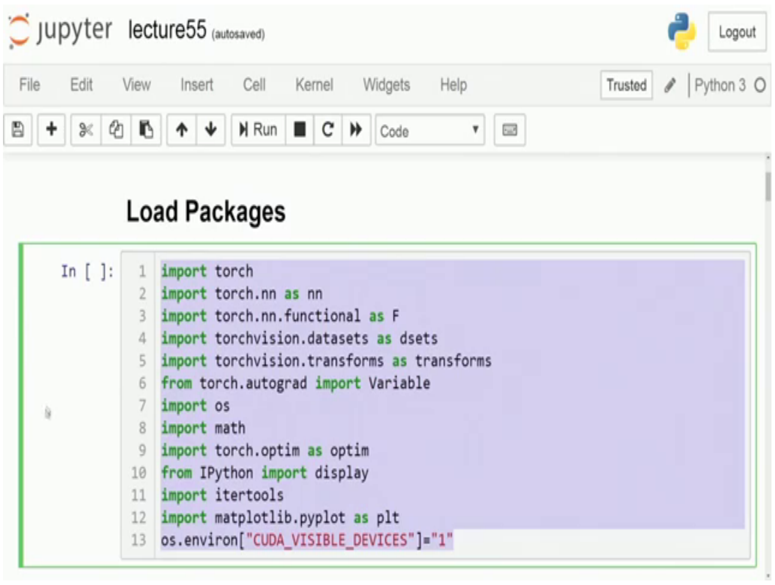
Deep Learning for Visual Computing

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

Lecture – 55
Adversarial Autoencoder for Classification

(Refer Slide Time: 00:18)



Load Packages

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision.datasets as dsets
import torchvision.transforms as transforms
from torch.autograd import Variable
import os
import math
import torch.optim as optim
from IPython import display
import itertools
import matplotlib.pyplot as plt
os.environ["CUDA_VISIBLE_DEVICES"]="1"
```

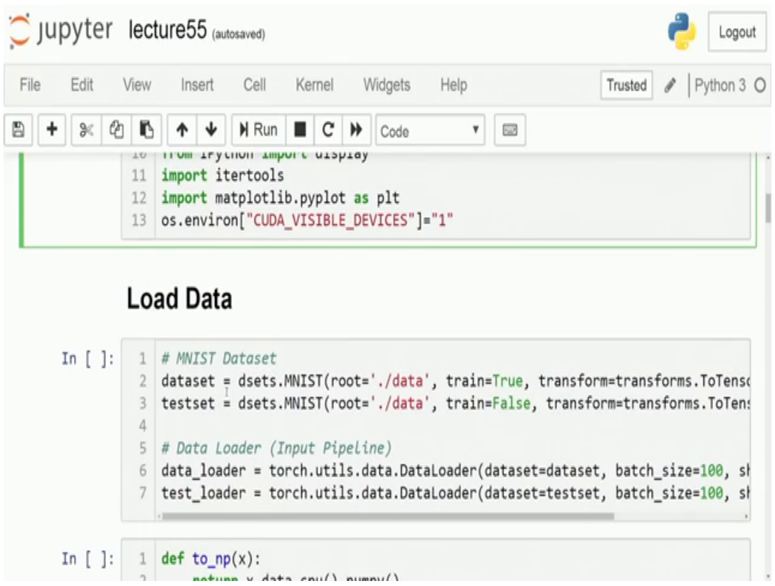
Welcome, so, this is on yet another interesting topic. So, what we are going to do is, use the adversarial autoencoder network, which we had learnt in the previous lecture in order to even build up a classifier. So, this is what I was telling you in the previous lecture that what we have been doing till now in that example was basically to use these networks in order to generate out synthetic samples. Now it is good to generate out synthetic examples, because now you can really augment out your data set with these kind of examples going on over there.

Now, at the question still remains that can I make a classifier network, robust to these kind of adversarial examples, which means that I can have unknown examples, unknown types of examples which comes from, because from your adversarial learning what exactly happens is that you have some discrete points around on your feature space over there which has spread out around everywhere. And then you are going to pull out from one point over here on the Gaussian, and you model this as a Gaussian network over there. And now that you are pulling out these random points over there, and pass it through the decoder network you have some sort of an image being generated. That is straightforward what you are doing down.

Now, the question which comes out over here on the adversarial autoencoder for classification is that, instead of trying to use all of those synthetic examples which I have generated, can I use this encoder block over here, in order to train a network, and is this assumption valid that the classification network which is built up on top of using this encoder block on the autoencoder is going to be robust. So, this is the straight question which we are going to answer in this particular lecture over there.

Now, the first part of it is just to look down into your standard usable routines and header files over there here there is going to make use of them.

(Refer Slide Time: 02:06)


jupyter lecture55 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

```
import itertools
import matplotlib.pyplot as plt
os.environ["CUDA_VISIBLE_DEVICES"]="1"
```

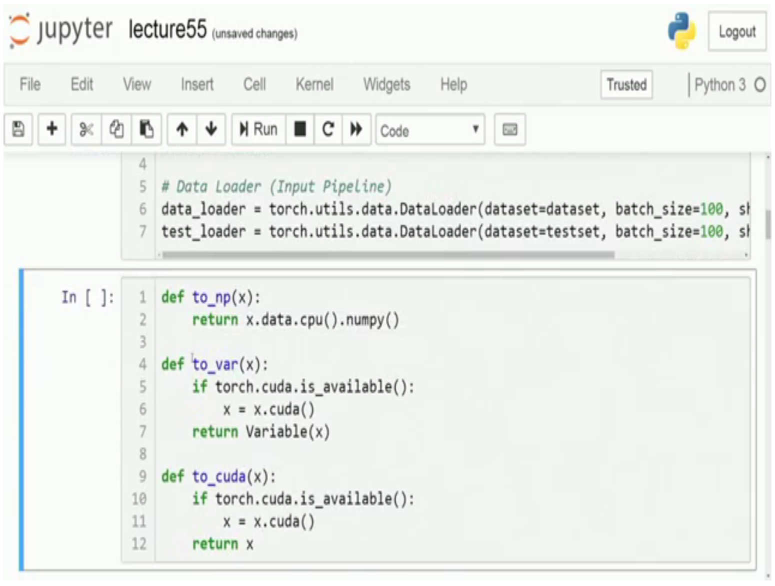
Load Data

```
# MNIST Dataset
dataset = dsets.MNIST(root='./data', train=True, transform=transforms.ToTens
testset = dsets.MNIST(root='./data', train=False, transform=transforms.ToTen

# Data Loader (Input Pipeline)
data_loader = torch.utils.data.DataLoader(dataset=dataset, batch_size=100, sh
test_loader = torch.utils.data.DataLoader(dataset=testset, batch_size=100, sh
```

```
def to_np(x):
```

Now, from there the second part of it is to load down your data set and have your data loader ready. So, this is where we do not make any change as such coming down from whatever was there in the earlier lecture so, it is it is pretty simple. So, let me just run each block over there as I keep on moving down.

(Refer Slide Time: 02:28)

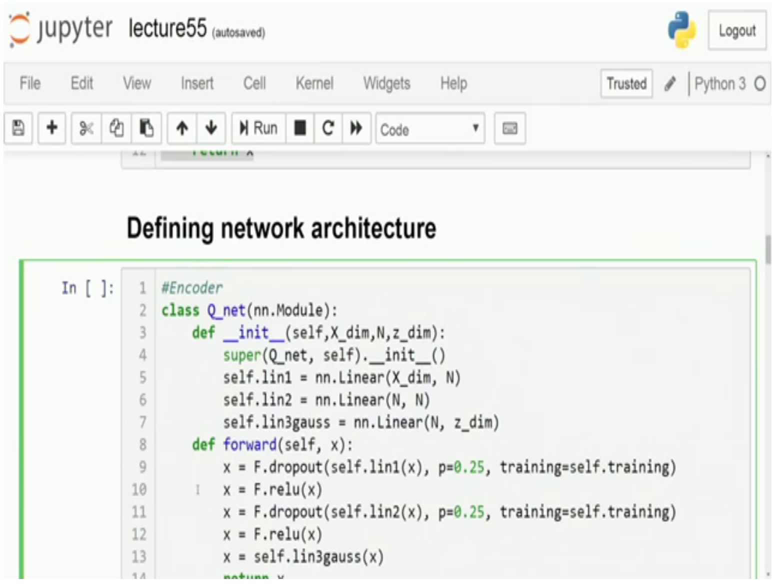

jupyter lecture55 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

```
# Data Loader (Input Pipeline)
data_loader = torch.utils.data.DataLoader(dataset=dataset, batch_size=100, sh
test_loader = torch.utils.data.DataLoader(dataset=testset, batch_size=100, sh
```

```
In [ ]:
def to_np(x):
    return x.data.cpu().numpy()

def to_var(x):
    if torch.cuda.is_available():
        x = x.cuda()
    return Variable(x)

def to_cuda(x):
    if torch.cuda.is_available():
        x = x.cuda()
    return x
```

So now on my data set let us run it so, you have your data set loaded down. And then you have your these extra variables on NumPy conversion as well as a extra functions on NumPy conversion or to conversion to an auto grad variable, and on converting it down to a cuda. Now once that is done we define our network over here.

(Refer Slide Time: 02:40)


jupyter lecture55 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

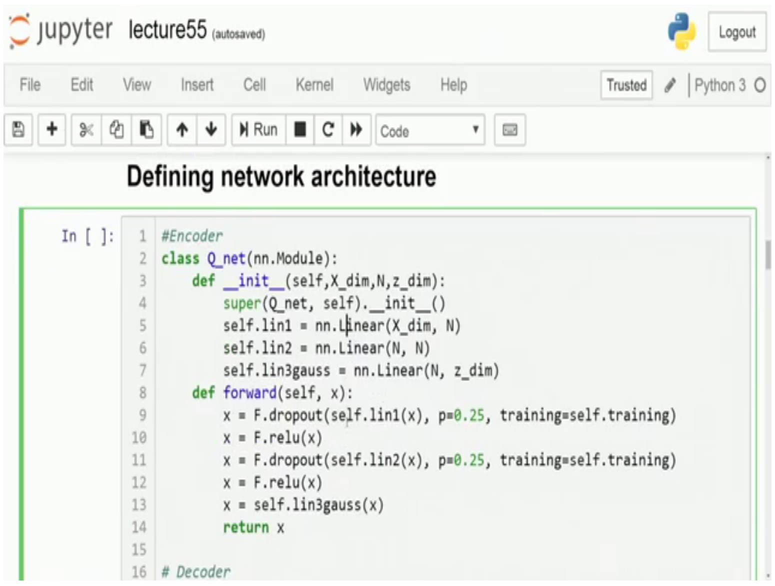
Defining network architecture

```
In [ ]:
#Encoder
class Q_net(nn.Module):
    def __init__(self,X_dim,N,z_dim):
        super(Q_net, self).__init__()
        self.lin1 = nn.Linear(X_dim, N)
        self.lin2 = nn.Linear(N, N)
        self.lin3gauss = nn.Linear(N, z_dim)
    def forward(self, x):
        x = F.dropout(self.lin1(x), p=0.25, training=self.training)
        x = F.relu(x)
        x = F.dropout(self.lin2(x), p=0.25, training=self.training)
        x = F.relu(x)
        x = self.lin3gauss(x)
```

So, the network is quite simple and straightforward. And it is at the same network which we had done in the earlier lecture, lecture number 54 where we were trying to generate out synthetic examples.

Now, so, we write it down in a very generic format such that you can take a any number of inputs on your any number of input neurons on your side over there. And then pass it through this network in order to get down your latent variable or the hidden variable space over there. And then the hidden variable space is something which you can change based on how you would like to do it. And the total number of a, hidden layers over there, and the number of neurons in the hidden layer is what you are going to keep it out straightforward.

(Refer Slide Time: 03:22)


jupyter lecture55 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

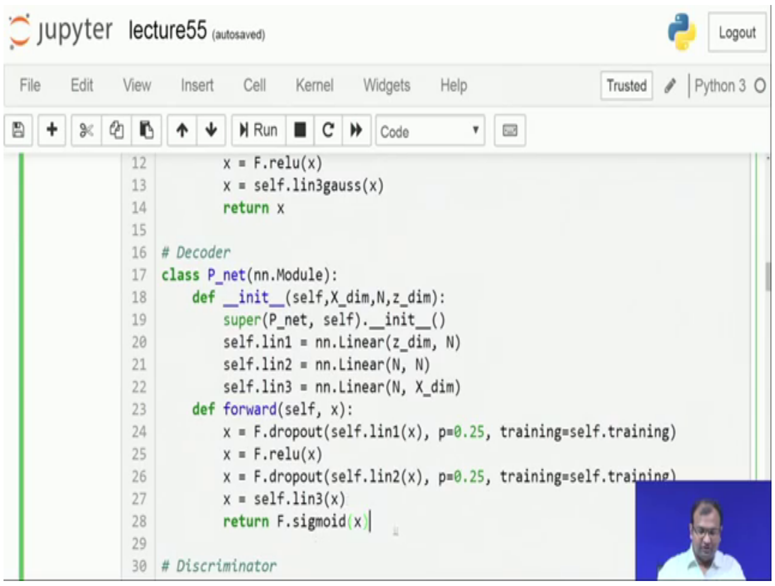
Defining network architecture

In [ ]:

```
#Encoder
class Q_net(nn.Module):
    def __init__(self,X_dim,N,z_dim):
        super(Q_net, self).__init__()
        self.lin1 = nn.Linear(X_dim, N)
        self.lin2 = nn.Linear(N, N)
        self.lin3gauss = nn.Linear(N, z_dim)
    def forward(self, x):
        x = F.dropout(self.lin1(x), p=0.25, training=self.training)
        x = F.relu(x)
        x = F.dropout(self.lin2(x), p=0.25, training=self.training)
        x = F.relu(x)
        x = self.lin3gauss(x)
        return x

# Decoder
```

So, that is how this model is going to work out. And then whether to choose and have dropout or not is up to you. And we are sticking down to the same, exactly the same network we are not making any change over there, ok.

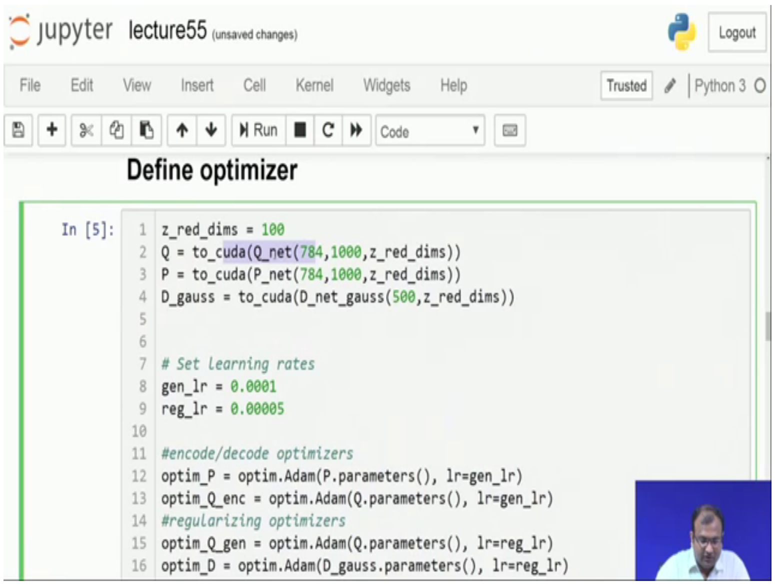
(Refer Slide Time: 03:34)


jupyter lecture55 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

```
            x = F.relu(x)
            x = self.lin3gauss(x)
            return x

# Decoder
class P_net(nn.Module):
    def __init__(self,X_dim,N,z_dim):
        super(P_net, self).__init__()
        self.lin1 = nn.Linear(z_dim, N)
        self.lin2 = nn.Linear(N, N)
        self.lin3 = nn.Linear(N, X_dim)
    def forward(self, x):
        x = F.dropout(self.lin1(x), p=0.25, training=self.training)
        x = F.relu(x)
        x = F.dropout(self.lin2(x), p=0.25, training=self.training)
        x = self.lin3(x)
        return F.sigmoid(x)

# Discriminator
```

So now you have your encoder defined, you have your decoder which is also defined in the similar way, and then you have your discriminator for your real versus fake discrimination over there. So, this is also defined in the same way, as a is, it was done in the previous lecture, and we are not going to make any changes on that one.

(Refer Slide Time: 03:51)


jupyter lecture55 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

Define optimizer

```
In [5]:
z_red_dims = 100
Q = to_cuda(Q_net(784,1000,z_red_dims))
P = to_cuda(P_net(784,1000,z_red_dims))
D_gauss = to_cuda(D_net_gauss(500,z_red_dims))


# Set learning rates
gen_lr = 0.0001
reg_lr = 0.00005

#encode/decode optimizers
optim_P = optim.Adam(P.parameters(), lr=gen_lr)
optim_Q_enc = optim.Adam(Q.parameters(), lr=gen_lr)
#regularizing optimizers
optim_Q_gen = optim.Adam(Q.parameters(), lr=reg_lr)
optim_D = optim.Adam(D_gauss.parameters(), lr=reg_lr)
```

Now, in order to define your optimizer; so, let us see yes that has run and I am ready. So, in order to define your optimizer this is where I make a few changes coming down. So, one is this important part over there. So, your z dimensions, in the earlier case was just limited down to 2 variables over there. So, you had a 2-d random number being generated out of this latent variable space. Now this is what we have changed it down and made it down to 100, and one of the reasons for changing it from 2 to 100 was if you closely recall from our experiments in using auto encoders in order to classify these digits in m nest, then we did realize that if you have lesser number of neurons in a given layer, and there is a significant bottlenecking which happens over there. Your reconstruction quality is affected, but on the other side of it you are going to squeeze out and compress all of these features so much that trying to classify out with these features also becomes really problematic.

So, essentially what we have to do is we need to retain say an over complete representation of these features which comes out over there. And for that particular reason instead of taking this latent variable space as just a 2 d tenser over there, I am going to take this as a 100-d tensor, so, this is the only change. So, definitely for that reason in the earlier case where you could vary down from minus 1 to plus 1 in unique in uniform intervals and you get down some 100 random values, and that helps you in order to synthesize images over there.

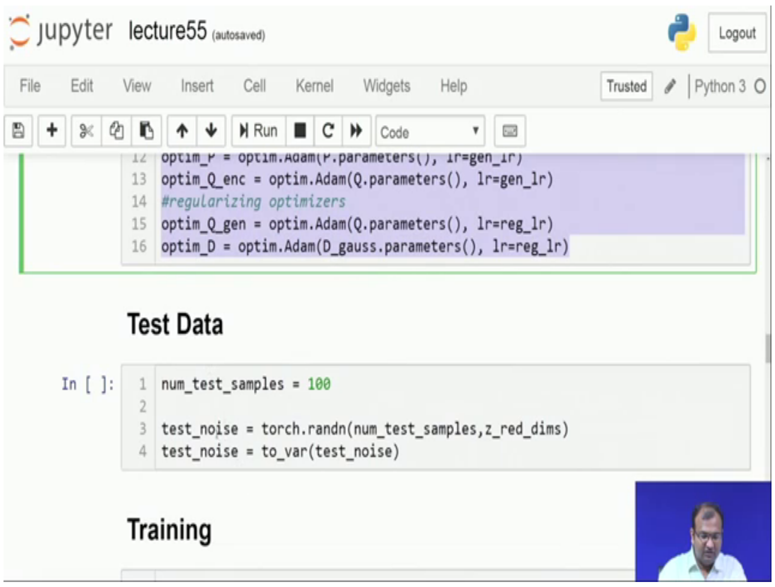
So, here it is it is no more possible because it is 100 dimension, now trying to visualize anything beyond 3 dimensions is definitely problematic. I mean, you can add color and maybe visualize a 4th dimension, but it said seriously problematic to go up to say 100 dimensions over there in anyway. So, that is where we do not make use of in that way, but I will I will exactly show you a bit later down as to the advantage which you get down when you do it with 100 dimensions, and how can you still pull out some random values on this 100 dimensional vector in order to show your synthetic examples synthetic samples, which are generated ok.

So, let us run this part while I go down through the rest of it. Now for my network, which is my encoder over there, it is a plain straightforward and similar to the earlier one you have 1000 neurons in your hidden layer over there. The only change which comes down is that you have 100 neurons in your bottleneck layer present over there. So, similarly comes down your decoder which has a similar kind of a configuration and your discriminator over there which also has a similar kind of a configuration so, we are not changing anything.

The only thing which changes in the discriminator is that my number of neurons in the hidden layer is no more as a 1000, but this number of neurons over the range is 500, 500 present over there. So, and this is a like what we had in the earlier lecture as well, ok. Now for the generator and the so for 2 of them one is the generational learning rate another is the regularization learning rate. So, what this basically means is that in one pass of it you are going to do a forward propagation over your encoder decoder network in the next pass, you are going to do a back propagation over your discriminator and a back propagation over your encoder as well.

So, for these 2 we are setting down 2 different learning rates over there, and that is to keep it conformal to the dynamic range of the gradients which get on generated over there, ok. So, that is that is the only point over here, and then you have your 2 optimizer set down for the 2 different passes over the network.

(Refer Slide Time: 07:23)


jupyter lecture55 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

```
optim_P = optim.Adam(P.parameters(), lr=gen_lr)
optim_Q_enc = optim.Adam(Q.parameters(), lr=gen_lr)
#regularizing optimizers
optim_Q_gen = optim.Adam(Q.parameters(), lr=reg_lr)
optim_D = optim.Adam(D_gauss.parameters(), lr=reg_lr)
```

Test Data

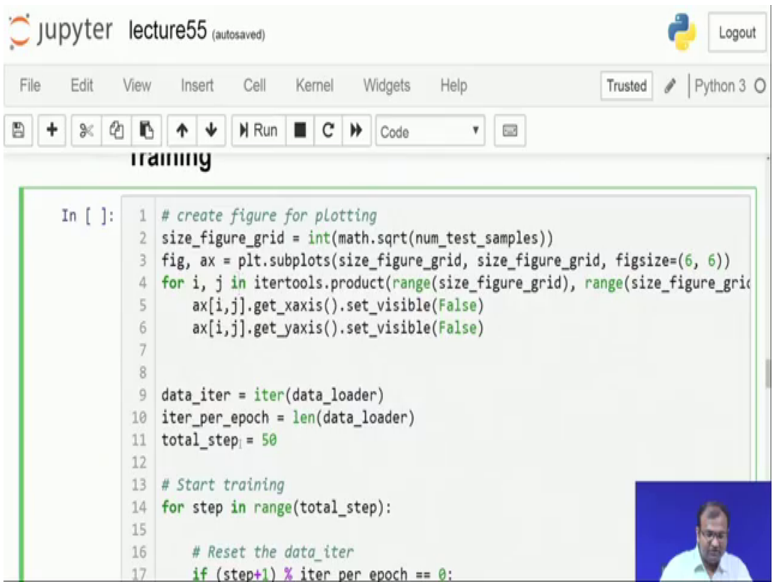
```
In [ ]: num_test_samples = 100

        test_noise = torch.randn(num_test_samples,z_red_dims)
        test_noise = to_var(test_noise)
```

Training

So, once done so, we have our test data which is being generated, now in the earlier case if you remember our test data was something which was sampled out in the range of minus 1 to plus 1 in 2 dimensions over there on a uniform sampling.

Now, here I do not have that option of doing it, because it is a 100-dimensional variable. So, what I essentially end up doing is, that I sample out some random 100 numbers on a 100-dimensional space which exists on a Gaussian distribution. So, you can have a multivariate Gaussian distribution. So, here we choose to have the total number of variables as 100 in this distribution from where I would like to pull out these numbers. So, I pull out some 100 random numbers from a 100-dimensional Gaussian distribution used over here, ok.

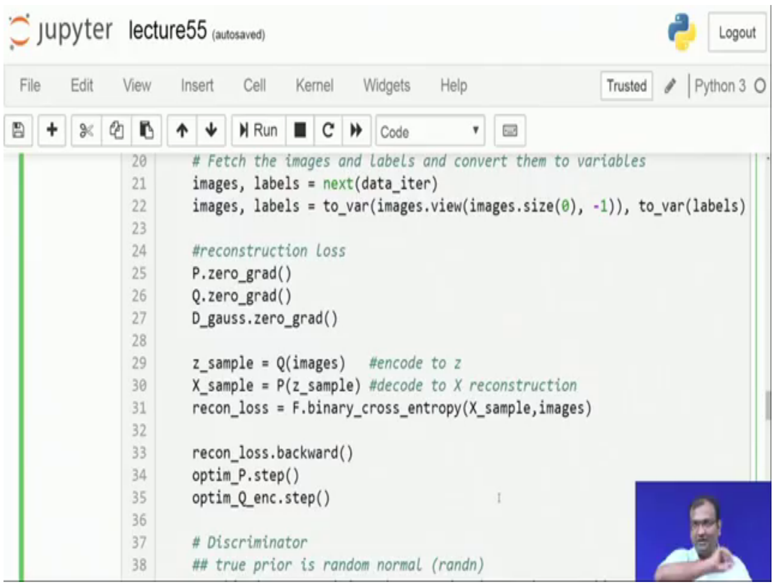
(Refer Slide Time: 08:16)


jupyter lecture55 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code
Training
In [ ]:

```
# create figure for plotting
size_figure_grid = int(math.sqrt(num_test_samples))
fig, ax = plt.subplots(size_figure_grid, size_figure_grid, figsize=(6, 6))
for i, j in itertools.product(range(size_figure_grid), range(size_figure_gri
    ax[i,j].get_xaxis().set_visible(False)
    ax[i,j].get_yaxis().set_visible(False)


data_iter = iter(data_loader)
iter_per_epoch = len(data_loader)
total_step = 50

# Start training
for step in range(total_step):

    # Reset the data_iter
    if (step+1) % iter_per_epoch == 0:
```

So, that gets in my test data created. So now, after that I start my training process over here. So, that is the straightforward training process which I have over there.

(Refer Slide Time: 18:24)


jupyter lecture55 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

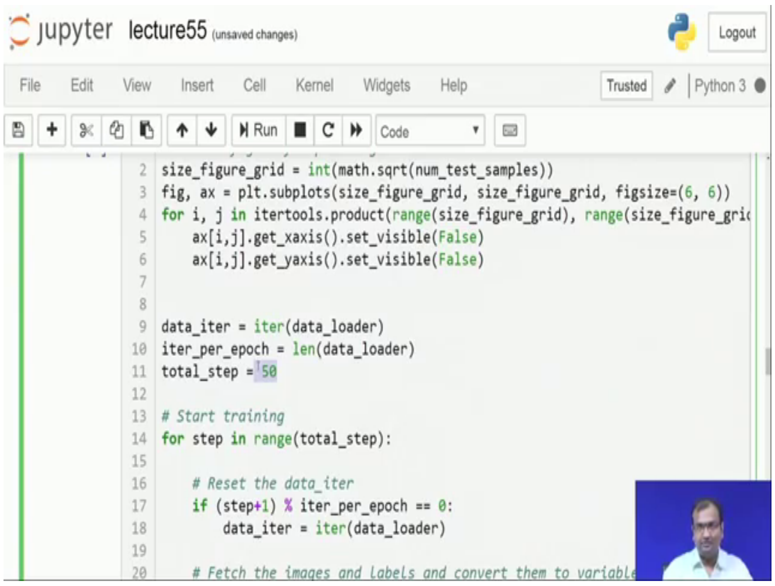
```
    # Fetch the images and labels and convert them to variables
    images, labels = next(data_iter)
    images, labels = to_var(images.view(images.size(0), -1)), to_var(labels)

    #reconstruction loss
    P.zero_grad()
    Q.zero_grad()
    D_gauss.zero_grad()

    z_sample = Q(images)   #encode to z
    X_sample = P(z_sample) #decode to X reconstruction
    recon_loss = F.binary_cross_entropy(X_sample,images)

    recon_loss.backward()
    optim_P.step()
    optim_Q_enc.step()

    # Discriminator
    ## true prior is random normal (randn)
```

And in which you do your first pass over the network for your encoder to decoder, and then the second pass is where you are going to train your discriminator, and also update your encoder part of the network. And now the loss functions which we take over here are binary cross entropy for both of them, now for a very simple reason that say your discriminator anyways is going to pull out 0 and 1, which is fake or real over there.

But then your encoder over there, whatever it produces it is in a range of 0 to 1. So, on these m nest you clearly remember that your background was all black and your digit was written down in white or say vice versa whichever way you want to take it up. So, anyways this is the one versus the other class 0 1 classification and binary cross entropy is definitely a good way of working out on this network over there. So, let us get this part of it running as well.

So, I have run till optimizer so, let us initialize out the test data, and then I get this one running while we look into how it works out as well. Now the rest part of the problem is pretty straightforward. So, there is no major change which comes down in the routine over there. Except for the fact that here we are not going to train it down over a lot number of Epochs, we just going to train it down over 50 number of Epochs um..

(Refer Slide Time: 09:40)

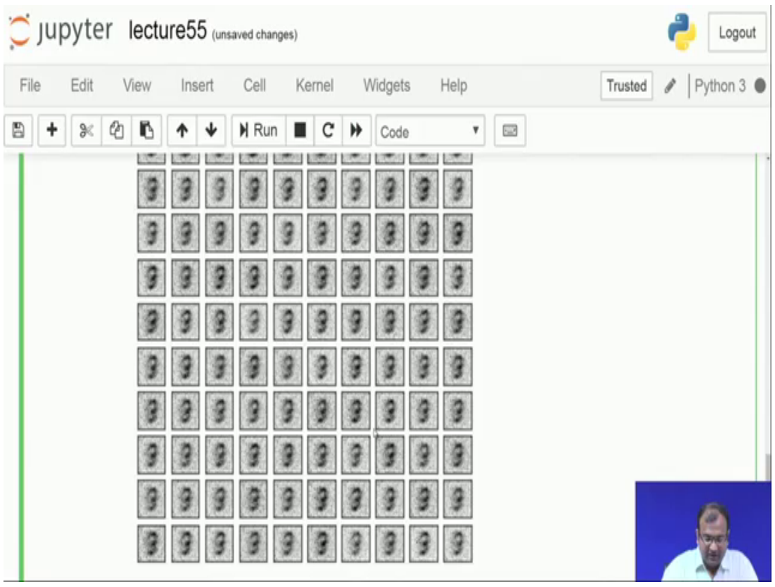

jupyter lecture55 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

```
size_figure_grid = int(math.sqrt(num_test_samples))
fig, ax = plt.subplots(size_figure_grid, size_figure_grid, figsize=(6, 6))
for i, j in itertools.product(range(size_figure_grid), range(size_figure_gri
    ax[i,j].get_xaxis().set_visible(False)
    ax[i,j].get_yaxis().set_visible(False)


data_iter = iter(data_loader)
iter_per_epoch = len(data_loader)
total_step = 50

# Start training
for step in range(total_step):

    # Reset the data_iter
    if (step+1) % iter_per_epoch == 0:
        data_iter = iter(data_loader)

    # Fetch the images and labels and convert them to variabl
```

And that is so, the change which comes over here is that since your latent variable is no more too or not much squished out over there.

So, you can actually have an over complete representation in this intermediate layer. Now that you have a over complete representation in this intermediate layer. So, there is not much of information loss which happens. And since there is lesser amount of information loss which as such can happen down over there. So, with lesser number of Epochs you will actually be starting to look out at a convergence point. So, we are just looking down at 50 Epochs over here in order to come down closer to a convergence point and um. So, as it keeps on training over here, you would be seen this these are the kind of images which keep on getting generated.

(Refer Slide Time: 10:21)


jupyter lecture55 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

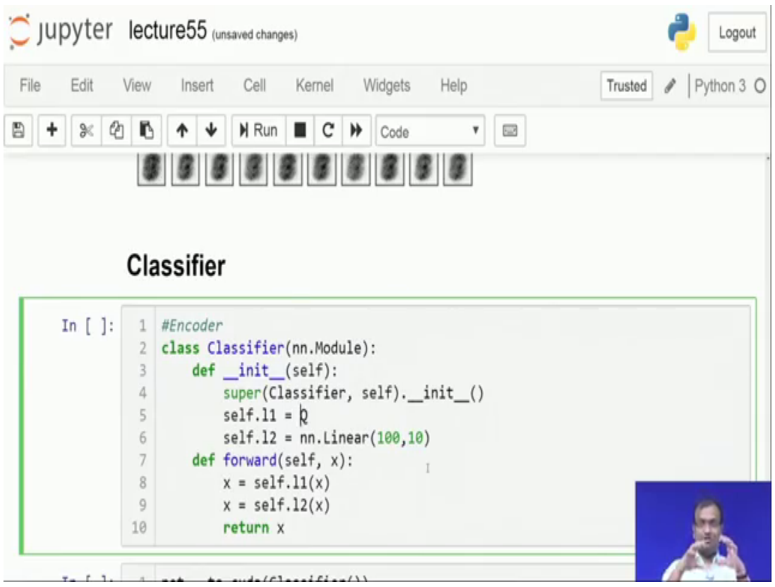
So, at that the initial part over there, you would see these things which look like 3, and then there are bit of these which are noisy. And eventually this would like some of them would start taking up the figure of an 8, some of them will still be looking down a 3 something which looks like almost a 0, but not exactly a 0 some of them start looking like 9. And these are the different varieties which it comes on creating. Now this is at the initial stage of it and a as it keeps on getting updated sometimes in certain Epochs you would see that it completely washes out and there are no distinct one.

So, in the earlier Epochs you could see out some certain figures, but here I am not able to see out any distinct figures which keep on coming on. Now that is a part of the generative model itself, because it is it is a game play which is happening. So, once your encoder decoder is trying to optimize itself, then there will be some sort of a shift in this latent variable distribution, then while you are trying to train discriminator, you are again going to enforce that these 2 come down closer to each other, and then this in this whole process that these 2 distributions start me making each other, which means that the latent variable starts me making the random value distribution from where you are drawing samples. You are going to update the encoder.

Now, that this encoder weights had changed so, the decoder also has to learn itself, and this keeps on going on and on. So, that is that is what you essentially keep on seeing over here. Now this might take some time, so now, it has run down for 50 Epochs and this is what we have over here. Now from a generative principle it does not look that clean and clear you can, but this is just for the sake of a the lectures which we are doing over here that we would try to minimize the amount of time needed.

Now typically what I would suggest is that keep on running this for about a thousand Epochs when you start looking at a very exact looking numbers created out over there, and then the generator actually is something which is converging, and your encoder decoder over there has also learnt out robust weights ok.

(Refer Slide Time: 12:24)


jupyter lecture55 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

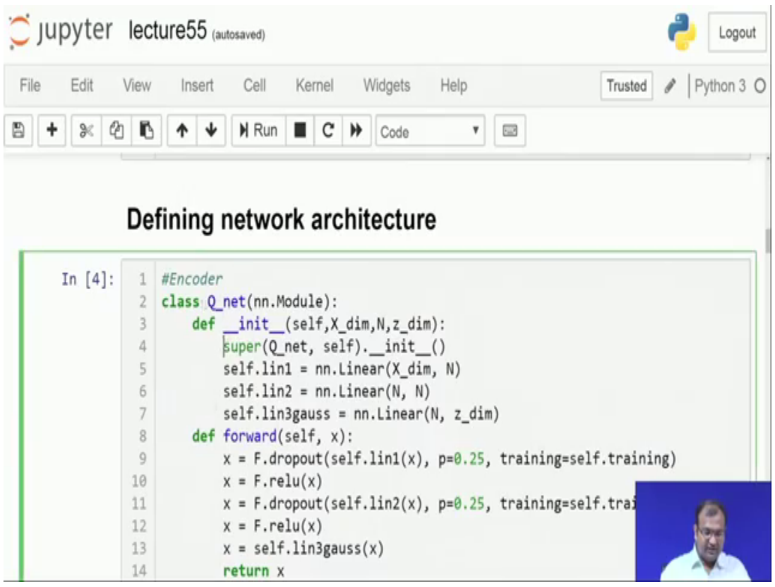
Classifier

```
In [ ]:
#Encoder
class Classifier(nn.Module):
    def __init__(self):
        super(Classifier, self).__init__()
        self.l1 = 
        self.l2 = nn.Linear(100,10)
    def forward(self, x):
        x = self.l1(x)
        x = self.l2(x)
        return x
```

So now let us get into the classifier over here in order to see what happens. So now, the whole objective is that I have my autoencoder which was trained out. So, I had an encoder decoder and I had a discriminator. But now what I am telling is that instead of trying to use this synthetic examples in order to train a classification model, can I use this encoder block alone in order to use a classification model? Now we have done our lectures on domain adaptation, we have known how to actually transfer a pre-trained model an existing model on to earn your kind of a mechanism; where you can actually modify the architecture over there as you go on.

So, that is what we are going to do over here as well. So, this is a plain simple domain adaptation thing, which we are going to solve out. So, here what we do is, we take this network Q or which is that encoder which we had defined earlier.

(Refer Slide Time: 13:22)


jupyter lecture55 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

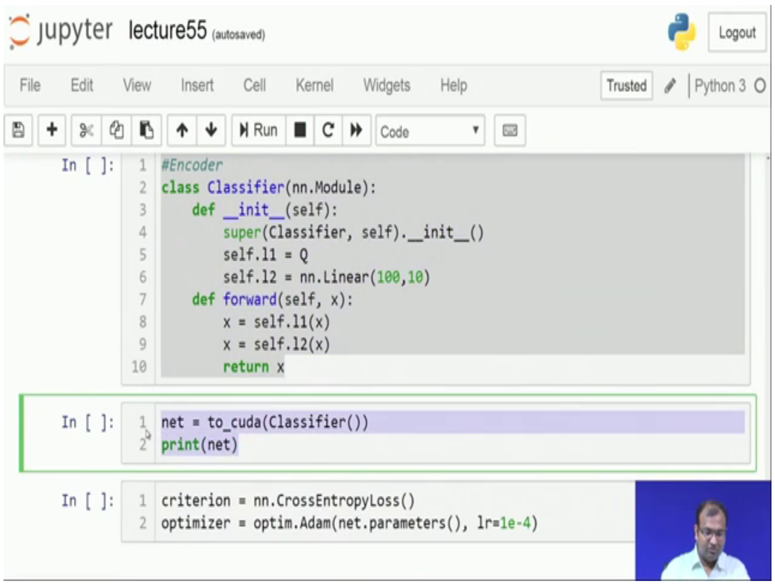
Defining network architecture

```
In [4]:
#Encoder
class Q_net(nn.Module):
    def __init__(self,X_dim,N,z_dim):
        super(Q_net, self).__init__()
        self.lin1 = nn.Linear(X_dim, N)
        self.lin2 = nn.Linear(N, N)
        self.lin3gauss = nn.Linear(N, z_dim)
    def forward(self, x):
        x = F.dropout(self.lin1(x), p=0.25, training=self.training)
        x = F.relu(x)
        x = F.dropout(self.lin2(x), p=0.25, training=self.trai
        x = F.relu(x)
        x = self.lin3gauss(x)
        return x
```

So, let us go up, so, you have this Q net over here which is my encoder block which was defined. So, what I am trying to do over here is create a new neural network in which the first part is just this Q on the encoder block which was defined. Now the output of that encoder block has a 100 neurons being thrown out, and if I am trying to solve a m nest digit classification problem then I have 10 classes it is 0 to 9 numbers over there which is just 10 classes.

So, I will create a separate part of the network which will have a just connection between 100 neurons on to 10 neurons, and this is my final classification, and on the forward pass what we are going to do is you do a forward pass over this Q to get down your latent representation from that 100, you do a forward pass over the next fully connected layer onto 10 neurons over there. So, that is my definition for the classifier which is created over here, ok.

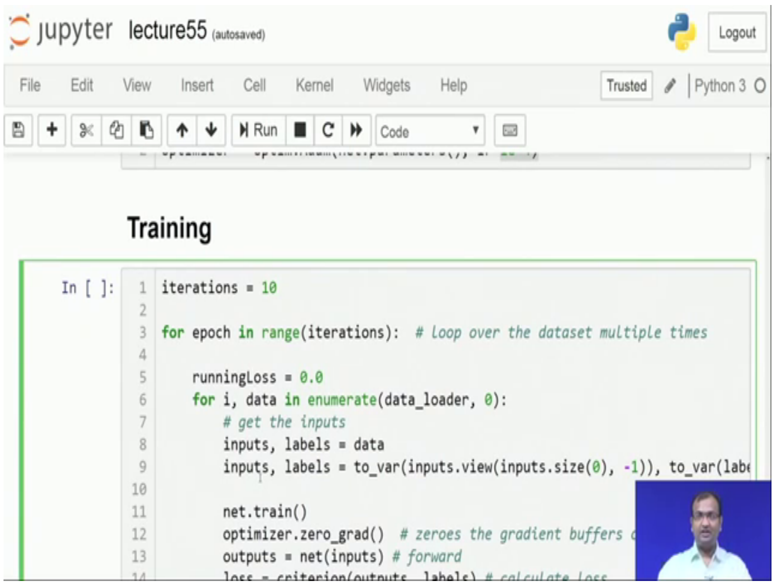
(Refer Slide Time: 14:11)



```
In [ ]:  #Encoder
         class Classifier(nn.Module):
             def __init__(self):
                 super(Classifier, self).__init__()
                 self.l1 = Q
                 self.l2 = nn.Linear(100,10)
             def forward(self, x):
                 x = self.l1(x)
                 x = self.l2(x)
                 return x

In [ ]:  net = to_cuda(Classifier())
         print(net)

In [ ]:  criterion = nn.CrossEntropyLoss()
         optimizer = optim.Adam(net.parameters(), lr=1e-4)
```

Now, if I have my cuda available I can just convert it on to my cuda system over there. So, that works out over here, and then I start defining my criterion functions. Now here instead of using binary cross entropy we are going to use the cross-entropy loss because this is a multi-hot vector which comes out over there and the optimizer which we use for this purpose is also Adam with it is own specific learning rate ok. Now starts down the training for a 10 number of Epochs. So, the reason why we need to train is one way in which you can train this one is you preserve down the encoder part over there and just update the classifier part.

(Refer Slide Time: 14:38)

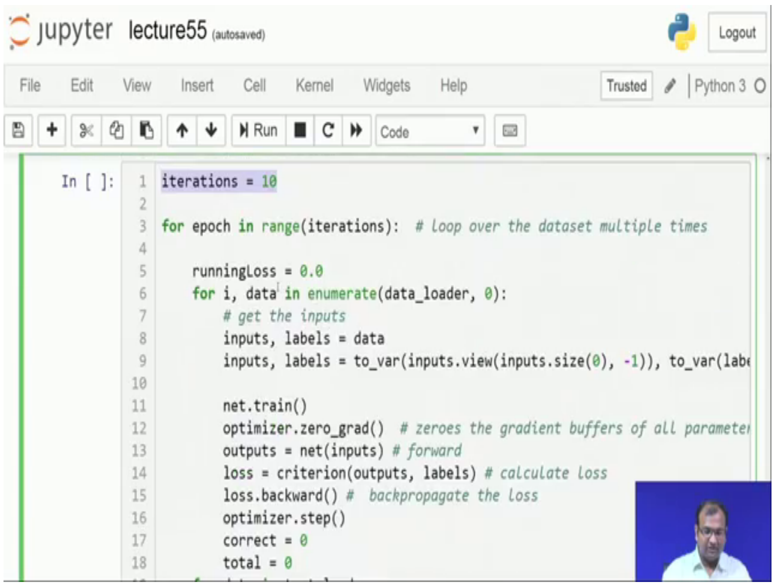


Training

```
In [ ]:  iterations = 10

         for epoch in range(iterations):  # Loop over the dataset multiple times

             runningLoss = 0.0
             for i, data in enumerate(data_loader, 0):
                 # get the inputs
                 inputs, labels = data
                 inputs, labels = to_var(inputs.view(inputs.size(0), -1)), to_var(labe

                 net.train()
                 optimizer.zero_grad()  # zeroes the gradient buffers
                 outputs = net(inputs) # forward
```

So, that is a frozen mechanism of training and doing a domain adaptation. The other mechanism of doing is where you have an end to end training. So, we are going to use the later mechanism in which you do an end to end update of all the weights within your network including your encoder and then your fully connected network over there. We train it over a 10 iterations and that is how it keeps on going.

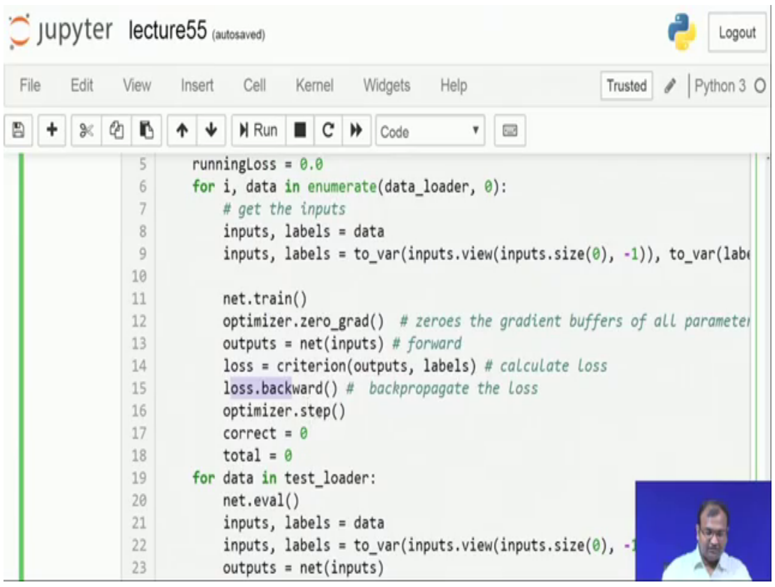
(Refer Slide Time: 15:06)


jupyter lecture55 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code
In [ ]:

```
iterations = 10

for epoch in range(iterations):  # Loop over the dataset multiple times

    runningLoss = 0.0
    for i, data in enumerate(data_loader, 0):
        # get the inputs
        inputs, labels = data
        inputs, labels = to_var(inputs.view(inputs.size(0), -1)), to_var(labe

        net.train()
        optimizer.zero_grad()  # zeroes the gradient buffers of all paramete
        outputs = net(inputs) # forward
        loss = criterion(outputs, labels) # calculate loss
        loss.backward() #  backpropagate the loss
        optimizer.step()
        correct = 0
        total = 0
```

So, the first part is where you have your data being converted on to a variables or auto grad variables over there, then you start your training routine, and within your training routine you are going to 0 down your gradients, and find out your find output over there. Find out your loss and then do a nabla of loss over there.

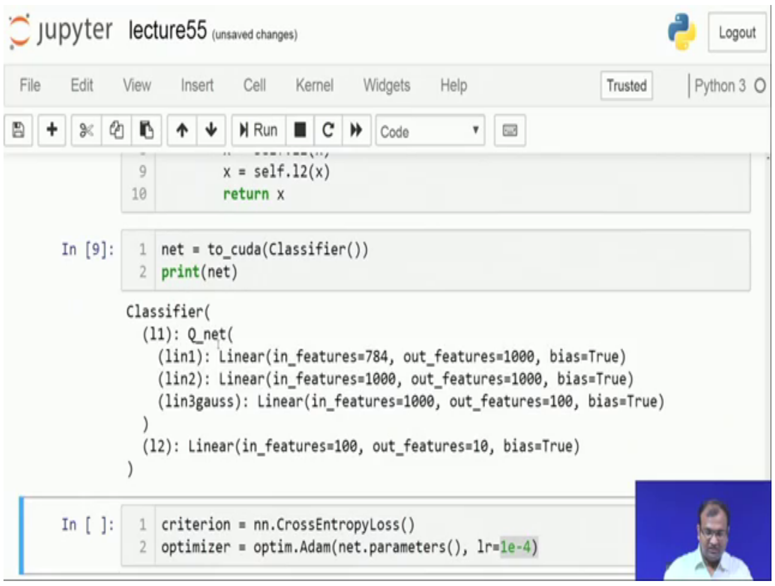
(Refer Slide Time: 15:28)



```
    runningLoss = 0.0
    for i, data in enumerate(data_loader, 0):
        # get the inputs
        inputs, labels = data
        inputs, labels = to_var(inputs.view(inputs.size(0), -1)), to_var(labe

        net.train()
        optimizer.zero_grad()  # zeroes the gradient buffers of all paramete
        outputs = net(inputs) # forward
        loss = criterion(outputs, labels) # calculate loss
        loss.backward() #  backpropagate the loss
        optimizer.step()
        correct = 0
        total = 0
    for data in test_loader:
        net.eval()
        inputs, labels = data
        inputs, labels = to_var(inputs.view(inputs.size(0), -
        outputs = net(inputs)
```

Now, once that is done now you can do your optimizer dot step or which is just a back-propagation update routine and the weight update equations which come into play. So, this is for my training part over there, now as I keep on training I also try to look into my testing mechanism over here and this is for my testing. So, let me just run these rest of the modules over here. So, I have my classification network defined, I have a type casted and this is what is printed out and you can pretty much see what it looks like.

(Refer Slide Time: 15:55)



```
        x = self.l2(x)
        return x
```

```
In [9]: net = to_cuda(Classifier())
        print(net)
```

```
Classifier(
  (l1): Q_net(
    (lin1): Linear(in_features=784, out_features=1000, bias=True)
    (lin2): Linear(in_features=1000, out_features=1000, bias=True)
    (lin3gauss): Linear(in_features=1000, out_features=100, bias=True)
  )
  (l2): Linear(in_features=100, out_features=10, bias=True)
)
```

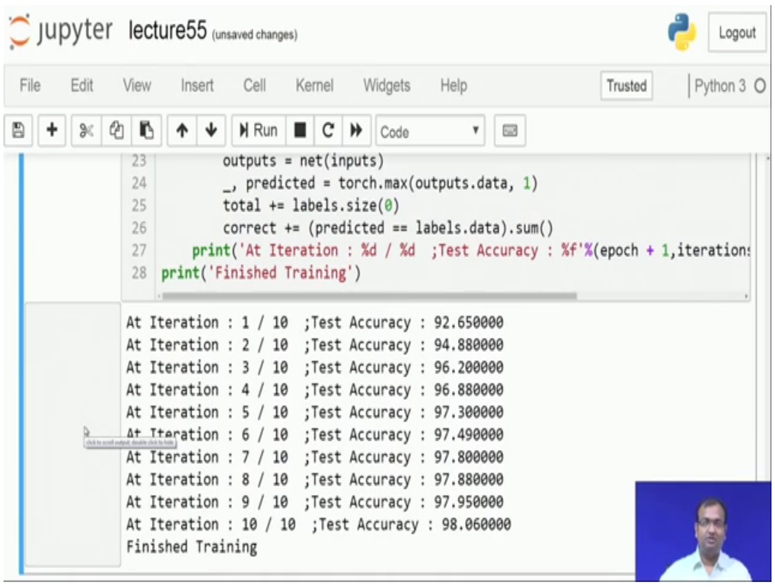
```
In [ ]: criterion = nn.CrossEntropyLoss()
        optimizer = optim.Adam(net.parameters(), lr=1e-4)
```

So, this is that first part of it which is my encoder part, and then you have this so, this part is my encoder part over there, and then I have my classification which is brought down over here. Then I run my optimizer and my criterion function, and then I can set my training module running down over there. Now this would not be taking much of a time because you straight forward are just doing it for 10 number of iterations over there, and the network which you have as such in your encoder or the decoder is still a much. Smaller network to go down except for the fact that this encoder is now trained down to be much more robust to a large number of unknown and unseen samples.

However, the challenge still is that whatever examples you have seen down over there is being generated out through this adversarial process, they are not that wide because we could not train it down for a large number of Epochs, we had just trained it down for a some 50 Epochs over there, or 50 iterations.

And now within these 50 Epochs you do not tend to technically get down a faster convergence coming down. Anyway, you keep on trading that part for a thousand iterations in the first part when you are trying to build your adversarial autoencoder, then the encoder which you build over there is a really robust one. And taking that encoder as an initial point in order to build your classified is definitely going to work out pretty good.

(Refer Slide Time: 17:24)


jupyter lecture55 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

```
        outputs = net(inputs)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels.data).sum()
    print('At Iteration : %d / %d  ;Test Accuracy : %f'%(epoch + 1,iteration
print('Finished Training')
```

```
At Iteration : 1 / 10  ;Test Accuracy : 92.650000
At Iteration : 2 / 10  ;Test Accuracy : 94.880000
At Iteration : 3 / 10  ;Test Accuracy : 96.200000
At Iteration : 4 / 10  ;Test Accuracy : 96.880000
At Iteration : 5 / 10  ;Test Accuracy : 97.300000
At Iteration : 6 / 10  ;Test Accuracy : 97.490000
At Iteration : 7 / 10  ;Test Accuracy : 97.800000
At Iteration : 8 / 10  ;Test Accuracy : 97.880000
At Iteration : 9 / 10  ;Test Accuracy : 97.950000
At Iteration : 10 / 10  ;Test Accuracy : 98.060000
Finished Training
```

So now this part of my training has run down for 10 number of iterations, and if you look at it starts with the initial accuracy of a 92.6 percent, and then somewhere around the tenth epoch it is it is already at an accuracy of 98 percent. Now if you recall back from your auto encoder strategies, where you were just using an auto encoder in order to classify it out, it was really hard for you to go down to that accuracy level of a 98 percent. So, we were with an auto encoder we were somewhere reaching a saturation of accuracy of 93 to 94 percent.

Now, the moment we have this adversarial mechanism because it starts becoming robust to all of these unknown examples which it had not seen earlier. So, this performance actually starts going out in a big way. So, this is one of these beauties of a trying to use an adversarial network in order to initialize the number of features which you learn and the gamut of features which you learn are is much more richer and wider in order to train a classification network over there. And it is not necessary that you need to have synthetic samples generated out over there and then train another network. In fact, the synthetic sample generating network itself can also be used for training a classifier in it is own way.

So, that is where we come to an end for this particular lecture on adversarial autoencoders, and trying to use these adversarial mechanisms in order to train a classifier as well. So, till then we come to an conclusion for this particular week with a generative models, and next week we are going to enter into another interesting module, which is on trying to understand and analyze videos which we have not done till now. So, that is where your data structure is quite different and the way you would be representing all of these data in terms of a tensor, and what happens to your channels match numbers and everything is this an interesting fact to play on. So, stay tuned till the next week and till then.

Goodbye.