Assignment 6

(a) Explain how the denoising autoencoder works.

A denoising autoencoder works to remove noise from the input data. It has 2 components, mainly encoder and decoder. The encoder compresses the data (input given to the encoder) and decoder reconstructs the original input data using the encoder's compressed output.

During the training, the input data is deliberately modified/corrupted by adding noise (*here add_noise function is used to add noise*) or some other modification. This corrupted data is the input of the encoder (*here CNN of encoder is defined in Encoder class*). Then the encoder will create a compressed data, and this compressed data is passed to the decoder (*here CNN of encoder is defined in Decoder class*). Decoder will try to create the same data as the original uncorrupted input data.

The network is trained (*here Training cycle part*) every epoch to minimize the difference between the reconstructed output (decoder's output) and the original uncorrupted input (*here MSE is used to calculate loss on both training and validation data).*

The encoder learns to denoise the corrupted input by extracting meaningful features from the noisy input, focusing on the most relevant information. The decoder then takes this compressed representation and attempts to reconstruct the original, clean data from it. The network is optimized to ensure that the reconstructed output closely matches the original input, hence removing the added noise.

(b) The model involves batch normalization. Learn what batch normalization is and briefly explain here.

Batch normalization is used to improve the training speed, stability, and performance of neural networks. It operates by normalizing the input of layers (*the layer where we want to normalize*) before applying the activation function (*here ReLU*) and typically after applying the linear transformations.

As activations in each layer vary widely during training, it impacts the learning process. So, generally batch normalization normalizes these activations by subtracting the mean and dividing by the standard deviation of the activations within a mini-batch.

So here batch normalization is helping to solve the issues like vanishing/exploding gradients. It allows faster convergence during training, reduces sensitivity to the initialization of weights and acts as a regularizer (often reducing the need for other regularization techniques).

In the provided code, batch normalization is used in the convolutional layers of both the encoder and decoder to stabilize and speed up the training process. Here, we are normalizing the encoder in the second convolutional layer and decoder in the first and second convolutional layer.

(c) Write a script (to be appended to the end of the script - see the file) that generates 9 random images of digits arranged in a 3 × 3 matrix. Labels are not important. Include the images with your report. Comment on the results.

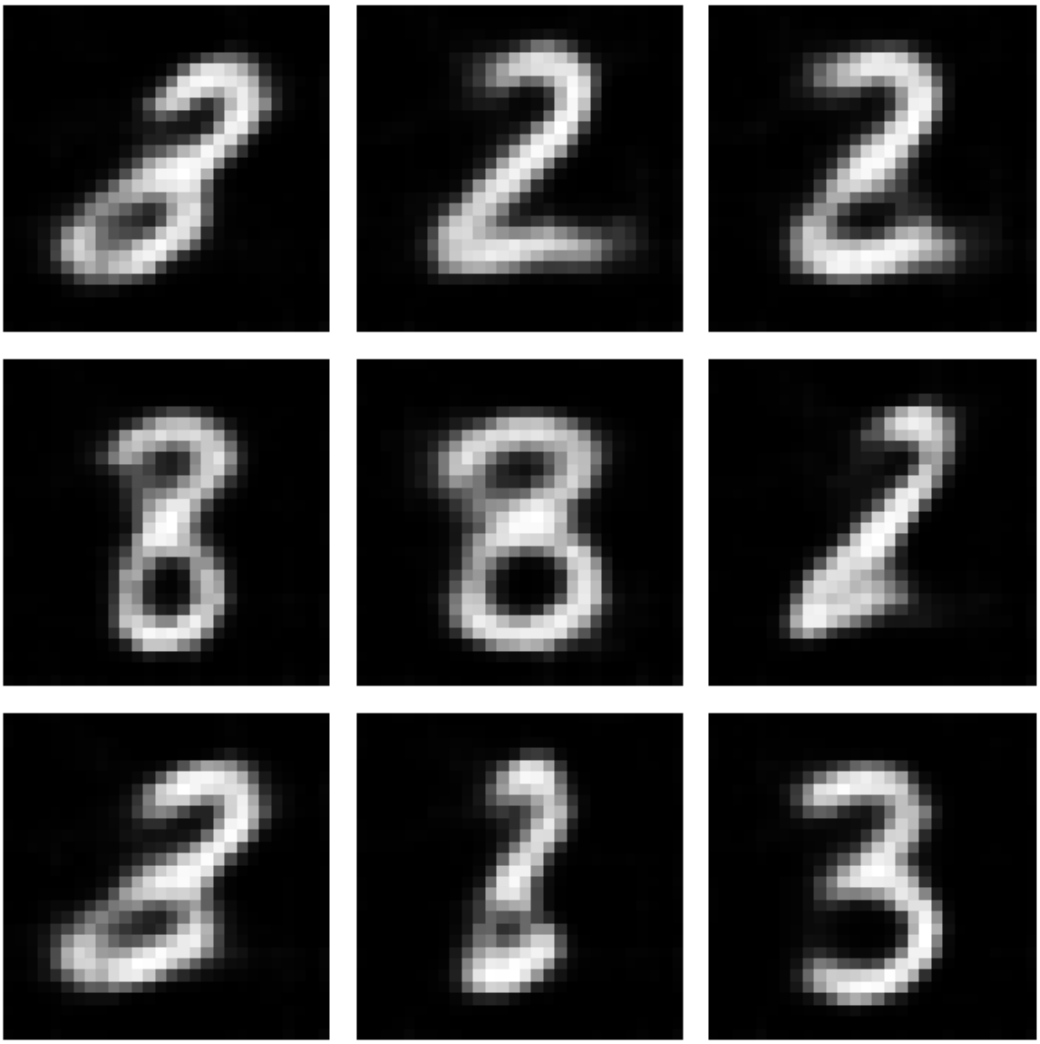
Description of the script :

This script generates nine new images using a trained denoising autoencoder's decoder. Inside a loop iterating nine times, it generates a random tensor, passes it through the decoder, and displays the resulting image in a 3x3 grid. The torch.no_grad() block ensures no gradient tracking during inference for efficiency. The images are shown using a grayscale colormap, and the final plot showcases the newly generated images without axis labels.

Script for generating 9 random images of digits arranged in a 3 × 3 matrix :

```
imgs_new = []

plt.figure(figsize=(6,6))

for n in range(9):

   inp_rnd = torch.rand(1, d).to(device)

   with torch.no_grad():
       imgs_gen = decoder(inp_rnd)

   plt.subplot(3, 3, len(imgs_new) + 1)
   plt.axis('off')
   plt.imshow(imgs_gen.cpu().squeeze().numpy(), cmap='gist_gray')
   imgs_new.append(imgs_gen)

plt.tight_layout()
plt.show()
```

Images obtained by running this script is :



Comment on Results :

The images are blurry but many are recognizable. The code (model) doesn't work well for test data(unknown data) as much as it works well for train data. So the model is not accurate and doesn't have efficiency for test data as much as train data. Efficiency is less as the output images of test data are more blurry (less clarity) than train data.

Since the images are random and generated from noise, there are variations in clarity, digit shapes, and consistency compared to the original MNIST dataset. So the test data is different from training data. For this model, test data has less accuracy compared to train data.

(d) Write a script that clusters the images in the dataset (to be appended at the end of the script - see the file). To do this, first obtain all encoder outputs for all elements of the

training set and cluster these outputs using the k-means algorithm. You will obtain a sequence of cluster assignments, like (1,1,2,2,2) under the assumption of 5 dataset elements and 3 clusters (you will have 48000 dataset elements and 10 clusters). You also have true labels from the dataset itself, such as (2,2,1,1,1). Now find the difference between the assignments found by k-means and the true labels to obtain the accuracy. You should find a good reassignment of indices such that the accuracy is maximized, because k-means is agnostic to labels. For example, the above example should yield zero errors, because we can assign 1s to 2s and 2s to 1s achieving perfect accuracy between the k-means labels and the true labels. Report the accuracy you obtain over the training set, and describe your index reassignment algorithm.

Description of the script :

The script conducts clustering evaluation using a denoising autoencoder's encoded data. It retrieves encoded representations of the dataset via the encoder, applies KMeans clustering on these representations, and determines the clustering accuracy using the Hungarian algorithm by aligning true and predicted labels. At the end, it prints the achieved accuracy of the clustering model.

Script that clusters the images using KMeans in the dataset and reports the accuracy :

```
from sklearn import metrics
from sklearn.cluster import KMeans
from collections import Counter
from scipy.optimize import linear_sum_assignment
from sklearn import metrics

def enc_opts(enc, dataLoader, dev):

   enc.eval()
   enc_opts = []
   t_labs = []

   with torch.no_grad():

       for img, lab in dataLoader:

           img = img.to(dev)
           t_labs.extend(lab.cpu().numpy())

           enc_data = enc(img)
           enc_opts.extend(enc_data.cpu().numpy())
```

```
    return np.array(enc_opts), np.array(t_labs)


def cls_kmeans(enc_opts, no_cls):
    kmen = KMeans(n_clusters=no_cls, random_state=0)
    cls_assig = kmen.fit_predict(enc_opts)
    return cls_assig


def cal_acc(t_labs, cls_assig):

    conf_mat = metrics.confusion_matrix(t_labs, cls_assig)
    r_idx, c_idx = linear_sum_assignment(-conf_mat)
    acc = conf_mat[r_idx, c_idx].sum() / len(t_labs)

    return acc

enc_opts, t_labs = enc_opts(encoder, train_loader, device)

no_cls = 10
cls_assig = cls_kmeans(enc_opts, no_cls)

acc = cal_acc(t_labs, cls_assig)

print(f"Accuracy obtained using Hungarian algorithm: {acc * 100:.2f}%")
```

Output obtained is :

```
print(f"Accuracy obtained using Hungarian algorithm: {acc * 100:.2f}%")

Accuracy obtained using Hungarian algorithm: 72.04%
```

I tried doing accuracy calculation using permutations (using for loops) of the label assignments to find the best match with the true labels. It iterated through all possible label permutations to find the one that maximizes accuracy. But, the permutations were computationally intensive as taking a long time to complete, as the dataset is large and clusters were also 10.

So I decided to use a Hungarian algorithm using a confusion matrix, to compute accuracy.

Final index reassignment algorithm :

Hungarian algorithm using confusion matrix :

The function cal_acc determines the accuracy of a clustering algorithm's predictions by comparing them to the true labels. It starts by creating a confusion matrix that counts how many samples from each true label were assigned to each cluster by the clustering algorithm.

The confusion matrix was looking like :

```
[[3542    0    2    2    0 1176   48    1    5    1]
 [   0    4    3   10 5197    1  143    0   21    0]
 [  26   29   51 3306   47    6 1321    1   28    0]
 [   2   15 3605   43   20   53 1095    1   57    8]
 [   1  250    0    1   30   54  121    1 4241    0]
 [   7    5  222    0   66 1465   49    0   62 2421]
 [  58    0    1    0   17 4563   43    0   35   13]
 [   0 2544    0    9  118    0  100 1808  370    0]
 [   3   29   47    2   31  238 4233    0   57   10]
 [   6 1661   13    0   17   28  114   26 2937    3]]
```

Next, it uses the Hungarian algorithm (linear_sum_assignment function) to find the best possible matching between the clusters and the true labels based on this confusion matrix. The output of this linear_sum_assignment function is row_index (r_idx) and col_index (c_idx): These variables store the row and column indices of the optimal assignment pairs that minimize the total cost. The indices represent the optimal matching between rows (True Labels) and columns (Predicted Labels).

Output from linear_sum_assignment function was looking like :

```
Linear_sum_assignment function
row_index :
[0 1 2 3 4 5 6 7 8 9]

column_index :
[0 4 3 2 8 9 5 7 6 1]
```

Then, by summing the counts of correctly matched samples, it computes the accuracy score, representing the proportion of correctly assigned samples compared to the total number of samples in the dataset.

Finally, this accuracy score indicates how well the clustering algorithm's assignments align with the actual labels in the dataset.

Using this algorithm I got an accuracy of 72.04%.

(e) Whole Code :

```
import matplotlib.pyplot as plt
import numpy as np # this module is useful to work with numerical arrays
import pandas as pd # this module is useful to work with tabular data
import random # this module will be used to select random samples from a
collection
import os # this module will be used just to create directories in the
local filesystem
from tqdm import tqdm # this module is useful to plot progress bars
import plotly.io as pio

import torch
import torchvision
from torchvision import transforms
from torch.utils.data import DataLoader,random_split
from torch import nn
import torch.nn.functional as F
import torch.optim as optim
from sklearn.manifold import TSNE
import plotly.express as px

from sklearn import metrics
from sklearn.cluster import KMeans
from collections import Counter
from scipy.optimize import linear_sum_assignment
from sklearn import metrics
```

```
data_dir = 'dataset'
### With these commands the train and test datasets, respectively, are
downloaded
### automatically and stored in the local "data_dir" directory.
train_dataset = torchvision.datasets.MNIST(data_dir, train=True,
download=True)
test_dataset  = torchvision.datasets.MNIST(data_dir, train=False,
download=True)


fig, axs = plt.subplots(5, 5, figsize=(8,8))
for ax in axs.flatten():
   # random.choice allows to randomly sample from a list-like object
(basically anything that can be accessed with an index, like our dataset)
   img, label = random.choice(train_dataset)
   ax.imshow(np.array(img), cmap='gist_gray')
   ax.set_title('Label: %d' % label)
   ax.set_xticks([])
   ax.set_yticks([])
plt.tight_layout()
plt.show()

train_transform = transforms.Compose([
   transforms.ToTensor(),
])
test_transform = transforms.Compose([
   transforms.ToTensor(),
])

# Set the train transform
train_dataset.transform = train_transform
# Set the test transform
test_dataset.transform = test_transform

m=len(train_dataset)

#random_split randomly split a dataset into non-overlapping new datasets
of given lengths
#train (55,000 images), val split (5,000 images)
```

```
train_data, val_data = random_split(train_dataset, [int(m-m*0.2),
int(m*0.2)])

batch_size=256

# The dataloaders handle shuffling, batching, etc...
train_loader = torch.utils.data.DataLoader(train_data,
batch_size=batch_size)
valid_loader = torch.utils.data.DataLoader(val_data,
batch_size=batch_size)
test_loader = torch.utils.data.DataLoader(test_dataset,
batch_size=batch_size,shuffle=True)




class Encoder(nn.Module):

   def __init__(self, encoded_space_dim,fc2_input_dim):
       super().__init__()

       ### Convolutional section
       self.encoder_cnn = nn.Sequential(
           # First convolutional layer
           nn.Conv2d(1, 8, 3, stride=2, padding=1),
           #nn.BatchNorm2d(8),
           nn.ReLU(True),
           # Second convolutional layer
           nn.Conv2d(8, 16, 3, stride=2, padding=1),
           nn.BatchNorm2d(16),
           nn.ReLU(True),
           # Third convolutional layer
           nn.Conv2d(16, 32, 3, stride=2, padding=0),
           #nn.BatchNorm2d(32),
           nn.ReLU(True)
       )

       ### Flatten layer
       #self.flatten = torch.flatten(start_dim=1)
```

```
        ### Linear section
        self.encoder_lin = nn.Sequential(
            # First linear layer
            nn.Linear(3 * 3 * 32, 128),
            nn.ReLU(True),
            # Second linear layer
            nn.Linear(128, encoded_space_dim)
        )

    def forward(self, x):
        # Apply convolutions
        x = self.encoder_cnn(x)
        # Flatten
        x = torch.flatten(x, start_dim=1)
        # # Apply linear layers
        x = self.encoder_lin(x)
        return x




class Decoder(nn.Module):

    def __init__(self, encoded_space_dim,fc2_input_dim):
        super().__init__()

        ### Linear section
        self.decoder_lin = nn.Sequential(
            # First linear layer
            nn.Linear(encoded_space_dim, 128),
            nn.ReLU(True),
            # Second linear layer
            nn.Linear(128, 3 * 3 * 32),
            nn.ReLU(True)
        )


        ### Convolutional section
        self.decoder_conv = nn.Sequential(
```

```
            # First transposed convolution
            nn.ConvTranspose2d(32, 16, 3, stride=2, output_padding=0),
            nn.BatchNorm2d(16),
            nn.ReLU(True),
            # Second transposed convolution
            nn.ConvTranspose2d(16, 8, 3, stride=2, padding=1,
output_padding=1),
            nn.BatchNorm2d(8),
            nn.ReLU(True),
            # Third transposed convolution
            nn.ConvTranspose2d(8, 1, 3, stride=2, padding=1,
output_padding=1)
        )

    def forward(self, x):
        # Apply linear layers
        x = self.decoder_lin(x)
        # Unflatten
        x = nn.Unflatten(dim=1, unflattened_size=(32, 3, 3))(x)
        # Apply transposed convolutions
        x = self.decoder_conv(x)
        # Apply a sigmoid to force the output to be between 0 and 1 (valid
pixel values)
        x = torch.sigmoid(x)
        return x




### Set the random seed for reproducible results
torch.manual_seed(0)

### Initialize the two networks
d = 4

encoder = Encoder(encoded_space_dim=d,fc2_input_dim=128)
decoder = Decoder(encoded_space_dim=d,fc2_input_dim=128)


### Define the loss function
```

```
loss_fn = torch.nn.MSELoss()

### Define an optimizer (both for the encoder and the decoder!)
lr= 0.001 # Learning rate


params_to_optimize = [
   {'params': encoder.parameters()},
   {'params': decoder.parameters()}
]

device = torch.device("cuda") if torch.cuda.is_available() else
torch.device("cpu")
# print(f'Selected device: {device}')

optim = torch.optim.Adam(params_to_optimize, lr=lr)

# Move both the encoder and the decoder to the selected device
encoder.to(device)
decoder.to(device)
#model.to(device)


def add_noise(inputs,noise_factor=0.3):
    noise = inputs+torch.randn_like(inputs)*noise_factor
    noise = torch.clamp(noise,0.,1.)
    return noise

### Training function
def train_epoch_den(encoder, decoder, device, dataloader, loss_fn,
optimizer,noise_factor=0.3):
   # Set train mode for both the encoder and the decoder
   encoder.train()
   decoder.train()
   train_loss = []
   # Iterate the dataloader (we do not need the label values, this is
unsupervised learning)
   for image_batch, _ in dataloader: # with "_" we just ignore the labels
(the second element of the dataloader tuple)
       # Move tensor to the proper device
```

```
        image_noisy = add_noise(image_batch,noise_factor)
        image_noisy = image_noisy.to(device)
        # Encode data
        encoded_data = encoder(image_noisy)
        # Decode data
        decoded_data = decoder(encoded_data)
        # Evaluate loss
        loss = loss_fn(decoded_data, image_batch)
        # Backward pass
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        # Print batch loss
        #print('\t partial train loss (single batch): %f' % (loss.data))
        train_loss.append(loss.detach().cpu().numpy())

    return np.mean(train_loss)


### Testing function
def test_epoch_den(encoder, decoder, device, dataloader,
loss_fn,noise_factor=0.3):
    # Set evaluation mode for encoder and decoder
    encoder.eval()
    decoder.eval()
    with torch.no_grad(): # No need to track the gradients
        # Define the lists to store the outputs for each batch
        conc_out = []
        conc_label = []
        for image_batch, _ in dataloader:
            # Move tensor to the proper device
            image_noisy = add_noise(image_batch,noise_factor)
            image_noisy = image_noisy.to(device)
            # Encode data
            encoded_data = encoder(image_noisy)
            # Decode data
            decoded_data = decoder(encoded_data)
            # Append the network output and the original image to the lists
            conc_out.append(decoded_data.cpu())
            conc_label.append(image_batch.cpu())
```

```
        # Create a single tensor with all the values in the lists
        conc_out = torch.cat(conc_out)
        conc_label = torch.cat(conc_label)
        # Evaluate global loss
        val_loss = loss_fn(conc_out, conc_label)
    return val_loss.data

def plot_ae_outputs_den(encoder,decoder,n=5,noise_factor=0.3):
    plt.figure(figsize=(10,4.5))
    for i  in range(n):

      ax = plt.subplot(3,n,i+1)
      img = test_dataset[i][0].unsqueeze(0)
      image_noisy = add_noise(img,noise_factor)
      image_noisy = image_noisy.to(device)

      encoder.eval()
      decoder.eval()

      with torch.no_grad():
         rec_img  = decoder(encoder(image_noisy))

      plt.imshow(img.cpu().squeeze().numpy(), cmap='gist_gray')
      ax.get_xaxis().set_visible(False)
      ax.get_yaxis().set_visible(False)
      if i == n//2:
        ax.set_title('Original images')
      ax = plt.subplot(3, n, i + 1 + n)
      plt.imshow(image_noisy.cpu().squeeze().numpy(), cmap='gist_gray')
      ax.get_xaxis().set_visible(False)
      ax.get_yaxis().set_visible(False)
      if i == n//2:
        ax.set_title('Corrupted images')

      ax = plt.subplot(3, n, i + 1 + n + n)
      plt.imshow(rec_img.cpu().squeeze().numpy(), cmap='gist_gray')
      ax.get_xaxis().set_visible(False)
      ax.get_yaxis().set_visible(False)
      if i == n//2:
         ax.set_title('Reconstructed images')
```

```
    plt.subplots_adjust(left=0.1,
                        bottom=0.1,
                        right=0.7,
                        top=0.9,
                        wspace=0.3,
                        hspace=0.3)
    plt.show()

### Training cycle
noise_factor = 0.3
num_epochs = 30
history_da={'train_loss':[],'val_loss':[]}

for epoch in range(num_epochs):
    print('EPOCH %d/%d' % (epoch + 1, num_epochs))
    ### Training (use the training function)
    train_loss=train_epoch_den(
        encoder=encoder,
        decoder=decoder,
        device=device,
        dataloader=train_loader,
        loss_fn=loss_fn,
        optimizer=optim,noise_factor=noise_factor)
    ### Validation  (use the testing function)
    val_loss = test_epoch_den(
        encoder=encoder,
        decoder=decoder,
        device=device,
        dataloader=valid_loader,
        loss_fn=loss_fn,noise_factor=noise_factor)
    # Print Validationloss
    history_da['train_loss'].append(train_loss)
    history_da['val_loss'].append(val_loss)
    print('\n EPOCH {}/{} \t train loss {:.3f} \t val loss
{:.3f}'.format(epoch + 1, num_epochs,train_loss,val_loss))
    plot_ae_outputs_den(encoder,decoder,noise_factor=noise_factor)


# put your image generator here
imgs_new = []
```

```
plt.figure(figsize=(6,6))

for n in range(9):

   inp_rnd = torch.rand(1, d).to(device)

   with torch.no_grad():
       imgs_gen = decoder(inp_rnd)

   plt.subplot(3, 3, len(imgs_new) + 1)
   plt.axis('off')
   plt.imshow(imgs_gen.cpu().squeeze().numpy(), cmap='gist_gray')
   imgs_new.append(imgs_gen)

plt.tight_layout()
plt.show()


# put your clustering accuracy calculation here


def enc_opts(enc, dataLoader, dev):

   enc.eval()
   enc_opts = []
   t_labs = []

   with torch.no_grad():

       for img, lab in dataLoader:

           img = img.to(dev)
           t_labs.extend(lab.cpu().numpy())

           enc_data = enc(img)
           enc_opts.extend(enc_data.cpu().numpy())

   return np.array(enc_opts), np.array(t_labs)
```

```
def cls_kmeans(enc_opts, no_cls):
   kmen = KMeans(n_clusters=no_cls, random_state=0)
   cls_assig = kmen.fit_predict(enc_opts)
   return cls_assig


def cal_acc(t_labs, cls_assig):

   conf_mat = metrics.confusion_matrix(t_labs, cls_assig)
   r_idx, c_idx = linear_sum_assignment(-conf_mat)
   acc = conf_mat[r_idx, c_idx].sum() / len(t_labs)

   return acc

enc_opts, t_labs = enc_opts(encoder, train_loader, device)

no_cls = 10
cls_assig = cls_kmeans(enc_opts, no_cls)

acc = cal_acc(t_labs, cls_assig)

print(f"Accuracy obtained using Hungarian algorithm: {acc * 100:.2f}%")
```