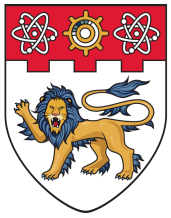

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

NANYANG TECHNOLOGICAL UNIVERSITY

EE7207 Neural Networks & Deep Learning

Assignment 1 Report

MSc in Electrical and Electronic Engineering
Yan Ziming
G2507084J

February 10, 2026

Contents

1 Introduction

The report introduces the process of classification with RBF neural network. The RBF contains three layers: input layer, hidden layer, and output layer.

The key obstacle during the development is the selection of parameters, like Gaussian width (σ) and hidden layer neuron number. I used different methods to deal with these difficulties, such as SOM, Kmeans, and nonlinear optimization apart from basic process.

Briefly, I divided my design into two parts: (1) basic method to develop basic RBF (2) improve parameters with backpropogation. The final result reflect positive improvement.

2 RBF Neural Network Frame

2.1 Input Layer

The input layer of RBF is different from normal neural network, it just transmit the training data vector $\mathbf{x} = [x_1, x_2, ..., x_n]^T$ to the hidden layer. In this stage, no mathematical transformations or learning processes occur.

2.1.1 Data Analysis

The datasets are provided as static files, namely *data_train.mat*, *data_test.mat* and *label_train.mat*; therefore, manual partitioning of training and testing sets is not required.

Data Information:

Table 1: Sample Sets

Parameter	*Training Data*	*Testing Data*
Number of Samples (n)	301	50
Number of Features (m)	33	33
Categories	2	2
Missing Value	0	0
Outlier Ratio (IQR)	49.83%	54.00%

Table 2: Label Set

Parameter	*Training Label*
Number of Samples (n)	301
Categories	{-1, 1}
Distribution	[106, 195]
Missing Value	0

I tried to seek for outliers with IQR metric and considered the sample which has outlier feature element as outlier sample to compute ratio. According to the data analysis, the outlier of sample set too large to be normal. Therefore, I plot boxplot to visualize the real distributions of training data in each feature.

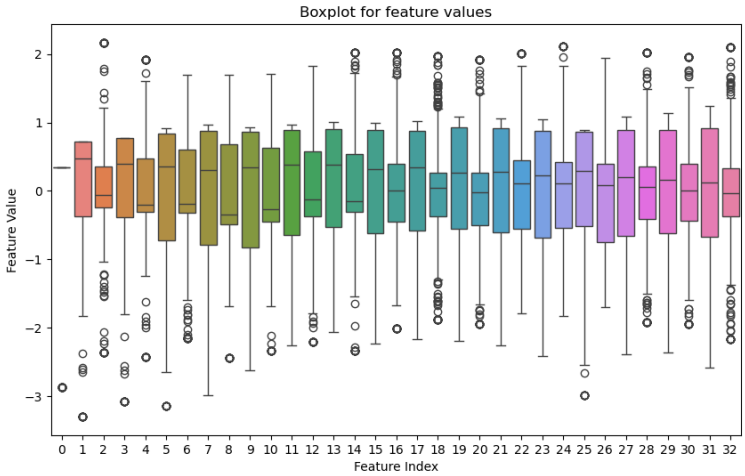

Figure 1: Boxplot of Data_train

According to figure, the main outliers come from several features. The distribution of data, like mean value and variation standardized and the range of value have normalized as well. There is no need to normalizztion and standarlization.

Generally, the high dimension leads to extreme complexity in feature space. It is the **Curse of Dimensionality**. Since the input layer need to transmit sample to hidden layer, the input layer should have 33 neurons.

2.2 Hidden Layer

The hidden layer is the computational core of the RBF network, responsible for mapping the input data into a new feature space based on the neurons in hidden layer. The RBF hidden layer utilizes radial basis functions to capture local responses. The output of the j-th hidden neuron, $\phi_j(\mathbf{x})$ is determined by the distance between the input vector $\mathbf{x}$ and a predefined center $\mathbf{c}_j$:

$$\phi_j(\mathbf{x}) = \exp\left(-\frac{|\mathbf{x}-\mathbf{c}_j|^2}{2\sigma_j^2}\right) \tag{1}$$

where $|\cdot|$ denotes the Euclidean norm, and σ_j represents the Gaussian width (spread) of the j-th neuron. This local mapping ensures that only inputs near the center $\mathbf{c}_j$ produce a significant activation.

2.2.1 Centers Selection

The selection of centers $\mathbf{c}_j$ is the most critical step in designing the hidden layer, as they serve as the "prototypes" that define the localized receptive fields within the input space. Given the 33-dimensional complexity, a robust and systematic selection strategy is employed.

In this implementation, the K-means clustering algorithm is utilized as the primary method to partition the feature space. To determine the optimal number of hidden neurons K, the Elbow Method is performed by plotting the Within-Cluster Sum of Squares (WCSS) against a range of cluster counts. The "elbow" point of the curve is identified as the optimal balance between model complexity and error reduction, providing a quantitative basis for the hidden layer's scale.

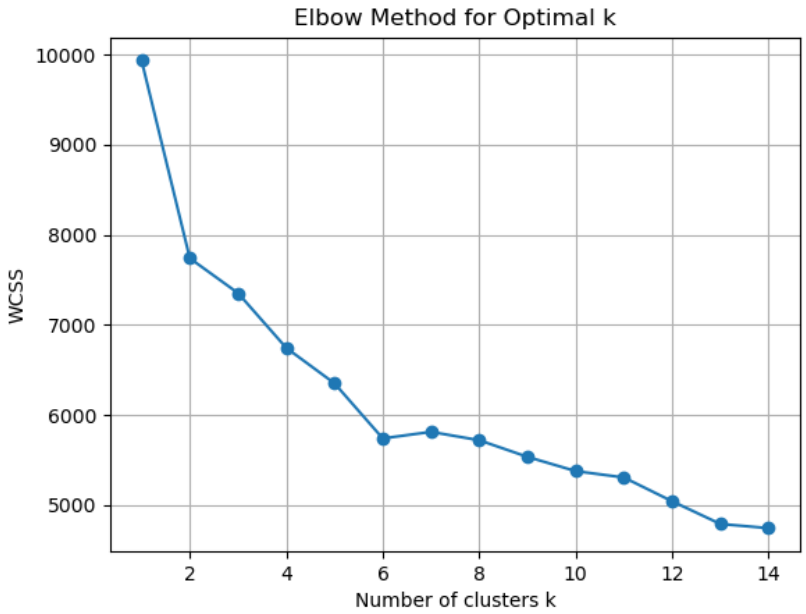

Figure 2: Cluster Number & WCSS

Since the elbow occours at $k = 6$ in the figure, I decide to use 6 hidden neuron to develop hidden layer.

To further validate the reliability of these centers in the presence of high-dimensional noise, a Self-Organizing Map (SOM) is used for auxiliary verification. While K-means efficiently minimizes local variance, the SOM provides a topological mapping that ensures the centers are not merely biased by dense clusters but also adequately cover t he "outlier-heavy" boundaries identified in our data analysis. This dual-verification approach ensures that the prototype centers $\mathbf{c}_j$ maintain a representative coverage of the manifold.

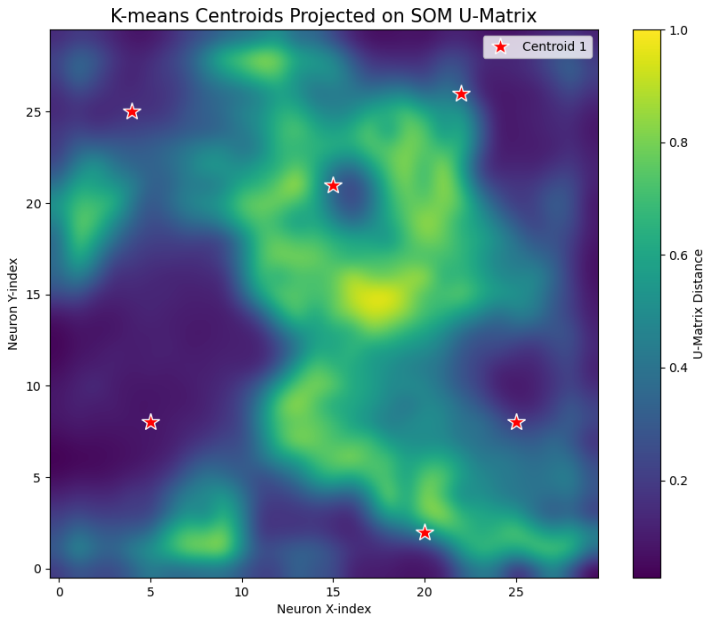

Figure 3: Centroids on U_matrix

The U_matrix map reflected the 30×30 neurons' distances, the color darker, the distances of neuron to neighbors closer, meaning more possible to belongs to same cluster. Since the centroids computed by Kmeans clustering fall into the blue "valley", the selection of 6 neurons in hidden layer roughly feasible.

2.2.2 Gaussian Width

The Gaussian width σ determines the receptive field of each hidden neuron. Setting an appropriate σ is a balancing act between sensitivity and generalization:

- **Addressing Dimensionality:** In a 33-dimensional space, the Euclidean distance between points tends to be large and sparse. If σ is too small, the receptive fields will not overlap, causing the network to lose its interpolation capability (the "dead neuron" problem).
- **Robustness to Outliers:** Conversely, an excessively large σ would over-smooth the decision boundary, potentially blurring the distinction between the categories, especially given the imbalanced distribution ([106, 195]).

A common heuristic is used to set a global width:

$$\sigma = \frac{d_{max}}{\sqrt{2K}} \tag{2}$$

where $d_{max}$is the maximum distance between chosen centers and K is the number of hidden units. This ensures that the hidden layer provides a continuous and smooth mapping across the entire input manifold.

In this experiment, the σ I computed is 2.4604.

2.3 Output Layer

The output layer serves as the final stage of the RBF network, responsible for integrating the local features processed by the hidden layer into a coherent global decision. Structurally, it is a single-layer feed-forward network with a linear activation function.

2.3.1 Structural Composition

For this binary classification task, the output layer consists of a single neuron that performs a weighted summation. The inputs to this layer are the K activation values $[\phi_1(x), \phi_2(x), \ldots, \phi_K(x)]^T$ generated by the hidden neurons. The connection between the j-th hidden neuron and the output neuron is defined by a scalar weight w_j.

The total input to the output neuron, denoted as z, is calculated as:

$$z = \sum_{j=1}^{K} w_j \phi_j(\mathbf{x}) + b \tag{3}$$

$$\mathbf{z} = \Phi_{bias} W \tag{4}$$

In order to get appropriate prediction, the error should be minimized. With the Ordinary Least Square method, the weight matrix W could be found. Subsequently, the label $\mathbf{z}$ could be predicted as well.

2.3.2 Linear Summation and Decision Rule

According to label data, the data belongs to two categories: $y \in \{-1, 1\}$. Since the weight matrix W and hidden layer output Φ are in float, the labels predicted will not follow these binary classes. Therefore, a signum function (or a threshold) should be applied to the raw result:

$$y = \text{sgn}(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{if } z < 0 \end{cases} \tag{5}$$

This simple decision rule converts the continuous output of the network into the discrete labels. The output $\mathbf{z} \in R^n$, n is the number of training sample, which means each input only output one number to represent their belongings. Therefore, the neuron number in output later should be 1.

2.4 Nonlinear Optimization

While the integration of K-means, SOM, and the heuristic formula $\sigma = \frac{d_{max}}{\sqrt{2K}}$ provides a statistically sound initialization for the hidden layer, these traditional methods often struggle with the high dimension data. In a 33-dimensional space, the predefined

centers and the static Gaussian width may not capture the complex, non-linear decision boundaries with sufficient precision, potentially leading to sub-optimal mapping.

To address this, I leverage Non-linear Optimization via the PyTorch framework. Instead of treating the hidden layer parameters as fixed constants, I redefine the centers $\mathbf{c}_j$, widths σ and weight matrix W as trainable parameters. By implementing the Gradient Descent-based Backpropagation (BP) algorithm, the network can iteratively refine these coefficients to minimize the global loss function. This transition from heuristic initialization to gradient-based fine-tuning allows the RBF neurons to adaptively "shift" and "resize" their receptive fields, significantly improving the model's accuracy and robustness against the high outlier ratio observed in our data.

3 Results Analysis

3.1 Baseline

The baseline reflects the result before the nonlinear optimization.

3.1.1 Prediction Result

Table 3: Prediction Class Label

Sample Index	*Class Label*
0-9	-1 1 -1 -1 1 1 -1 1 -1 -1
10-19	-1 1 -1 -1 -1 -1 -1 1 -1 -1
20-29	1 1 1 -1 -1 1 -1 1 -1 1
30-39	-1 1 -1 1 1 1 1 1 1 1
40-49	1 -1 1 1 -1 1 1 1 1 1

3.1.2 Neuron Coverage

One important method to check whether the neuron and σ selected are feasible is the coverage of neurons. Since the dimension of data is too high to plot, I decrease the dimension with PCA and plot the corresponding neuron coverage.

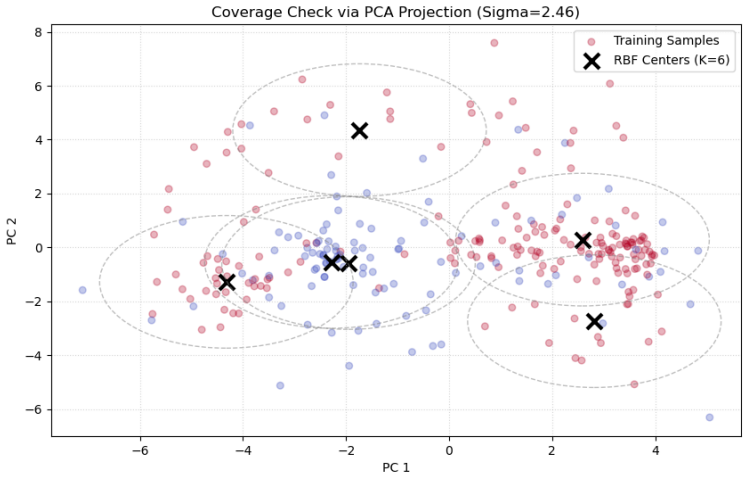

Figure 4: Coverage of Centroids

3.2 After Backpropogation

3.2.1 Prediction Result

Table 4: Prediction with Optimization

Sample Index	*Class Label*
0-9	-1 1 1 -1 1 1 -1 1 1 -1
10-19	-1 1 -1 -1 -1 -1 -1 1 1 -1
20-29	1 1 1 -1 -1 1 -1 1 -1 1
30-39	-1 1 -1 1 1 1 1 1 1 1
40-49	1 1 1 1 1 1 1 1 1 1

3.2.2 Coverage & Loss Curve

For the backpropogation, I use *torch.nn* and set learning rate = 0.001, epoch = 2000. The following figures shows the training loss and neuron coverage after optimization. From loss curve, the BP does reduces the loss function with epoch increasing.

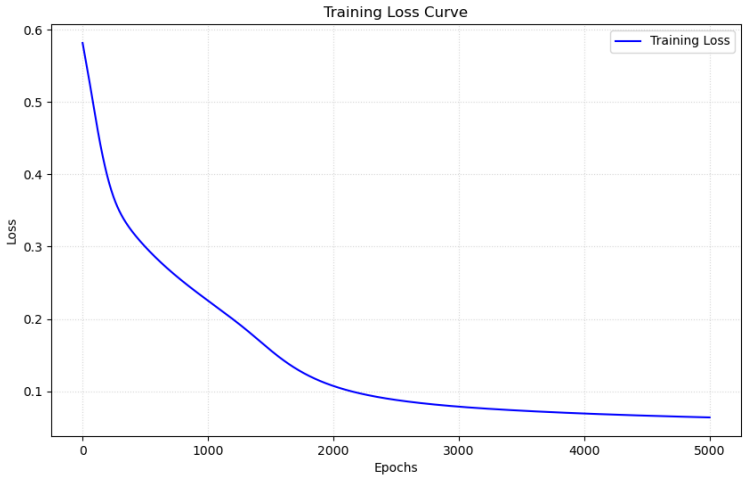

Figure 5: Training Loss

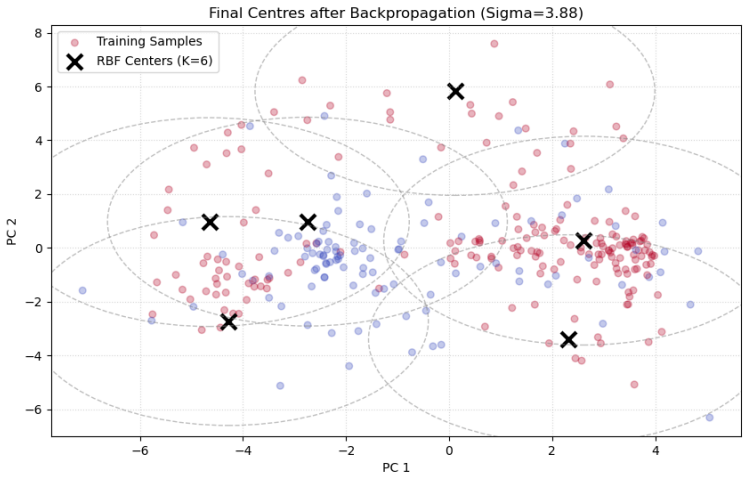

Figure 6: Center Coverage After Optimization

3.3 Comparition

To evaluate the RBF_nn roughly and statistically, I predict the training label to get metrics, comparing with given label. The results of baseline and optimization are shown in table.

Table 5: Metrics of Model

Metric	*Base*	*Optimized*
Accuracy	0.920	0.983
Precision	0.907	0.985
Recall	0.927	0.979
F1 Score	0.915	0.982
Poor coverage number	36/301	0/301

By comparing the metrics and coverage visualizations, it is evident that non-linear optimization significantly enhances the model's fit to the training data and refines the spatial distribution of hidden neurons. However, in the absence of ground-truth testing labels, it remains inconclusive whether this performance gain represents a genuine improvement in predictive power or an artifact of over-fitting. The results suggest that while the network has successfully minimized the empirical risk on the training manifold, its generalization capability requires further cross-validation or regularization audits.

4 Appendix

4.1 Core Implementation

4.1.1 K-means Algorithm

```python
## Neuron centres selection
## Kmeans method
import numpy as np

class ncs_Kmeans:
    def __init__(self, X, n_clusters, max_iters=300, tol=1e-4):
        self.X = X
        self.n_clusters = n_clusters
        self.max_iters = max_iters
        self.tol = tol
        self.n_samples, self.n_features = X.shape

    def fit(self):
        np.random.seed(42)
        indices = np.random.choice(self.n_samples, self.n_clusters, replace=False)
        self.centres = self.X[indices].copy()

        for _ in range(self.max_iters):
            distances = np.linalg.norm(self.X[:, np.newaxis] - self.centres, axis=2)
            clusters = np.argmin(distances, axis=1)

            new_centres = np.zeros_like(self.centres)
            for i in range(self.n_clusters):
                points = self.X[clusters == i]
                if len(points) > 0:
                    new_centres[i] = points.mean(axis=0)
                else:
                    new_centres[i] = self.X[np.random.choice(self.n_samples)]

            center_shift = np.linalg.norm(self.centres - new_centres)
            if center_shift < self.tol:
                break

            self.centres = new_centres

        self.clusters = clusters
        return self.centres, self.clusters
```

4.1.2 RBF Neuron Network Configuation

```python
class RBFNN:
    def __init__(self, X, centres, sigma):
        self.X=X
        self.centres=centres
        self.sigma=sigma
        self.n_centres=centres.shape[0]

    def gaussian_rbf(self,x,centre):
        return np.exp(-np.linalg.norm(x-centre)**2/(2*self.sigma**2))

    def hidden_layer_activation(self):
        n_samples=self.X.shape[0]
        H=np.zeros((n_samples, self.n_centres))
        for i in range(n_samples):
            for j in range(self.n_centres):
                H[i,j]=self.gaussian_rbf(self.X[i], self.centres[j])
        return H

    def train(self,y):
        H=self.hidden_layer_activation()
        H_bias=np.hstack((np.ones((H.shape[0],1)), H))
        self.weights=np.linalg.pinv(H_bias).dot(y)
        self.W=self.weights

    def predict(self, X_test):
        n_samples=X_test.shape[0]
        H_test=np.zeros((n_samples, self.n_centres))
        for i in range(n_samples):
            for j in range(self.n_centres):
                H_test[i,j]=self.gaussian_rbf(X_test[i], self.centres[j])
        H_test_bias=np.hstack((np.ones((H_test.shape[0],1)), H_test))
        y_temp=H_test_bias.dot(self.weights)
        y_pred=np.where(y_temp>=0,1,-1)
        self.H=H_test_bias
        return y_pred

    ##check coverage and sigma
    def check(self):
        return self.W, self.H
```

4.1.3 SOM for U-matrix Map

```
kmeans = ncs_Kmeans(X_scaled,
n_clusters=6)
centroids, _=kmeans.fit()
# Train
som_size = 30
som = MiniSom(som_size, som_size,
X_scaled.shape[1], sigma=2.0,
learning_rate=0.5, random_seed=42)
som.pca_weights_init(X_scaled)
som.train_batch(X_scaled, 50000)
u_matrix = som.distance_map().T
```

4.1.4 Backpropagation Initialization & Forward Training

```python
#use backpropogation to adjust sigma and weights
import torch
import torch.nn as nn

class RBFNet(nn.Module):
    def __init__(self, centers, sigma):
        super(RBFNet, self).__init__()
        self.centers = nn.Parameter(torch.tensor(centers, dtype=torch.float32))
        self.sigma = nn.Parameter(torch.tensor(sigma, dtype=torch.float32))
        self.weights = nn.Linear(centers.shape[0], 1)

    def forward(self, x):
        dists = torch.cdist(x, self.centers)
        out = torch.exp(-dists**2 /  2 * self.sigma**2))
        return self.weights(out)   (

model = RBFNet(centres, sigma)
criterion = nn.HuberLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)


X_tensor = torch.tensor(X, dtype=torch.float32)
y_tensor = torch.tensor(y, dtype=torch.float32).reshape(-1, 1)

X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
```

The full code for this assignment are stored in github repository. URL: `https://github.com/yzmyyds/NTU_EE7207`

4.2 Dependency List

- Python: 3.11.13
- PyTorch: 2.10.0
- Scikit-learn: 1.7.2
- Matplotlib: 3.10.6
- SciPy: 1.16.1
- NumPy: 2.2.6