NEURAL NETWORKS - DEEP LEARNING THIRD ASSIGNMENT

Dimitrios Diakoloukas 10642, Electrical and Computer Engineering, *AUTH, Student*

***Abstract*—This study implements an Autoencoder and a Radial Basis Function Neural Network (RBFNN) on the CIFAR-10 dataset for image reconstruction and classification. PCA was applied for dimensionality reduction, preserving 90% of data variability. The Autoencoder demonstrated effective image reconstruction, while the RBFNN achieved competitive classification performance. Metrics such as accuracy, reconstruction error, and confusion matrices were analyzed, with comparisons to PCA, Nearest Neighbor, and Nearest Class Centroid classifiers. Results highlight the strengths of these methods and the impact of hyperparameter tuning on performance.**

I. Introduction

IMAGE classification is an essential task in computer vision, used in areas like autonomous driving and facial recognition. The CIFAR-10 dataset, a widely recognized dataset, presents us with the problem of classifying images into 10 distinct categories, each of which has natural objects such as animals and vehicles. Autoencoders and Radial Basis Function Neural Networks (RBFNNs), known for their ability to handle complex data representations, are explored in this study for image reconstruction and classification. The main point of this assignment is to experiment with these neural network-based approaches adjusting their hyperparameters and compare them to traditional classifiers.

II. Data Loading and Preprocessing

This section explains how I loaded the CIFAR-10 dataset and prepared it so that it could be used with the Radial Basis Function Neural Networks (RBFNNs), Autoencoder implementations but also (k-NN) algorithm for testing purposes.

A. *Loading CIFAR-10 Data*

The CIFAR-10 dataset contains 60,000 small color images (32x32 pixels) across 10 different classes. There are 50,000 images for training and 10,000 images for testing. Each image is stored as a set of pixel values and the data is divided into multiple files.

I used the following steps to load the data:

- **Unpacking Data Files:** The training data is stored in five files, each containing 10,000 images. I created a helper function called unpickle to open each file and load the data inside.
- **Combining Training Files:** I combined the data from all five training files into one large set. This gave us a total of 50,000 images for training. I did the same for the labels, so each image has a label showing which class it belongs to.
- **Loading Test Data:** The test data is stored in a separate file. I loaded this file in the same way to get 10,000 test images and their labels.

B. *Data Preprocessing*

After loading the data, I prepared it for use with machine learning by following these steps:

- **Reshaping Images:** Each image was originally stored as a long row of numbers (a flat array). I reshaped these numbers back into the original 32x32 size with 3 color channels (Red, Green, and Blue).
- **Normalizing Pixel Values:** The pixel values in each image were originally between 0 and 255. To make the data easier to work with, I scaled these values to be between 0 and 1 by dividing each value by 255. This helps the model perform better and faster.

The final output of our data loading and preprocessing is:

- `x_train`: A set of 50,000 training images, each with shape (32, 32, 3) and pixel values between 0 and 1.
- `y_train`: A set of 50,000 labels for the training images.
- `x_test`: A set of 10,000 test images, also with shape (32, 32, 3) and pixel values between 0 and 1.
- `y_test`: A set of 10,000 labels for the test images.

This data is now ready to be used in our Radial Basis Function Neural Networks and Autoencoders (based on CNN) as well as the k-NN algorithm (for comparisons), making it easy to classify each image based on its pixel values.

III. Initial Implementation of Radial Basis Function Neural Network (RBFNN) (Trainable RBF Network)

In my initial experiment with for this assignment I implemented trained and evaluated a **Radial Basis Function Neural Network (RBFNN)** on the CIFAR-10 dataset while experimenting with various configurations and hyperparameters. The following describes how the specific asked tasks of in the exercise were addressed:

A. *Data Preparation*

- **Dataset:** The CIFAR-10 dataset, consisting of 60,000 images classified into 10 categories, was used. It was loaded using a custom function `load_cifar10_data`.

- **Normalization:** Pixel values of the images were normalized to the range [0, 1] to ensure faster and more stable convergence during training.
- **Dimensionality Reduction:** Principal Component Analysis (PCA) was applied to the flattened image data to reduce its dimensionality to 300 components while preserving approximately 90% of the variance. This step addresses the exercise requirement for feature extraction and computational efficiency.

B. *RBF Neural Network Layer Implementation*

- A custom `RBFLayer` was implemented using TensorFlow/Keras to model the Radial Basis Function neurons.
- The layer computes outputs using the Gaussian activation function, which is based on the squared distances between inputs and predefined centers.
- **Centers Initialization:** Centers were initialized using either:
  - *KMeans clustering:* Cluster centroids were used as RBF centers.
  - *Random Initialization:* Centers were chosen randomly.
- The `gamma` hyperparameter was used to control the width of the Gaussian functions.

C. *RBFNN Model Architecture*

The RBFNN model consisted of:

- A dense input layer to process the PCA-reduced features.
- A custom RBF layer to map inputs into the RBF space.
- A dense output layer with a softmax activation function for classification into 10 categories.

D. *Training and Evaluation*

- **Training:** The RBFNN was trained using the Adam optimizer and categorical crossentropy loss for 50 epochs. Different configurations were explored by varying:
  - *Number of RBF centers:* 100, 300, and 500 centers.
  - *Centers Initialization:* Random or KMeans.

  Training time was recorded to evaluate the computational efficiency.
- **Evaluation:** The following metrics were used to assess the model's performance:
  - Accuracy and classification reports on the test set.
  - Confusion matrices to analyze class-wise performance.
- **Cross-validation:** Although I didn't explicitly implement this method, training and testing data splitting ensured robust evaluation of the model.

E. *Results and Visualization*

The performance of the RBFNN was evaluated for different configurations, including the number of RBF centers (100, 300, 500) and the method of center initialization (KMeans or Random). Below are the detailed results for each configuration:

1) *Number of Centers: 100:*

- **Centers Initialization:** KMeans
- **Training Time:** 353.06 seconds
- **Accuracy:** 51.19%

Classification Report:

```
              precision    recall  f1-score   support

           0       0.56      0.59      0.58      1000
           1       0.60      0.65      0.62      1000
           2       0.41      0.31      0.35      1000
           3       0.37      0.29      0.32      1000
           4       0.47      0.40      0.44      1000
           5       0.47      0.36      0.41      1000
           6       0.44      0.69      0.54      1000
           7       0.52      0.64      0.58      1000
           8       0.65      0.62      0.63      1000
           9       0.57      0.57      0.57      1000

    accuracy                           0.51     10000
   macro avg       0.51      0.51      0.50     10000
weighted avg       0.51      0.51      0.50     10000
```

2) *Number of Centers: 300:*

- **Centers Initialization:** KMeans
- **Training Time:** 850.57 seconds
- **Accuracy:** 52.68%

Classification Report:

```
              precision    recall  f1-score   support

           0       0.67      0.46      0.55      1000
           1       0.72      0.58      0.64      1000
           2       0.40      0.41      0.40      1000
           3       0.40      0.28      0.33      1000
           4       0.39      0.56      0.46      1000
           5       0.48      0.41      0.44      1000
           6       0.55      0.61      0.58      1000
           7       0.65      0.56      0.60      1000
           8       0.59      0.71      0.65      1000
           9       0.52      0.68      0.59      1000

    accuracy                           0.53     10000
   macro avg       0.54      0.53      0.52     10000
weighted avg       0.54      0.53      0.52     10000
```

3) *Number of Centers: 500:*

- **Centers Initialization:** KMeans
- **Training Time:** 1340.94 seconds
- **Accuracy:** 52.09%

Classification Report:

```
              precision    recall  f1-score   support

           0       0.60      0.62      0.61      1000
           1       0.73      0.55      0.63      1000
           2       0.48      0.27      0.35      1000
           3       0.37      0.28      0.32      1000
           4       0.36      0.61      0.45      1000
           5       0.48      0.37      0.42      1000
           6       0.43      0.78      0.55      1000
           7       0.77      0.46      0.58      1000
           8       0.70      0.62      0.66      1000
           9       0.57      0.66      0.61      1000

    accuracy                           0.52     10000
   macro avg       0.55      0.52      0.52     10000
weighted avg       0.55      0.52      0.52     10000
```

4) *Number of Centers: 500 (Random Initialization):*

- **Centers Initialization:** Random
- **Training Time:** 1351.35 seconds
- **Accuracy:** 10.00%

Classification Report:

```
              precision    recall  f1-score   support

           0       0.00      0.00      0.00      1000
           1       0.00      0.00      0.00      1000
           2       0.00      0.00      0.00      1000
           3       0.00      0.00      0.00      1000
           4       0.10      1.00      0.18      1000
           5       0.00      0.00      0.00      1000
           6       0.00      0.00      0.00      1000
           7       0.00      0.00      0.00      1000
           8       0.00      0.00      0.00      1000
           9       0.00      0.00      0.00      1000

    accuracy                           0.10     10000
   macro avg       0.01      0.10      0.02     10000
weighted avg       0.01      0.10      0.02     10000
```

These results highlight the impact of the number of centers and the initialization method on training time and classification accuracy. Models with centers initialized using KMeans performed significantly better than those with random initialization, achieving an accuracy of up to 52.68% for 200 centers.

5) *Overall Observations*:

- The use of KMeans for center initialization consistently outperformed random initialization across all configurations. I can also add here that the result when I used random initialization seemed to be close to random guessing.
- Increasing the number of centers improved accuracy up to a point (lets say about 300 centers), after which the performance slightly decreased or maybe remained the same.
- Training time increased significantly with the number of centers, highlighting the trade-off between computational cost and model performance.
- Proper initialization and tuning of hyperparameters, such as the number of centers and the gamma value, are critical for achieving optimal results with RBFNNs.

Below I also provide the confusion matrixes for visualization of the results and also the training and validation accuracy and losses throughout the epochs.

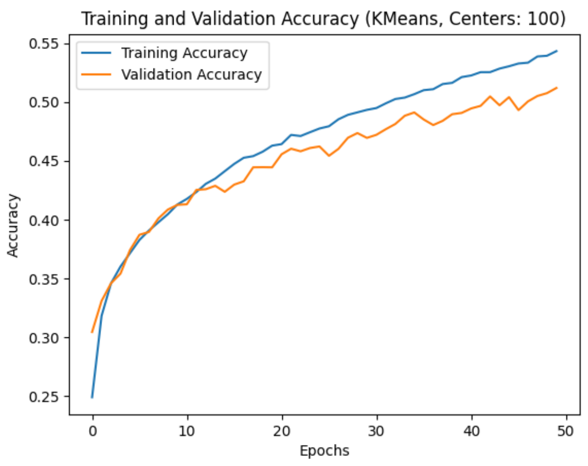


Fig. 2. Training and Validation accuracy throughout epochs KMeans initialization (100 Centers).

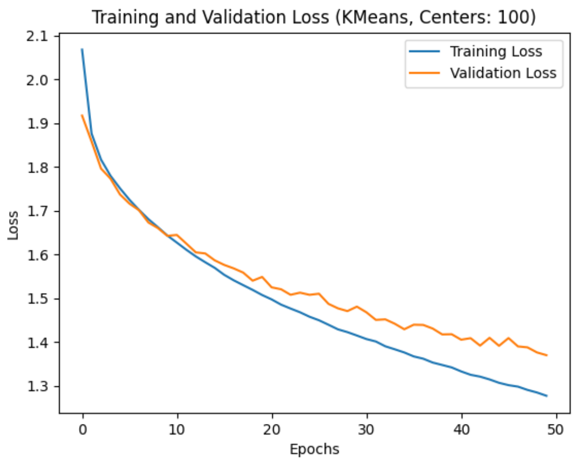
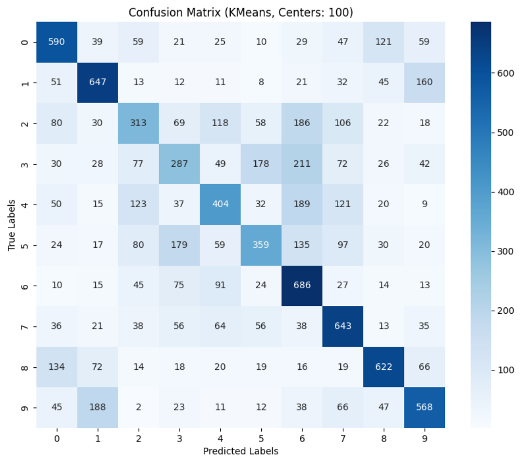
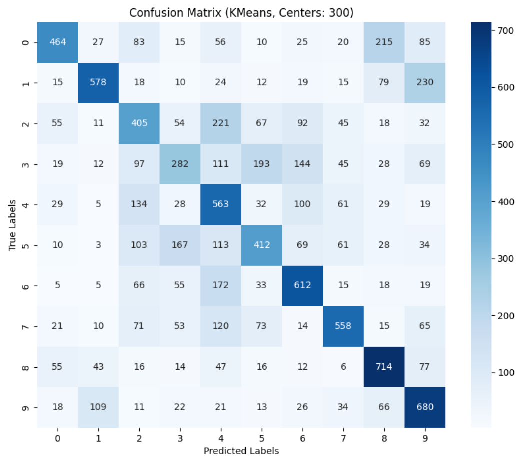


Fig. 3. Training and Validation loss throughout epochs KMeans initialization (100 Centers).



	0	1	2	3	4	5	6	7	8	9
0	590	39	59	21	25	10	29	47	121	59
1	51	647	13	12	11	8	21	32	45	160
2	80	30	313	69	118	58	186	106	22	18
3	30	28	77	287	49	178	211	72	26	42
4	50	15	123	37	404	32	189	121	20	9
5	24	17	80	179	59	359	135	97	30	20
6	10	15	45	75	91	24	686	27	14	13
7	36	21	38	56	64	56	38	643	13	35
8	134	72	14	18	20	19	16	19	622	66
9	45	188	2	23	11	12	38	66	47	568

Fig. 1. Confusion Matrix for KMeans initialization (100 Centers).



	0	1	2	3	4	5	6	7	8	9
0	464	27	83	15	56	10	25	20	215	85
1	15	578	18	10	24	12	19	15	79	230
2	55	11	405	54	221	67	92	45	18	32
3	19	12	97	282	111	193	144	45	28	69
4	29	5	134	28	563	32	100	61	29	19
5	10	3	103	167	113	412	69	61	28	34
6	5	5	66	55	172	33	612	15	18	19
7	21	10	71	53	120	73	14	558	15	65
8	55	43	16	14	47	16	12	6	714	77
9	18	109	11	22	21	13	26	34	66	680

Fig. 4. Confusion Matrix for KMeans initialization (300 Centers).

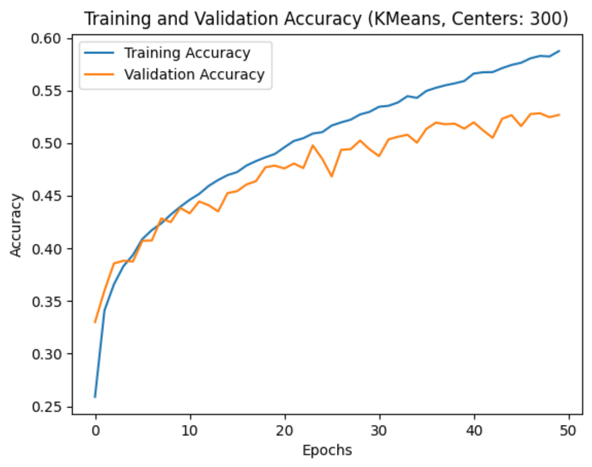


Fig. 5. Training and Validation accuracy throughout epochs KMeans initialization (300 Centers).

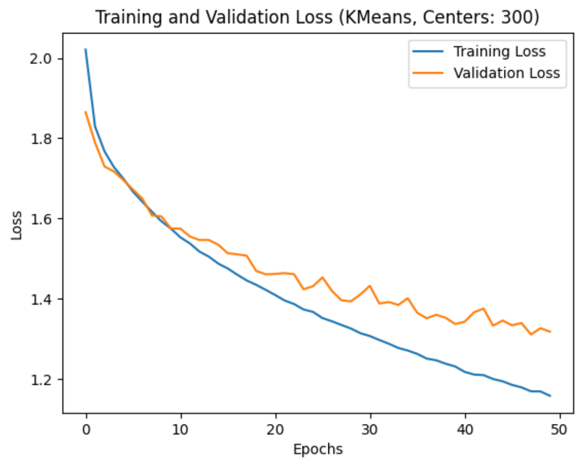


Fig. 6. Training and Validation loss throughout epochs KMeans initialization (300 Centers).

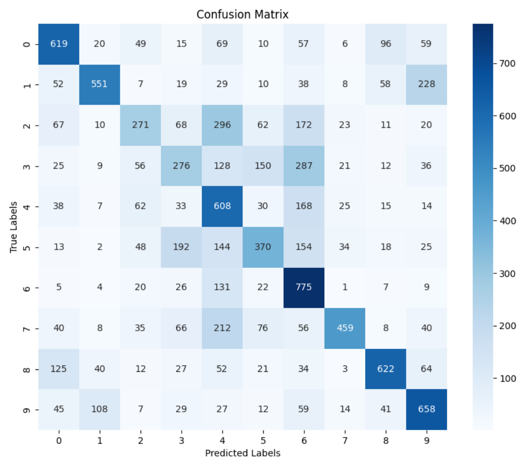


Fig. 7. Confusion Matrix for KMeans initialization (500 Centers).

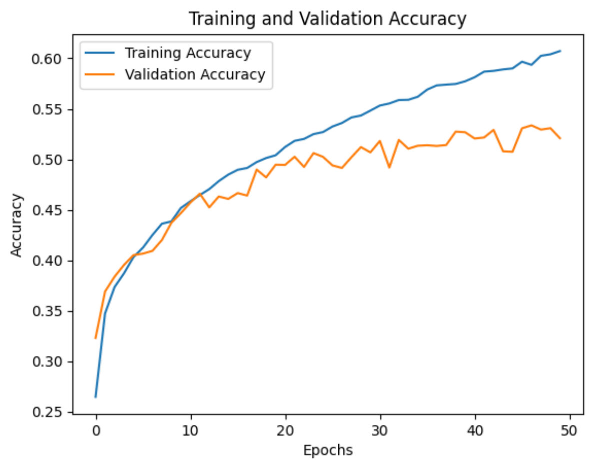


Fig. 8. Training and Validation accuracy throughout epochs KMeans initialization (500 Centers).

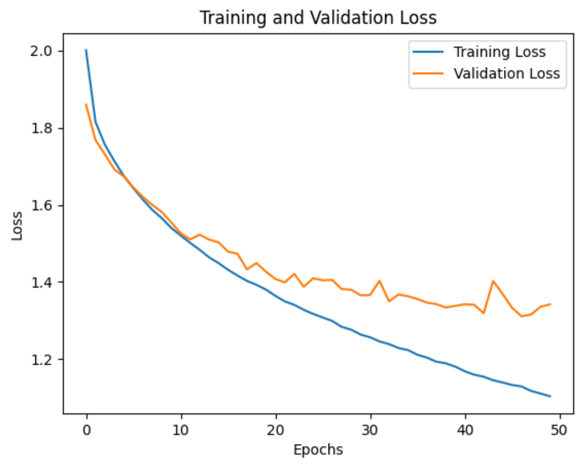


Fig. 9. Training and Validation loss throughout epochs KMeans initialization (500 Centers).

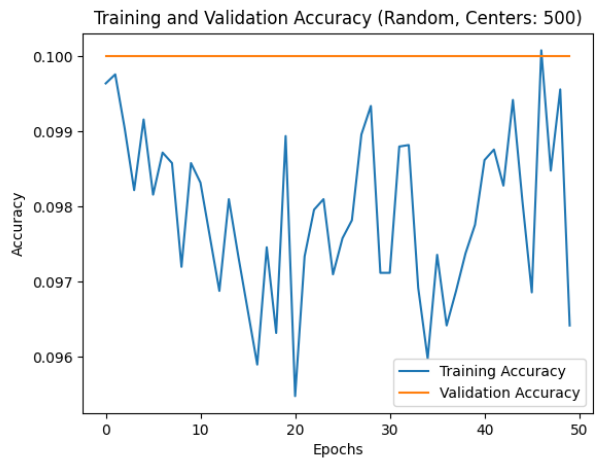


Fig. 10. Training and Validation accuracy throughout epochs RANDOM initialization (500 Centers).

The code corresponding to this initial implementation is in the `RBF_gradientdescent.py` file.

IV. Process for Comparative Experimentation with RBF Network and Other Classifiers

In this section, I detail the process followed to implement and evaluate the Radial Basis Function (RBF) network, k-Nearest Neighbors (k-NN), and Nearest Centroid classifiers as part of a comparative study on the CIFAR-10 dataset. The steps below describe how the tasks were carried out and the methodologies used.

A. Data Preparation and Dimensionality Reduction

- **Dataset:** The CIFAR-10 dataset was loaded using a custom function `load_cifar10_data`, and it consists of 60,000 images classified into 10 categories. The images were flattened to vectors for further processing.
- **Principal Component Analysis (PCA):** To reduce the computational burden and extract meaningful features, PCA was applied to reduce the dimensionality of the data while retaining 90% of the variance.
- The reduced-dimension data was then used as input for the RBF network, k-NN classifier, and Nearest Centroid classifier.

B. Implementation of the RBF Network

- **Network Structure:** The RBF network was implemented with a custom class, where the number of RBF centers (`num_centers`) and the output dimensions (`output_dim`) were defined as key hyperparameters.
- **Centers Initialization:** KMeans clustering was used to initialize the centers of the RBF units. The calculated cluster centroids represent the RBF centers, ensuring optimal placement.
- **Sigma Calculation:** The width parameter (`sigma`) of the RBF units was computed as the mean of the pairwise Euclidean distances between centers.
- **Interpolation Matrix:** A Gaussian RBF function was applied to calculate the interpolation matrix, which maps input data to the RBF space.
- **Weights Computation:** The network weights were computed using the inverse of the interpolation matrix and the one-hot encoded labels.
- **Training and Prediction:** Training involved computing weights, while prediction used the interpolation matrix to calculate output probabilities. Accuracy and confusion matrices were recorded for evaluation.

C. Implementation of k-NN and Nearest Centroid Classifiers

- **k-NN Classifier:** The k-NN classifier was implemented with different values of k (1, 3, and 5). The classifier was trained on the PCA-reduced data, and predictions were made on the test set. Metrics such as training time, testing time, accuracy, confusion matrices, and classification reports were recorded for each k.
- **Nearest Centroid Classifier:** The Nearest Centroid classifier was implemented to classify samples based on the closest class centroid in the feature space. Similar metrics as the k-NN classifier were computed for evaluation.

D. Results of Various Classifiers

This section provides a comparative analysis of the performance of different classifiers: the RBF Network, k-Nearest Neighbors (k-NN), and the Nearest Centroid Classifier. Key metrics such as training time, testing time, accuracy, confusion matrices, and classification reports are summarized below.

*1) **RBF Network Results**:*

- **Training Time:** 18.05 seconds
- **Testing Time:** 3.24 seconds
- **Accuracy:** 41.45%

Confusion Matrix:

$$\begin{bmatrix}
442 & 51 & 51 & 12 & 25 & 14 & 38 & 43 & 267 & 57 \\
29 & 497 & 16 & 26 & 15 & 16 & 60 & 46 & 108 & 187 \\
113 & 47 & 243 & 73 & 138 & 73 & 158 & 73 & 49 & 33 \\
59 & 75 & 59 & 227 & 55 & 181 & 148 & 83 & 38 & 75 \\
55 & 45 & 119 & 42 & 318 & 45 & 205 & 102 & 42 & 27 \\
31 & 51 & 86 & 150 & 61 & 327 & 120 & 99 & 43 & 32 \\
18 & 65 & 72 & 57 & 88 & 54 & 552 & 44 & 11 & 39 \\
41 & 57 & 37 & 50 & 118 & 57 & 77 & 419 & 40 & 104 \\
70 & 82 & 8 & 24 & 15 & 29 & 18 & 35 & 622 & 97 \\
40 & 193 & 10 & 17 & 10 & 17 & 50 & 53 & 112 & 498
\end{bmatrix}$$

Classification Report:

```
              precision    recall  f1-score   support
           0       0.49      0.44      0.47      1000
           1       0.43      0.50      0.46      1000
           2       0.35      0.24      0.29      1000
           3       0.33      0.23      0.27      1000
           4       0.38      0.32      0.35      1000
           5       0.40      0.33      0.36      1000
           6       0.39      0.55      0.46      1000
           7       0.42      0.42      0.42      1000
           8       0.47      0.62      0.53      1000
           9       0.43      0.50      0.46      1000

    accuracy                           0.41     10000
   macro avg       0.41      0.41      0.41     10000
weighted avg       0.41      0.41      0.41     10000
```

*2) **k-NN Classifier Results**:* The k-NN classifier was evaluated with three values of k: 1, 3, and 5. Below are the results:

k = 1:

- **Training Time:** 0.04 seconds
- **Testing Time:** 1.67 seconds
- **Accuracy:** 38.58%

k = 3:

- **Training Time:** 0.01 seconds
- **Testing Time:** 1.52 seconds
- **Accuracy:** 36.59%

k = 5:

- **Training Time:** 0.01 seconds
- **Testing Time:** 1.37 seconds
- **Accuracy:** 38.07%

*3) **Nearest Centroid Classifier Results**:*

- **Training Time:** 0.06 seconds
- **Testing Time:** 0.00 seconds
- **Accuracy:** 27.66%

Classification Report:

```
              precision    recall  f1-score   support
           0       0.27      0.54      0.36      1000
           1       0.28      0.19      0.22      1000
           2       0.28      0.11      0.16      1000
           3       0.26      0.06      0.09      1000
           4       0.27      0.12      0.16      1000
           5       0.27      0.29      0.28      1000
           6       0.22      0.54      0.31      1000
           7       0.26      0.16      0.20      1000
           8       0.42      0.37      0.39      1000
           9       0.33      0.41      0.36      1000

    accuracy                           0.28     10000
   macro avg       0.29      0.28      0.25     10000
weighted avg       0.29      0.28      0.25     10000
```

4) ***Key Observations***:

- The RBF Network outperformed other methods in terms of accuracy (41.45%) but required significantly more training time than k-NN and Nearest Centroid classifiers.
- Among the k-NN configurations, $k = 1$ provided the best accuracy (38.58%), but its results were closer to random guessing compared to the RBF Network.
- The Nearest Centroid classifier had the fastest training and testing times but achieved the lowest accuracy (27.66%).
- These results highlight the trade-off between computational efficiency and model accuracy.

E. ***Visualizations and Outputs***

Below are visualizations of the confusion matrices and metrics for all classifiers:

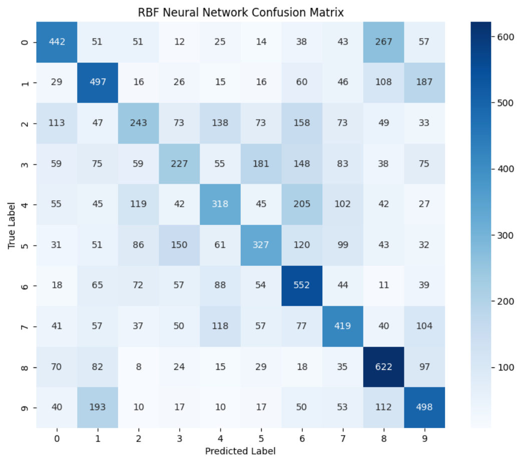


Fig. 11. Confusion Matrix for RBF Network.

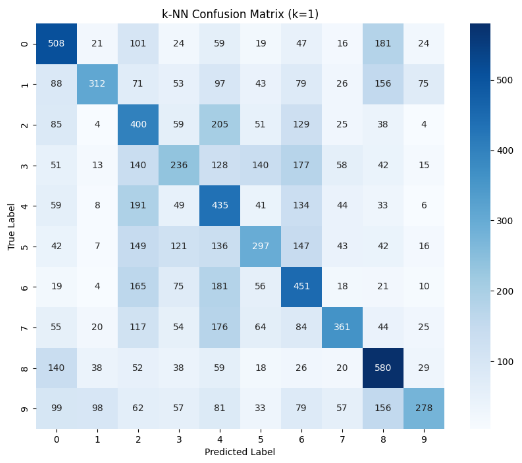


Fig. 12. Confusion Matrix for k-NN Classifier ($k = 1$).

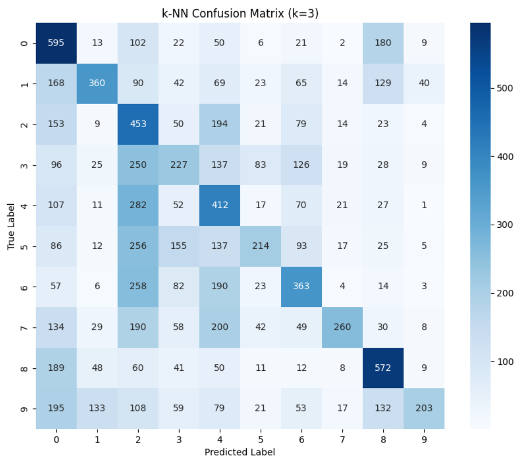


Fig. 13. Confusion Matrix for k-NN Classifier ($k = 3$).

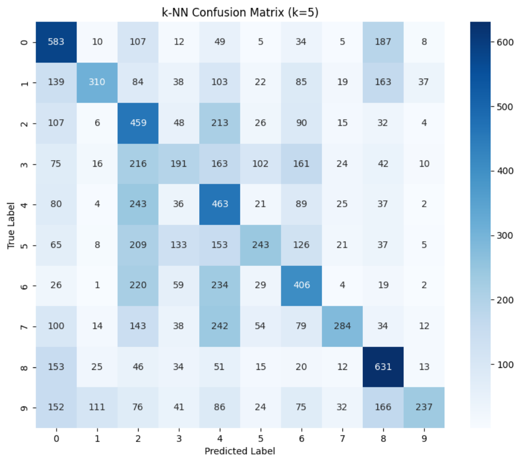


Fig. 14. Confusion Matrix for k-NN Classifier ($k = 5$).

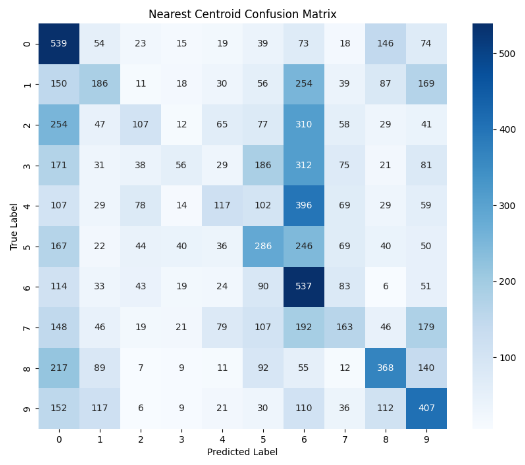


Fig. 15. Confusion Matrix for Nearest Centroid Classifier.

The code corresponding to this RBFN implementation is in the `rbf_all.py` file.

F. *An improved version of RBFN (experiment) ...*

In this experiment, an improved version of the Radial Basis Function Network (RBFN) was implemented to address the limitations of the previous model. Key enhancements include the incorporation of adaptive sigmas for the RBF kernels, regularization to stabilize weight calculations, and batch-wise kernel computations for memory efficiency. The CIFAR-10 dataset was used to evaluate the model's performance.

*1) **Model Improvements**:* The following improvements were made to the RBFN:

- **Adaptive Sigmas:** - The sigma values for each RBF center were calculated based on the average distance to its nearest neighbors. This allows each RBF unit to adjust its width dynamically, improving the model's ability to capture local features in the data.
- **Regularization:** - A small regularization term was added to the weights calculation to prevent overfitting and numerical instability during matrix inversion.
- **Batch-wise Kernel Computation:** - To handle the high-dimensional CIFAR-10 data efficiently, the kernel matrix was computed in batches, reducing memory consumption and ensuring scalability.

*2) **Results**:* The improved RBFN was evaluated on the CIFAR-10 dataset, with the following results:

- **Accuracy:** 47.19%
- **Confusion Matrix:**

$$\begin{bmatrix}
528 & 47 & 54 & 14 & 23 & 7 & 34 & 44 & 189 & 60 \\
27 & 606 & 12 & 29 & 10 & 22 & 27 & 35 & 78 & 154 \\
92 & 50 & 302 & 79 & 128 & 58 & 150 & 78 & 35 & 28 \\
45 & 46 & 69 & 288 & 64 & 154 & 150 & 73 & 41 & 70 \\
56 & 37 & 142 & 43 & 361 & 36 & 167 & 102 & 36 & 20 \\
24 & 42 & 74 & 181 & 48 & 344 & 110 & 103 & 37 & 37 \\
9 & 33 & 63 & 59 & 101 & 41 & 609 & 39 & 22 & 24 \\
34 & 50 & 32 & 58 & 85 & 58 & 50 & 506 & 31 & 96 \\
80 & 85 & 14 & 29 & 14 & 19 & 14 & 22 & 646 & 77 \\
40 & 205 & 10 & 24 & 8 & 20 & 37 & 38 & 89 & 529
\end{bmatrix}$$

Classification Report:

```
              precision    recall  f1-score   support
           0       0.56      0.53      0.55      1000
           1       0.50      0.61      0.55      1000
           2       0.39      0.30      0.34      1000
           3       0.36      0.29      0.32      1000
           4       0.43      0.36      0.39      1000
           5       0.45      0.34      0.39      1000
           6       0.45      0.61      0.52      1000
           7       0.49      0.51      0.50      1000
           8       0.54      0.65      0.59      1000
           9       0.48      0.53      0.51      1000
    accuracy                           0.47     10000
   macro avg       0.47      0.47      0.46     10000
weighted avg       0.47      0.47      0.46     10000
```

*3) **Key Observations**:*

- The improved RBFN achieved an accuracy of 47.19%, which is higher than the traditional RBFN (41.45%). This demonstrates the effectiveness of adaptive sigmas and regularization.
- The confusion matrix shows better class separation compared to the previous implementation, though certain classes (e.g., 3 and 5) still exhibit significant confusion.
- The classification report highlights that precision and recall values are fairly balanced across classes, with higher scores for simpler classes like 0 and 8.
- The improved RBFN remains computationally efficient due to the use of batch-wise kernel computation.

*4) **Visualizations**:* The following visualizations illustrate the performance of the improved RBFN:

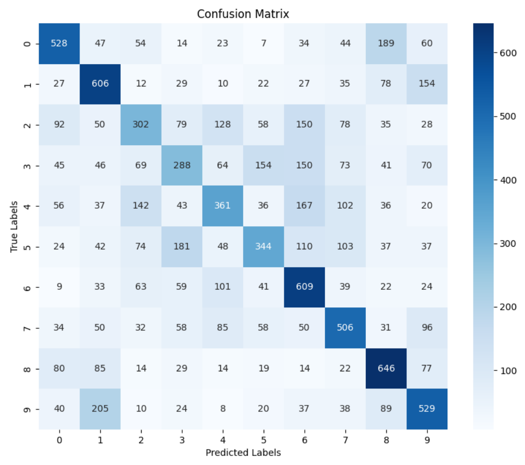


Fig. 16. Confusion Matrix for the Improved RBFN.

*5) **Summary**:* The improved RBFN shows significant gains over the basic RBFN by taking advantage of adaptive sigmas, regularization, and batch-wise computations. These additions resulted in better class separability, improved accuracy, and efficient training.

The code corresponding to this improved implementation is in the `RBF_improved.py` file.

G. *Summary of Observations*

- The RBF network performed well but required significant computational resources for training compared to the other classifiers.

- The k-NN classifier achieved the highest accuracy, but its testing time increased with the dataset size.
- The Nearest Centroid classifier was the fastest and simplest to implement but lagged slightly in accuracy compared to the k-NN classifier.
- The choice of classifier depends on the trade-off between accuracy and computational efficiency for the given application.
- **It is very very important** to note from this experiment that the trainable RBF Neural Network (RBFNN) implemented in the previous sector performed better than the Radial Basis Function Network (RBFN) (even the attempt for an improved version) where the weights were computed using the inverse of the interpolation matrix rather than with backpropagation and gradient-based optimization and sigmas are not trainable. This result highlights the advantage of allowing trainable centers and weights, which enable the model to better adapt to the dataset through iterative optimization. In contrast, the fixed-weight RBFN's reliance on analytical solutions, while computationally simpler, limits its capacity to learn complex patterns. Now I should also add here that the RBFNN I implemented in the previous sector outperforms the k-NN and Nearest Centroid Classifier as well, not only the RBF Network I showcased here.

V. Initial Autoencoder (MNIST)

This section explains how an autoencoder was used to reconstruct images from the MNIST dataset. An autoencoder is a model that compresses data into a smaller size and then learns to recreate the original data from this compressed version.

A. *Model Design*

The autoencoder has two main parts:

- **Encoder:**
  - Shrinks the input image into a smaller, compressed form (latent space).
  - Uses two layers of filters (convolutions) and pooling to extract features.
  - Compresses the features into a 32-dimensional vector.
- **Decoder:**
  - Recreates the original image from the 32-dimensional vector.
  - Uses layers that expand the size of the image step by step.

B. *How the Model was Trained*

- **Data:** - The MNIST dataset contains 60,000 gray images of numbers (size 28 × 28).
- **Preprocessing:** - The image pixel values were scaled to be between 0 and 1.
- **Training:**
  - The model learned to match its outputs to the original images.
  - It was trained for 15 rounds (epochs) with a batch size of 128 images at a time.
  - A part of the data (20%) was used for validation.

C. *Results*

The autoencoder was able to recreate images from the MNIST dataset with high accuracy:

- **Reconstruction Accuracy (1 - MSE):** 99.25%
- **Training and Validation Loss:** - Figure 17 shows the error steadily reducing as the model improved.
- **Reconstructed Images:** - Figure 18 compares the original and recreated images. The recreated images look almost the same as the originals.

D. *Visualizations and Plots*

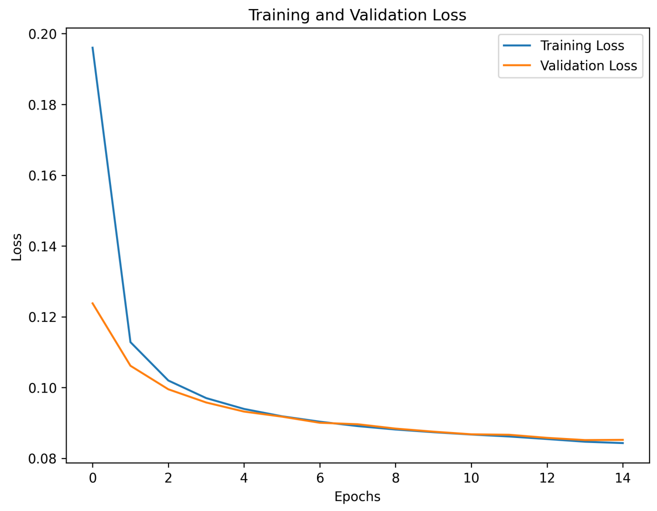


Fig. 17. Training and Validation Loss for the Autoencoder on MNIST.

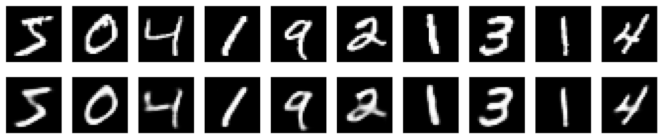


Fig. 18. Comparison of Original and Reconstructed Images. Top row: Original images. Bottom row: Reconstructed images. (MNIST)

E. *What We Learned*

- The model did a great job at recreating images, with a very high reconstruction accuracy (99.25%).
- The recreated images are very similar to the original ones, as seen in Figure 18.
- The model's error reduced steadily during training, as shown in Figure 17.
- This is a good first step in testing autoencoders. The simple MNIST data made it easier for the model to perform well.

F. *Summary*

This experiment showed that an autoencoder can effectively compress and recreate MNIST images. The model's high accuracy and quality outputs make it a strong starting point for more advanced experiments, such as using harder datasets or adding new features like removing noise.

The code corresponding to this initial autoencoder for MNIST dataset implementation is in the `autoencoder_mnist.py` file.

VI. Autoencoder for CIFAR-10 (TensorFlow Implementation)

This section describes the use of an autoencoder to reconstruct images from the CIFAR-10 dataset. The autoencoder was built and trained using TensorFlow and evaluated for its ability to recreate the original images.

A. *Model Design*

The autoencoder has two main parts:

- **Encoder:**
  - Compresses the input image into a smaller, simpler representation using two convolutional layers.
  - Each layer reduces the size of the image while learning important features.
- **Decoder:**
  - Recreates the original image from the compressed representation.
  - Uses two transposed convolutional layers to restore the image size and details.

B. *Training Details*

- **Data:** - CIFAR-10 dataset with 50,000 images was used for training.
- **Preprocessing:** - Images were normalized so their pixel values ranged between 0 and 1.
- **Training Configuration:**
  - Batch size: 64 images
  - Number of epochs: 10
  - Loss function: Mean squared error
  - Optimizer: Adam with a learning rate of 0.001
- **Output:** - You can find my results in the the file: `autoencoder_results_tf_cifar.txt`.

C. *Results*

The autoencoder's performance was evaluated by comparing the reconstructed images to the originals. The following results were obtained:

- **Training Time:** 95.14 seconds
- **Final Training Loss:** 0.0013
- **Reconstruction Accuracy (1 - MSE):** 0.9988

D. *Visualizations*

The comparison between the original and reconstructed images is shown in Figure 19. The top row shows the original images, while the bottom row shows their reconstructions.

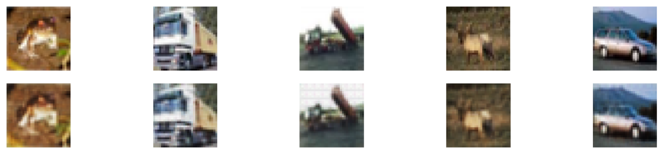


Fig. 19. Comparison of Original and Reconstructed Images. Top row: Original images. Bottom row: Reconstructed images.

E. *What We Learned*

- The autoencoder performed exceptionally well, achieving a reconstruction accuracy of 99.88%.
- The training process was efficient, taking only 95.14 seconds to complete 10 epochs, with a very low final loss of 0.0013.
- The reconstructed images are nearly identical to the original ones, as seen in Figure 19.
- This experiment demonstrates that a simple autoencoder architecture can handle the colorful and complex images in CIFAR-10 effectively.

F. *Summary*

The TensorFlow autoencoder successfully compressed and recreated CIFAR-10 images with extremely high accuracy. The results show that the model is highly efficient and a solid foundation for further experiments or extensions.

The code corresponding to this autoencoder for the Cifar10 dataset implementation is in the `autoencoder_cifar10.py` file.

VII. References Section

References

[1] https://www.ibm.com/topics/knn
[2] https://en.wikipedia.org/wiki/Radial_basis_function_kernel
[3] https://www.geeksforgeeks.org/auto-encoders/
[4] https://keras.io/api/datasets/mnist/
[5] https://www.cs.toronto.edu/ kriz/cifar.html
[6] https://en.wikipedia.org/wiki/Radial_basis_function_network