Neural Networks - Deep Learning Third Assignment

Dimitrios Karatis 10775, *Electrical and Computer Engineering, AUTH*

***Abstract*—This document provides an overview of the model training and evaluation process using the CIFAR-10 dataset. The focus is on leveraging Radial Basis Function Neural Networks (RBFNNs) to classify images into ten categories, such as airplanes, cars, and animals, making this a challenging yet foundational problem in computer vision.**

***Index Terms*—Radial Basis Function Neural Network, Autoencoder, deep learning, CIFAR-10, MNIST, image classification, computer vision.**

I. Introduction

TO effectively train and evaluate the RBFNN models presented in this work, the **CIFAR-10 Dataset** was selected as the primary data source (available at: CIFAR-10 Dataset). CIFAR-10 is a popular image classification dataset containing 60,000 32x32 color images divided into ten classes: airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. This dataset serves as a benchmark in computer vision and deep learning for evaluating classification model performance.

II. Dataset Description

The CIFAR-10 dataset consists of 50,000 training images and 10,000 test images, each labeled with one of the ten classes. Due to its diversity, it presents a complex classification problem and provides a comprehensive platform for testing and training CNN architectures in the context of image recognition.

A. Loading and Preparing the Dataset

To load the dataset, I used the ***cifar10.load_data()*** function from ***tensorflow.keras.datasets***. Additionally, a custom function, ***load_cifar***, uses this method to load the CIFAR-10 dataset while applying necessary preprocessing adjustments. The preprocessing steps, including normalization, rotation-based data augmentation, and PCA (Principal Component Analysis), are controlled by a matrix called ***preprocess***. Each element in this matrix determines whether a specific preprocessing step is applied (e.g., normalization is enabled if the first element is set to 1). Finally, the function outputs the preprocessed training, test, and validation datasets, as well as the number of PCA components needed to reach a certain variance.

This document was prepared by Dimitrios Karatis as part of the third assignment for the Neural Networks course at Aristotle University of Thessaloniki.

The ***custom_rotation*** function augments an image dataset by adding randomly rotated versions of the images, enhancing variability to improve model generalization. Each flattened image (shape (3072,)) is reshaped into its original 3D form (32, 32, 3), rotated by a random angle between -20 and 20 degrees (excluding 0), and then flattened back. The function collects these rotated images and their corresponding labels, concatenates them with the original dataset, and returns the augmented dataset. This process increases the dataset size and helps the model handle rotational variations in real-world scenarios, reducing overfitting and improving robustness.

The ***apply_pca*** function reduces the dimensionality of the input data using Principal Component Analysis (PCA) while preserving a specified proportion of the total variance. It takes training and testing datasets along with a target variance ratio as inputs, fits the PCA model to the training data, and applies the resulting transformation to both datasets. The function returns the transformed datasets and the optimal number of principal components selected to achieve the desired variance retention. This approach helps simplify the data, improving computational efficiency and potentially enhancing model performance by focusing on the most relevant features. The code for all the functions used to load and preprocess the dataset can be found at the ***dataset_functions.py***. Also, in the ***paths_config.py*** file, the user can change the necessary paths.

```
# Function to load cifar-10 data
# preprocess is matrxix indicating:
# [using normalize, using roatations, using PCA]
def load_cifar(self, preprocess):
    n_components = 0

    # Load the CIFAR-10 dataset
    (x_train, y_train), (x_test, y_test) =
        cifar10.load_data()

    # Flatten the image data for processing
    # Each image is reshaped from (32, 32, 3) to
        a single vector
    x_train = x_train.reshape(x_train.shape[0],
        -1)
    x_test = x_test.reshape(x_test.shape[0], -1)


    # Check if normalization is enabled
    # If enabled, scale pixel values to the
        range [0, 1] using MinMaxScaler
    if preprocess[0] == 1:
        scaler = MinMaxScaler()
        x_train = scaler.fit_transform(x_train)
        x_test = scaler.transform(x_test)

    # Flatten label arrays to make them
        compatible with training functions
```

```
y_train = y_train.ravel()
y_test = y_test.ravel()

# Check if rotation augmentation is enabled
# If enabled, apply a custom rotation
    function to the training data
if preprocess[1] == 1:
    x_train, y_train = self.custom_rotation(
        x_train, y_train)
    self.name = "rot.png"
else:
    self.name = "no_rot.png"

# Check if PCA is enabled
# If enabled, apply PCA to reduce
    dimensionality
if preprocess[2] != 0.0:
    x_train, x_test, n_components = self.
        apply_pca(x_train, x_test,
        preprocess[2])
    self.name = "pca_" + self.name
else:
    self.name = "no_pca_" + self.name


# Return the preprocessed training and test
    datasets
return x_train, y_train, x_test, y_test ,
    n_components
```

B. Training and Evaluating Radial Basis Function Neural Network (RBFNN)

To address the challenges posed by the high-dimensional and complex nature of the CIFAR-10 dataset, I implemented a Radial Basis Function Neural Network (RBFNN) for classification tasks, leveraging radial basis functions as activation units. I designed the network to use either K-means clustering or random selection to determine the centers, which serve as key reference points for the Gaussian kernels. The spreads (sigmas) of these kernels are calculated based on the distances between the centers, ensuring the network captures the variability in the data effectively. To optimize the weights, I employed a pseudo-inverse solution, allowing the model to efficiently map inputs to outputs. I also incorporated functionality to evaluate how the number of centers affects test accuracy, by plotting accuracies versus number of centers. The code for my RBFNN implementation can be found in the ***RBFNN.py*** file and the corresponding main function in the ***main_RBFNN.py.***

Key points of my implementation:

- I used the pseudo-inverse approach to compute the weights because it provides a straightforward and efficient way to solve the linear system G.W=Y, where G is the interpolation matrix, W are the weights, and Y is the target output.
- K-means clustering was chosen as one of the center selection methods because it ensures that the centers are representative of the data's distribution. This reduces redundancy and enhances the model's ability to generalize
- The sigma values (widths of the Gaussian kernels) were computed as the mean of the distances between a center and its neighbors. This ensures that each radial basis function has an appropriate spread, capturing the local characteristics of the data.
- By systematically varying the number of centers and observing its impact on test accuracy, I aimed to understand the trade-off between model complexity and performance. This analysis provides insights into selecting the optimal number of centers for a given dataset.

Below are the results for different number of centers and different center selection methods (kmeans and random):

Detailed results, along with examples of both correct and incorrect classifications, can be found in the ***results_txt*** *folder.*

RANDOM CENTER SELECTION:

At first I used a preprocess matrix of [1, 0, 0.0] meaning only normalization will be used. The results obtained using the RBFNN with random center selection for CIFAR-10 classification highlight the relationship between the number of centers, training accuracy, and test accuracy. As the number of centers increases, both training and test accuracy improve, although with diminishing returns. For example, with just 10 centers, the model achieves a test accuracy of 28.85% in 5.39 seconds. Increasing the centers to 100 raises the test accuracy to 41.12%, with a more substantial training accuracy of 40.37%, at the cost of increased computation time (54.42 seconds). As the number of centers reaches 1000, test accuracy peaks at 49.03%, while training accuracy improves to 52.40%; however, the elapsed time for this configuration significantly rises to 554.89 seconds. These trends indicate that increasing the number of centers enhances the model's capacity to represent the data but comes with a computational cost. Moreover, we can see that by gradually increasing the number of centers, overfitting starts to occur. The use of random centers appears effective for moderately sized configurations but may struggle to fully capture the complexity of CIFAR-10 when the number of centers is too low, as evidenced by the relatively low initial accuracy values. Below we can see a diagram of how the number of centers affected the accuracies of the model.

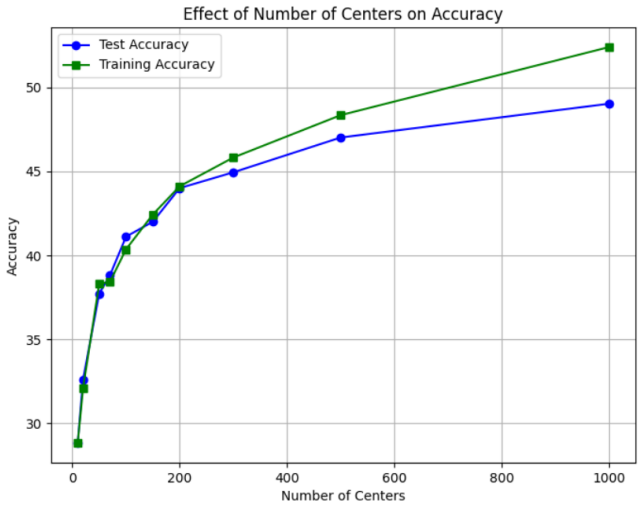

Fig. 1. Accuracies vs num of centers (NO PCA and NO rotations)

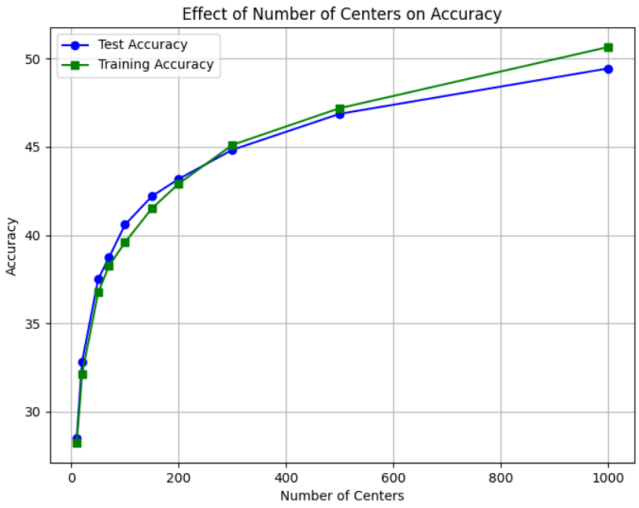

Fig. 2. Accuracies vs num of centers (NO PCA, only rotations)

Next, I tried a preprocess matrix of [1, 1, 0.0], meaning that now rotations are enabled.

The results obtained using the RBFNN with random center selection and data augmentation through rotations (along with normalization) demonstrate a consistent trend of improved accuracy as the number of centers increases, though at the expense of higher computational time. With 10 centers, the test accuracy is 28.52% and training accuracy is 28.23%, achieved in 10.66 seconds. As the number of centers grows, the model's capacity to learn complex patterns improves, resulting in a test accuracy of 40.59% and training accuracy of 39.59% at 100 centers, with an elapsed time of 109.85 seconds. When 1000 centers are used, the test accuracy peaks at 49.44%, with the training accuracy reaching 50.65%, although the computational cost is significant in that case (1100.72 seconds). Comparing these results to the non-rotated data reveals that incorporating rotations as a data augmentation technique provides a modest improvement in both training and test accuracy across all configurations while also helping the model's ability to generalize. However, as with the non-augmented case, diminishing returns in accuracy improvements are observed beyond 200 centers, emphasizing the need to balance model complexity and computational efficiency. Below we can see a diagram of how the number of centers affected the accuracies of the model.

The last thing I tried was a preprocess matrix of [1, 1, 0.95], meaning that now rotations and PCA (with 95% of the original data's variance) are enabled.

PCA reduced the original input dimensionality to 209 components, significantly decreasing computation time across all configurations while maintaining comparable or slightly better accuracy than the non-PCA results. For instance, with 10 centers, the test accuracy was 28.39%, achieved in just 0.55 seconds, compared to 10.66 seconds without PCA. As the number of centers increased, both training and test accuracies improved steadily. At 200 centers, the model achieved a training accuracy of 43.21% and test accuracy of 43.61%, in just 11.62 seconds, indicating an optimal balance between complexity and efficiency. The highest test accuracy of 53.75% was observed with 3000 centers, where training accuracy reached 59.08%, much higher than the test accuracy, indicating possible overfitting. Although, for 1000 centers the model reached a memorable taining accuracy of 51.41% and a test accuracy of 50.08% slightly better than before. Below we can see a diagram of how the number of centers affected the accuracies of the model.

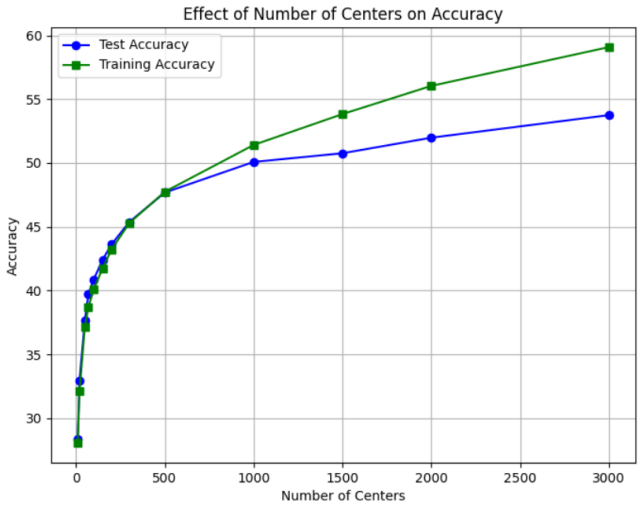

Fig. 3. Accuracies vs num of centers (PCA and rotations)

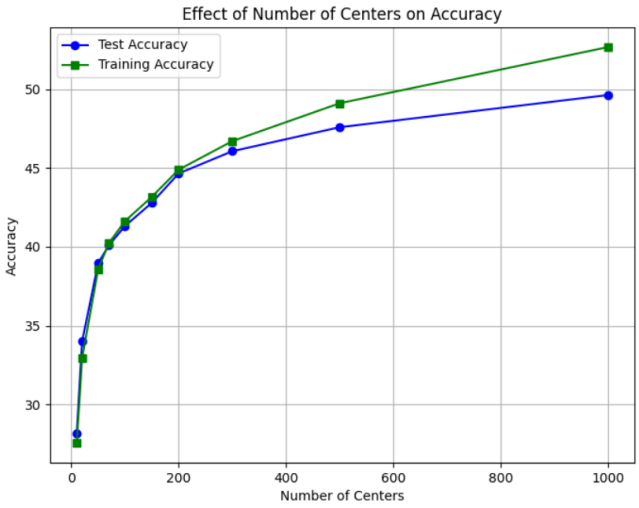

Fig. 4. Accuracies vs num of centers (NO PCA, NO rotations)

K-MEANS CENTER SELECTION:

Next, I experimented by using the K-means clustering algorithm in order to find representative centers of the data's distribution. Using the K-means clustering algorithm to determine the centers for the RBFNN, combined with normalization but without PCA or rotational augmentation, reveals a gradual improvement in both training and test accuracies as the number of centers increases. At 10 centers, the test accuracy was 28.18% with an elapsed time of 11.63 seconds. This baseline performance is similar to the results obtained with random center selection under similar conditions. However, as the number of centers grows, the K-means approach begins to outpace the random selection method in accuracy while maintaining alomst the same computational complexity. For instance, with 200 centers, the model achieved a test accuracy of 44.65% and training accuracy of 44.89%, in 150.98 seconds.

The increasing accuracy trend continues with 500 centers (test accuracy: 47.58%) and peaks at 1000 centers, achieving a training accuracy of 52.66% and a test accuracy of 49.62%. However, the computational cost also grows substantially, with an elapsed time of 673.45 seconds at 1000 centers. These results demonstrate that K-means clustering effectively identifies representative centers that improve model performance. Compared to random center selection, K-means clustering provides a more structured approach, particularly beneficial at moderate center counts (e.g., 200–500), where it strikes a balance between accuracy and efficiency. Below we can see a diagram of how the number of centers affected the accuracies of the model.

Next, I tried a preprocess matrix of [1, 1, 0.0], meaning that now only rotations are enabled.

Incorporating rotational augmentation into the dataset while using the k-means clustering algorithm to determine centers resulted in notable performance improvements, that came with much higher computational times. At 10 centers, the test accuracy of 28.77% and training accuracy of 28.30% (elapsed time: 46.66 seconds) showed a slight improvement over the non-rotated results, suggesting that rotational augmentation helps capture additional variability in the data. As the number of centers increased, the model achieved progressively better accuracies, with the most substantial gains observed up to 500 centers. For example, with 200 centers, the training accuracy was 44.52%, and the test accuracy reached 44.81% (elapsed time: 334.33 seconds).

The improvements were particularly noticeable at larger center counts, with 500 centers achieving a test accuracy of 48.27% (elapsed time: 783.29 seconds), up from 47.58% in the non-rotated case. The peak performance was observed with 1000 centers, achieving a training accuracy of 51.63% and a test accuracy of 50.44% in 1442.03 seconds. However, using more centers with rotation turned out to be significantly demanding computationally. So, next I decided to experiment using PCA. Below we can see a diagram of how the number of centers affected the accuracies of the model.

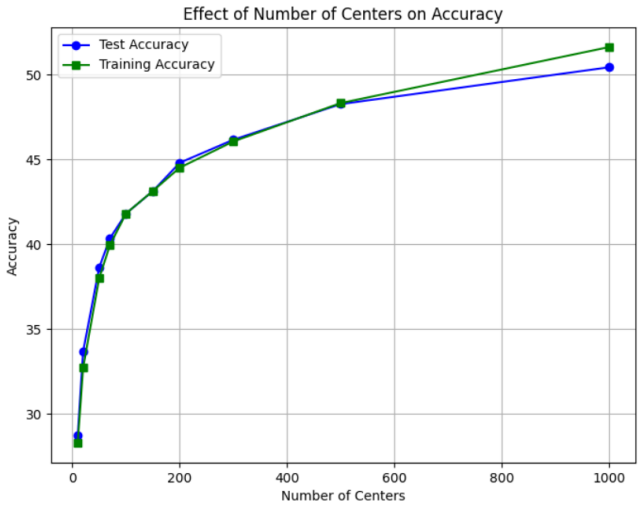

Fig. 5. Accuracies vs num of centers (NO PCA, only rotations)

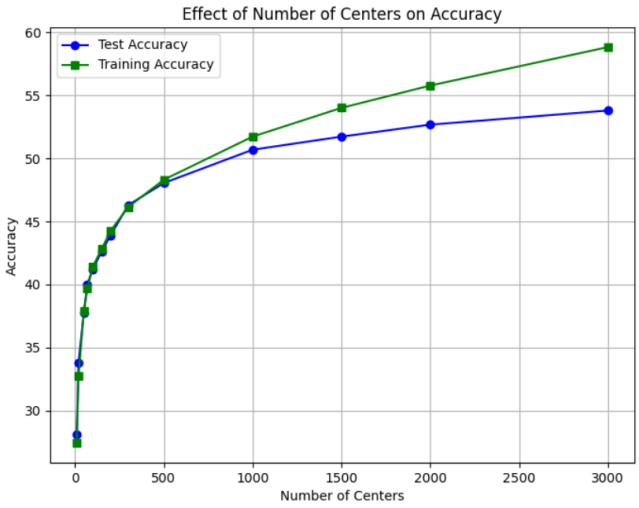

Fig. 6. Accuracies vs num of centers (PCA, and rotations)

The new preprocess matrix I tried was [1, 1, 0.95], meaning that now rotations and PCA (with 95% of the original data's variance) are enabled.

Introducing Principal Component Analysis (PCA) to reduce dimensionality, combined with rotational augmentation, significantly enhanced both the computational efficiency and the performance of the K-means clustering algorithm. PCA reduced the dataset to 209 components while preserving 95% of the variance, resulting in faster processing times. For example, with 10 centers, the elapsed time was 2.50 seconds, drastically shorter than the 46.66 seconds observed with rotation alone. Similarly, for 1000 centers, the elapsed time was 86.89 seconds compared to 1442.03 seconds without PCA.

The combination of PCA and rotation also improved model accuracy. At 10 centers, the test accuracy was 28.15%, slightly lower than with rotation alone. However, as the number of centers increased, the advantages of PCA became apparent. For instance, at 200 centers, the test accuracy was 43.89%, closely aligning with the results of previous experiments, but with substantially reduced computation time (22.00 seconds compared to 334.33 seconds without PCA). The trend continued, with 3000 centers achieving the highest test accuracy of 53.80% in 277.55 seconds, and training accuracy of 58.83%, a sign of overfitting. Although, for 1000 centers the model reached a memorable taining accuracy of 51.73% and a test accuracy of 50.69% slightly better than before. Below we can see a diagram of how the number of centers affected the accuracies of the model.

Overall, the addition of rotations proved beneficial for improving the RBFNN's ability to generalize, particularly in cases with moderate to high numbers of centers. PCA also significantly reduced the computational burden while maintaining or slightly improving accuracy, especially at higher numbers of centers. This highlights its effectiveness as a preprocessing step when combined with rotation, particularly for large-scale datasets where computational resources are a consideration.

C. Training and Evaluating Classifiers

Now we need to compare the performance of the RBFNN to different classifiers. In order to do that, I fisrt initialized the classifiers, and then I trained every single one of them to a set of training images with corresponding labels. After training, the classifiers are evaluated on a test set, where they predict labels for the test images and compute accuracy based on these predictions. The initialization, training and evaluation has been done by using the KNeighborsClassifier and NearestCentroid packages from sklearn. The code for the classifiers implementation can be found in the ***classifier.py*** file and the corresponding main function in the ***main_classifiers.py.***

First I only normalized the pixel values of the dataset and the results were:

```
k-NN (k=1) Accuracy Normalized: 35.39%, train_acc: 100.00%
Elapsed Time: 65.31 seconds
k-NN (k=3) Accuracy Normalized: 33.03%, train_acc: 57.90%
Elapsed Time: 64.91 seconds
Nearest Centroid Accuracy Normalized: 27.74%, train_acc: 26.97%
Elapsed Time: 2.16 seconds
```

Fig. 7. Results for different classifiers (only normalized))

After that, I tried applying rotations and PCA to the dataset. The results were:

```
k-NN (k=1) Accuracy Normalized (Augmented): 37.10%, train_acc: 100.00%
Elapsed Time: 24.64 seconds
k-NN (k=3) Accuracy Normalized (Augmented): 36.68%, train_acc: 93.27%
Elapsed Time: 24.65 seconds
Nearest Centroid Accuracy Normalized (Augmented): 27.63%, train_acc: 26.74%
Elapsed Time: 0.52 seconds
```

Fig. 8. Results for different classifiers (using normalization and rotations)

Introducing PCA and rotational augmentation to the CIFAR-10 dataset in conjunction with a k-NN classifier (k=1, k=3) and a nearest centroid classifier demonstrated notable

differences in both computational efficiency and classification performance. For k-NN with k=1, the addition of PCA and rotations improved the test accuracy from 35.39% to 37.10%, while significantly reducing the elapsed time from 65.31 seconds to 24.64 seconds. Similarly, for k=3, the test accuracy increased from 33.03% to 36.68%, with the training accuracy also rising from 57.90% to 93.27%, indicating that the augmented dataset better captured variations within the data, thus improving the classifier's ability to fit the training set. The nearest centroid classifier, in contrast, showed marginal changes in performance, with the test accuracy dropping slightly from 27.74% to 27.63% and training accuracy decreasing from 26.97% to 26.74%. However, the computational efficiency for the nearest centroid classifier improved significantly, with the elapsed time reduced from 2.16 seconds to just 0.52 seconds.

Comparing the results of the k-NN and nearest centroid classifiers to the RBFNN with the K-means center method (using PCA and rotations) reveals notable differences in accuracy, computational efficiency, and scalability. The RBFNN with K-means centers achieved significantly higher accuracy, with test accuracies ranging from 28.15% to 50.69%, depending on the number of centers, whereas the k-NN classifier (k=1) reached a maximum of 37.10%, and the nearest centroid classifier plateaued at 27.63%. This demonstrates that the RBFNN can model complex decision boundaries more effectively when equipped with a sufficiently large number of centers. In terms of computational efficiency, while the RBFNN showed increasing elapsed times as the number of centers grew (e.g., 277.55 seconds for 3000 centers), it was still faster for lower center counts compared to the unoptimized k-NN (e.g., 2.50 seconds for 10 centers in RBFNN vs. 24.64 seconds for k-NN). PCA also had a more pronounced effect on the RBFNN, reducing dimensionality to 209 components while maintaining high accuracy.

Overall, the RBFNN, especially with k-means centers, strikes a better balance between accuracy and flexibility, while k-NN and nearest centroid classifiers remain simpler, less tunable options with limited potential for improvement through augmentation and dimensionality reduction.

D. Training and Evaluating Autoencoder model

As a final task, I decided to implement an autoencoder model that generates an image of the next digit based on a handwritten digit image from the MNIST dataset. For instance, if the input is a 3, the model will produce an image of a 4 etc.

The autoencoder was created to learn the relationship between an input digit image and its corresponding next digit image in the MNIST dataset. The development process began with defining the model architecture, which consists of two key components: an encoder and a decoder. The encoder compresses the input image into a lower-dimensional latent space by applying convolutional layers, pooling, and a dense layer to capture meaningful features. The decoder then reconstructs the image of the next digit using transposed convolutional layers and reshaping techniques, ensuring the output image closely resembles the target. The model was trained on normalized and reshaped images from the MNIST dataset, where each input image corresponds to a target image representing the next digit. By minimizing the mean squared error loss between the generated and target images, the autoencoder learned to map digit images to their successors. After training, visualization techniques were applied to evaluate its performance, showcasing the generated next-digit images alongside their inputs. The code for the autoencoder implementation can be found in the ***autoencoder.py*** file and the corresponding main function in the ***main_autoencoder.py.***. Below we can see the validation results as well as a diagram of the losses throughout the epochs.

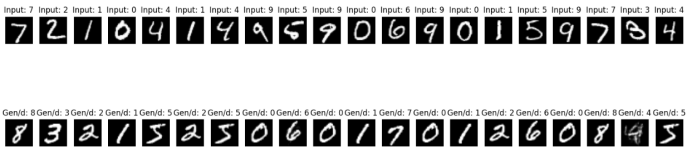

Fig. 9. Test Results

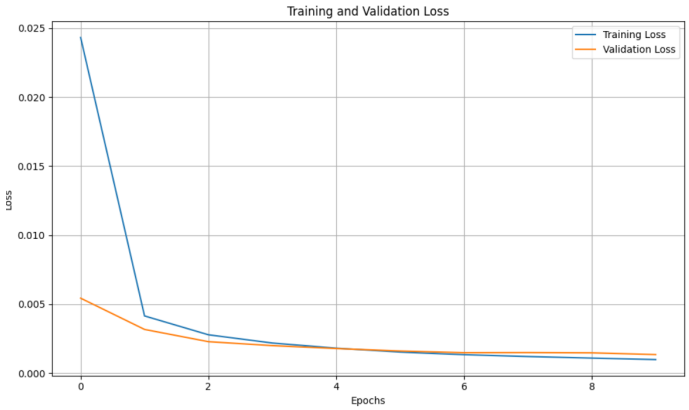

Fig. 10. Losses vs epochs

The training and validation loss curves in the diagram show that the autoencoder performs effectively. The training loss decreases significantly during the first few epochs, indicating that the model is learning to minimize the reconstruction error quickly. Additionally, the validation loss closely follows the training loss and stabilizes by the end of training. This close alignment suggests that the model generalizes well to new, unseen data and is not overfitting. These results demonstrate that the autoencoder successfully learns the relationship between input digit images and their corresponding next digit images, making it suitable for the intended task.

Finally, I decided to train my RBFNN on the MNIST dataset to see if the network could recognize the reconstructed digits generated by the autoencoder. For simplicity, I skipped using PCA and rotation during training and focused only on normalizing the pixel values.

```
For centr_method = kmeans and num_centers = 1000 :
Elapsed Time: 313.18 seconds
Training Accuracy: 96.81%
Test Accuracy: 96.51%
Predicted Labels: ['7', '2', '1', '0', '4', '1', '4', '9', '6', '9']
Actual Labels:    ['7', '2', '1', '0', '4', '1', '4', '9', '5', '9']

Test Accuracy: 98.63%
Predicted Labels: ['8', '3', '2', '1', '5', '2', '5', '0', '6', '0']
Actual Labels:    ['8', '3', '2', '1', '5', '2', '5', '0', '6', '0']
```

Fig. 11. RBFNN tested on autoencoder results

As we can see, by training the RBFNN with 1,000 centers selected using the K-means method, we achieved a very high training accuracy, which is promising. When feeding the reconstructed images from the autoencoder into the network, the model achieved a test accuracy of 98.63%, indicating that the RBFNN effectively recognizes the reconstructed digits. However, it's important to note a limitation: we don't know the "Actual Labels" of the images generated by the autoencoder. Therefore, the reported test accuracy of 98.63% might be misleading.

To clarify, if we feed images of digits 1, 2, and 3 into the autoencoder, the expected output would ideally be images of the next digits, such as 2, 3, and 4. In this case, the actual labels would be ['2', '3', '4']. However, due to factors like noise, the autoencoder might instead generate images of 2, 3, and 3, making the actual labels ['2', '3', '3']. Since we can't determine the actual labels with certainty, this creates ambiguity.

A better approach than relying solely on the "Test Accuracy" to tell if the RBFNN can recognize the digits from the autoencoder, would be to visualize the generated images (as I've done previously) and manually verify if the predicted labels align with the reconstructed digits. This would provide a more accurate evaluation of the model's performance.