ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
& ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΑΡΙΣΤΟΤΕΛΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΟΝΙΚΗΣ

RBF Neural Network

Panagiotis Karvounaris
AEM 10193

January 10, 2024

A report presented for the course of Neural Networks
Aristotle University of Thessaloniki Faculty of Electrical and Computer Engineering

Contents

1 Introduction

The goal of this project is to use a RBF Neural Network to classify correctly correctly the pictures of CIFAR-10 dataset. The dataset is consisted of pictures 32x32 pixels and coloured. We will use different techniques to train our network, different number of RBF layer neurons and 3 different types of networks at the end. We will see clustering strategies and we will change the parameters of the RBF's neurons to find the best fit.

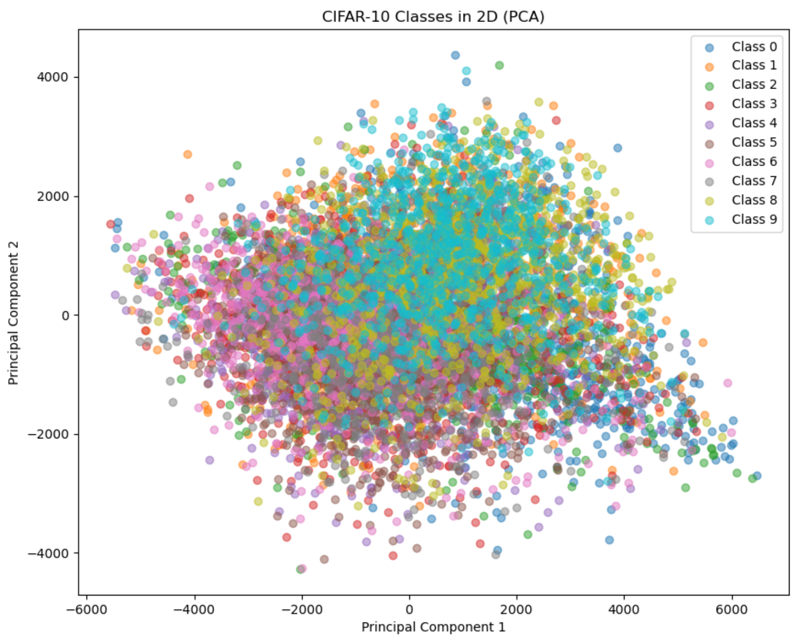


Figure 1: Dataset CIFAR-10

2 Code Description

In order to test multiple architectures and a variety of parameters, we used COTS algorithms from sklearn library and pytorch. The code analysis is explained below.

2.1 Load and Preprocess the CIFAR-10 dataset

We use the 'tensorflow.keras.datasets' library to load the CIFAR-10 dataset. First of all, we load the data for training and testing, then we reshape the image data and scales the pixel values dividing by 255. We reshape the original image data from a 4D array to a 2D array. It uses information about the number of samples, image height, image width, and the number of channels. So, each image has a total of 32x32x3 = 3072 features. Furthermore, we shuffle the data in order to have better results in the training for 5000 and 10000 samples.

```
import tensorflow as tf
from sklearn.utils import shuffle
from keras.utils import to_categorical

# Load and preprocess training and test data
# for better and more efficient usage
(x_train, y_train), (x_test, y_test) =
                tf.keras.datasets.cifar10.load_data()
x_train = x_train.reshape((-1, 3072)) / 255.0
x_test = x_test.reshape((-1, 3072)) / 255.0
y_train = y_train.ravel()
y_test = y_test.ravel()
x_train, y_train = shuffle(x_train, y_train,
                            random_state=0)

# One-hot encode the labels
y_train_encoded = to_categorical(y_train, 10)
y_test_encoded = to_categorical(y_test, 10)
```

2.2 PCA

For most of the training instances we use Primary Component Analysis (PCA), in order to speed up the training process. PCA is a linear dimensionality reduction technique and we use to to reduce the every image dimension from 3072 to about 100 dimensions, while keeping at least 90 percent of the initial information of the image. The speed of the training is improving, thus we can use more training samples, so we end up training a bigger potion of the dataset at the same time, that increases the performance of the network.

```
from sklearn.decomposition import PCA
```

```
pca = PCA(0.9).fit(x_train)
pca_train_data = pca.transform(x_train)
pca_test_data = pca.transform(x_test)
```

2.3 Save Data to File

In order to keep track of the results and use them afterwards to reach conclusions, we save the results of the the training in a .txt file. We could use grid search in order to obtain the best combination of parameters, but we chose to run all the models and find the best combination manually. We also print the results in terminal for progress notification reasons.

```
def print_and_log(text, log_file):
    print(text)
    log_file.write(text + '\n')

with open("data_log.txt", "w") as log_file:
    print_and_log('data-to-be-saved', log_file)
```

2.4 Clustering

Clustering is a methodology where we try to find the center of classes in a dataset. We have a multi-class classification with ten class, which means that it would be good for as to use 10 centers. But that is not the case, let's see why. We have to work with a highly non linear dataset, so if we use the K-means clustering algorithm straight forward we will take 10 centers that does not actually give us enough information to continue. That is why we separate the 10 classes from the whole dataset (5000 samples its class) and we use K-means clustering to every one of them separately, so now we have more meaningful centers to initialize our RBF neurons, because we know that every center is for just one class and it is not mixed. This will improve the result of our network.

```
from sklearn.cluster import KMeans
# Separate the classes from the PCA-transformed
# training dataset
def separate_classes(pca_train_data, y_train,
                                        num_classes=10):
    class_subsets = [pca_train_data[y_train == i]
                          for i in range(num_classes)]
    return class_subsets

# Apply kmeans to everyone of the 10 classes
# of the PCA-transformed dataset
def apply_kmeans_to_classes(class_subsets,
                                n_clusters, max_iters=300):
    kmeans_results = []
    for class_data in class_subsets:
        kmeans = KMeans(n_clusters=n_clusters,
        max_iter=max_iters, random_state=0,
                                        n_init='auto')
        kmeans.fit(class_data)
        kmeans_results.append((kmeans.cluster_centers_,
                                             kmeans.labels_))
    return kmeans_results

#Implement kmean clustering
class_subsets = separate_classes(pca_train_data, y_train)

# Extracting and combining the centroids from
# kmeans_results
all_rbf_centers = np.vstack([centers for centers,
                             _ in kmeans_results])

# Apply k-means clustering with current n_clusters
kmeans_results = apply_kmeans_to_classes(class_subsets,
                                  n_clusters=n_clusters)
```

2.5 RBF Layer

We use the output of PCA as input for the RBF layer. We know that the output of the PCA has 99 dimensions, so the RBF layer will have more neurons than 99. That is generally a good practise to go on a higher dimension that the initial, so the problem would be easier to solve.

```
import numpy as np
from scipy.spatial.distance import cdist

# Implement gaussin rbf function for both
# gaussian and quadratic function
def rbf_function(x_train, class_center, parameter,
                                  type_of_rbf):
    if type_of_rbf == 'gaussian':
        return np.exp(-cdist(x_train, class_center,
                        'sqeuclidean') / (parameter**2))
    elif type_of_rbf == 'multiquadratic':
        return (np.sqrt(cdist(x_train, class_center,
                        'sqeuclidean') + parameter**2))

# Applying the Gaussian RBF function
rbf_transformed_data_train = rbf_function(
                    pca_train_data, all_rbf_centers,
                    parameter, type_of_rbf=type_of_rbf)
```

```
rbf_transformed_data_test = rbf_function(pca_test_data,
                            all_rbf_centers, parameter,
                            type_of_rbf=type_of_rbf)
```

2.6 Logistic Regression Output Layer

We use logistic regression layer from sklearn library as an output layer of our RBF neural network. Logistic regression is a type of statistical model used for binary classification, but the sklearn has a great implementation of logistic regression for multi-class classification, so we used this version.

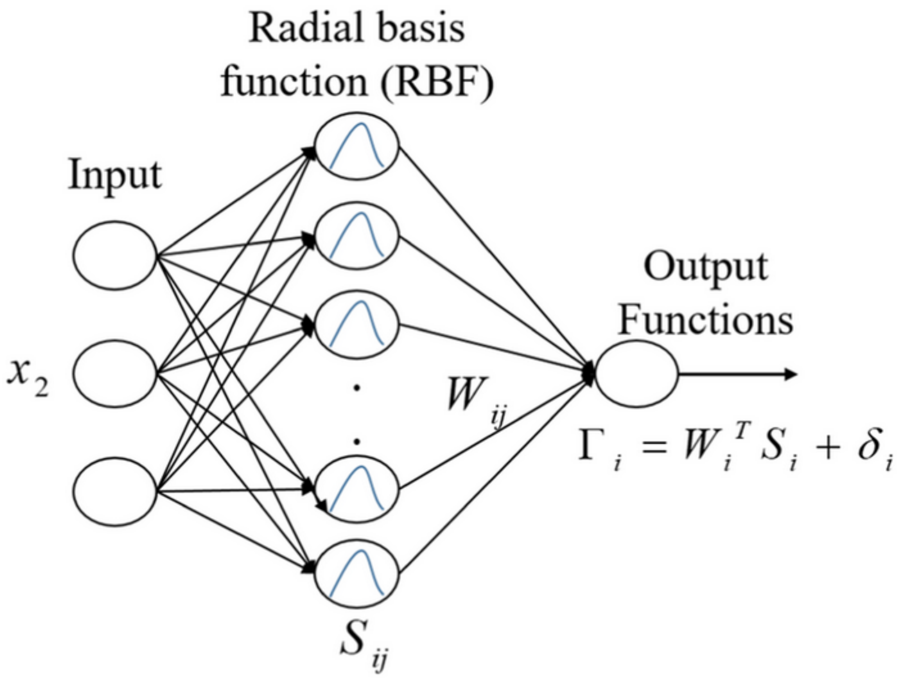
```
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler

# Train logistic regression model with increased max_iter
logistic_regression = LogisticRegression(
multi_class='multinomial', solver='lbfgs',
                                max_iter=20000)
logistic_regression.fit(
        rbf_transformed_data_train_scaled, y_train)

# Calculate and print accuracies
train_accuracy = logistic_regression.score(
            rbf_transformed_data_train_scaled, y_train)
test_accuracy = logistic_regression.score(
            rbf_transformed_data_test_scaled, y_test)
```

2.7 MLP Output Layer

Instead of using logistic regression, we use a multi layer perceptron wit different types of architecture from the number of hidden layers to the number of neuron per layer. We used the MLP provided by torch library. Due to this decision we have to transform numpy arrays to pytorch tensors to make the MLP effective.

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader

# Define the MLP model
class MLP(nn.Module):
    def __init__(self, input_size, hidden_size1,
                         hidden_size2, num_classes):
        super(MLP, self).__init__()
        self.linear = nn.Sequential(
        nn.Linear(input_size, hidden_size1),
        nn.BatchNorm1d(hidden_size1),
        nn.ReLU(),
        nn.Linear(hidden_size1, hidden_size2),
        nn.BatchNorm1d(hidden_size2),
        nn.ReLU(),
        nn.Linear(hidden_size2, num_classes),
        )
        # self.linear = nn.Sequential(
        # nn.Linear(input_size, hidden_size1),
        # nn.BatchNorm1d(hidden_size1),
        # nn.ReLU(),
        # nn.Linear(hidden_size1, num_classes),
        # )

    def forward(self, x):
        return self.linear(x)

# Convert numpy arrays to PyTorch tensors
def numpy_to_tensor(data):
    return torch.tensor(data, dtype=torch.float32)

# Convert data to PyTorch tensors
X_train_tensor = numpy_to_tensor(
        rbf_transformed_data_train_scaled)
Y_train_tensor = numpy_to_tensor(y_train_encoded)
X_test_tensor = numpy_to_tensor(
        rbf_transformed_data_test_scaled)
Y_test_tensor = numpy_to_tensor(y_test_encoded)

# Create DataLoader for batch processing
train_data = TensorDataset(X_train_tensor,
                           Y_train_tensor)
test_data = TensorDataset(X_test_tensor,
                           Y_test_tensor)
train_loader = DataLoader(dataset=train_data,
                batch_size=batch_size, shuffle=True)
test_loader = DataLoader(dataset=test_data,
                batch_size=batch_size, shuffle=False)

# Model, loss, and optimizer
model = MLP(X_train_tensor.shape[1], hidden_size1,
                         hidden_size2, num_classes)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Initialize lists to store metrics for plotting
```

```
epoch_numbers = []
test_accuracies = []
training_losses = []

# Training and evaluation loop
for epoch in range(epochs):
    # Training
    model.train()
    train_loss = 0.0
    train_correct = 0
    train_total = 0
    for i, (inputs, labels) in enumerate(train_loader):
        # Forward pass
        outputs = model(inputs)
        loss = criterion(outputs,
                         torch.max(labels, 1)[1])
        train_loss += loss.item() * inputs.size(0)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        _, predicted = torch.max(outputs.data, 1)
        train_total += labels.size(0)
        train_correct += (predicted ==
                    torch.max(labels, 1)[1]).sum().item()

    train_loss /= train_total
    train_accuracy = 100 * train_correct / train_total

    # Evaluation
    model.eval()
    test_correct = 0
    test_total = 0
    with torch.no_grad():
        for inputs, labels in test_loader:
            outputs = model(inputs)
            predicted = torch.max(outputs.data, 1)[1]
            test_total += labels.size(0)
            test_correct += (predicted ==
                    torch.max(labels, 1)[1]).sum().item()

    test_accuracy = 100 * test_correct / test_total

    # Store metrics
    epoch_numbers.append(epoch + 1)
    test_accuracies.append(test_accuracy)
    training_losses.append(train_loss)

    # Log training and test results
    print_and_log(f'Number of clusters per class:
            {n_clusters}, Parameter: {parameter},
            Type of RBF: {type_of_rbf}\n'
            f'Epoch [{epoch+1}/{epochs}], '
            f'Training Loss: {train_loss:.4f}, '
            f'Training Accuracy: {train_accuracy:.2f}%, '
            f'Test Accuracy: {test_accuracy:.2f}%',
                                        log_file)
```



Radial basis function (RBF)
Input
x_2
Output Functions
W_{ij}
$\Gamma_i = W_i^T S_i + \delta_i$
S_{ij}

Figure 2: RBF Diagram

3 Results

We will see and compare the results of different architectures.

3.1 MLP with one Hidden Layer

Let's start our tests using MLP neural network with one hidden layer (128 neurons) and one output layer, 10 neurons, as many as the number of classes of our dataset. We train the model for 100 epochs and using batch size 100. These values was tested amongst others and seemed to have the best results. Below we can see test accuracy plots and loss plots. We present the best combination of parameters that gave us the best results for each one of our clustering strategies. Number of clusters per class 50, 100 and 150.

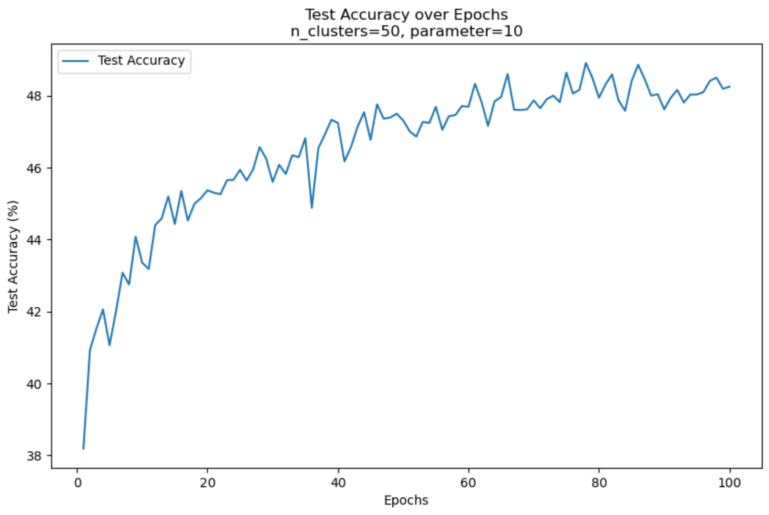


Figure 3: Number of Clusters per Class 50, RBF's Parameter 10

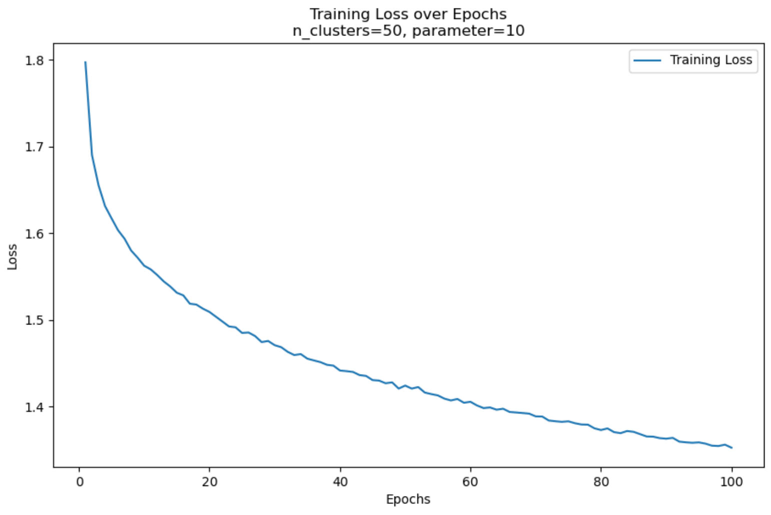


Figure 4: Number of Clusters per Class 50, RBF's Parameter 10

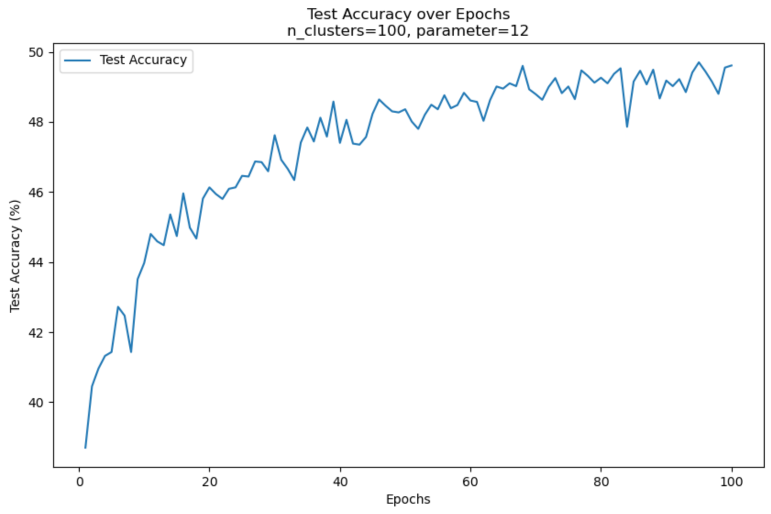


Figure 5: Number of Clusters per Class 100, RBF's Parameter 12

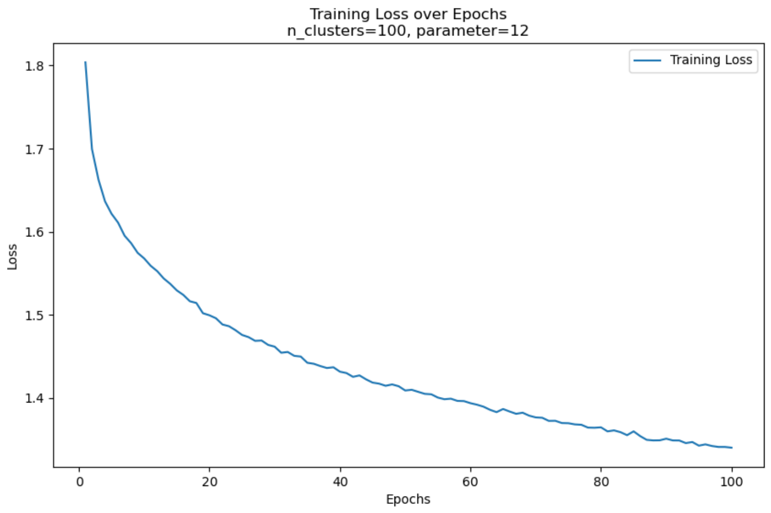


Figure 6: Number of Clusters per Class 100, RBF's Parameter 12

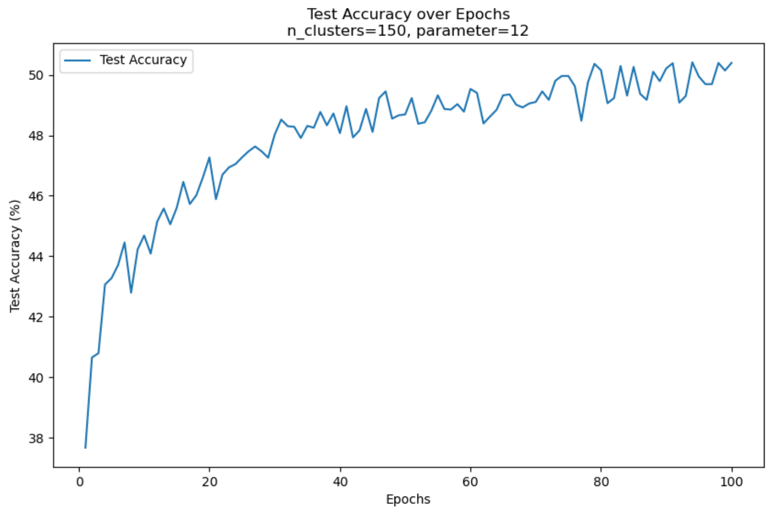


Figure 7: Number of Clusters per Class 150, RBF's Parameter 12

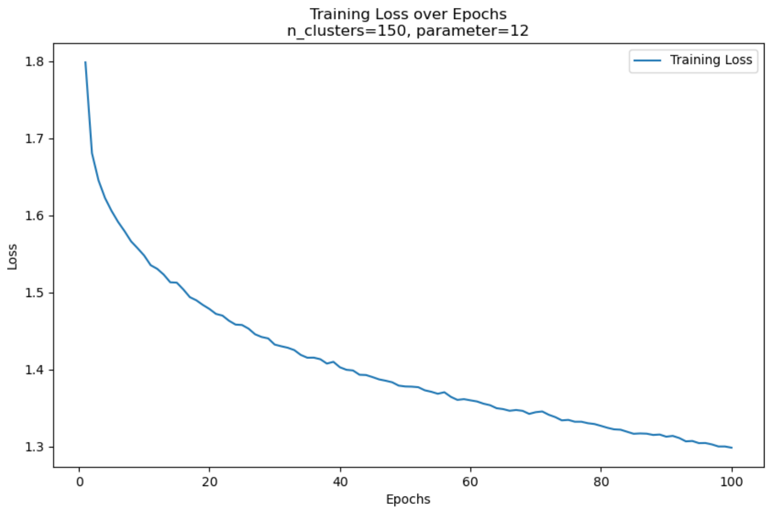


Figure 8: Number of Clusters per Class 150, RBF's Parameter 12

3.2 MLP with two Hidden Layer

Let's continue using MLP neural network with two hidden layer (256 neurons and 64 neurons) and one output layer, 10 neurons, as many as the number of classes of our dataset. We train the model for 100 epochs and using batch size 100. These values was tested amongst others and seemed to have the best results. Below we can see test accuracy plots and loss plots. We present the best combination of parameters that gave us the best results for each one of our clustering strategies. Number of clusters per class 50, 100 and 150.

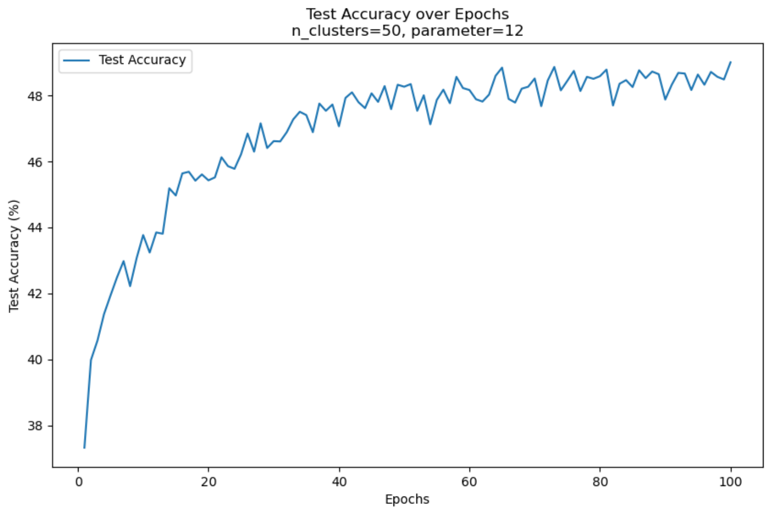


Figure 9: Number of Clusters per Class 50, RBF's Parameter 12

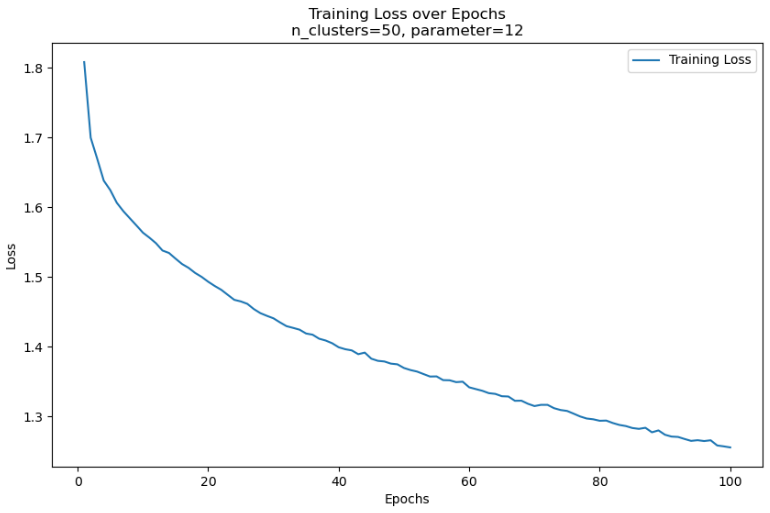


Figure 10: Number of Clusters per Class 50, RBF's Parameter 12

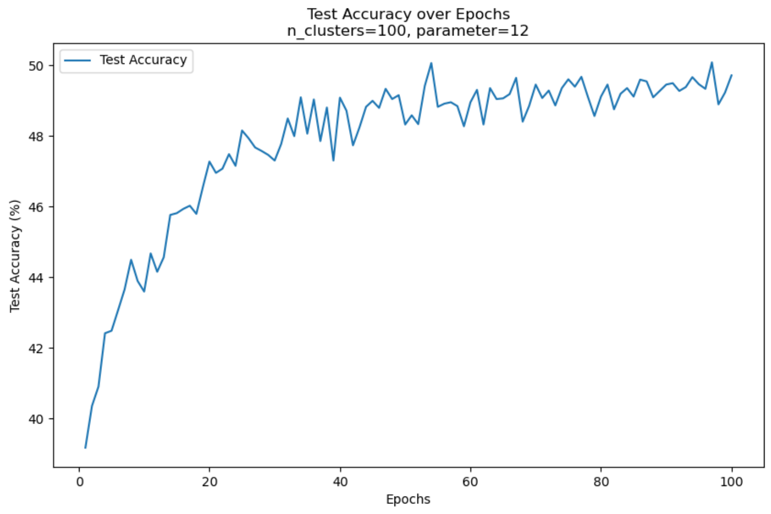


Figure 11: Number of Clusters per Class 100, RBF's Parameter 12

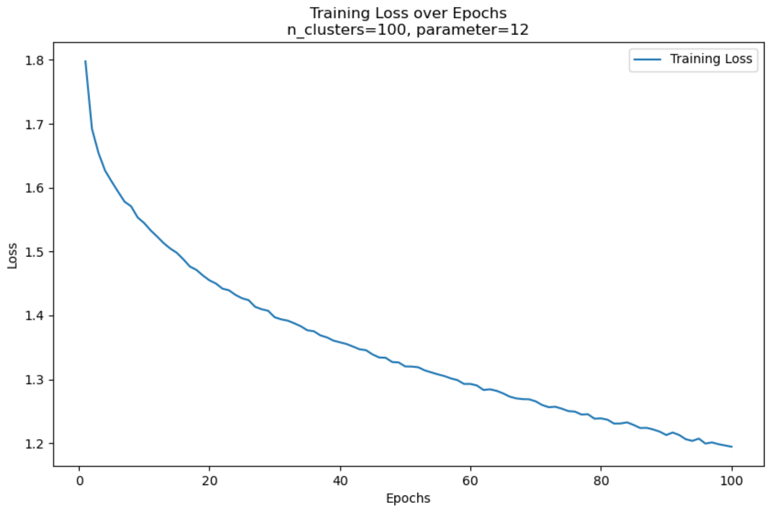


Figure 12: Number of Clusters per Class 100, RBF's Parameter 12

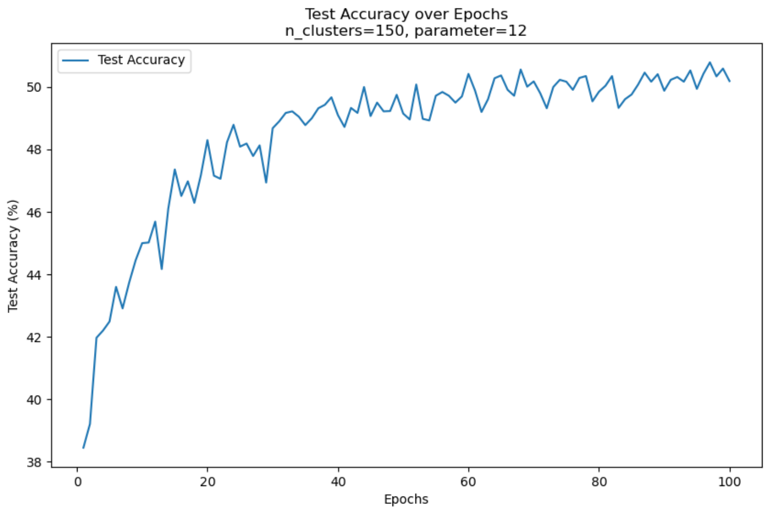


Figure 13: Number of Clusters per Class 150, RBF's Parameter 12

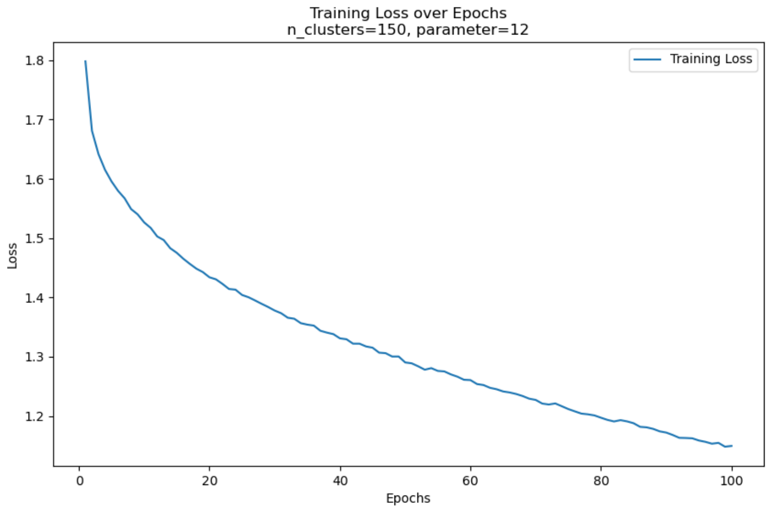


Figure 14: Number of Clusters per Class 150, RBF's Parameter 12

3.3 Conclusion of MLP

- Both architectures top the test accuracy to 0.5
- Not a meaningful advantage gained by adding one more hidden layer.
- The higher value you use for number clusters per initial class the better results you get, but there is an upper limit, depending from the number of training samples and the value of the RBF's parameter.
- The change of dimensions from the RBF layer seems to have positive impact to RBF's training.

Here we have to mention that there are probably better architectures of MLP that would perform better, for examples adding more hidden layers, change the number of neurons to each layer etc.

3.4 Logistic Regression

We continue by changing the philosophy of MLP to a more simple one layer output structure. We are going to use a one layer - 10 neuron output structure with logistic regression for multi-class classification. We will train our model for both gaussian RBF and multiquadratic RBF, for a variety of number of clusters per class and RBF's parameters. Below, we can see the best results.

We run both gaussian and multiquadratic for

```
n_clusters_options = [5, 10, 20, 30, 40, 50]
parameter_options = [0.01, 0.1, 0.5, 1, 2, 3]
```

and we get here
Number of clusters: 50, Parameter: 1, Type of RBF: multiquadratic
Time collapsed: 99.54 seconds
Training Accuracy: 0.4899, Test Accuracy: 0.4791
Number of clusters: 50, Parameter: 0.5, Type of RBF: multiquadratic
Time collapsed: 102.99 seconds
Training Accuracy: 0.4902, Test Accuracy: 0.4785
Number of clusters: 50, Parameter: 2, Type of RBF: multiquadratic
Time collapsed: 108.49 seconds
Training Accuracy: 0.4893, Test Accuracy: 0.4785

We now run only the gaussian version because the multiquadratic version needed a long time to train due to the creation of many centers for the RBF layer and even though it performs better with low values for RBF's parameter, that changes drastically when the value of the parameter goes up.

We run gaussion only for

```
n_clusters_options = [100, 150, 200, 300]
parameter_options = [1, 2, 3, 5, 8, 10, 12, 15, 20]
```

and we get
Number of clusters: 300, Parameter: 10, Type of RBF: gaussian
Time collapsed: 1034.44 seconds
Training Accuracy: 0.6474, Test Accuracy: 0.5316
Number of clusters: 300, Parameter: 12, Type of RBF: gaussian
Time collapsed: 909.34 seconds
Training Accuracy: 0.6171, Test Accuracy: 0.5334

3.5 Conclusion of Logistic Regression

- Performs pretty good for a one layer structure and gives the best results.
- Much quicker training process than a MLP.
- Works fine for gaussian and multiquadratic RBFs.

4 Machine Learning Classifiers

We will compare our results with two famous machine learning classifiers, k-nn and centroid.

4.1 k-Nearest Neighbours Classifier

The results for the k-Nearest Neighbours Classifier (k=3), are:

```
k−Nearest Neighbours Classifier Results:
Accuracy: 0.3303
Recall: 0.3303
F1 Score: 0.3191924379899631
              precision    recall  f1−score   support

           0       0.32      0.57      0.41      1000
           1       0.58      0.24      0.34      1000
           2       0.20      0.45      0.28      1000
           3       0.26      0.23      0.24      1000
           4       0.25      0.44      0.32      1000
           5       0.43      0.21      0.28      1000
           6       0.36      0.23      0.28      1000
           7       0.73      0.20      0.31      1000
           8       0.44      0.61      0.51      1000
           9       0.73      0.12      0.21      1000

    accuracy                           0.33     10000
   macro avg       0.43      0.33      0.32     10000
weighted avg       0.43      0.33      0.32     10000
```

4.2 Nearest Centroid Classifier

The results for the Nearest Centroid Classifier, are:

```
Nearest Centroid Classifier Results:
Accuracy: 0.2774
Recall: 0.2774
F1 Score: 0.25408598962127454
              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       0.27      0.06      0.09      1000
           4       0.28      0.12      0.17      1000
           5       0.27      0.29      0.28      1000
           6       0.22      0.54      0.31      1000
           7       0.27      0.17      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
```

5 Conclusion

For the above results we can conclude that:

- PCA is really helpful for the quality of the results and the speed of training process.
- The more number of clusters we have, the better results we have.
- Gaussian RBF performs better at high values for the RBF's parameter and multiquadratic performs better at low values for the RBF's parameter. This is expected considering their formulas.
- A single layer using Logistic Regression performs better than the above specific MLP architectures that has been used.
- Two hidden layers MLP work the same as one hidden layer MLP.
- Both linear regression and MLP tactics in combination with RBF layer, k-mean clustering and PCA, out perform the machine learning algorithms k-Nearest Neighbours (k=3) and Nearest Centroid. Although, we have to find the right number of clusters per class, RBF's parameter and the type of RBF function we will use, in order for the RBF-nn to work perfectly and give as good results. Of course, we have to admit that the machine learning algorithms run much faster and do not need as much computational power.

In the zip file of this project you can see the log files with more data and plots about the combination of parameters and their results, using different architectures.