasets.MNIST(root='./MNIST', train=False,
                                     download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=BatchSize,
                                         shuffle=False, num_workers=4) # Creating
```

Next, you are going to get down your data now look over here that though I defined down something called as a data within a batch size of 100 and that is just from the data

loader perspective. It does not have anything to do with my trainer perspective inside over there.

(Refer Slide Time: 04:04)



```
download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=BatchSize,
                                           shuffle=True, num_workers=4) # Creating

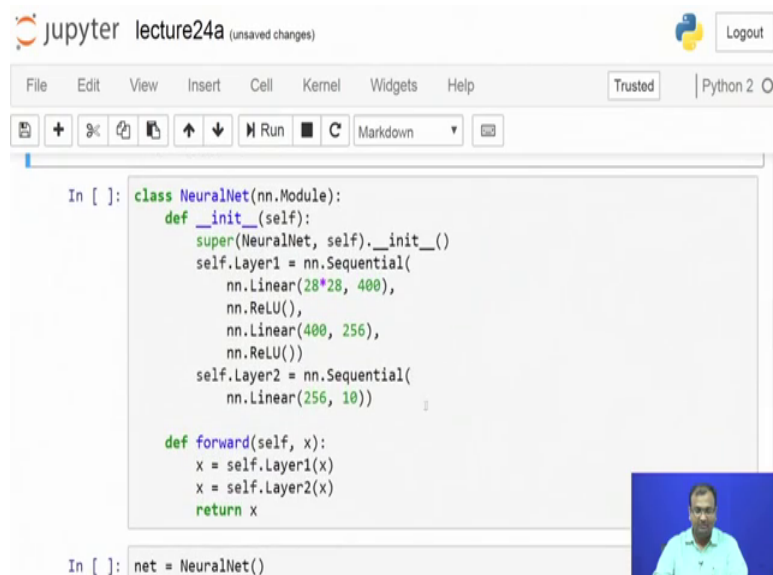
testset = torchvision.datasets.MNIST(root='./MNIST', train=False,
                                     download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=BatchSize,
                                         shuffle=False, num_workers=4) # Creating

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

Neural Network

Now, I checked down my GPU if that is available let us well and good.

(Refer Slide Time: 04:10)



```
In [ ]: class NeuralNet(nn.Module):
def __init__(self):
    super(NeuralNet, self).__init__()
    self.Layer1 = nn.Sequential(
        nn.Linear(28*28, 400),
        nn.ReLU(),
        nn.Linear(400, 256),
        nn.ReLU())
    self.Layer2 = nn.Sequential(
        nn.Linear(256, 10))

def forward(self, x):
    x = self.Layer1(x)
    x = self.Layer2(x)
    return x

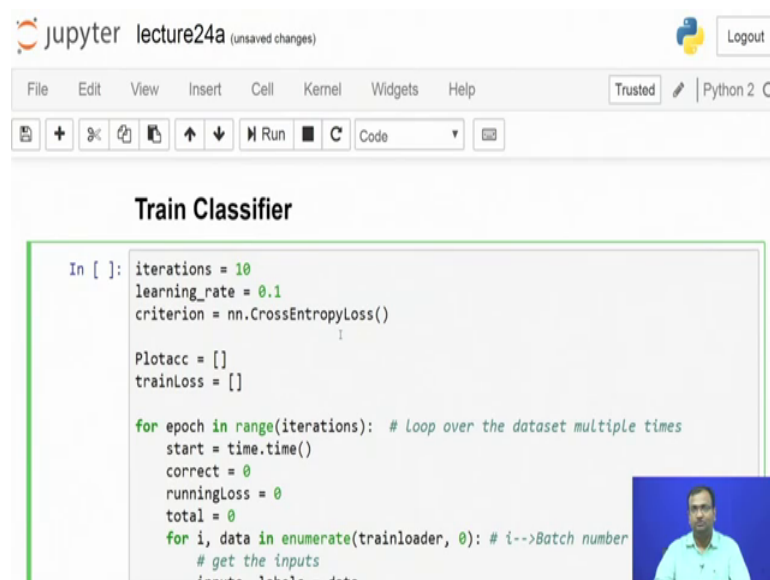
In [ ]: net = NeuralNet()
```

And, then now, I define my network and this is the plain valley network as we were using down in the earlier example for doing for showing you different kind of cost functions and the dynamics. So, here also it is the same one. So, you have a 28 cross 28

batch of your 110 digits in MNIST which amounts to 700 and 84 neurons. They are connected down to 400 neurons, these are connected down to 256 neurons and this part of it which is your layer one is what constitutes the feature discovery part of the layer with 2 different hidden neural network layers and that is connected down to the final classification which match from 256 neurons on to just simple 10 neurons and then this completes down my a complete network.

Now, on the forward pass of the network what I need to do is given any input to layer 1, I am going to get down certain output and then I feed that to my layer 2 definition over here which is my classification part of the wing and I get my output coming down over here and this defines my plain simple neural net.

(Refer Slide Time: 05:19)



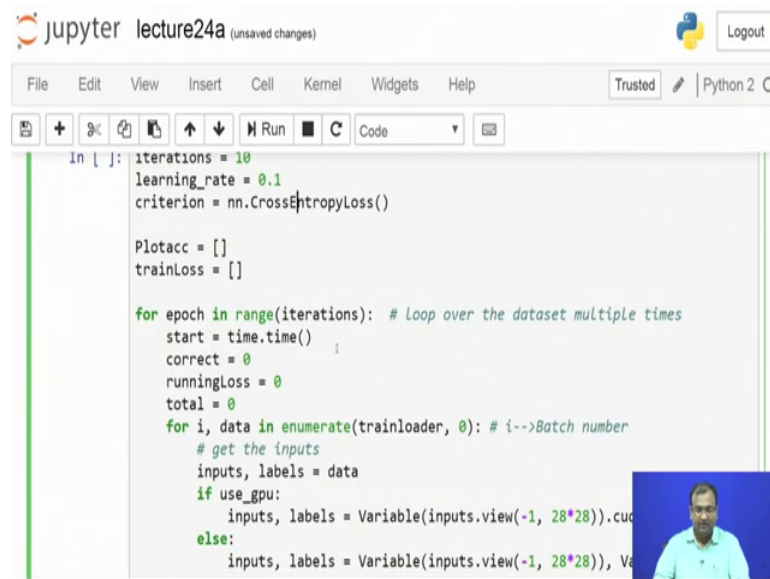
```
In [ ]: iterations = 10
learning_rate = 0.1
criterion = nn.CrossEntropyLoss()

Plotacc = []
trainLoss = []

for epoch in range(iterations): # Loop over the dataset multiple times
    start = time.time()
    correct = 0
    runningLoss = 0
    total = 0
    for i, data in enumerate(trainloader, 0): # i-->Batch number
        # get the inputs
        inputs, labels = data
```

Now, if I have a GPU available then let us just convert it onto a GPU array. So, that it is compatible while running it out on a GPU and then we start down our simple classified training routine over there. Now, in the classified training routine I choose to just do with 10 epochs, this is just to keep it plain, short, sweet and simple I do not have any issues, a learning rate of 0.1 and the criterion over here because it is a classification problem which we are just trying to do we are just going to stick down with the cross entropy loss. Now, you can use a negative log, likelihood loss you can use a multi margin loss, anything as you would on, but we are just going to stick down to simple cross entropy or binary cross entropy loss over here.

(Refer Slide Time: 05:55)



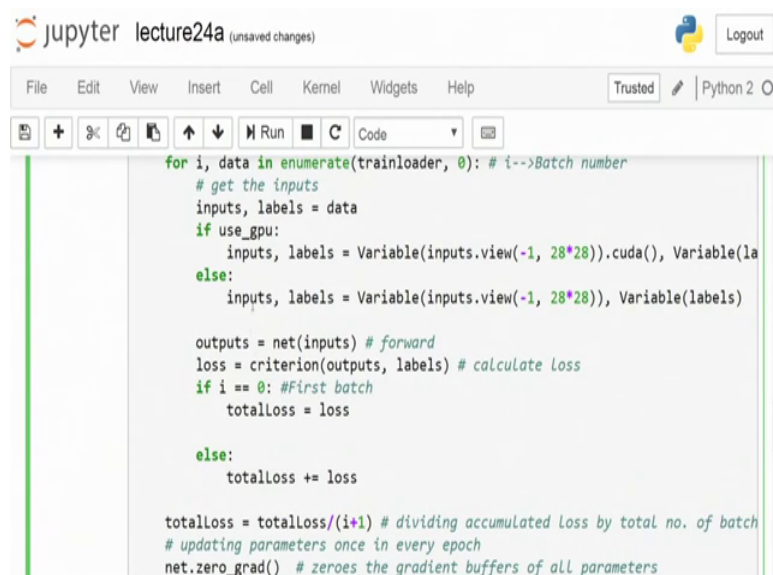
```
iterations = 10
learning_rate = 0.1
criterion = nn.CrossEntropyLoss()

Plotacc = []
trainLoss = []

for epoch in range(iterations): # Loop over the dataset multiple times
    start = time.time()
    correct = 0
    runningLoss = 0
    total = 0
    for i, data in enumerate(trainloader, 0): # i-->Batch number
        # get the inputs
        inputs, labels = data
        if use_gpu:
            inputs, labels = Variable(inputs.view(-1, 28*28)).cuda()
        else:
            inputs, labels = Variable(inputs.view(-1, 28*28)), Variable(labels)
```

So, now, within my iteration of over epochs, it is going to iterate over a lot of epochs and for me it is going to be just any epochs over which I am going to iterate it out. Now, over here what I am going to do is one is just starting timer to keep a tab of how long it takes to execute each of them and that is to show you exactly what is the difference and trade off you get down while shifting from one to the other.

(Refer Slide Time: 06:28)



```
for i, data in enumerate(trainloader, 0): # i-->Batch number
    # get the inputs
    inputs, labels = data
    if use_gpu:
        inputs, labels = Variable(inputs.view(-1, 28*28)).cuda(), Variable(labels)
    else:
        inputs, labels = Variable(inputs.view(-1, 28*28)), Variable(labels)

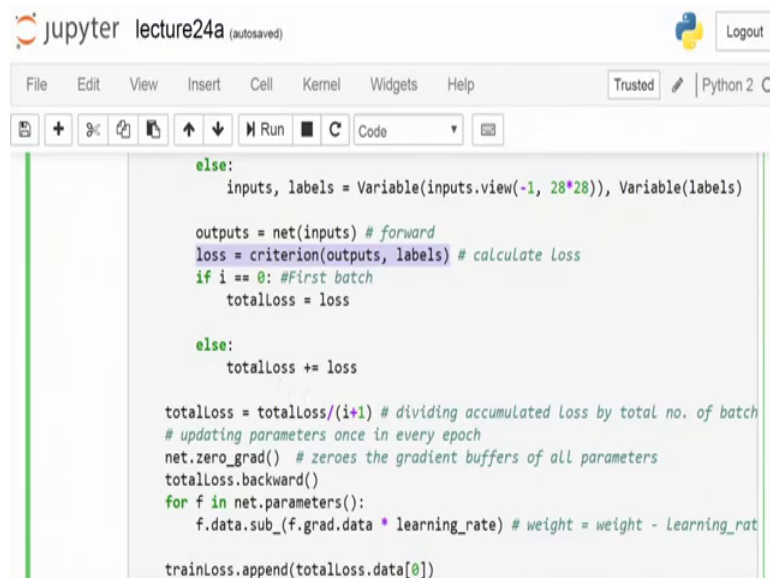
    outputs = net(inputs) # forward
    loss = criterion(outputs, labels) # calculate loss
    if i == 0: #First batch
        totalLoss = loss
    else:
        totalLoss += loss

    totalLoss = totalLoss/(i+1) # dividing accumulated loss by total no. of batch
    # updating parameters once in every epoch
    net.zero_grad() # zeroes the gradient buffers of all parameters
```

Now, here the first part is basically create down your data loader such that it can keep on loading all the data and then you in case you have your GPU available then we are just

going to convert my inputs and all the labels into a GPU compatible array and then you do a forward pass over the network. Now, within each epoch once you have a forward pass of the network you get certain output coming down, over that you need to find out your loss now once your loss comes down over there.

(Refer Slide Time: 06:50)



```
jupyter lecture24a (autosaved) Python 2 O
File Edit View Insert Cell Kernel Widgets Help Trusted Python 2 O
else:
    inputs, labels = Variable(inputs.view(-1, 28*28)), Variable(labels)

    outputs = net(inputs) # forward
    loss = criterion(outputs, labels) # calculate loss
    if i == 0: #First batch
        totalLoss = loss

    else:
        totalLoss += loss

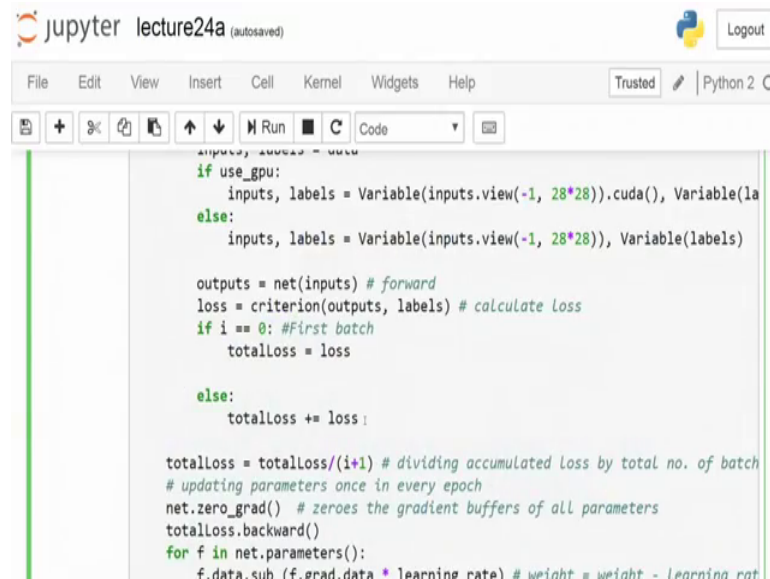
totalLoss = totalLoss/(i+1) # dividing accumulated loss by total no. of batch
# updating parameters once in every epoch
net.zero_grad() # zeroes the gradient buffers of all parameters
totalLoss.backward()
for f in net.parameters():
    f.data.sub_(f.grad.data * learning_rate) # weight = weight - Learning_rat

trainLoss.append(totalLoss.data[0])
```

Now, what you need to actually understand is that it is going to run over all the batches. So, in your batch loader what you have done is that it is going to fetch down some 100 images or something. So, let us go down yeah. So, your batch size is 100, it means that technically it is going to fetch down 100 images in one burst from your hard drive and now, what I am going to do is getting back over here I am actually going to cumulate my errors over all such ones which are fetched out. So, technically on my training data set within MNIST you have 60000 samples, I am fetching down 100 at a time.

So, that means, that I will have 600 such fetch operations within one single epoch in order to get all my data set coming down from my hard drive side over there.

(Refer Slide Time: 07:40)



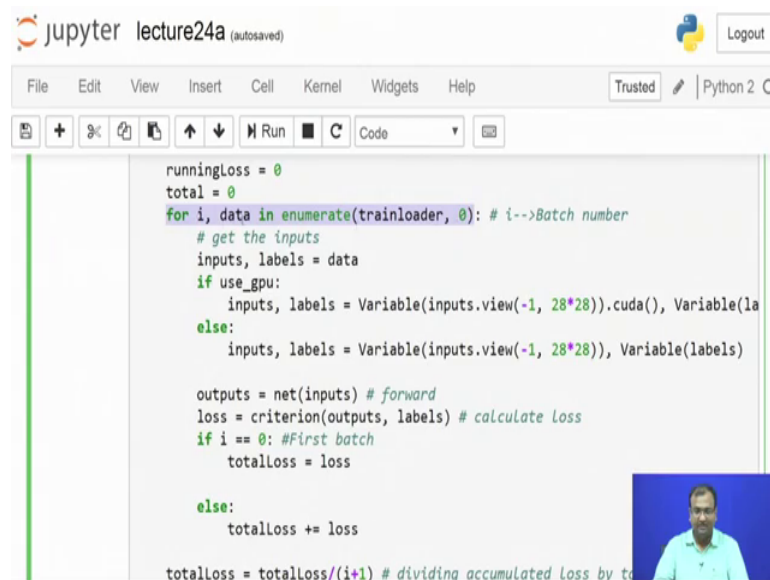
```
inputs, labels = data
if use_gpu:
    inputs, labels = Variable(inputs.view(-1, 28*28)).cuda(), Variable(labels)
else:
    inputs, labels = Variable(inputs.view(-1, 28*28)), Variable(labels)

outputs = net(inputs) # forward
loss = criterion(outputs, labels) # calculate loss
if i == 0: #First batch
    totalLoss = loss
else:
    totalLoss += loss

totalLoss = totalLoss/(i+1) # dividing accumulated loss by total no. of batch
# updating parameters once in every epoch
net.zero_grad() # zeroes the gradient buffers of all parameters
totalLoss.backward()
for f in net.parameters():
    f.data.sub_(f.grad.data * learning_rate) # weight = weight - learning rate
```

So, when I am calculating out the loss, I need to accumulate all of these losses. So, that I get known the total error at the end of the epoch itself and that is what I am just doing it over here. Now, the point which goes down is that I need to find out what is the total loss at the end of being able to load down all of these ones.

(Refer Slide Time: 08:08)



```
runningLoss = 0
total = 0
for i, data in enumerate(trainloader, 0): # i-->Batch number
    # get the inputs
    inputs, labels = data
    if use_gpu:
        inputs, labels = Variable(inputs.view(-1, 28*28)).cuda(), Variable(labels)
    else:
        inputs, labels = Variable(inputs.view(-1, 28*28)), Variable(labels)

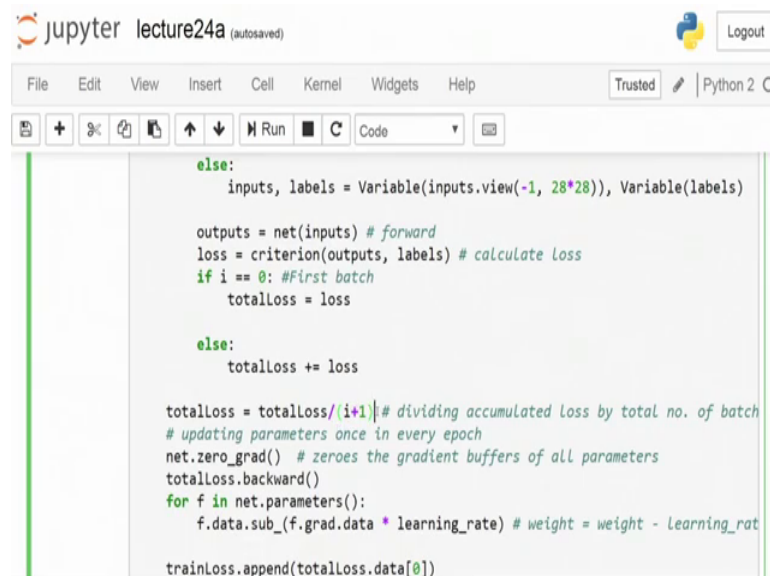
    outputs = net(inputs) # forward
    loss = criterion(outputs, labels) # calculate loss
    if i == 0: #First batch
        totalLoss = loss
    else:
        totalLoss += loss

totalLoss = totalLoss/(i+1) # dividing accumulated loss by total no. of batch
```

And, now given the fact that this i which I am using over here this is a variable commodity. So, for me it comes down that this will be iterating from 0 till 599 and that is

equal to 600 over there and it depends on how many number of batch fetches you had over there. So, that is why I just divided down by i plus 1.

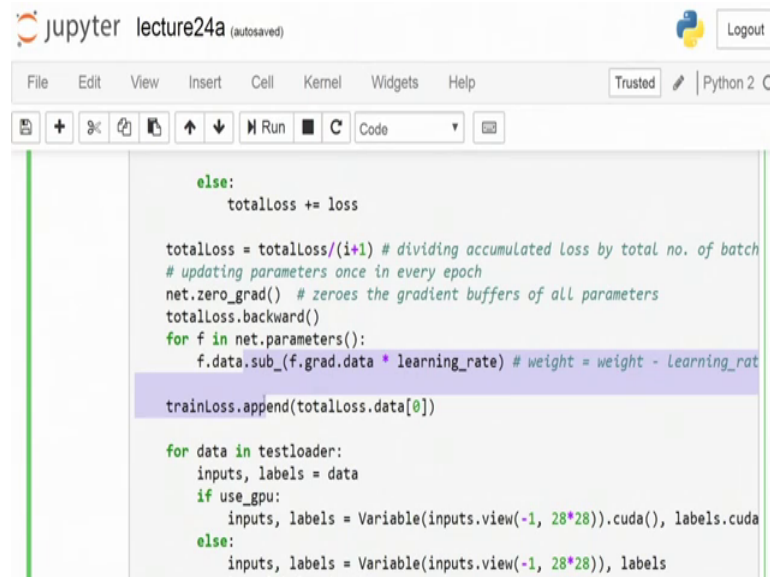
(Refer Slide Time: 08:16)



```
jupyter lecture24a (autosaved)
File Edit View Insert Cell Kernel Widgets Help Trusted Python 2
else:
    inputs, labels = Variable(inputs.view(-1, 28*28)), Variable(labels)
    outputs = net(inputs) # forward
    loss = criterion(outputs, labels) # calculate loss
    if i == 0: #First batch
        totalLoss = loss
    else:
        totalLoss += loss
totalLoss = totalLoss/(i+1) # dividing accumulated loss by total no. of batch
# updating parameters once in every epoch
net.zero_grad() # zeroes the gradient buffers of all parameters
totalLoss.backward()
for f in net.parameters():
    f.data.sub_(f.grad.data * learning_rate) # weight = weight - learning_rate
trainLoss.append(totalLoss.data[0])
```

Now, this plus 1 comes down from the factor that i is a number which starts from a zero indexing. So, the highest number which it would get down is 599, in order to get down the total size or the total number of numbers between the range of 0 to 599, that is the last value plus 1. So, that is a simple argument why you have an i plus 1 written over there. Now, after that you zero down your gradients then you have a backward on the total loss which is taking down nabla of the cost function itself.

(Refer Slide Time: 08:56)



```
else:
    totalLoss += loss

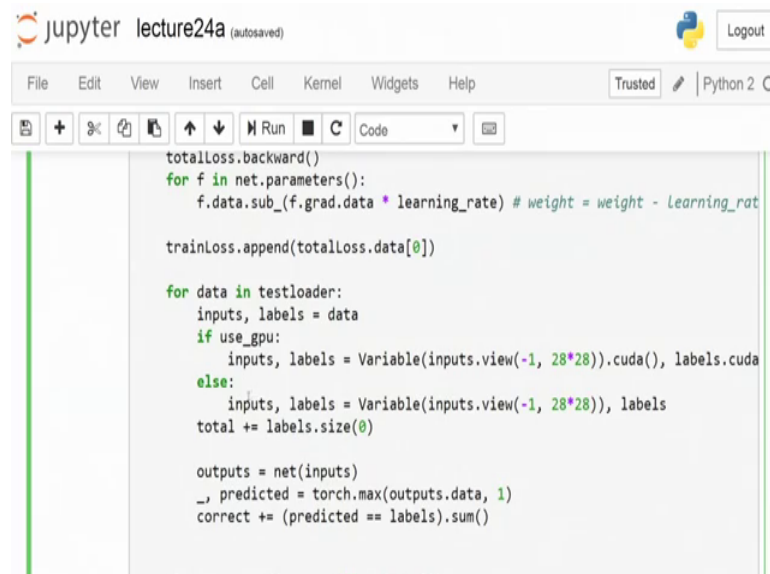
totalLoss = totalLoss/(i+1) # dividing accumulated loss by total no. of batch
# updating parameters once in every epoch
net.zero_grad() # zeroes the gradient buffers of all parameters
totalLoss.backward()
for f in net.parameters():
    f.data.sub_(f.grad.data * learning_rate) # weight = weight - Learning_rate

trainLoss.append(totalLoss.data[0])

for data in testloader:
    inputs, labels = data
    if use_gpu:
        inputs, labels = Variable(inputs.view(-1, 28*28)).cuda(), labels.cuda()
    else:
        inputs, labels = Variable(inputs.view(-1, 28*28)), labels
```

Now, once you have this derivative of the cost function taken down and what we will be doing done is actually to do an update for following down the vanilla gradient descent approach over there and that is typically what we do.

(Refer Slide Time: 09:04)



```
totalLoss.backward()
for f in net.parameters():
    f.data.sub_(f.grad.data * learning_rate) # weight = weight - Learning_rate

trainLoss.append(totalLoss.data[0])

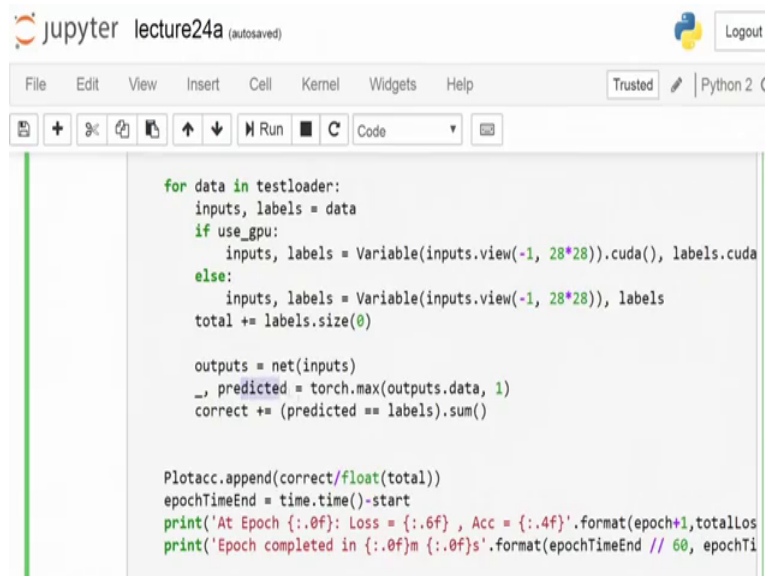
for data in testloader:
    inputs, labels = data
    if use_gpu:
        inputs, labels = Variable(inputs.view(-1, 28*28)).cuda(), labels.cuda()
    else:
        inputs, labels = Variable(inputs.view(-1, 28*28)), labels
    total += labels.size(0)

    outputs = net(inputs)
    _, predicted = torch.max(outputs.data, 1)
    correct += (predicted == labels).sum()
```

Now, once we have this whole thing updated over here, my whole purpose is that let us see what sorts the validation which comes from over there. Now, for validation we are just going to load down it on my testing data set which is 10000 examples over there. Now, for each of them I just get my output coming down provided a input to this

network, so, the network which is updated as part of this particular epoch at any given epoch whatever is the state of the network.

(Refer Slide Time: 09:32)



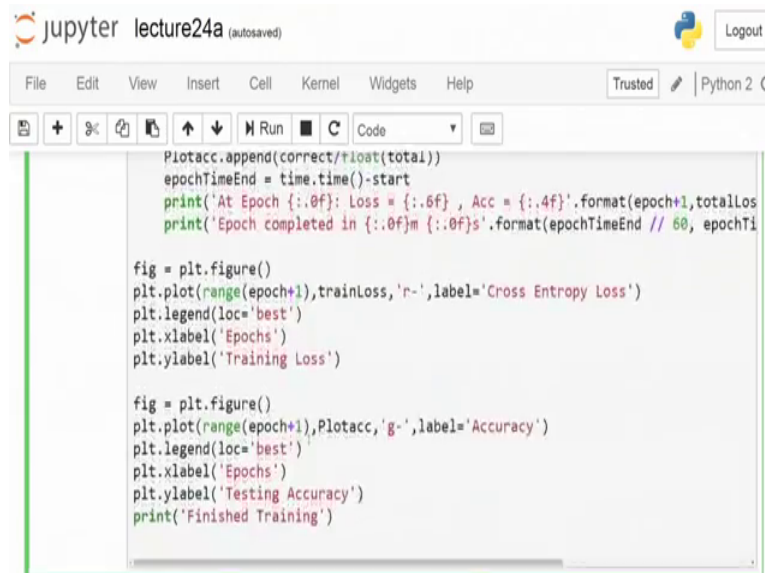
```
for data in testloader:
    inputs, labels = data
    if use_gpu:
        inputs, labels = Variable(inputs.view(-1, 28*28)).cuda(), labels.cuda()
    else:
        inputs, labels = Variable(inputs.view(-1, 28*28)), labels
    total += labels.size(0)

    outputs = net(inputs)
    _, predicted = torch.max(outputs.data, 1)
    correct += (predicted == labels).sum()

Plotacc.append(correct/float(total))
epochTimeEnd = time.time()-start
print('At Epoch {:.0f}: Loss = {:.6f} , Acc = {:.4f}'.format(epoch+1,totalLoss))
print('Epoch completed in {:.0f}m {:.0f}s'.format(epochTimeEnd // 60, epochTimeEnd % 60))
```

Then we get down our predicted values and then finally, we check if the predicted value is equal to the label over there and then sum it up in order to get done the number of correctly classified samples over the whole batch.

(Refer Slide Time: 09:37)



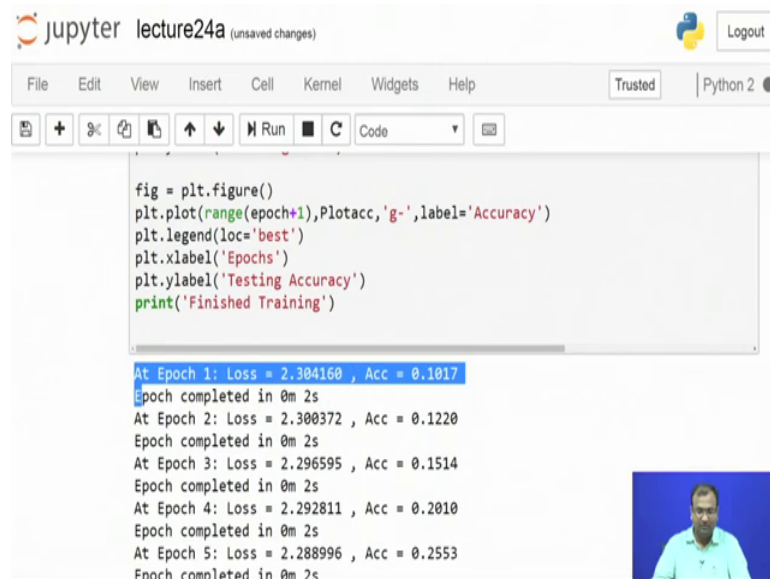
```
Plotacc.append(correct/float(total))
epochTimeEnd = time.time()-start
print('At Epoch {:.0f}: Loss = {:.6f} , Acc = {:.4f}'.format(epoch+1,totalLoss))
print('Epoch completed in {:.0f}m {:.0f}s'.format(epochTimeEnd // 60, epochTimeEnd % 60))

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

fig = plt.figure()
plt.plot(range(epoch+1),Plotacc,'g-',label='Accuracy')
plt.legend(loc='best')
plt.xlabel('Epochs')
plt.ylabel('Testing Accuracy')
print('Finished Training')
```

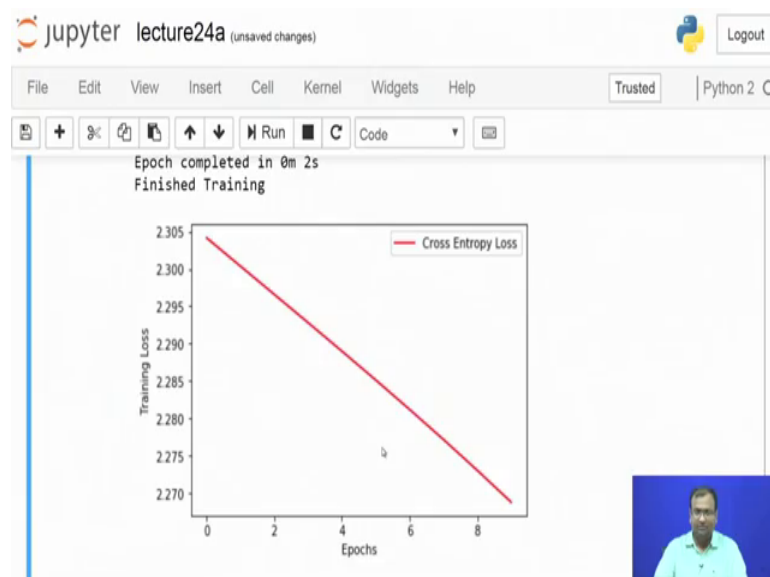
And that is what is going to run out my complete system for trying to do an update once every epoch.

(Refer Slide Time: 09:48)



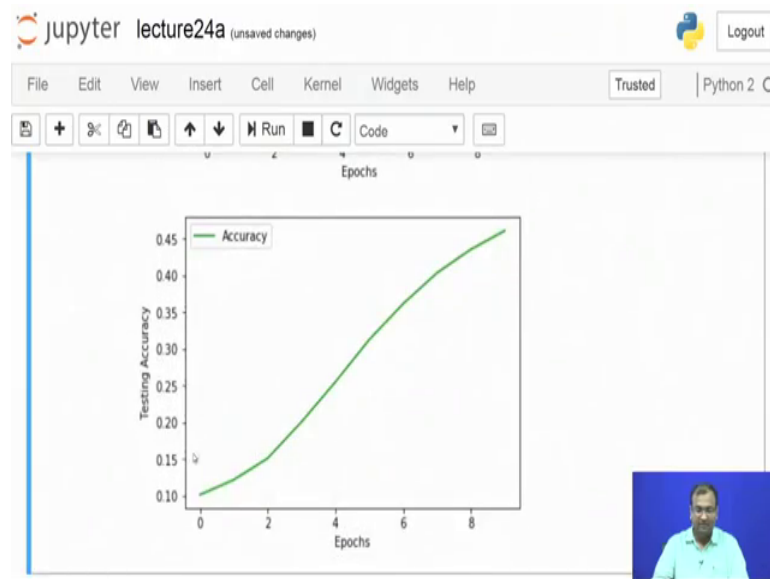
You can pretty much see how much time it takes. It takes almost like 2 seconds to complete an epoch which is pretty fast to say. There are 60000 samples which are going through it and it is getting updated at every point of thing. However, though the accuracy is not that great over there. So, somewhere around the tenth epoch we are standing at just merely 46 percent of accuracy.

(Refer Slide Time: 10:10)



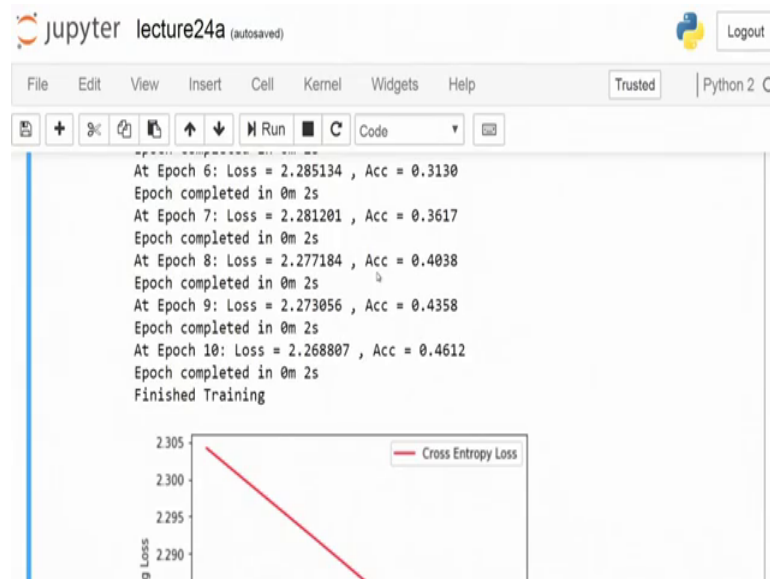
Yes, not something to be really excited at a point of time.

(Refer Slide Time: 10:14)



However given the fact that we started with a 10 percent accuracy which is perfect I mean that is a random coin toss, because you have 10 classes. If you are randomly trying to, it is something like this; you have 10 buckets and I am throwing a ball to enter into one bucket and this ball may enter into the right bucket or it may enter into a wrong bucket. So, there is one ball which has the number 1 written to it, if it there is a 1 by 10 chance that it will land into my bucket number 1. The next ball will also have a random one tenth chance that it will land up into my current bucket. So, technically your random guess accuracy will be somewhere around 10 percent whereas, once you start learning down over there, so the accuracy keeps on growing and then we figure out that it goes to somewhere around 45 percent on this case.

(Refer Slide Time: 10:58)



Now, the critical part to see is that it takes roughly 2 seconds to train every epoch and that is pretty fast over there also the loss curve which you see is quite smooth.

(Refer Slide Time: 11:19)

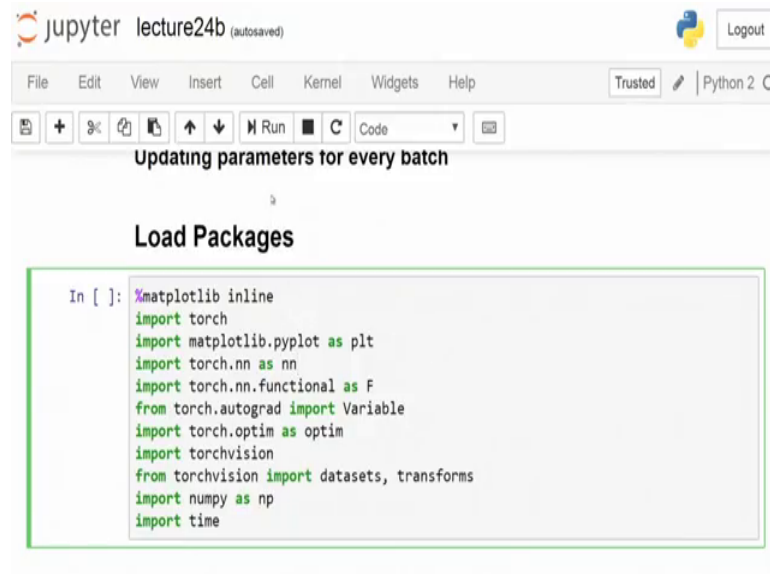
The screenshot shows a Jupyter Notebook interface for 'lecture24b'. The code cell contains the following Python code:

```
In [ ]: %matplotlib inline
import torch
import matplotlib.pyplot as plt
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
import torch.optim as optim
import torchvision
from torchvision import datasets, transforms
import numpy as np
import time
```

Below the code cell is a small video thumbnail of a person speaking.

However, given that fact there is still a challenge which we were facing down and that challenge is if I am not able to load all of those 60000 samples of data together onto my memory and then I will have to actually update it at every single point of time.

(Refer Slide Time: 11:23)

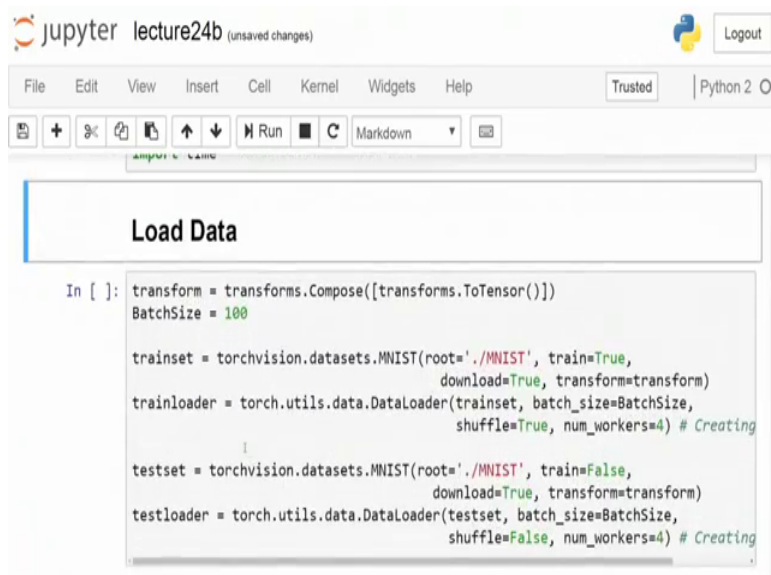


The screenshot shows a Jupyter Notebook interface with the title 'lecture24b (autosaved)'. The menu bar includes File, Edit, View, Insert, Cell, Kernel, Widgets, and Help. The status bar shows 'Trusted' and 'Python 2'. The code cell is titled 'Updating parameters for every batch' and 'Load Packages'. The code in the cell is as follows:

```
In [ ]: %matplotlib inline
import torch
import matplotlib.pyplot as plt
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
import torch.optim as optim
import torchvision
from torchvision import datasets, transforms
import numpy as np
import time
```

And, that is where this batch learning rule comes down to play.

(Refer Slide Time: 11:26)



The screenshot shows a Jupyter Notebook interface with the title 'lecture24b (unsaved changes)'. The menu bar includes File, Edit, View, Insert, Cell, Kernel, Widgets, and Help. The status bar shows 'Trusted' and 'Python 2'. The code cell is titled 'Load Data'. The code in the cell is as follows:

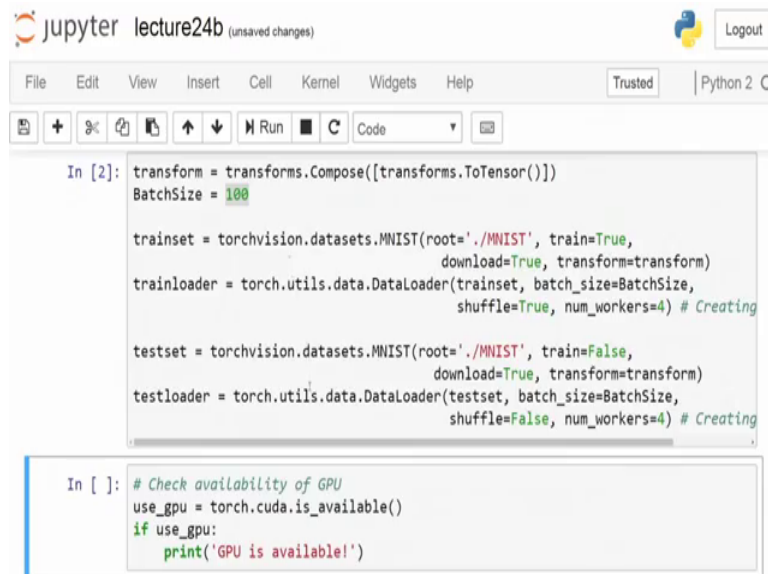
```
In [ ]: transform = transforms.Compose([transforms.ToTensor()])
BatchSize = 100

trainset = torchvision.datasets.MNIST(root='./MNIST', train=True,
                                     download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=BatchSize,
                                          shuffle=True, num_workers=4) # Creating

testset = torchvision.datasets.MNIST(root='./MNIST', train=False,
                                     download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=BatchSize,
                                         shuffle=False, num_workers=4) # Creating
```

Now, batch learning over here what we will be doing is you have a batch size which is set down as 100, it means that every time you are fetching down some 100 images at one single shot from the drive and then using it.

(Refer Slide Time: 11:36)



```
lecture24b (unsaved changes) Python 2.0
File Edit View Insert Cell Kernel Widgets Help Trusted Python 2.0
Run Code
In [2]: transform = transforms.Compose([transforms.ToTensor()])
BatchSize = 100

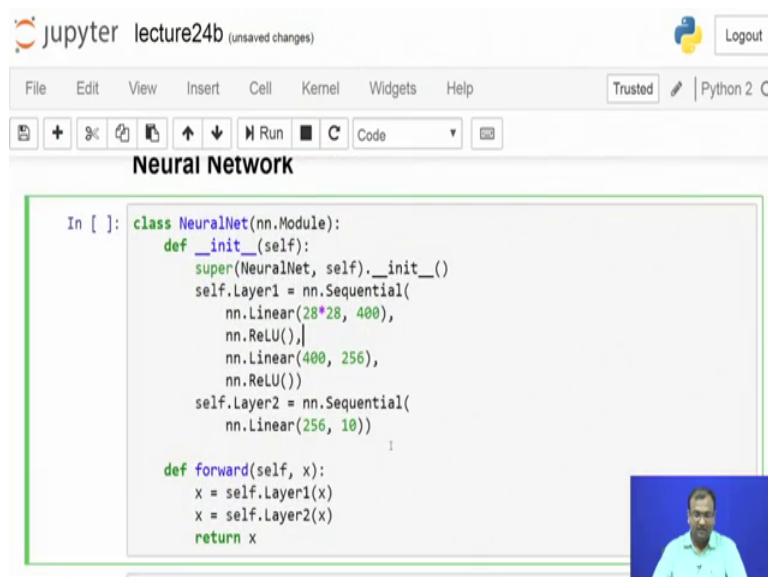
trainset = torchvision.datasets.MNIST(root='./MNIST', train=True,
                                     download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=BatchSize,
                                          shuffle=True, num_workers=4) # Creating

testset = torchvision.datasets.MNIST(root='./MNIST', train=False,
                                     download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=BatchSize,
                                          shuffle=False, num_workers=4) # Creating

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

Now, once the data loader is done, I check out whether my GPU is available and voila.

(Refer Slide Time: 11:42)

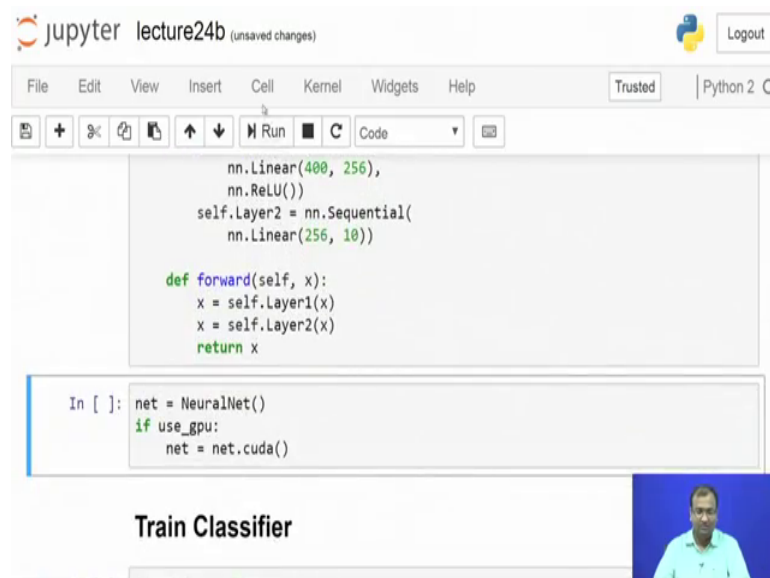


```
lecture24b (unsaved changes) Python 2.0
File Edit View Insert Cell Kernel Widgets Help Trusted Python 2.0
Run Code
Neural Network
In [ ]: class NeuralNet(nn.Module):
def __init__(self):
    super(NeuralNet, self).__init__()
    self.Layer1 = nn.Sequential(
        nn.Linear(28*28, 400),
        nn.ReLU(),
        nn.Linear(400, 256),
        nn.ReLU())
    self.Layer2 = nn.Sequential(
        nn.Linear(256, 10))

def forward(self, x):
    x = self.Layer1(x)
    x = self.Layer2(x)
    return x
```

Once that is there, I have my neural network which is synthesized and that it is the same neural network. So, I am not technically changing out anything over there on the network.

(Refer Slide Time: 11:51)



The screenshot shows a Jupyter Notebook window titled "lecture24b (unsaved changes)". 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 in the notebook is as follows:

```
nn.Linear(400, 256),
nn.ReLU())
self.Layer2 = nn.Sequential(
    nn.Linear(256, 10))

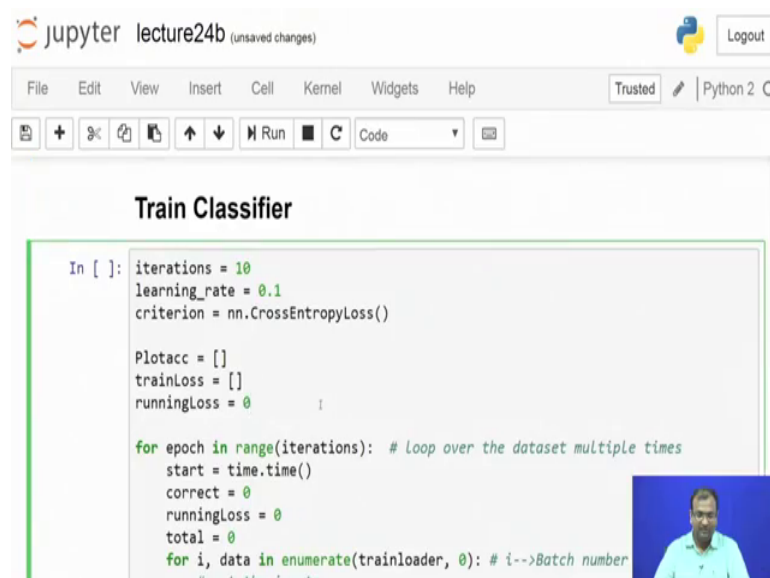
def forward(self, x):
    x = self.Layer1(x)
    x = self.Layer2(x)
    return x

In [ ]: net = NeuralNet()
if use_gpu:
    net = net.cuda()
```

Below the code, the text "Train Classifier" is visible, and a small video inset shows a person speaking.

And, that does not need any further definition. So, check out I get my new electric converted on to cuda and then I try to start my classifier.

(Refer Slide Time: 11:57)



The screenshot shows the same Jupyter Notebook window. The code in the notebook is as follows:

```
iterations = 10
learning_rate = 0.1
criterion = nn.CrossEntropyLoss()

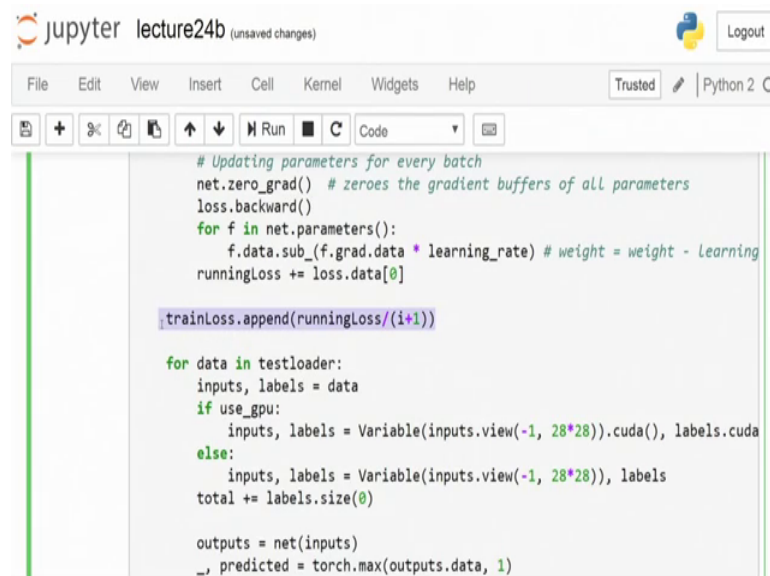
Plotacc = []
trainLoss = []
runningLoss = 0

for epoch in range(iterations): # Loop over the dataset multiple times
    start = time.time()
    correct = 0
    runningLoss = 0
    total = 0
    for i, data in enumerate(trainloader, 0): # i-->Batch number
        # get the inputs
```

Below the code, the text "Train Classifier" is visible, and a small video inset shows a person speaking.

Now, here also I am going to train it over 10 epochs.

(Refer Slide Time: 12:08)



```
jupyter lecture24b (unsaved changes) Python 2 O
File Edit View Insert Cell Kernel Widgets Help Trusted Python 2 O
# Updating parameters for every batch
net.zero_grad() # zeroes the gradient buffers of all parameters
loss.backward()
for f in net.parameters():
    f.data.sub_(f.grad.data * learning_rate) # weight = weight - Learning
runningLoss += loss.data[0]

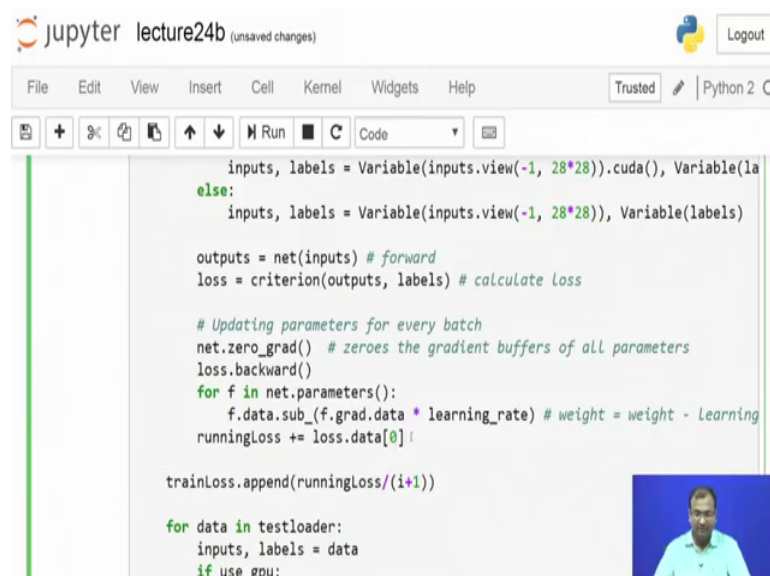
trainLoss.append(runningLoss/(i+1))

for data in testloader:
    inputs, labels = data
    if use_gpu:
        inputs, labels = Variable(inputs.view(-1, 28*28)).cuda(), labels.cuda()
    else:
        inputs, labels = Variable(inputs.view(-1, 28*28)), labels
    total += labels.size(0)

    outputs = net(inputs)
    _, predicted = torch.max(outputs.data, 1)
```

However, the difference which comes down is in terms of my loss over here, because this loss over here is just what is computed poor, sorry. So, you have your loss which is computed over here which is for every epoch. Now, if you remember the earlier one in the earlier case I was basically calculating out these losses and then cumulating out the loss over here somewhere which is outside the inner loop which runs over the total number of batches which is loading.

(Refer Slide Time: 12:33)



```
jupyter lecture24b (unsaved changes) Python 2 O
File Edit View Insert Cell Kernel Widgets Help Trusted Python 2 O
inputs, labels = Variable(inputs.view(-1, 28*28)).cuda(), Variable(labels)
else:
    inputs, labels = Variable(inputs.view(-1, 28*28)), Variable(labels)

outputs = net(inputs) # forward
loss = criterion(outputs, labels) # calculate loss

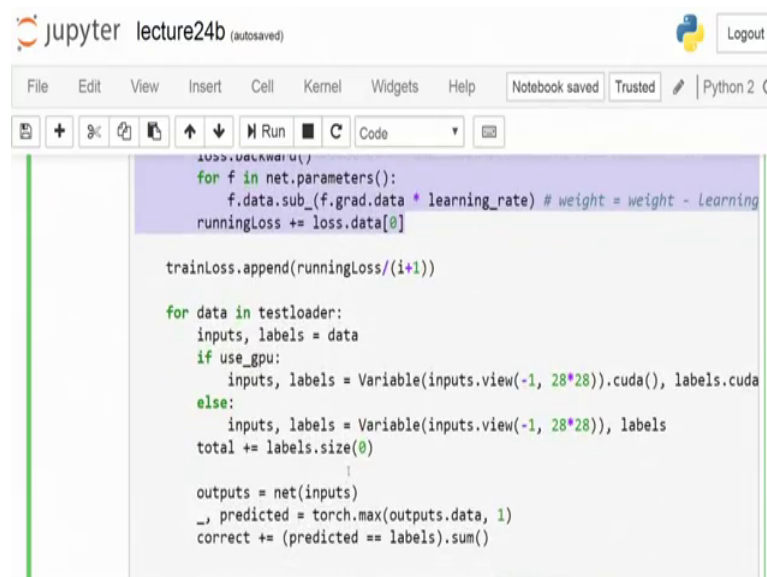
# Updating parameters for every batch
net.zero_grad() # zeroes the gradient buffers of all parameters
loss.backward()
for f in net.parameters():
    f.data.sub_(f.grad.data * learning_rate) # weight = weight - Learning
runningLoss += loss.data[0]

trainLoss.append(runningLoss/(i+1))

for data in testloader:
    inputs, labels = data
    if use_gpu:
```

So, this update over here was happening once every epoch itself whereas, over here you would see this update is happening within every batch itself. So, this is present within the inner loop not within the outer loop and that is the difference which comes down with a batch loader.

(Refer Slide Time: 12:48)



```
loss.backward()
for f in net.parameters():
    f.data.sub_(f.grad.data * learning_rate) # weight = weight - learning_rate * grad
runningLoss += loss.data[0]

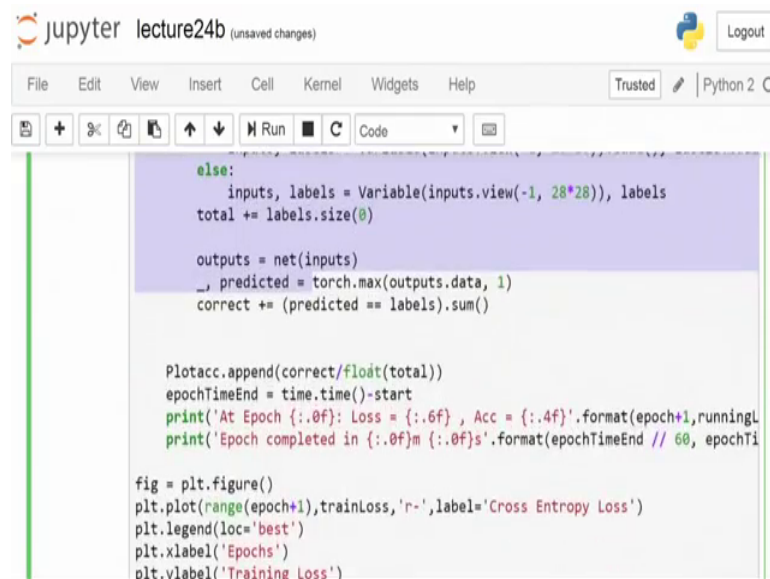
trainLoss.append(runningLoss/(i+1))

for data in testloader:
    inputs, labels = data
    if use_gpu:
        inputs, labels = Variable(inputs.view(-1, 28*28)).cuda(), labels.cuda()
    else:
        inputs, labels = Variable(inputs.view(-1, 28*28)), labels
    total += labels.size(0)

    outputs = net(inputs)
    _, predicted = torch.max(outputs.data, 1)
    correct += (predicted == labels).sum()
```

Obviously, it is supposed to be a bit more computationally expensive. So, let me just run it down because. So, we can just utilize the rest of the time to discuss around it. Now, once you have this updated down, so, there will be multiple number of updates which will happen down in every epoch and then you find out what is the total loss at the end of that epoch and that is what you are storing now.

(Refer Slide Time: 13:20)



```
else:
    inputs, labels = Variable(inputs.view(-1, 28*28)), labels
    total += labels.size(0)

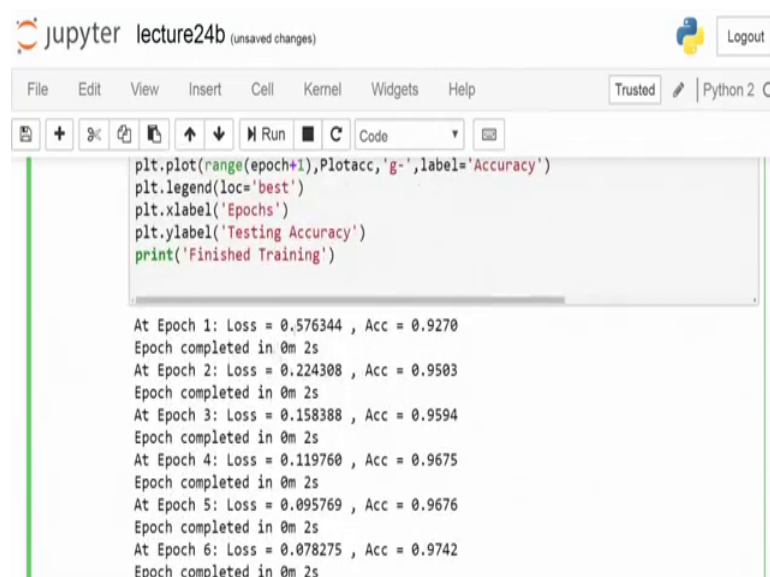
    outputs = net(inputs)
    _, predicted = torch.max(outputs.data, 1)
    correct += (predicted == labels).sum()

    Plotacc.append(correct/float(total))
    epochTimeEnd = time.time()-start
    print('At Epoch {:.0f}: Loss = {:.6f} , Acc = {:.4f}'.format(epoch+1,runningL
    print('Epoch completed in {:.0f}m {:.0f}s'.format(epochTimeEnd // 60, epochTi

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

On the other side, within every epoch I am also trying to do a validation with the test loader coming down over there and then we will have our timing and all of the other stuff predicted out over there.

(Refer Slide Time: 13:24)

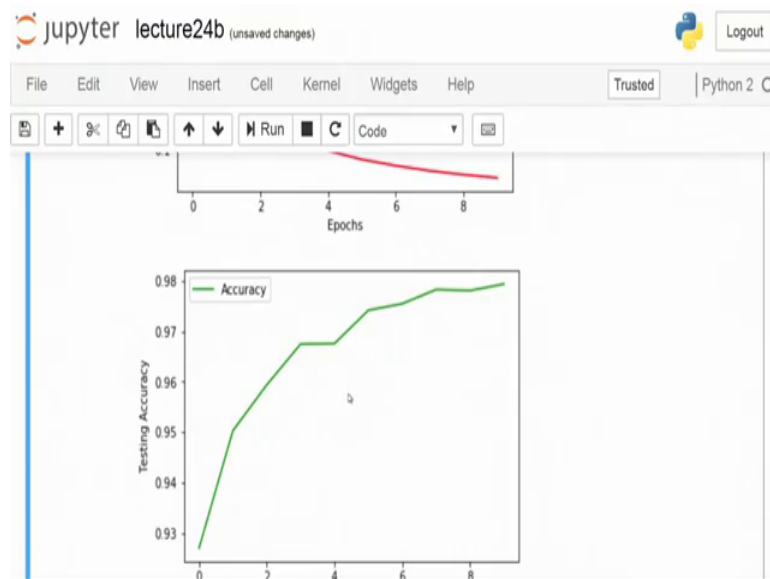


```
plt.plot(range(epoch+1),Plotacc,'g-',label='Accuracy')
plt.legend(loc='best')
plt.xlabel('Epochs')
plt.ylabel('Testing Accuracy')
print('Finished Training')
```

```
At Epoch 1: Loss = 0.576344 , Acc = 0.9270
Epoch completed in 0m 2s
At Epoch 2: Loss = 0.224308 , Acc = 0.9503
Epoch completed in 0m 2s
At Epoch 3: Loss = 0.158388 , Acc = 0.9594
Epoch completed in 0m 2s
At Epoch 4: Loss = 0.119760 , Acc = 0.9675
Epoch completed in 0m 2s
At Epoch 5: Loss = 0.095769 , Acc = 0.9676
Epoch completed in 0m 2s
At Epoch 6: Loss = 0.078275 , Acc = 0.9742
Epoch completed in 0m 2s
```

Now, typically you see that here also it takes almost the same time, about 2 seconds to do it and it starts with a decently better accuracy than you had in the earlier case. Now, this should be coming down as a question to you, as to why did it actually start with a higher accuracy as compared to what it was in the earlier case.

(Refer Slide Time: 13:38)



Because, in the earlier case you did see that barely it was able to get down to about 45 percent of our accuracy, whereas here, it just immediately started with having 92.7 percent of accuracy.

(Refer Slide Time: 13:43)

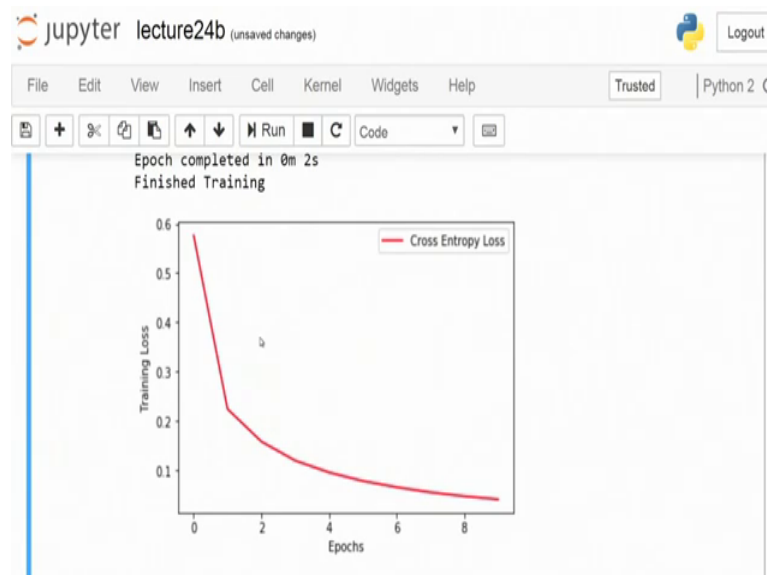
```
At Epoch 1: Loss = 0.576344 , Acc = 0.9270
Epoch completed in 0m 2s
At Epoch 2: Loss = 0.224308 , Acc = 0.9503
Epoch completed in 0m 2s
At Epoch 3: Loss = 0.158388 , Acc = 0.9594
Epoch completed in 0m 2s
At Epoch 4: Loss = 0.119760 , Acc = 0.9675
Epoch completed in 0m 2s
At Epoch 5: Loss = 0.095769 , Acc = 0.9676
Epoch completed in 0m 2s
At Epoch 6: Loss = 0.078275 , Acc = 0.9742
Epoch completed in 0m 2s
At Epoch 7: Loss = 0.065687 , Acc = 0.9755
Epoch completed in 0m 2s
At Epoch 8: Loss = 0.055337 , Acc = 0.9783
Epoch completed in 0m 2s
At Epoch 9: Loss = 0.047407 , Acc = 0.9781
```

Now, remember this one thing in the earlier case, the whole network was actually updated at most 10 times that is a total number of times though everything was getting updated. Here, within every epoch it actually gets updated 600 times. So, your weights

have changed down 600 times within every single iteration what it has done and now by the end of tenth epoch there have been 6000 such updates which have happened.

Now, if you go back to the earlier case and actually run it over an iteration of 6000 epochs you would actually tend to come down to the same accuracy level and the same kind of a curve as you are getting down over there. So, in the earlier case, if you were running it down initially for 600 epochs you would be getting down this accuracy. If you have run it down for say 602, which is 1200 epochs you would be somewhere around this accuracy point as compared to what you are having over here.

(Refer Slide Time: 14:47)

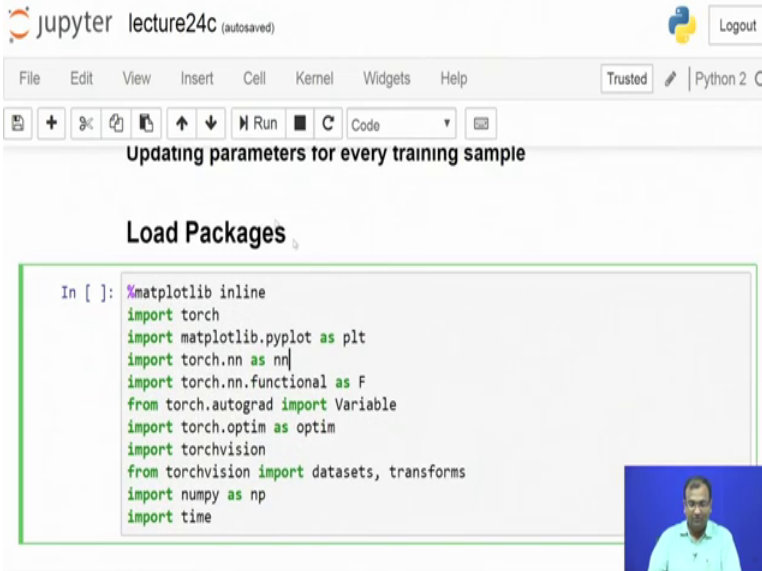


And, that is a distinct advantage we are trying to do a batch update you are taking almost the same amount of time. But, you have a much faster way of coming down to a convergence, because the number of times it is actually updating itself and correcting the mistakes which it was making is much large, though it is not updating itself on the same data. It is getting a newer instance of the data and it is looking down.

So, it is the same thing as say if you are trying to learn up a subject. So, the more the number of books across which you study and you give an exam in between or get yourself assessed the better you are in learning a subject's task rather than taking a bigger book, a fatter book and trying to mug it up across the whole semester. So, that does not technically always find out a way whereas, if you have this smaller handout books or notes taken down from multiple people and you try to get into a subject because you are

getting exposed to more and more different variants of how you learned on a particular topic. So, it is the same thing which comes down as a batch update learning rule for a neural network as well.

(Refer Slide Time: 15:58)



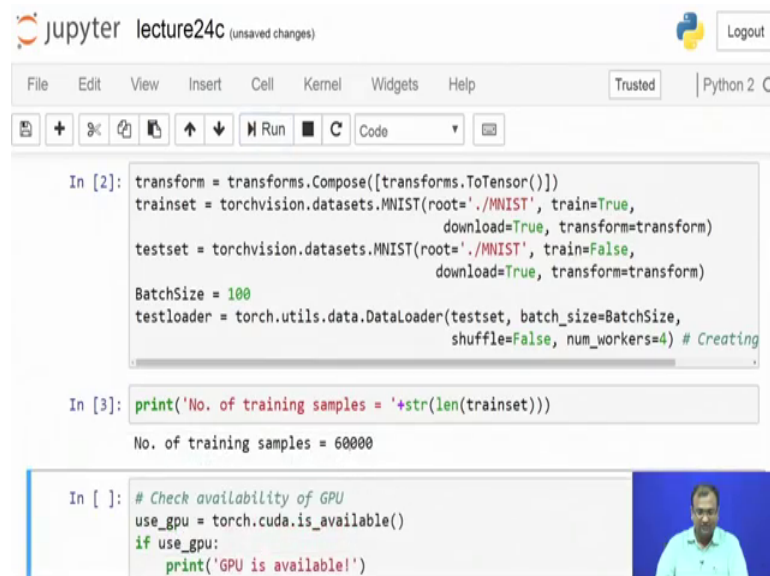
The screenshot shows a Jupyter Notebook window titled "lecture24c (autosaved)". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for file operations and execution. The notebook content is titled "Updating parameters for every training sample" and "Load Packages". A code cell contains the following Python code:

```
In [ ]: %matplotlib inline
import torch
import matplotlib.pyplot as plt
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
import torch.optim as optim
import torchvision
from torchvision import datasets, transforms
import numpy as np
import time
```

A small video inset in the bottom right corner shows a person speaking.

Now, that would bring me to the next one which is a instance or every sample update and this is let us say the same as you basically, read one section of a topic in a book and try to correct yourself for anything else asked on that particular topic and let us see what comes down. This is quite interesting because here what would happen. So, let us just do a walk through. The walk through is pretty simple you start with just loading down your initial forms over there then you do a data loader.

(Refer Slide Time: 16:22)



```

jupyter lecture24c (unsaved changes)
File Edit View Insert Cell Kernel Widgets Help Trusted Python 2 O
In [2]: transform = transforms.Compose([transforms.ToTensor()])
trainset = torchvision.datasets.MNIST(root='./MNIST', train=True,
                                     download=True, transform=transform)
testset = torchvision.datasets.MNIST(root='./MNIST', train=False,
                                     download=True, transform=transform)
BatchSize = 100
testloader = torch.utils.data.DataLoader(testset, batch_size=BatchSize,
                                         shuffle=False, num_workers=4) # Creating

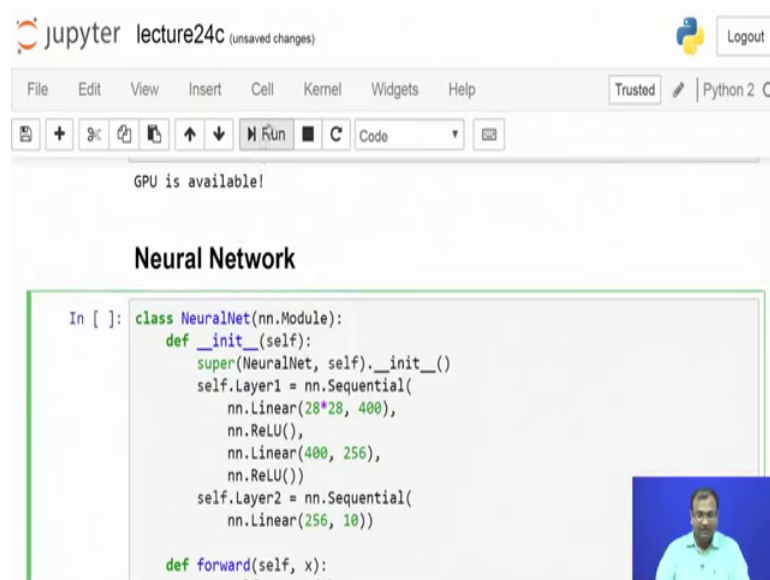
In [3]: print('No. of training samples = '+str(len(trainset)))

No. of training samples = 60000

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

And, then over here it is just to print down your number of samples and there are 60000 samples. So, technically, that means, that within every epoch if I am going to do an update per sample, so, I am going to do that update 60000 times over there.

(Refer Slide Time: 16:31)



```

jupyter lecture24c (unsaved changes)
File Edit View Insert Cell Kernel Widgets Help Trusted Python 2 O
GPU is available!

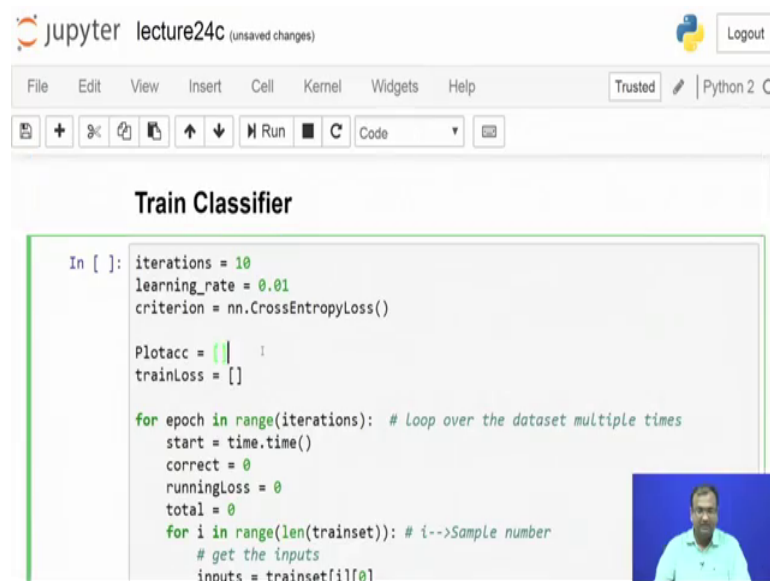
Neural Network

In [ ]: class NeuralNet(nn.Module):
def __init__(self):
    super(NeuralNet, self).__init__()
    self.Layer1 = nn.Sequential(
        nn.Linear(28*28, 400),
        nn.ReLU(),
        nn.Linear(400, 256),
        nn.ReLU())
    self.Layer2 = nn.Sequential(
        nn.Linear(256, 10))

def forward(self, x):
```

Now, I define my network which is the same network. I get it on my GPU put it down because I have a GPU available and then I start my training instance over here.

(Refer Slide Time: 16:40)



```
In [ ]: iterations = 10
learning_rate = 0.01
criterion = nn.CrossEntropyLoss()

Plotacc = []
trainLoss = []

for epoch in range(iterations): # Loop over the dataset multiple times
    start = time.time()
    correct = 0
    runningLoss = 0
    total = 0
    for i in range(len(trainset)): # i-->Sample number
        # get the inputs
        inputs = trainset[i][0]
```

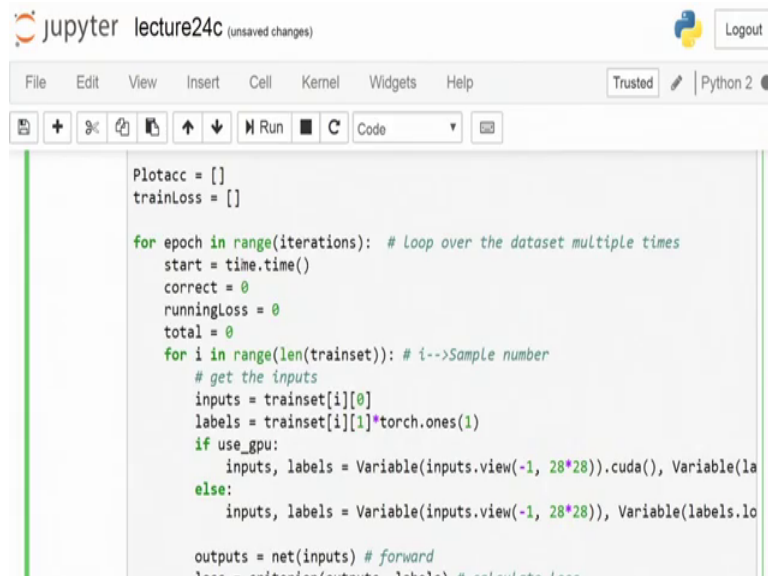
I will really set this one running; it is going to take a bit amount of time. In fact, it is a significant amount of time as you would see down over there. So, this is going to run down over 10 iterations you see a distinct difference in the learning rate which I have taken down over there because this is significantly lower than the learning it in the earlier cases now. The reason why this is so is, you are going to update down at every single sample as it keeps on coming down. Now, your total gradient which comes down is dependent very much on what sample was over there. So, every sample is not going to have the same range of error.

In the earlier cases because you were taking on an average over all the samples over there, average of the error, so, it was getting down to some sort of average tendency a lower bound over there, a much more consistent bound between multiple batches which come down in an epoch. Whereas, over here for every single sample it is going to be either it can be 0 or it can be very large. So, if there is a significant deviation over there would be a very large error which comes down and that is going to impact actually the total weight update which you are getting down and in your whole thing.

Now, in order to keep it on a safer side we would like to put down a learning rate which is lower so that at any point of time the dynamic range of the weights do not just get overwhelmed by an unguided down by the dynamic range in which my updates are

changing. So, that is the only reason for keeping this learning rate at a lower point of time.

(Refer Slide Time: 18:10)



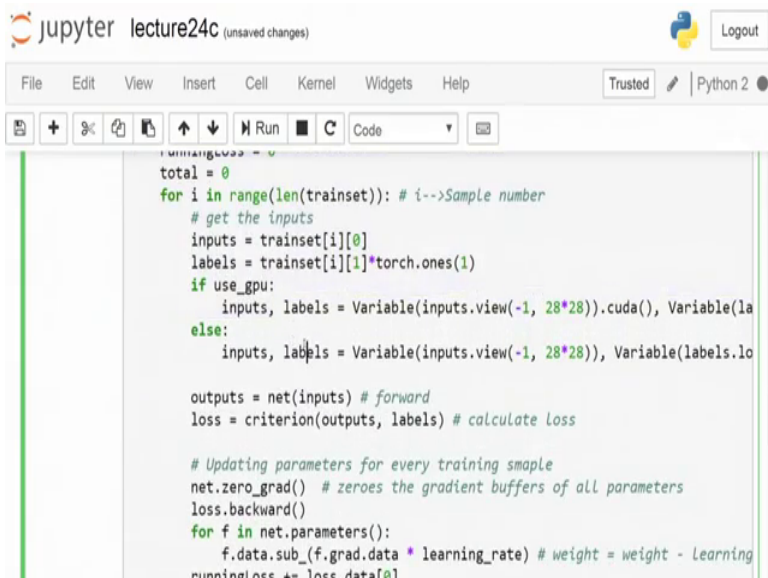
```
Plotacc = []
trainLoss = []

for epoch in range(iterations): # Loop over the dataset multiple times
    start = time.time()
    correct = 0
    runningLoss = 0
    total = 0
    for i in range(len(trainset)): # i-->Sample number
        # get the inputs
        inputs = trainset[i][0]
        labels = trainset[i][1]*torch.ones(1)
        if use_gpu:
            inputs, labels = Variable(inputs.view(-1, 28*28)).cuda(), Variable(labels)
        else:
            inputs, labels = Variable(inputs.view(-1, 28*28)), Variable(labels)

        outputs = net(inputs) # forward
        loss = criterion(outputs, labels) # calculate loss
```

And, then, I basically start my whole iterator over here.

(Refer Slide Time: 18:14)



```
total = 0
for i in range(len(trainset)): # i-->Sample number
    # get the inputs
    inputs = trainset[i][0]
    labels = trainset[i][1]*torch.ones(1)
    if use_gpu:
        inputs, labels = Variable(inputs.view(-1, 28*28)).cuda(), Variable(labels)
    else:
        inputs, labels = Variable(inputs.view(-1, 28*28)), Variable(labels)

    outputs = net(inputs) # forward
    loss = criterion(outputs, labels) # calculate loss

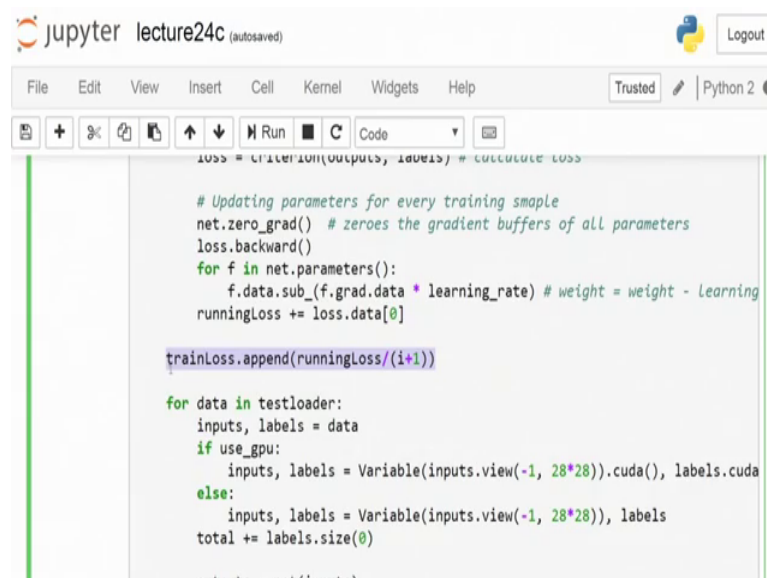
    # Updating parameters for every training sample
    net.zero_grad() # zeroes the gradient buffers of all parameters
    loss.backward()
    for f in net.parameters():
        f.data.sub_(f.grad.data * learning_rate) # weight = weight - Learning
    runningLoss += loss.data[0]
```

Now, in the inner range what I do is, as in the earlier case I was actually doing a match loader. So, I was able to load down 100 images per batch, but here I am just going to

load down 1 image at a point of time or this is equal to setting down a match size of 1, you can say.

Now, for each image I am going to find out my output whatever it is then get down the loss which is my out of my criterion. Now, once the loss is calculated down we do a zeroing of all the gradients and then do a back propagation over find out a gradient of the loss itself on nabla of the cost function and then write an update rule over here and here what I am doing is basically, I sum up the average loss taken down over each of them.

(Refer Slide Time: 18:57)



```
loss = criterion(outputs, labels) # calculate loss

# Updating parameters for every training sample
net.zero_grad() # zeroes the gradient buffers of all parameters
loss.backward()
for f in net.parameters():
    f.data.sub_(f.grad.data * learning_rate) # weight = weight - Learning
    runningLoss += loss.data[0]

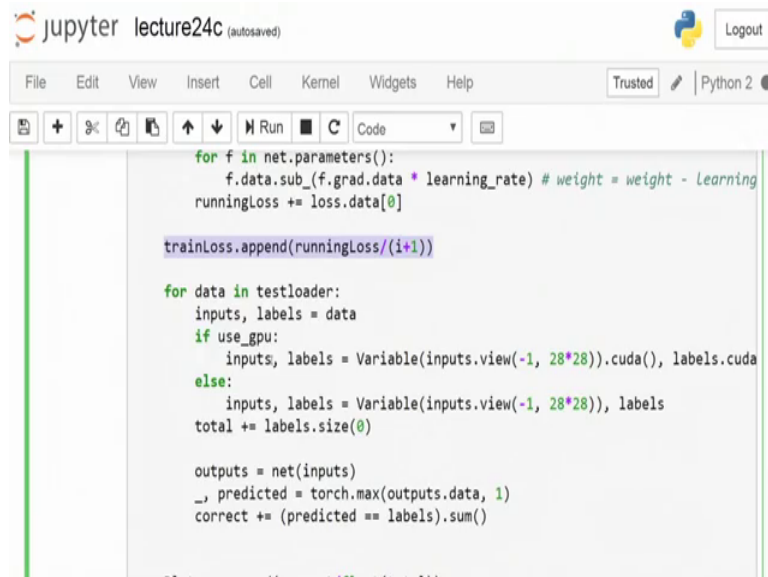
trainLoss.append(runningLoss/(i+1))

for data in testloader:
    inputs, labels = data
    if use_gpu:
        inputs, labels = Variable(inputs.view(-1, 28*28)).cuda(), labels.cuda()
    else:
        inputs, labels = Variable(inputs.view(-1, 28*28)), labels
    total += labels.size(0)

    outputs = net(inputs)
```

So, technically this means that, since I have 60000 examples present down within one single training set in 1 epoch. So, there will be 60000 times there would be updates happening over there. Now, pitch this against the earlier case. In the earlier case you had 600 times of an update per epoch, here you are able to have 60000 times of an update per epoch. The downside do is that every time it is with one single sample. So, you are not tending to be on the average case of the errors which is coming down over there. As well as you are going to solve much more number of update equations.

(Refer Slide Time: 19:34)



```
for f in net.parameters():
    f.data.sub_(f.grad.data * learning_rate) # weight = weight - Learning
    runningLoss += loss.data[0]

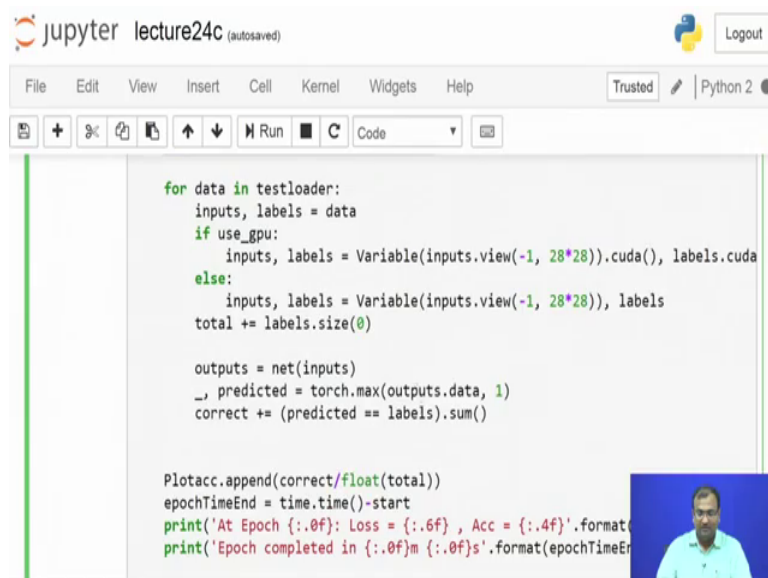
trainLoss.append(runningLoss/(i+1))

for data in testloader:
    inputs, labels = data
    if use_gpu:
        inputs, labels = Variable(inputs.view(-1, 28*28)).cuda(), labels.cuda
    else:
        inputs, labels = Variable(inputs.view(-1, 28*28)), labels
    total += labels.size(0)

    outputs = net(inputs)
    _, predicted = torch.max(outputs.data, 1)
    correct += (predicted == labels).sum()
```

So, 6000 versus 60000 that is 100 times more is the amount of compute which will happen downpour epoch over here and that is how it is getting solved.

(Refer Slide Time: 19:42)



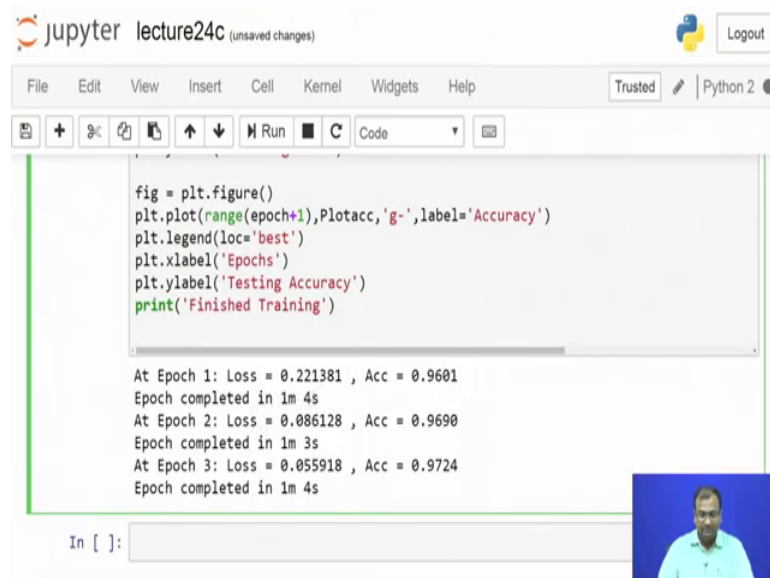
```
for data in testloader:
    inputs, labels = data
    if use_gpu:
        inputs, labels = Variable(inputs.view(-1, 28*28)).cuda(), labels.cuda
    else:
        inputs, labels = Variable(inputs.view(-1, 28*28)), labels
    total += labels.size(0)

    outputs = net(inputs)
    _, predicted = torch.max(outputs.data, 1)
    correct += (predicted == labels).sum()

Plotacc.append(correct/float(total))
epochTimeEnd = time.time()-start
print('At Epoch {:.0f}: Loss = {:.6f} , Acc = {:.4f}'.format(epochTimeEnd, trainLoss[-1], Plotacc[-1]))
print('Epoch completed in {:.0f}m {:.0f}s'.format(epochTimeEnd, epochTimeEnd))
```

So, the rest of it is pretty simple that within each epoch once the training per sample based is over and then we write down the test data loader which loads up the test example and then with within every epoch, I basically try to look into it.

(Refer Slide Time: 19:55)



```
fig = plt.figure()
plt.plot(range(epoch+1),Plotacc,'g-',label='Accuracy')
plt.legend(loc='best')
plt.xlabel('Epochs')
plt.ylabel('Testing Accuracy')
print('Finished Training')
```

```
At Epoch 1: Loss = 0.221381 , Acc = 0.9601
Epoch completed in 1m 4s
At Epoch 2: Loss = 0.086128 , Acc = 0.9690
Epoch completed in 1m 3s
At Epoch 3: Loss = 0.055918 , Acc = 0.9724
Epoch completed in 1m 4s
```

In []:

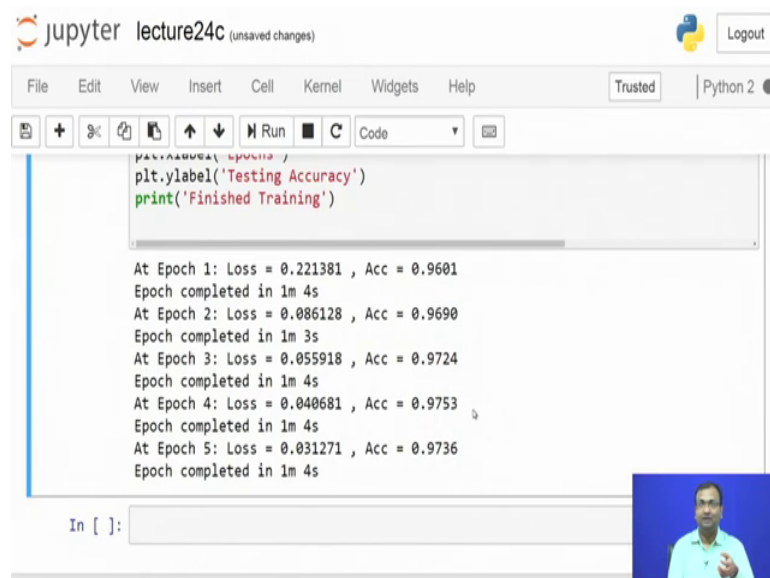
So, yeah this actually consumes a significant amount of time. You can see pretty much it consumes of almost over 1 minute 4 second which is close to a more than a minute over there. However, the starting accuracy is really large; you just start down with an accuracy of 96 percent. Now, that is great right, because I just solved down all the examples and that my 0 at epoch whatever it was it is 96 percent; keep one thing in mind, in the first example where I was doing an update over the whole epoch. So, that is where I was randomly guessing before I made any updates over here. Here, at the end of my first epoch it is already has been updated by 60000 times, at the end of my second epochs it is basically 1, 20,000 times that the whole thing is has been updated.

At the end of third epoch it is 1, 80,000 times that it is getting updated and that means, that the number of times it is seeing and it is trying to update itself is much larger. So, if you take the example from the first case where we had just trained it over 10 epochs and there were updates just for 10 times over there, you train it down for 60000 you would inherently get down an accuracy which matches down about 96 percent, quite easily with without much of it.

So, this is a place where you will have to actually now start to think of syncing and tuning around to play with it as to how they mix and match and work. On one side while just trying to do one update per epoch is not a great idea because you will have to run more number of epochs, the other one is trying to do a batch update which takes almost

the same time as doing one update per epoch whereas, in the in the batch update case you actually reach down a much higher accuracy much faster. The other one which is a instance based object or per sample, this is where you would possibly be reaching down the highest accuracy at the shortest possible epoch number. The time consumed per epoch is going to be significantly large and that is really problematic.

(Refer Slide Time: 22:01)



```
At Epoch 1: Loss = 0.221381 , Acc = 0.9601
Epoch completed in 1m 4s
At Epoch 2: Loss = 0.086128 , Acc = 0.9690
Epoch completed in 1m 3s
At Epoch 3: Loss = 0.055918 , Acc = 0.9724
Epoch completed in 1m 4s
At Epoch 4: Loss = 0.040681 , Acc = 0.9753
Epoch completed in 1m 4s
At Epoch 5: Loss = 0.031271 , Acc = 0.9736
Epoch completed in 1m 4s
```

If you can think of sparing that amount of time it is well and good, but for most of us that is that is not a practical feasible solution and for that perspective we do understand that actually trying to have a batch update rule is a much practical approach to do it and which is what is followed on. In fact, the way that data loader was written the reason why we were loading out in batches and trying to do, is everything is guided because this learning rule itself is much more better and comprehensive. So, it is a good trade off between trying to reach down your saturation accuracy to the amount of time you take to reach down that accuracy per epoch. So, if you look into it that what is the total number of seconds I have spent in order to come down to say 96 percent of accuracy.

So, over here this is going to be somewhat like 1 minute 4 second is when I come down to a 96 percent of an accuracy. As compared to this batch update where I was able to come down to 96.7 percent of accuracy within just the fourth epoch and that is like 2 second per epoch. So, it is basically 8 seconds is my total time taken down in order to come down to the same accuracy as versus 1 minute of time to come down to an

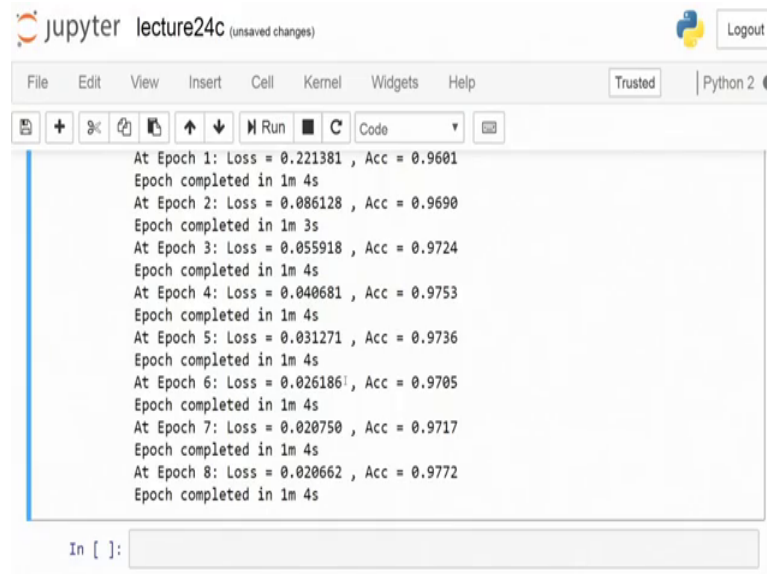
accuracy which is closer to that in this first case and that is a significant advantage of trying to use a batch update and why we would be sticking down to that. Since, I have kept it running for 10 epochs you just have to wait over it whereas, if you look into this accuracy facts over there you see that after some point of time it just does not increase to that much of a level.

So, the amount of time I am spending down in going down from 96 percent of an accuracy to say 97 percent of an accuracy is close to about more than is about 7 minutes let us say 1 minute 4 second into 6 is close to over 7 minutes whereas, it does not actually pay off. In 7 minutes you are being able to just gain one percent of accuracy as compared to the earlier case in a batch update where like just within 2 seconds you have been able to see a change in accuracy of 3 percent. Now, that is a significant change which comes down and this initial curve, that in the first 2 seconds itself you reached out about 92 percent and that compared to here where it took you almost a minute to each up to 96 percent.

So, these are simple tricks around the point of why we would be using one and not the other way of doing it down and what can be a better way of doing and the other point is that within a batch update you definitely save out on a lot of your RAM space. You might not have enough of space to load the whole dataset, but again just loading one single sample at a time is also not a feasible. You are going to spend a lot of time just doing a hardware IO operation, you are going to fetch down something from your hard drive and then copy it down.

So, there is a significant amount of DMA transfer operations which is involved over there and since typically your DMA buses. So, from your knowledge of computer organizations and operating systems you do understand that your DMA bus is almost like 10 times or even 20 times slower than your CPU block and the bus between your CPU to your RAM and that is going to guide it down. Whereas, you can do a batch update or say a significant number of throughput updates over there and with a batch loader this is where you basically come down to a advantageous position as versus doing it with just one single sample at a point of time.

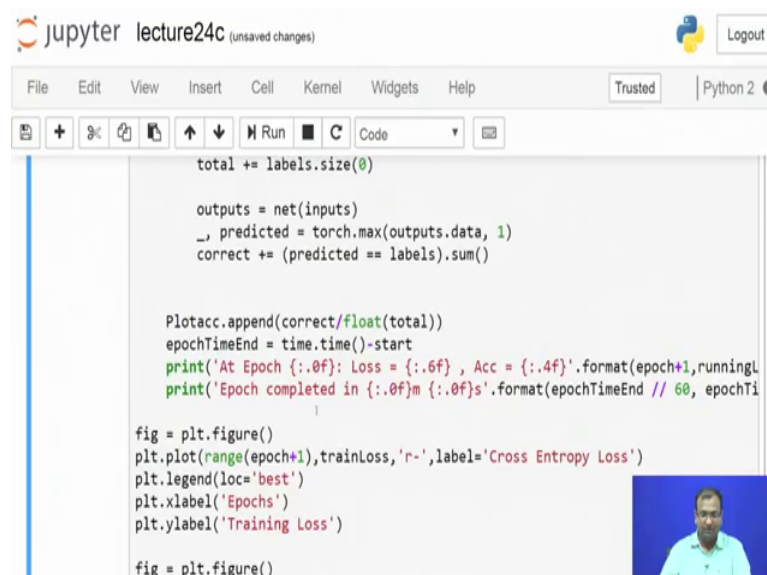
(Refer Slide Time: 25:23)



```
jupyter lecture24c (unsaved changes) Python 2
File Edit View Insert Cell Kernel Widgets Help Trusted
+ %< > Run C Code
At Epoch 1: Loss = 0.221381 , Acc = 0.9601
Epoch completed in 1m 4s
At Epoch 2: Loss = 0.086128 , Acc = 0.9690
Epoch completed in 1m 3s
At Epoch 3: Loss = 0.055918 , Acc = 0.9724
Epoch completed in 1m 4s
At Epoch 4: Loss = 0.040681 , Acc = 0.9753
Epoch completed in 1m 4s
At Epoch 5: Loss = 0.031271 , Acc = 0.9736
Epoch completed in 1m 4s
At Epoch 6: Loss = 0.026186 , Acc = 0.9705
Epoch completed in 1m 4s
At Epoch 7: Loss = 0.020750 , Acc = 0.9717
Epoch completed in 1m 4s
At Epoch 8: Loss = 0.020662 , Acc = 0.9772
Epoch completed in 1m 4s
In [ ]:
```

So, if you look down, somewhere around 8 epoch which is almost close to 9 minutes that we have been speaking around, it is somewhere at 1.7 percent of a change which is there and then that is not something which is appreciated in any way for your own practical purposes. So, let us just wait for some more time and we can actually see the same kind of a curve coming.

(Refer Slide Time: 25:54)



```
jupyter lecture24c (unsaved changes) Python 2
File Edit View Insert Cell Kernel Widgets Help Trusted
+ %< > Run C Code
total += labels.size(0)

outputs = net(inputs)
_, predicted = torch.max(outputs.data, 1)
correct += (predicted == labels).sum()

Plotacc.append(correct/float(total))
epochTimeEnd = time.time()-start
print('At Epoch {:.0f}: Loss = {:.6f} , Acc = {:.4f}'.format(epoch+1,runningL
print('Epoch completed in {:.0f}m {:.0f}s'.format(epochTimeEnd // 60, epochTi

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

fig = plt.figure()
```

However, if there is one crazy thing over here instead of trying to plot down what we plot over here is basically the loss at every epoch. Now, the whole point over here is that

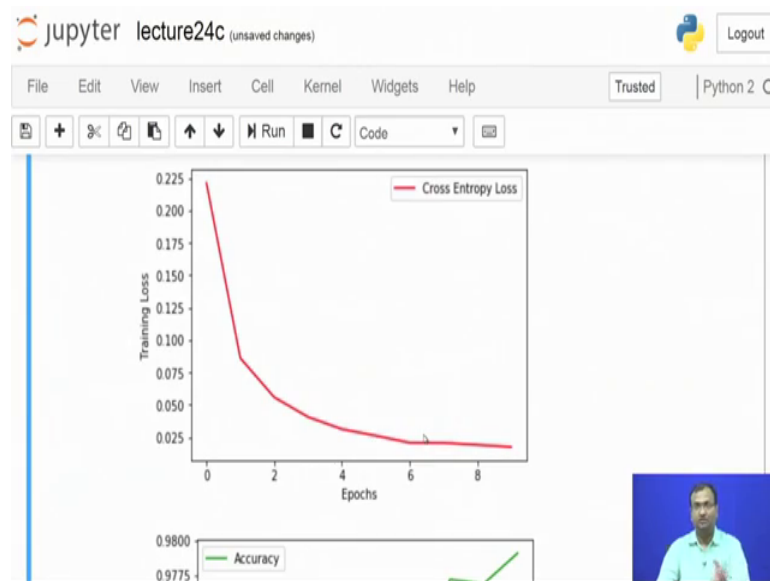
instead of trying to plot the loss at every epoch, now let us try to plot the loss after every single sample as it keeps on going. You would see a very crazy curve like it will come down as a jittery curve and in certain times you would see that it rises up then comes down then rises up and comes down. But, however, the average trend over that particular line is something which will follow down. Now, compare that with learning curve for every batch itself you would see that that every batch one has a much smoother decrease as it goes down. So, I am almost done with my ninth epoch and if you wait for some more time you can actually see the tenth epoch going out.

Now, look into one point; this is what we call as a point of inflection. This is where you see that the test accuracy over there has actually started decreasing. Now, this is a point in classical learning theory is what we call that the network has started to memorize whatever has been given down to it is training example.

So, it is getting into something which is called as an over fitting problem. It is very good, it is reducing down the error if you look into these losses. The loss has decreased because this loss was computed on the training dataset itself; whereas, this accuracy is decreasing instead of increasing. If your loss was decreasing then your accuracy should be increasing whereas, here do you see that your accuracy had decreased over here as compared to this one. It suddenly again increases over, here it does increase, but this is a point of inflection which is now, starting to happen over here and that is something called as an over fitting problem.

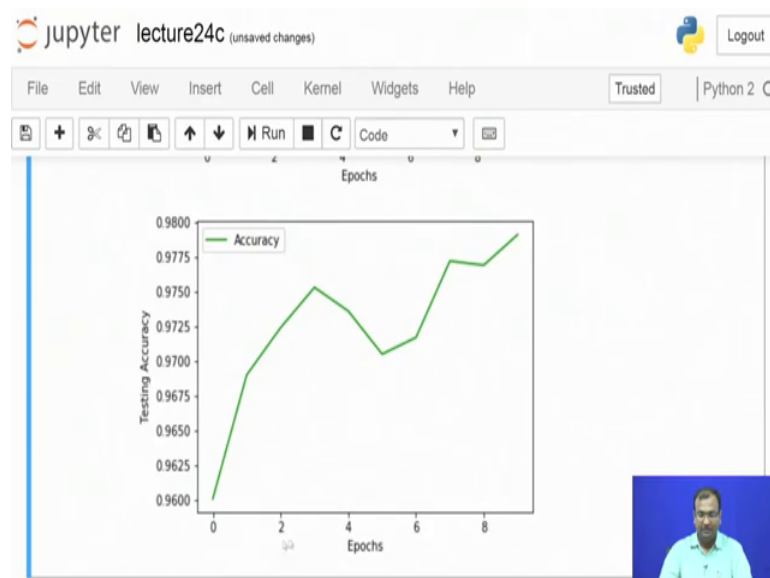
So, later on, in the other lectures we will be going down to how to identify these points of inflection.

(Refer Slide Time: 27:46)



And, when the over fitting is happening and what you can do to actually cut down these issues of any over fitting happening.

(Refer Slide Time: 27:49)



You can typically see like around these epochs is when it started these inflection. And, from our experiences we know that once it is at this point and beyond everything it is actually trying to memorize the network in a big way. It is just trying to pull itself out, though of that practice, but this is something which happens if you are training it over a longer period of epochs.

So, with that we have done some basic understandings of the different kind of learning rules from your global updates per epoch to batch updates to single sample based updates and the pros and cons around with that. So, with then, stay tuned until the next lecture comes up.

Thanks.