Part 1

Implementation

To implement an autoencoder, we will utilize the torch neural network as our superclass. Firstly, we need to transform our input data, which is an N*(28*28) dataset, into a tensor object. Since autoencoders are unsupervised learning models, we do not require labels for training. Additionally, we will normalize our data by mapping it to the range of [0, 1]. This is achieved by dividing each pixel value by 255, as 255 represents the maximum value for a pixel in the dataset. Normalization is essential because we will be using a sigmoid activation function at the end of the autoencoder, which produces outputs between 0 and 1.

Regarding the autoencoder architecture, we inherit from the torch.nn module, and thus our init method remains the same. The encoder component of our autoencoder consists of five layers, with each layer employing the rectified linear unit (ReLU) as the activation function. ReLU is chosen due to its excellent performance in neural networks and its computational efficiency. Through these five layers, the input data is transformed from a 28*28-dimensional representation to a compact 10-dimensional representation. For decoding, we reverse the steps of encoding, ensuring that the dimensions of the data are restored to their original form. However, we introduce a sigmoid function at the end of the decoder to map the weights to the [0, 1] range. The forward function of the autoencoder first encodes the input data and then returns the decoded form.

Once the autoencoder is defined, we proceed to train it. This requires an optimizer and a loss function. We optimize the encoder parameters at each step of the training process and compute the loss in each epoch

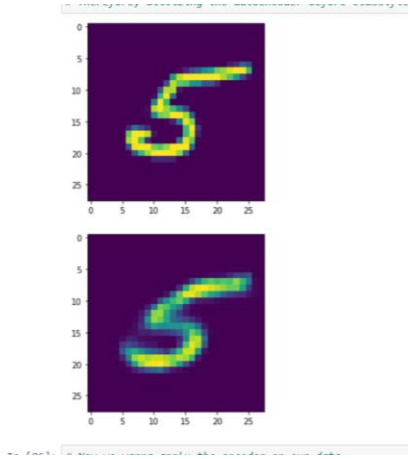
```
print( epoch : {}/{}, loss = {:.6f}
```

```
epoch : 1/50, loss = 0.000053
epoch : 10/50, loss = 0.000050
epoch : 20/50, loss = 0.000048
epoch : 30/50, loss = 0.000058
epoch : 40/50, loss = 0.000053
epoch : 50/50, loss = 0.000060
```

```
In [ ]: # Architectures and parameters are discusse
```

Visualize

After training our autoencoder, we can assess its performance visually. By applying the autoencoder to a sample image, we can observe that it successfully reconstructs the image with high fidelity. This indicates that the autoencoder has learned to accurately reproduce the input images. Additionally, we can infer that the 10-dimensional representation generated by the encoder captures the essential information necessary for image reconstruction. This suggests that the autoencoder effectively reduces the dimensionality of the input data while preserving important features.



Furthermore, it is worth noting that the reconstructed images exhibit denoising properties. Since the autoencoder was trained on diverse data samples, it has learned to reconstruct images by filtering out noise or variations present in the input data. This denoising capability can be highly advantageous in scenarios where noisy or distorted images need to be processed or analyzed.

Overall, the visual assessment of the autoencoder's performance highlights its ability to faithfully reconstruct images and extract informative features in a compact representation. These qualities make autoencoders valuable tools for tasks such as dimensionality reduction and denoising in various domains

Hyperparameters

The hyperparameters employed in our autoencoder architecture played a crucial role in its performance. The number of layers used in the encoder and decoder components determined the complexity and capacity of the model to capture and represent the input data. In our implementation, we utilized a five-layer architecture to progressively reduce the dimensionality of the data. This choice allowed us to extract essential features while maintaining a manageable latent space representation.

Regarding the activation function, we opted for the rectified linear unit (ReLU) activation function. ReLU has demonstrated excellent performance in neural networks, characterized by its computational efficiency and ability to mitigate the vanishing gradient problem. By employing ReLU, we ensured efficient training and effective feature extraction, as validated by subsequent classification experiments where ReLU yielded superior results.

Furthermore, the selection of the loss function and optimizer played a significant role in training the autoencoder. We employed the mean squared error (MSE) loss function, which is commonly used for continuous data. MSE measures the average squared difference between the reconstructed output and the original input, driving the autoencoder to minimize reconstruction error.

For optimization, we utilized the Adam optimizer, which is widely adopted in neural network training. Adam combines the benefits of both adaptive gradient algorithms and root mean square propagation. It adjusts the learning rate dynamically, improving convergence speed and robustness to different datasets.

By carefully selecting these hyperparameters, including the number of layers, activation function, loss function, and optimizer, we achieved an autoencoder architecture capable of effectively reducing dimensionality, extracting essential features, and reconstructing input data with high fidelity.

Feature extraction

As mentioned in the visualization part, our autoencoder can successfully take an image as input and generate a highly similar image as output. This indicates that the reduced dimensionality of the encoded representation contains sufficient information, as we can reconstruct the original image from it. Therefore, we can utilize this autoencoder as a dimension reducer. Furthermore, when using the autoencoder, we observe that the reconstructed images do not contain unnecessary details, resulting in a more generalized image. This is advantageous for prediction purposes.

Additionally, it is important to note that the dimension reduction achieved by the autoencoder is non-linear in nature. Unlike linear dimensionality reduction techniques such as Principal Component Analysis (PCA), which can only capture linear relationships in the data, autoencoders have the ability to capture complex and non-linear relationships. This non-linearity allows the autoencoder to extract more informative and discriminative features, enabling it to better represent the underlying structure of the data. As a result, the autoencoder's non-linear dimension reduction can be particularly advantageous for classification tasks that involve complex and non-linear patterns in the data.

Part II

Just like in previous assignments, we utilized a 5-fold approach to determine the optimal hyperparameters for our dataset. During this exercise, we experimented with these parameters to assess their effectiveness:

```
best_score = 0
for hidden_layer_size in [(100,), (100, 25), (80, 50, 20)]:
    for activations in ['relu', 'logistic', 'tanh', 'identity']:
        for learning_rate in ['constant','adaptive']:
```

In the end, the best parameters that yielded an accuracy of 0.95% were as follows.

```
0.9501111111111111
{'hidden': (100,), 'act': 'relu', 'learn': 'adaptive'}
```

Now, we need to train our MLP model using these parameters on the entire training dataset and subsequently test it on our test data. However, it is important to note that our training data is 10-dimensional. Therefore, for testing purposes, we should also reduce the dimensionality of our test data. To achieve this, we will encode the test data using the trained autoencoder. As a result, we will obtain a 10-dimensional test dataset, which we can then use to evaluate the performance of our model. The results of our MLP model are as follows:

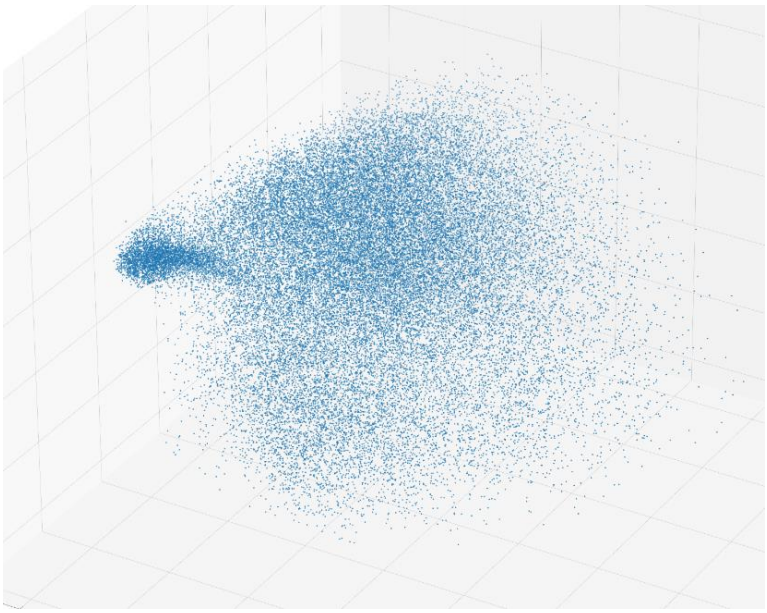
```
For Test Data
Accuracy :  0.9311
[[ 955    0    1    0    0    7   12    0    3    2]
 [   0 1110    3    1    6    1    4    3    6    1]
 [  10    3  958   19    6    3    5    9   16    3]
 [   2    3   11  931    0   14    1    8   34    6]
 [   1    2    1    0  909    3    8    5    6   47]
 [   7    1    5   24    5  798    9    1   33    9]
 [  12    3    1    0   13   12  911    0    2    4]
 [   0    3   11   10    4    1    1  947    4   47]
 [   1    4    4   27    8   41    2    6  871   10]
 [   4    7    2    6   33    4    2   10   20  921]]
For Train Data
Accuracy :  0.9752222222222222
[[1765    0    0    3    0    4    3    0    2    0]
 [   0 2025   11    3    1    1    1    1    1    2]
 [   6    0 1775    7    3    1    0    3    7    2]
 [   1    0    8 1750    3   13    0   10   40    7]
 [   0    5    3    0 1673    1    1    3    1   28]
 [   2    0    1   15    4 1532    8    1   10    4]
 [   2    2    1    1    1    7 1751    0    4    0]
 [   0    7    4    2    7    1    0 1825    1   26]
 [   3    3    5   31    3   27    5    2 1705    7]
 [   0    0    2   12   17    2    1   16   13 1753]]
```

As you can see, we achieved a high accuracy in our test dataset. Upon closer inspection, it becomes apparent that the recognition of the number 9 posed more challenges compared to other digits. The overall high accuracy in both the test and train datasets indicates that our model has been successful in capturing the underlying patterns and features of the data.

Part III

Clustering

The first challenge we encounter is selecting the appropriate clustering algorithm for our data. To gain insights into the structure of the data, we start by plotting it. To make the data visually understandable, we need to reduce its dimensionality to 3. This reduction allows us to create a 3D plot of the data. As you can observe, our data exhibits high density and complexity, which poses a significant challenge for clustering purposes.



The complexity and density of our data suggest that using the DBSCAN algorithm with small values for distance or the number of adjacencies may result in very large clusters. This can be problematic as we require a specific number of clusters (10). In DBSCAN, we do not manually specify the number of clusters, which can lead to an unpredictable number of clusters that may not align with our desired count. Consequently, we may need to merge or split clusters to achieve the desired number, which often produces unsatisfactory results.

On the other hand, the k-means algorithm only requires us to specify the number of clusters, which we already know in our case. The algorithm then handles the clustering process accordingly. This makes k-means a more suitable choice for our data compared to DBSCAN.

Additionally, fuzzy c-means is similar to k-means, but it considers the possibility of a point belonging to multiple clusters. However, since our objective is classification, we do not require points to be assigned to multiple clusters.

In summary, for our data, the k-means algorithm is preferable over DBSCAN or Fuzzy c-means.

Combined classification

To combine the data, we need to first transform the unlabeled data into a 10-dimensional representation using the autoencoder. Then, we can utilize the labels obtained from the k-means clustering as our new labels. Next, we combine this labeled unlabeled data with our original labeled training dataset. Finally, we apply the MLP model to perform classification. The results of this combined classification are as follows:

```
Training done
For Test Data
Accuracy :  0.142
[[679 207  75   9   9   0   0   0   1   0]
 [402 680  20  33   0   0   0   0   0   0]
 [665 308  46   1  12   0   0   0   0   0]
 [731 222  45   8   3   0   1   0   0   0]
 [718  66 128  61   7   2   0   0   0   0]
 [757 101  30   4   0   0   0   0   0   0]
 [343 238 348  28   1   0   0   0   0   0]
 [153  99 475 234  67   0   0   0   0   0]
 [882  73  17   2   0   0   0   0   0   0]
 [177  60 546 207  14   5   0   0   0   0]]
For Train Data
(28000, 10)
Accuracy :  1.0
[[2684    0    0    0    0    0    0    0    0    0]
 [   0 3248    0    0    0    0    0    0    0    0]
 [   0    0 2747    0    0    0    0    0    0    0]
 [   0    0    0 3652    0    0    0    0    0    0]
 [   0    0    0    0 2209    0    0    0    0    0]
 [   0    0    0    0    0 2615    0    0    0    0]
 [   0    0    0    0    0    0 2448    0    0    0]
 [   0    0    0    0    0    0    0 3984    0    0]
 [   0    0    0    0    0    0    0    0 2250    0]
 [   0    0    0    0    0    0    0    0    0 2163]]
```

We expected to achieve higher accuracy by incorporating additional data into our training dataset. However, we observed a significant decline in accuracy, which can be attributed to two main factors:

1. Clustering is not perfect: Clustering algorithms are not flawless, and inaccuracies in clustering can lead to incorrect classification and subsequently corrupt our training data.

2. Mapping clusters to digits: Even if the clustering was perfect, we would still face the challenge of determining which cluster corresponds to which digit. Without this mapping, it becomes difficult to assign the correct labels to the clusters.

As a result of these problems, especially the second one, our model performs poorly. It has achieved 100% accuracy on the training data because it has learned to fit the incorrect inputs. However, when it encounters real data, it struggles to make accurate predictions, resulting in poor performance on the test set.

To overcome these challenges, we can consider a couple of alternatives:

1. Combine labeled and unlabeled data before clustering: Instead of clustering the unlabeled data separately, we can combine it with the labeled data and perform clustering on the combined dataset. After clustering, we can assign labels to each cluster based on which class it predominantly represents.

2. Visual inspection of cluster samples: Another approach is to perform clustering on the unlabeled data and then visually inspect the images of randomly selected points within each cluster. By analyzing the main digit represented in each cluster, we can infer the corresponding class for that cluster.

By adopting these alternative methods, we can address the issues of incorrect labeling and uncertain cluster-to-digit mapping, leading to improved performance in our classification task.