CMPT726 Assignment 4

Name: Yin Yu Kevani Chow Student ID: 301436160

Question 1

1. A soft-margin linear SVM with $C = 0.02$.

Answer: **Diagram 4 in figure 1**

Explanation:
The given decision boundary of a soft-margin linear SVM is linear and C parameter is smaller than that of question 2 which will lead to a larger margin hyperplane and allow more error. Since diagram 4 in figure 1 does not separate circles and squares strictly, it should be represented by this soft-margin linear SVM with smaller C value.

2. A soft-margin linear SVM with $C = 20$.

Answer: **Diagram 3 in figure 1**

Explanation:
Similar to question 1, the C parameter in question 2 is larger than that of question 1 which will lead to a smaller margin hyperplane which would perform better to classify two classes correctly comparatively. Since diagram 3 in figure 1 separates circles and squares strictly, it should be represented by this soft-margin linear SVM with larger C value.

3. A hard-margin kernel SVM with $k(u, v) = u^\top v + (u^\top v)^2$

Answer: **Diagram 5 in figure 1**

Explanation:
We can substitute *k(u, v)*, polynomial (quadratic) kernel SVM, into a decision function (classifier) and rewrite as below:

$$f(x) = \sum_i \alpha_i (uv + (uv)^2) + b$$

The decision boundary is $f(x) = 0$ and polynomial degree of $f(x)$ is 2 which is a second order function of x so it is possible to be ellipse or hyperbola. As a result, since **diagram 5 in Figure 1** is a hyperbolic curve, it is possible representation of kernel SVM $k(u, v) = u^T v + (u^T v)^2$.

4. A hard-margin kernel SVM with $k(u, v) = \exp(-5||u - v||_2^2)$

Answer: **Diagram 6 in figure 1**

Explanation:

We can substitute *k(u, v)* this Gaussian kernel SVM into a decision function (classifier) and rewrite as below:

$$f(x) = \sum_i \alpha_i exp\ (-5||u - v||_2^2)$$

In this question, γ = 5 which is larger than that of question 5. When γ is large, the kernel value is small correspondingly which implies that the model performs better to classify the circle when there are more supporting vectors (overfitting) but it is hard to perform classification of the circles point in diagram 1 with few supporting vectors.

5. A hard-margin kernel SVM with $k(u, v) = \exp(-\frac{1}{5}||u - v||_2^2)$

Answer: **Diagram 1 in figure 1**

Explanation:

We can substitute *k(u, v)* this Gaussian kernel SVM into a decision function (classifier) and rewrite as below:

$$f(x) = \sum_i \alpha_i exp\ (-\frac{1}{5}||u - v||_2^2)$$

In this question, $\gamma = \frac{1}{5}$ which is smaller than that of question 4. When γ is small, the kernel value is large correspondingly which means that the model is able to avoid overfitting and performs quite well to classify the circle points when there are few supporting vectors as what is shown in diagram 1.

Question 2

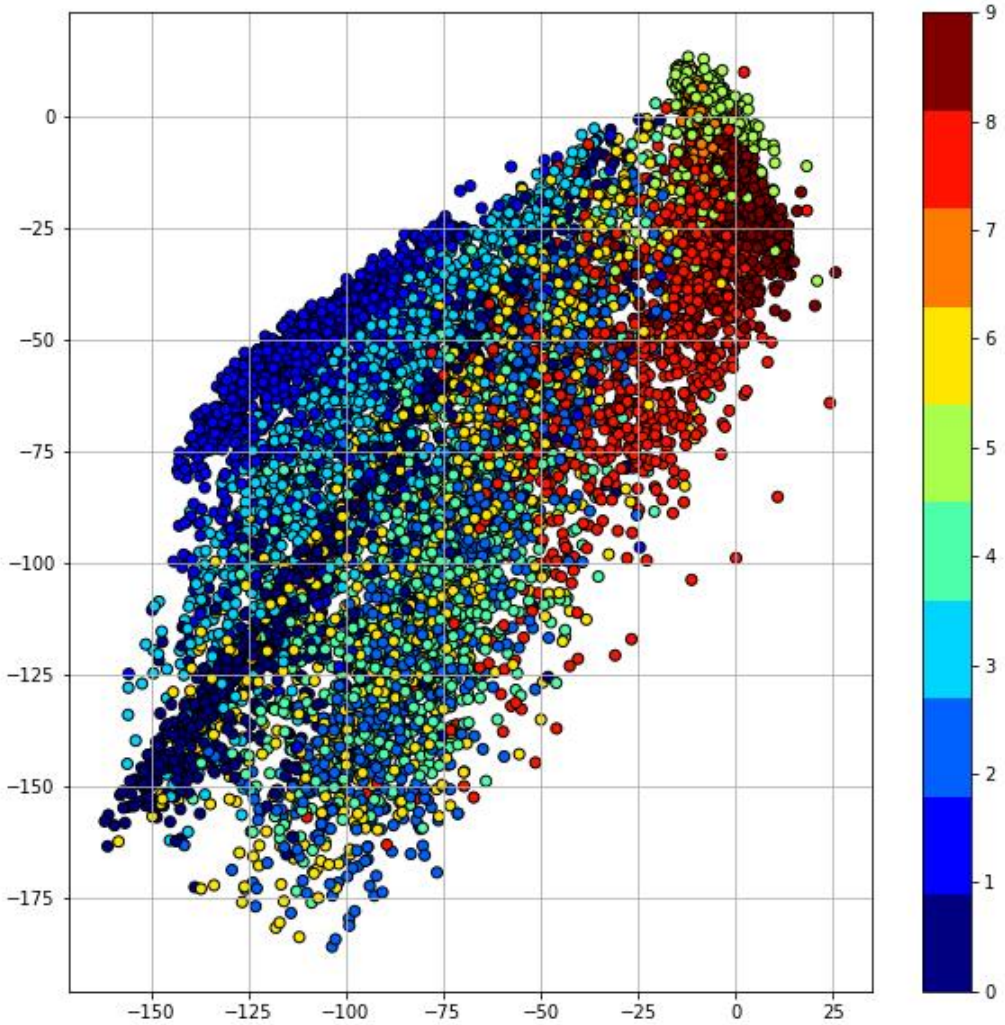
Part 1 – linear models

Screenshot of the code that implements the architecture

```
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__()
              # needs your implementation
              self.layers = nn.Sequential(
                nn.Linear(in_features = 784, out_features = 2), # no hidden layer 28*28 ->784
              )

          def forward(self, x):
              # needs your implementation
              one_d_x = x.shape[0]
              encoded_x = self.layers(torch.reshape(x,(one_d_x, 784))) # no hidden layer 28*28 ->784
              return encoded_x

      class Decoder(nn.Module):
          def __init__(self, input_size=2):
              super(Decoder, self).__init__()
              # needs your implementation
              self.layers = nn.Sequential(
                nn.Linear(in_features = 2, out_features = 784), # no hidden layer 28*28 ->784
                nn.Sigmoid()
              )

          def forward(self, z):
              # needs your implementation
              decoded_z = self.layers(z)
              one_d_z = decoded_z.shape[0]
              output_img = torch.reshape(decoded_z,(one_d_z,1,28,28))
              return output_img

      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):
  #     if noise_type=='Gaussian':
  #         # return (x with Gaussian noise)
  #     elif noise_type=='Dropout':
  #         # return (x with Dropout noise)

  # Implement this function for the VAE model
  # def reparameterise(self, mu, logvar):
  #     if self.training:
  #         # return reparametrized mu
  #     else:
  #         return mu

  def forward(self,x):
      # This function should be modified for the DAE and VAE
      x = self.encoder(x)
      x = self.decoder(x)
      # for the VAE forward function should also return mu and logvar
      return x
```

Screenshot of the generated scatter plot



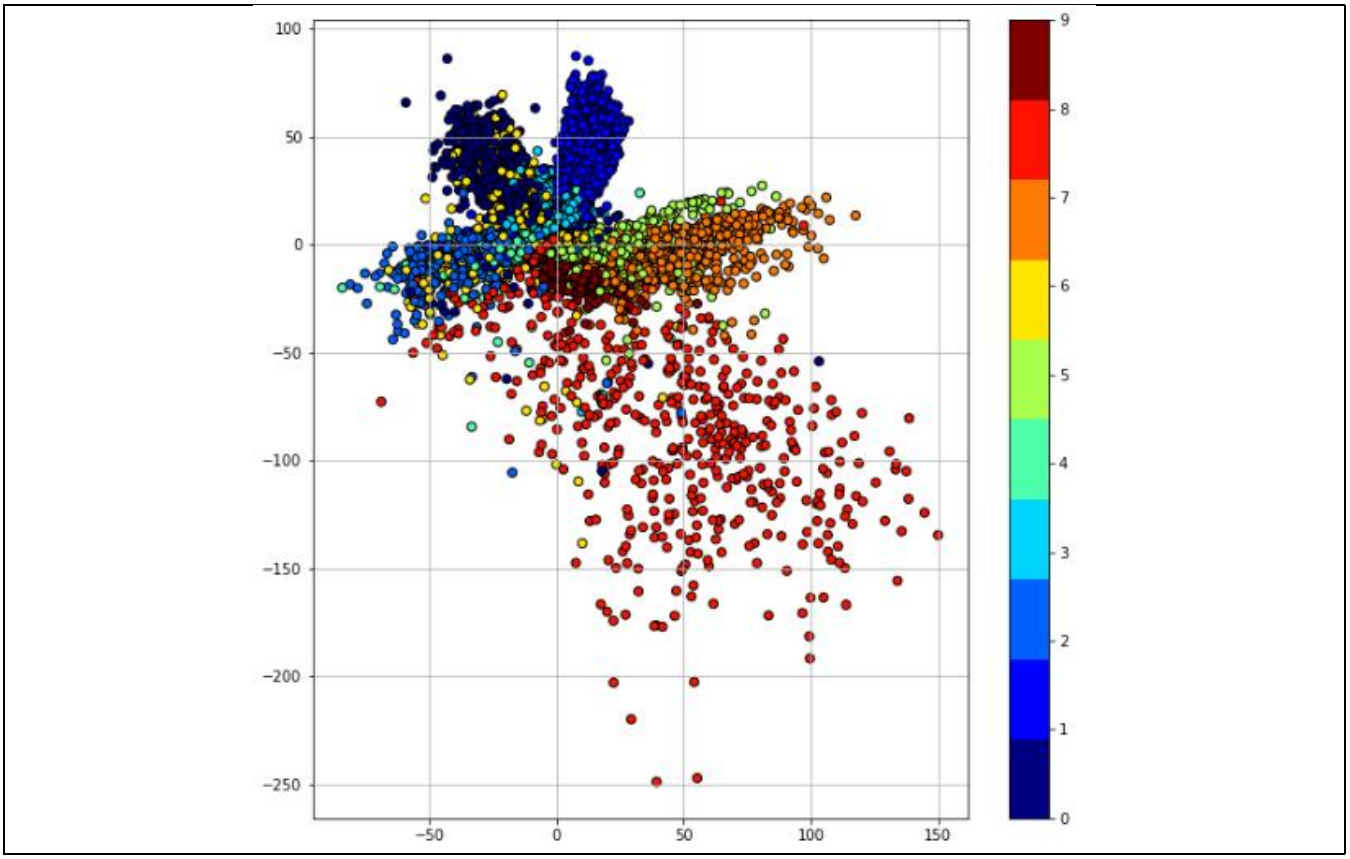
Part 2 – Two layers

Screenshot of the code that implements the architecture

```
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__()
                # needs your implementation
                self.layers = nn.Sequential(
                  nn.Linear(in_features = 784, out_features = 1024), # 1st layer
                  nn.ReLU(),
                  nn.Linear(in_features =1024, out_features = 2) # 2nd layer
                )

            def forward(self, x):
                # needs your implementation
                one_d_x = x.shape[0]
                encoded_x = self.layers(torch.reshape(x,(one_d_x, 784)))
                return encoded_x

        class Decoder(nn.Module):
            def __init__(self, input_size=2):
                super(Decoder, self).__init__()
                # needs your implementation
                self.layers = nn.Sequential(
                  nn.Linear(in_features = 2, out_features = 1024), # 1st layer
                  nn.ReLU(),
                  nn.Linear(in_features = 1024, out_features = 784), # 2nd layer
                  nn.Sigmoid()
                )

            def forward(self, z):
                # needs your implementation
                decoded_z = self.layers(z)
                one_d_z = decoded_z.shape[0]
                output_img = torch.reshape(decoded_z,(one_d_z,1,28,28))
                return output_img

        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):
    #     if noise_type=='Gaussian':
    #         # return (x with Gaussian noise)
    #     elif noise_type=='Dropout':
    #         # return (x with Dropout noise)

    # Implement this function for the VAE model
    # def reparameterise(self, mu, logvar):
    #     if self.training:
    #         # return reparametrized mu
    #     else:
    #         return mu

    def forward(self,x):
        # This function should be modified for the DAE and VAE
        x = self.encoder(x)
        x = self.decoder(x)
        # for the VAE forward function should also return mu and logvar
        return x
```

Screenshot of the generated scatter plot



Describe how the plot differs from the one in part (1)

The plot in part (2) are separated more clearly into different classes as we would see the dots with same colours are grouped together. However, in part (1), although the colour points (classes) are grouped together, all the colour points (classes) are overlapping to each other.

Explain what this says about the architectures in this part and part (1).

The architecture in part (2) has a deeper neural network than part (1), as there are 2 layers which are fully connected but there is no hidden layer in part (1). By comparing the 2 scatter plot, deeper neural network which is in part (2) defining the classes better.

Why do you think the architecture in this part gave rise to the results shown in the scatter plot?

Since, the architecture in part (2) has deeper network which should perform better with more appropriate capacity, lower loss and higher accuracy. Yet, part (1) has shallow network which will tend to be underfitting, higher loss and lower accuracy.

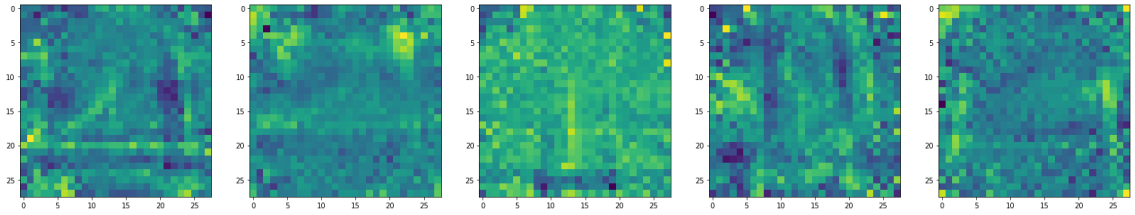
Part 3 - No added noise

Screenshot of the code that implements the architecture

```
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__()
                # needs your implementation
                self.layers = nn.Sequential(
                  nn.Linear(in_features = 784, out_features = dim_latent_representation), # no hidden layer 28*28 ->784
                  nn.Tanh() # tanh activation function
                )

            def forward(self, x):
                # needs your implementation
                one_d_x = x.shape[0]
                encoded_x = self.layers(torch.reshape(x,(one_d_x, 784))) # no hidden layer 28*28 ->784
                return encoded_x

        class Decoder(nn.Module):
            def __init__(self, input_size=2):
                super(Decoder, self).__init__()
                # needs your implementation
                self.layers = nn.Sequential(
                  nn.Linear(in_features = dim_latent_representation, out_features = 784), # no hidden layer 28*28 ->784
                  nn.Tanh()
                )

            def forward(self, z):
                # needs your implementation
                decoded_z = self.layers(z)
                one_d_z = decoded_z.shape[0]
                output_img = torch.reshape(decoded_z,(one_d_z,1,28,28))
                return output_img

        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):
    #     if noise_type=='Gaussian':
    #         # return (x with Gaussian noise)
    #     elif noise_type=='Dropout':
    #         # return (x with Dropout noise)

    # Implement this function for the VAE model
    # def reparameterise(self, mu, logvar):
    #     if self.training:
    #         # return reparametrized mu
    #     else:
    #         return mu

    def forward(self,x):
        # This function should be modified for the DAE and VAE
        x = self.encoder(x)
        x = self.decoder(x)
        # for the VAE forward function should also return mu and logvar
        return x
```

Screenshot of generated kernel plots - (captured 5 for example)



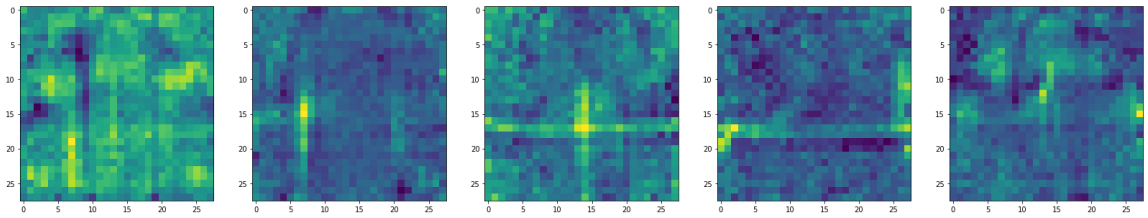
Added dropout noise turning some of input values to zero

Screenshot of the code that implements the architecture

```
class DAE(nn.Module):

    def __init__(self,dim_latent_representation=2):

        super(DAE,self).__init__()

        class Encoder(nn.Module):
            def __init__(self, output_size=2):
                super(Encoder, self).__init__()
                # needs your implementation
                self.layers = nn.Sequential(
                  nn.Linear(in_features = 784, out_features = dim_latent_representation), # no hidden layer 28*28 ->784
                  nn.Tanh() # tanh activation function
                )

            def forward(self, x):
                # needs your implementation
                one_d_x = x.shape[0]
                encoded_x = self.layers(torch.reshape(x,(one_d_x, 784))) # no hidden layer 28*28 ->784
                return encoded_x

        class Decoder(nn.Module):
            def __init__(self, input_size=2):
                super(Decoder, self).__init__()
                # needs your implementation
                self.layers = nn.Sequential(
                  nn.Linear(in_features = dim_latent_representation, out_features = 784), # no hidden layer 28*28 ->784
                  nn.Tanh()
                )

            def forward(self, z):
                # needs your implementation
                decoded_z = self.layers(z)
                one_d_z = decoded_z.shape[0]
                output_img = torch.reshape(decoded_z,(one_d_z,1,28,28))
                return output_img

        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):
      if noise_type=='Gaussian':
        # return (x with Gaussian noise)
        noise_factor = 0.1**0.5
        gaussian = noise_factor*torch.randn_like(x)
        noisy_x = x + gaussian
        return noisy_x
      elif noise_type=='Dropout':
        # return (x with Dropout noise)
        dropout = nn.Dropout(p=0.2)
        noisy_x = dropout(x)
        return noisy_x

    # Implement this function for the VAE model
    # def reparameterise(self, mu, logvar):
    #     if self.training:
    #         # return reparametrized mu
    #     else:
    #         return mu

    def forward(self,x):
        # This function should be modified for the DAE and VAE
        #dropout noise: turning some of inoput values to zero
        x = self.add_noise(x,'Dropout')
        x = self.encoder(x)
        x = self.decoder(x)
        # for the VAE forward function should also return mu and logvar
        return x
```

Screenshot of generated kernel plots - (captured 5 for example)



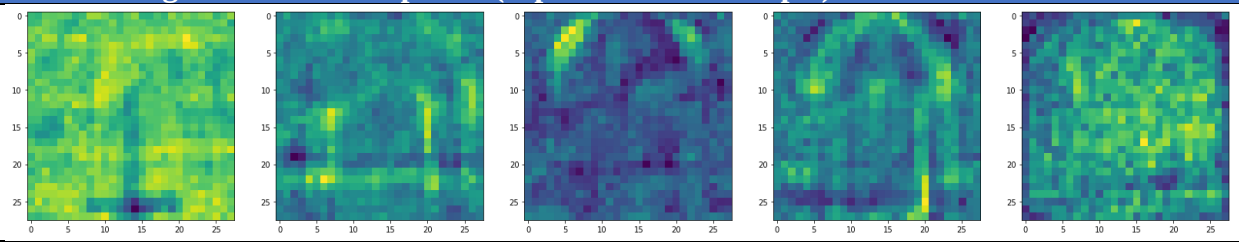
Added Gaussian nose

Screenshot of the code that implements the architecture

```
class DAE(nn.Module):

    def __init__(self,dim_latent_representation=2):

        super(DAE,self).__init__()

        class Encoder(nn.Module):
            def __init__(self, output_size=2):
                super(Encoder, self).__init__()
                # needs your implementation
                self.layers = nn.Sequential(
                  nn.Linear(in_features = 784, out_features = dim_latent_representation), # no hidden layer 28*28 ->784
                  nn.Tanh() # tanh activation function
                )

            def forward(self, x):
                # needs your implementation
                one_d_x = x.shape[0]
                encoded_x = self.layers(torch.reshape(x,(one_d_x, 784))) # no hidden layer 28*28 ->784
                return encoded_x

        class Decoder(nn.Module):
            def __init__(self, input_size=2):
                super(Decoder, self).__init__()
                # needs your implementation
                self.layers = nn.Sequential(
                  nn.Linear(in_features = dim_latent_representation, out_features = 784), # no hidden layer 28*28 ->784
                  nn.Tanh()
                )

            def forward(self, z):
                # needs your implementation
                decoded_z = self.layers(z)
                one_d_z = decoded_z.shape[0]
                output_img = torch.reshape(decoded_z,(one_d_z,1,28,28))
                return output_img

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

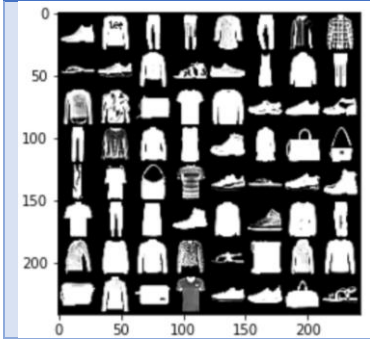
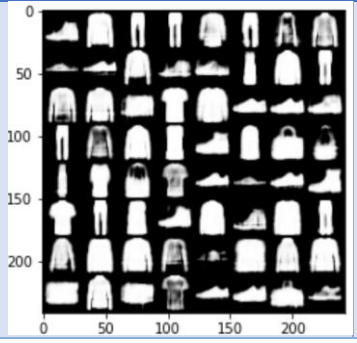
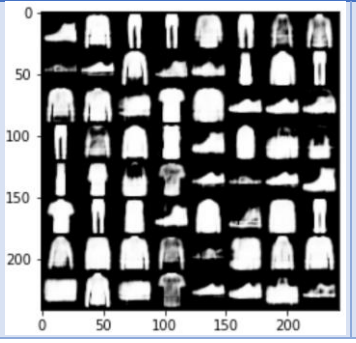
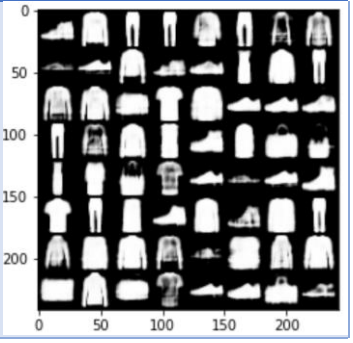
```
    # Implement this function for the DAE model
    def add_noise(self, x, noise_type):
      if noise_type=='Gaussian':
        # return (x with Gaussian noise)
        noise_factor = 0.1**0.5
        gaussian = noise_factor*torch.randn_like(x)
        noisy_x = x + gaussian
        return noisy_x
      elif noise_type=='Dropout':
        # return (x with Dropout noise)
        dropout = nn.Dropout(p=0.2)
        noisy_x = dropout(x)
        return noisy_x

    # Implement this function for the VAE model
    # def reparameterise(self, mu, logvar):
    #     if self.training:
    #         # return reparametrized mu
    #     else:
    #         return mu


    def forward(self,x):
        # This function should be modified for the DAE and VAE
        # Gaussian noise
        x = self.add_noise(x,'Gaussian')
        x = self.encoder(x)
        x = self.decoder(x)
        # for the VAE forward function should also return mu and logvar
        return x
```

Screenshot of generated kernel plots - (captured 5 for example)



Describe how the kernels differ from each other and explain what this says about the denoising autoencoder.

When comparing the kernel plot with dropout noise to that with no noise added, some details were distorted. For the kernel plots with gaussian noise, the pictures are more distorted and more detail loss comparing with that with dropout noise.

Original images	DAE result with no noise added	DAE result with Dropout noise added	DAE result with Gaussian noise added
			

For the result above, even if the data is partially corrupted by noises added to the input vector in a stochastic manner, denoising autoencoder can still preform to decode the sample well by calculate the MSE loss between generated sample and original input which proves its robustness. It shows a clearer result with the noise is added because the model is trained with noise. Therefore, the denoising autoencoder learns to reconstruct (denoise) the original images from noisy or corrupted images and avoids the risk of learning the identity function.

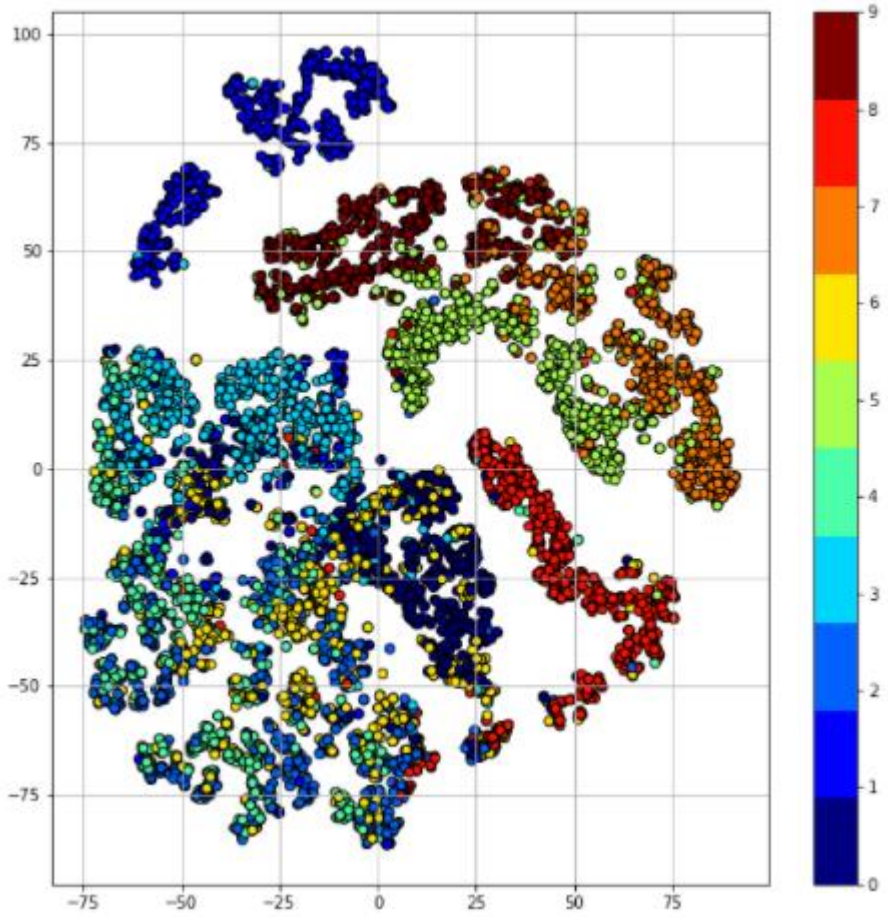
Part 4

Screenshot of the code that implements the architecture

```
dim_latent_representation = 30
class VAE(nn.Module):

    def __init__(self,dim_latent_representation=2):
        super(VAE,self).__init__()


        class Encoder(nn.Module):
            def __init__(self, output_size=2):
                super(Encoder, self).__init__()
                # needs your implementation
                self.layers = nn.Sequential(
                  nn.Linear(in_features = 784,  out_features = dim_latent_representation*2),
                  nn.Tanh()
                )


            def forward(self, x):
                # needs your implementation
                one_d_x = x.shape[0]
                encoded_x = self.layers(torch.reshape(x,(one_d_x, 784)))
                #encoded_x = self.layers(x)
                return encoded_x


        class Decoder(nn.Module):
            def __init__(self, input_size=2):
                super(Decoder, self).__init__()
                # needs your implementation
                self.layers = nn.Sequential(
                    nn.Linear(in_features = dim_latent_representation, out_features = 784), # change to 1 decoder
                    nn.Tanh()
                )


            def forward(self, z):
              # needs your implementation
              decoded_z = self.layers(z)
              one_d_z = decoded_z.shape[0]
              output_img = torch.reshape(decoded_z,(one_d_z,1,28,28))
              return output_img
              #return decoded_z

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

```
# Implement this function for the VAE model
def reparameterise(self, mu, logvar):
  if self.training:
    # return reparametrized mu
    standard_deviation = torch.exp(0.5*logvar)
    epsilon  = torch.randn_like(standard_deviation)
    reparametrized_mu = mu + (epsilon * standard_deviation)
    return reparametrized_mu
  else:
    return mu

def forward(self, x):
    # This function should be modified for the DAE and VAE
    #x = F.tanh(self.encoder.encoder1(x))
    x = self.encoder(x).view(-1, 2, dim_latent_representation)
    mu = x[:, 0, :]
    logvar = x[:, 1, :]
    re_mu = self.reparameterise(mu, logvar)
    #x = F.tanh(self.decoder.decoder1(re_mu))
    # reconstruction = torch.sigmoid(self.decoder.decoder2(x))
    reconstruction = self.decoder(re_mu)
    # for the VAE forward function should also return mu and logvar
    return reconstruction, mu, logvar
```

Screenshot of latent space plots



Describe how the latent features are differ from each other for two models and explain what this says about the variational autoencoders.

For laten space plot, the classes are separated quite well and spread out more across the space. VAE assumes the input has probability distribution. It has a layer of data means and standard deviations which are the parameters for the distribution which it is not the same as DAE and autoencoder (AE). The main purpose of VAE is to generate new data related to the original source data. Therefore, it performs well for new data which were not trained (tends to have good accuracy for testing and validation).