CMPT-726 Assignment 4

Dhruv Patel, 301471961

1. A soft-margin linear SVM with $C = 0.02$.
Answer: ***Plot 4*** represents this soft-margin linear SVM with $C = 0.02$, due to the small value of C (Regularization Parameter) there is more scope of misclassification and soft-margin increases and plot 4 has more visible errors then plot 3.

2. A soft-margin linear SVM with $C = 20$.
Answer: ***Plot 3*** represents this soft-margin linear SVM with $C = 20$. Since regularization parameters increase, the margin will become smaller and have less misclassification. So plot 3 has less visible errors then plot 4.

3. A hard-margin kernel SVM with $k(u, v) = u^{\top}.v + (u^{\top}.v)^2$.
Answer: This hard-margin kernel SVM is represented by ***plot 5***. Since this kernel function is of hyperbola form (i.e quadratic form) and plot 5 represents a perfect match for hyperbola. Since it's hard-margin and compared to plot 2, plot 5 has no misclassification.

4. A hard-margin kernel SVM with $k(u, v) = \exp(-5\|u - v\|_2^2)$.
Answer: This hard-margin kernel SVM is represented by ***plot 6***. Since this kernel function has a close form with a circle. Also, if overall kernel value is increased, it will be able to identify almost all the support vectors and less chance for mis-classification. And accordingly plot 6 has less errors than plot1.

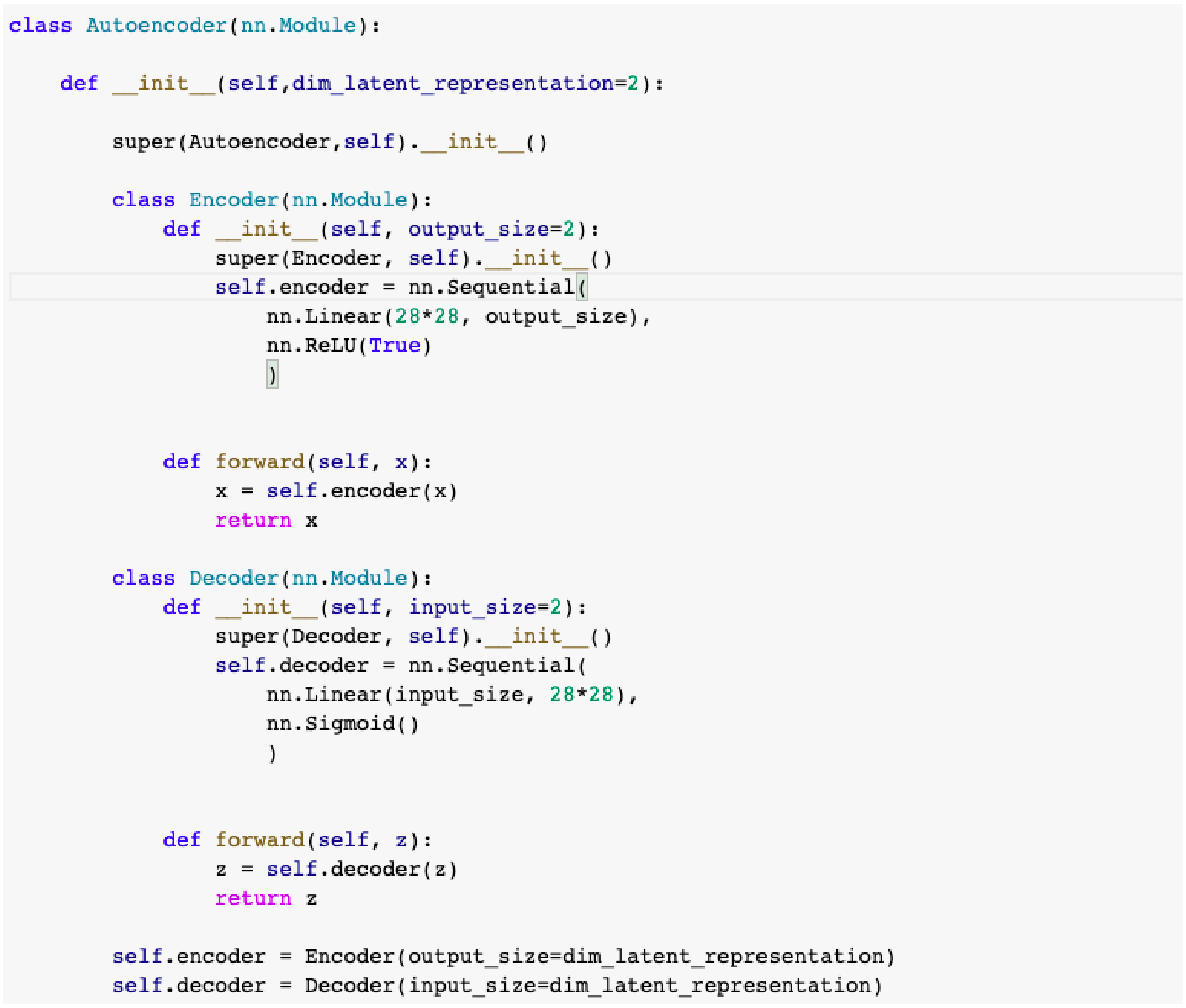
5. A hard-margin kernel SVM with $k(u, v) = \exp(-1^5 \|u - v\|_2^2)$.
Answer: This hard-margin kernel SVM is represented by ***plot 1***. Since this kernel function also has a close match with a circle equation. Since (-1^5) increases more kernel function but, it fails to identify all support vectors which is clearly represented in plot 1.

For Question 2 all the code is developed on colab and sharing the links and files of the same.

1) https://colab.research.google.com/drive/1GBjFh6IktdSv8AmBCREiWhdyhIASSdEw?usp=sharing
2) https://colab.research.google.com/drive/1h757_1bEupyJhrUicRKkMFgEFIS2T8hO?usp=sharing
3) https://colab.research.google.com/drive/1PcD2VKA6vSSlm56Wp4Tp9uULXHcNmUzW?usp=sharing
4) https://colab.research.google.com/drive/1vCpUqUjdA5xeTUcMGMoRZQ9AWAAEYW01?usp=sharing

2. Autoencoders

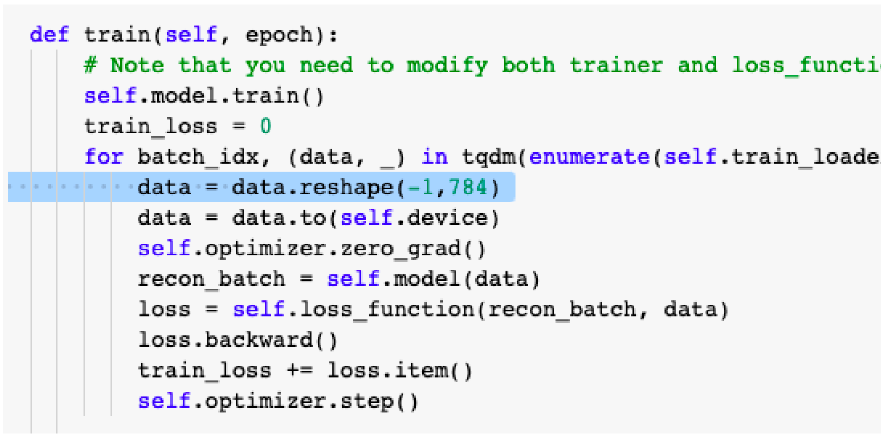
1) Implemented encoder where the decoder with linear models and Sigmoid function added to the decoder so output lies in range [0,1].



```
class Autoencoder(nn.Module):

    def __init__(self,dim_latent_representation=2):

        super(Autoencoder,self).__init__()

        class Encoder(nn.Module):
            def __init__(self, output_size=2):
                super(Encoder, self).__init__()
                self.encoder = nn.Sequential(
                    nn.Linear(28*28, output_size),
                    nn.ReLU(True)
                    )


            def forward(self, x):
                x = self.encoder(x)
                return x

        class Decoder(nn.Module):
            def __init__(self, input_size=2):
                super(Decoder, self).__init__()
                self.decoder = nn.Sequential(
                    nn.Linear(input_size, 28*28),
                    nn.Sigmoid()
                    )


            def forward(self, z):
                z = self.decoder(z)
                return z

        self.encoder = Encoder(output_size=dim_latent_representation)
        self.decoder = Decoder(input_size=dim_latent_representation)
```

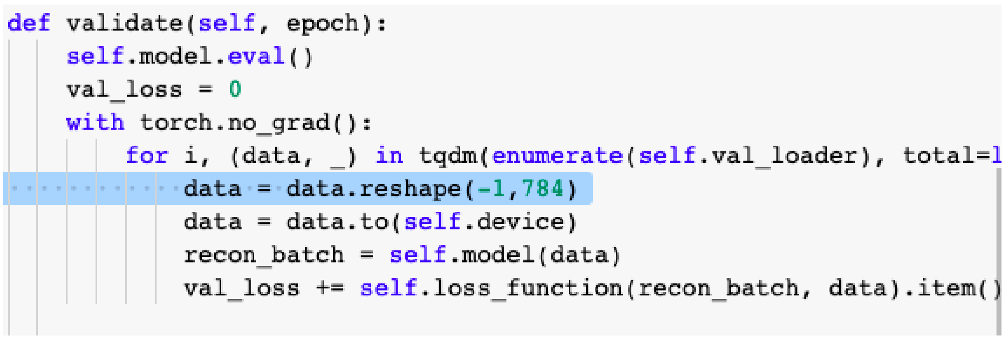
As per hint reshape was used to reduce the dimension and below changes were made in *autoencoderstarter.py*

In train function:



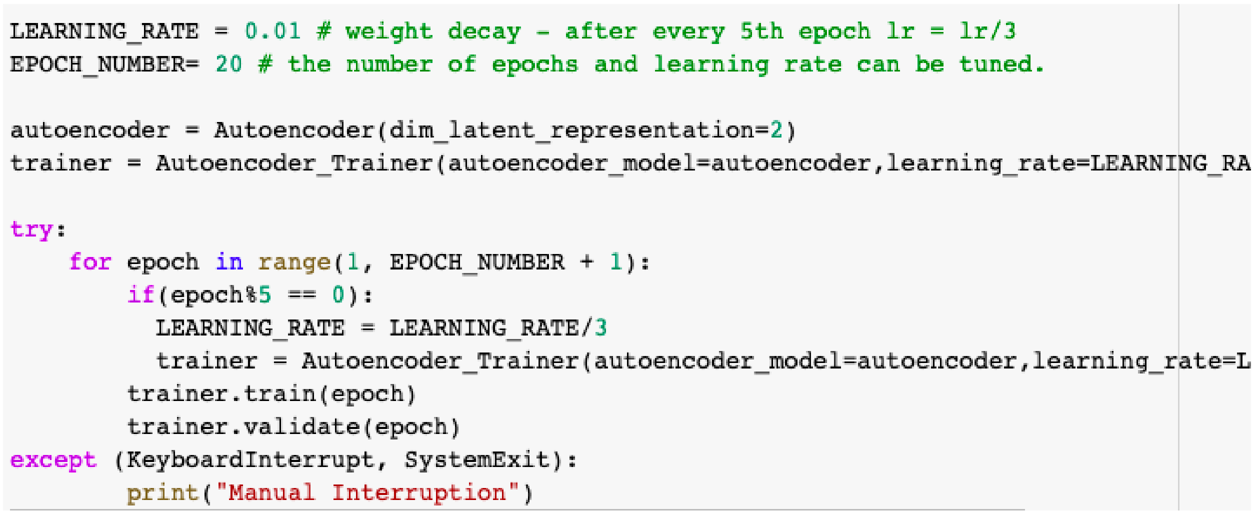
```
def train(self, epoch):
    # Note that you need to modify both trainer and loss_functi
    self.model.train()
    train_loss = 0
    for batch_idx, (data, _) in tqdm(enumerate(self.train_loade
        data = data.reshape(-1,784)
        data = data.to(self.device)
        self.optimizer.zero_grad()
        recon_batch = self.model(data)
        loss = self.loss_function(recon_batch, data)
        loss.backward()
        train_loss += loss.item()
        self.optimizer.step()
```

In validate function:



```
def validate(self, epoch):
    self.model.eval()
    val_loss = 0
    with torch.no_grad():
        for i, (data, _) in tqdm(enumerate(self.val_loader), total=l
            data = data.reshape(-1,784)
            data = data.to(self.device)
            recon_batch = self.model(data)
            val_loss += self.loss_function(recon_batch, data).item()
```

Further hyper-parameters were tuned to get minimum loss of are LR = 0.01 and epoch = 20



```
LEARNING_RATE = 0.01 # weight decay - after every 5th epoch lr = lr/3
EPOCH_NUMBER= 20 # the number of epochs and learning rate can be tuned.

autoencoder = Autoencoder(dim_latent_representation=2)
trainer = Autoencoder_Trainer(autoencoder_model=autoencoder,learning_rate=LEARNING_RA

try:
    for epoch in range(1, EPOCH_NUMBER + 1):
        if(epoch%5 == 0):
          LEARNING_RATE = LEARNING_RATE/3
          trainer = Autoencoder_Trainer(autoencoder_model=autoencoder,learning_rate=L
        trainer.train(epoch)
        trainer.validate(epoch)
except (KeyboardInterrupt, SystemExit):
        print("Manual Interruption")
```

After 20 epochs min average loss is

```
100%|██████████| 1875/1875 [00:15<00:00, 118.70it/s]====> Epoch: 20
Average loss: 0.7248
100%|██████████| 313/313 [00:02<00:00, 128.89it/s]====> Val set loss
(reconstruction error) : 0.7227
```

Scatter Plot

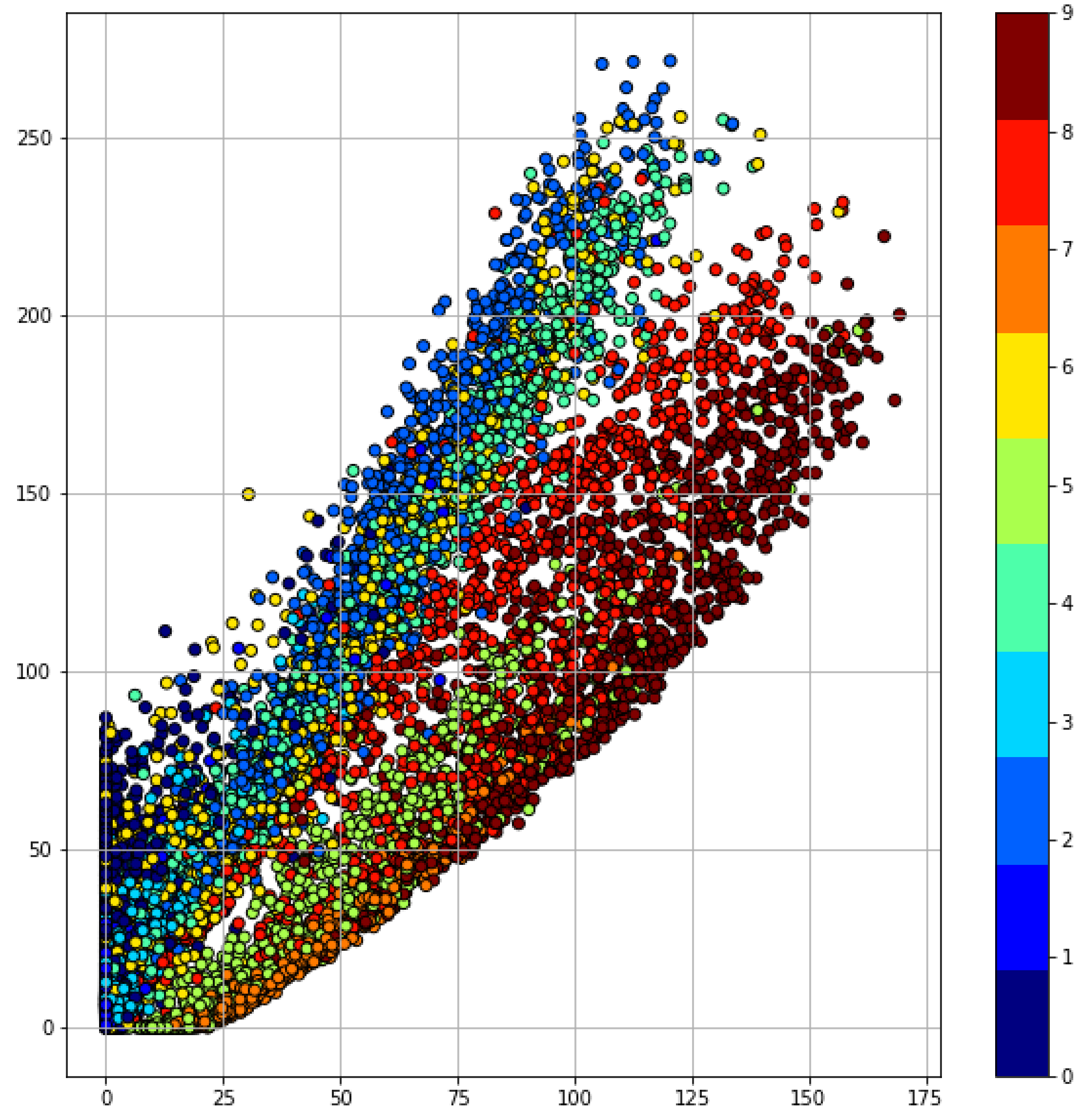
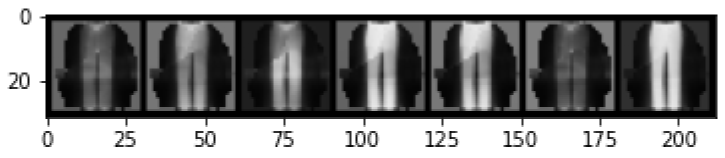
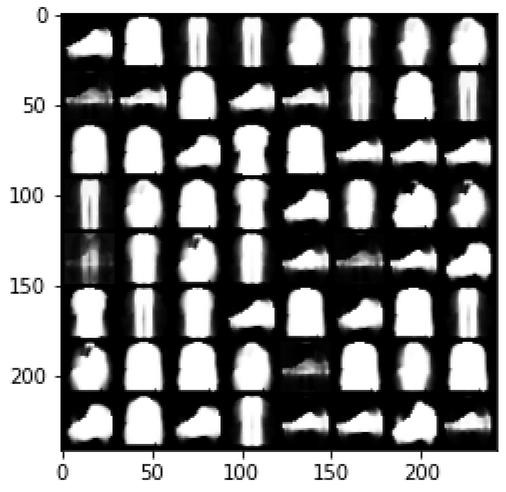


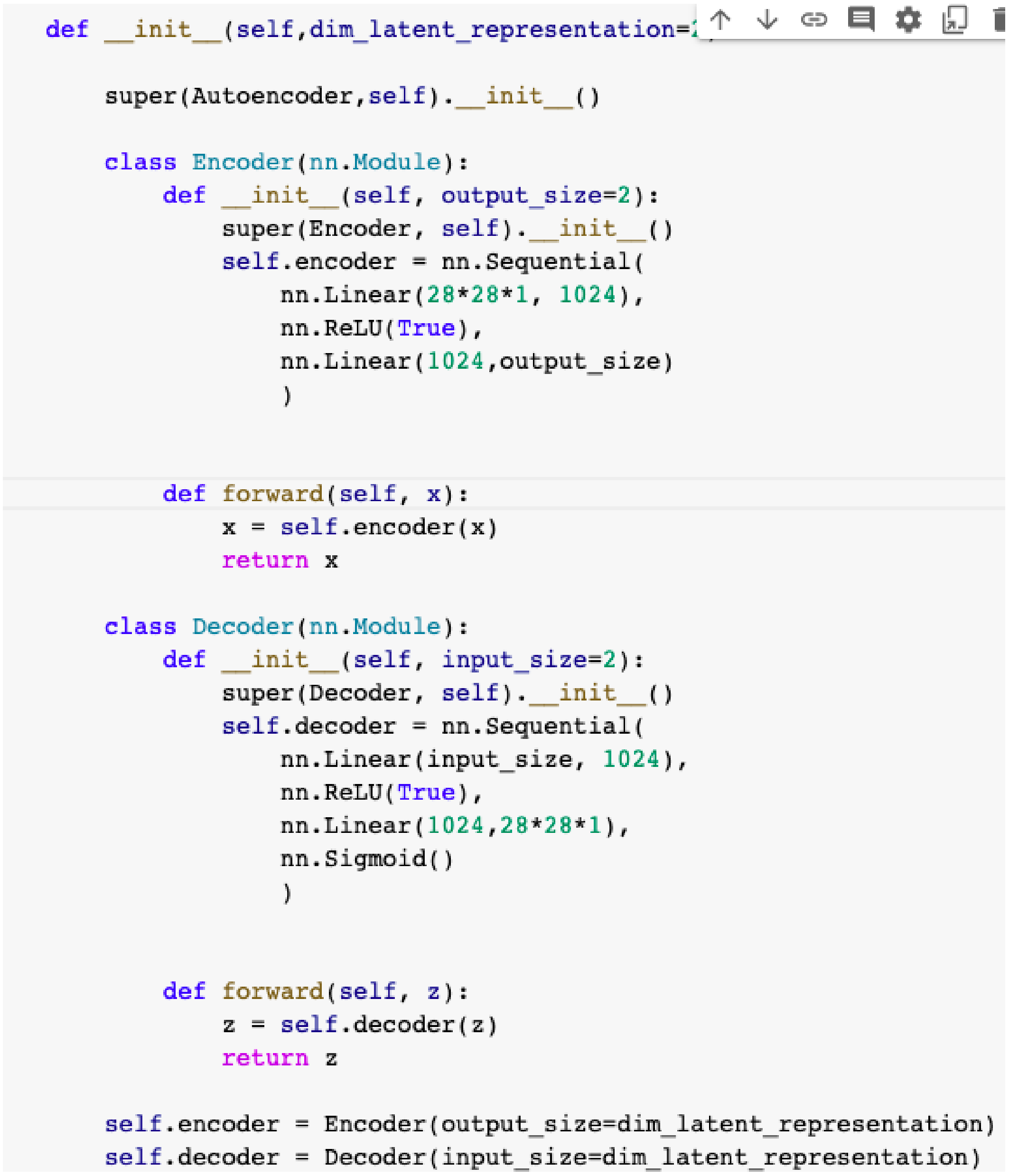
Image output of Sampling bottleneck feature



Reconstructed Image



2. Architecture implemented as per mentioned in question (adding hidden layer and ReLU activation function to both Encoder and Decoder) and below is the screenshot.

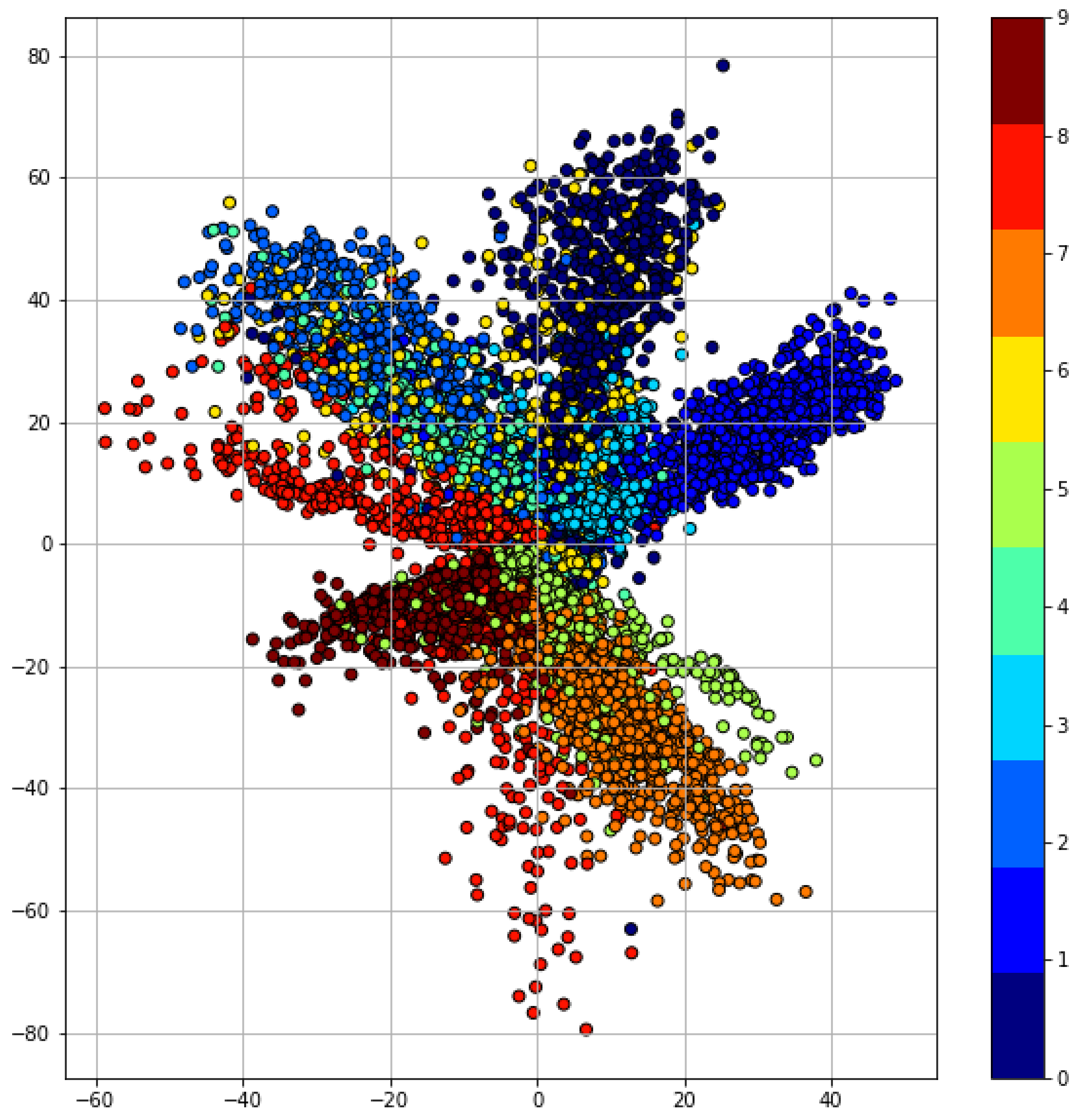


```
def __init__(self,dim_latent_representation=

    super(Autoencoder,self).__init__()

    class Encoder(nn.Module):
        def __init__(self, output_size=2):
            super(Encoder, self).__init__()
            self.encoder = nn.Sequential(
                nn.Linear(28*28*1, 1024),
                nn.ReLU(True),
                nn.Linear(1024,output_size)
                )


        def forward(self, x):
            x = self.encoder(x)
            return x

    class Decoder(nn.Module):
        def __init__(self, input_size=2):
            super(Decoder, self).__init__()
            self.decoder = nn.Sequential(
                nn.Linear(input_size, 1024),
                nn.ReLU(True),
                nn.Linear(1024,28*28*1),
                nn.Sigmoid()
                )


        def forward(self, z):
            z = self.decoder(z)
            return z

    self.encoder = Encoder(output_size=dim_latent_representation)
    self.decoder = Decoder(input_size=dim_latent_representation)
```

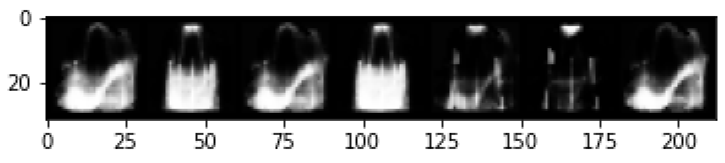
Best Avg. Loss after running the model on several combination

```
100%|██████████| 1875/1875 [01:02<00:00, 29.86it/s]====> Epoch: 20 Average
loss: 0.6519
100%|██████████| 313/313 [00:02<00:00, 108.39it/s]====> Val set loss
(reconstruction error) : 0.6530
```

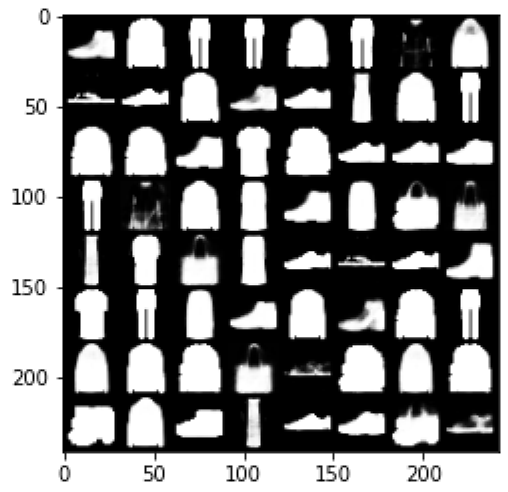
Scatter Plot



Generating Images by Sampling bottleneck features



Reconstructed images

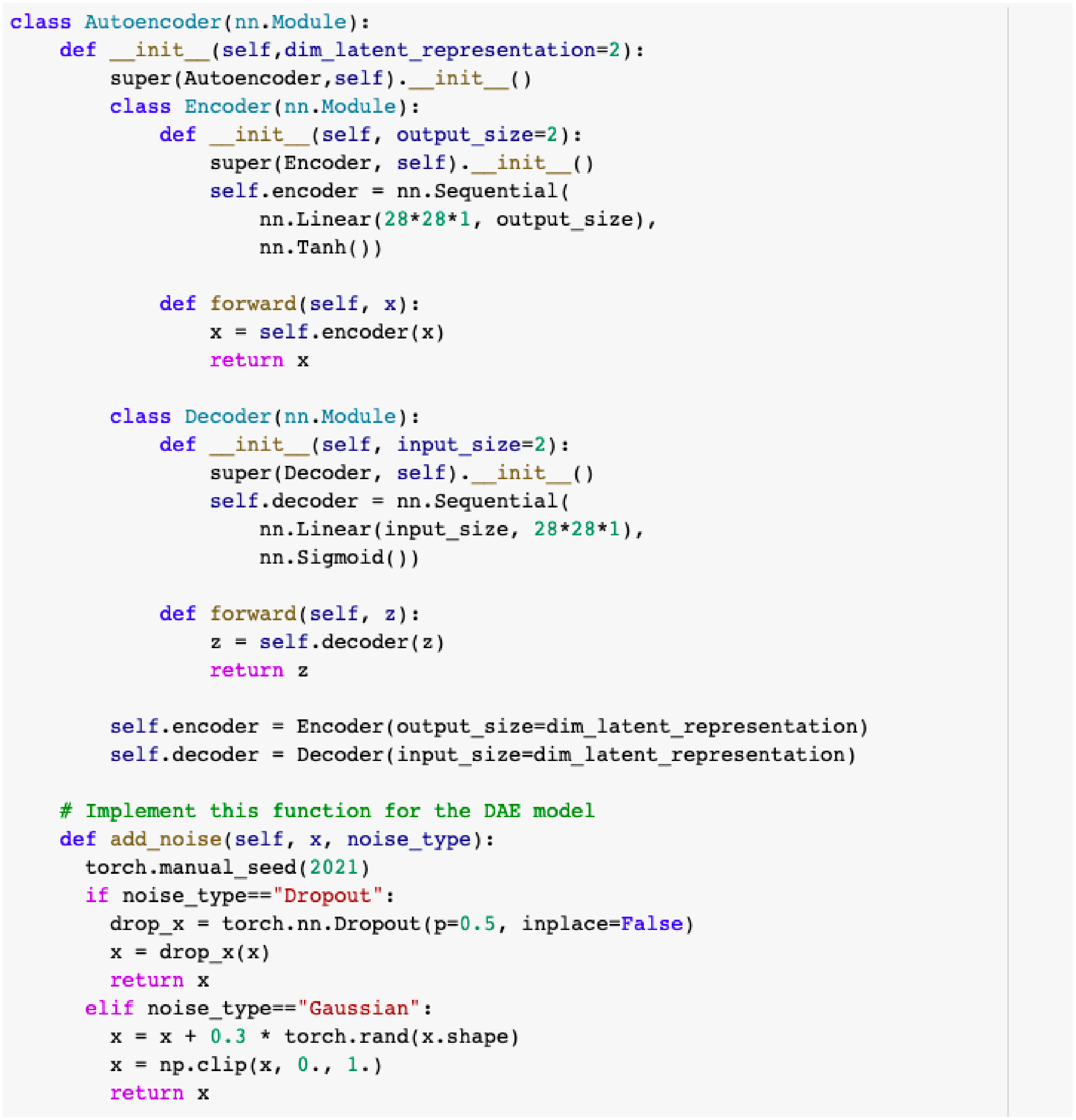


Explanation:

Scatter plot in part 1 was different then in part 2. Since we used only Linear models to train and validate our model in Part 1. So it was just a reduction in dimensionality since there was no activation function added. However, in part we used an additional hidden layer with ReLU activation function which further added non-linearity and helped the model to learn,train and build a complex model with non-linear relationships then in part1. Also, Furthermore, we can verify from two different scatter plots. And because the model picked on other such correlation the reconstruction images differ by a lot.

Also, another important thing here is the trade-off between the computation time and the No. of hidden layers from both different architectures. Since we added a new hidden layer with 1024 inputs. That took extra computation time and cost to train the model. But since, the model was able to carry more weights and hence store more information. It was clearly able to classify classes in a better way than part 1. Also, both training and testing loss was reduced from 0.7428 to 0.6519.

3. Implemented DAE to restrict autoencoders from learning the identify function. Added code with bottleneck size i.e (`dim_latent_representation = 30`) with tanh activation function.



```
class Autoencoder(nn.Module):
    def __init__(self,dim_latent_representation=2):
        super(Autoencoder,self).__init__()
        class Encoder(nn.Module):
            def __init__(self, output_size=2):
                super(Encoder, self).__init__()
                self.encoder = nn.Sequential(
                    nn.Linear(28*28*1, output_size),
                    nn.Tanh())

            def forward(self, x):
                x = self.encoder(x)
                return x

        class Decoder(nn.Module):
            def __init__(self, input_size=2):
                super(Decoder, self).__init__()
                self.decoder = nn.Sequential(
                    nn.Linear(input_size, 28*28*1),
                    nn.Sigmoid())

            def forward(self, z):
                z = self.decoder(z)
                return z

        self.encoder = Encoder(output_size=dim_latent_representation)
        self.decoder = Decoder(input_size=dim_latent_representation)

    # Implement this function for the DAE model
    def add_noise(self, x, noise_type):
      torch.manual_seed(2021)
      if noise_type=="Dropout":
        drop_x = torch.nn.Dropout(p=0.5, inplace=False)
        x = drop_x(x)
        return x
      elif noise_type=="Gaussian":
        x = x + 0.3 * torch.rand(x.shape)
        x = np.clip(x, 0., 1.)
        return x
```

```
  ###Readme:
  """ To run CASE 1: Without Dropout and Gaussian: Comment both add_noise lines in foward function
             CASE 2: With Dropout : Comment add_noise with noise_type = Gaussian
             CASE 3: With Gaussian : Comment add_noise with noise_type = Dropout
  """
def forward(self,x):
  # x = self.add_noise(x, noise_type='Dropout') #uncomment and run for case 2
  x = self.add_noise(x, noise_type='Gaussian') #uncomment and run for case 3
  x = self.encoder(x)
  x = self.decoder(x)
  return x
```

Tuned hyper parameter to best min avg loss with 20 epochs and lr = 0.001

```
LEARNING_RATE = 0.001
EPOCH_NUMBER= 20 # the number of epochs and learning rate can be tuned.

autoencoder = Autoencoder(dim_latent_representation=30)
trainer = Autoencoder_Trainer(autoencoder_model=autoencoder,learning_rate=LEARNING_
```

```
100%|██████████| 1875/1875 [00:16<00:00, 113.69it/s]====> Epoch: 20
Average loss: 0.6159
100%|██████████| 313/313 [00:02<00:00, 127.34it/s]====> Val set loss
(reconstruction error) : 0.6158
```

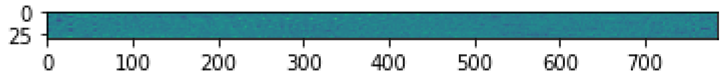
Plotting kernel weights for different cases

```
from autoencoder_starter import Plot_Kernel
plot_kernel = Plot_Kernel(autoencoder)
plt.imshow(plot_kernel.detach().numpy())
```

Implemented tsne function using **hint** from `sklearn.manifold`

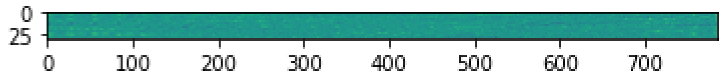
```
tsne = TSNE(n_components=2, verbose=1, perplexity=40, n_iter=300)
tsne_image_plot = tsne.fit_transform(latent_representations)

plt.figure(figsize=(16,10))
sns.scatterplot(tsne_image_plot[:,0], tsne_image_plot[:,1], hue = labels,
        data=latent_representations,
        palette=sns.color_palette("husl", 10),
        legend="full",
        alpha=0.5
)
```

Case 1: plt.imshow for weights of 1st layer without noise



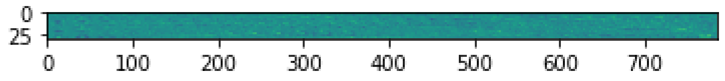
Min Avg. Training Loss = 0.6073 for 20 epochs
Min Avg. Validation Loss = 0.6063 for 20 epochs
We can observe, without noise, that our model learns more and that also confirmed from Avg. Training loss and validation loss. So there is a possibility that our function can learn an identity function and just try to map inputs with outputs.

Case 2: plt.imshow for weights of 1st layer with Dropout noise



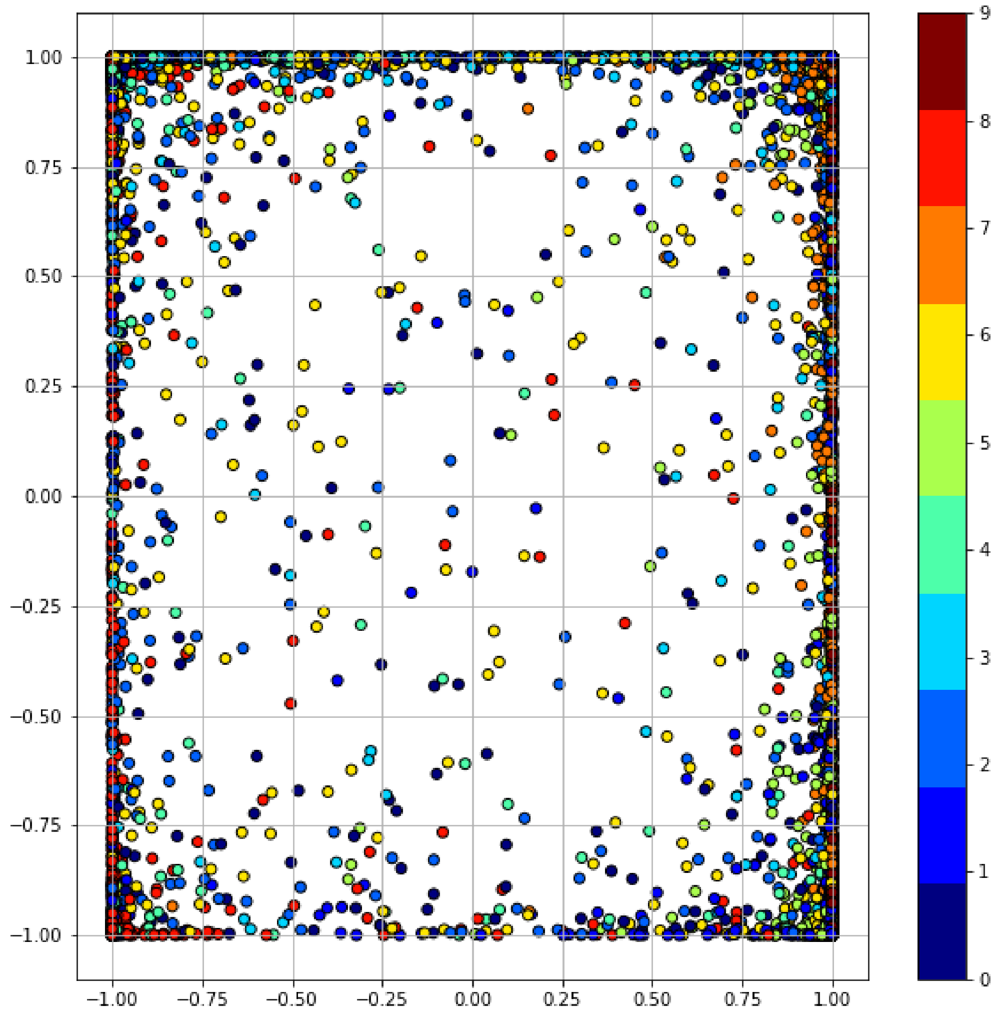
Min Avg. Training Loss = 0.6157 for 20 epochs
Min Avg. Validation Loss = 0.6153 for 20 epochs
Also, we notice small differences in plot. As we add noise we get more disrupted images as compared to before. As a result, the model is not overtrained.

Case 3: plt.imshow for weights of 1st layer with Gaussian noise.

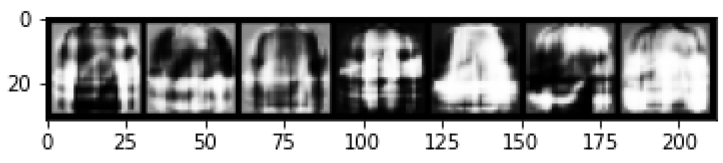


Min Avg. Training Loss = 0.6059 for 20 epochs.
Min Avg. Validation Loss = 0.6052 for 20 epochs.
Also, we can notice that when we add Gaussian noise. The image plot for kernel weights is not smooth as when dropout noise. Which shows that the model is learning on gaussian noise and also, we can verify from training and validation loss since its better then we add dropout noise.

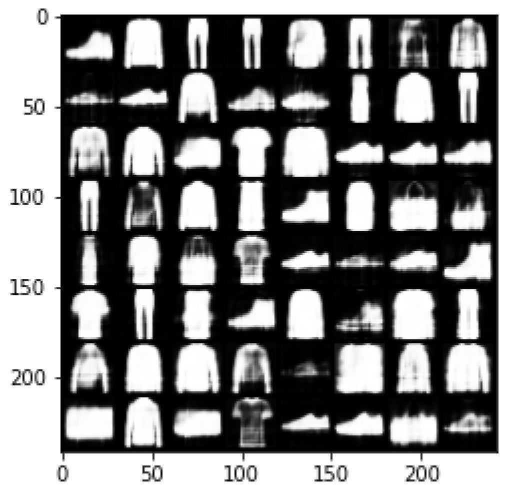
Scatter Plot



Generating image by sampling bottle neck Feature



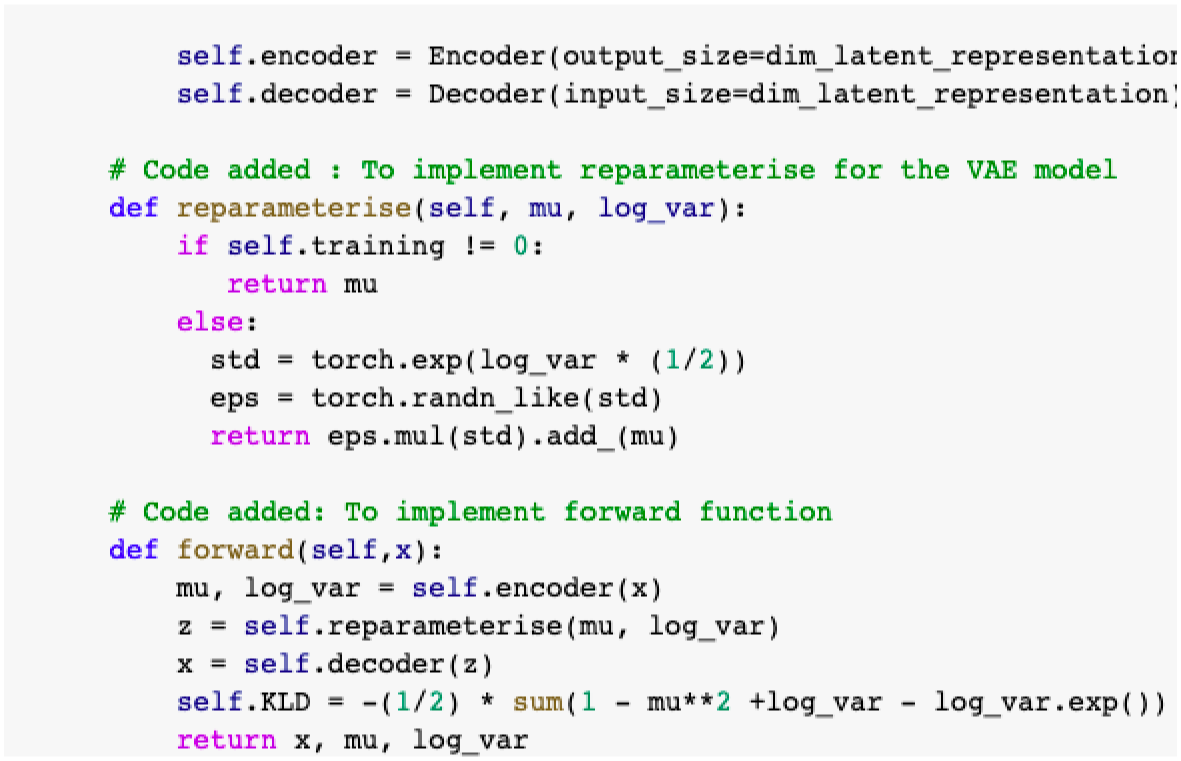
Reconstructed Image



Explanation: Autoencoder as we know, learns to reduce dimension to efficiently compress to lowest dimensionality possible and again reconstruct the images. They also have the capacity to memorize the mapping between inputs and outputs. So to avoid that we introduce noise with inputs, so it will train better and give a clear image on reconstruction. Linear Kernel in part 1 was trained on linear models so reconstruction image was not as compared to part 2. Adding noise to the autoencoder produced the best reconstruction image so far. Important thing to notice here is that adding noise to the autoencoder avoids it from learning identity function and getting an over-fit model.

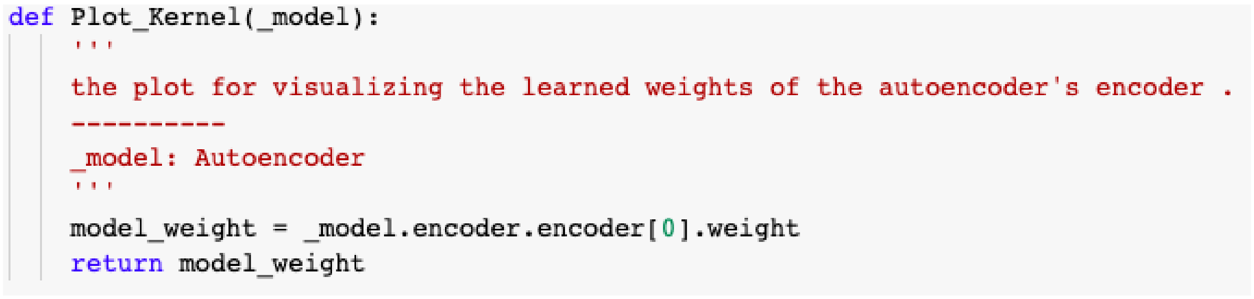
4. Implemented code for (VAE) with bottleneck size 30 and tanh activation function, also added KDL term for loss function.

```
class VAE(nn.Module):

    def __init__(self,dim_latent_representation = 2):

        super(VAE,self).__init__()

        # Code added: To define KLD variable
        self.KLD = 0.0

        # Code added: To implement Decoder Class
        class Encoder(nn.Module):
            def __init__(self, output_size = 2):
                super(Encoder, self).__init__()
                self.encoder_layer_1 = nn.Linear(28*28*1,1024)
                self.encoder_mu = nn.Linear(1024,output_size)
                self.encoder_log_var = nn.Linear(1024,output_size)

            def forward(self, x):
                x = torch.tanh(self.encoder_layer_1(x))
                self.mu = self.encoder_mu(x)
                self.log_var = self.encoder_log_var(x)
                return self.mu, self.log_var

        # Code added: To implement Decoder Class
        class Decoder(nn.Module):
            def __init__(self, input_size = 2):
                super(Decoder, self).__init__()
                self.decoder_layer_1 = nn.Linear(input_size,1024)
                self.decoder_layer_2 = nn.Linear(1024,28*28*1)

            def forward(self, z):
                z = funcs.relu(self.decoder_layer_1(z))
                z_sig = torch.sigmoid(self.decoder_layer_2(z))
                return z_sig
```



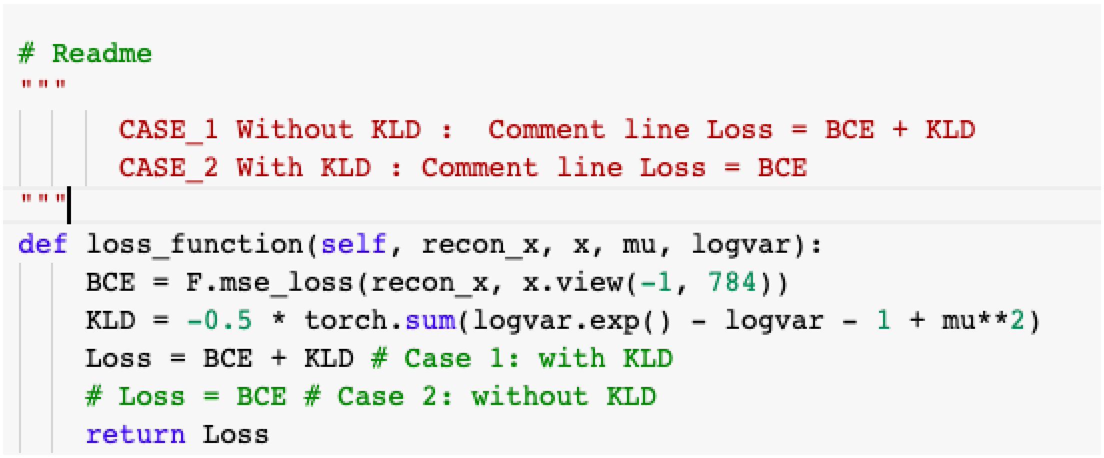
```
        self.encoder = Encoder(output_size=dim_latent_representatio
        self.decoder = Decoder(input_size=dim_latent_representation

    # Code added : To implement reparameterise for the VAE model
    def reparameterise(self, mu, log_var):
        if self.training != 0:
           return mu
        else:
          std = torch.exp(log_var * (1/2))
          eps = torch.randn_like(std)
          return eps.mul(std).add_(mu)

    # Code added: To implement forward function
    def forward(self,x):
        mu, log_var = self.encoder(x)
        z = self.reparameterise(mu, log_var)
        x = self.decoder(z)
        self.KLD = -(1/2) * sum(1 - mu**2 +log_var - log_var.exp())
        return x, mu, log_var
```

Implemented Plot_Kernel in `VAE_starter.py`



```
def Plot_Kernel(_model):
    '''
    the plot for visualizing the learned weights of the autoencoder's encoder .
    ----------
    _model: Autoencoder
    '''
    model_weight = _model.encoder.encoder[0].weight
    return model_weight
```

Implemented loss function for VAE_starter.py



```
# Readme
"""
      CASE_1 Without KLD :  Comment line Loss = BCE + KLD
      CASE_2 With KLD : Comment line Loss = BCE
"""
def loss_function(self, recon_x, x, mu, logvar):
    BCE = F.mse_loss(recon_x, x.view(-1, 784))
    KLD = -0.5 * torch.sum(logvar.exp() - logvar - 1 + mu**2)
    Loss = BCE + KLD # Case 1: with KLD
    # Loss = BCE # Case 2: without KLD
    return Loss
```

Best Avg loss after 20 epochs, we get

Without KLD

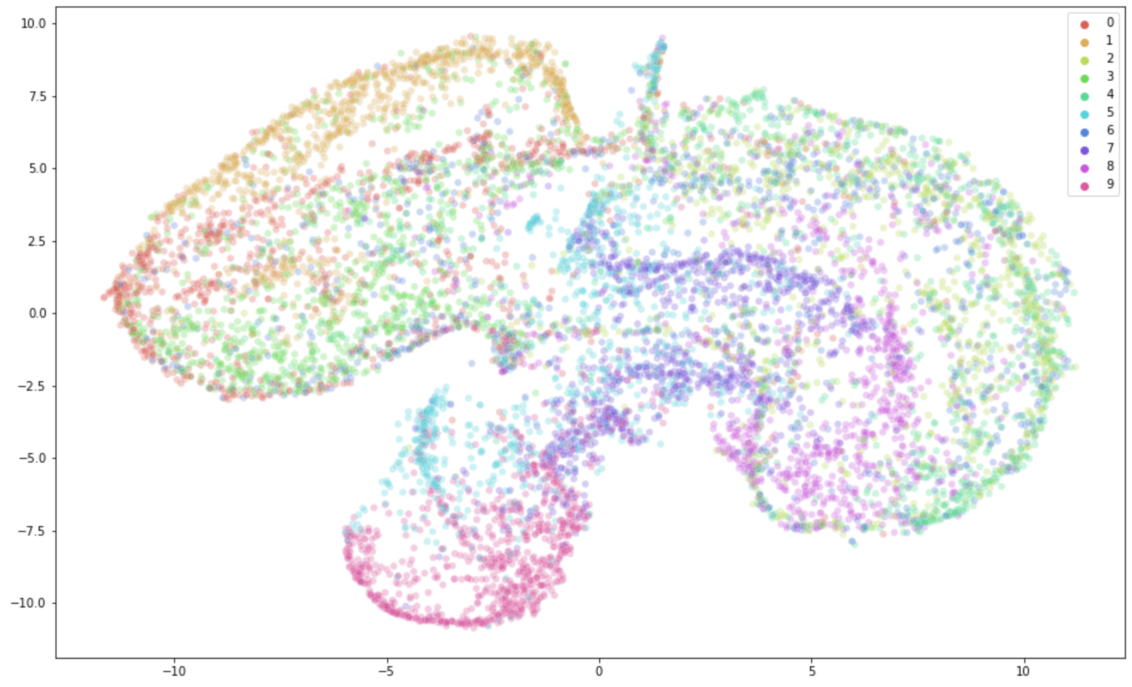
```
100%|██████████| 1875/1875 [07:09<00:00,  4.37it/s]====> Epoch: 20 Average
loss: 0.9937
100%|██████████| 313/313 [00:03<00:00, 95.90it/s] ====> Val set loss
(reconstruction error) : 0.9941
```

With KLD

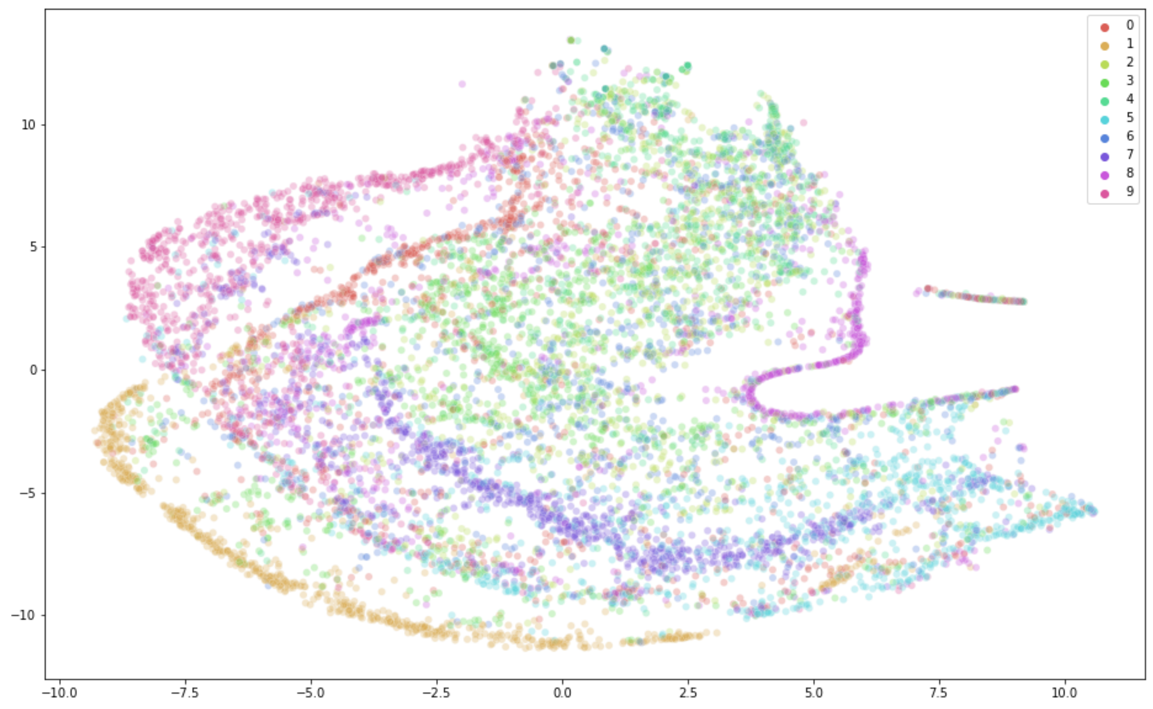
```
100%|██████████| 1875/1875 [01:01<00:00, 30.46it/s]====> Epoch: 20 Average
loss: 0.9920
100%|██████████| 313/313 [00:03<00:00, 93.94it/s]====> Val set loss
(reconstruction error) : 0.9930
```

Scatter Plot for view latent space

Without KLD:



With KLD:



Explanation: Latent space is very important, because it enables learning the features from the data and finding patterns. The compressed data by encoder is also encoded space or latent space. VAE's encodings distribution is regularised during the training in order to ensure that its latent space has good properties allowing us to generate some new data. As we introduce the KLD term, values are concentrated towards the mean, and the still boundary is easily identifiable. Before that, we could easily separate the classes but the values were not concentrated towards the mean.