Custom Autoencoder and Clustering on MNIST Dataset using ABO_clusterer

Ahnaf Bin Obaid
ID: 20201044

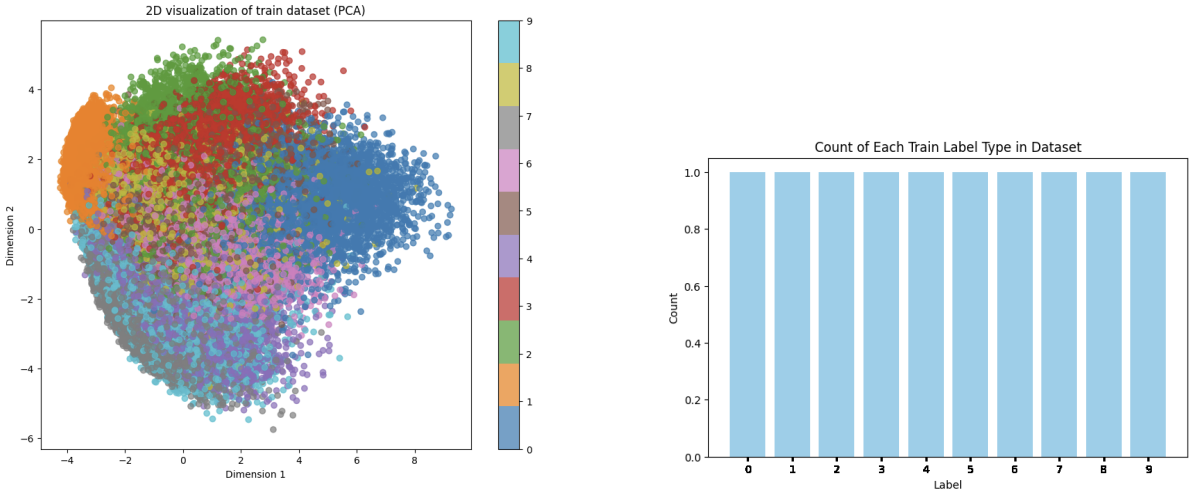
May 19, 2025

1 Dataset

1.1 Dataset Analysis

The dataset used in this project is the MNIST dataset, which consists of 60,000 grayscale images of handwritten digits (0–9) for training, and 10,000 images for testing. Each image is of size 28x28 pixels, flattened to 784 features when fed into the model. The dataset is well-balanced with equal samples per digit class, enabling effective training and evaluation of the clustering performance.

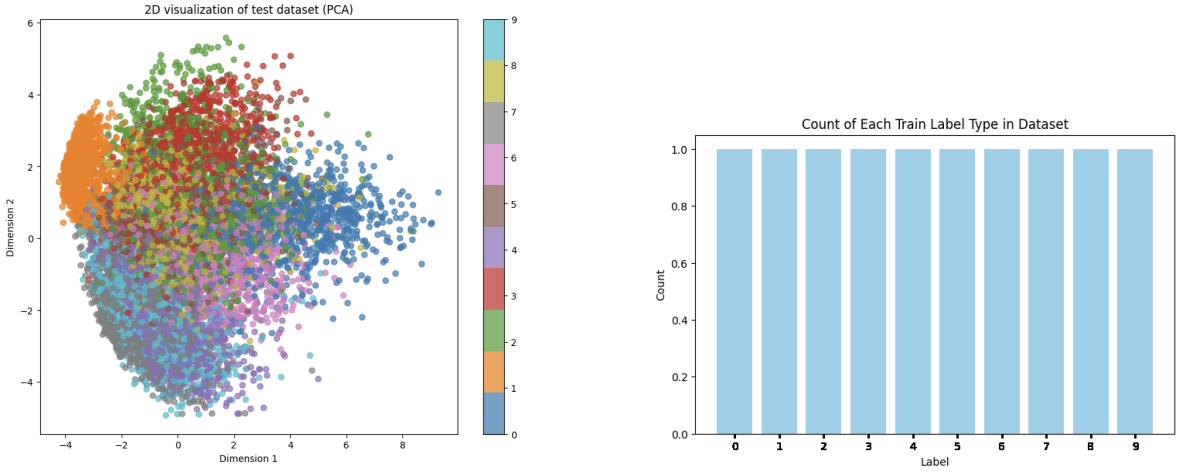
Figure 1: dataset images



(a) 2D visualization

(b) Count

Figure 2: Train dataset



(a) 2D visualization

(b) Count

Figure 3: Test Dataset

1.2 Dataset Pre-processing

The dataset preprocessing begins by loading the MNIST dataset using the `load_dataset` function, which provides both training and testing splits. To prepare the images for neural network input, a tensor transformation is applied using PyTorch's `transforms.ToTensor()` method, which converts the image data from PIL format to PyTorch tensors and normalizes pixel values to the $[0, 1]$ range. This transformation is implemented in the `transform_example` function, which takes an example from the dataset, applies the tensor conversion to the image, and retains the associated label. The function is then mapped over both the training and test subsets to ensure all images are transformed consistently. Finally, the dataset format is set to PyTorch tensors for the `image` and `label` columns, enabling seamless integration into PyTorch training pipelines and facilitating efficient batch processing during model training and evaluation.

2 Methodology

The goal is to create a custom clustering model(ABO_clusterer) which can be trained using unsupervisd data. After database pre-procesing we feed the data to the autoencoder and extract features of each image. The autoencoder will use training dataset images to train and extract features to get training embeddings. Following this we use test images to get test embeddings from the autoencoder. We get embeddings of both training data and testing data. We use the training embeddings to train the ABO_clusterer and then use the training embeddings to test the ABO_clusterer. Those same training and testing embeddings will be then used for getting clusters from KMeans which will then be used to compare against our custom model.

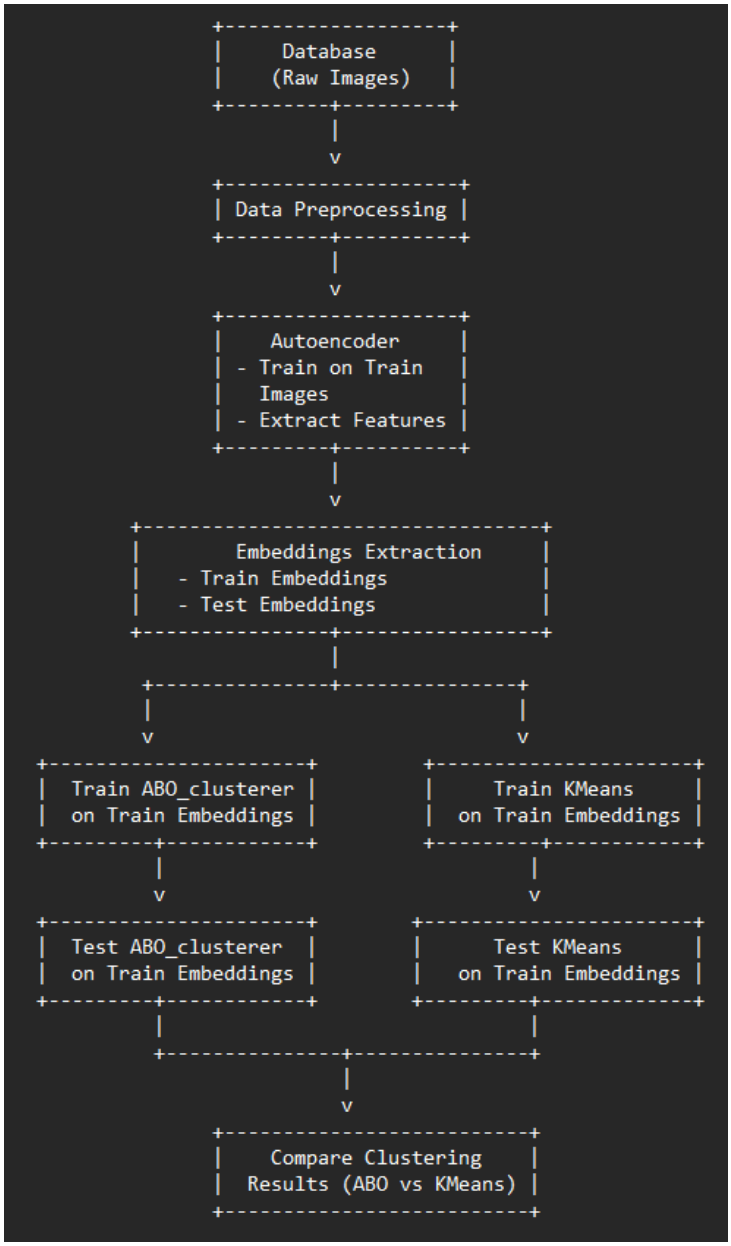


Figure 4: Methodolody

3 Models

3.1 Autoencoder

The simple yet effective autoencoder is designed to learn compressed representations of input images. The input consists of 28 by 28 pixel grayscale images from the MNIST dataset, which are first flattened into vectors of length 784. These vectors are passed through the encoder, which includes two fully connected linear layers. The first layer reduces the input dimension from 784 to 128 features, followed by a ReLU activation function. The second layer further compresses the representation down to a 64-dimensional latent space vector, which serves as the learned embedding or feature representation of the input. The decoder mirrors the encoder architecture by first expanding the latent vector back to 128 dimensions, applying another ReLU activation, and finally recon-

structing the original 784-dimensional input vector with a sigmoid activation to produce normalized pixel values. This architecture balances simplicity with sufficient capacity to capture meaningful features for downstream clustering tasks.

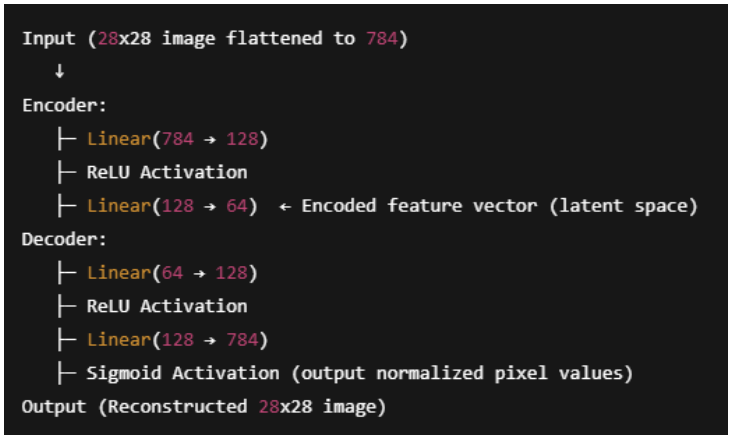


Figure 5: Block diagram of Autoencoder

Auto-encoder model:

AE(
encoder: Sequential(
(0) : Linear(in_features=784, out_features=128, bias=True)
(1) : ReLU()
(2) : Linear(in_features=128, out_features=64, bias=True)
)
decoder: Sequential(
(0) : Linear(in_features=64, out_features=128, bias=True)
(1) : ReLU()
(2) : Linear(in_features=128, out_features=784, bias=True)
(3) : Sigmoid()
)
)

Total parameters: 218192
Trainable parameters: 218192

3.2 ABO_clusterer

The ABO_clusterer is a custom clustering neural network module implemented using PyTorch. It is designed to learn cluster centers directly from the data features by optimizing a soft assignment of data points to clusters. Initialization of cluster centers is performed by selecting the data points furthest from the mean feature vector, which helps spread the initial centers across the feature space for better coverage. During the forward pass, the model computes the squared Euclidean distance between each input sample and the cluster centers, then applies a softmax over the negative distances to generate a probabilistic cluster assignment, encouraging samples to be assigned to the closest cluster. The

training loop optimizes the cluster centers by minimizing the expected distance between points and their assigned centers weighted by these soft assignments. This approach allows the model to adapt the cluster centers dynamically rather than relying on static initializations, making it more flexible for complex data distributions compared to traditional methods like KMeans. The trained cluster labels can then be extracted by taking the cluster with the highest soft assignment probability for each data point, enabling unsupervised categorization of the dataset.

ABO_clusterer model:
ABO_clusterer()
Total parameters: 640
Trainable parameters: 640

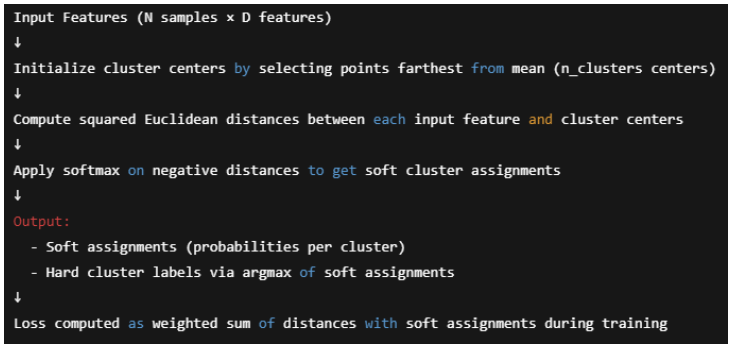
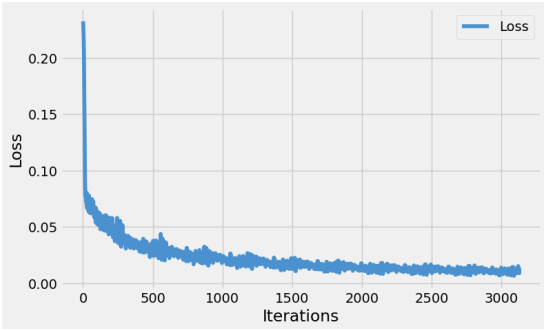


Figure 6: Block diagram of ABO_clusterer

4 Optimization and Hyperparameter Tuning

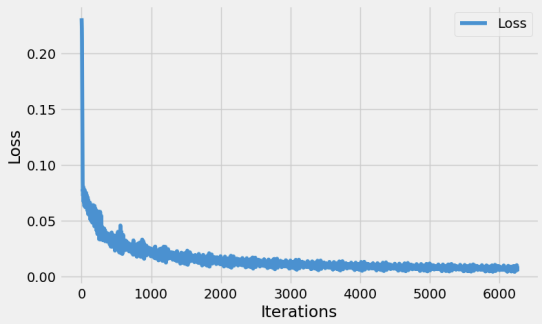
The autoencoder and clustering models were trained using Adam optimizer with a learning rate of 0.001 for the autoencoder and 0.1 for the custom clusterer. Hyperparameters such as number of clusters (set to 10), batch size (32), and number of epochs (200) were tuned based on empirical observations of loss convergence and clustering metrics such as Silhouette Score. The autoencoder was also run with different epochs. At 20 epochs to 30 epochs the loss was neglible so the epoch was set to 20. Lower epochs were also used, such as 10, but the decoded images at 20 epochs were much better than 10 epochs. Hence, 20 epochs was used for the Autoencoder.



(a) Loss curve

(b) Reconstruction

Figure 7: 10 epoch



(a) Loss curve

(b) Reconstruction

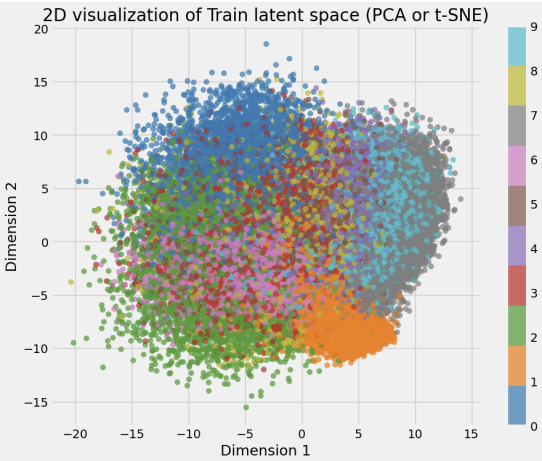
Figure 8: 20 epoch

5 Regularizations used

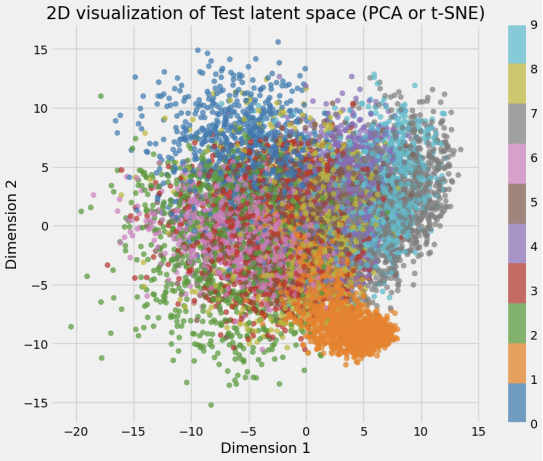
In this project, weight decay (L2 regularization) was applied through the Adam optimizer using a very small value of 1×10^{-8} to help prevent overfitting by penalizing large weights. The model architecture did not include batch normalization or dropout layers in order to maintain simplicity. However, incorporating batch normalization and dropout in future iterations could further improve the model's generalization ability by stabilizing and regularizing the training process.

6 Results

6.1 Embeddings



(a) Train



(b) Test

Figure 9: Visualization of latent Embedding Space

These embeddings, produced by the autoencoder, will be used to train and test ABO_clusterer and also KMeans. The same embeddings will be used for both to make a fair comparison of the results.

6.2 ABO_clusterer clusters

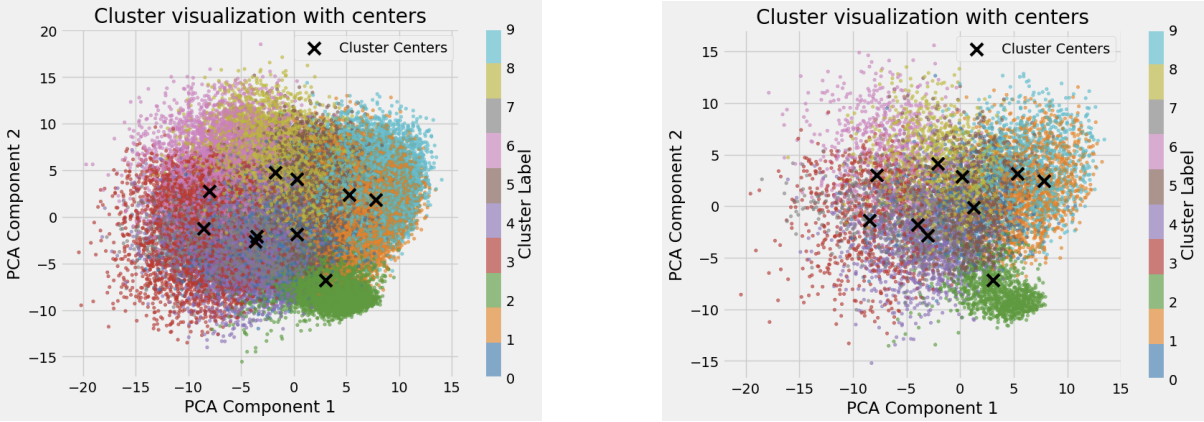
(a) Train (b) Test

Figure 10: Clustering visualization of ABO_clusterer

Table 1: Clustering Performance Metrics

Dataset	Silhouette Score	Davies-Bouldin Index	Calinski-Harabasz Index
Training	0.0868	2.5214	2760.4556
Testing	0.0871	2.5693	458.5000

6.3 KMeans clusters



(a) Train (b) Test

Figure 11: Clustering visualization of KMeans

Table 2: Clustering Performance Metrics

Dataset	Silhouette Score	Davies-Bouldin Index	Calinski-Harabasz Index
Training	0.0944	2.4995	2980.4283
Testing	0.0949	2.5091	499.0422

6.4 Comparison of Test Results between ABO_clusterer and KMeans

The testing results for the two clustering models show some differences in clustering quality metrics. The KMeans model achieves a higher Silhouette Score (0.0949) compared to ABO_clusterer (0.0871), indicating slightly better-defined clusters with more cohesive and well-separated groupings. Additionally, KMeans has a lower Davies-Bouldin Index (2.5091 versus 2.5693), which suggests improved cluster separation and compactness compared to ABO_clusterer. The Calinski-Harabasz Index for KMeans (499.0422) is also higher than that of ABO_clusterer (458.5000), further indicating that KMeans clusters are more distinct and better structured. Overall, the KMeans model demonstrates marginally superior clustering performance on the test dataset relative to the custom ABO_clusterer.

7 Limitations, Obstacles, and Future Work

One challenge encountered was initializing cluster centers effectively in the high-dimensional feature space, which was addressed by selecting initial centers farthest from the feature mean. Another limitation was the sensitivity to hyperparameters such as learning rate and cluster count, requiring careful tuning. Computational overhead was managed by batching. Future work includes incorporating more robust initialization and adaptive cluster numbers, building a better autoencoder for feature extraction, tuning the hyperparameters further to get closer to optimal results.

8 References

dataset: https://huggingface.co/datasets/ylecun/mnist
KMeans: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html
PyTorch: https://pytorch.org/