Neural Networks Assignment 6

a. The denoising autoencoder encodes MNIST images and decodes them. The input is first sent into the model with some noise added using the add_noise function. A noise factor is defined which is added to each pixel value by multiplying a random generated tensor the same shape as image and multiplying by the noise factor. This noisy image is then sent to the encoder and decoder. The encoder contains the following layers: 3 Convolution layers which start with 8 output channels and the output channels double each time, so the final layer has 32 output channels. Each layer also has a stride of 2, padding of 1 and kernel size of 3. Each layer also has a ReLU activation. After the 2nd layer, there is a Batch Normalization layer in between the 2nd and 3rd layer. The final 3 * 3 * 32 size output is connected to a fully connected layer with 128 neurons with ReLU activation. The final output layer has 'd' neurons without any activation function. In this case d is set to 4. The decoder is the same but in reverse. MSE is selected as the loss function and Adam is selected as the optimizer method, with a learning rate of 0.001. There are two methods for training and testing the encoder and decoder. Within these functions, the images are loaded with the DataLoader class and noise is added with the add_noise function. The training is done for 30 epochs.

b. Batch normalization is a technique where the output of each layer for a batch is normalized. The mean of the batch is subtracted from each output and each output is also divided by the variance of this batch. There is also a scale factor (gamma) and a shift factor (beta) which is applied to this normalized data, and these parameters are trainable. This is how it works during training but during testing or inference, a running mean of the mean and variance is used to normalize. Without gamma and beta, the expressiveness or representation of the nn on the data may be fixed and limited due to the normalization of the data.

d. I have used sklearn.cluster.KMeans method to perform clustering. Index reassignment is done in the following way: the KMeans method has labels array which can be accessed using KMeans.labels_. By using a mask which takes values 0 through 9, a boolean array is extracted which has values true wherever the mask value is true in the labels array. Eg: if the mask value is 4, from the line mask = (kmeans_labels == i) returns an array of Boolean values which are true wherever 4 is existing in the kmeans_labels array. This mask array is used to extract the values in the same true indices for the actual labels array. So if 4's true value is 9, mask array will extract an array of 9s majorly and some more values from the true label array. The mode function from the scipy.stats module is used to find the most common value, which would be 9 in our example. Thus a map is maintained for the kmeans_label value : true_label value. Using this map to create a new array replacing the values in kmeans_labels array with the mapped values, and the true labels array, the accuracy is calculated using the sklearn.metrics.accuracy_score method. The accuracy with this method was reported to be around 77 percent.

e. Here is the code added to the autoencoder python file:

```
# put your image generator here

# Generate 9 random latent vectors
random_encoded_vectors = torch.randn(9, d).to(device)  # Generate random
vectors

# Decode the random vectors
decoder.eval()  # Set the decoder to evaluation mode
with torch.no_grad():
```

```
    generated_images = decoder(random_encoded_vectors)

# Convert the output tensors to image format and plot
fig, axs = plt.subplots(3, 3, figsize=(8, 8))
for i, ax in enumerate(axs.flatten()):
    img = generated_images[i].cpu().detach().numpy()  # Convert to numpy array
    img = np.squeeze(img)  # Remove unnecessary dimensions
    ax.imshow(img, cmap='gray')
    ax.axis('off')

plt.show()
from sklearn.metrics import accuracy_score
from scipy.stats import mode

# put your clustering accuracy calculation here

encoded_images = []
true_labels = []
encoder.eval()
with torch.no_grad():
    for images, labels in train_loader:
        images = images.to(device)
        encoded = encoder(images).cpu().numpy()
        encoded_images.extend(encoded)
        true_labels.extend(labels.numpy())

encoded_images = np.array(encoded_images)
true_labels = np.array(true_labels)
best_kmeans = None
best_inertia = float('inf')

for i in range(10):  # For example, 10 different initializations
    kmeans = KMeans(n_clusters=10, n_init=10, random_state=i)
    kmeans.fit(encoded_images)

    if kmeans.inertia_ < best_inertia:
        best_kmeans = kmeans
        best_inertia = kmeans.inertia_

kmeans_labels = best_kmeans.labels_


# Step 3: Find the best mapping between k-means assignments and true labelse

def find_best_mapping(kmeans_labels, true_labels):
    label_mapping = {}
    for i in range(10):  # Assuming 10 clusters/digits
        mask = (kmeans_labels == i)
        # Find the most common true label in this cluster
```

```
        print(mask)
        print(true_labels[mask])
        mode_result = mode(true_labels[mask])
        print(mode_result)
        # mode_result.mode might be a scalar or an array
        if np.isscalar(mode_result.mode):
            most_common_label = mode_result.mode
        else:
            most_common_label = mode_result.mode[0]

        label_mapping[i] = most_common_label
    return label_mapping


label_mapping = find_best_mapping(kmeans_labels, true_labels)
print(label_mapping)
mapped_labels = [label_mapping[label] for label in kmeans_labels]
print(mapped_labels)
# Step 4: Calculate and report accuracy
accuracy = accuracy_score(true_labels, mapped_labels)
print(f'Clustering accuracy: {accuracy}')
```

Here are the generated 9 images: