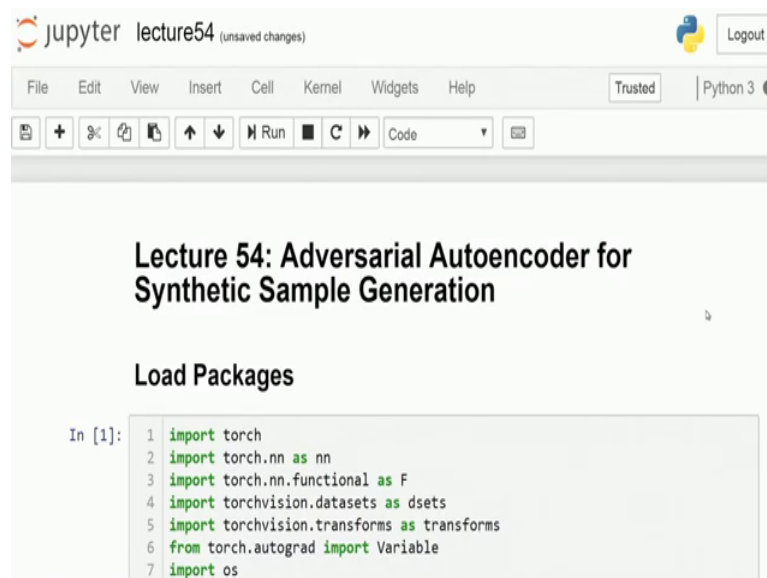


**Deep Learning for Visual Computing**  
**Prof. Debdoot Sheet**  
**Department of Electrical Engineering**  
**Indian Institute of Technology, Kharagpur**

**Lecture - 54**  
**Adversarial Autoencoder for Synthetic Sample Generation**

Welcome to today's lecture and today on this one, we are going to learn about how to start generating these samples with a generative model?

(Refer Slide Time: 00:21)

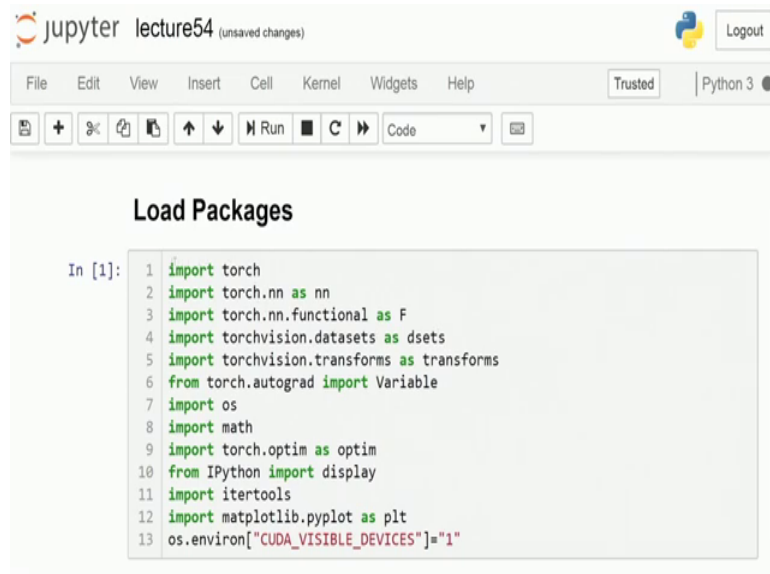


```
jupyter lecture54 (unsaved changes)
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3
+ %< ↻ ↵ ⬆ ⬇ ⬆ ⬇ Run C Code
Lecture 54: Adversarial Autoencoder for Synthetic Sample Generation
Load Packages
In [1]: 1 import torch
2 import torch.nn as nn
3 import torch.nn.functional as F
4 import torchvision.datasets as dsets
5 import torchvision.transforms as transforms
6 from torch.autograd import Variable
7 import os
```

And, for our generative model as we had studied down in the theory lectures as well; was basically an adversarial auto encoder I am going to stick down to the same kind of a philosophy over here and take an adversarial autoencoder in order to start generating synthetic samples.

Now, let us start with the starting part of it.

(Refer Slide Time: 00:41)



```
In [1]: 1 import torch
2 import torch.nn as nn
3 import torch.nn.functional as F
4 import torchvision.datasets as dsets
5 import torchvision.transforms as transforms
6 from torch.autograd import Variable
7 import os
8 import math
9 import torch.optim as optim
10 from IPython import display
11 import itertools
12 import matplotlib.pyplot as plt
13 os.environ["CUDA_VISIBLE_DEVICES"]="1"
```

So, in general we are taking down most of the header files which are standard as for any kind of network which we have been doing. There are a few other functions which you see over here a few of them are related to fetching out files from my directory some of them are related to an interactive display coming down over there and the rest of it is just about managing out which kind of cuda device or a GPU which particular GPU because, we are running it down on a remote terminal which basically has three GPUs over there and I am not going to use all the three GPUs patristic down to one specific one.

So, this is basically, a just a bookkeeping exercise for running on old side of it we are going you can pretty much comment out and in fact, we would be releasing the codes with a commented out version. So, that there is no confusion around over there. Ok.

So, this is my standard header which goes over there. Next what we are going to do is?

(Refer Slide Time: 01:34)

```
os.environ["CUDA_VISIBLE_DEVICES"]="1"
```

### Load Data

```
In [2]: 1 # MNIST Dataset
2 dataset = datasets.MNIST(root='./data', train=True, transform=transforms.ToTensor())
3 testset = datasets.MNIST(root='./data', train=False, transform=transforms.ToTensor())
4
5 # Data Loader (Input Pipeline)
6 data_loader = torch.utils.data.DataLoader(dataset=dataset, batch_size=100, shuffle=True)
7 test_loader = torch.utils.data.DataLoader(dataset=testset, batch_size=100, shuffle=True)
```

```
In [3]: 1 def to_np(x):
2         return x.data.cpu().numpy()
3
4 def to_var(x):
```

Take down to the very classical example of trying to generate digits and that is from MNIST repository. So, what my objective over here is, basically to you have some random 2 d numbers going into my generator over there and then, in order to be able to generate these kind of handwritten digits coming out of it; so 1 2 3 4 5 6.

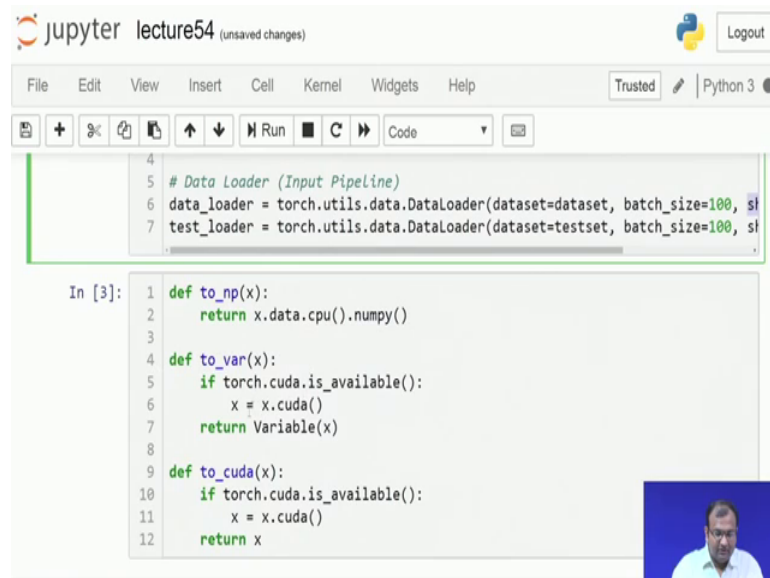
So, and then what we are doing today is, the plain and simple vanilla version of our adversarial auto encoder not a semi supervisor or a supervisor or any of them. So, this is plain simple vanilla auto encoder where I am not in even enforcing some sort of class label as to, which particular sample I am supposed to generate out over there. So, that is also independent over there. Ok.

So now, on the first part of it is that I have my two data sets over there which are my train and tests from my MNIST which are taken down and then I can actually initiate my data loader pipeline over there, to do a batch load now over here we consider a different batch size of 100 images and this is again going down conformal to, what is the total memory footprint during training and inferencing on my GPU? And, this is going to support me about batch size of 100. So, that is on the safer side what we are taking over here. Ok.

Now, we do definitely ensure that, we shuffle it up, so that; whenever you are training down over there you have the stochastic included. So, that there is a stochastic randomization and as a result of this, stochastic randomization you are getting down much better results coming from the system and then, that the system does not tend to

over fit in the earlier epochs itself. So, this was the basic rationale which we had in stochastic gradient descent while we had also looked into the advantage of trying to work it out with the stochastic gradient descent. Ok.

(Refer Slide Time: 03:26)



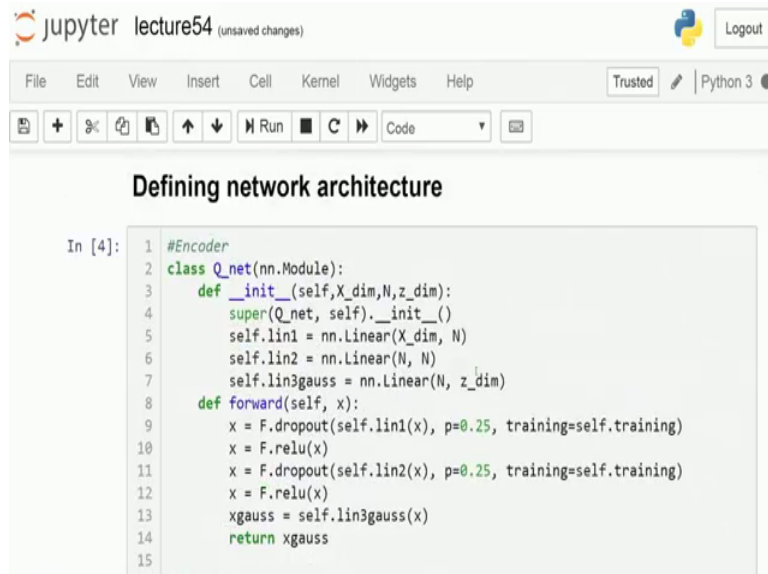
```
jupyter lecture54 (unsaved changes) Python 3
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3
4
5 # Data Loader (Input Pipeline)
6 data_loader = torch.utils.data.DataLoader(dataset=dataset, batch_size=100, si
7 test_loader = torch.utils.data.DataLoader(dataset=testset, batch_size=100, si

In [3]:
1 def to_np(x):
2     return x.data.cpu().numpy()
3
4 def to_var(x):
5     if torch.cuda.is_available():
6         x = x.cuda()
7     return Variable(x)
8
9 def to_cuda(x):
10    if torch.cuda.is_available():
11        x = x.cuda()
12    return x
```

Now, comes down so these are three extra functions, which is just for typecasting written down over here. Now, this is these functions were just written down so that, you have a clean and neatly written down lesser number of lines of codes in the subsequent trainer module over there.

Nothing beyond it, you can pretty much go and replace all of these converters over there, one is to convert down torch tensor to a numpy array, another is to define a torch tensor as an autocrat variable over there and the other one is to typecast it down to cuda, so that, you have all the data being transferred on to your GPU memory such that it executes on the GPU device itself.

(Refer Slide Time: 04:07)



```
In [4]: 1 #Encoder
2 class Q_net(nn.Module):
3     def __init__(self,X_dim,N,z_dim):
4         super(Q_net, self).__init__()
5         self.lin1 = nn.Linear(X_dim, N)
6         self.lin2 = nn.Linear(N, N)
7         self.lin3gauss = nn.Linear(N, z_dim)
8     def forward(self, x):
9         x = F.dropout(self.lin1(x), p=0.25, training=self.training)
10        x = F.relu(x)
11        x = F.dropout(self.lin2(x), p=0.25, training=self.training)
12        x = F.relu(x)
13        xgauss = self.lin3gauss(x)
14        return xgauss
15
```

Now, having said that; let us get into our actual network over here. Now, if you remember from the theory classes which we had done. So, what you essentially have is one part of it is the plain simple vanilla auto encoder. So, you will have an encoder block, you have your bottleneck variable or the latent variable over there and you have your decoder block over there.

Now, on the other side of it, you will have a discriminator, we just trying to find out real from a frame. So, technically you will be having three different neural networks present over here; one is encoder, another is decoder, another is a discriminator.

So, encoder will pass down all the variables on to your decoder block over there during training, in the second pass of the training what you have is that this encoder is going to represent all the images in terms of this latent variable and pass it over to the discriminator as well as you are going to draw a random number of samples from Gaussian distribution and pass it to the discriminator. And then, this discriminator is just going to find out, which is fake? And, which is real over here? So, that is flat plain simple which comes down.

So, here this is the module which we have written down over here. So, what we have is basically encoder module, which consists of one fully connected neural network over there, which is connecting down the total number of pixels present on my input, now here since I am using MNIST kind of data. So this, are images of 28 cross 28. So, you have 784 neurons and these are all grayscale images.

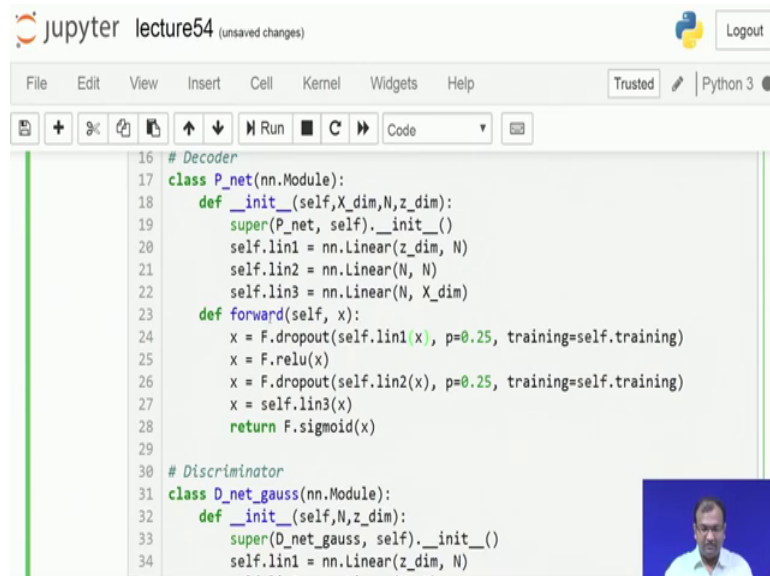
So, you do not have that color channel extra multiplier factor coming over here. So, you have 784 neurons which, come as the input of this first hidden layer and then connect down to some  $n$ , which is the total number of neurons in the in the in the first hidden layer over there.

Now, you have another fully connected network, which connects down  $n$  number of neurons to another  $n$  number of neurons and this is my second hidden layer which is created over here. Ok. Now, once this part is done then, what you have is basically, you do some sort of a Sigmoidal transformation over there and you connect down these  $n$  number of neurons on to your latent variable representation.

So, if I ah, look into this whole network over here, it gets down 784 neurons. My first hidden layer has  $n$  number of neurons, my second hidden layer has  $n$  number of neurons and then, it goes down to my third hidden layer which is also incidentally my bottleneck layer over there and that is equal to  $z$  dim, where  $z$  is what is called as this latent variable over there.

Now, if we want to do it as a 2 d latent variable over there, then i can put down  $z$  dim is equal to 2 and that is going to just map it down. Now, this layer is given down as a name of a lin 3 Gaussian because, it is it is a third linear layer over there. And, gauss for the perspective of coming down that we would expect that this latent variable over there, is generated from a Gaussian distribution. So, that that is just a naming convention and nothing beyond it.

(Refer Slide Time: 07:03)

The image shows a Jupyter Notebook window titled "lecture54 (unsaved changes)". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help), a toolbar with icons for file operations and execution, and a code editor. The code is written in Python and defines two classes: `P_net` (Decoder) and `D_net_gauss` (Discriminator). The `P_net` class has an `__init__` method that initializes three linear layers (`lin1`, `lin2`, `lin3`) and a `forward` method that applies dropout (p=0.25), ReLU, and sigmoid functions. The `D_net_gauss` class has an `__init__` method that initializes a single linear layer (`lin1`). A small video feed of a person is visible in the bottom right corner of the notebook interface.

```
16 # Decoder
17 class P_net(nn.Module):
18     def __init__(self, X_dim, N, z_dim):
19         super(P_net, self).__init__()
20         self.lin1 = nn.Linear(z_dim, N)
21         self.lin2 = nn.Linear(N, N)
22         self.lin3 = nn.Linear(N, X_dim)
23     def forward(self, x):
24         x = F.dropout(self.lin1(x), p=0.25, training=self.training)
25         x = F.relu(x)
26         x = F.dropout(self.lin2(x), p=0.25, training=self.training)
27         x = self.lin3(x)
28         return F.sigmoid(x)
29
30 # Discriminator
31 class D_net_gauss(nn.Module):
32     def __init__(self, N, z_dim):
33         super(D_net_gauss, self).__init__()
34         self.lin1 = nn.Linear(z_dim, N)
```

So, on the forward pass, let us look at what happens over that? So, I have my lin 1, which is my first linear layer over there. So, any input which comes down it comes out of this linear layer. Ok. Then, you have a dropout incorporated over here and this drop out is with the dropout fraction of 25 percent. Ok. And, once you have this thing implemented.

So, essentially what you are doing is? 784 neurons go and on a forward pass, they are going to lead down to n number of neurons coming out over here. Now, with the dropout fraction of 25 percent, I am going to basically switch off 25 percent of these neurons over there before I pass it down to the next layer. And, this fraction is constant, but which particular neurons are going to be switched off in a dropper that is going to change across every iteration and epoch.

So, for a given batch it is going to be the same number over there, but for the next forward pass which is happening, it is going to be a different dropout fraction a different node which is going to be dropped out over there. So, this robot is more of in the same lines to just prevent any kind of an over fitting coming out over there.

Now, once you have that part next, we employ a relu or a rectified linear unit as one non-linear transfer function over there. Ok. Now, on the output which comes over here, we pass it down through the second linear unit and then with the similar kind of a dropout fraction and employed as in over here.

Then, you get down the output which comes out of a, rectified linear units. So, this is where, I have my outputs done till my second hidden layer. Ok. So, I still have n number

of outputs which comes down over that and after that, I pass all of these outputs through my third hidden layer transformation to get my, latent variable or the hidden variables over there.

Now, remember one thing over there we are no more doing a rectified linear unit or a dropout over here, when trying to pass down from my second hidden layer onto my last hidden layer. And, for one of the very specific reasons is that you do not want this fractionating thing coming down everywhere and, I would want every output which comes over here, to directly go down to my output layer over there. So, this is the very straightforward reason for doing it out. Ok.

Now, like this hidden layer or say the latent variable over there, should not be a dropout fraction that; that should always be consistently coming out output that that is the reason. If I have some sort of a dropout, so what it would mean is? That, whenever there is a drop out, so essentially the activation is made down 0. Now, if I am going to have say 25 percent of my dropout fraction it means that, 25 percent or one fourth of these values are going to be exactly 0.

So, what that would employ is that, when I look into my Gaussian distribution, then I will have a peak at my 0 value. Now, if I am shifting this; if I take a non 0 centered Gaussian distribution, so I can have a 0 mean unity variance you know mean 2 variance, these kind of different Gaussian distributions where my peak is anyways coming down at my 0 location over there.

But, if I have a non 0 say Gaussian distribution, which comes from a mean of 5 and a variance of 3 but; if I have this dropout fraction, this dropout layer implemented over here, then what it would essentially do is, that it would force 25 percent of them to pick up at 0. And that, that is going to skew out the distribution. It is no more going to be a unimodal Gaussian distribution as the expert. So, that is the whole reason, why we do not have a dropout layer implemented over here.

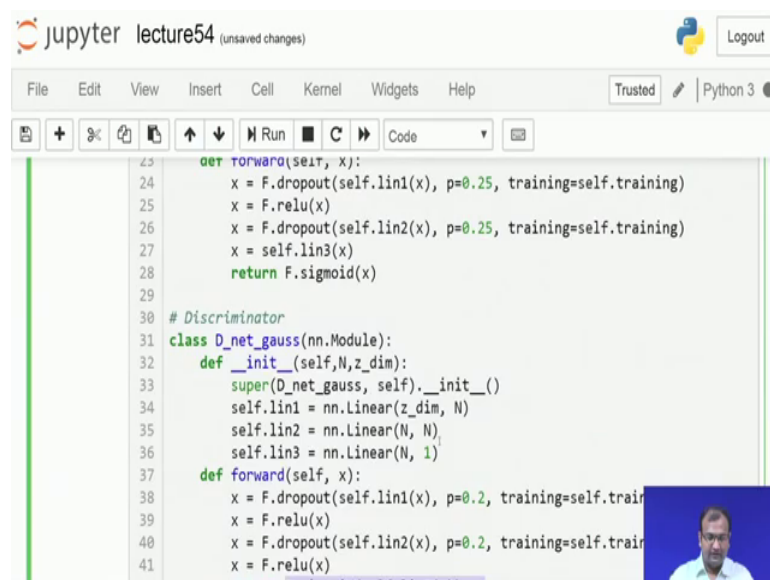
Next comes, the decoder module over there. So, the decoder looks almost like a mirror image of the encoder layer over there. So, I have my  $z$  dimensions, this latent variable which goes through a linear transformation and connects down to  $n$  number of neurons.



So, this is my first hidden layer, in my, decoder module. Then, I have my second hidden layer or my decoder module which connects on n number of neurons on to a number of neurons. And, then I have my third output neuron connection which connects this final second hidden layer's n number of neurons, to 784 neurons on my output. And, that that is to be conformal with my encoder decoder standard structure over there.

And, on my forward pass of the operation it is the same way. So, what I do is? I have my z dimensional input, so this z dim or the latent variable z, which is connected and you do a forward pass. Then, you do a drop out over there with a 25 percent fraction and you have a rectified linear unit as the non-linearity taken care of.

(Refer Slide Time: 11:30)



```
23 def forward(self, x):
24     x = F.dropout(self.lin1(x), p=0.25, training=self.training)
25     x = F.relu(x)
26     x = F.dropout(self.lin2(x), p=0.25, training=self.training)
27     x = self.lin3(x)
28     return F.sigmoid(x)
29
30 # Discriminator
31 class D_net_gauss(nn.Module):
32     def __init__(self, N, z_dim):
33         super(D_net_gauss, self).__init__()
34         self.lin1 = nn.Linear(z_dim, N)
35         self.lin2 = nn.Linear(N, N)
36         self.lin3 = nn.Linear(N, 1)
37     def forward(self, x):
38         x = F.dropout(self.lin1(x), p=0.2, training=self.training)
39         x = F.relu(x)
40         x = F.dropout(self.lin2(x), p=0.2, training=self.training)
41         x = F.relu(x)
42         return F.sigmoid(x)
```

Do you have similarly for the second neuron output over there. And then, what you do is; whatever comes out as output of the second neuron, that is passed through the third connection over there, from the second fully connected layer or the second hidden layer whatever is there you pass it down exactly without any kind of a drop out.

And, that is also in the same logic that whatever, if you have some sort of a drop out implemented over here, it means; that 25 percent of the images will randomly we made down 0 and that is going to create additional noise or additional artifact within the reconstructed image for epoch.

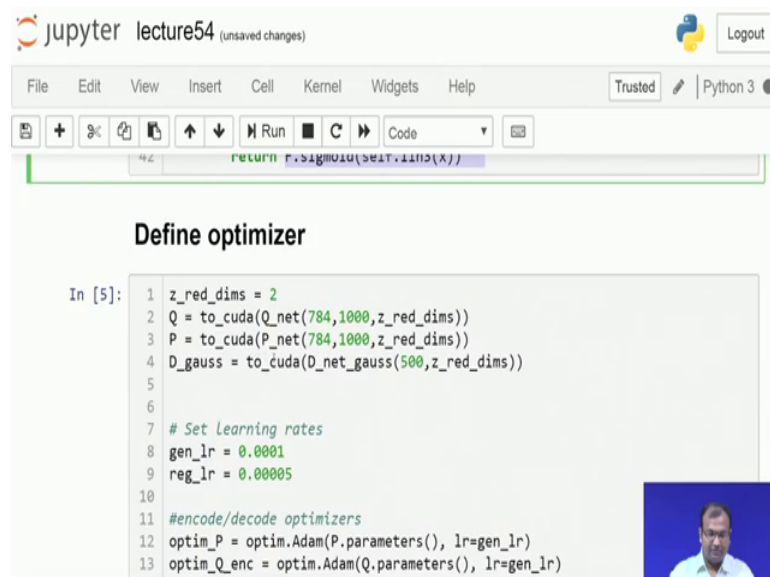
So, that is [laughter] a constant error which will always not keep keeping on and be whatever, great you are trained out the network you are never going to end up getting a 0 over there and the image will always be a 1, which will be full of artifacts. So, we just have this third layer connected down over there and then, in order to stretch out all my results in the range of 0 to 1, I have this sigmoid layer which comes over here. Ok.

So, this is my encoder decoder structure which comes over here and then you have your discriminator, which is my third neural network to come down over here. Now, in my discriminator what I essentially do is, I have my  $z$  dimensions or input taken down over there and now we choose to connect it down to  $n$  number of neurons over here, now this  $n$  is not constant factor everywhere.

So, you can actually change these  $n$  as you keep on going down over here. Now, I connect it down and then again they are connected down to  $n$  number of neurons and then finally, the point is that it is just one neuron, one single class classification, whether it is a fake sample or a real sample. So, you have one neuron in the final decision making layer going down over there.

And on the forward pass, you have a forward pass with the linear fully connected layer with a dropout fraction of 20 percent and then you do a rectified linear unit as a non-linear transformation. You again have a forward pass and a dropout fraction of 20 percent and then relu for non-linearity over there and finally, you have your  $c$  point coming out over the which forces it to go from 0 to 1 and that is on the final output over there. So, these are the three different components of the neural network, which we need to work out for our auto encoder.

(Refer Slide Time: 13:43)



The screenshot shows a Jupyter Notebook window titled "lecture54 (unsaved changes)". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help), a toolbar with icons for file operations and execution, and a code editor. The code cell is titled "Define optimizer" and contains the following Python code:

```
In [5]: 1 z_red_dims = 2
2 Q = to_cuda(Q_net(784,1000,z_red_dims))
3 P = to_cuda(P_net(784,1000,z_red_dims))
4 D_gauss = to_cuda(D_net_gauss(500,z_red_dims))
5
6
7 # Set Learning rates
8 gen_lr = 0.0001
9 reg_lr = 0.00005
10
11 #encode/decode optimizers
12 optim_P = optim.Adam(P.parameters(), lr=gen_lr)
13 optim_Q_enc = optim.Adam(Q.parameters(), lr=gen_lr)
```

So, having said that, we get into defining our optimizers over here; now, for that the first and foremost thing is you need to define, what is your z dimension over there? Now, we are going to take down this latent variable as just a 2 d variable, it makes it easy and convenient for us to do any kind of an explanation around over there. Ok.

Now, the next part is basically, you need to connect down and create these encoder decoder and the discriminator network over there. And, for each of them my n or the number of neurons in my hidden layer, I am taking them down as 1000 over there. Ok. And, then the last component over here is basically the total number of neurons in the hidden in the latent variable coming out over there, that is equal to 2.

So, this is what I am going to take it down over here. Now, that I have this part established and next what we are going to do is quite straightforward.

(Refer Slide Time: 14:41)

The screenshot shows a Jupyter Notebook window titled "lecture54 (unsaved changes)". The code in the cell is as follows:

```
10
11 #encode/decode optimizers
12 optim_P = optim.Adam(P.parameters(), lr=gen_lr)
13 optim_Q_enc = optim.Adam(Q.parameters(), lr=gen_lr)
14 #regularizing optimizers
15 optim_Q_gen = optim.Adam(Q.parameters(), lr=reg_lr)
16 optim_D = optim.Adam(D_gauss.parameters(), lr=reg_lr)
```

Below the code cell, there is a section titled "Test Data" with the following code:

```
In [*]: 1 num_test_samples = 100
2
3 test_noise = torch.Tensor(num_test_samples,z_red_dims)
4 p = [-1,1,-1,1]
5 q = [-1,1,1,-1]
6 for loop in range(4):
```

That we need to define our optimizers and the cost over there; so, I am defining my optimizer for each of them as Adam and going out straight that helps me in solving out the problem much faster. Ok.

(Refer Slide Time: 14:55)

The screenshot shows a Jupyter Notebook window titled "lecture54 (unsaved changes)". The code in the cell is as follows:

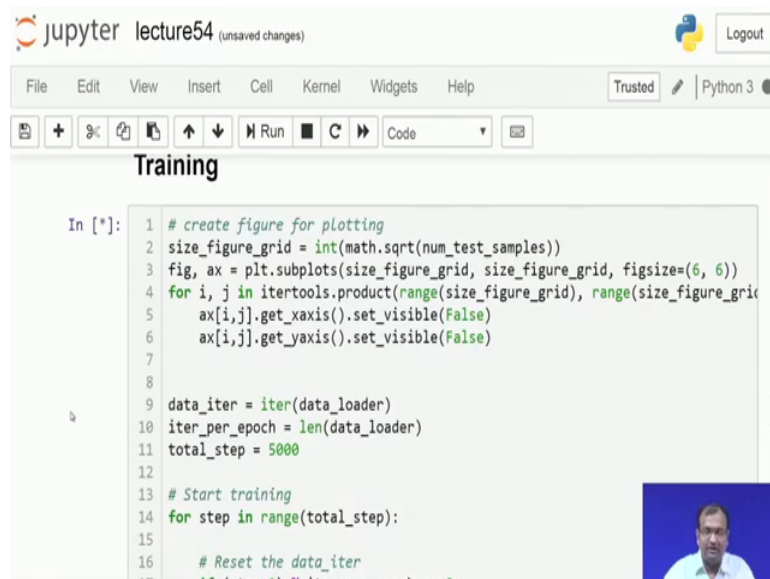
```
In [*]: 1 num_test_samples = 100
2
3 test_noise = torch.Tensor(num_test_samples,z_red_dims)
4 p = [-1,1,-1,1]
5 q = [-1,1,1,-1]
6 for loop in range(4):
7
8     for i in range(5):
9         for j in range(5):
10            test_noise[25*loop+i*5+j] = torch.Tensor([(i+1)*0.4*p[loop], (j+1)*0.4*q[loop]])
11
12
13 test_noise = to_var(test_noise)
```

Next, we need to create out some mechanism of generating out the test data. So, this is just to have this function called done in place. So, what we are essentially trying to do is? that as it proceeds over epoch. So, I am going to show you, how it is coming close towards actually generating a synthetic sample.

And, for that purpose, what we have chosen is very straightforward. So, I have my x axis, which is varying from minus 1 to plus 1 and I have my y axis, which is also varying from minus 1 to plus 1. Now, between these ranges, I am going to take down some 100 number of samples.

So, I have ten graduations between minus 1 to plus 1 and ten graduations between minus 1 to plus 1 on my y axis. So, I basically have a 10 cross 10 random number matrix being generated. So, I have 100 number of random points coming over here and that is going to generate out my output. Ok. So, that is straightforward and now, let us enter into the training part over here.

(Refer Slide Time: 15:50)

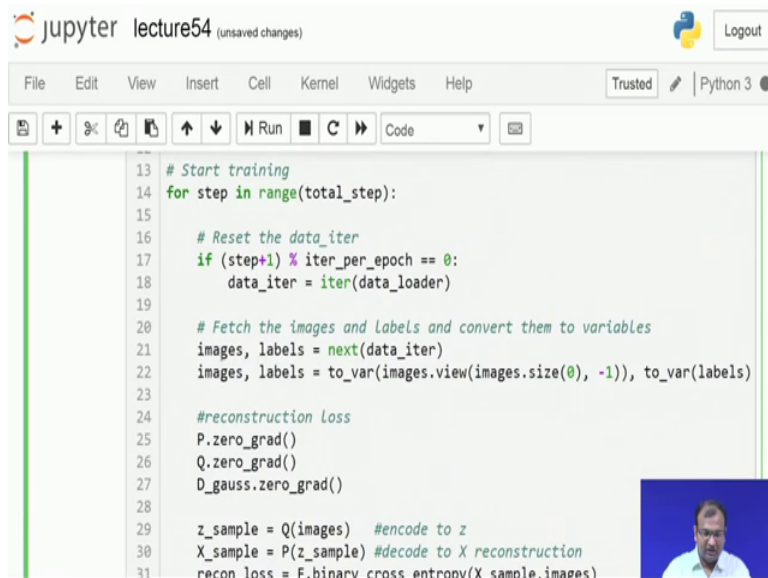


```
In [*]: 1 # create figure for plotting
2 size_figure_grid = int(math.sqrt(num_test_samples))
3 fig, ax = plt.subplots(size_figure_grid, size_figure_grid, figsize=(6, 6))
4 for i, j in itertools.product(range(size_figure_grid), range(size_figure_grid)):
5     ax[i,j].get_xaxis().set_visible(False)
6     ax[i,j].get_yaxis().set_visible(False)
7
8
9 data_iter = iter(data_loader)
10 iter_per_epoch = len(data_loader)
11 total_step = 5000
12
13 # Start training
14 for step in range(total_step):
15
16     # Reset the data_iter
17     if (step % iter_per_epoch == 0):
```

So, this is just a visualization routine which we have. So, that you can like in the earlier cases you had seen the plots for your error come down over there. So, here we just going to visualize out, how these things are coming out? So, you can still include. So, we have not get included out the functions for the plots of, how these cost functions and errors are going now? That is, I leave it up to you to do that part of it.

Now, what you can do is, over here what we are doing essentially is, we use that test data in order to show you over the epochs how its going towards generating better and better looking samples. Ok. Now, we start our training part over that now, the first part is,

(Refer Slide Time: 16:32)



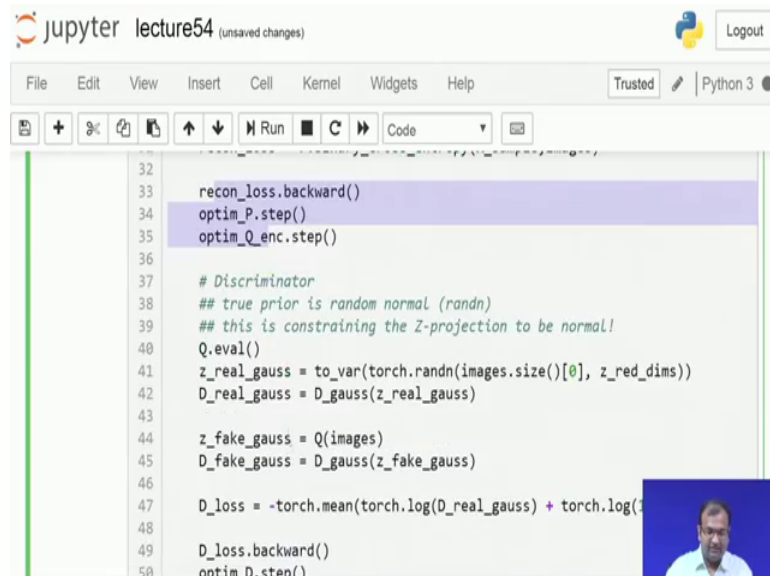
```
13 # Start training
14 for step in range(total_step):
15
16     # Reset the data_iter
17     if (step+1) % iter_per_epoch == 0:
18         data_iter = iter(data_loader)
19
20     # Fetch the images and labels and convert them to variables
21     images, labels = next(data_iter)
22     images, labels = to_var(images.view(images.size(0), -1)), to_var(labels)
23
24     #reconstruction Loss
25     P.zero_grad()
26     Q.zero_grad()
27     D_gauss.zero_grad()
28
29     z_sample = Q(images) #encode to z
30     X_sample = P(z_sample) #decode to X reconstruction
31     recon_loss = F.binary_cross_entropy(X_sample.images)
```

You have your images and labels or basically, your output is going to be the same thing and whether it is a fake or a real is the extra label which you need to push down for training your discriminator and then, you convert them to a auto grad variable over here.

Now, this two var function is; what we are using and this is something which we had defined earlier. So, in order to reduce out the overhead we had defined these three functions over there, one is to convert it to numpy, another is to convert it to auto grad variable or rather is to convert it down to cuda. Ok.

So, now that we have it over there. So, the next part is to 0 down your gradients over there and then, you can do a forward pass over each of these networks and then, you get down your losses coming.

(Refer Slide Time: 17:20)

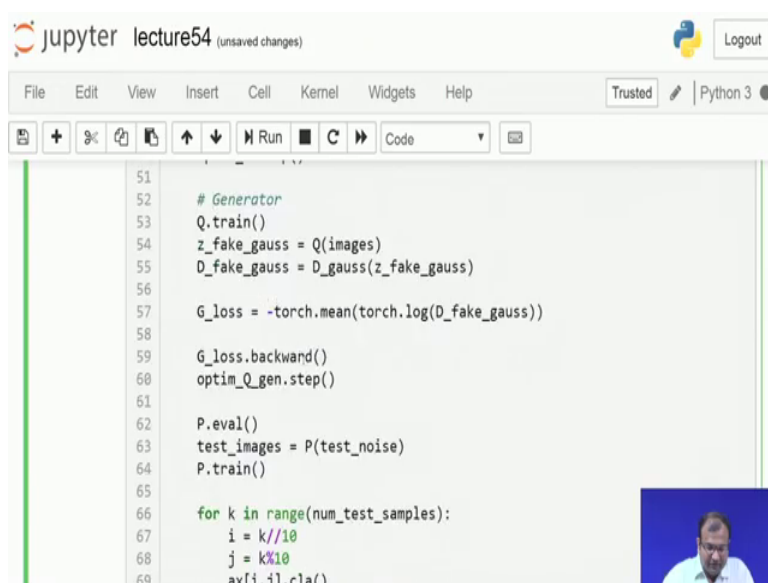


```
32 recon_loss.backward()
33
34 optim_P.step()
35 optim_Q_enc.step()
36
37 # Discriminator
38 ## true prior is random normal (randn)
39 ## this is constraining the Z-projection to be normal!
40 Q.eval()
41 z_real_gauss = to_var(torch.randn(images.size()[0], z_red_dims))
42 D_real_gauss = D_gauss(z_real_gauss)
43
44 z_fake_gauss = Q(images)
45 D_fake_gauss = D_gauss(z_fake_gauss)
46
47 D_loss = -torch.mean(torch.log(D_real_gauss) + torch.log(D_fake_gauss))
48
49 D_loss.backward()
50 optim_D.step()
```

So, there will be two parts of the losses one is your encoder decoder reconstruction loss, the second part is. So, once your network is updated. So, you remember that within every epoch there are two passes, the first passes where your encoder decoder network is getting updated. So, that you have perfect reconstruction coming out of it.

In the second part or the second pass of this data over the network, is where you do a forward pass over encoder you get down your latent variables you pull out something from the Gaussian distribution and you have this discriminator over here which will be finding out whether, it I a fake or a real over there. So, that is the second part of the network which is coming to play over here.

(Refer Slide Time: 17:58)



```
51
52 # Generator
53 Q.train()
54 z_fake_gauss = Q(images)
55 D_fake_gauss = D_gauss(z_fake_gauss)
56
57 G_loss = -torch.mean(torch.log(D_fake_gauss))
58
59 G_loss.backward()
60 optim_Q_gen.step()
61
62 P.eval()
63 test_images = P(test_noise)
64 P.train()
65
66 for k in range(num_test_samples):
67     i = k//10
68     j = k%10
69     ax[i, j].cla()
```

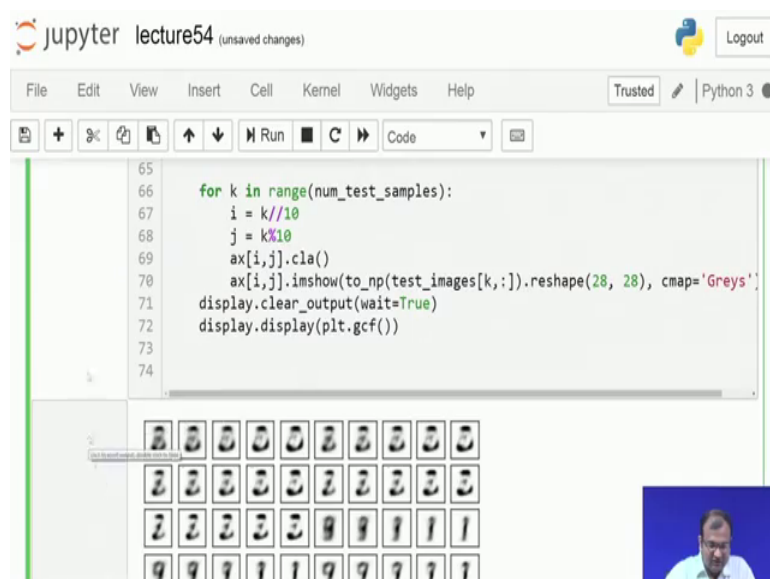
Now, once you have that; then, what we do is so, one pass of it is completely done. Now, what I said is, like you can keep on training and then, at a point of time you will get down a loss saturation coming down for your encoder decoder network. And, for your discriminator, what it will happen is that since the encoder is trying to fool the discriminator to believe that whatever it is generating is real.

So, this if you have equal number of samples of real images and equal number of those drawn from this distribution over there, then finally, this loss is going to come down to about 0.5. So, the discriminated loss technically does not go down to 0. It might start down with a lower value, but it keeps on increasing and then saturates around 0.5.

And, what we technically do is that we look for the saturation loss of saturation in the loss of this discriminator at around 0.5. And then, you would see that the encoder decoder loss which you have over here during reconstruction on the auto encoder, that is something which is also tabling out at the least possible value, might not always necessarily go down to 0, but a value which is closer to 0. And, that is where you are going to stop it out.

So, we are not stopping it over here, we keep on running this one for a larger number of epochs and then we keep on looking into it.

(Refer Slide Time: 19:10)



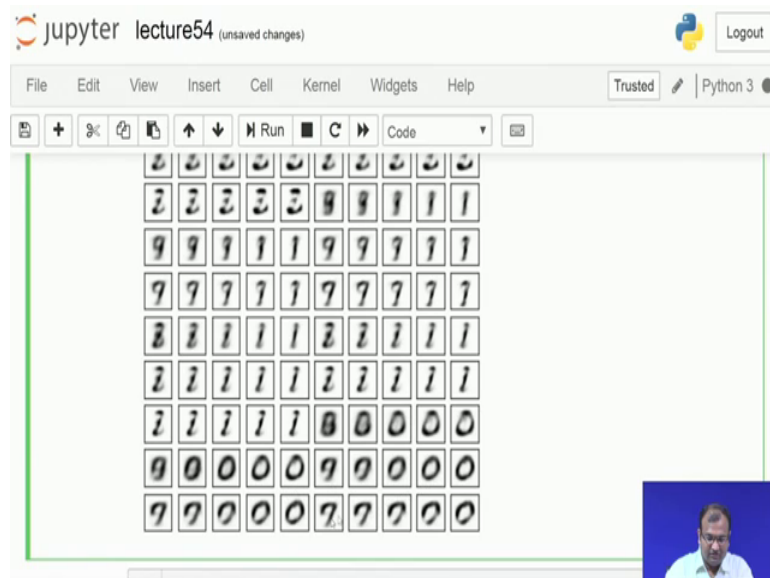
```
65
66 for k in range(num_test_samples):
67     i = k//10
68     j = k%10
69     ax[i,j].cla()
70     ax[i,j].imshow(to_np(test_images[k,:]).reshape(28, 28), cmap='Greys')
71     display.clear_output(wait=True)
72     display.display(plt.gcf())
73
74
```

The screenshot shows a Jupyter Notebook window titled "lecture54 (unsaved changes)". The code cell contains a loop that iterates over test samples, displaying a 28x28 grayscale image of handwritten digits. The digits are arranged in a 4x10 grid, showing various styles of handwritten numbers. A small video inset in the bottom right corner shows a person speaking.



So, if you look over here, this whole thing is running now and. So, over the epochs as it keeps on running and showing, you can see that these 100 points which we have drawn out in the range of minus 1 to plus 1.

(Refer Slide Time: 19:17)



That is why you would be which is seeing some sort of a change. So, these are locations on top of it, which looks something between 3 and 2. You see a lot of these nines and then, a lot of these ones coming out over here. Over here, it looks almost like an 8, there are 0's, then there are 7 and 0, these kind of things which are getting created.

Now, you keep on running this for, for a longer period of time and then, you would see this lot of these random synthetic samples which keeps on getting generated over here and, eventually they would come down to a saturation at some point of time. And, the point of saturation is basically to look down at the saturation of the discriminator loss which comes around 0.5 and also a saturation at the lowest value for the encoder decoder loss as well.

So, that is how this keeps on going, and keep in mind one thing that, this does take a bit longer to train and while I am speaking down over here, the training is actually going on still. So, we keep on running this one for real long duration, it will happen [laughter] it takes about 2 to 3 days at a stretch to actually come down to a very good version of the generated created.

But, nonetheless it does work out as it should be, and that is where is to keep you guys excited. So, that is where; we come to an end on to today's lecture on trying to find out; how to use these kind of encoder decoder structures? And, adversarial learning strategy and a discriminator over there to facilitate this adversarial learning strategy in order to generate synthetic samples over there.

Now, in the next class what we are going to do is, this encoder decoder which you are now forming over here is something, which is very much robust and strong in order to work out even for classifying unknown images and samples. So, can we make a network for classifying images based on this kind of a network and do the, do the features which are learned down by the encoder over here are they very robust, in order to work out good in this kind of an environment.

So, that is something for which we keep you tuned on to the next class, and stay tuned and glued up and if you are running it on your side, then just keep it running for a longer duration of time. So, till then see you out for the next class and.

Thanks.