Assignment 3 Solutions

1 Neural Network

a) Layer 1:

$$\vec{z}_1 = W^{(0)}\vec{x} = \begin{pmatrix}0.3\\0.3\\0.3\end{pmatrix} \implies \vec{h}_1 = g(\vec{z}_1) = \begin{pmatrix}0.5744\\0.5744\\0.5744\end{pmatrix}$$

Layer 2:

$$\vec{z}_2 = W^{(1)}\vec{h}_1 = \begin{pmatrix}0.5117\\0.5117\\0.5117\end{pmatrix} \implies \vec{h}_2 = g(\vec{z}_2) = \begin{pmatrix}0.6264\\0.6264\\0.6264\end{pmatrix}$$

Layer 3 (output layer):

$$\hat{y} = W^{(2)}\vec{h}_2 = 0.9397$$

b) Step 1: Gradients calculation

$$\frac{\partial L}{\partial \hat{y}} = \hat{y} - y = -1.0603 \tag{1}$$

$$\begin{aligned}
\frac{\partial L}{\partial \vec{z}_2} &= \frac{\partial \hat{y}}{\partial \vec{z}_2}\frac{\partial L}{\partial \hat{y}} && \text{(2a)}\\
&= \frac{\partial \vec{h}_2}{\partial \vec{z}_2}\frac{\partial \hat{y}}{\partial \vec{h}_2}\frac{\partial L}{\partial \hat{y}} && \text{(2b)}\\
&= \begin{pmatrix} g'(z_{2,1}) & 0 & 0\\ 0 & g'(z_{2,2}) & 0\\ 0 & 0 & g'(z_{2,3})\end{pmatrix}(W^{(2)})^\top\frac{\partial L}{\partial \hat{y}} && \text{(2c)}\\
&= \begin{pmatrix} g'(z_{2,1})w^{(2)}_{1,1}\frac{\partial L}{\partial \hat{y}}\\ g'(z_{2,2})w^{(2)}_{1,2}\frac{\partial L}{\partial \hat{y}}\\ g'(z_{2,3})w^{(2)}_{1,3}\frac{\partial L}{\partial \hat{y}}\end{pmatrix} && \text{(2d)}\\
&= \begin{pmatrix}-0.1241\\-0.1241\\-0.1241\end{pmatrix} && \text{(2e)}
\end{aligned}$$

$$\frac{\partial L}{\partial \vec{z}_1} = \frac{\partial \vec{z}_2}{\partial \vec{z}_1}\frac{\partial L}{\partial \vec{z}_2} \tag{3a}$$

$$= \frac{\partial \vec{h}_1}{\partial \vec{z}_1}\frac{\partial \vec{z}_2}{\partial \vec{h}_1}\frac{\partial L}{\partial \vec{z}_2} \tag{3b}$$

$$= \begin{pmatrix} g'(z_{1,1}) & 0 & 0 \\ 0 & g'(z_{1,2}) & 0 \\ 0 & 0 & g'(z_{1,3}) \end{pmatrix} (W^{(1)})^\top \frac{\partial L}{\partial \vec{z}_2} \tag{3c}$$

$$= \begin{pmatrix} g'(z_{1,1}) \left(\frac{\partial L}{\partial z_{2,1}} w_{1,1}^{(1)} + \frac{\partial L}{\partial z_{2,2}} w_{2,1}^{(1)} + \frac{\partial L}{\partial z_{2,3}} w_{3,1}^{(1)}\right) \\ g'(z_{1,2}) \left(\frac{\partial L}{\partial z_{2,1}} w_{1,2}^{(1)} + \frac{\partial L}{\partial z_{2,2}} w_{2,2}^{(1)} + \frac{\partial L}{\partial z_{2,3}} w_{3,2}^{(1)}\right) \\ g'(z_{1,3}) \left(\frac{\partial L}{\partial z_{2,1}} w_{1,3}^{(1)} + \frac{\partial L}{\partial z_{2,2}} w_{2,3}^{(1)} + \frac{\partial L}{\partial z_{2,3}} w_{3,3}^{(1)}\right) \end{pmatrix} \tag{3d}$$

$$= \begin{pmatrix} -0.0273 \\ -0.0273 \\ -0.0273 \end{pmatrix} \tag{3e}$$

Step 2: Update weights

$$W^{(2)} \longleftarrow W^{(2)} - \eta(\vec{h}^{(2)})^\top \frac{\partial L}{\partial \hat{y}} = \begin{bmatrix} 1.1642 & 1.1642 & 1.1642 \end{bmatrix} \tag{4a}$$

$$W^{(1)} \longleftarrow W^{(1)} - \eta\, \vec{h}_1 \left(\frac{\partial L}{\partial \vec{z}_2}\right)^\top = \begin{bmatrix} 0.3713 & 0.3713 & 0.3713 \\ 0.3713 & 0.3713 & 0.3713 \\ 0.3713 & 0.3713 & 0.3713 \end{bmatrix} \tag{4b}$$

$$W^{(0)} \longleftarrow W^{(0)} - \eta\, \vec{x} \left(\frac{\partial L}{\partial \vec{z}_1}\right)^\top = \begin{bmatrix} 0.1273 & 0.1273 & 0.1273 \\ 0.1273 & 0.1273 & 0.1273 \\ 0.1273 & 0.1273 & 0.1273 \end{bmatrix} \tag{4c}$$

Note that we could have also computed the partial derivatives with respect to each component of the weight matrices, or with respect to each row of the weight matrices, to get the same result.

2 Autoencoders

2.1 part a

The attached code block shows how we implemented the Vanilla autoencoder including Encoder, Decoder, and forward passes. We also modified *Train* and *Validate* function in the .py file so as to return loss values for plotting loss curves. Although it is not required for students to implement/report, it helps to interpret results.

```
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.nn = nn.Sequential(
                nn.Linear(28 * 28, output_size),
                )

            def forward(self, x):
                return self.nn(x)

        class Decoder(nn.Module):
            def __init__(self, input_size=2):
                super(Decoder, self).__init__()

                self.nn = nn.Sequential(
                nn.Linear(input_size, 28 * 28),
                nn.Sigmoid(),
                )

            def forward(self, z):
                return self.nn(z)

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

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

Listing 1: Part (a) architecture

Results from this autoencoder are as follows:

- Epoch: 20
- Training loss: 0.7271
- Validation loss (reconstruction error) : 0.7330

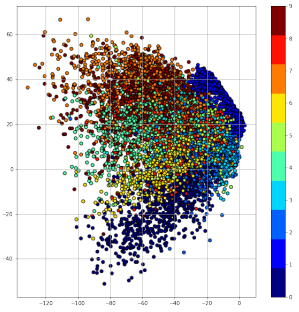
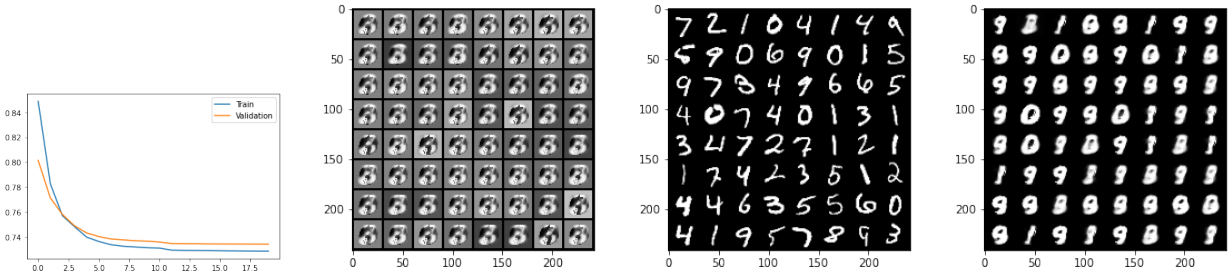


Figure 1: Scatter plot of the latent representation (Part a)



(a) Loss curve (b) Bottleneck samples (c) Input (d) Output

Figure 2: Results from part a

2.2 part b

```python
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.nn = nn.Sequential(
                nn.Linear(28 * 28, 1024),
                nn.ReLU(),
                nn.Linear(1024, output_size)

                )

            def forward(self, x):
                return self.nn(x)

        class Decoder(nn.Module):
            def __init__(self, input_size=2):
                super(Decoder, self).__init__()

                self.nn = nn.Sequential(
                nn.Linear(input_size, 1024),
                nn.ReLU(),
                nn.Linear(1024, 28 * 28),
                )

            def forward(self, z):
                return self.nn(z)

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


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

Listing 2: Part (b) architecture

- Epoch: 20
- Training loss: 0.6621
- Validation loss (reconstruction error) : 0.6717

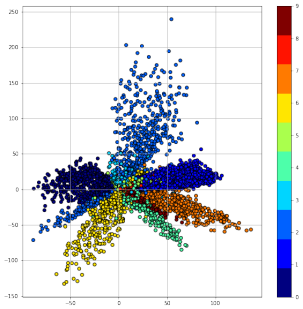
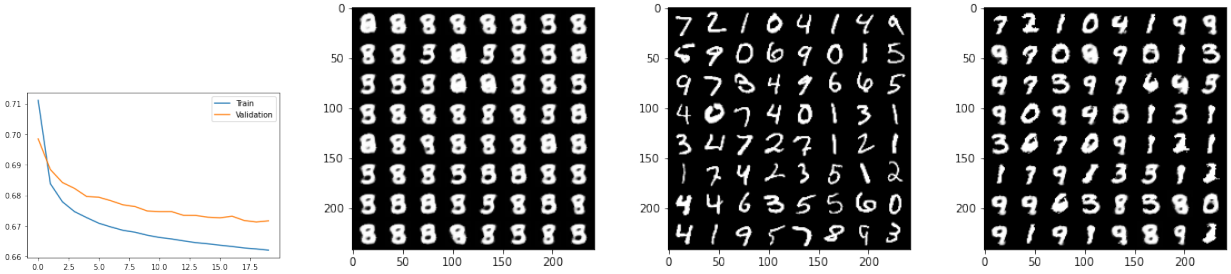


Figure 3: Scatter plot of the latent representation (Part b)



(a) Loss curve (b) Bottleneck samples (c) Input (d) Output

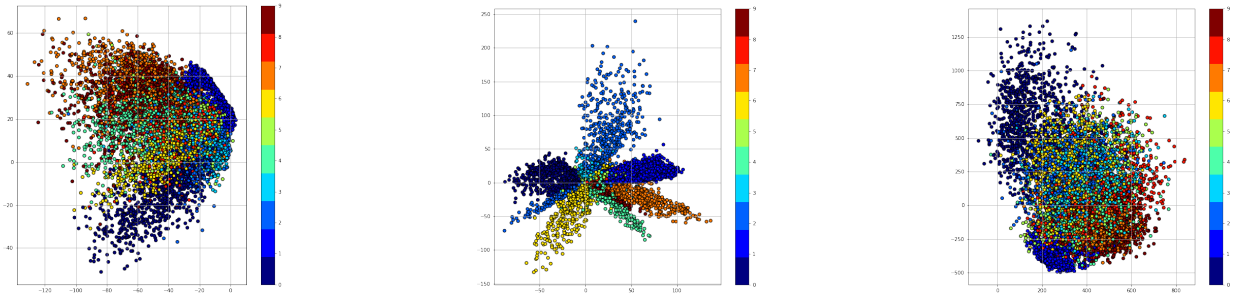
Figure 4: Results from part b

Now we compare two factors we changed here compared to the previous part. In part (a), we see undercomplete autoencoder since the 2D hidden layer is too smaller than the input (784). Using too few hidden units will result in underfitting. This constraint will also impose the model to lose part of the data information during the compression process. However, additional layers can learn more expressive model with complex representations. This can be concluded from the reconstruction loss values as well as the reconstructed sample images.

As we can see in the scatter plot, although some of the labels are encoded with similar values, this model is providing more distinctive code to the decoder than the previous model. In fact, any neural network with one fully connected layer can be used to represent linearly separable functions while adding an activation function to the encoder enables the network to capture non-linearity.

For better understanding of contribution of each change (not required for students), we also trained the same model, with 1024 hidden units and no *ReLU* activation function so called ablation study. In the following, we included a side-by-side scatter plot of these three models. Training this model is beneficial since it has only one changed factor compared to part a and part b.

```
Autoencoder(
  (encoder): Encoder(
    (nn): Sequential(
      (0): Linear(in_features=784, out_features=1024, bias=True)
      (1): Linear(in_features=1024, out_features=2, bias=True)
    )
  )
  (decoder): Decoder(
    (nn): Sequential(
      (0): Linear(in_features=2, out_features=1024, bias=True)
      (1): Linear(in_features=1024, out_features=784, bias=True)
      (2): Sigmoid()
    )
```



(a) part a (b) part b (c) part b w/o *ReLU*

Figure 5: Latent space scatter plot for part a, part b, and part b without ReLU activation function

```
  )
)
```

Listing 3: Part (b) architecture without ReLU activation

Note that we know that another advantage of ReLU could be avoiding vanishing gradients. However, since we use a shallow network, this answer would not be acceptable.

2.3 part c

In the following we provided the implementation and results for part c.

```
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.nn = nn.Sequential(
                nn.Linear(28*28, 1024),
                nn.Sigmoid(),
                nn.Linear(1024, output_size)
                )


            def forward(self, x):
                return self.nn(x)


        class Decoder(nn.Module):
            def __init__(self, input_size=2):
                super(Decoder, self).__init__()

                self.nn = nn.Sequential(
                nn.Linear(input_size, 1024),
                nn.Sigmoid(),
                nn.Linear(1024, 28*28),
                nn.Sigmoid(),

                )


            def forward(self, z):
                return self.nn(z)


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


    def forward(self,x):
        x = self.encoder(x)
        x = self.decoder(x)
```

```
46         return x
```

Listing 4: Part (b) architecture without ReLU activation

- Epoch: 20
- Training loss: 0.6655
- Validation loss (reconstruction error) : 0.6770

We plotted the activation functions in fig. 6. Regarding fig. 3 and fig. 7, we can immediately observe that the benefit of ReLUs is sparsity. Sparsity arises when $input \leq 0$. The more units with negative values exist in a layer the more sparse the resulting representation. Hence, if certain neurons are less important in terms of their weights, they will be removed, and the model is sparse. On the other hand, Sigmoids shape a dense representation since they are likely to generate some non-zero value for most of the input values.

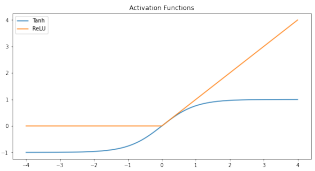


Figure 6: RelU and Tanh activation functions.

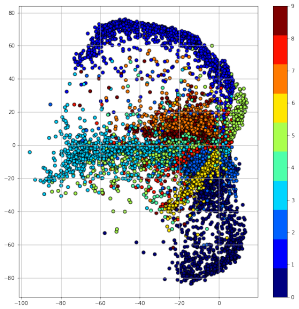
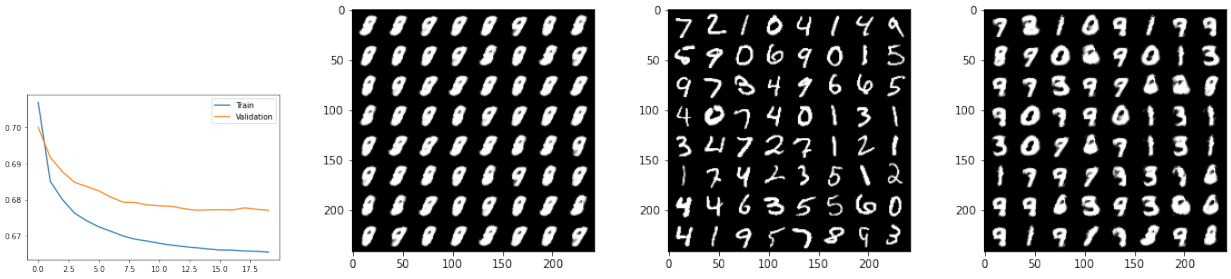


Figure 7: Scatter plot of the latent representation (Part c)



(a) Loss curve (b) Bottleneck samples (c) Input (d) Output

Figure 8: Results from part c

2.4 Part d

First, we implement a vanilla autoencoder similar to the part (a) with 30 dimensions as the bottleneck size and replace *Sigmoid* to *Tanh*. The following code block shows the architecture of this step (not the implementation).

```
Autoencoder(
  (encoder): Encoder(
    (nn): Sequential(
      (0): Linear(in_features=784, out_features=30, bias=True)
    )
  )
  (decoder): Decoder(
    (nn): Sequential(
      (0): Linear(in_features=30, out_features=784, bias=True)
      (1): Tanh()
    )
  )
)
```

Listing 5: Part (d) vanilla autoencoder architecture.

- Epoch: 20
- Training loss: 0.4089
- Validation loss (reconstruction error) : 0.4171

Afterward, we implemented the VAE nodel as shown in the code block 6. We trained this model with different KLD weights from the set 0.5, 1, 5 and loss values were as follows. Also, Fig. 9, 10, 11, and 12 show the latent space in 2D space, unseen generated samples, and reconstructed inputs for all trained models in part d.

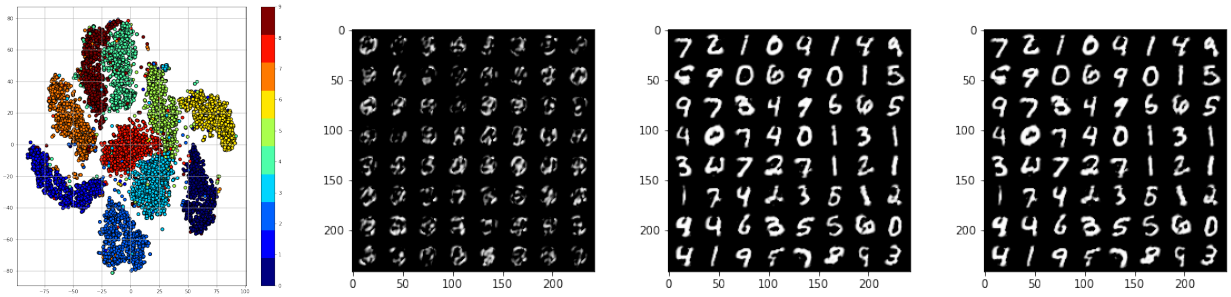
```

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.nn = nn.Sequential(
                nn.Linear(28 * 28, output_size),
                )

            def forward(self, x):
                # needs your implementation
                return self.nn(x)

        class Decoder(nn.Module):
            def __init__(self, input_size=2):
                super(Decoder, self).__init__()
                # needs your implementation
                self.nn = nn.Sequential(
                nn.Linear(input_size, 28 * 28),
                nn.Tanh(),
                )

            def forward(self, z):
                # needs your implementation
                return self.nn(z)

        self.dim_latent_representation = dim_latent_representation
        self.encoder = Encoder(output_size=dim_latent_representation)
        self.mu_layer = nn.Linear(self.dim_latent_representation, self.dim_latent_representation)
        self.logvar_layer = nn.Linear(self.dim_latent_representation, self.dim_latent_representation)
        self.decoder = Decoder(input_size=dim_latent_representation)
    # Implement this function for the VAE model
    def reparameterise(self, mu, logvar):

        if self.training:
            std = logvar.mul(0.5).exp_()
```

```
            eps = std.data.new(std.size()).normal_()
            return eps.mul(std).add_(mu)
        else:
            return mu

    def forward(self,x):

        # This function should be modified for the DAE and VAE
        x = self.encoder(x)
        mu, logvar = self.mu_layer(x), self.logvar_layer(x)
        z = self.reparameterise(mu, logvar)
        return self.decoder(z), mu, logvar
```

Listing 6: Part (d) vanilla autoencoder architecture.

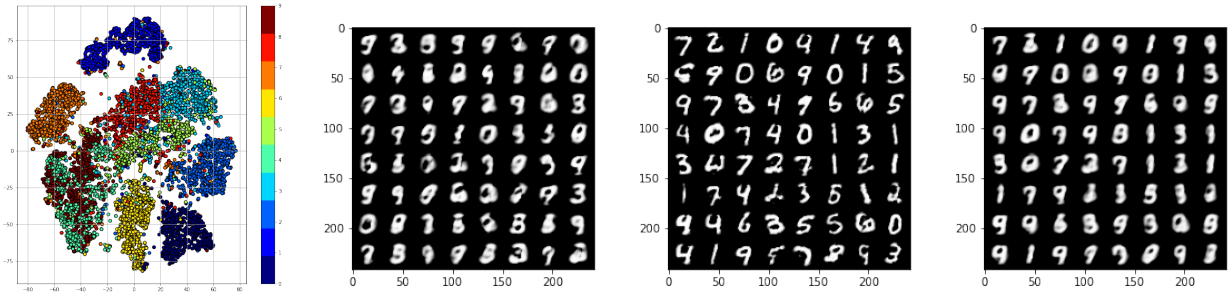
- Loss = MSE + 0.5 * KLD
  - Epoch: 20
  - Training loss: 0.6420
  - Validation loss (reconstruction error) : 0.6060
- Loss = MSE + 1 * KLD
  - Epoch: 20
  - Training loss: 0.6821
  - Validation loss (reconstruction error) : 0.6550
- Loss = MSE + 5 * KLD
  - Epoch: 20
  - Training loss: 0.7122
  - Validation loss (reconstruction error) : 0.7154

We observe a low loss value for the vanilla autoencoder with 30D bottleneck size which results in high quality reconstructed images. Although loss values increased in VAEs, the VAE arranged the latent space such that gaps between latent representations for different classes decreased. It means we can get better manipulated (mixed latents) output. Since VAE follows an isotropic multivariate normal distribution at the latent space, we can generate new unseen images by taking samples from the latent space with higher quality compared to the Vanilla autoencoder. However, the reconstruction quality reduced (loss values increased) since loss function is a weighted combination of MSE and KLD terms to be optimized where the KLD term forces the latent space to resembles a Gaussian distribution. As we increased the KLD weight, we achieved a more compact latent space closer to the prior distribution by sacrificing the reconstruction quality.



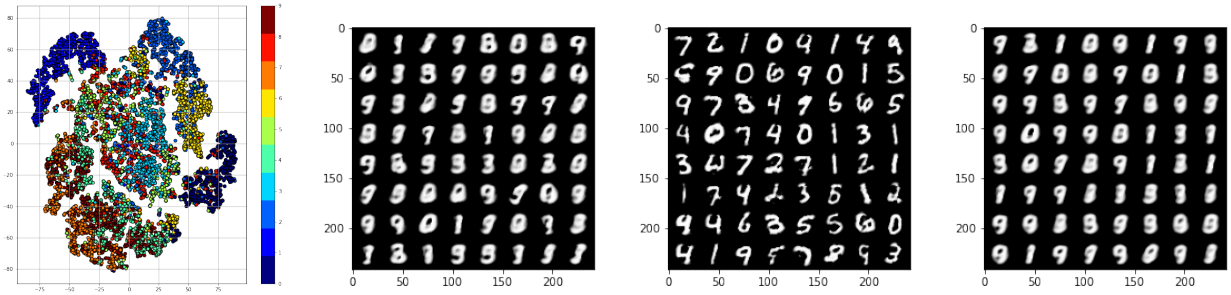
(a) Latemt space (b) Bottleneck samples (c) Input (d) Output

Figure 9: Results from part d Vanilla Autoencoder



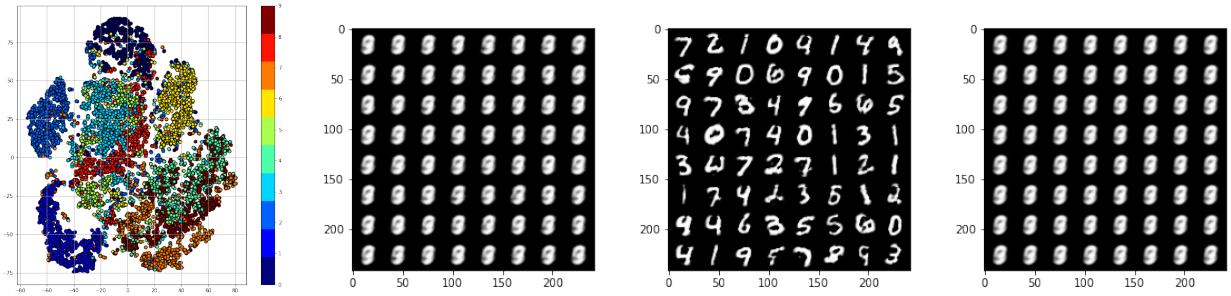
(a) Latemt space (b) Bottleneck samples (c) Input (d) Output

Figure 10: Results from part d Variational Autoencoder, KLD weight = 0.5



(a) Latemt space (b) Bottleneck samples (c) Input (d) Output

Figure 11: Results from part d Variational Autoencoder, KLD weight = 1



(a) Latemt space (b) Bottleneck samples (c) Input (d) Output

Figure 12: Results from part d Variational Autoencoder, KLD weight = 5

3 Support Vector Machines

a) We are trying to maximize $L(\vec{\lambda}) = \sum_{i=1}^{N} \lambda_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \lambda_i \lambda_j y_i y_j \phi(\vec{x}_j)^\top \phi(\vec{x}_i)$

Focusing on $\sum_{i=1}^{N} \sum_{j=1}^{N} \lambda_i \lambda_j y_i y_j \phi(\vec{x}_j)^\top \phi(\vec{x}_i)$

If $i = j$, we have $9\lambda_i^2$

If $i \neq j$, we have $\lambda_i \lambda_j (y_i)(y_j)$

Therefore

$$\sum_{i=1}^{N} \lambda_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \lambda_i \lambda_j y_i y_j \phi(\vec{x}_j)^\top \phi(\vec{x}_i) =$$

$$\lambda_1 + \lambda_2 + \lambda_3 + \lambda_4$$

$$- \frac{1}{2} \left(9\lambda_1^2 - 2\lambda_1\lambda_2 - 2\lambda_1\lambda_3 + 2\lambda_1\lambda_4 + 9\lambda_2^2 + 2\lambda_2\lambda_3 - 2\lambda_2\lambda_4 + 9\lambda_3^2 - 2\lambda_3\lambda_4 + 9\lambda_4^2\right) \quad (5)$$

For each λ_i, we need to maximize the following objective:

$$L_1(\lambda_1) = \lambda_1 - \frac{1}{2} \left(9\lambda_1^2 - 2\lambda_1\lambda_2 - 2\lambda_1\lambda_3 + 2\lambda_1\lambda_4\right) \quad (6)$$

$$L_2(\lambda_2) = \lambda_2 - \frac{1}{2} \left(9\lambda_2^2 - 2\lambda_2\lambda_1 + 2\lambda_2\lambda_3 - 2\lambda_2\lambda_4\right) \quad (7)$$

$$L_3(\lambda_3) = \lambda_3 - \frac{1}{2} \left(9\lambda_3^2 - 2\lambda_3\lambda_1 + 2\lambda_3\lambda_2 - 2\lambda_3\lambda_4\right) \quad (8)$$

$$L_4(\lambda_4) = \lambda_4 - \frac{1}{2} \left(9\lambda_4^2 + 2\lambda_4\lambda_1 - 2\lambda_4\lambda_2 - 2\lambda_4\lambda_3\right) \quad (9)$$

Since each $L_i(\lambda_i)$ is concave, finding λ_i such that $\frac{\partial L_i}{\partial \lambda_i} = 0$ for each i will maximize the dual of SVM. Solving the following linear equation:

$$\begin{bmatrix} 9 & -1 & -1 & 1 \\ -1 & 9 & 1 & -1 \\ -1 & 1 & 9 & -1 \\ 1 & -1 & -1 & 9 \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \\ \lambda_4 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

gives: $\lambda_1 = \lambda_2 = \lambda_3 = \lambda_4 = \frac{1}{8}$

Note that $\sum_{i=1}^{4} \lambda_i y_i = 0$ so $\vec{\lambda}$ satisfies the constrained dual form.

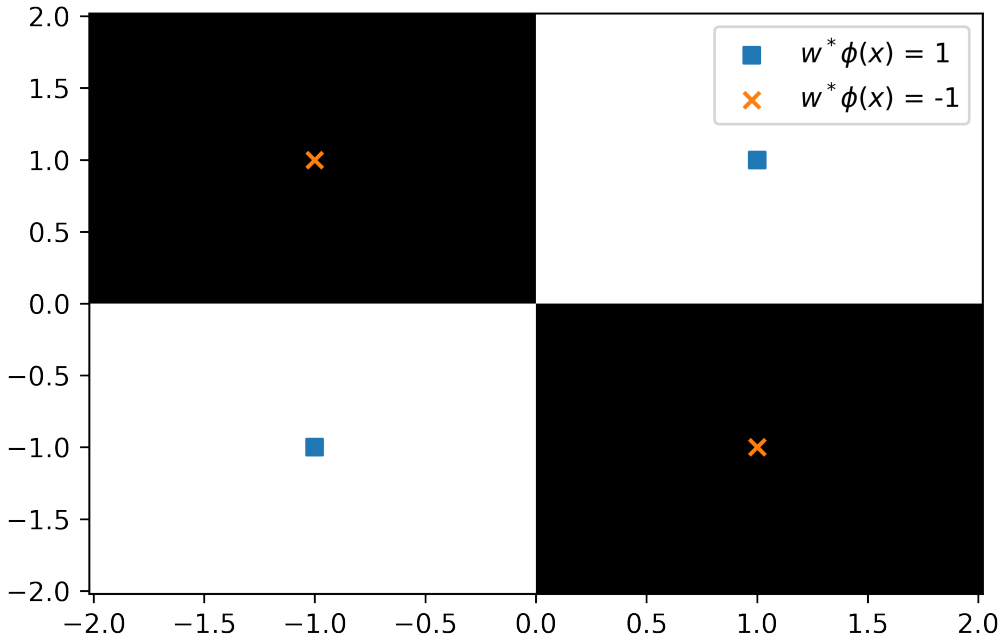
b)

$$\vec{w}^* = \sum_{i=1}^{N} \lambda_i^* y_i \phi(\vec{x}_i) = \frac{1}{8} \left[- \begin{bmatrix} 1 \\ -\sqrt{2} \\ -\sqrt{2} \\ 1 \\ \sqrt{2} \\ 1 \end{bmatrix} + \begin{bmatrix} 1 \\ -\sqrt{2} \\ \sqrt{2} \\ 1 \\ -\sqrt{2} \\ 1 \end{bmatrix} + \begin{bmatrix} 1 \\ \sqrt{2} \\ -\sqrt{2} \\ 1 \\ -\sqrt{2} \\ 1 \end{bmatrix} - \begin{bmatrix} 1 \\ \sqrt{2} \\ \sqrt{2} \\ 1 \\ \sqrt{2} \\ 1 \end{bmatrix} \right] = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ -\frac{\sqrt{2}}{2} \\ 0 \end{bmatrix}$$

$$b^* = \frac{\max\{-1,-1\} + \min\{1,1\}}{2} = 0$$

c) The decision boundary are given at ${w^*}^\top \phi(\vec{x}) - b^* = 0$. From b), this is when $-x_1 x_2 = 0$; equivalently, $x_1 = 0$ or $x_2 = 0$

The support vectors are when ${w^*}^\top \phi(\vec{x}) - b^* = \pm 1$. From b), this is when $x_2 = -\frac{1}{x_1}$ or $x_2 = \frac{1}{x_1}$. All of the vectors are support vectors.



d) Plot 1 corresponds to when $\gamma = 5$ and Plot 2 corresponds to when $\gamma = 0.05$. In Plot 1 less support vectors are used compared to Plot 2. Large γ penalizes when points are misclassified

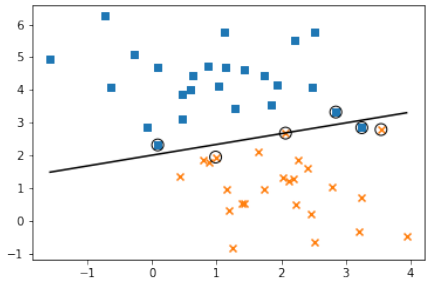


Figure 13: Plot 1

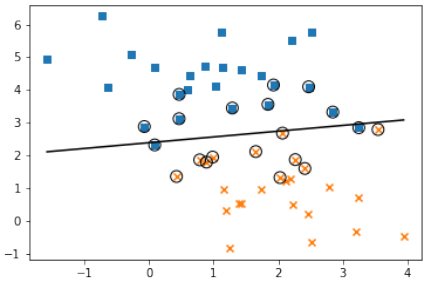


Figure 14: Plot 2

and makes the margin small since SVM would want the least number of misclassification mistakes. Small γ does the opposite so there are more support vectors in Plot 2 in comparison to Plot 1.