

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

Lecture - 60
Activity Recognition using CNN-LSTM

Welcome to the last lecture of this series. And this is where the sweet part and so over that. Now nonetheless so, we are working on this week on video analysis and video activity recognition classification and these kind of problems. So, while on one part you have learned about the theory of how to treat down these videos and as a tensor representation. And what is the implication of having these tensors in a newer format of representation.

So, well down where what dimensions are going down in which particular level over there on your tensor. And that is the last lecture we have also done using a 3D CNN, which was you try to do a spatiotemporal convolution on top of videos, and then come up with a classification system for your videos as well.

Now, today we are going to do come down to that other version of it where you can use some sort of recurrent neural network, or which is a time scale modelling neural network over there. And the particular one which we had studied in the lectures were called as a LSTM or long short-term memory.

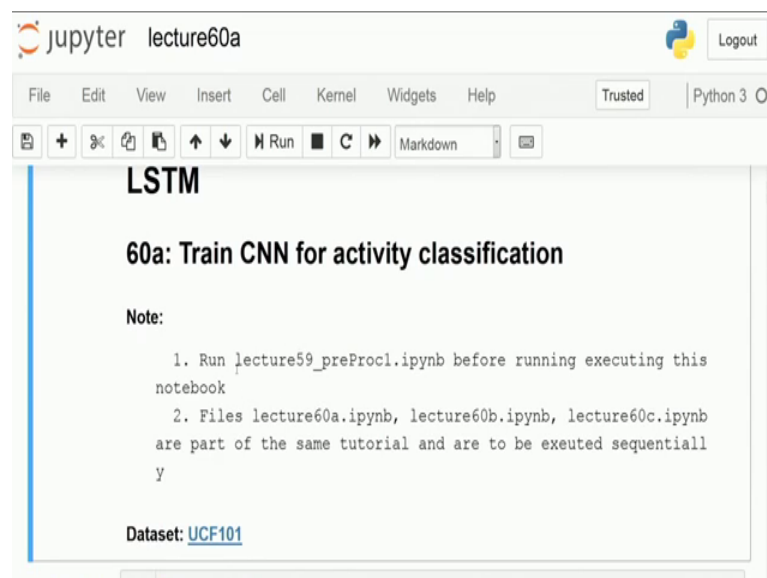
So, today's particular exercise on the lab implementation is basically using an LST, but nonetheless you do recall that while we were studying LSTM and in order to handle down videos, what we had was you would take in and decompose a video in terms of it is frames. Now, you have a CNN from where you get down some sort of latent representation in terms of features, and then these are not exactly your class labels over there for classification, but then these are features which would end up going down as a time series stamped data onto your LSTM in order to come down with prediction per frame basis.

Now, either you can predict all the frames in a series, and what is the class level for them or the other one is where you can predict only the terminal frame over there based on all the previous frames which have come. So, we are going to stick down to the latter, where

you are going to speak just classify the terminal frame based on whatever has happened on all the frames before it. And that is the kind of a classification which we are going to do.

So, we are going to use the same data set, which is this UCF 1 1 data set, where you have 101 different classes of activities which have to be recognized by this kind of a learning engine. Now, as in the earlier case, we had not used all the 101, but just 5 of them. So, here also we are going just going to stick down with 5 of them. And in fact, the data preparation and everything is going to stick down to the same way.

(Refer Slide Time: 02:38)



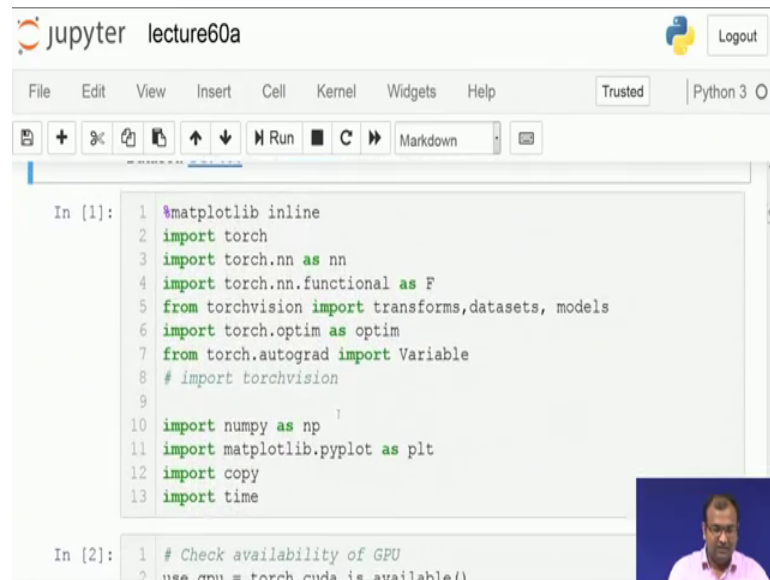
So, let us get in over here so, what we have is we have broken down these lectures into 3 different notebooks to make it easier for you to understand and keep on running. The first one is where we do the preprocessing of training CNN in order to extract out these features over there.

Now, if you look through it the first and foremost thing is that you need to have this lecture 59's preprocessing part already executed, because you would need to have access to these frames extracted from the video and kept down in one particular ordering.

Now, if that is not present and this part does not work; So, that is one part of the exercise which from the last lecture which you need to essentially implement and keep it ready on your side. The next part is that you have three different parts over there which is 60 a, 60

b and 60 c. So, I am starting with 60 a now, and then eventually would be going down to 60 b and 60 c. So, 60 a we are just going to learn down on how to train the first level CNN and not the LSTM yet coming into picture.

(Refer Slide Time: 03:42)



```

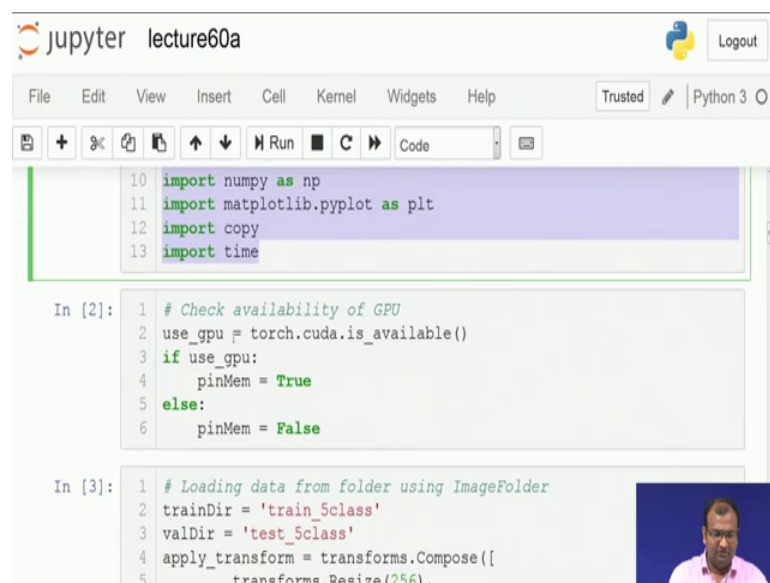
In [1]: 1 %matplotlib inline
        2 import torch
        3 import torch.nn as nn
        4 import torch.nn.functional as F
        5 from torchvision import transforms, datasets, models
        6 import torch.optim as optim
        7 from torch.autograd import Variable
        8 # import torchvision
        9
        10 import numpy as np
        11 import matplotlib.pyplot as plt
        12 import copy
        13 import time

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

```

Now, as in with any kind of a CNN which we had so, we are just going to stick down to the first part of the header, and that is a pretty standard header which we have been using for all of them.

(Refer Slide Time: 03:55)



```

10 import numpy as np
11 import matplotlib.pyplot as plt
12 import copy
13 import time

In [2]: 1 # Check availability of GPU
        2 use_gpu = torch.cuda.is_available()
        3 if use_gpu:
        4     pinMem = True
        5 else:
        6     pinMem = False

In [3]: 1 # Loading data from folder using ImageFolder
        2 trainDir = 'train_5class'
        3 valDir = 'test_5class'
        4 apply_transform = transforms.Compose([
        5     transforms.Resize(256),

```

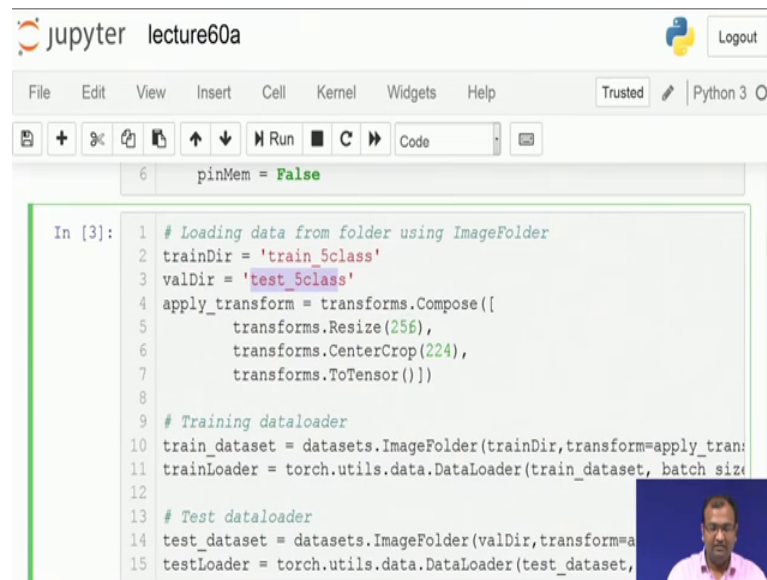
Now, from there it is to check down if GPU is available or if it is not available and. So, what we do addition to that one is that we stick down this extra configuration which is called as pin memory, and pin memory is basically something which is hardware dependent. So, what it essentially does is that your CPU has multiple number of buses over there, which connect down to your hardware resources including your RAM including the PCI bus, and then the dm allocation over there.

So, if pin memory set down as true. So, what it allows is that across different batches which are getting loaded down over there, because you have some sort of a parallel data loader coming into play. Now, once this data loader comes into play if the memory is channel is defined over that, then the arbitrator does not come into play over there and this will speed up the whole process. So, this is what we need to define over here.

Now, in case that you do not have a GPU, then you can pretty much set it to false and that is something which is recommended. The only concern about putting the pin memory true for using within a GPU was so that because it is a shared bus between your CPU to your RAM and your PCI bus via which your GPU is connected. And if you are setting this true then this arbitration does not have a lot of overhead time requirement coming down over there.

So, this is the only reason over there, you can implement the same thing with all your earlier exercises as well. But you will not be seeing too much of a change in terms of execution time. Here the whole change comes in because the amount of data which you are handling which needs to be loaded down from the hard drive side over there is seriously massive. And for that reason, this really makes a significant impact on what would happen with using it or without using it.

(Refer Slide Time: 05:31)



```
pinMem = False

In [3]: 1 # Loading data from folder using ImageFolder
        2 trainDir = 'train_5class'
        3 valDir = 'test_5class'
        4 apply_transform = transforms.Compose([
        5     transforms.Resize(256),
        6     transforms.CenterCrop(224),
        7     transforms.ToTensor()])
        8
        9 # Training dataloader
       10 train_dataset = datasets.ImageFolder(trainDir, transform=apply_tran
       11 trainLoader = torch.utils.data.DataLoader(train_dataset, batch size
       12
       13 # Test dataloader
       14 test_dataset = datasets.ImageFolder(valDir, transform=a
       15 testLoader = torch.utils.data.DataLoader(test_dataset,
```

So, the next part is we basically have two different folders over there, which is created down from the lecture 59 process preprocessing part of the code. And that is one of them is called as the train underscore 5 class, which is at the training folder, and the other one is your testing data folder over there.

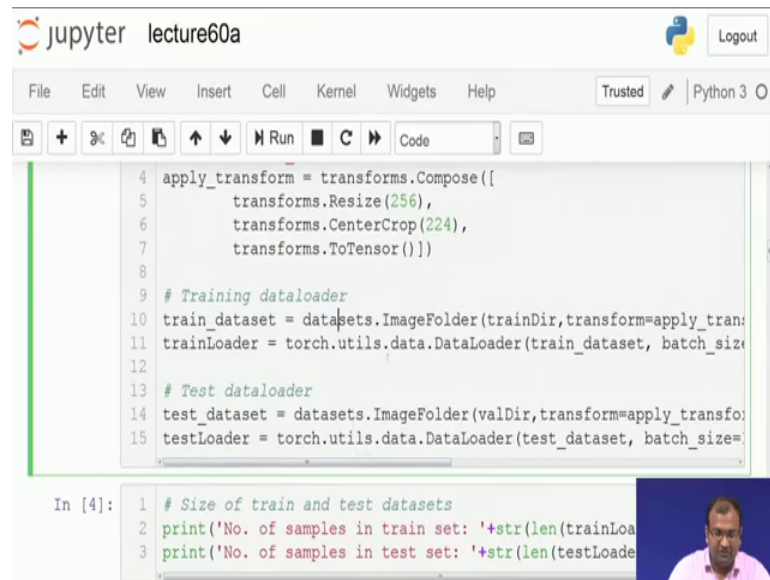
Now, what we need to do is that we take in all of these videos, and we resize them to 256, 256 size, and then we do a center crop of 224 cross 224. Now one of the reasons for doing the center crop of 224 cross 224 is that; obviously, you your rest of the network over there most of these being ImageNet kind of a network. They would be needing a data of 224 cross 224.

The other part of it is that significant amount of pixels on the boundary over here for these particular frames; they are not in any way informative. And that is the reason why we decide to crop from the bit way middle part over there. Now, this is very much typical to what you are trying to do and what is your philosophy over there.

Now, we have just found out this kind of technique to be much better suited for most of these problems over there. And sometimes if you feel that your peripheral parts of the image also have some info information which is significant towards video analytics and does capture some of the motion information, then please do not discard by cropping it out please retain those parts of the information as well.

The next part is to load down all of these data, now what you have essentially within these two folders is basically a few more folders which have image frames extracted from the video.

(Refer Slide Time: 07:11)



```
4 apply_transform = transforms.Compose([
5     transforms.Resize(256),
6     transforms.CenterCrop(224),
7     transforms.ToTensor()
8 ]
9 # Training dataloader
10 train_dataset = datasets.ImageFolder(trainDir, transform=apply_trans
11 trainLoader = torch.utils.data.DataLoader(train_dataset, batch_size
12
13 # Test dataloader
14 test_dataset = datasets.ImageFolder(valDir, transform=apply_transfo
15 testLoader = torch.utils.data.DataLoader(test_dataset, batch_size=

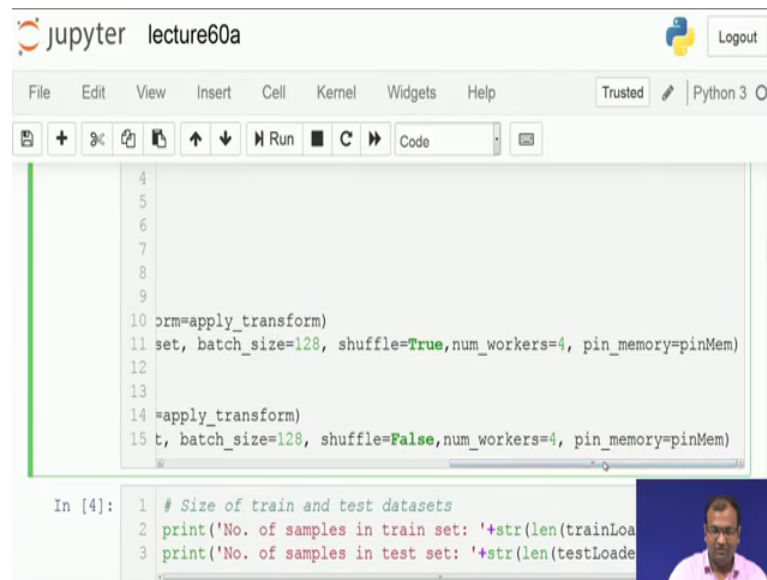
In [4]: 1 # Size of train and test datasets
2 print('No. of samples in train set: '+str(len(trainLoa
3 print('No. of samples in test set: '+str(len(testLoade
```

And so, here the whole job is basically from your image folder, you would be initializing your data loader. So, your trained data set is something which resides inside your image folder. So, this is what we have done in the earlier cases, where your whole data was located within a folder.

So, what you could do is you have to point down a pointer which goes into the directory structure over there, and that acts as initialization pointer for your data loader. And then your trained data loader or as well as your test data loader any of them can initiate their workers to keep on loading it out.

Now, you need to specify your batch size and we are choosing down 128 batch size over here, and the number of data loading workers has 4.

(Refer Slide Time: 07:37)



```
4
5
6
7
8
9
10 orm=apply_transform)
11 set, batch_size=128, shuffle=True,num_workers=4, pin_memory=pinMem)
12
13
14 =apply_transform)
15 t, batch_size=128, shuffle=False,num_workers=4, pin_memory=pinMem)

In [4]: 1 # Size of train and test datasets
        2 print('No. of samples in train set: '+str(len(trainLoa
        3 print('No. of samples in test set: '+str(len(testLoade
```

Now, this number of workers is pretty much dependent on your hardware configuration which is the total number of memory channels which are available to your system. So, if that is 4 then you setting it down to 4 is good. If you are setting it down to any value lesser than that it works out good, but then you are under killing you are you are basically killing down the advantages which you heard about otherwise gone.

And now, if you are setting it to a value greater than 4 then what would happen is that most of the time you will not be able to have the best optimization of resources, because these extra workers will still be part, they are not going to do it because you do not have a physical channel available anyway.

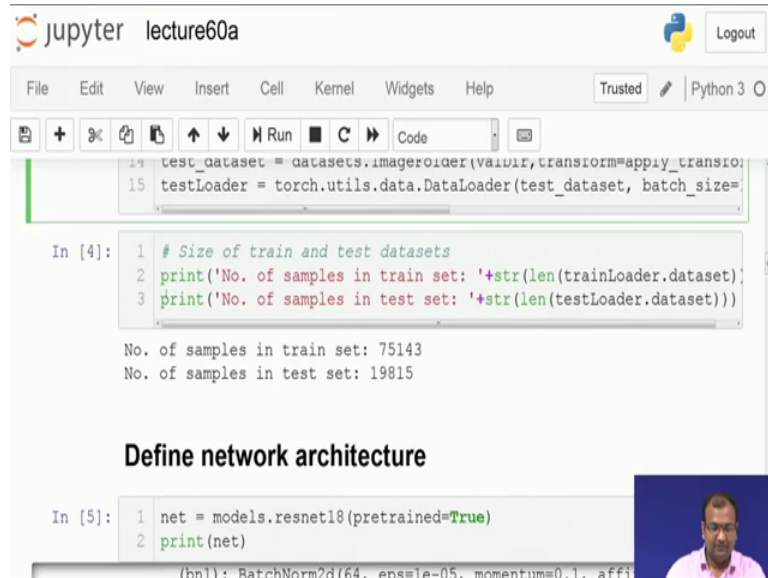
Now, another part is that we decide on shuffling over here, and this is to get into your option of a stochasticity coming into play, which meant that basically if we have a stochastic gradient descent or an Adam kind of an optimizer coming over there.

Then across different Epochs when it starts any invoking this data loader over there, then the same batch at different Epochs should not be having the same amount of data. or the same day should not be having the same samples over there basically.

So, then you do with some sort of a shuffled out so, this shuffling set down to true is something which is helping me do that, but in case of my test data I am not going to use

that shuffle because it is just a feed forward and a validation state. So, in fact, this shuffle has an some extra time incurred on top of it. So, we do not make use of that.

(Refer Slide Time: 09:10)



```
test_dataset = datasets.imagefolder(val_dir, transform=apply_transfo
testLoader = torch.utils.data.DataLoader(test_dataset, batch_size=

In [4]: 1 # Size of train and test datasets
2 print('No. of samples in train set: '+str(len(trainLoader.dataset))
3 print('No. of samples in test set: '+str(len(testLoader.dataset)))

No. of samples in train set: 75143
No. of samples in test set: 19815
```

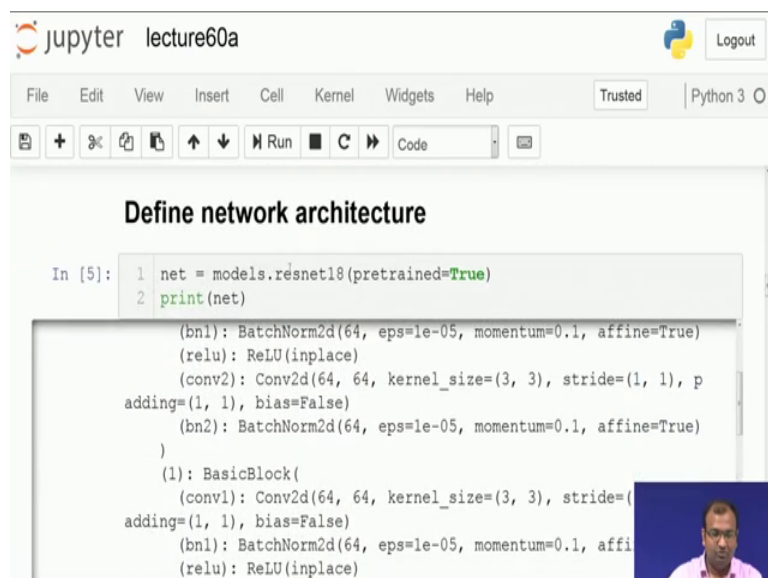
Define network architecture

```
In [5]: 1 net = models.resnet18(pretrained=True)
2 print(net)
```

(bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=

Next so, we just get down what is our total number of train and test samples and this was pretty straightforward.

(Refer Slide Time: 09:12)



```
In [5]: 1 net = models.resnet18(pretrained=True)
2 print(net)
```

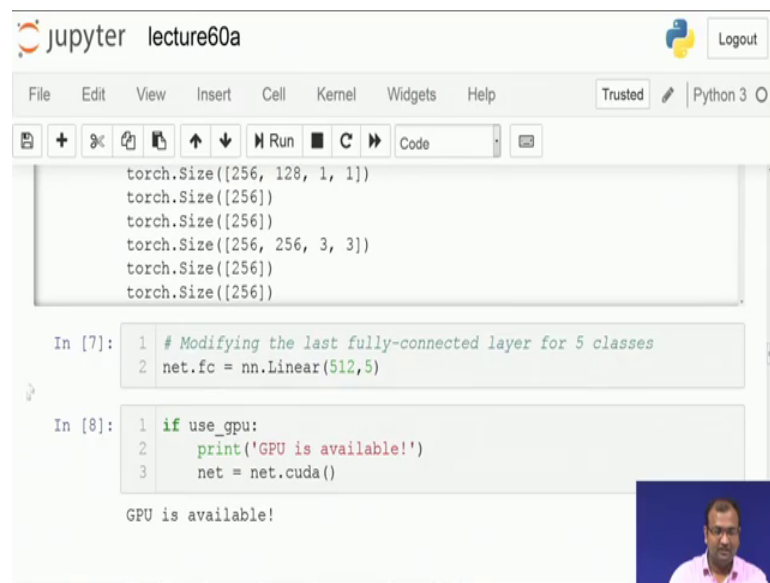
(bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True)
(relu): ReLU(inplace)
(conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
(bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True)
)
(1): BasicBlock(
(conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(
adding=(1, 1), bias=False)
(bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=1)
(relu): ReLU(inplace)

Now, getting into our network what we are trying to do is use a ResNet 18, and we use a pre-trained network which was trained on the ImageNet data set itself. So, that there are 2 advantages one is because it is a pre-trained network. So, it is easier to converge over

there um, and definitely the major advantage which you get down is it since it converges that is no way a great advantage over there; But then your total number of Epochs which we would have otherwise taken to train this whole network that is also going to be less.

So, it is a very faster way of getting down. So, this is one stage of domain adaptation which we also apply to over video analytics problems. So, this is just a printed version of it, and since we have done a lot of them I am not going through these any further.

(Refer Slide Time: 10:10)



```
lecture60a
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 O
torch.Size([256, 128, 1, 1])
torch.Size([256])
torch.Size([256])
torch.Size([256, 256, 3, 3])
torch.Size([256])
torch.Size([256])

In [7]: 1 # Modifying the last fully-connected layer for 5 classes
        2 net.fc = nn.Linear(512,5)

In [8]: 1 if use_gpu:
        2     print('GPU is available!')
        3     net = net.cuda()

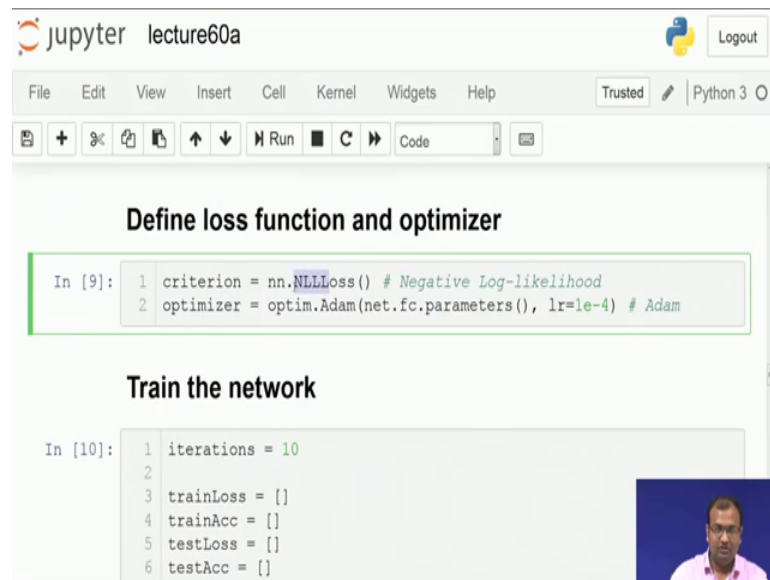
GPU is available!
```

Now, the next point is getting into your parameters and printing it out now this is also straightforward from your earlier exercises, and there is nothing major which comes in. Except for the fact that now we will have to modify this network to be suitable for the kind of a model which we are using.

Now, our job is to classify it into 5 different classes. So, I am just going to replace this last layer over there. So, my last layer was basically on my resnet was something which was connecting 512 neurons on to a 1000 classes, or 1000 neurons over there for classification.

Now, here I do not have 1000 classes, but I just have 5 classes. So, I decide to replace this last layer over there with a linear connection of 512 neurons to 5 neurons straightforward. Now, if a GPU is available just typecast and convert it on to GPU; So, that you can set everything running on the GPU side over there.

(Refer Slide Time: 10:55)



The screenshot shows a Jupyter Notebook interface with the title 'lecture60a'. The menu bar includes File, Edit, View, Insert, Cell, Kernel, Widgets, and Help. The toolbar shows icons for file operations and execution. The notebook content is divided into two sections: 'Define loss function and optimizer' and 'Train the network'. The first section contains a code cell with the following code:

```
In [9]: 1 criterion = nn.NLLLoss() # Negative Log-likelihood
2 optimizer = optim.Adam(net.fc.parameters(), lr=1e-4) # Adam
```

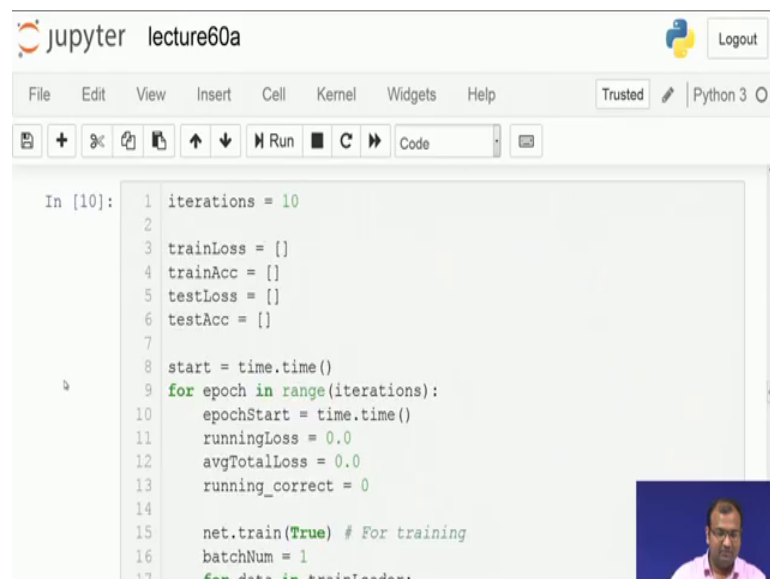
The second section, 'Train the network', contains a code cell with the following code:

```
In [10]: 1 iterations = 10
2
3 trainLoss = []
4 trainAcc = []
5 testLoss = []
6 testAcc = []
```

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

Now, on my criterion now the cost function for training is negative log likelihood loss, and it is because it is a classification problem, and we have been sticking down to the same loss function for majority of them and for optimizer we stick down to taking Adam for faster and better convergence coming into it.

(Refer Slide Time: 11:14)



The screenshot shows the same Jupyter Notebook interface as before, but with the 'Train the network' code cell expanded. The code is as follows:

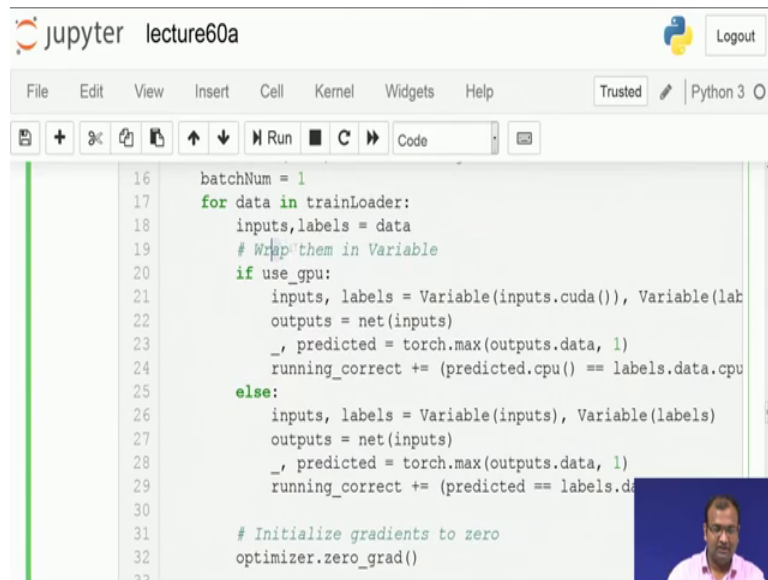
```
In [10]: 1 iterations = 10
2
3 trainLoss = []
4 trainAcc = []
5 testLoss = []
6 testAcc = []
7
8 start = time.time()
9 for epoch in range(iterations):
10     epochStart = time.time()
11     runningLoss = 0.0
12     avgTotalLoss = 0.0
13     running_correct = 0
14
15     net.train(True) # For training
16     batchNum = 1
17     for data in trainLoader:
```

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

Next comes into our training part over there, now for the training it is against straightforward that we just initial initialize these losses which we need to take down over there, and then start our data loader working down over there now within the data

loader, what I am going to essentially do is, look whether my GPU is available, then typecast my inputs and which are input images and the labels over there onto a GPU variable.

(Refer Slide Time: 11:23)

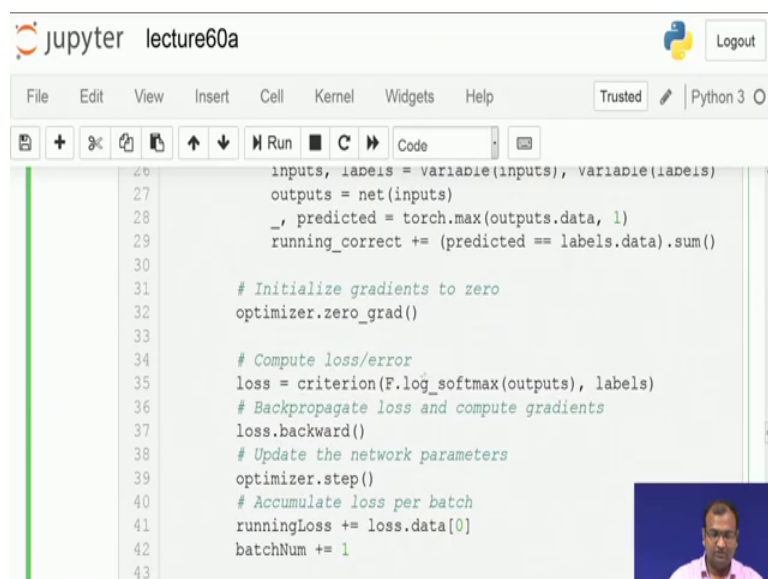


```
16 batchNum = 1
17 for data in trainLoader:
18     inputs, labels = data
19     # Wrap them in Variable
20     if use_gpu:
21         inputs, labels = Variable(inputs.cuda()), Variable(labels.cuda())
22         outputs = net(inputs)
23         _, predicted = torch.max(outputs.data, 1)
24         running_correct += (predicted.cpu() == labels.data.cpu()).sum()
25     else:
26         inputs, labels = Variable(inputs), Variable(labels)
27         outputs = net(inputs)
28         _, predicted = torch.max(outputs.data, 1)
29         running_correct += (predicted == labels.data).sum()
30
31 # Initialize gradients to zero
32 optimizer.zero_grad()
33
```

And then do a feed forward over the network and get my output. And then I have my prediction of the predicted classes over there.

Now, these predicate classes can be used for finding out how many of them are correct and how many of them are wrong.

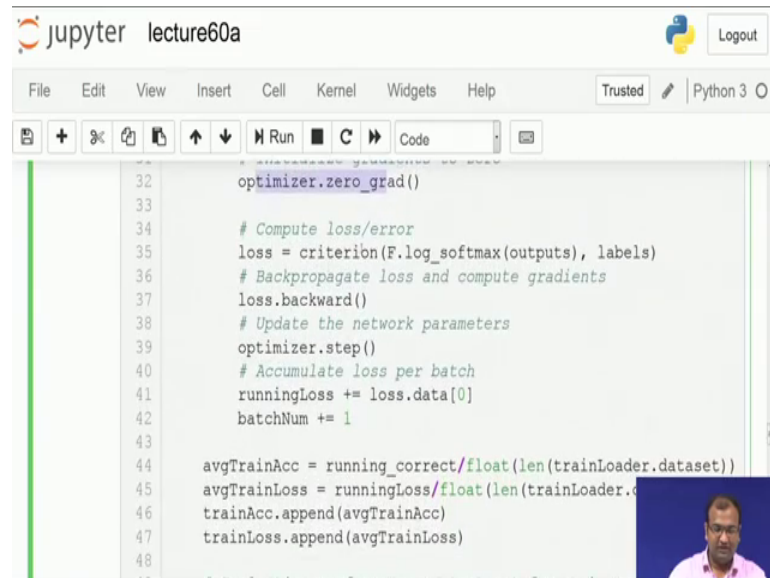
(Refer Slide Time: 11:54)



```
26 inputs, labels = variable(inputs), variable(labels)
27 outputs = net(inputs)
28 _, predicted = torch.max(outputs.data, 1)
29 running_correct += (predicted == labels.data).sum()
30
31 # Initialize gradients to zero
32 optimizer.zero_grad()
33
34 # Compute loss/error
35 loss = criterion(F.log_softmax(outputs), labels)
36 # Backpropagate loss and compute gradients
37 loss.backward()
38 # Update the network parameters
39 optimizer.step()
40 # Accumulate loss per batch
41 runningLoss += loss.data[0]
42 batchNum += 1
43
```

Now, once this part is done, so, I can actually set my optimizer coming into play so, for that the first part is to 0 down all my gradients.

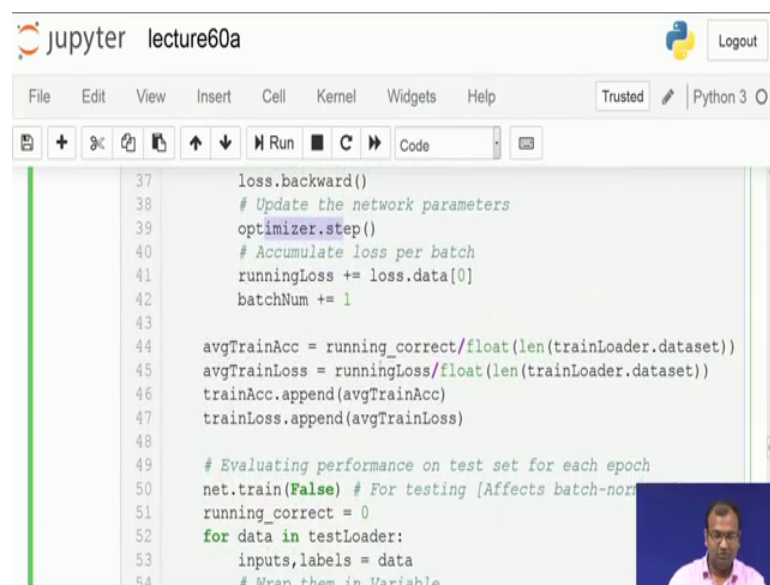
(Refer Slide Time: 12:03)



```
32 optimizer.zero_grad()
33
34 # Compute loss/error
35 loss = criterion(F.log_softmax(outputs), labels)
36 # Backpropagate loss and compute gradients
37 loss.backward()
38 # Update the network parameters
39 optimizer.step()
40 # Accumulate loss per batch
41 runningLoss += loss.data[0]
42 batchNum += 1
43
44 avgTrainAcc = running_correct/float(len(trainLoader.dataset))
45 avgTrainLoss = runningLoss/float(len(trainLoader.dataset))
46 trainAcc.append(avgTrainAcc)
47 trainLoss.append(avgTrainLoss)
48
```

Which we have done, and then you find out what is your total loss. So, this loss is based on the criterion evaluation of negative log likelihood. Then you do a nabla off your loss with this backward, and then you have your optimizer dot step for the update part of the function to run down.

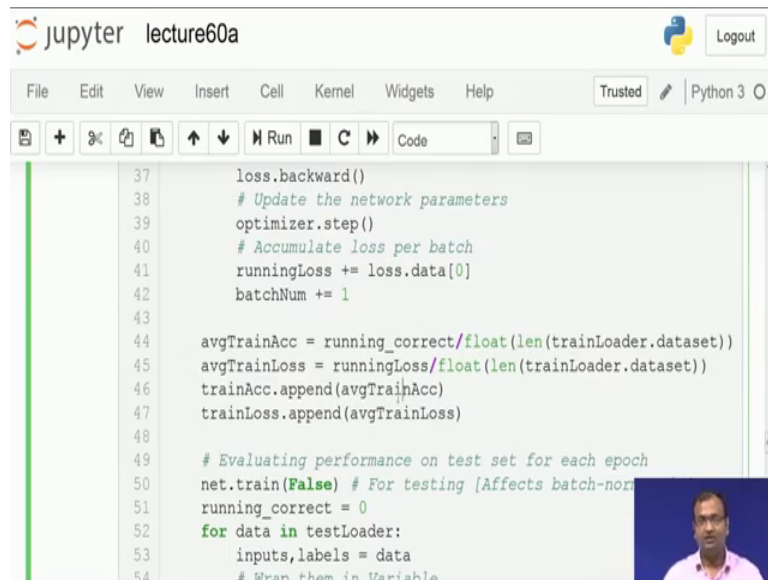
(Refer Slide Time: 12:18)



```
37 loss.backward()
38 # Update the network parameters
39 optimizer.step()
40 # Accumulate loss per batch
41 runningLoss += loss.data[0]
42 batchNum += 1
43
44 avgTrainAcc = running_correct/float(len(trainLoader.dataset))
45 avgTrainLoss = runningLoss/float(len(trainLoader.dataset))
46 trainAcc.append(avgTrainAcc)
47 trainLoss.append(avgTrainLoss)
48
49 # Evaluating performance on test set for each epoch
50 net.train(False) # For testing [Affects batch-normalization]
51 running_correct = 0
52 for data in testLoader:
53     inputs, labels = data
54     # Wrap them in Variable
```

And then you have this whole thing running and finally, now you get your accuracy at the end of one epoch during train. Now once that is done, and your network is updated. What we do is, we put down our validation set into play or this is the test data which was there over there.

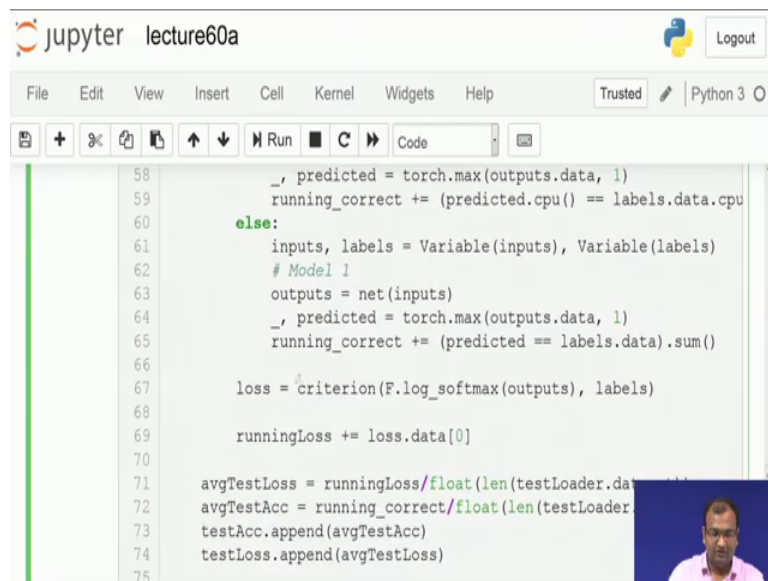
(Refer Slide Time: 12:21)



```
37     loss.backward()
38     # Update the network parameters
39     optimizer.step()
40     # Accumulate loss per batch
41     runningLoss += loss.data[0]
42     batchNum += 1
43
44     avgTrainAcc = running_correct/float(len(trainLoader.dataset))
45     avgTrainLoss = runningLoss/float(len(trainLoader.dataset))
46     trainAcc.append(avgTrainAcc)
47     trainLoss.append(avgTrainLoss)
48
49     # Evaluating performance on test set for each epoch
50     net.train(False) # For testing (Affects batch-normalization)
51     running_correct = 0
52     for data in testLoader:
53         inputs, labels = data
54         # Wrap them in Variable
```

So, you do the same thing you do a forward over there, and then find out what is your loss using your criterion as well as you find out what is your. So, basically what is the total number of correct predictions in order to get your accuracy.

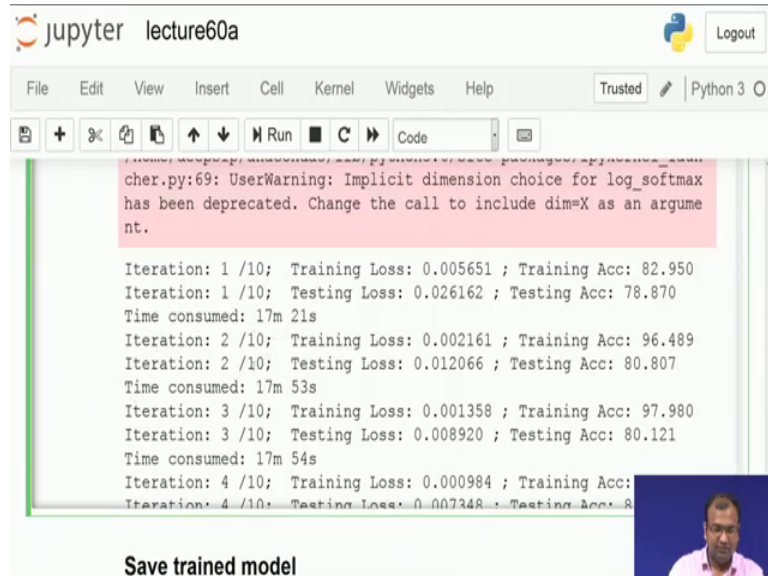
(Refer Slide Time: 12:37)



```
58         _, predicted = torch.max(outputs.data, 1)
59         running_correct += (predicted.cpu() == labels.data.cpu())
60     else:
61         inputs, labels = Variable(inputs), Variable(labels)
62         # Model 1
63         outputs = net(inputs)
64         _, predicted = torch.max(outputs.data, 1)
65         running_correct += (predicted == labels.data).sum().item()
66
67         loss = criterion(F.log_softmax(outputs), labels)
68
69         runningLoss += loss.data[0]
70
71     avgTestLoss = runningLoss/float(len(testLoader.dataset))
72     avgTestAcc = running_correct/float(len(testLoader.dataset))
73     testAcc.append(avgTestAcc)
74     testLoss.append(avgTestLoss)
75
```

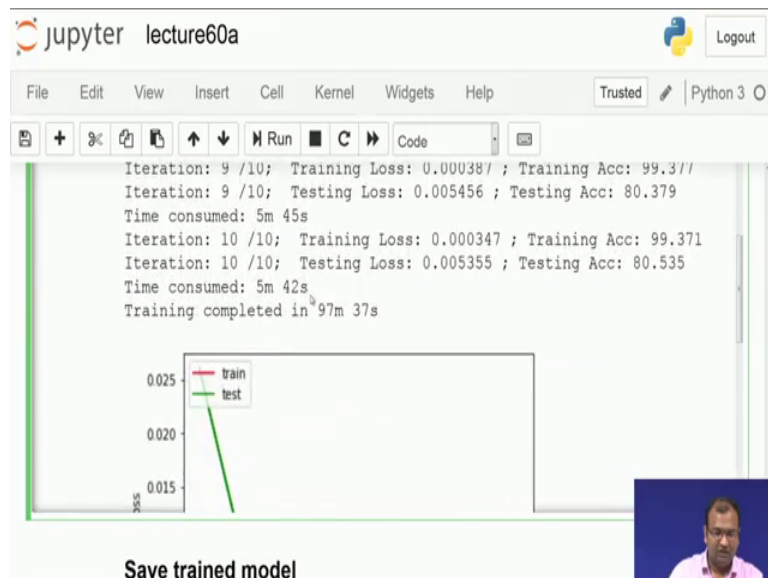
And then you have those also computed and loaded down into your blank tensor. And then comes your standard plotting part of it, and finally, your timing part of it.

(Refer Slide Time: 13:00)



Now, if we keep on running this one for 10 Epochs as we have done over here.

(Refer Slide Time: 13:03)

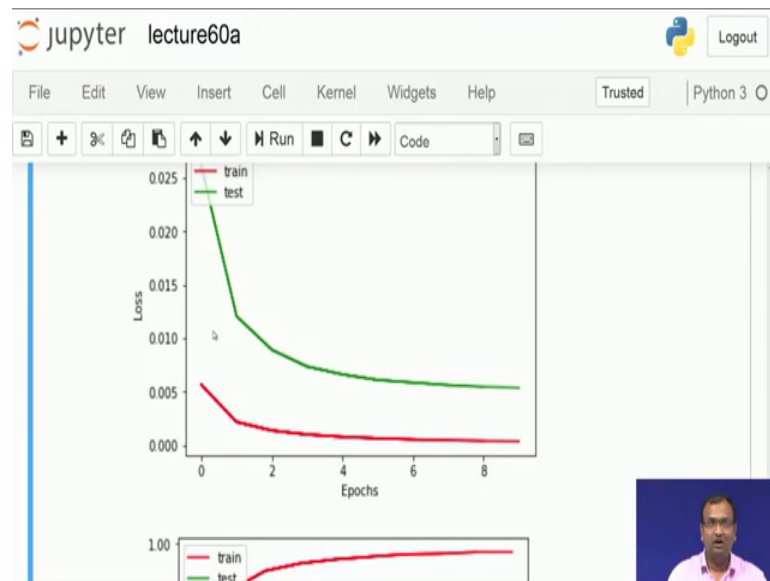


So, it does not take much of time basically so, poor epoch it is it is going to still take down about 5 minutes 42 seconds and in total over all the 10 Epochs, it takes me about 97 minutes. So, that is really about one and a half hours of what it would be taking.

Now, keep in mind one part that these are very dense data a lot of images over there on which you are training.

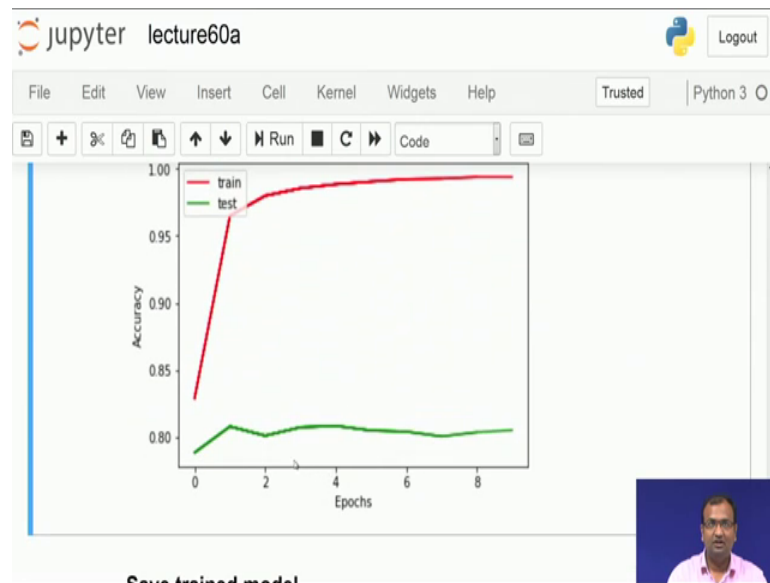
So, the time is not something viewed of one and a half hour. So, these are by now you know that on practical datasets on bigger ones, one of the major challenges which you would be facing done with any kind of a deep neural network is that it is going to take a significant amount of time.

(Refer Slide Time: 13:43)



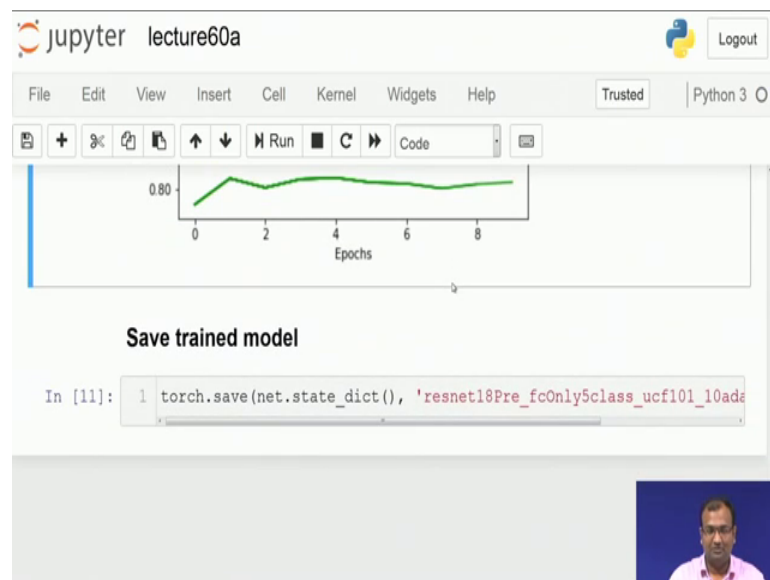
So, if you look into our losses, you would see that your test loss or the validation loss is still above the training loss. So, that is a good symbol that it is it is you have a good amount of generalizability, but you are still not exactly overfitting over there, because you do not have an influx point coming up over there.

(Refer Slide Time: 14:03)



Now, if you look into your accuracy, your accuracy over there for a frame level prediction is somewhere around 80 percent over there.

(Refer Slide Time: 14:17)

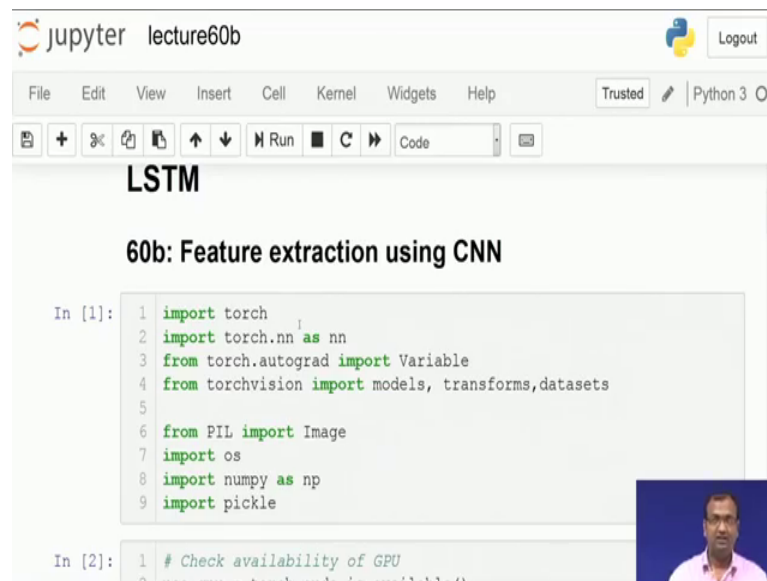


Now, that is something to be very happy around, and in fact, your trained accuracy is something which is going down close to 100 percent like, stay very happy over here because this happiness is something which is quite important.

Now, the next part is and then this also gives you another important part that just a frame level one is 80 percent accurates that is great. Now, on the next part what we do is, we

need to use this network and the weights which are trained over here for another purpose as well, and that is when we try to find out the features for analysium. So, for that what we are trying to do is, we just saved on just the weights in terms of dictionary set. You do not need to save your derivatives or gradients or anything over there, it is just the pure weights over there which need to be saved.

(Refer Slide Time: 14:59)

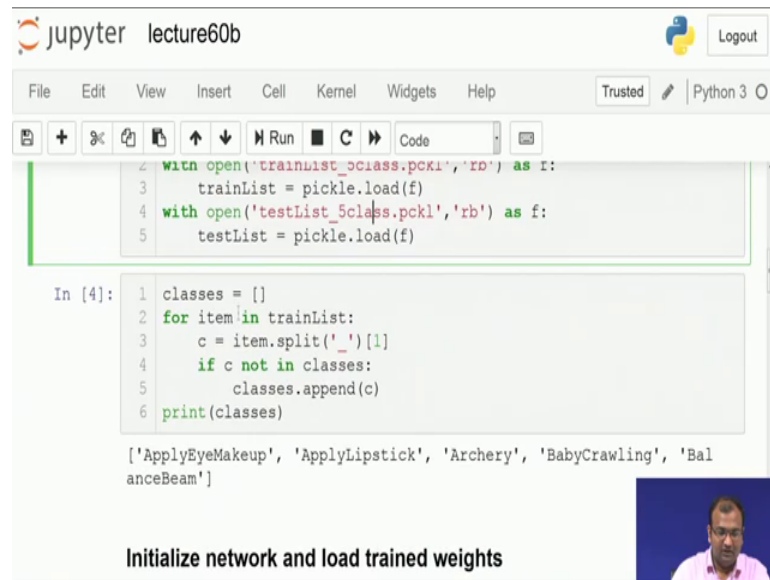


```
Jupyter lecture60b Python 3 O  
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 O  
+ - 🔍 📄 ⬆️ ⬇️ ⬆️ ⬇️ Run 🛑 ↻ ⏩ Code 🗨️  
LSTM  
60b: Feature extraction using CNN  
In [1]:  
1 import torch  
2 import torch.nn as nn  
3 from torch.autograd import Variable  
4 from torchvision import models, transforms, datasets  
5  
6 from PIL import Image  
7 import os  
8 import numpy as np  
9 import pickle  
  
In [2]:  
1 # Check availability of GPU  
2 use_gpu = torch.cuda.is_available()
```

Now, at this point I would move down to the next series of the one which is 60 b. And here what we do is essentially use this earlier pre-trained CNN in order to extract out features which represent one individual frame over there.

And this is important because now what we are going to essentially do is for every single over here; we get down a tensor representation. It is no more an image file, but a single tensor corresponding to that particular image over there.

(Refer Slide Time: 15:52)



```
with open('trainList_0class.pkl', 'rb') as f:
    trainList = pickle.load(f)
with open('testList_5class.pkl', 'rb') as f:
    testList = pickle.load(f)
```

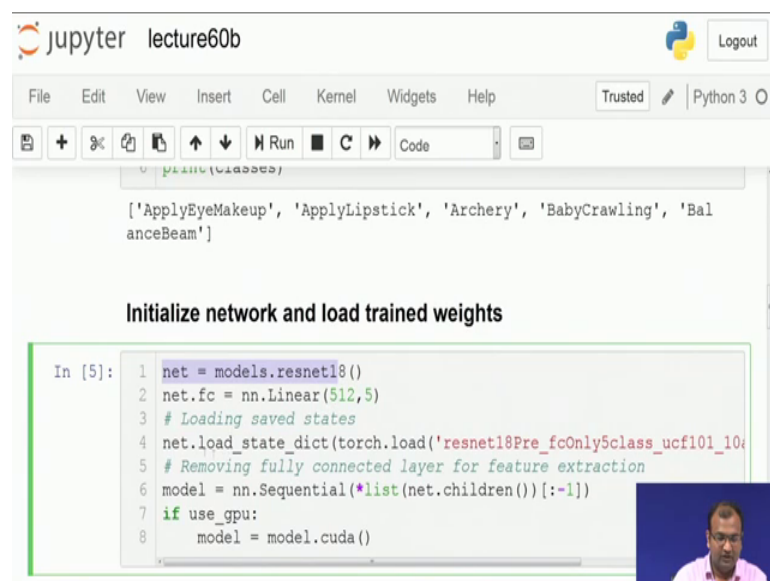
```
In [4]: 1 classes = []
        2 for item in trainList:
        3     c = item.split('_')[1]
        4     if c not in classes:
        5         classes.append(c)
        6 print(classes)
```

['ApplyEyeMakeup', 'ApplyLipstick', 'Archery', 'BabyCrawling', 'BalanceBeam']

Initialize network and load trained weights

So, we start by loading these ones and then look into what is the classes present over there. So, these were the 5 classes which across which we were classifying in the earlier case. So, well you got down somewhere about 88 percent accuracy.

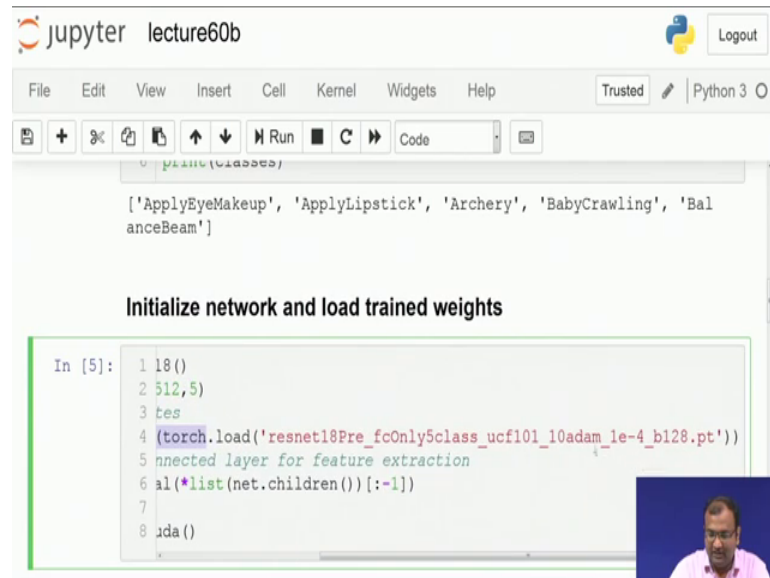
(Refer Slide Time: 16:05)



```
In [5]: 1 net = models.resnet18()
        2 net.fc = nn.Linear(512,5)
        3 # Loading saved states
        4 net.load_state_dict(torch.load('resnet18Pre_fcOnly5class_ucf101_10a
        5 # Removing fully connected layer for feature extraction
        6 model = nn.Sequential(*list(net.children())[:-1])
        7 if use_gpu:
        8     model = model.cuda()
```

Now, the next part is to create your network over here. So, what we are essentially going to do is, you load down so, we are still going to use the resnet 18 over there, ok. Now, on the resnet 18, what we are going to do is, you need to load your dictionary states over there. And so, what these dictionary states are essentially is all the weights which were there from the previously stored one.

(Refer Slide Time: 16:32)



The screenshot shows a Jupyter Notebook window titled 'lecture60b'. The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help), a toolbar with icons for file operations and execution, and a 'Trusted' status indicator. The code cell contains the following Python code:

```
print(classes)

['ApplyEyeMakeup', 'ApplyLipstick', 'Archery', 'BabyCrawling', 'BalanceBeam']

Initialize network and load trained weights

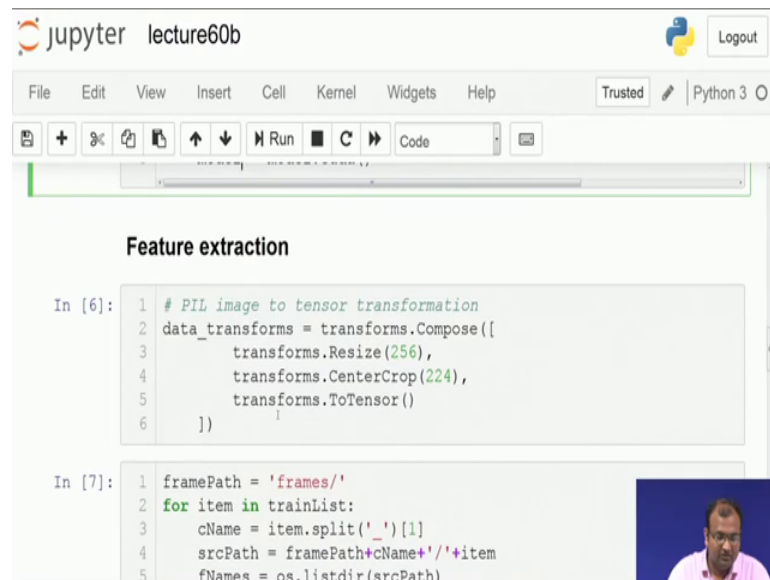
In [5]: 1 18()
        2 512,5)
        3 tes
        4 (torch.load('resnet18Pre_fcOnly5class_ucf101_10adam_1e-4_b128.pt'))
        5 nected layer for feature extraction
        6 al(*list(net.children())[:-1])
        7
        8 .uda()
```

So, this was the file name which we had used for storing down all our weights, in the earlier case and I am I am just going to reload it over here.

Now, once I have that one, what I next do is on my model over here. I will be deleting this last layer. So, my last layer is basically this 512 to 5 neurons over there, and I am just going to delete that part over there so, this thing gets deleted. Now, once that is deleted.

So, I am just left with given an input image over there I am going to get a 512 cross one tensor straightforward. So, every image in terms of just a simple tensor of 512 elements present over there.

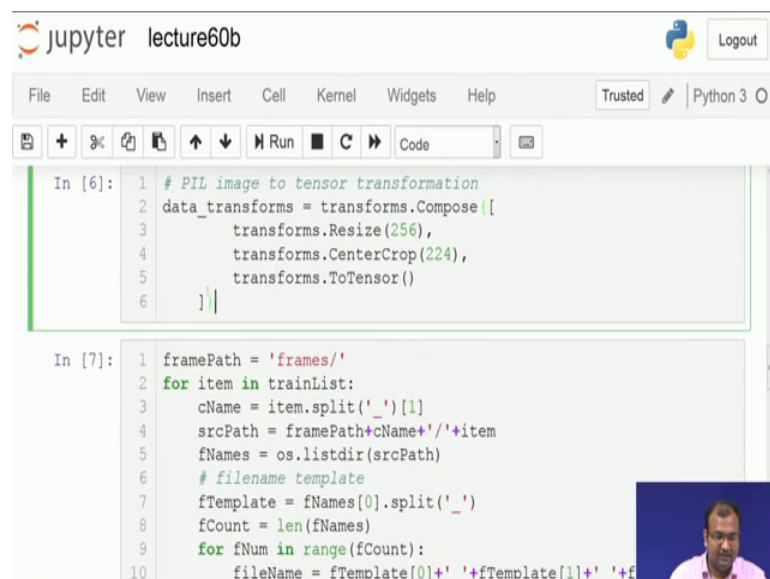
(Refer Slide Time: 17:13)



```
lecture60b Python 3 O
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 O
+ %< > Run C Code
Feature extraction
In [6]: 1 # PIL image to tensor transformation
2 data_transforms = transforms.Compose([
3     transforms.Resize(256),
4     transforms.CenterCrop(224),
5     transforms.ToTensor()
6 ])
In [7]: 1 framePath = 'frames/'
2 for item in trainList:
3     cName = item.split('_')[1]
4     srcPath = framePath+cName+'/'+item
5     fNames = os.listdir(srcPath)
```

Next if we have a GPU we just convert this model to cuda. So, that it is available over there, and now comes our next part of it which is for feature extraction or representing each frame in terms of just this simple one d tensor.

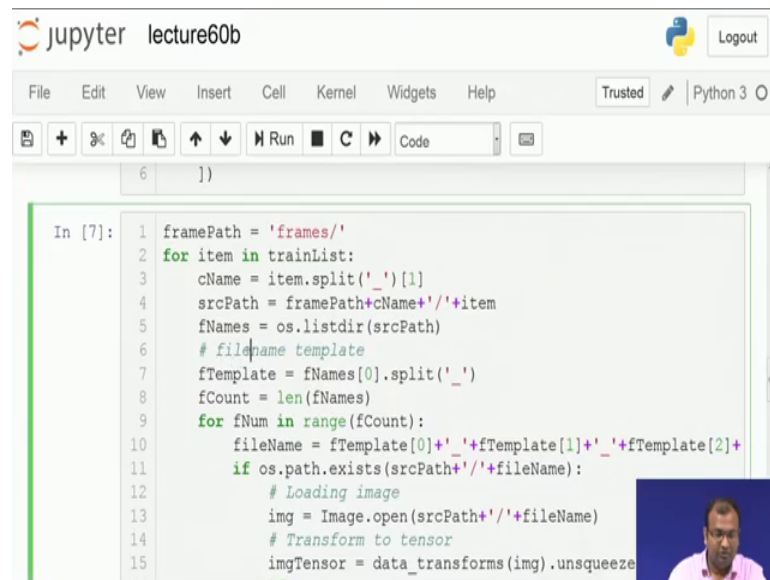
(Refer Slide Time: 17:24)



```
lecture60b Python 3 O
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 O
+ %< > Run C Code
In [6]: 1 # PIL image to tensor transformation
2 data_transforms = transforms.Compose([
3     transforms.Resize(256),
4     transforms.CenterCrop(224),
5     transforms.ToTensor()
6 ])
In [7]: 1 framePath = 'frames/'
2 for item in trainList:
3     cName = item.split('_')[1]
4     srcPath = framePath+cName+'/'+item
5     fNames = os.listdir(srcPath)
6     # filename template
7     fTemplate = fNames[0].split('_')
8     fCount = len(fNames)
9     for fNum in range(fCount):
10        fileName = fTemplate[0]+'_'+fTemplate[1]+'_'+f
```

So, the data transform is still going to remain the same, that you resize it to 5, 256 and then crop the center part for 224 cross 224 sized image over there.

(Refer Slide Time: 17:34)



```

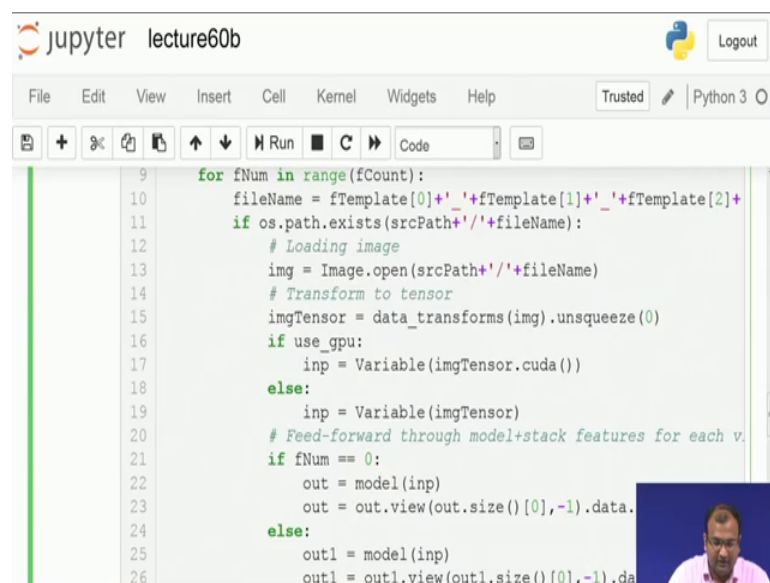
jupyter lecture60b
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 O
Run Code
6 ]])
In [7]: 1 framePath = 'frames/'
2 for item in trainList:
3     cName = item.split('_')[1]
4     srcPath = framePath+cName+'/'+item
5     fName = os.listdir(srcPath)
6     # filename template
7     fTemplate = fName[0].split('_')
8     fCount = len(fNames)
9     for fNum in range(fCount):
10        fileName = fTemplate[0]+'_'+fTemplate[1]+'_'+fTemplate[2]+
11        if os.path.exists(srcPath+'/'+fileName):
12            # Loading image
13            img = Image.open(srcPath+'/'+fileName)
14            # Transform to tensor
15            imgTensor = data_transforms(img).unsqueeze(0)

```

And then what we do is here is, where you are going to convert down each and every frame onto a tensor.

Now, it is independent of whether it is present on my training data set or on my testing data set, I am going to have it for all of them because that is the input which goes down onto my LSTM over there. So, what we essentially do is we find out all the frames present within this frames folder over there.

(Refer Slide Time: 17:54)



```

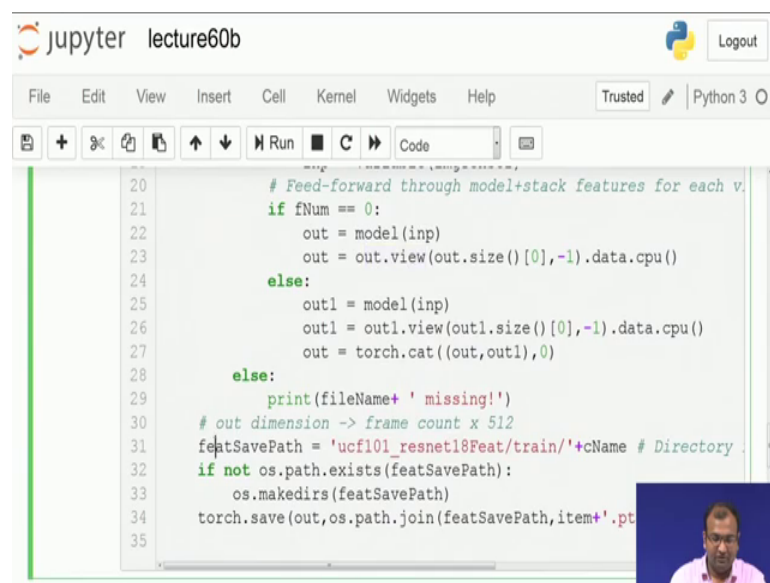
jupyter lecture60b
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 O
Run Code
9     for fNum in range(fCount):
10        fileName = fTemplate[0]+'_'+fTemplate[1]+'_'+fTemplate[2]+
11        if os.path.exists(srcPath+'/'+fileName):
12            # Loading image
13            img = Image.open(srcPath+'/'+fileName)
14            # Transform to tensor
15            imgTensor = data_transforms(img).unsqueeze(0)
16            if use_gpu:
17                inp = Variable(imgTensor.cuda())
18            else:
19                inp = Variable(imgTensor)
20            # Feed-forward through model+stack features for each v.
21            if fNum == 0:
22                out = model(inp)
23                out = out.view(out.size()[0],-1).data.
24            else:
25                out1 = model(inp)
26                out1 = out1.view(out1.size()[0],-1).da

```

And then for each of them this image is opened up and then you have the transformation applied over there.

Now, what you would be doing is, you do a forward pass over my model or the straight pass over this network over there, now keeping one thing in mind that whatever comes out on the output over there, again needs to be recasted into one single tensor over there. And then this is this view operation which sets up my last part of the tensor coming out of it.

(Refer Slide Time: 18:26)

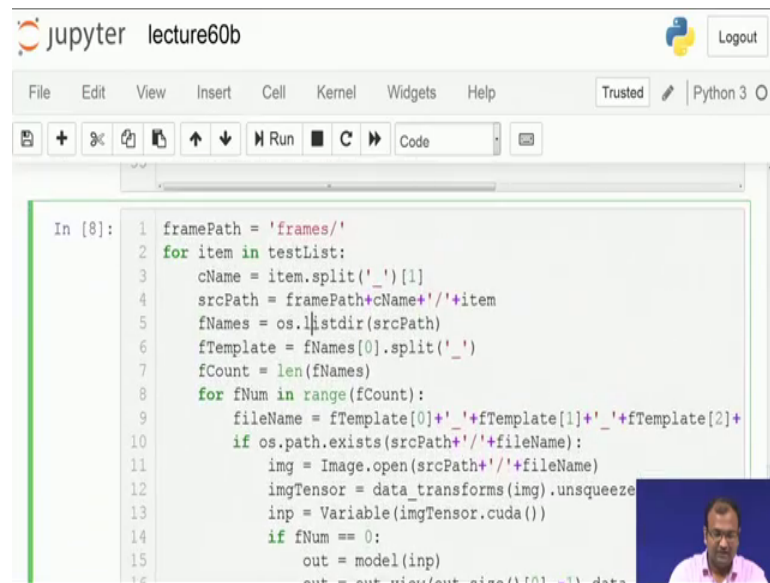


```
20 # Feed-forward through model+stack features for each v
21 if fNum == 0:
22     out = model(inp)
23     out = out.view(out.size()[0],-1).data.cpu()
24 else:
25     out1 = model(inp)
26     out1 = out1.view(out1.size()[0],-1).data.cpu()
27     out = torch.cat((out,out1),0)
28 else:
29     print(fileName+ ' missing!')
30 # out dimension -> frame count x 512
31 featSavePath = 'ucf101_resnet18Feat/train/'+cName # Directory
32 if not os.path.exists(featSavePath):
33     os.makedirs(featSavePath)
34 torch.save(out,os.path.join(featSavePath,item+'.pt'))
35
```

Now, once we have all of this available, then I need to save it out. And so, this is part where I save my tensor in terms of a dot pt file. So, each image now stored in terms of a tensor in terms of a pytorch tensor over there.

Now, once this is done for my training one, I will be doing the same thing for my test list as well.

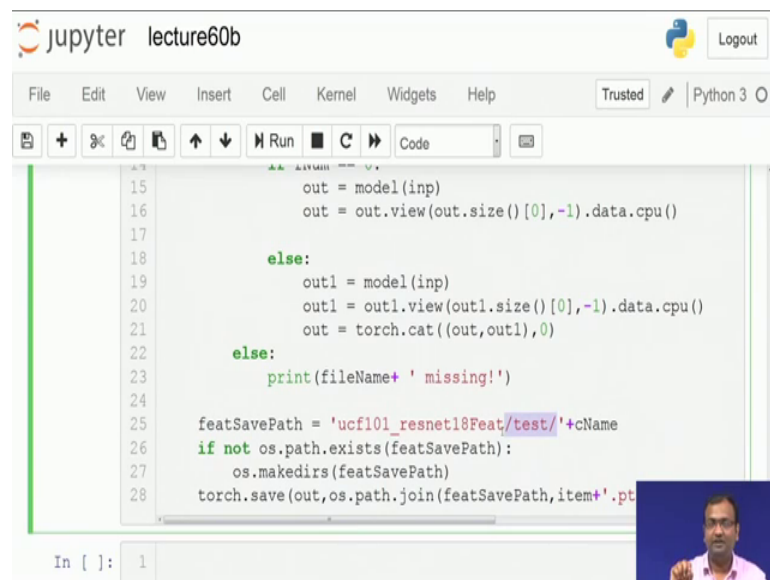
(Refer Slide Time: 18:45)



```
In [8]: 1 framePath = 'frames/'
2 for item in testList:
3     cName = item.split('_')[1]
4     srcPath = framePath+cName+'/'+item
5     fName = os.listdir(srcPath)
6     fTemplate = fName[0].split('_')
7     fCount = len(fNames)
8     for fNum in range(fCount):
9         fileName = fTemplate[0]+'_'+fTemplate[1]+'_'+fTemplate[2]+
10        if os.path.exists(srcPath+'/'+fileName):
11            img = Image.open(srcPath+'/'+fileName)
12            imgTensor = data_transforms(img).unsqueeze
13            inp = Variable(imgTensor.cuda())
14            if fNum == 0:
15                out = model(inp)
16                out = out.view(out.size()[0],-1).data
```

So, this is where I load it down from my test list which I had loaded on my pickle files over here. So, this was my test list which is over here.

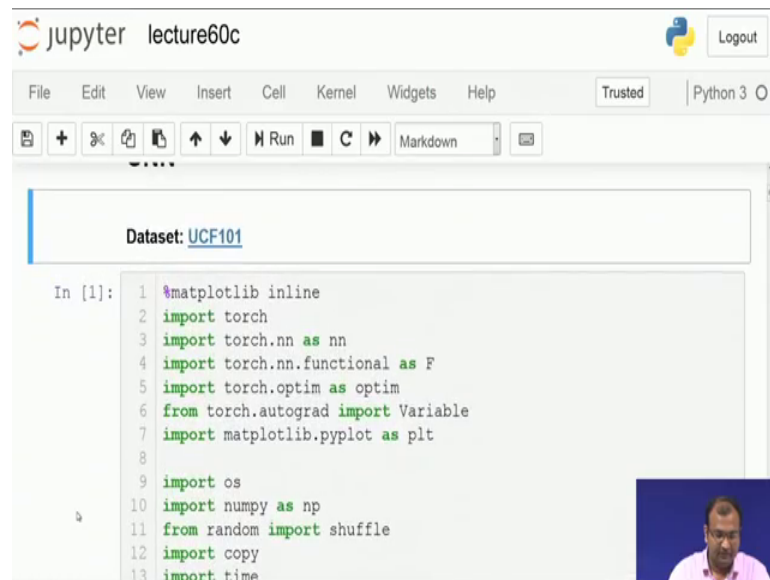
(Refer Slide Time: 19:02)



```
15         out = model(inp)
16         out = out.view(out.size()[0],-1).data.cpu()
17
18     else:
19         out1 = model(inp)
20         out1 = out1.view(out1.size()[0],-1).data.cpu()
21         out = torch.cat((out,out1),0)
22
23     else:
24         print(fileName+ ' missing!')
25
26     featSavePath = 'ucf101_resnet18Feat/test/'+cName
27     if not os.path.exists(featSavePath):
28         os.makedirs(featSavePath)
29     torch.save(out,os.path.join(featSavePath,item+'.pt
```

Now, based on that I am going to again run down everything, and then also stored them out. And now this gets stored in terms of a different directory called as test. So, I have each frame extracted and represented in terms of it is 512 cross 1 1 d tensor. Now, this is the second part which gets over. And then we get into the last and final part which is where you are going to train your LSTM, ok.

(Refer Slide Time: 19:27)

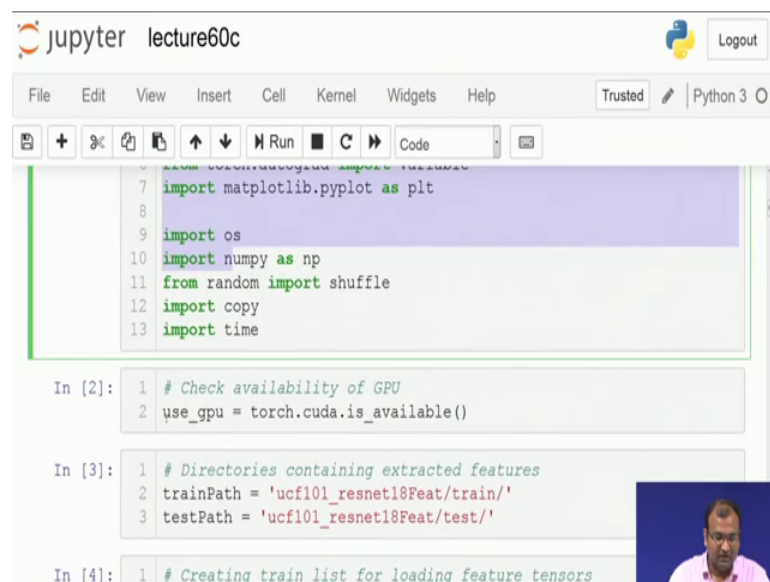


```
Dataset: UCF101

In [1]: 1 %matplotlib inline
        2 import torch
        3 import torch.nn as nn
        4 import torch.nn.functional as F
        5 import torch.optim as optim
        6 from torch.autograd import Variable
        7 import matplotlib.pyplot as plt
        8
        9 import os
       10 import numpy as np
       11 from random import shuffle
       12 import copy
       13 import time
```

Now, within this part, what comes down is we would be for the first part is just your header over there. So, that is straightforward we are not going to make any change over there.

(Refer Slide Time: 19:36)



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

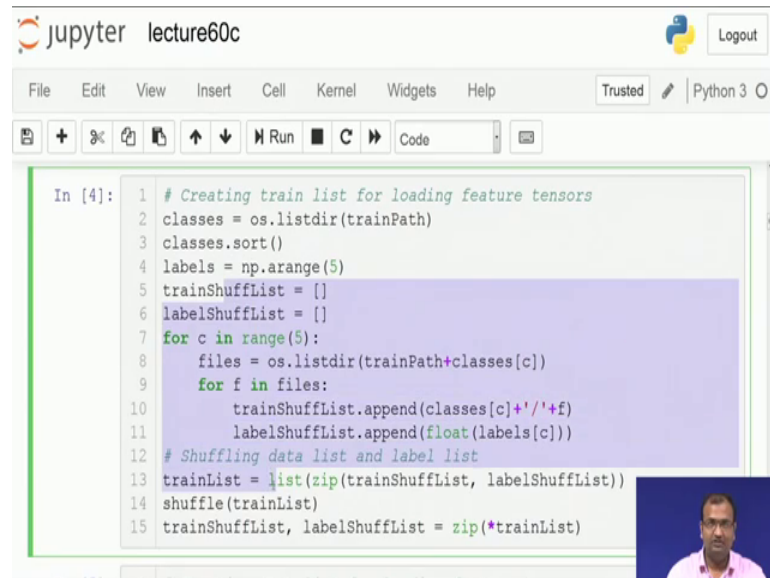
In [3]: 1 # Directories containing extracted features
        2 trainPath = 'ucf101_resnet18Feat/train/'
        3 testPath = 'ucf101_resnet18Feat/test/'

In [4]: 1 # Creating train list for loading feature tensors
```

Now, and you again check out for your GPU, because you are switching away from one python notebook on to the other python notebook session you need to reinvoke and check out all the variables once again as well. And then you again load down your train path and test path, this is to get down what are you where which all images are there in

your training data set and which all images are part of your testing data set over there. And then you basically create out list for each of them in order to facilitate your loader.

(Refer Slide Time: 20:02)

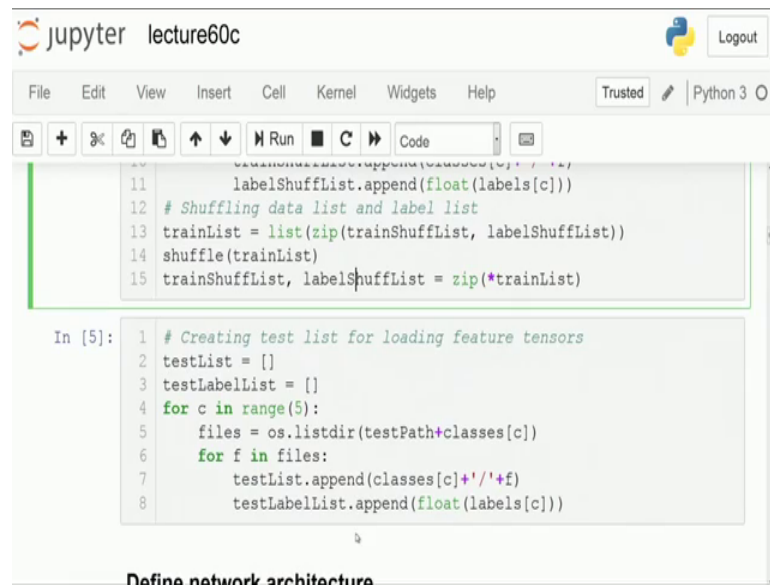


```
In [4]: 1 # Creating train list for loading feature tensors
2 classes = os.listdir(trainPath)
3 classes.sort()
4 labels = np.arange(5)
5 trainShuffList = []
6 labelShuffList = []
7 for c in range(5):
8     files = os.listdir(trainPath+classes[c])
9     for f in files:
10        trainShuffList.append(classes[c]+'/'+f)
11        labelShuffList.append(float(labels[c]))
12 # Shuffling data list and label list
13 trainList = list(zip(trainShuffList, labelShuffList))
14 shuffle(trainList)
15 trainShuffList, labelShuffList = zip(*trainList)
```

Now, one important point which we do over here is that, we need because you are not making use of the data loader any further. Data loader is something which can work only on images. So, here I have dot pt files or pi touch files with all of these tensors present. So, I do not have an option of making use of data loader. So, what I am essentially trying to do is that within each run over there I am trying to shuffle up these indices.

So, in order to shuffle these lists over there what I will have to do is, I have a frame a list of all the frames for one particular example, and corresponding to that I have a class label given now over there. I need to have this pair and shuffle of this pair. And for that reason, the simple way of doing it is, just zip these pair into a zip file, then you can shuffle up this list over here, and then again unzip this folder in order to create get this shuffled up lists over there.

(Refer Slide Time: 21:02)

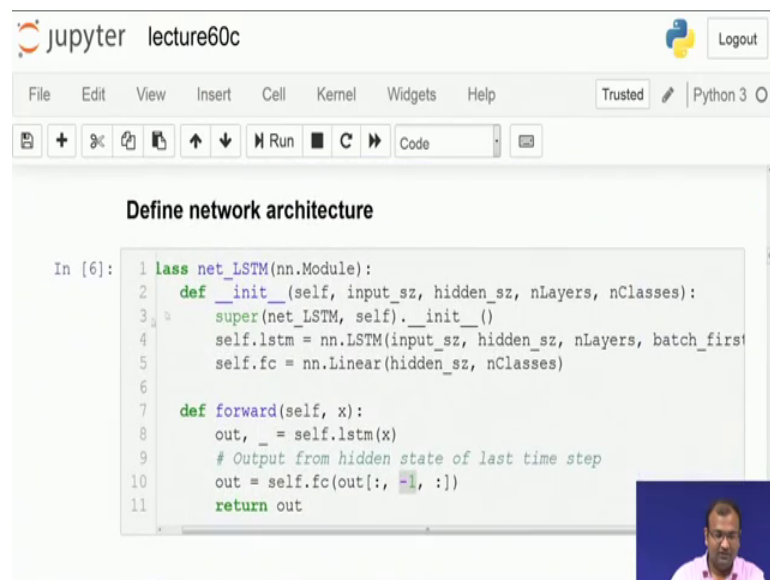


```
11     labelShuffList.append(float(labels[c]))
12     # Shuffling data list and label list
13     trainList = list(zip(trainShuffList, labelShuffList))
14     shuffle(trainList)
15     trainShuffList, labelShuffList = zip(*trainList)

In [5]: 1 # Creating test list for loading feature tensors
2 testList = []
3 testLabelList = []
4 for c in range(5):
5     files = os.listdir(testPath+classes[c])
6     for f in files:
7         testList.append(classes[c]+'/'+f)
8         testLabelList.append(float(labels[c]))
```

So now once that part is done, then we are going to load down all of these features over there.

(Refer Slide Time: 21:10)

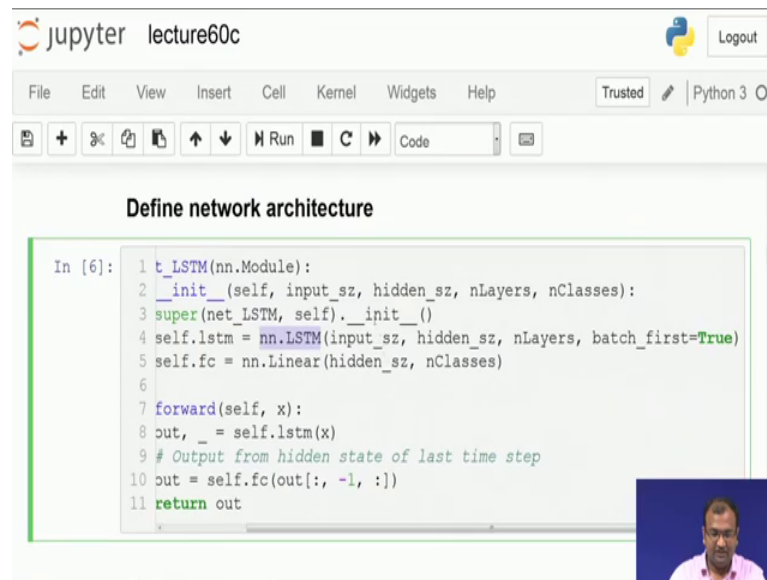


```
Define network architecture

In [6]: 1 class net_LSTM(nn.Module):
2     def __init__(self, input_sz, hidden_sz, nLayers, nClasses):
3         super(net_LSTM, self).__init__()
4         self.lstm = nn.LSTM(input_sz, hidden_sz, nLayers, batch_first=True)
5         self.fc = nn.Linear(hidden_sz, nClasses)
6
7     def forward(self, x):
8         out, _ = self.lstm(x)
9         # Output from hidden state of last time step
10        out = self.fc(out[:, -1, :])
11        return out
```

Now, once it is done we start defining our LSTM. Now the LSTM which we have defined over here is using the model called as nn dot LSTM.

(Refer Slide Time: 21:20)



The screenshot shows a Jupyter Notebook window titled "lecture60c". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help), a toolbar with icons for file operations and execution, and a "Python 3" kernel indicator. The main content area is titled "Define network architecture" and contains a code cell with the following Python code:

```
In [6]: 1 lstm(nn.Module):
2     __init__(self, input_sz, hidden_sz, nLayers, nClasses):
3         super(nn.Module, self).__init__()
4         self.lstm = nn.LSTM(input_sz, hidden_sz, nLayers, batch_first=True)
5         self.fc = nn.Linear(hidden_sz, nClasses)
6
7     forward(self, x):
8         out, _ = self.lstm(x)
9         # Output from hidden state of last time step
10        out = self.fc(out[:, -1, :])
11        return out
```

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

And within that what goes in is basically your input size; input size is something which is mapped from the input tensor size. So, for each time frame over there, I have a 512 cross one tensor, ok.

So, if I have say 20 times stamps over there it becomes 20 cross 512 for me. So, this input size is what goes down over there, next you give down what is the size of the hidden layer. And for our purpose over here, you basically have all the hidden layers with the same number of neurons given down; we do not make any change within this.

Now, you can also give this hidden size as a tensor separate tensor where you can specify what is the total number of neurons across each of them. Now, remember one thing that, there are no convolutional connections or anything they are all fully connected layers. So, I did you do not need to put any further parameters, you can just specify what is the total number of neurons per hidden layer.

Then what is the total number of layers over here? And finally, this rest of the part argument over there which is batch first is equal to true is to set this ket1 over here, because if you remember on our video tensors what we were doing is, you had the batch number, then you had channel number, then you had your time over there, then you had your x and then your y. So, this is how it was going down.

Now, here since we have transform all each frame over there onto a tensor, where what you have is you have your batch number, then you have your timestamp, and then you have your tensor representation. So, if I have say 20 frames over there, and a batch size of 100, and then I have a 100 cross 20, cross 512; because each frame is now represented in terms of a 512 cross one tensor over there.

So, that goes over there, and your final part is just a classification over there, which connects down hidden neurons; the total number of hidden neurons on to a class. So, that is because each new each hidden layer has the same number of hidden neurons.

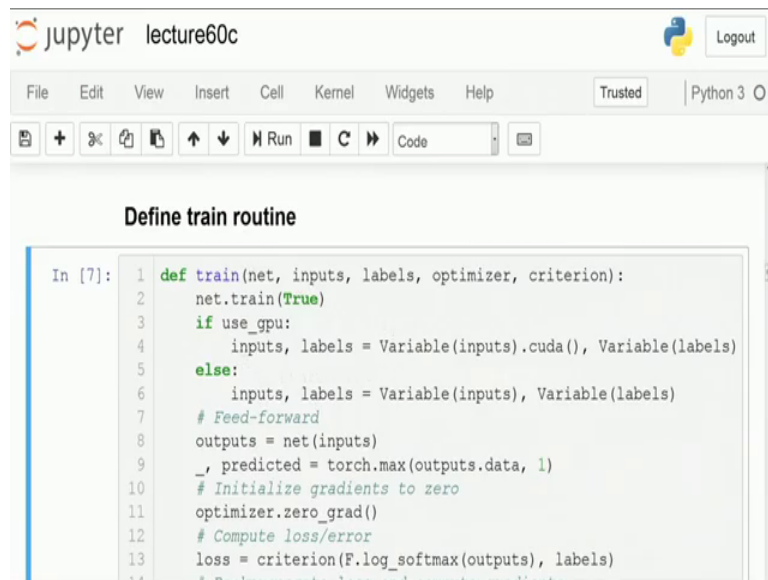
So, we are not making any change over there, if you had passed down basically a tensor over here, defining each different layer in a different way, then you have to have this hidden size as the last hidden layers number of neurons over there. And that maps down to my total number of classes which for me is 5, ok.

Then my forward pass for the LSTM is pretty simple that I whatever is my input, that is what maps down to my output, now remember, I think that this LSTM module inside has all of the rest of the operations for get retain get implemented over there, and how it does time unwrapping and then training within it. So, we do not need to specify each of them we just need to specify the atomic structure for an LSTM block over here.

Now, once you have your output then what you can do is that we just need to take down the state of the last time stamp. Because essentially, if I have 20 frames over there, then I am going to get down a prediction for each of these frames coming out, but I need to get it only for the twentieth frame, and not for 19, 18, 16, 17 up to my first frame. That is that is not necessary for my work over there.

I just want the terminal frames classification coming out so, that is what we are essentially doing over here, that your output for a bunch over here which is for a tensor a times stamp tensor is what is going to give me just one single classification output for this whole time frame volume.

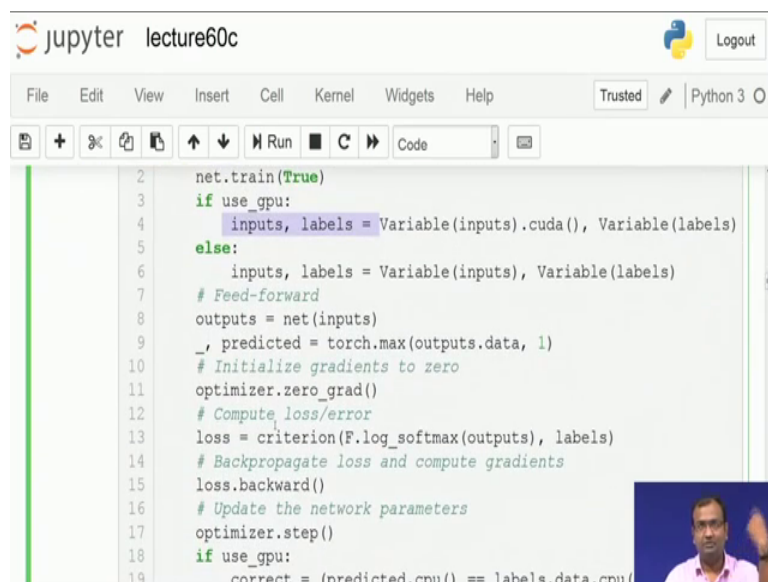
(Refer Slide Time: 24:38)



```
def train(net, inputs, labels, optimizer, criterion):
    net.train(True)
    if use_gpu:
        inputs, labels = Variable(inputs).cuda(), Variable(labels)
    else:
        inputs, labels = Variable(inputs), Variable(labels)
    # Feed-forward
    outputs = net(inputs)
    _, predicted = torch.max(outputs.data, 1)
    # Initialize gradients to zero
    optimizer.zero_grad()
    # Compute loss/error
    loss = criterion(F.log_softmax(outputs), labels)
    # Backpropagate loss and compute gradients
```

Now, then we start our training routine over here. So, within your training routine what you do is, one is just check down for your available GPU one not, and then typecast your inputs and labels over there.

(Refer Slide Time: 24:50)

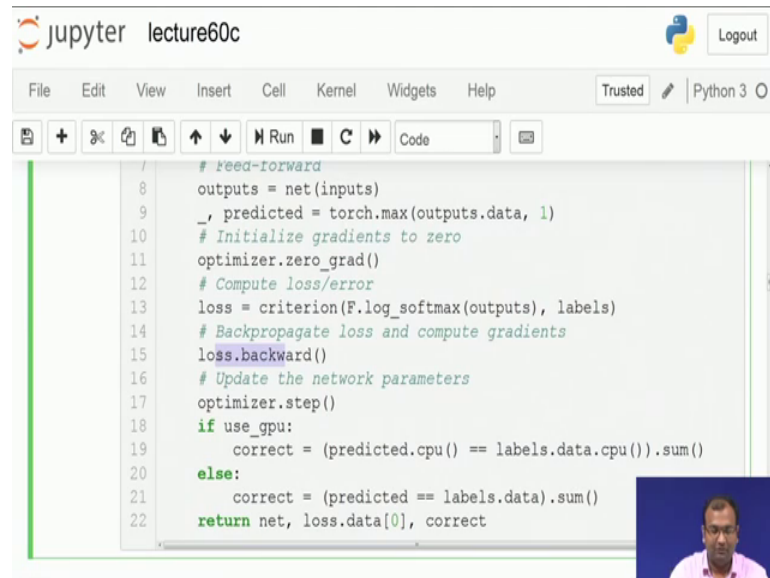


```
net.train(True)
if use_gpu:
    inputs, labels = Variable(inputs).cuda(), Variable(labels)
else:
    inputs, labels = Variable(inputs), Variable(labels)
# Feed-forward
outputs = net(inputs)
_, predicted = torch.max(outputs.data, 1)
# Initialize gradients to zero
optimizer.zero_grad()
# Compute loss/error
loss = criterion(F.log_softmax(outputs), labels)
# Backpropagate loss and compute gradients
loss.backward()
# Update the network parameters
optimizer.step()
if use_gpu:
    correct = (predicted.cpu() == labels.data.cpu())
```

Now, remember your inputs are basically your inputs of that tensor over there, and fetched out in terms of a batch. And label is for each single video block within that black batch what is the label for it, ok.

Next, I do a feed forward get my output over there, and find out what is my prediction coming out of it. Now, once that is done, next is 0 down your gradient. So, that we can start our optimizer, find out your loss using your criterion function, now here also it is a classification problem which we are solving. So, we are going to stick down to the same criteria of a negative log likelihood coming in to over here.

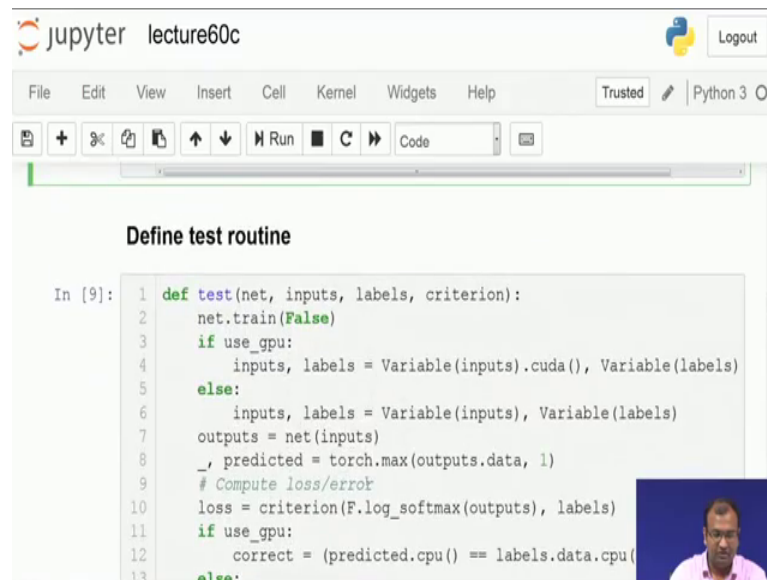
(Refer Slide Time: 25:28)



```
7 # Feed-forward
8 outputs = net(inputs)
9 _, predicted = torch.max(outputs.data, 1)
10 # Initialize gradients to zero
11 optimizer.zero_grad()
12 # Compute loss/error
13 loss = criterion(F.log_softmax(outputs), labels)
14 # Backpropagate loss and compute gradients
15 loss.backward()
16 # Update the network parameters
17 optimizer.step()
18 if use_gpu:
19     correct = (predicted.cpu() == labels.data.cpu()).sum()
20 else:
21     correct = (predicted == labels.data).sum()
22 return net, loss.data[0], correct
```

And then we do a derivative of the criterion function on nabla of the cost function. Then you have your optimizer dot step or the update routine written down over there. Then you are just going to find out whether how many are correctly classified and what is the accuracy which comes over there.

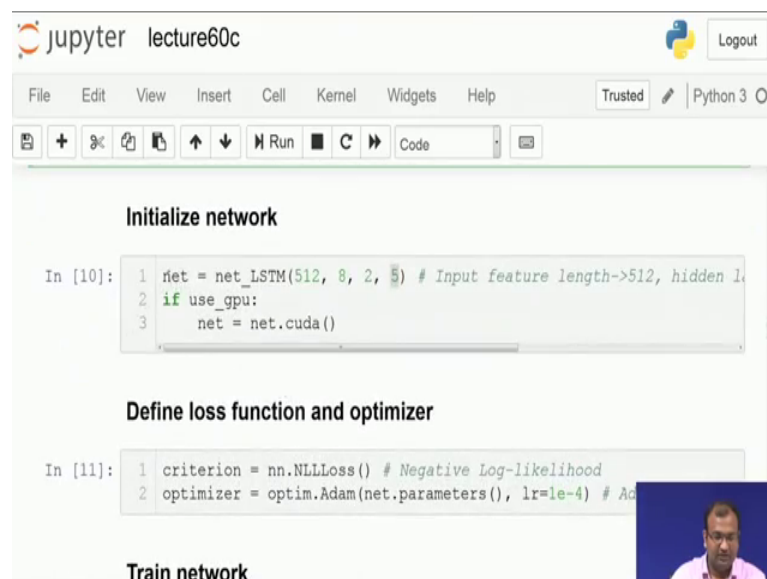
(Refer Slide Time: 25:48)



```
def test(net, inputs, labels, criterion):
    net.train(False)
    if use_gpu:
        inputs, labels = Variable(inputs).cuda(), Variable(labels)
    else:
        inputs, labels = Variable(inputs), Variable(labels)
    outputs = net(inputs)
    _, predicted = torch.max(outputs.data, 1)
    # Compute loss/error
    loss = criterion(F.log_softmax(outputs), labels)
    if use_gpu:
        correct = (predicted.cpu() == labels.data.cpu())
    else:
```

Now, this is what goes down within your training part over there, and then you have your test routine written down in order to do the testing part over there. So, typically in the earlier case we were writing this together, but you are just splitting it out ok.

(Refer Slide Time: 25:55)



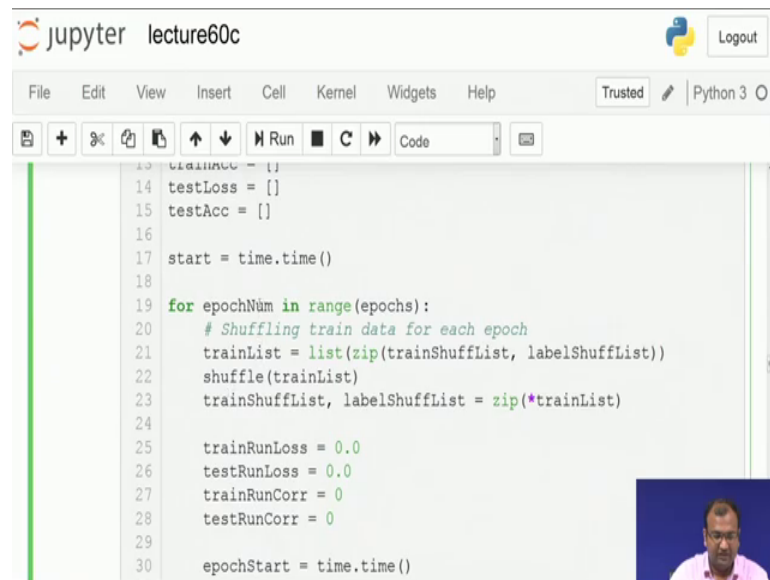
```
net = net_LSTM(512, 8, 2, 5) # Input feature length->512, hidden 1
if use_gpu:
    net = net.cuda()

criterion = nn.NLLLoss() # Negative Log-likelihood
optimizer = optim.Adam(net.parameters(), lr=1e-4) # Adam

net.train()
optimizer.zero_grad()
loss = criterion(net(inputs), labels)
loss.backward()
optimizer.step()
```

Next you can initialize your network. So, what we do over here is my input feature dimension is 512. So, that is my input size which goes over here. I have every hidden layer which has just 8 8 neurons over there. I have two such hidden layers over there, and my total number of outputs is 5. So, this is what I do in order to initialize my LSTM, and

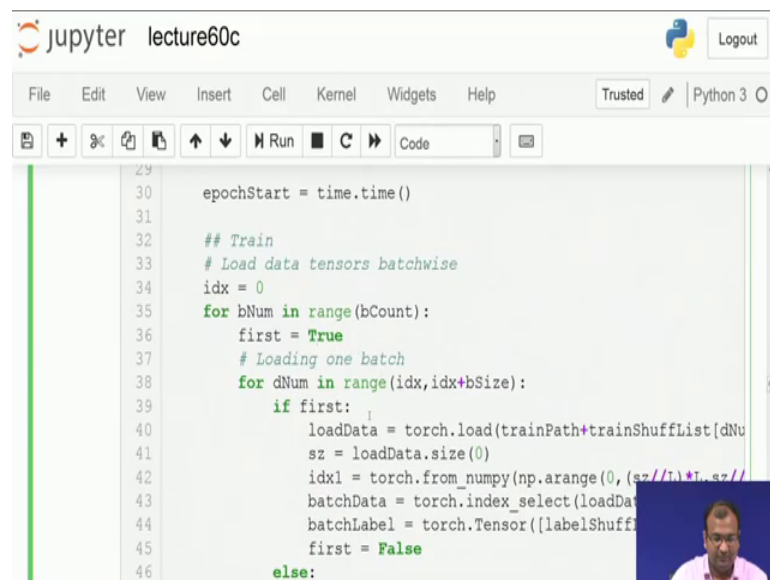
(Refer Slide Time: 27:33)



```
13 trainAcc = []
14 testLoss = []
15 testAcc = []
16
17 start = time.time()
18
19 for epochNum in range(epochs):
20     # Shuffling train data for each epoch
21     trainList = list(zip(trainShuffList, labelShuffList))
22     shuffle(trainList)
23     trainShuffList, labelShuffList = zip(*trainList)
24
25     trainRunLoss = 0.0
26     testRunLoss = 0.0
27     trainRunCorr = 0
28     testRunCorr = 0
29
30     epochStart = time.time()
```

Now, we start across all of our Epochs, and then set down our losses to 0, and then it starts up ok.

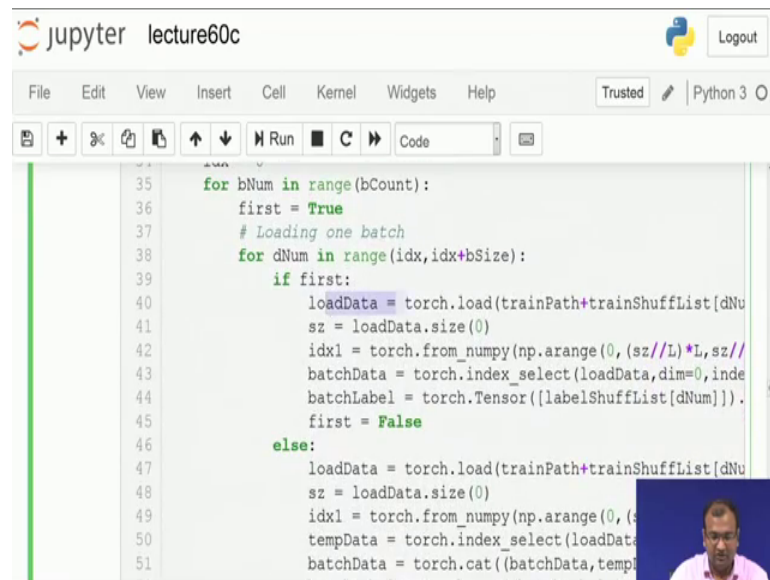
(Refer Slide Time: 27:38)



```
29
30     epochStart = time.time()
31
32     ## Train
33     # Load data tensors batchwise
34     idx = 0
35     for bNum in range(bCount):
36         first = True
37         # Loading one batch
38         for dNum in range(idx, idx+bSize):
39             if first:
40                 loadData = torch.load(trainPath+trainShuffList[dNum])
41                 sz = loadData.size(0)
42                 idx1 = torch.from_numpy(np.arange(0, (sz//T)*T, sz//T))
43                 batchData = torch.index_select(loadData, 0, idx1)
44                 batchLabel = torch.Tensor([labelShuffList[dNum]])
45                 first = False
46             else:
```

Now, once it starts you have your data loader coming into play and this was because we had modified and stored it down in terms of our pickle files, which could not be load it down, sorry not pickle files, but pit wash files which could not be loaded. So, they were just pit wash tensor over there.

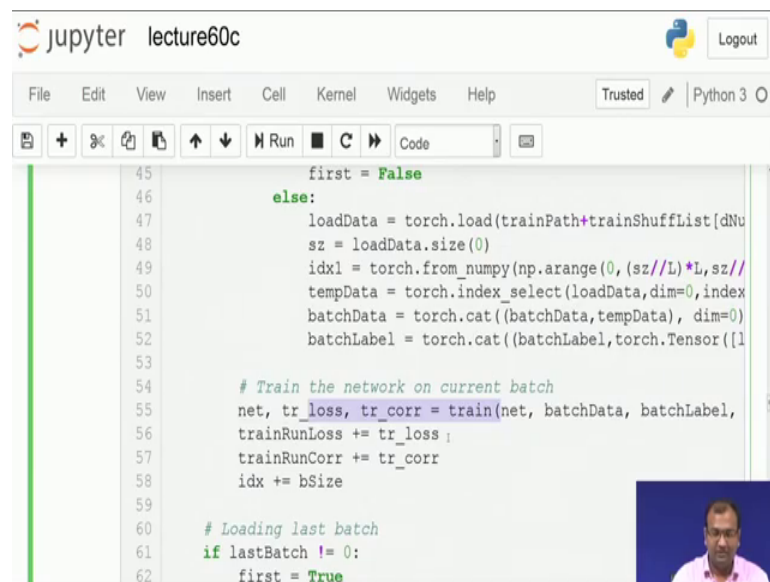
(Refer Slide Time: 27:42)



```
35     for bNum in range(bCount):
36         first = True
37         # Loading one batch
38         for dNum in range(idx, idx+bSize):
39             if first:
40                 loadData = torch.load(trainPath+trainShuffList[dNum])
41                 sz = loadData.size(0)
42                 idx1 = torch.from_numpy(np.arange(0, (sz//L)*L, sz//L))
43                 batchData = torch.index_select(loadData, dim=0, index=idx1)
44                 batchLabel = torch.Tensor([labelShuffList[dNum]])
45                 first = False
46             else:
47                 loadData = torch.load(trainPath+trainShuffList[dNum])
48                 sz = loadData.size(0)
49                 idx1 = torch.from_numpy(np.arange(0, (sz//L)*L, sz//L))
50                 tempData = torch.index_select(loadData, dim=0, index=idx1)
51                 batchData = torch.cat((batchData, tempData), dim=0)
```

Now, what I do from there is, once I load it down I need to convert and get back my labels and my tensors appropriately represented; so, that is what is said down over here.

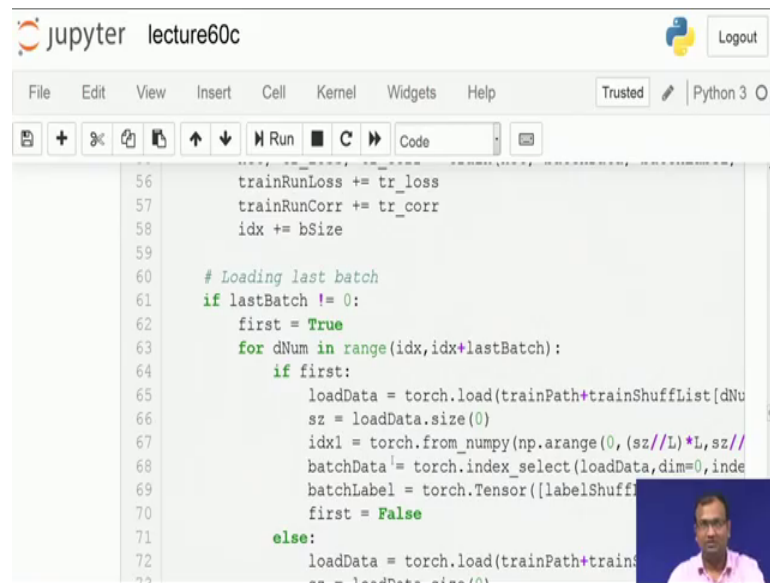
(Refer Slide Time: 28:10)



```
45         first = False
46     else:
47         loadData = torch.load(trainPath+trainShuffList[dNum])
48         sz = loadData.size(0)
49         idx1 = torch.from_numpy(np.arange(0, (sz//L)*L, sz//L))
50         tempData = torch.index_select(loadData, dim=0, index=idx1)
51         batchData = torch.cat((batchData, tempData), dim=0)
52         batchLabel = torch.cat((batchLabel, torch.Tensor([labelShuffList[dNum]])), dim=0)
53
54     # Train the network on current batch
55     net, tr_loss, tr_corr = train(net, batchData, batchLabel, trainRunLoss, trainRunCorr)
56     trainRunLoss += tr_loss
57     trainRunCorr += tr_corr
58     idx += bSize
59
60     # Loading last batch
61     if lastBatch != 0:
62         first = True
```

Now, next part is to get into my training routine over there, and find out what is my loss is coming down. So, this train routine is something which we have written down in the earlier case over here, ok. So, this is my train routine written down for my network, when I am training it out and this is my test routine which has been written.

(Refer Slide Time: 28:33)

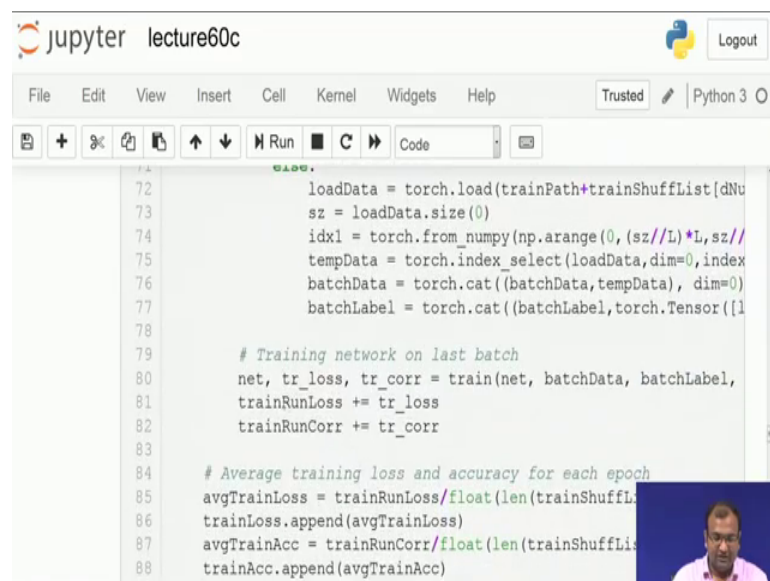


```
56     trainRunLoss += tr_loss
57     trainRunCorr += tr_corr
58     idx += batchSize
59
60     # Loading last batch
61     if lastBatch != 0:
62         first = True
63         for dNum in range(idx, idx+lastBatch):
64             if first:
65                 loadData = torch.load(trainPath+trainShuffList[dNum])
66                 sz = loadData.size(0)
67                 idx1 = torch.from_numpy(np.arange(0, (sz//L)*L, sz//L))
68                 batchData = torch.index_select(loadData, dim=0, index=idx1)
69                 batchLabel = torch.Tensor([labelShuffList[dNum]])
70                 first = False
71             else:
72                 loadData = torch.load(trainPath+trainShuffList[dNum])
73                 sz = loadData.size(0)
```

Now, in the earlier exercises on CNNs and stuff what you were looking down is that we used to freeze a fuse everything together and right, but this is also an option, because I am can now allow myself to do just a pure simple function call over there.

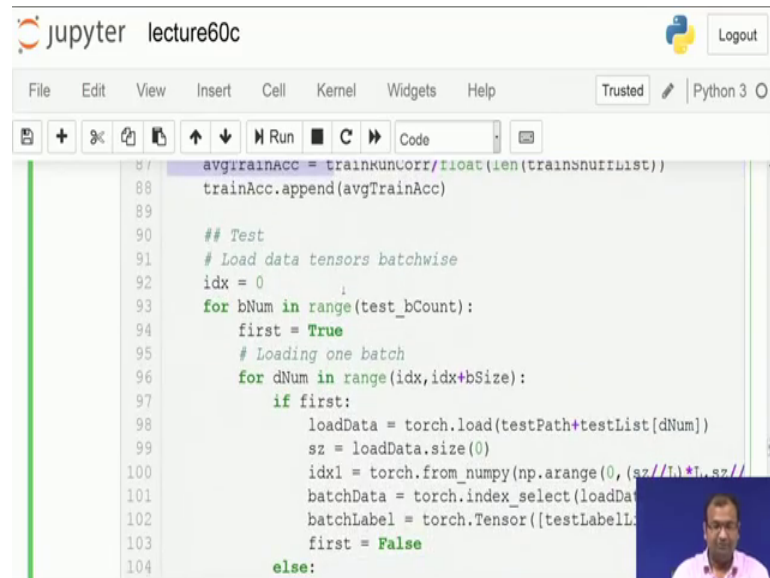
Now, once you have that thing written down, you get down your training losses coming down over there, then you have your train accuracies calculated as well.

(Refer Slide Time: 28:40)



```
72     loadData = torch.load(trainPath+trainShuffList[dNum])
73     sz = loadData.size(0)
74     idx1 = torch.from_numpy(np.arange(0, (sz//L)*L, sz//L))
75     tempData = torch.index_select(loadData, dim=0, index=idx1)
76     batchData = torch.cat((batchData, tempData), dim=0)
77     batchLabel = torch.cat((batchLabel, torch.Tensor([labelShuffList[dNum]])), dim=0)
78
79     # Training network on last batch
80     net, tr_loss, tr_corr = train(net, batchData, batchLabel)
81     trainRunLoss += tr_loss
82     trainRunCorr += tr_corr
83
84     # Average training loss and accuracy for each epoch
85     avgTrainLoss = trainRunLoss/float(len(trainShuffList))
86     trainLoss.append(avgTrainLoss)
87     avgTrainAcc = trainRunCorr/float(len(trainShuffList))
88     trainAcc.append(avgTrainAcc)
```

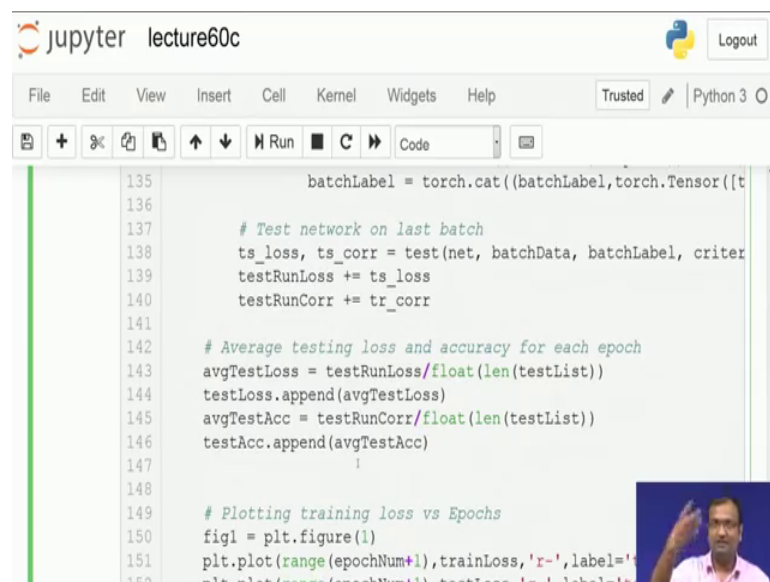
(Refer Slide Time: 28:48)



```
87 avgTrainAcc = trainRunCorr/float(len(trainNumList))
88 trainAcc.append(avgTrainAcc)
89
90 ## Test
91 # Load data tensors batchwise
92 idx = 0
93 for bNum in range(test_bCount):
94     first = True
95     # Loading one batch
96     for dNum in range(idx,idx+bSize):
97         if first:
98             loadData = torch.load(testPath+testList[dNum])
99             sz = loadData.size(0)
100             idx1 = torch.from_numpy(np.arange(0, (sz//1)*1, sz//1))
101             batchData = torch.index_select(loadData, 0, idx1)
102             batchLabel = torch.Tensor([testLabelList[dNum]])
103             first = False
104         else:
```

Now, once that update and everything happens down during your training, then you have your test and validation part being taken care of over here. And then this is for your last batch which you need to take care of.

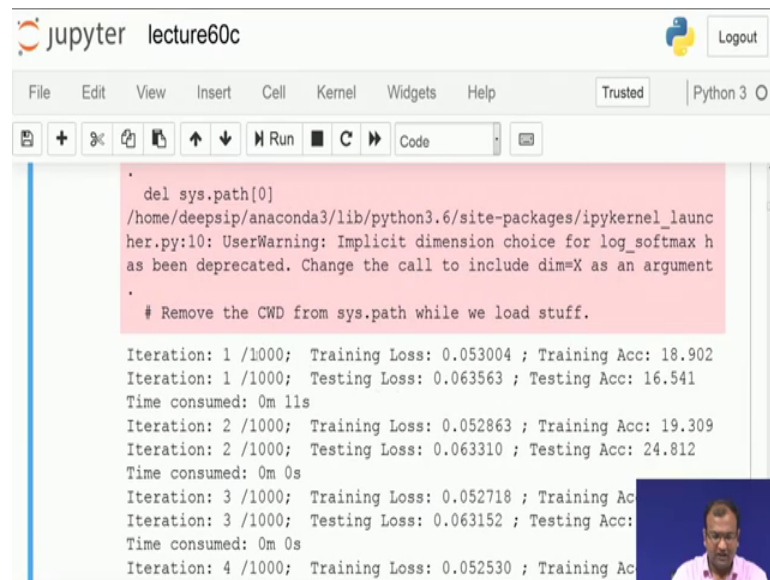
(Refer Slide Time: 29:00)



```
135     batchLabel = torch.cat((batchLabel, torch.Tensor([testLabelList[dNum]])))
136
137     # Test network on last batch
138     ts_loss, ts_corr = test(net, batchData, batchLabel, criterion)
139     testRunLoss += ts_loss
140     testRunCorr += ts_corr
141
142     # Average testing loss and accuracy for each epoch
143     avgTestLoss = testRunLoss/float(len(testList))
144     testLoss.append(avgTestLoss)
145     avgTestAcc = testRunCorr/float(len(testList))
146     testAcc.append(avgTestAcc)
147
148
149 # Plotting training loss vs Epochs
150 fig1 = plt.figure(1)
151 plt.plot(range(epochNum+1), trainLoss, 'r-', label='Train Loss')
152 plt.plot(range(epochNum+1), testLoss, 'g-', label='Test Loss')
```

So, once that is so, once you have everything running over all the regular number of batches so, the last residual number of things is what gets taken care of in the last patch present over here.

(Refer Slide Time: 29:19)



The screenshot shows a Jupyter Notebook window titled 'lecture60c'. The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help), a toolbar with icons for file operations and execution, and a code editor. The code in the cell is:

```
del sys.path[0]
/home/deepsip/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:10: UserWarning: Implicit dimension choice for log_softmax has been deprecated. Change the call to include dim=X as an argument
# Remove the CWD from sys.path while we load stuff.

Iteration: 1 /1000; Training Loss: 0.053004 ; Training Acc: 18.902
Iteration: 1 /1000; Testing Loss: 0.063563 ; Testing Acc: 16.541
Time consumed: 0m 11s
Iteration: 2 /1000; Training Loss: 0.052863 ; Training Acc: 19.309
Iteration: 2 /1000; Testing Loss: 0.063310 ; Testing Acc: 24.812
Time consumed: 0m 0s
Iteration: 3 /1000; Training Loss: 0.052718 ; Training Acc:
Iteration: 3 /1000; Testing Loss: 0.063152 ; Testing Acc:
Time consumed: 0m 0s
Iteration: 4 /1000; Training Loss: 0.052530 ; Training Acc:
```

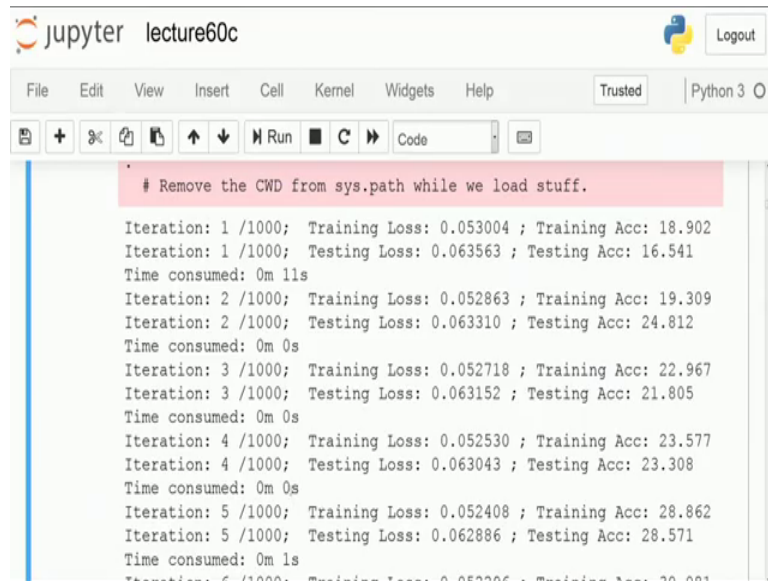
The output shows training and testing metrics for four iterations. A small video inset of a man is visible in the bottom right corner of the notebook window.

Now, having done that, now you can get poor epoch wise what is your average accuracy in and coming out of it, and then you can decide on plotting it out. So, this is straight forward what we do over here, and then we said this one running. If you see this one running this it initially starts over here with the training loss of about 0.05, and an accuracy of 18 percent, and a test accuracy of about 16 percent and takes about 11 seconds to finish off.

So, one thing for sure is that while your CNNs are taking much larger to finish off because it was a residual network and a dense connection over there. Here you just have simple tensors which come down you do not have the whole image going into it, and a fully connected neural network for your LSTM modules over there. So, they the number of parameters is much lesser, so, it takes much faster to converge onto them.

But nonetheless the static accuracy is and a much smaller z1 over there.

(Refer Slide Time: 30:00)



```

# Remove the CWD from sys.path while we load stuff.

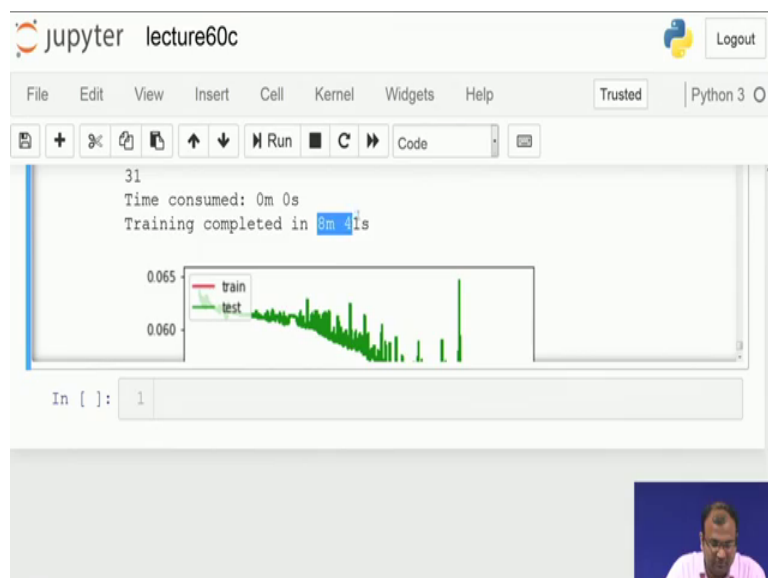
Iteration: 1 /1000; Training Loss: 0.053004 ; Training Acc: 18.902
Iteration: 1 /1000; Testing Loss: 0.063563 ; Testing Acc: 16.541
Time consumed: 0m 11s
Iteration: 2 /1000; Training Loss: 0.052863 ; Training Acc: 19.309
Iteration: 2 /1000; Testing Loss: 0.063310 ; Testing Acc: 24.812
Time consumed: 0m 0s
Iteration: 3 /1000; Training Loss: 0.052718 ; Training Acc: 22.967
Iteration: 3 /1000; Testing Loss: 0.063152 ; Testing Acc: 21.805
Time consumed: 0m 0s
Iteration: 4 /1000; Training Loss: 0.052530 ; Training Acc: 23.577
Iteration: 4 /1000; Testing Loss: 0.063043 ; Testing Acc: 23.308
Time consumed: 0m 0s
Iteration: 5 /1000; Training Loss: 0.052408 ; Training Acc: 28.862
Iteration: 5 /1000; Testing Loss: 0.062886 ; Testing Acc: 28.571
Time consumed: 0m 1s
Iteration: 6 /1000; Training Loss: 0.052266 ; Training Acc: 29.081

```

And if you look over here so, it shows down at 0 second. So, technically it is not like 0, but it is a much lesser value. In fact, the finish off in a few milliseconds and you see this accuracy starting down about 11 percent and then it keeps on going down.

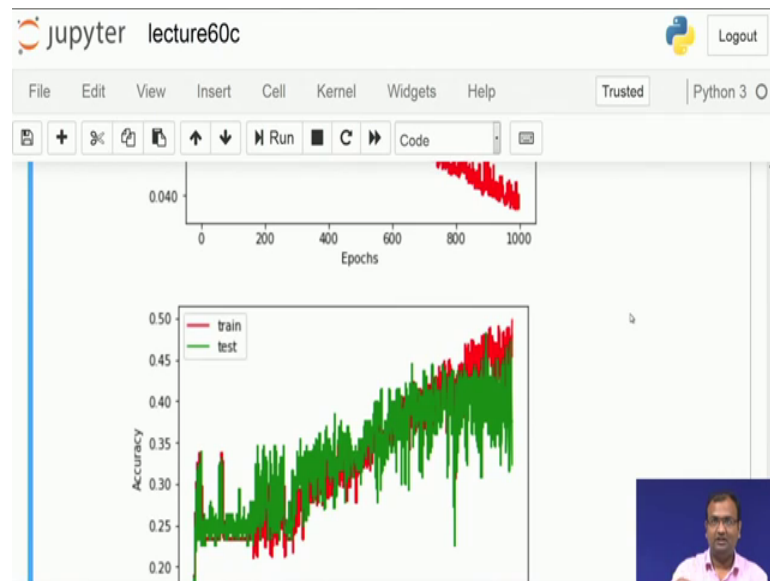
Now, if we go till the end of this routine, then we would be select me yeah, ok. So, just come down yeah so, what 800, 900 and then you are close to a 1000 Epochs over here ok. So, the total time it took for me to converge over 1000 Epochs is just merely 8 minutes over there.

(Refer Slide Time: 30:48)



Now, let us look into the accuracy graphs over there.

(Refer Slide Time: 30:54)



It starts with about 25 percent of an accuracy, and then here by the end of 1000 Epochs, my test accuracy somewhere barely about 45 percent. My train accuracy is about 50 percent; now that is not an impressive figure given the fact that by CNN already had an 88 percent of test case accuracy over there.

But nonetheless you need to keep another thing in mind, that is the this whole network over here has not yet actually converged in any way it still shows a increasing trained in terms of it is accuracy as well as a falling trained in terms of it is loss which is coming down over here.

And that keeps in mind one thing that we need to train it for more and more number of Epochs as well. So, you cannot keep on doing this for beyond 1000 Epochs and typically I would be suggesting about going down to close to 10000 or even more number of Epochs in order to see your saturation coming down.

So, that is where we come to an end for understanding LSTM for video analysis and end of this lecture series as such. So, for those of you who are taking down the test of the final examination as well, so, we wish you all the best for your exams and all endeavors in future and life.

Well then thanks and we wish you good bye from this course.