**Assignment 4: Support Vector Machines /Autoencoders**

**Due December 7 at 11:59pm**

**This assignment is to be done individually.**

---

**Important Note:** The university policy on academic dishonesty (cheating) will be taken very seriously in this course. You may not provide or use any solution, in whole or in part, to or by another student.

You are encouraged to discuss the concepts involved in the questions with other students. If you are in doubt as to what constitutes acceptable discussion, please ask! Further, please take advantage of office hours offered by the instructor and the TA if you are having difficulties with this assignment.

**DO NOT**:

- Give/receive code or proofs to/from other students
- Use Google to find solutions for assignment

**DO**:

- Meet with other students to discuss assignment (it is best not to take any notes during such meetings, and to re-work assignment on your own)
- Use online resources (e.g. Wikipedia) to understand the concepts needed to solve the assignment.

---

**Submitting Your Assignment**

The assignment must be submitted on Canvas. You must submit one zip file (student number a3.zip) containing:

1. An assignment report in PDF format, named `report.pdf`. This report should contain your solutions to questions 1-2.
2. Your code for question 2 named `code.zip` which contains
   - `autoencoder_starter_2_x.py`: codes for part x of question 2.
   - `Autoencoder_sample.ipynb`: Jupyter Notebook that you used for training and generating results for different parts.

---

## 1 SVM

Support Vector Machines can be used to perform non-linear classification with a kernel trick. Recall the hard-margin SVM from class:

$$min\frac{1}{2}w^T w$$
$$s.t.y^{(i)}(w^T x^{(i)} + b) \geq 1$$

The dual of this primal problem can be specified as a procedure to learn the following linear classifier:

$$f(x) = \sum_{i}^{N} \alpha_i y_i (x_i^T x) + b$$

Note that now we can replace $x_i^T x$ with a kernel $k(x_i, x)$, and have a non-linear decision boundary. In Figure 1, there are different SVMs with different shapes/patterns of decision boundaries. The training data is labeled as $yi \in \{1, 1\}$, represented as the shape of circles and squares respectively. Support vectors are drawn in solid circles. Match the scenarios described below to one of the 6 plots (note that one of the plots does not match to anything). Each scenario should be matched to a unique plot. Explain in less than two sentences why it is the case for each scenario.

1. A soft-margin linear SVM with $C = 0.02$.
2. A soft-margin linear SVM with $C = 20$.
3. A hard-margin kernel SVM with $k(u, v) = u^T v + (u^T v)^2$
4. A hard-margin kernel SVM with $k(u, v) = exp(-5\|u - v\|_2^2)$
5. A hard-margin kernel SVM with $k(u, v) = exp(-\frac{1}{5}\|u - v\|_2^2)$

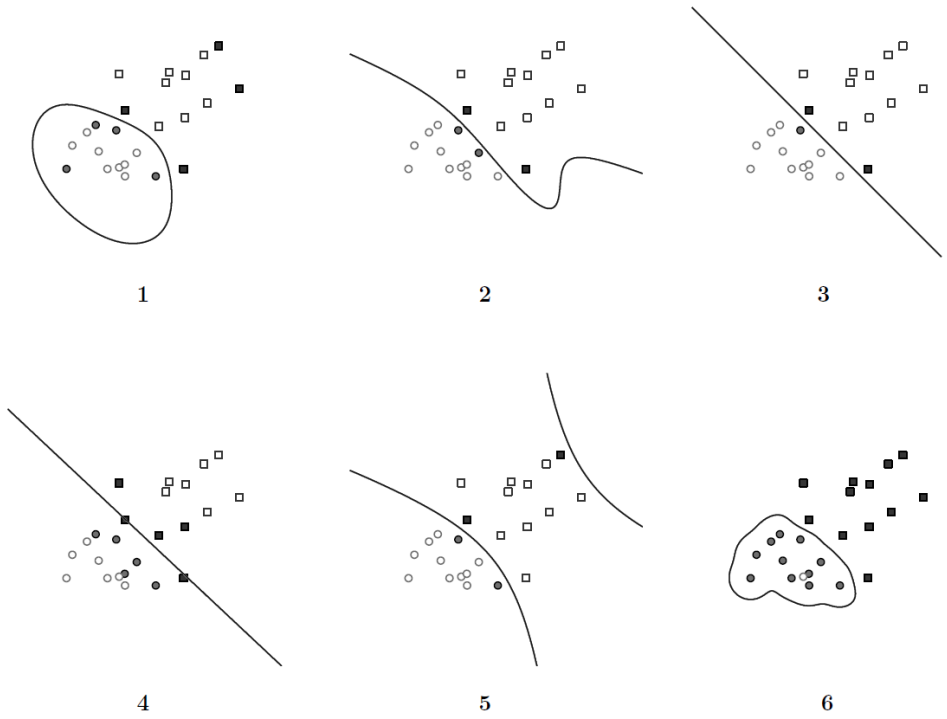



Figure 1: SVM boundaries

---

1. Soft margin *Linear* SVM with hyper-parameter C = 0.02 is the one in the **Picture 4**. This is because the general rule of thumb is that when C is very low the number of mistakes allowed by the SVM is high and in this example the value of C is very low and we can see from the figure that there have been some mistakes in the $4^{th}$ plot.

2. Following the same logic as in the previous part, the soft margin *Linear* SVM is the one with C equal to 20 is represented by the **Picture 3**. Formula: $L = \frac{1}{2}\|w\|^2 + C(\#ofmistakes)$. When C is high the number of mistakes allowed is very low, which means that the classifier tries to minimize the training loss.

3. This is a quadratic kernel and the decision function is given by $f(x) = \sum_i^N \alpha_i(x_i^T x + (x_i^T x)^2) + b$ and a quadratic function of $x$ will yield a hyperbola or ellipse, it is a hyperbola in this case, i.e., **Plot 5** is the correct decision boundary for this quadratic kernel.

4. This kernel is an example of Radial Basis Function and for this type of kernel the bigger the value of $\gamma$ the more support vectors there are and support vectors in this Figure are denoted by solid black circles/squares. Hence the **Plot 6** matches this decision boundary the best.

5. Using the similar argument as Part 4, the lower the value of $\gamma$ the lower is the number of support vectors used, hence **Picture 1** represents this kernel.

## 2 Autoencoders

Autoencoders are an unsupervised learning technique in which we leverage neural networks for the task of representation learning. Specifically, we impose a bottleneck in the network, which forces a compressed knowledge (in lower dimensionality) representation of the original input. The autoencoder captures significant data features and discards spurious patterns by minimizing the reconstruction error. Autoencoders have two components: (1) an encoder function $f$ that converts inputs x to a latent variable $z := f(x)$ with lower dimensions. (2) a decoder function $g$ that produces a reconstruction of the inputs as $\hat{y} = g(z)$ from the bottleneck feature representation $z$. The overall neural network is $g \circ f$, which takes an input $x$ and produces a reconstruction $\hat{y}$ as follows.

$$\hat{y} = g(f(x)) \tag{1}$$

Generally, $f$ and $g$ can be any type of network including convolutional Neural Network (CNN), Multi Layer Perceptron (MLP), Recurrent Neural Network (RNN), etc. In the simplest case, both $f$ and $g$ are MLP; in this case, the overall neural network, $g \circ f$, is an MLP as well. The primary difference between a typical MLP and an autoencoder is that an autoencoder is unsupervised, therefore, it can be trained without labels. The objective function we use to train the autoencoder is the same as that of multi-output nonlinear regression, namely Mean Square Error (MSE) loss between the reconstruction $y$ and the input $x$.

An interesting capability of autoencoders is that they can generate new data by manipulating the latent feature representation $z$. In this question we are going to explore different autoencoder architectures as well as manipulation. We will work on the Fashion-MNIST dataset (28x28 grayscale images of ten classes including Dress, Coat, Sandal, Shirt, etc.) In the starter code, we provide the routines for data loading, training loop and visualization in `autoencoder_starter.py`. We also provide a Jupyter notebook `autoencoder_sample.ipynb`, where we demonstrate how to train the model and use some essential visualization functions.

1. Implement an autoencoder where the encoder and the decoder are linear models (i.e. they consist of no hidden layer and one fully-connected / dense output layer (known as a linear layer in PyTorch). The bottleneck feature representation $z$ should be two-dimensional. The decoders output should be transformed by a sigmoid function, so that the output lies within the range [0; 1].

   Implement the architecture of the encoder and decoder and print out the reconstruction losses on the train and validation sets. Also use the scatter plot function in `autoencoder_starter.py` to plot the 2D bottleneck feature representation as a scatter plot, where different classes are represented by dots of different colours. Include a screenshot of your code that implements the architecture and the generated scatter plot in your PDF submission.

   **Hints:**

   - Take a look at `Autoencoder_sample.ipynb`, part (1) is partially already implemented, but you will have to find good values for several hyperparameters like the learning rate and the number of epochs to train for.

- The training and testing routines are already provided, and they should print out the reconstruction error.
- Flatten the image into a one-dimension vector before feeding it into the encoder, and reshape the output of the decoder back into an image as the last step (use reshape / view functions in PyTorch)

2. Starting from the model from part (1), add one fully-connected / dense layer with 1024 hidden units and ReLU activations to both the encoder and the decoder, while keeping the bottleneck feature representation 2-dimensional.

   Specifically, the encoder should consist of two layers:

   - 1st layer: a fully-connected / dense layer with 1024 hidden units and ReLU activation function. It should map a 784-dimensional input to a 1024-dimensional vector, where $784 = 28 \times 28 \times 1$ is the number of dimensions for grayscale images in the Fashion-MNIST dataset.
   - 2nd layer: a fully-connected / dense layer, known as a linear layer in PyTorch, with 2 output units and no activation function. It should map a 1024-dimensional vector to a 2-dimensional bottleneck feature vector.

   The decoder should have a similar architecture as the encoder, but the layers should be in reverse order.

   - 1st layer: a fully-connected / dense layer with 1024 hidden units and ReLU activation function. It should map 2-dimensional bottleneck features to a 1024-dimensional vector.
   - 2nd layer: a fully-connected / dense layer with 784 output units and sigmoid activation function. It should map a 1024-dimensional to a 784-dimensional vector, which can be reshaped into $28 \times 28 \times 1$ that is interpreted as an image.

   Print out the reconstruction losses on the train and validation sets and generate a scatter plot of the same form as in part (1). Describe how the plot differs from the one in part (1) and explain what this says about the architectures in this part and part (1). Why do you think the architecture in this part gave rise to the results shown in the scatter plot? Include a screenshot of the code that implements the architecture and the generated scatter plot in the PDF submission.

3. Autoencoders have the risk of learning the identity function, meaning that the output simply equals the input, making the Autoencoder useless. Denoising Autoencoders (DAE) solve this problem by corrupting the data on purpose at the input. Note that the MSE loss is calculated between a generated sample and the original input (not the corrupted one). Therefore, the DAE learns to reconstruct (denoise) the original images from noisy/corrupted images. Starting from the model from part (1), implement an autoencoder with bottleneck size of $30$ dimensions and $tanh$ activation function. Afterward, change the autoencoder to a Denoising Autoencoder with two different noises, turning some of input values to zero, and adding Gaussian noise. Print out the reconstruction losses on the train and validation sets for three

networks. Also, using $plt.imshow$ function, plot the kernel of each models (weights of the first layer) as an image (implement it inside the $plot\_kernel$ function). Describe how the kernels differ from each other and explain what this says about the denoising autoencoder. Include a screenshot of the code that implements the architecture and the generated kernel plots in the PDF submission.

- create a copy of $Autoencoder$ class in the Notebook and name it $DAE$
- implement a function named $add\_noise$ that adds tow type of noises
- use the $add\_noise$ function in the $forward$ function before feeding the input to the encoder
- implement the $plot\_kernel$ function in `autoencoder_starter.py`

4. A Variational Autoencoder (VAE) is an autoencoder whose encodings distribution is regularised during the training by adding a KullbackLeibler divergence ($D_{KL}$) term to the loss function. Indeed, VAE encodes an input as a distribution over the latent space. In this setup, the VAE reconstruct the output by decoding a point sampled from the latent space distribution.

   Implement an autoencoder with bottleneck size of $30$ dimensions and $tanh$ activation function. Afterward, change the autoencoder to a Variational Autoencoder by adding $D_{KL}$ term to the loss function. Print out the reconstruction losses on the train and validation sets for two networks. Use scatter plot function to visualize the latent space for two networks. You may use a TSNE or UMAP transformation to visualize 30 dimensional vectors in 2D scatter. Describe how the latent features are different from each other for two models and explain what this says about the variational autoencoders. Include a screenshot of the code that implements the architecture and the latent space plots in the PDF submission.

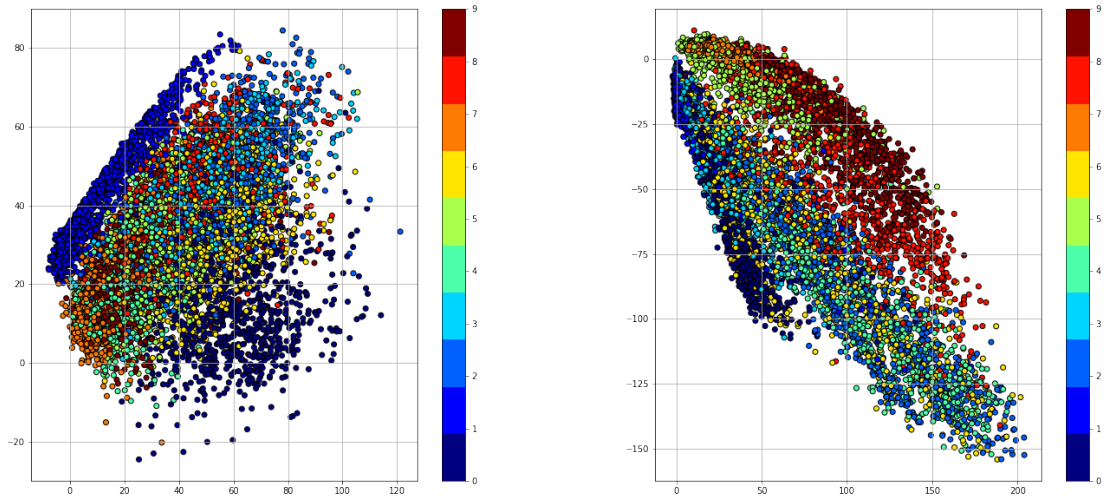
   - create a copy of $Autoencoder$ class in the Notebook and name it $VAE$
   - create a copy of `autoencoder_starter.py` and name it `VAE_starter.py`
   - Rename class $Autoencoder\_Trainer$ class to $VAE\_Trainer$
   - Modify the $loss\_function$ for this question that considers the $D_{KL}$ term too.
   - Implement the $reparametrise$ function in the $VAE$ class in the notebook
   - modify trainer function inside the `VAE_starter.py` that is compatible with the $VAE$ $forward$ function

### 2.1 Autoencoder without any Hidden Layer

**Code**

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

            def forward(self, x):
                x_flat=torch.reshape(x, (x.shape[0], 28*28))
                return self.fc(x_flat)

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

            def forward(self, z):
                z_post_act = self.sigmoid(self.out(z))
                # FashionMNIST contains images of dimension
                   28*28 with 1 channel
                z_img=torch.reshape(z_post_act,
                   (z_post_act.shape[0], 1, 28, 28))
                return z_img

        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
```

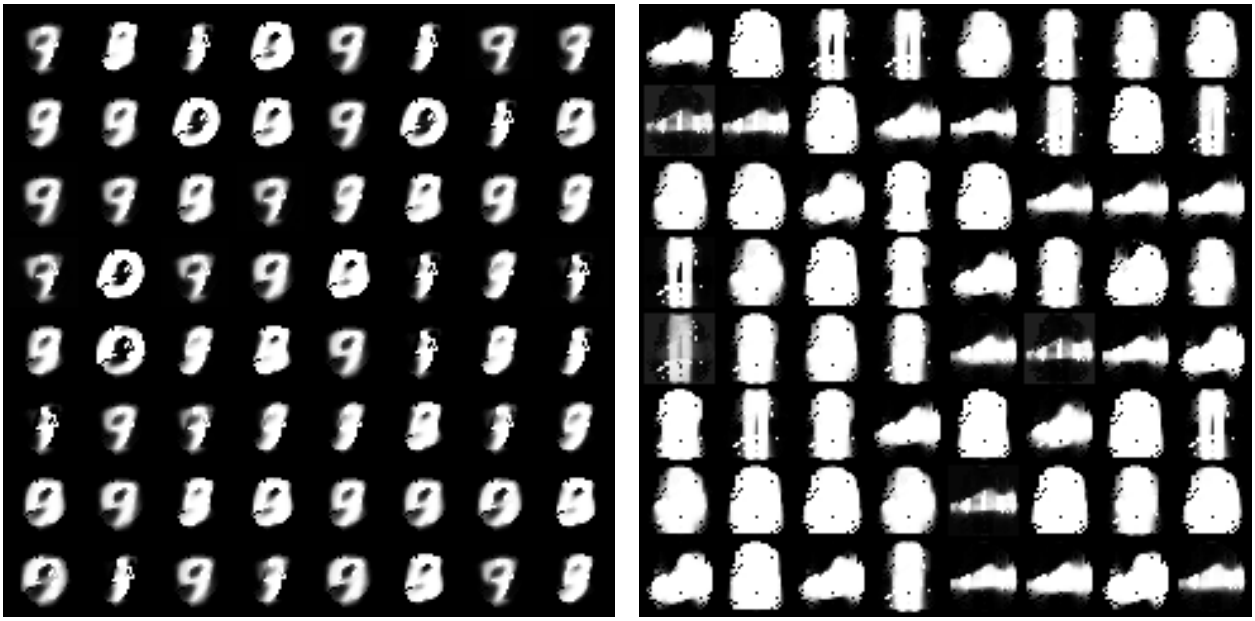
**Visualizations**



(a) Scatter Plot for MNIST

(b) Scatter Plot for FashionMNIST

Figure 2: Scatter Plots for Autoencoder with bottleneck feature representation of 2



(a) Reconstructed Images for MNIST

(b) Reconstructed Images for FashionMNIST

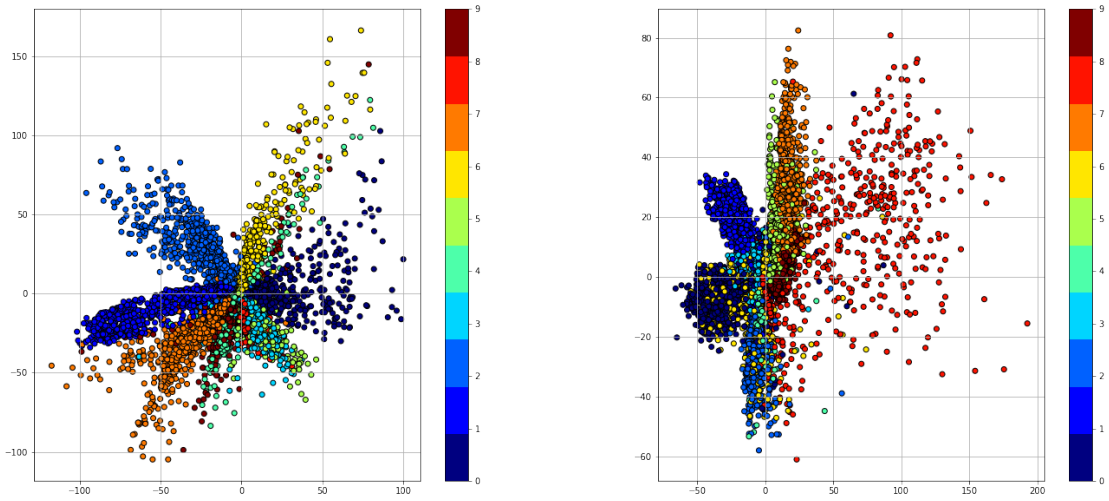
Figure 3: Reconstructed Images for Autoencoder with bottleneck feature representation of 2

### 2.2 Autoencoder with a Hidden Layer

**Code**

```
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.hidden_layer = nn.Linear(28*28, 1024)
                self.relu = nn.ReLU()
                self.fc = nn.Linear(1024, output_size)
            def forward(self, x):
                x_flat=torch.reshape(x, (x.shape[0], 28*28))
                hid_out = self.relu(self.hidden_layer(x_flat))
                return self.fc(hid_out)

        class Decoder(nn.Module):
            def __init__(self, input_size=2):
                super(Decoder, self).__init__()
                self.hidden_layer = nn.Linear(input_size, 1024)
                self.relu = nn.ReLU()
                self.out = nn.Linear(1024, 28*28)
                self.sigmoid = nn.Sigmoid()
            def forward(self, z):
                hid_out = self.relu(self.hidden_layer(z))
                out = self.sigmoid(self.out(hid_out))
                # FashionMNIST contains images of dimension
                   28*28 with 1 channel
                img=torch.reshape(out, (out.shape[0], 1, 28, 28))
                return img

        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
```

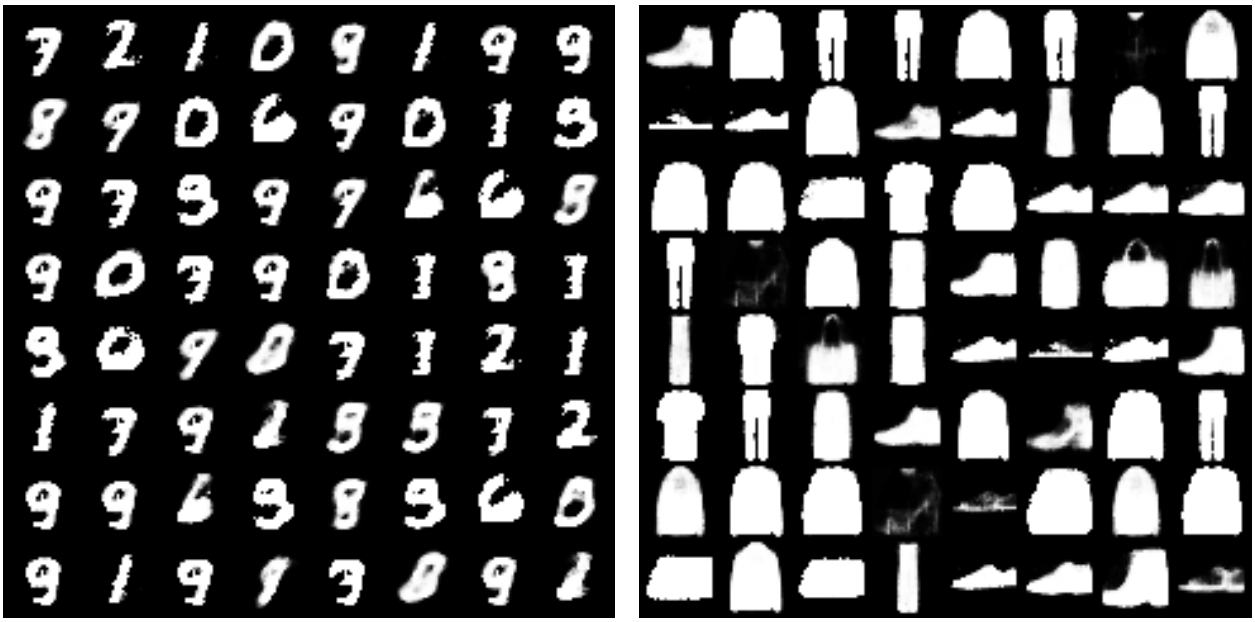
**Visualizations**



(a) Scatter Plot for MNIST

(b) Scatter Plot for FashionMNIST

Figure 4: Scatter Plots for Autoencoder with bottleneck feature representation of 2 (Hidden units = 1024, Activation Function = ReLU)



(a) Reconstructed Images for MNIST (b) Reconstructed Images for FashionMNIST

Figure 5: Reconstructed Images for Autoencoder with bottleneck feature representation of 2 (Hidden units = 1024, Activation Function = ReLU)

**Description**

The Scatter Plots from the previous part show that the first model (Figure 2) isn't able to distinguish between the 10 classes in the dataset, this is because the latent dimension is 2 and there is no hidden layer (model complexity is not sufficient). Whereas, in the Second model (Figure 4) the model is able to somewhat (there is still not a lot of separation among the classes in the plot) distinguish between the classes because of the $1024$ hidden units in the hidden layer (with ReLU activation).

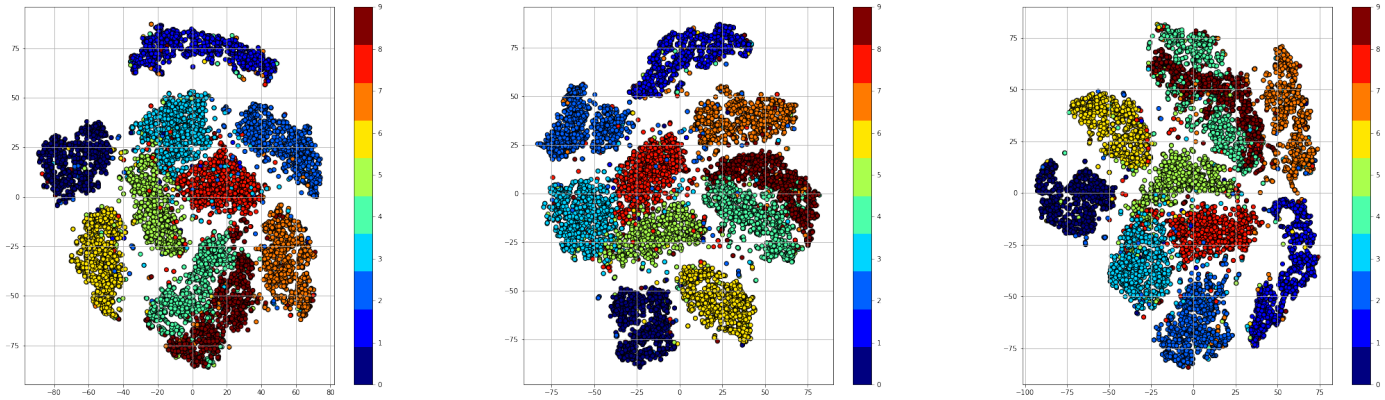
### 2.3 Denoising Autoencoders (DAE)

**Code**

```
class DAE(nn.Module):
    def __init__(self,dim_latent_representation=30,
       noise_type=None):
        super(DAE,self).__init__()

        class Encoder(nn.Module):
            def __init__(self, output_size=30):
                super(Encoder, self).__init__()
                self.fc = nn.Linear(28*28, output_size)
                self.tanh = nn.Tanh()

            def forward(self, x):
                x_flat=torch.reshape(x, (x.shape[0], 28*28))
                return self.tanh(self.fc(x_flat))

        class Decoder(nn.Module):
            def __init__(self, input_size=30):
                super(Decoder, self).__init__()
                self.out = nn.Linear(input_size, 28*28)
                self.sigmoid = nn.Sigmoid()

            def forward(self, z):
                z_post_act = self.sigmoid(self.out(z))
                # FashionMNIST contains images of dimension
                   28*28 with 1 channel
                z_img=torch.reshape(z_post_act,
                   (z_post_act.shape[0], 1, 28, 28))
                return z_img

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

    #Implement this function for the DAE model
    def add_noise(self, x, noise_type):
        if noise_type=='Gaussian':
            # Gaussian Normal Noise: Mean = 0, Standard
               Deviation = 0.2
            noise = torch.normal(mean=0, std=0.2, size=x.shape)
```

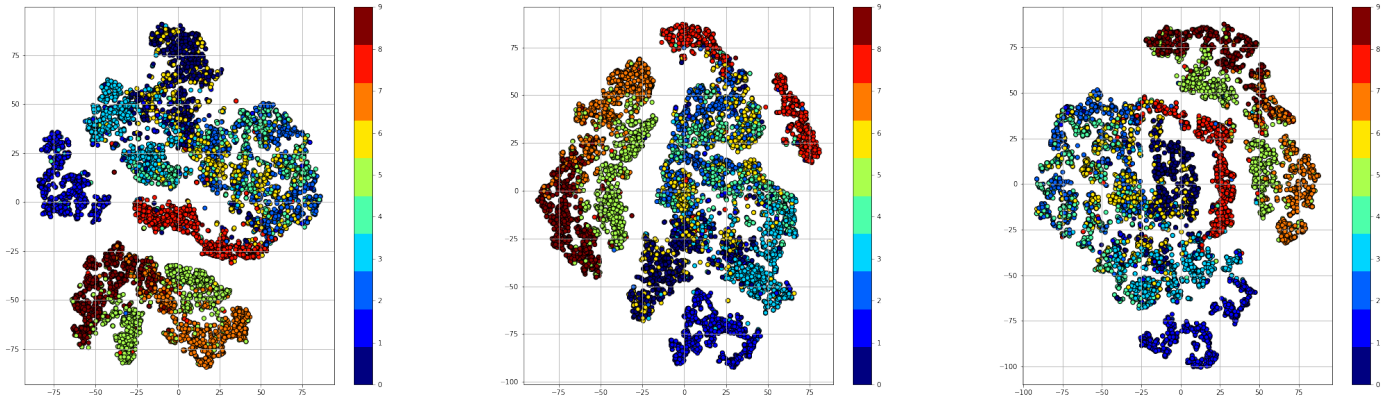
```
            return x + noise.cuda()
        elif noise_type=='Dropout':
            # amount of salt
            salt = nn.Dropout(p=0.4)
            return salt(x)

    def forward(self, x):
        if(self.noise_type != None and self.training):
            x = self.add_noise(x, self.noise_type)
        x = self.encoder(x)
        x = self.decoder(x)
        return x


def Plot_Kernel(_model):
    '''
    the plot for visualizing the learned weights of the
       autoencoder's encoder .
    ----------
    _model: Autoencoder
    '''
    kernel_enc = _model.encoder.fc.weight
    kernels = torch.reshape(kernel_enc, shape=(30, 28,
       28)).cpu().detach().numpy()
    fig, axs = plt.subplots(nrows=5, ncols=6, figsize=(10, 10))

    kernel_idx = 0
    for ax in axs:
        for a in ax:
            a.imshow(kernels[kernel_idx])
            a.axis("off")
            kernel_idx+=1

    plt.savefig("kernels.png")
    plt.show()
```

**Visualizations**



No Noise Gaussian Noise Dropout Noise

Scatter Plots for MNIST



No Noise Gaussian Noise Dropout Noise

Scatter Plots for FashionMNIST

Figure 6: Scatter Plots for the three DAE models (Latent Dimension=30, Activation=tanh)

No Noise

Gaussian Noise

Dropout Noise

Reconstructed Images for MNIST

No Noise

Gaussian Noise

Dropout Noise

Reconstructed Images for FashionMNIST

Figure 7: Reconstructed Images for the three DAE models (Latent Dimension=30, Activation=tanh)

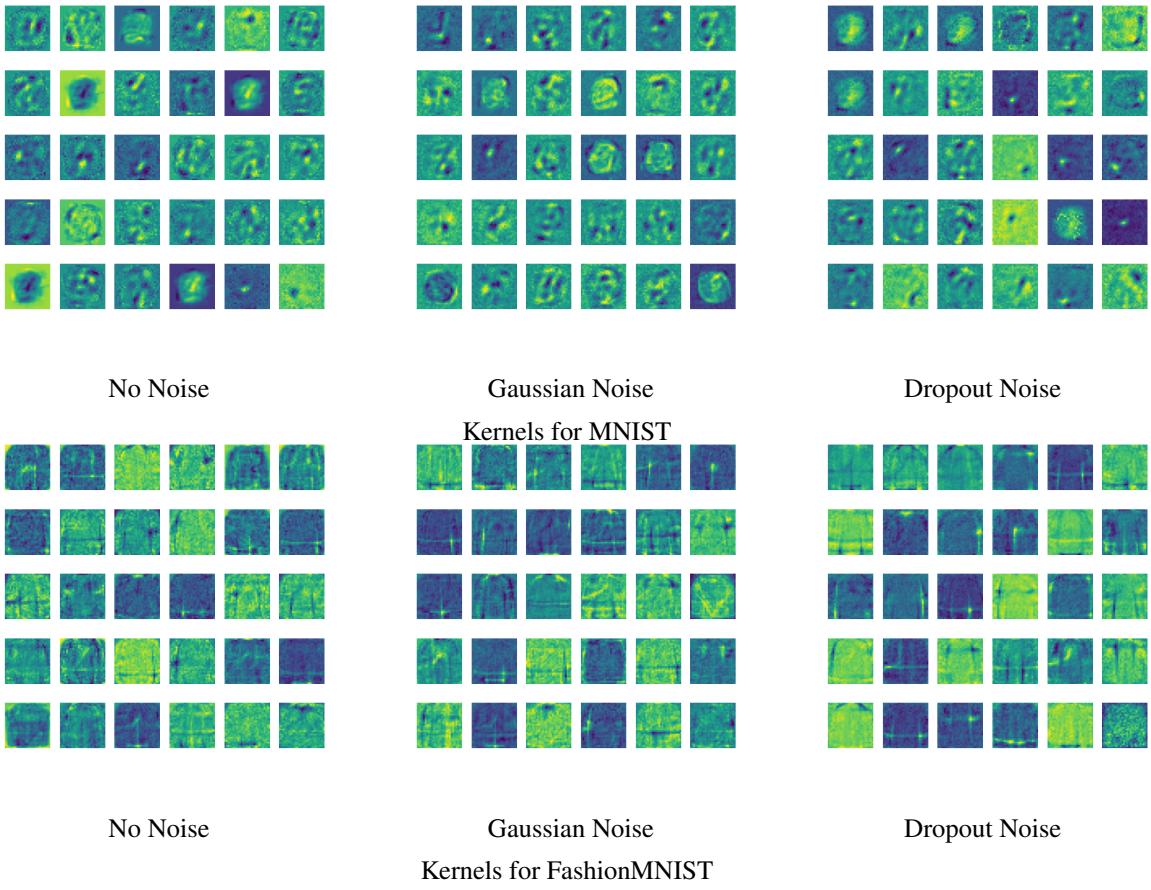



Figure 8: Kernels for the three DAE models (Latent Dimension=30, Activation=tanh)

**Description**

It is quite evident that the noise added to the network makes it more robust (Compare the last column in $4^{th}$ row for FashionMNIST and the last column in the $2^{nd}$ row for MNIST in the Figure 7). The Figure 8 shows us that adding noise (either Gaussian or Dropout) makes the Autoencoder learn more robust features and the learnt features are not just one-to-one/identity mapping of the inputs. For this look at the last column in $2^{nd}$ row of MNIST kernels where the identity mapping problem (5 is quite visible) can be witnessed. Whereas, the other two models don't have this problem. The DAE models learn the features common across the dataset, which makes it much more robust and as a result better looking results are achieved. The Gaussian Noise added to the inputs has the mean of 0 and a standard deviation of 0.2 and the dropout applied to the third DAE is p=0.4. In the first epoch, the validation loss (reconstruction error) for the no noise model is 0.6473; 0.6482 for the one with Gaussian noise and 0.6508 for the one with dropout noise. This behavior is as expected because we are adding noise to the inputs and comparing the losses with images without the noise. Similar trend is seen for the average loss (training loss) but in the model with Gaussian noise the training error goes lower than the model with no noise DAE till the last epoch. I employed t-SNE if the latent dimension is greater than 2, to reduce the dimension to 2 for visualization purposes.

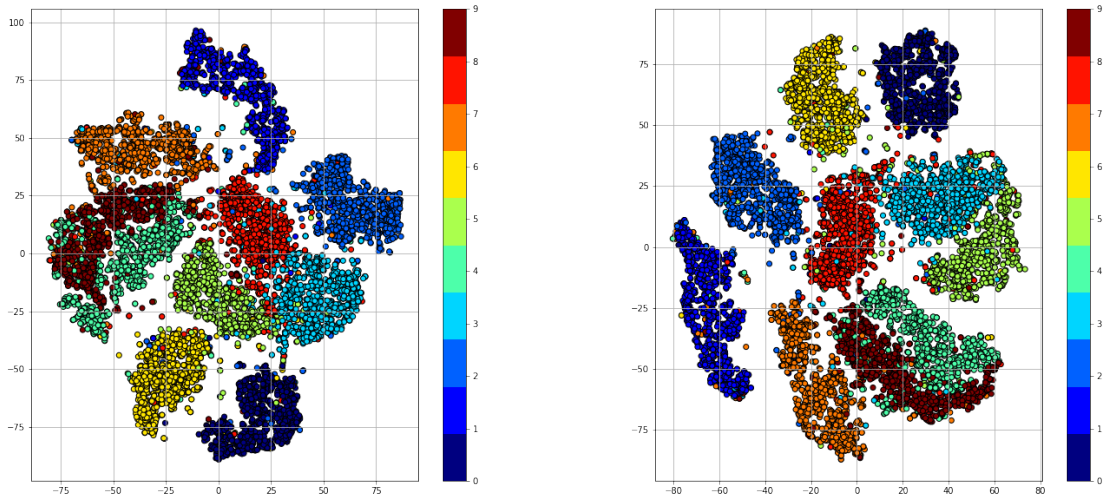
### 2.4 Variational Autoencoders (VAE)

**Code**

```
class VAE(nn.Module):
    def __init__(self,dim_latent_representation=30):
        super(VAE,self).__init__()

        class Encoder(nn.Module):
            def __init__(self, output_size=30):
                super(Encoder, self).__init__()
                self.fc = nn.Linear(28*28, output_size)
                self.tanh = nn.Tanh()
                self.mu_layer = nn.Linear(output_size,
                   dim_latent_representation)
                self.std_layer = nn.Linear(output_size,
                   dim_latent_representation)

            def forward(self, x):
                x_flat=torch.reshape(x, (x.shape[0], 28*28))
                z = self.tanh(self.fc(x_flat))
                mu = self.mu_layer(z)
                logvar = self.std_layer(z)
                return z, mu, logvar

        class Decoder(nn.Module):
            def __init__(self, input_size=30):
                super(Decoder, self).__init__()
                self.out = nn.Linear(input_size, 28*28)
                self.sigmoid = nn.Sigmoid()

            def forward(self, z):
                z_post_act = self.sigmoid(self.out(z))
                # FashionMNIST contains images of dimension
                   28*28 with 1 channel
                z_img=torch.reshape(z_post_act,
                   (z_post_act.shape[0], 1, 28, 28))
                return z_img

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

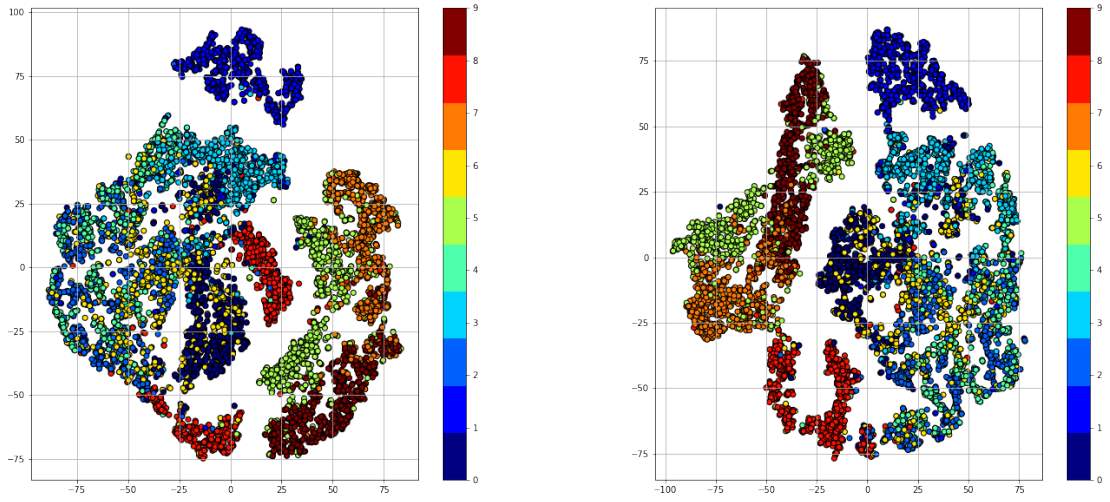
```
    # Implement this function for the VAE model
    # This will sample from the Gaussian Normal Distribution
       N(0, 1)
    def reparameterise(self, mu, logvar):
        if self.training:
            return mu + torch.exp(logvar * 0.5) *
               torch.randn_like(mu)
        else:
            return mu

    def forward(self, x):
        z, mu, logvar = self.encoder(x)
        z = self.reparameterise(mu, logvar)
        reconstructed_x = self.decoder(z)
        # for the VAE forward function should also return mu and
           logvar
        return reconstructed_x, mu, logvar


def Plot_Kernel(_model):
    '''
    the plot for visualizing the learned weights of the
       autoencoder's encoder .
    ----------
    _model: Autoencoder
    '''
    kernel_enc = _model.encoder.fc.weight
    kernels = torch.reshape(kernel_enc, shape=(30, 28,
       28)).cpu().detach().numpy()
    fig, axs = plt.subplots(nrows=5, ncols=6, figsize=(10, 10))

    kernel_idx = 0
    for ax in axs:
        for a in ax:
            a.imshow(kernels[kernel_idx])
            a.axis("off")
            kernel_idx+=1

    plt.savefig("kernels.png")
    plt.show()
```

**Visualizations**



Autoencoder Variational Autoencoder

Scatter Plots for MNIST



Autoencoder Variational Autoencoder

Scatter Plots for FashionMNIST

Figure 9: Scatter Plots AE vs VAE (Latent Dimension=30, Activation=tanh)

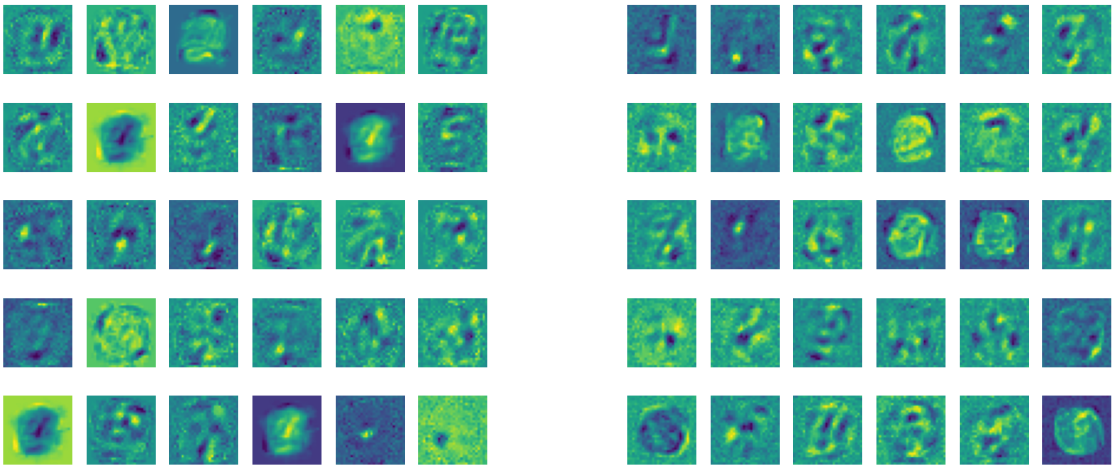
Autoencoder Variational Autoencoder

Reconstructed Images for MNIST

Autoencoder Variational Autoencoder

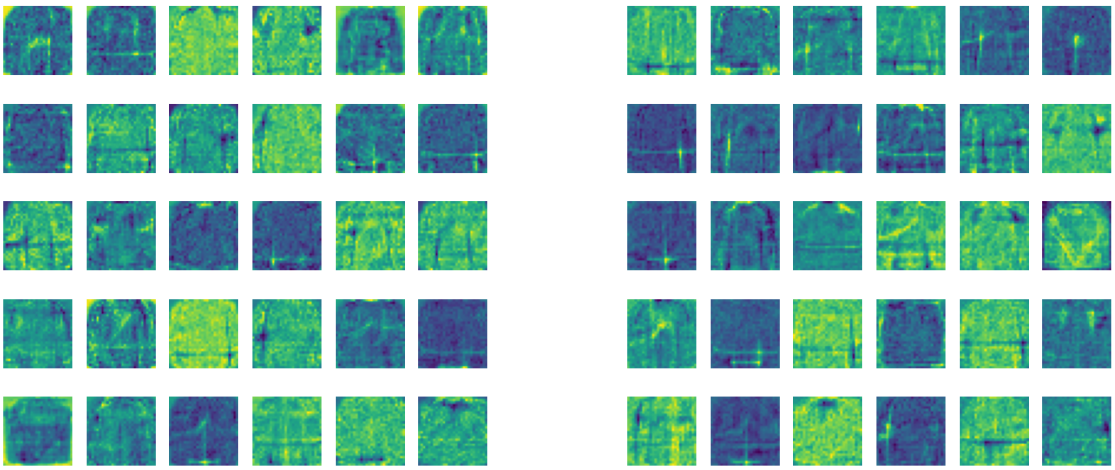
Reconstructed Images for FashionMNIST

Figure 10: Reconstructed Images AE vs VAE (Latent Dimension=30, Activation=tanh)



Autoencoder Variational Autoencoder

Kernels for MNIST



Autoencoder Variational Autoencoder

Kernels for FashionMNIST

Figure 11: Kernels AE vs VAE (Latent Dimension=30, Activation=tanh)

**Description**

Both of the models under consideration have no hidden layer and latent dimension of 30. The autoencoder asked for this part is just a DAE without any noise which is just a vanilla autoencoder (it is the same as the one used in subsection 2.3). The training loss and validation loss for the VAE is very high if the $D_{KL}$ is added to the loss as is. We need to weight the loss, Upon tuning the $kld_weight$ I found that weight value of $kld_weight = 0.00008$ yields quite good results. And on validation, only reconstruction loss should be considered (and not the KLD term). The losses of the VAE are slightly higher than that of the vanilla AE and that is because a term is added in the loss value (Training Loss for the last epoch (AE: 0.6080; VAE: 0.6816)). This results in very good results (Compare the second last column in the second last row for the MNIST dataset in Figure 10) Similar to the reasoning in section 2.3 there is a problem of identity mapping between the inputs and outputs, and the important features are not learnt, whereas, VAE does not have this problem (See Figure 11). Other than that the classes are slightly further apart in Figure 9 See that the 3rd class for MNIST is very far away from the the class 1 and 2. Similarly see the class 8 (which is a Bag) for the FashionMNIST is bunched up in a single blob in VAE whereas, Autoencoder fails to capture this detail. Which makes VAE much more robust.