1 a)

Denoising AutoEncoder:

AutoEncoder has an encoder, decoder architecture where the encoder takes the input and reduces it to latent representation. The latent representation will be less than the dimensions of the input. Decoder takes this latent representation and tries to transform it into the original image. And the network is trained using Mean Square Loss between original image and decoded output. By having an encoder,decoder architecture we can ensure the latent representation is better learnt.

Denoising AutoEncoder are autoencoders which are used to remove noise from images. So in order for them to do this the encoder is fed a noise input and the decoder output is compared with the original image without noise. By training in this form we can remove the noise from an image.

b)

Batch Normalization: Batch normalization is a regularization technique used when training neural networks. We generally normalize the input before feeding to the neural network but after that the hidden layers modify the input and add activation to it. At every layer the output has a different distribution which is called internal covariate shift. Internal covariate shift makes it tougher to train the network as we may require to use lower learning rates and careful parameter initialization. By introducing a batch normalization layer we can speed up the training with less steps or epochs. Batch normalization consists of two steps as the name suggests it acts on batches of data and the first step is normalization for every layer output, the batchnorm layer calculates the mean and standard deviation and then normalizes the input to bring to a Gaussian distribution then we have a final operation which is rescaling and offsetting of the input wherein the batch normalization algorithm comes into picture it takes two values ɣ(Gamma) and β(beta) these parameters are used for rescaling and offsetting the input. These two are learnable parameters during training the neural network ensures the optimal values of Gamma and Beta. That will enable accurate normalization of each batch.

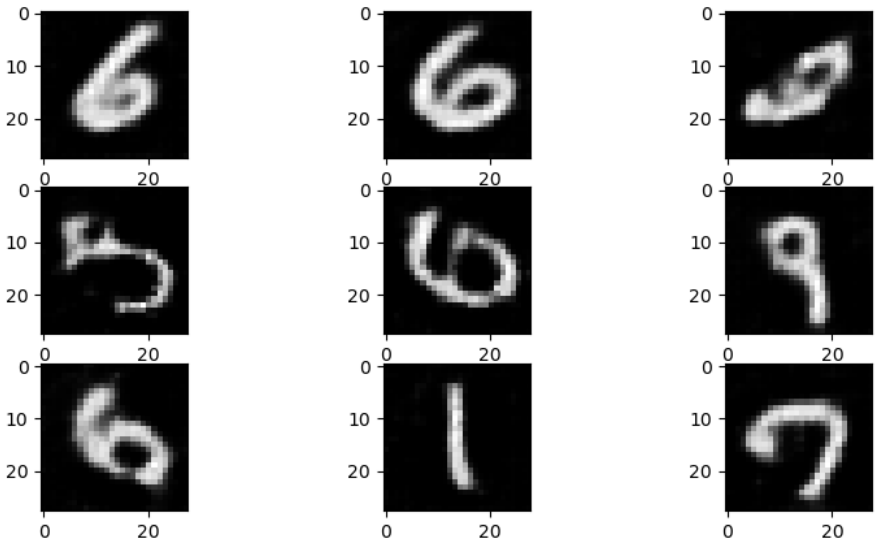
$h_{i(norm)} = \frac{h_i - \mu}{\sigma + \xi}$ - Normalization operation ϵ ensures numerical stability within the operation by stopping a division by zero value.

$h_i = \gamma h_{norm} + \beta$ - Offsetting operation

c)

By looking at the output we can see that even though we have passed random input from a normal distribution the decoder decodes them to images which are similar to the dataset. Though most of them are blurry. The blurry effect is due to Mean square loss which is used to

train the network, the decoder learns to minimize the loss by selecting pixel values which are the mean of corresponding pixels.



d)

For the index reassignment algorithm I have first constructed a cost matrix which 10*10 dimensions as we have 10 classes and 10 clusters. In this matrix the rows represent the true label class and the column represents the cluster labels. To populate the cost matrix I have used a for loop which iterates 48000 elements when we come across a true label and corresponding cluster label together we update the cost matrix value by 1 for that location for example if true label is 1 and cluster label is 8 then 1st row, 8th column value is updated by one. After getting the cost matrix, to find the optimal assignment which maximizes the cost I have used the Hungarian algorithm from Scipy.optimize these gave me the rows and their corresponding columns which give the optimal cost. Using these values I have built a dictionary which maps the cluster labels to true labels. In another loop I have changed the cluster label values to their corresponding true label mapping using the dictionary built earlier. After this I have used sklearn accuracy score to calculate the accuracy between the cluster labels and true labels.

Accuracy Score after index reassignment 0.7671041666666667

To reproduce the same results random.seed(0),torch.manual_seed(0) were set before running the code

Code:

```
fig = plt.figure(figsize=(10,7))
rows = 3
columns = 3

random_samples = torch.randn((9,4),device=device)
with torch.no_grad():
  samples = decoder(random_samples).cpu()
  for i in range(random_samples.shape[0]):
    fig.add_subplot(rows, columns, i+1)
    plt.imshow(samples[i][0],cmap="gist_gray")

plt.show()
# put your clustering accuracy calculation here

dataset_x = np.empty((48000,4))
dataset_y = np.empty((48000,))
i = 0

for X,y in train_loader:
  image_noisy = add_noise(X,noise_factor)
  image_noisy = image_noisy.to(device)
  encoded_output = encoder(image_noisy.to(device))
  dataset_x[i:i+batch_size,:] = encoded_output.cpu().detach().numpy()
  dataset_y[i:i+batch_size] = y
  i+=batch_size

kmeans = KMeans(n_clusters=10,random_state=1)
cluster_labels = kmeans.fit_predict(dataset_x)

dp = np.zeros((10,10))
dataset_y = dataset_y.astype(int)
for i in range(len(cluster_labels)):
  dp[dataset_y[i]][cluster_labels[i]] += 1

row,col = linear_sum_assignment(dp,maximize=True)

index_reassignment = {}

for i,j in zip(row,col):
  index_reassignment[j] = i

labels = np.copy(cluster_labels)
for i in range(0,len(cluster_labels)):
```

```
  labels[i] = index_reassignment[labels[i]]

print("Accuracy Score after index reassignment",accuracy_score(labels,dataset_y))
```