Deep Learning: Homework #5

Due on January 10, 2020 at 11:55pm

Professor Emad Fatemizadeh

Amirhossein Yousefi

97206984

Practical Excercise 1

This exercise is done using *Pytorch*.
I used two *Dropout* in **Generator** of the **Gan** and also linear fully connected layer is used instead of **convolutional** .Here is the structure of **Generator** and **Discriminator** and also **binary cross entropy** is used as **loss** function and **optimizer** here is **Adam**.

```python
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.dropout1 = nn.Dropout()
        self.lin=nn.Linear(784,256)
        self.lin2=nn.Linear(256,128)
        self.lin3=nn.Linear(128,64)
        self.dropout2 = nn.Dropout()
        self.lin4=nn.Linear(64,1)

    def forward(self, img):
      img=self.dropout1(img)
      img=self.lin(img)
      img=F.leaky_relu(img,.2)
      img=self.lin2(img)
      img=F.leaky_relu(img,.2)
      img=self.lin3(img)
      img=self.dropout2(img)
      img=F.leaky_relu(img,.2)
      img=self.lin4(img)
      return F.sigmoid(img)

class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.lin=nn.Linear(128,128)
        self.lin2=nn.Linear(128,256)
        self.lin3=nn.Linear(256,512)
        self.lin4=nn.Linear(512,784)

    def forward(self, z):
      z=self.lin(z)
      z=F.leaky_relu(z,.2)
      z=self.lin2(z)
      z=F.leaky_relu(z,.2)
      z=self.lin3(z)
      z=F.leaky_relu(z,.2)
      z=self.lin4(z)
      return F.sigmoid(z)

D = Discriminator()
G = Generator()
```

Figure 1: Structure of GAN
:

Here is the *hyper parameter* of the mentioned **GAN**

```python
# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Set Hyper-parameters (change None)
BATCH_SIZE = 64
LEARNING_RATE_D = 0.0002
LEARNING_RATE_G = 0.0002
N_EPOCH = 100
```

Figure 2: Hyper parameters of GAN
:

At the end results is shown.

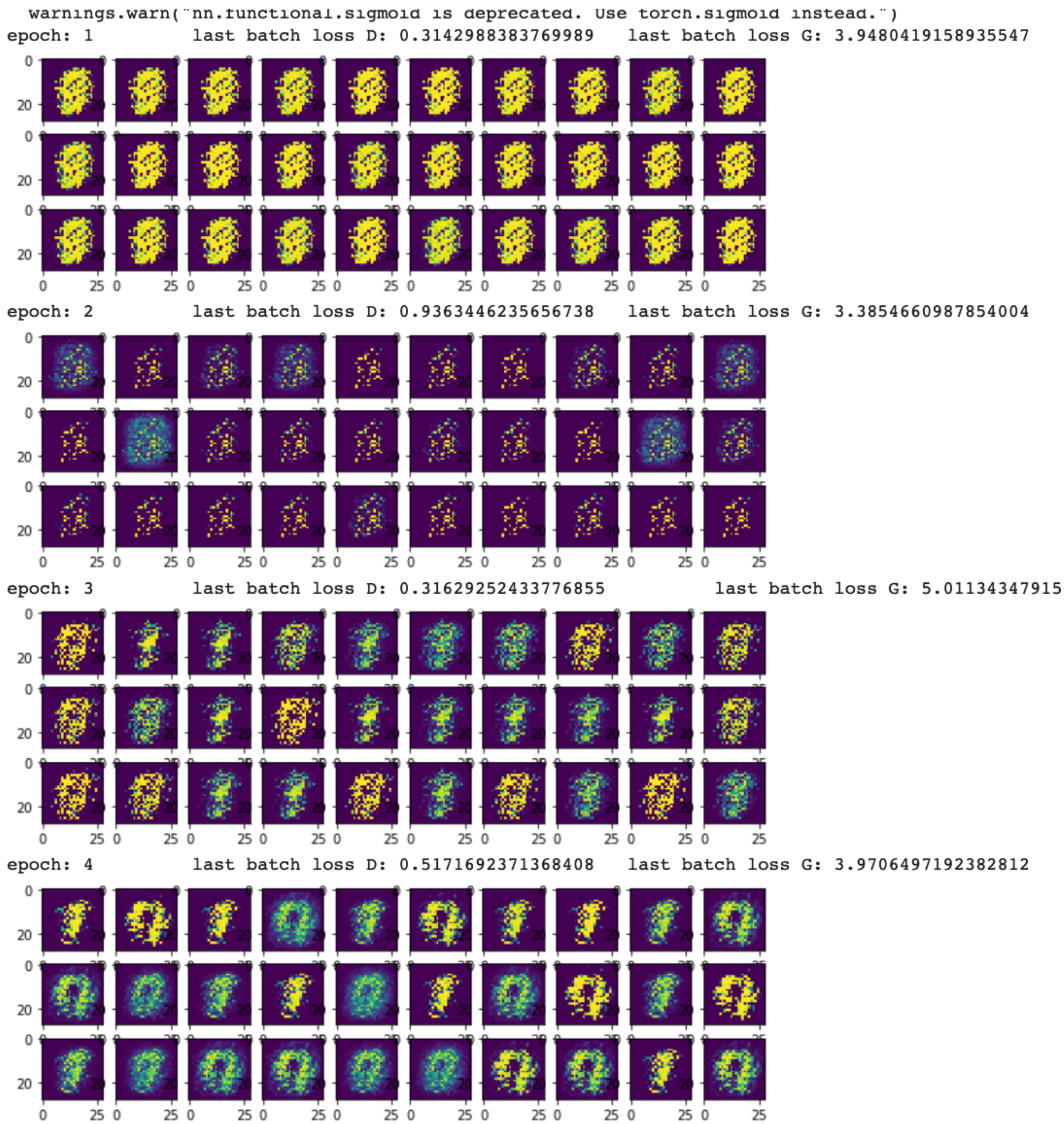

Figure 3: Some training procedure of GAN

:

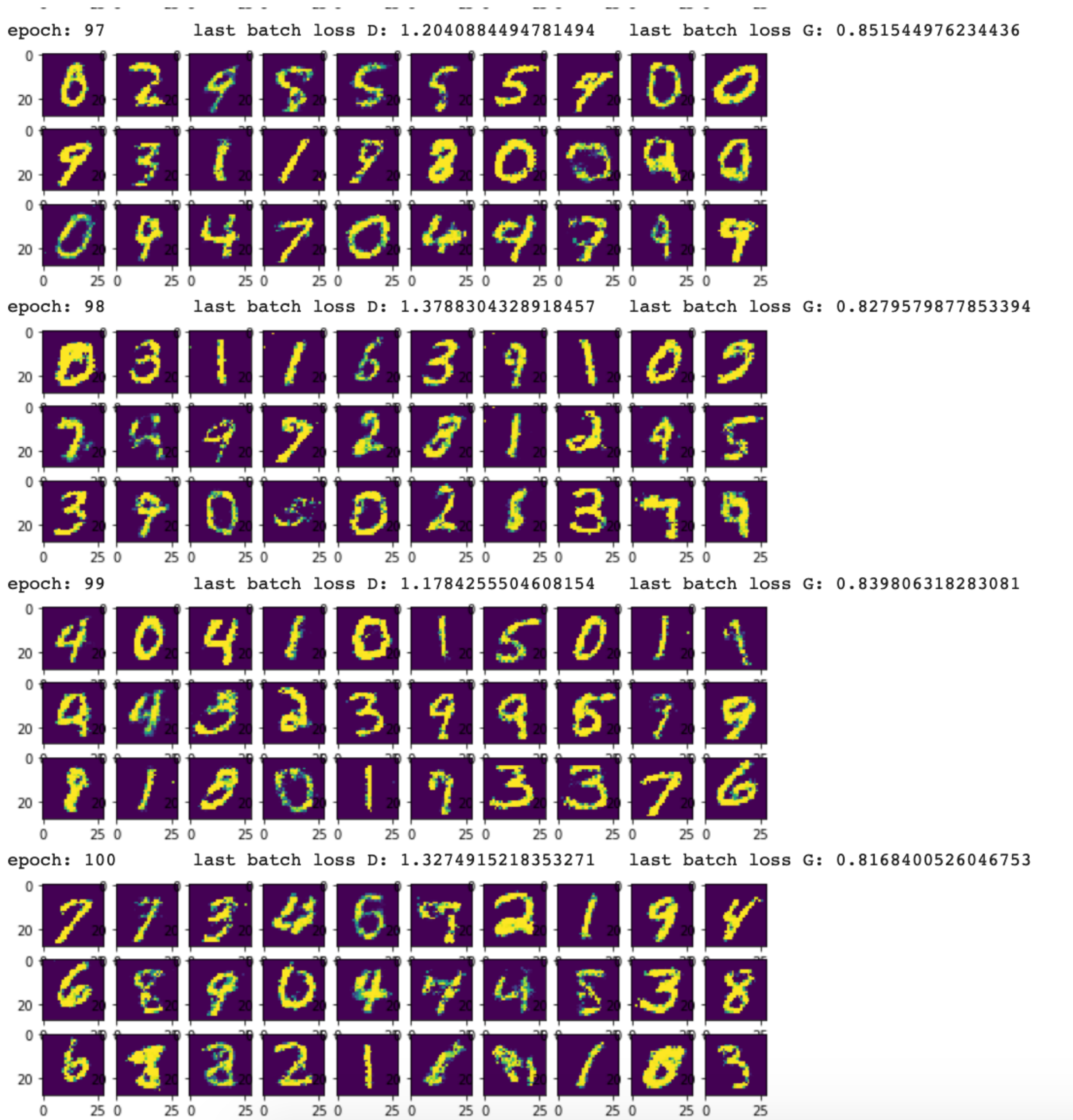

Figure 4: Some training procedure of GAN

:

Practical Excercise 2

In this part **Conditional Vaariational Auto Encoder** is trained that its structure and hyper parameters is mentioned bellow .

```python
# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Set Hyper-parameters (change None)
BATCH_SIZE = 64
LEARNING_RATE = 1e-3
N_EPOCH = 100

```

Figure 5: Hyper parameters of CVAE

:

```python
class CVAE(nn.Module):
    def __init__(self, x_dim, z_dim, c_dim):
        super(CVAE, self).__init__()

        #######################################
        ##       Define Encoder layers       ##
        ## use linear or convolutional layer ##
        #######################################
        self.linear = nn.Linear(x_dim+c_dim , 256)
        self.mu = nn.Linear(256, z_dim)
        self.var = nn.Linear(256, z_dim)
        #######################################
        ##       Define Decoder layers       ##
        ## use linear or convolutional layer ##
        #######################################
        self.latent_to_hidden = nn.Linear(z_dim+c_dim, 256)
        self.hidden_to_out = nn.Linear(256, x_dim)
    def encoder(self, x, c):
        x = torch.cat((x, c), dim=1)
        hidden = F.relu(self.linear(x))
        mean = self.mu(hidden)
        log_var = self.var(hidden)
        return  mean, log_var

    def decoder(self, z, c):
        x = torch.cat((z, c), dim=1)
        # x is of shape [batch_size, latent_dim + num_classes]
        x = F.relu(self.latent_to_hidden(x))
        # x is of shape [batch_size, hidden_dim]
        generated_x = F.sigmoid(self.hidden_to_out(x))
        # x is of shape [batch_size, output_dim]
        return  generated_x

    def sampling(self, mu, log_var):
        std = torch.exp(log_var / 2)
        eps = torch.randn_like(std)
        return eps.mul(std).add(mu)


    def forward(self, x, c):
        z_mu, z_var = self.encoder(x,c)
        x_sample=self.sampling(z_mu,z_var)
        gene_x = self.decoder(x_sample,c)
        return gene_x,z_mu,z_var
```

Figure 6: Structure of CVAE

:

Note that the **latent dim** is 75 and **optimizer** is**Adam** and **loss function** is **cross entropy**. Now turn to results:

```
  warnings.warn(warning.format(ret))
Epoch: 1/100     Average loss: 161.2641
Epoch: 2/100     Average loss: 122.8209
Epoch: 3/100     Average loss: 113.6140
Epoch: 4/100     Average loss: 109.6745
Epoch: 5/100     Average loss: 107.5117
Epoch: 6/100     Average loss: 106.0893
Epoch: 7/100     Average loss: 105.0730
Epoch: 8/100     Average loss: 104.3528
Epoch: 9/100     Average loss: 103.7868
Epoch: 10/100    Average loss: 103.3357
Epoch: 11/100    Average loss: 102.9739
Epoch: 12/100    Average loss: 102.6376
Epoch: 13/100    Average loss: 102.3341
Epoch: 14/100    Average loss: 102.0738
Epoch: 15/100    Average loss: 101.8479
Epoch: 16/100    Average loss: 101.7382
Epoch: 17/100    Average loss: 101.5214
Epoch: 18/100    Average loss: 101.3419
Epoch: 19/100    Average loss: 101.2114
Epoch: 20/100    Average loss: 101.0211
Epoch: 21/100    Average loss: 100.9132
Epoch: 22/100    Average loss: 100.7905
Epoch: 23/100    Average loss: 100.6251
Epoch: 24/100    Average loss: 100.5392
Epoch: 25/100    Average loss: 100.4281
Epoch: 26/100    Average loss: 100.3412
Epoch: 27/100    Average loss: 100.2732
Epoch: 28/100    Average loss: 100.1803
Epoch: 29/100    Average loss: 100.0894
Epoch: 30/100    Average loss: 100.0003
Epoch: 31/100    Average loss: 99.9382
Epoch: 32/100    Average loss: 99.9148
Epoch: 33/100    Average loss: 99.7634
Epoch: 34/100    Average loss: 99.7244
Epoch: 35/100    Average loss: 99.6141
Epoch: 36/100    Average loss: 99.5699
Epoch: 37/100    Average loss: 99.5074
Epoch: 38/100    Average loss: 99.4333
Epoch: 39/100    Average loss: 99.4050
Epoch: 40/100    Average loss: 99.3190
Epoch: 41/100    Average loss: 99.3207
Epoch: 42/100    Average loss: 99.1825
Epoch: 43/100    Average loss: 99.1221
Epoch: 44/100    Average loss: 99.1207
Epoch: 45/100    Average loss: 99.0267
Epoch: 46/100    Average loss: 99.0046
Epoch: 47/100    Average loss: 98.9378
Epoch: 48/100    Average loss: 98.8713
Epoch: 49/100    Average loss: 98.8227
Epoch: 50/100    Average loss: 98.8121
Epoch: 51/100    Average loss: 98.7702
Epoch: 52/100    Average loss: 98.7240
Epoch: 53/100    Average loss: 98.6870
```

Figure 7: Results for CVAE

:

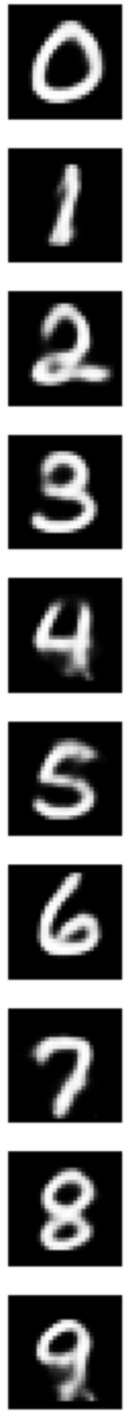

Figure 8: Reconstructed MNIST when both mentioned coefficient in loss is one
:

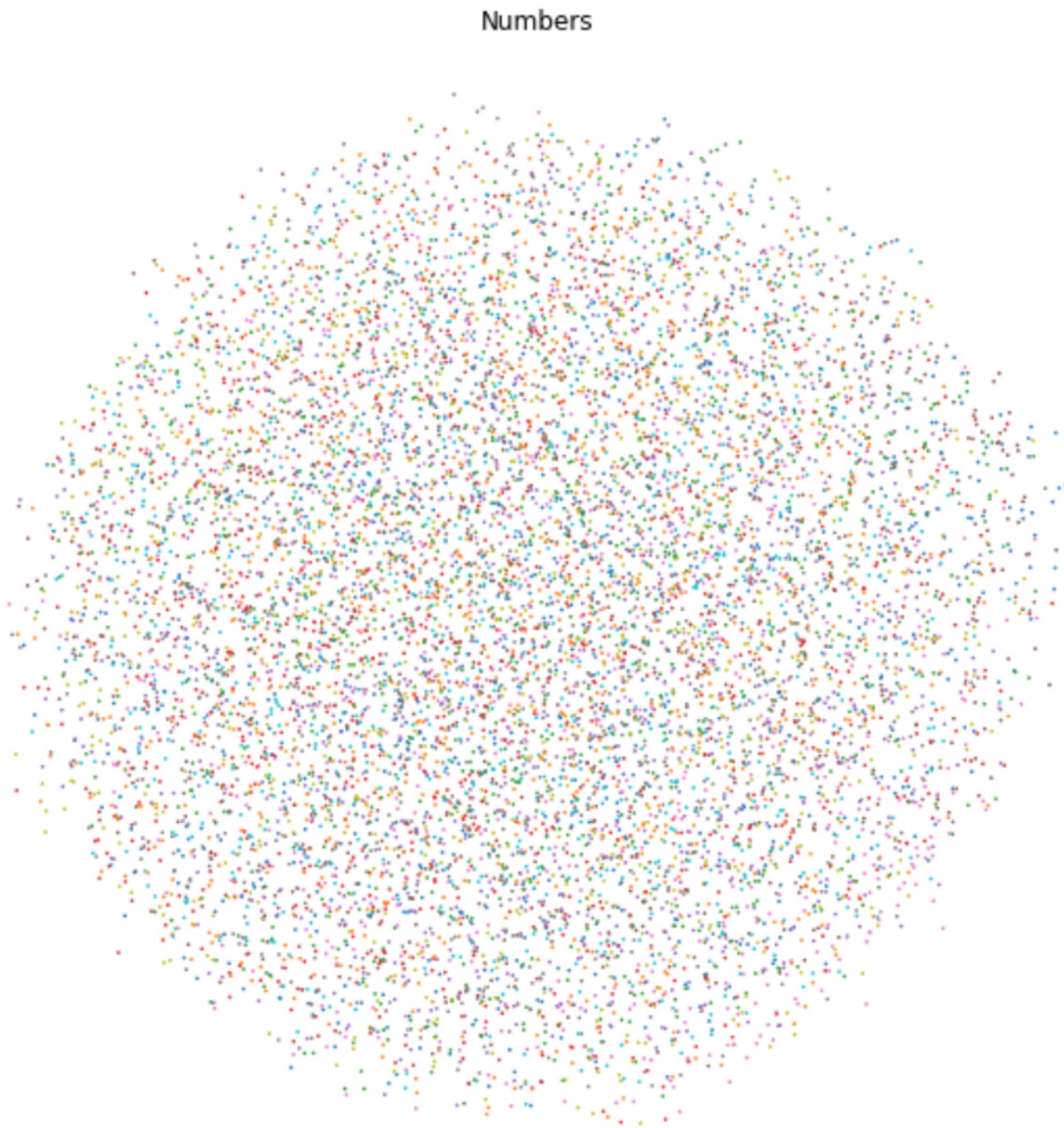

Figure 9: Umap visualization when both mentioned coefficient in loss is one

:

Now let's change the coefficients of loss function .

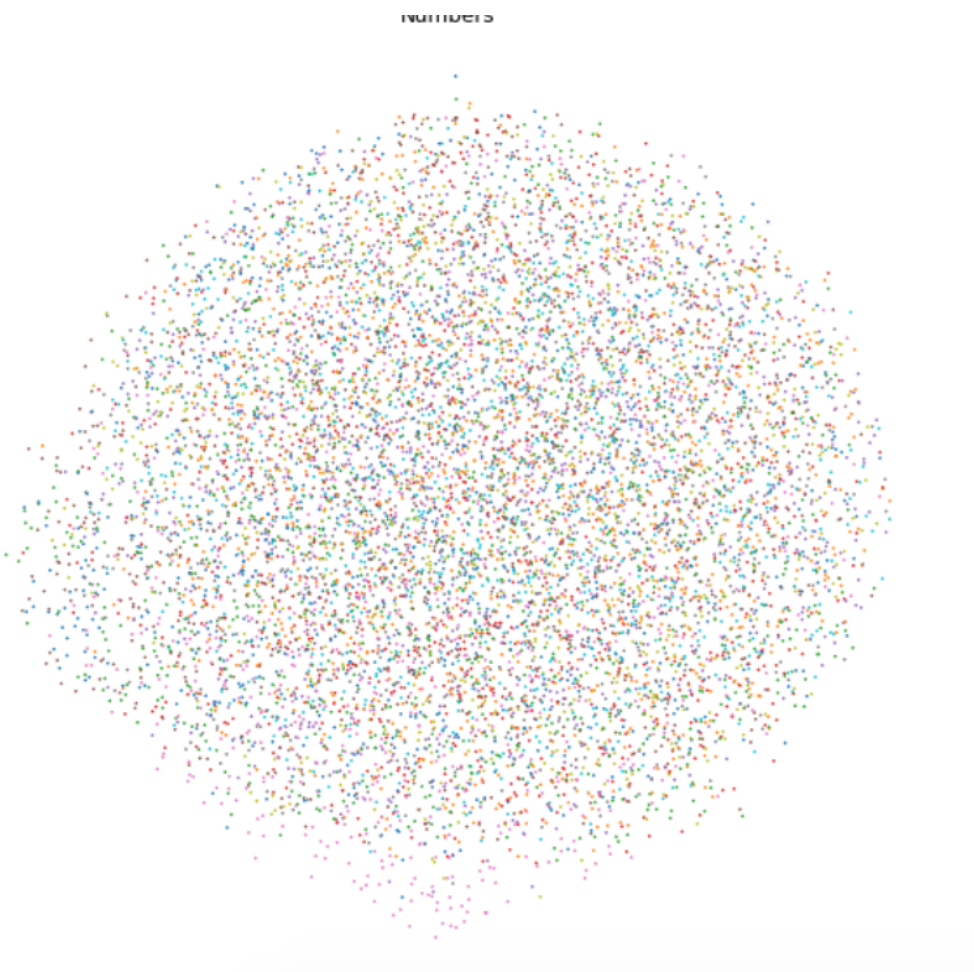

Figure 10: Umap visualization for bellow coefficient
:

```python

# return reconstruction error + KL divergence losses
def loss_function(recon_x, x, mu, log_var):
    kl_loss = -0.5 * torch.sum(1 + log_var - mu.pow(2) - log_var.exp())
    recon_loss = F.binary_cross_entropy(recon_x, x, size_average=False)
    return 1 * kl_loss + 2 * recon_loss #You can change constants
```

Figure 11: Coefficients for figure10
:

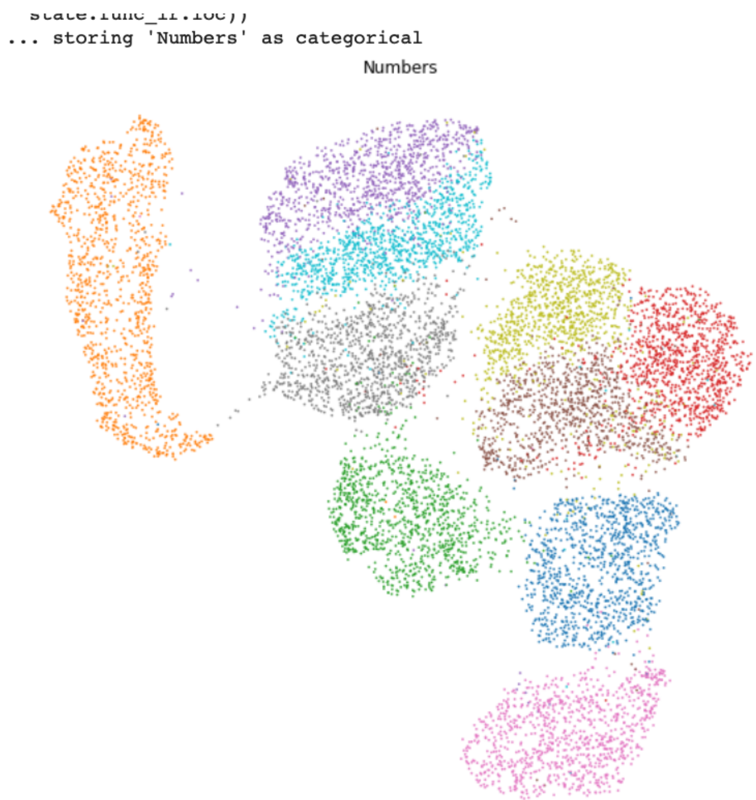

Figure 12: Umap visualization for bellow coefficient
:

```
# return reconstruction error + KL divergence losses
def loss_function(recon_x, x, mu, log_var):
    kl_loss = -0.5 * torch.sum(1 + log_var - mu.pow(2) - log_var.exp())
    recon_loss = F.binary_cross_entropy(recon_x, x, size_average=False)
    return .001 * kl_loss + 1 * recon_loss #You can change constants
```

Figure 13: Coefficients for figure12
:

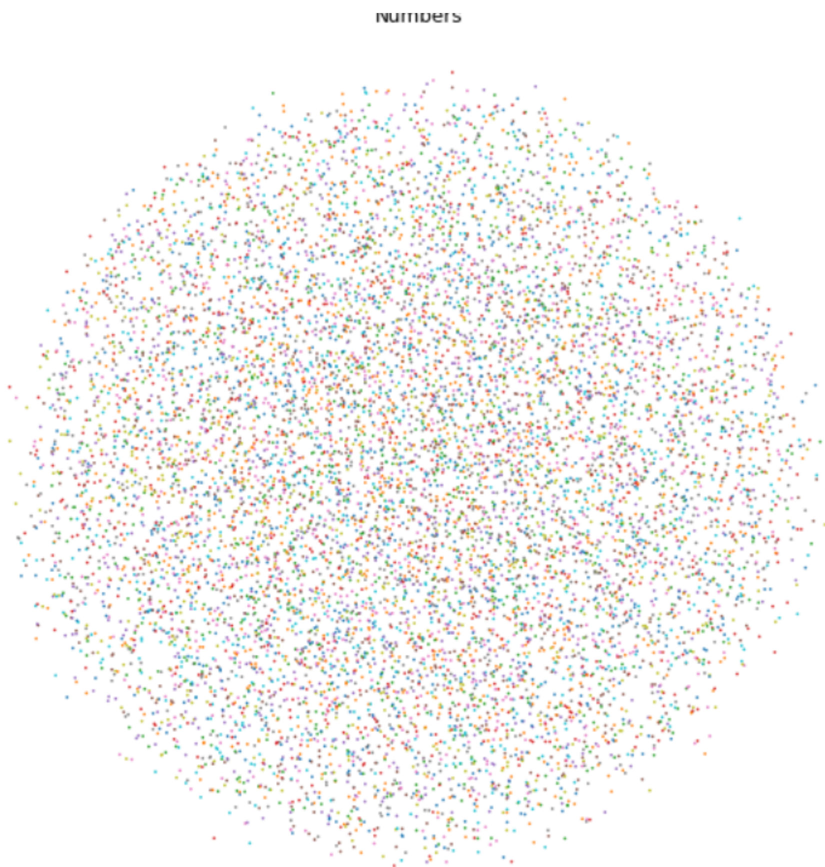

Figure 14: Umap results when kl loss coefficient is 1 and coefficient of recon loss is .001

:

Problem 2 continued on next page. . .

As we decrease the coefficient of **kl_loss** ,there is more opportunity of discriminating clusters but the reconstructed image has less quality.In other words as discrimination is more the reconstructed **MNIST** has less quality like bellow figure which is the reconstructed **MNIST** for figure 14.

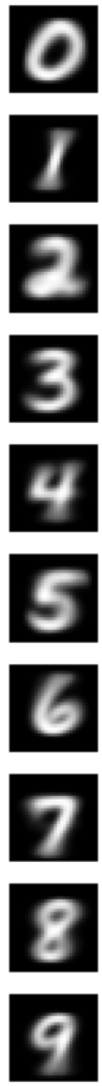

Figure 15: MNIST reconstructed photos for figure14

: