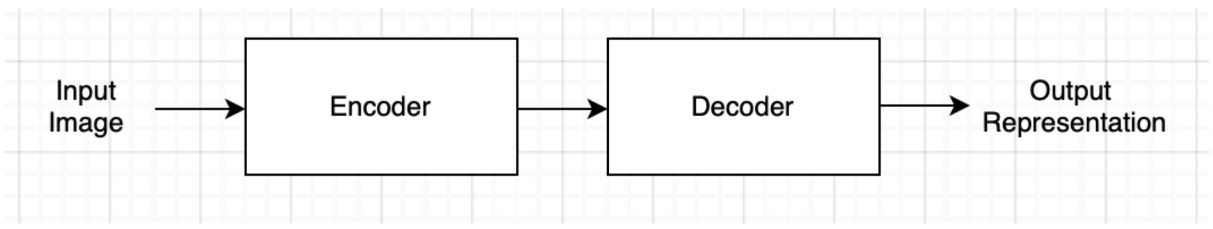
Name: Advait Milind Pai
Email: apai21@uic.edu
UIN: 677368201

Homework 6

A) Denoising autoencoder

Overall Structure of autoencoder



Denoising

Noising is the process of deliberately adding noise to data. Usually we add noise to the input data to prevent overfitting. Now if we were to create an simple Autoencoder without Denoising, we would have two problems, first being overfitting. The second is that the autoencoder may learn weights such that input = output, which actually makes the autoencoder redundant. By adding noise, we have the autoencoder comparing the loss on the output of a noised image and the input image, making the weights more useful as they generalise well over the noised data.

Our program adds noise with the following formula:

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

torch.clamp is used to adjust the input back between 0 and 1 as we add noise which may push the information beyond 1.

Encoder

The encoder is made up of two major parts i.e. the convolution layers and the linear layers. The convolution layers have 3 convolution layers and the linear layers is made up of a 2 linear layers. Between the two parts, we have a flatten function, the flatten function is necessary for PyTorch as we need to transform the tensor from the convolution layers into the format required by the linear layers.

i. The purpose of the encoder is to create a representation of the input image in lesser dimensions.
ii. The purpose of the convolution layers is for feature extraction. The convolution layer starts with extracting a primitive feature and as we progress in the convolution layers, we build on more and more complex features finally making up the image itself.
iii. The purpose of the linear layer is to then convert these features from the convolution layers into representations in the dimensions we require (like the one we have used for k-means).

Decoder

The decoder's purpose is to recreate the noised input information. We do this to see how good our encoder is at representing the input image in lower dimensions. Intuitively, if we are to recreate an image from our representation, we must reverse engineer the process we did to encode the image, which is what we do. Therefore the decoder layer first has 2 linear layers and then 3 convolution layers to recreate an output image.

Encoder + Decoder

The combination of the encoder and decoder are then used to calculate the weights for the image encoder. The decoder's output is used to calculate the loss of the noised input image against the original image such that the weights can be updated for the autoencoder so that we get a proper representation of the image in the lower dimensions from the encoder as we need.

I have used the encoder to pass the training data to the K-Means algorithm. The process of using autoencoders helped increase the accuracy to 73.46% instead of just 53% if I had used the 28x28 (784) pixel data.

B) **Batch Normalization**

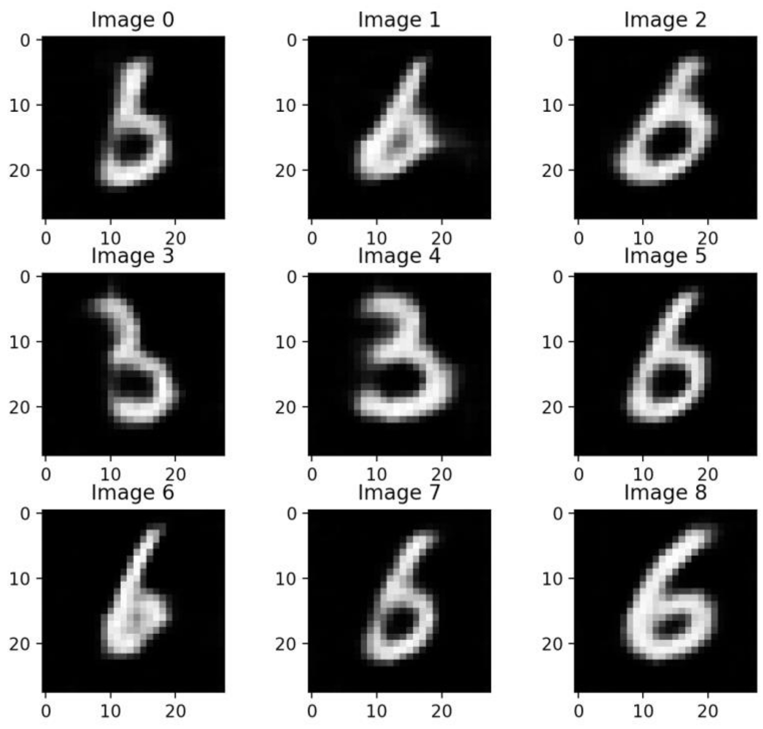
On studying about Batch Normalisation in general, I understand that it is used for regularization in deep neural networks. The first advantage of doing batch normalisation is that we increase the training speed by significantly reducing the number of training epochs by doing batch training.

Now when we show our training samples in the form of batches, we are only using a small subset of the data, and each subset of the data can vary from each other. This causes a problem where the training always pursues "moving target", and the weights being calculated may also keep shifting. For deep networks, the weights are calculated progressively backwards using the batch input and outputs having a more cascading effect. This causes more of a shift during training. To prevent this, we need to do batch normalisation.

Batch normalisation will rescale the outputs of the previous activation layers to mean and a standard deviation (usually mean = 0, stdev = 1, but in Batchnorm2d, Pytorch sets the mean and standard deviation it deems best). Doing this allows a standardisation of the presumptions of the underlying layers during training, thus increasing the speed of training as well as stabilising the neural network.

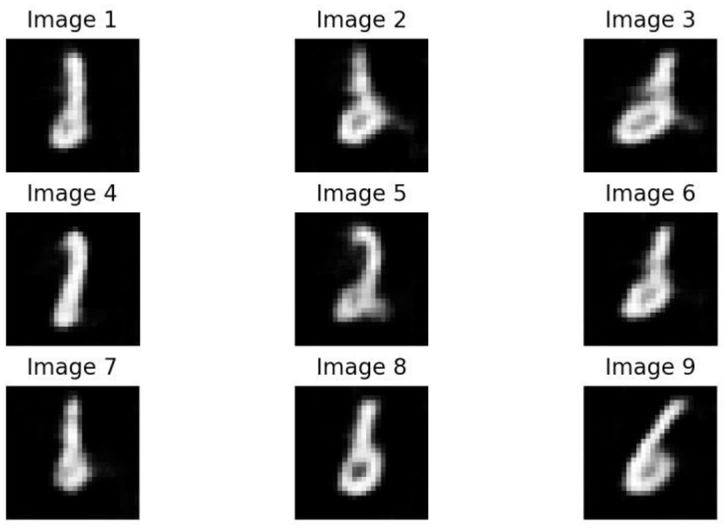
C) **Image Generation – code snippet at end of section**

Output:



Comments:

For the first image, while we are able to replicate the digits by the following code. The only thing observed is that the digits come out very pixelated and not consistent as well. In the bottom row, all the three 6's look very different as well, for the first image.



Comments:

I ran this for two times, the second has more ambiguous outputs. Note: the second image has the axis off as well as the labels corrected. Since I got an ambiguous output the second time, I felt it necessary to include the first one in my report as well. In the second output, while the 2's seem clearer, Image 7 is almost unrecognisable, it may be a 1, 8 or 6. Image 1 and Image 6 are unclear as well.

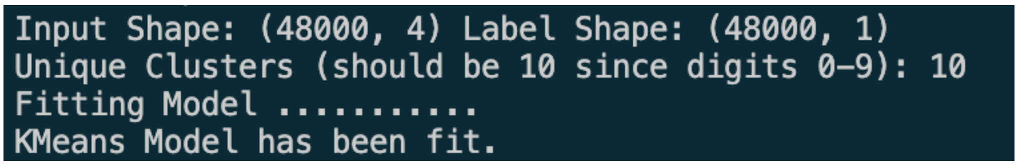
This could be because of the weights trained for autoencoder as well as the random number generated by torch. I have not set a seed because I wanted to see how the outputs change with different random numbers.

Code:

```
output = decoder(torch.rand(9,4))
# plt.subplot(3,3,0)
for i in range(0,len(output)):
    with torch.no_grad():
        plt.subplot(3,3,i+1)
        plt.title("Image "+str(i+1))
        plt.imshow(output[i].cpu().squeeze().numpy(), cmap='gist_gray')
        plt.axis('off')
plt.show()
```

D) **Cluster Script for k-means algorithm and Index Reassignment – code snippet at the end of section**

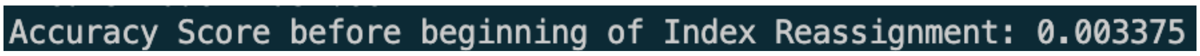
Model Fitting:



```
Input Shape: (48000, 4) Label Shape: (48000, 1)
Unique Clusters (should be 10 since digits 0-9): 10
Fitting Model ...........
KMeans Model has been fit.
```

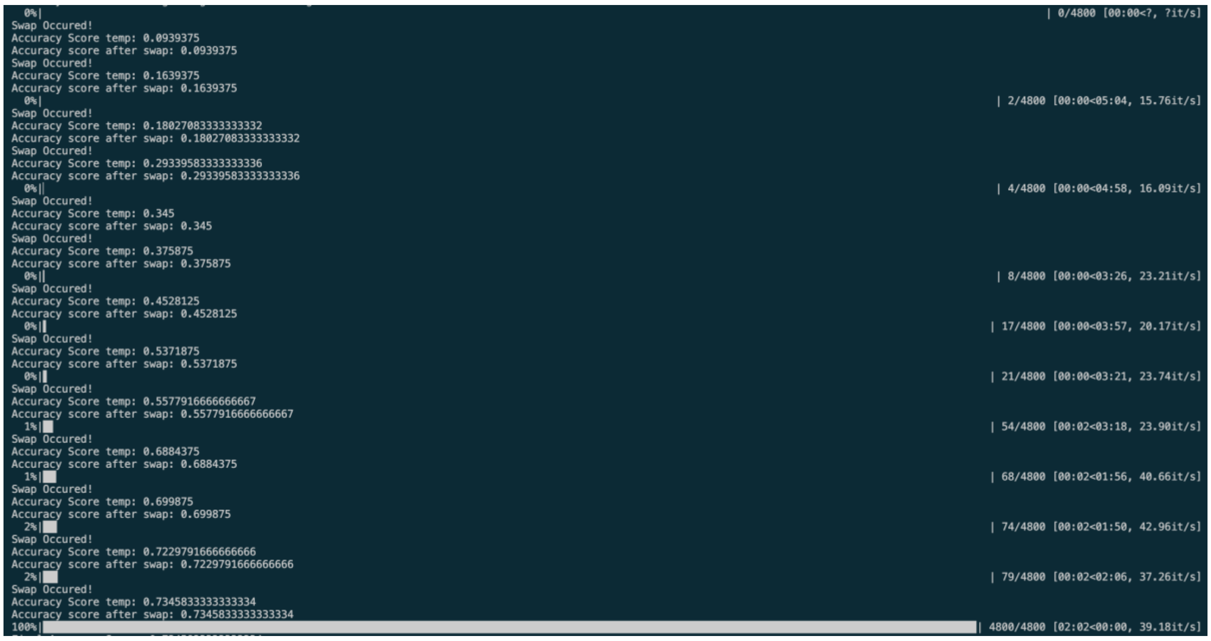
Input Data: Output from encoder of shape (48000,4) verifying we have 48000 samples.

Accuracy Score before Index Reassignment:



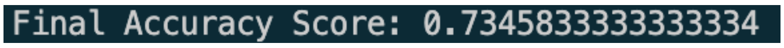
```
Accuracy Score before beginning of Index Reassignment: 0.003375
```

Index Reassignment output on next page:



```
  0%|                                                            | 0/4800 [00:00<?, ?it/s]
Swap Occured!
Accuracy Score temp: 0.0939375
Accuracy score after swap: 0.0939375
Swap Occured!
Accuracy Score temp: 0.1639375
Accuracy score after swap: 0.1639375
  0%|                                                       | 2/4800 [00:00<05:04, 15.76it/s]
Swap Occured!
Accuracy Score temp: 0.18027083333333332
Accuracy score after swap: 0.18027083333333332
Swap Occured!
Accuracy Score temp: 0.29339583333333336
Accuracy score after swap: 0.29339583333333336
  0%||                                                     | 4/4800 [00:00<04:58, 16.09it/s]
Swap Occured!
Accuracy Score temp: 0.345
Accuracy score after swap: 0.345
Swap Occured!
Accuracy Score temp: 0.375875
Accuracy score after swap: 0.375875
  0%||                                                     | 8/4800 [00:00<03:26, 23.21it/s]
Swap Occured!
Accuracy Score temp: 0.4528125
Accuracy score after swap: 0.4528125
  0%|▌                                                   | 17/4800 [00:00<03:57, 20.17it/s]
Swap Occured!
Accuracy Score temp: 0.5371875
Accuracy score after swap: 0.5371875
  0%|▌                                                   | 21/4800 [00:00<03:21, 23.74it/s]
Swap Occured!
Accuracy Score temp: 0.5577916666666667
Accuracy score after swap: 0.5577916666666667
  1%|█                                                   | 54/4800 [00:02<03:18, 23.90it/s]
Swap Occured!
Accuracy Score temp: 0.6884375
Accuracy score after swap: 0.6884375
  1%|█                                                   | 68/4800 [00:02<01:56, 40.66it/s]
Swap Occured!
Accuracy Score temp: 0.699875
Accuracy score after swap: 0.699875
  2%|█                                                   | 74/4800 [00:02<01:50, 42.96it/s]
Swap Occured!
Accuracy Score temp: 0.7229791666666666
Accuracy score after swap: 0.7229791666666666
  2%|█                                                   | 79/4800 [00:02<02:06, 37.26it/s]
Swap Occured!
Accuracy Score temp: 0.7345833333333334
Accuracy score after swap: 0.7345833333333334
100%|██████████████████████████████████████████████| 4800/4800 [02:02<00:00, 39.18it/s]
```

<u>Final Accuracy after Index Reassignment:</u>



```
Final Accuracy Score: 0.7345833333333334
```

<u>Thus we see that we obtain a Final Accuracy score of 73.46%</u>

<u>Algorithm for Index Reassignment:</u>

The logic I have used for index reassignment is to swap the indices in a manner we obtain the maximum accuracy score, implying all clusters are correctly assigned. Here is the algorithm:

```
a. Get initial accuracy score before
b. Iterate through (predicted labels/10). **
    a. Compare with the true label.
    b. If the labels do not match:
        i. Create a temporary label list containing the predicted labels to hold the
           values for manipulations stated below.
        ii. Swap the labels. For example, if we predicted the cluster label as 3, but
            the true label is 1, change all the labels 3 to 1 and 1 to 3 in the
            temporary list.
        iii. If the accuracy score of the temporary label list is greater than the
             previous accuracy score:
            1. Make the temporary label list as the predicted label list.
            2. Store the new accuracy score for further comparisons.
c. Print the final accuracy score.
```

** I only iterate through the first 4800 predicted labels as:

  a. I have observed that the swapping actually occurs in the first 1%-2% of 4800 itself, but hardcoding the number is not a good.
  b. I believed that by using predicted labels/10, we would encountered each cluster at least once for consideration of swapping.

c. Iterating through all 48000 labels takes 20 mins, whereas iterating over 4800 labels takes only 2 mins.

<u>Code for clustering and index reassignment:</u>

```
from sklearn.cluster import KMeans # For clustering
from sklearn.metrics import accuracy_score

training_set = torch.utils.data.DataLoader(train_data,batch_size=48000) # Loading the entire dataset as the batch
for img_data,label_data in training_set:
    with torch.no_grad():
        img_data = encoder(img_data)
    img_train = img_data.numpy()
    img_label = label_data.numpy().reshape(48000,1)
    print("Input Shape:",img_train.shape,"Label Shape:",img_label.shape)

total_clusters = len(np.unique(img_label))
print("Unique Clusters (should be 10 since digits 0-9):",total_clusters)

model = KMeans(n_clusters = total_clusters, random_state=2702,n_init=10)
print("Fitting Model ..........")
model.fit_transform(img_train)
print("KMeans Model has been fit.")

kmeans_label = model.labels_
acc_score = accuracy_score(kmeans_label,img_label)
print("Accuracy Score before beginning of Index Reassignment:", acc_score)

for i in tqdm(range(0,int(len(kmeans_label)/10))):
    if (kmeans_label[i] != img_label[i]): # Index Swapping if the label does not match
        temp_label = kmeans_label
        correct_label = img_label[i] # True Value
        incorrect_label = kmeans_label[i] # Predicted Value
        for i in range(0,len(temp_label)): # Swapping indexes
            if(temp_label[i] == incorrect_label):
                temp_label[i] = correct_label
            elif(temp_label[i] == correct_label):
                temp_label[i] = incorrect_label
        temp_acc_score = accuracy_score(temp_label,img_label)
        if (temp_acc_score > acc_score):
            kmeans_label = temp_label
            acc_score = temp_acc_score
            print("Swap Occured!")
            print("Accuracy Score temp:",temp_acc_score)
            print("Accuracy score after swap:",accuracy_score(kmeans_label,img_label))

print("Final Accuracy Score:",acc_score)
```

FULL CODE (page 13 has codes for Part C and Part D):

```
# https://github.com/eugeniaring/Medium-Articles/blob/main/Pytorch/denAE.ipynb

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


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):
```

```python
    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() Has been commented to prevent a pause between every 1 of the 30 epochs, uncomment to
display the graph after every epoch

### 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
output = decoder(torch.rand(9,4))
# plt.subplot(3,3,0)
for i in range(0,len(output)):
    with torch.no_grad():
        plt.subplot(3,3,i+1)
        plt.title("Image "+str(i+1))
        plt.imshow(output[i].cpu().squeeze().numpy(), cmap='gist_gray')
        plt.axis('off')
plt.show()

# put your clustering accuracy calculation heres

from sklearn.cluster import KMeans # For clustering
from sklearn.metrics import accuracy_score

training_set = torch.utils.data.DataLoader(train_data,batch_size=48000) # Loading the entire dataset as the batch
for img_data,label_data in training_set:
    with torch.no_grad():
        img_data = encoder(img_data)
    img_train = img_data.numpy()
    img_label = label_data.numpy().reshape(48000,1)
    print("Input Shape:",img_train.shape,"Label Shape:",img_label.shape)

total_clusters = len(np.unique(img_label))
print("Unique Clusters (should be 10 since digits 0-9):",total_clusters)

model = KMeans(n_clusters = total_clusters, random_state=2702,n_init=10)
print("Fitting Model ..........")
model.fit_transform(img_train)
```

```python
print("KMeans Model has been fit.")

kmeans_label = model.labels_
acc_score = accuracy_score(kmeans_label,img_label)
print("Accuracy Score before beginning of Index Reassignment:", acc_score)

for i in tqdm(range(0,int(len(kmeans_label)/10))):
    if (kmeans_label[i] != img_label[i]): # Index Swapping if the label does not match
        temp_label = kmeans_label
        correct_label = img_label[i] # True Value
        incorrect_label = kmeans_label[i] # Predicted Value
        for i in range(0,len(temp_label)): # Swapping indexes
            if(temp_label[i] == incorrect_label):
                temp_label[i] = correct_label
            elif(temp_label[i] == correct_label):
                temp_label[i] = incorrect_label
        temp_acc_score = accuracy_score(temp_label,img_label)
        if (temp_acc_score > acc_score):
            kmeans_label = temp_label
            acc_score = temp_acc_score
            print("Swap Occured!")
            print("Accuracy Score temp:",temp_acc_score)
            print("Accuracy score after swap:",accuracy_score(kmeans_label,img_label))

print("Final Accuracy Score:",acc_score)
```