EE5907 Pattern Recognition
Continuous Assessment - 1
AY-2024/25

**Datta Souvik**
Student ID: A0295746L
Email ID - e1349884@u.nus.edu
National University of Singapore

# Classification with MLP updated by Backprop from Scratch and RBF Networks with Random Selected Centre Neurons.

## INDEX

# ABSTRACT

This assignment report focuses on two main parts of implementing machine learning models for classification tasks: a **Multi-Layer Perceptron (MLP)** and a **Radial Basis Function (RBF) Network**.

## Part 1: Classification with MLP Updated by Backpropagation

- The MLP was implemented from scratch using Python. It consists of a 2-layer structure with ReLU as the activation function for the hidden layer and Sigmoid for the output layer.
- Initially, the MLP achieved a classification accuracy of 63% on binary class data, generated by a provided script.
- The weights were updated using the **Backpropagation algorithm** with gradient descent, **improving the model's accuracy to 84%.**
- The assignment details the steps of the forward pass, weight initialization, and backpropagation. Visualizations, such as decision boundaries, are included to show the effect of weight updates and improved decision-making by the MLP.

## Part 2: Classification with RBF Network

- An RBF network was implemented with Gaussian radial basis functions for hidden neurons, starting with three RBF centres.
- The classification accuracy of the RBF network with 3 centres was 71%, and after increasing the number of centres to 6, **accuracy improved to 80%**.
- The report discusses the role of the RBF matrix and the impact of increasing the number of neurons on classification performance. Comparisons are made between the performance of the RBF networks with different numbers of centres and the MLP trained with backpropagation.

We conclude that while both MLP and RBF networks are effective for classification, backpropagation allows the MLP to adjust weights more effectively, while the RBF network benefits from increasing hidden neurons to improve accuracy.

## PART 1 - Classification with Multi-Layer Perceptron (MLP) updated by Backpropagation following Gradient Descent

### *A) Scatter Plot of the Generated Data*

For this assignment, Python programming language was used for generating the binary class data. The data generation was handled by the provided script, `"Generate_data.py"`, which outputs the data as `.npy` and `.mat` files and creates a scatter plot illustrating the generated data points.

Below is the scatter plot of the binary class data generated using the `Generate_data.py` script.

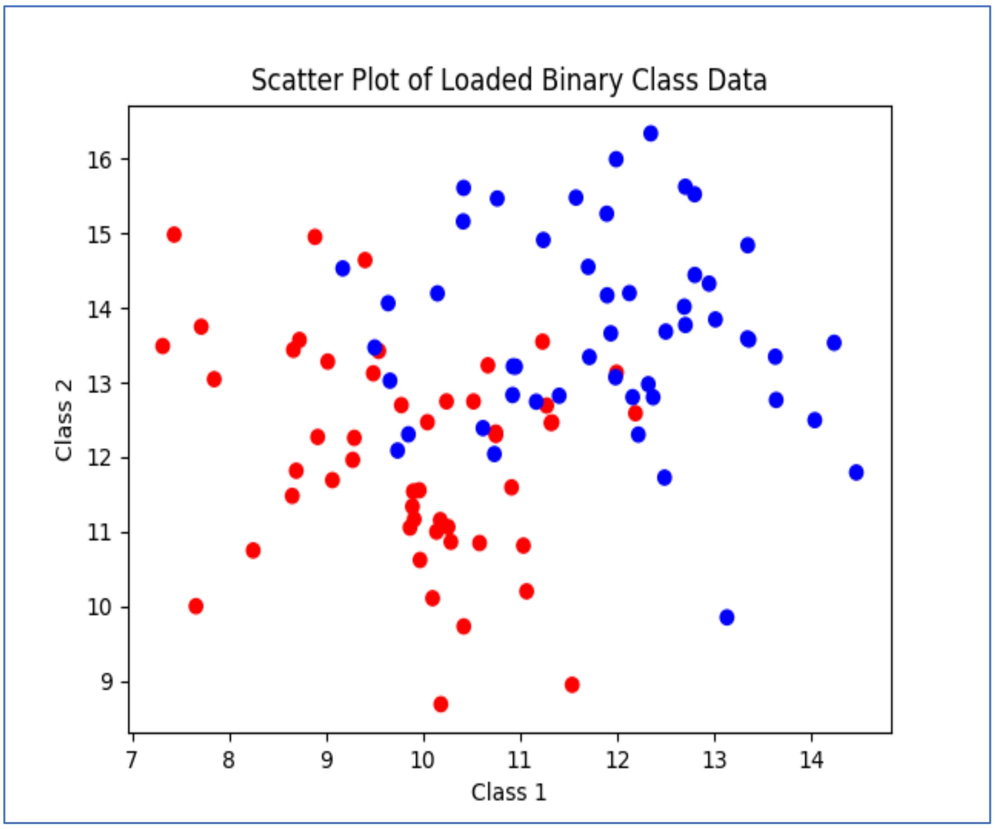


Figure 1. Scatter Plot of the Binary Class Dataset

In this plot, the data points are clearly distinguished into two classes: **Class 1 and Class 2**. The ground-truth output for Class 1 is labelled as 0, while the ground-truth output for Class 2 is labelled as 1.

`For reference, all code related to this assignment is available in the following GitHub repository: https://github.com/souvik0306/EE5907-Pattern_Recognition.`

## *B) Implementation of a Multi-Layer Perceptron (MLP) for Classification.*

The implementation of the 2-layer MLP with one hidden layer is designed to perform binary classification on the generated data. The structure includes 2 input neurons, 3 neurons in the hidden layer, and 1 output neuron. The activation functions used in this MLP are ReLU for the hidden layer and Sigmoid for the output layer. The initial accuracy of the MLP on the given dataset is **approximately 63%.**

### *1. Data Loading and Visualization*

The data is loaded from `.npy` files, assumed to be `class1.npy` and `class2.npy`, which contain the points for class 1 and class 2, respectively.

```python
import numpy as np
import matplotlib.pyplot as plt

# Load the data from npy files
X1 = np.load('class1.npy')
X2 = np.load('class2.npy')

# Labels for the classes
c1 = ['red'] * len(X1)  # Class 1
c2 = ['blue'] * len(X2) # Class 2

# Combine data
X = np.concatenate((X1, X2))
color = np.concatenate((c1, c2))

# Ground-truth labels: 0 for Class 1, 1 for Class 2
T = np.array([0] * len(X1) + [1] * len(X2))

# Plot the data
plt.scatter(X[:, 0], X[:, 1], marker='o', c=color)
plt.xlabel('Class 1')
plt.ylabel('Class 2')
plt.title('Scatter Plot of Loaded Binary Class Data')
plt.show()
```

2. *Activation Functions*

```
class MLP:
# Define activation functions
def relu(x):
    return np.maximum(0, x)

def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

- The relu function takes an input x and returns the maximum of 0 and x, effectively zeroing out negative values while leaving positive values unchanged. This introduces non-linearity in the model and helps the neural network learn complex relationships.
- The sigmoid function squashes the input x into a range between 0 and 1. It is often used in binary classification tasks because the output can be interpreted as a probability score, which allows the model to decide between two classes.

3. *Forward Pass Function*

```
# Forward pass function
def forward_pass(X, W1, b1, W2, b2):
    # Hidden layer computation
    z1 = X @ W1 + b1
    a1 = relu(z1)

    # Output layer computation
    z2 = a1 @ W2 + b2
    a2 = sigmoid(z2)  # Binary classification output between
0 and 1
    return a2
```

- **Hidden Layer:** The input X is multiplied by weights W1 and added to biases b1. The result is stored in z1. Afterward, the ReLU activation is applied, producing a1, the activated output of the hidden layer.
- **Output Layer:** Similarly, the output from the hidden layer, a1, is multiplied by W2 and added to biases b2, giving z2. The sigmoid activation is then applied to z2, producing a2. Since the sigmoid function is used, a2 represents the output of the neural network in the form of

probabilities between 0 and 1, making it suitable for binary classification.

*4. Forward Pass Function*

```
np.random.seed(2091)
# Initialize weights and biases
W1 = np.random.randn(2, 3)  # Weights between input and hidden
layer
b1 = np.random.randn(3)     # Biases for hidden layer
W2 = np.random.randn(3, 1)  # Weights between hidden and
output layer
b2 = np.random.randn(1)     # Bias for output layer
```

- W1: A matrix of shape (2, 3) representing weights between the input layer (with 2 features) and the hidden layer (with 3 neurons).
- b1: A vector of length 3 for biases in the hidden layer.
- W2: A matrix of shape (3, 1) representing weights between the hidden layer (3 neurons) and the output layer (1 output).
- b2: A scalar bias for the single output neuron.

*5. Prediction and Accuracy Calculation*

```
# Compute predictions and classification accuracy
predictions = forward_pass(X, W1, b1, W2, b2)
predicted_labels = (predictions > 0.5).astype(int).flatten()
accuracy = np.mean(predicted_labels == T)
print(f'Initial Classification Accuracy: {accuracy *
100:.2f}%')
```

- **Predictions:** The forward_pass function returns a probability output between 0 and 1.
- **Thresholding:** Any predictions greater than 0.5 are classified as class 1 and anything below or equal to 0.5 as class 0. This is done by (predictions > 0.5).astype(int).
- **Accuracy Calculation:** The predicted labels are compared to the true labels T, and the mean accuracy is calculated as the percentage of correctly classified points.

*6. Plotting the Decision Boundary*

```
def plot_decision_boundary(X, T, forward_pass, W1, b1, W2,
b2):
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, 200),
np.linspace(y_min, y_max, 200))
    grid = np.c_[xx.ravel(), yy.ravel()]
    probs = forward_pass(grid, W1, b1, W2,
b2).reshape(xx.shape)

    plt.contourf(xx, yy, probs, levels=[0, 0.5, 1], alpha=0.3,
colors=['blue', 'red'])
    plt.scatter(X[:, 0], X[:, 1], c=T, cmap='bwr',
edgecolor='k')
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.title(f'Decision Boundary of Initial MLP with Accuracy
= {accuracy*100}%')
    plt.show()

# Plot the decision boundary
plot_decision_boundary(X, T, forward_pass, W1, b1, W2, b2)
```

- **Meshgrid Creation:** The function first defines a grid over the feature space, using np.meshgrid to generate coordinates that cover the entire range of feature values with a small step size.
- **Grid Predictions:** For each point in the grid, the forward pass is applied to calculate predicted probabilities. The predictions are reshaped to match the grid's shape.
- **Contour Plot:** The decision boundary is then plotted as a contour plot, with levels defined at 0.5 (i.e., where the probability is 50%, which marks the boundary between classes).
- **Data Points:** Lastly, the actual data points from the dataset are plotted over the decision boundary for visual comparison, coloured based on their true class labels.

**Accuracy Calculation:** The predicted labels are compared to the true labels T, and the proportion of correct predictions is calculated as the model's accuracy. In this case, when the model is first initialized with random weights, the initial accuracy is found to be **63%**

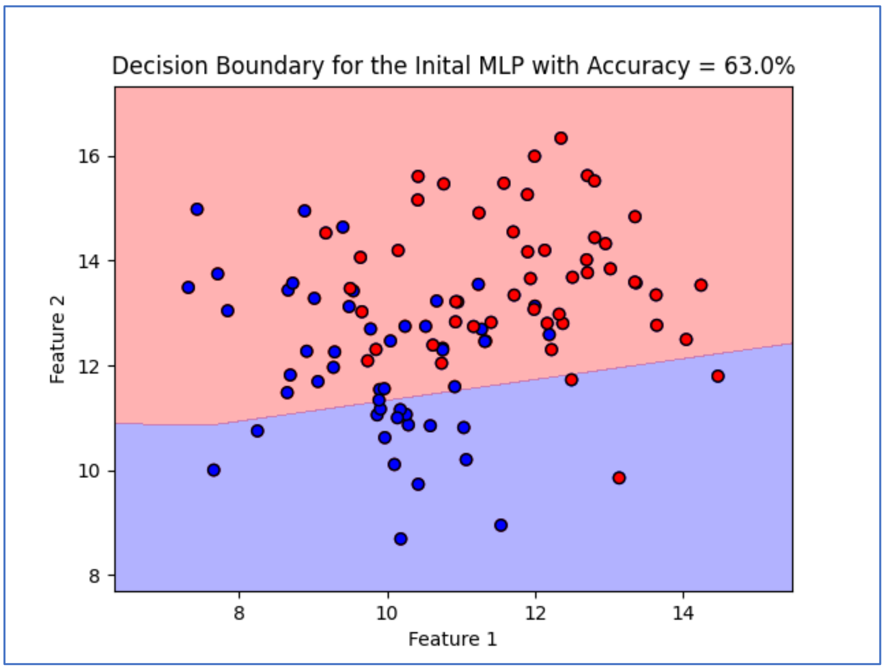


Figure 2. Decision Boundary of the MLP with 63% Accuracy

C) *Updating Weights of the Multi-Layer Perceptron using Backpropagation*

This code implements weight updates for an MLP using the Backpropagation algorithm with gradient descent, aiming to improve classification accuracy on a binary class dataset.

Here's a section-by-section explanation of the code:

1. *Data Loading and Visualization*

The data is loaded from `.npy` files, assumed to be `class1.npy` and `class2.npy`, which contain the points for class 1 and class 2, respectively.

```
import numpy as np
import matplotlib.pyplot as plt

# Load the data from npy files
X1 = np.load('class1.npy')
X2 = np.load('class2.npy')

# Labels for the classes
c1 = ['red'] * len(X1)  # Class 1
c2 = ['blue'] * len(X2) # Class 2

# Combine data
X = np.concatenate((X1, X2))
color = np.concatenate((c1, c2))

# Ground-truth labels: 0 for Class 1, 1 for Class 2
T = np.array([0] * len(X1) + [1] * len(X2))

# Plot the data
plt.scatter(X[:, 0], X[:, 1], marker='o', c=color)
plt.xlabel('Class 1')
plt.ylabel('Class 2')
plt.title('Scatter Plot of Loaded Binary Class Data')
plt.show()
```

2. *Activation Functions and Loss Function*

```
# Activation functions and their derivatives
def tanh(x):
    return np.tanh(x)

def tanh_derivative(x):
    return 1 - np.tanh(x) ** 2

def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

**Explanation:**

- **Tanh Activation Function**: Used for the hidden layer, the `tanh` function outputs values between -1 and 1, which helps in centreing the data and accelerates convergence. Its derivative, `1-np.tanh(x)**2,` is used for backpropagation to calculate gradients efficiently.
- **Sigmoid Activation Function**: Used in the output layer, the sigmoid function maps any real-valued number into the range (0, 1), making it suitable for binary classification tasks as it provides a probabilistic interpretation of the outputs.

Benefits Over other Activation Functions:

- **Tanh:** Preferred over ReLU in this context because it handles both positive and negative values, avoiding the issue of "dead neurons" that can occur with ReLU when values are negative.
- **Sigmoid:** Though alternatives like `softmax` are used in multi-class classification, sigmoid is apt for binary outcomes as in this case.

3. *Loss Function: Binary Cross-Entropy*

```
# Loss function: Binary cross-entropy
def binary_cross_entropy(y_true, y_pred):
    epsilon = 1e-15  # To prevent log(0)
    y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
    return -np.mean(y_true * np.log(y_pred) + (1 - y_true) *
np.log(1 - y_pred))
```

**Explanation:**

- **Binary Cross-Entropy Loss**: Measures the performance of a classification model whose output is a probability value between 0 and 1. This loss increases as the predicted probability diverges from the actual label.
- The use of an epsilon prevents taking the log of zero, which would otherwise cause computational errors.

**Benefits Over Other Loss Functions:**

This loss is ideal for binary classification as it penalizes confident and incorrect predictions more heavily than others, which helps in faster convergence and more reliable training outcomes.

4. *Forward Pass Function*

```
# Forward pass function
def forward_pass(X, W1, b1, W2, b2):
    z1 = X @ W1 + b1
    a1 = tanh(z1)
    z2 = a1 @ W2 + b2
    a2 = sigmoid(z2)
    return z1, a1, z2, a2
```

**Explanation**:

- **Forward Pass**: Computes the activations at each layer using the weights `(W1, W2)` and biases `(b1, b2)`. The `tanh` function is applied at the hidden layer, while sigmoid is applied at the output layer to get final predictions.
- This process mirrors the structure of a basic MLP and sets the groundwork for the subsequent backward pass.

5. *Backward Pass Function (Backpropagation)*

```
# Backward pass function (Backpropagation)
def backward_pass(X, T, z1, a1, z2, a2, W1, W2):
    # Output layer error and gradient
    output_error = a2 - T.reshape(-1, 1)
    dW2 = a1.T @ output_error
    db2 = np.sum(output_error, axis=0)
    # Hidden layer error and gradient
    hidden_error = (output_error @ W2.T) * tanh_derivative(z1)
    dW1 = X.T @ hidden_error
    db1 = np.sum(hidden_error, axis=0)

    return dW1, db1, dW2, db2
```

**Explanation:**

- **Output Layer Gradients**: Calculates the error (output_error) by comparing predictions (a2) to true labels (T). The gradients of the weights (dW2) and biases (db2) are then computed using the error and activations from the hidden layer (a1).

- **Hidden Layer Gradients**: Error is backpropagated to the hidden layer by multiplying the output error by the derivative of the activation function (tanh_derivative(z1)), and gradients for weights (dW1) and biases (db1) are computed.

6. *Training Loop with Gradient Descent*

```
# Training loop
for epoch in range(epochs):
    # Forward pass
    z1, a1, z2, a2 = forward_pass(X, W1, b1, W2, b2)

    # Compute loss
    loss = binary_cross_entropy(T, a2)

    # Backward pass
    dW1, db1, dW2, db2 = backward_pass(X, T, z1, a1, z2, a2,
W1, W2)

    # Update weights and biases using gradient descent
    W1 -= learning_rate * dW1
    b1 -= learning_rate * db1
    W2 -= learning_rate * dW2
    b2 -= learning_rate * db2

    # Calculate accuracy
    predictions = (a2 > 0.5).astype(int).flatten()
    accuracy = np.mean(predictions == T)

    # Keep track of the best accuracy
    if accuracy > best_accuracy:
        best_accuracy = accuracy
        best_W1, best_b1, best_W2, best_b2 = W1, b1, W2, b2

    # Print progress every 100 epochs
    if epoch % 100 == 0:
        print(f'Epoch  {epoch},  Loss:  {loss:.4f},  Accuracy:
{accuracy * 100:.2f}%')

print(f'Best   Classification   Accuracy   after   training:
{best_accuracy * 100:.2f}%')
```

**Explanation:**

- **Initialization**: Weights are initialized using He initialization (`np.random.randn`) for efficient training by accounting for layer sizes, which prevents vanishing/exploding gradients.
- **Training Loop**: For each epoch, the forward pass computes activations, loss is calculated using binary cross-entropy, gradients are computed in the backward pass, and weights and biases are updated using gradient descent.
- **Accuracy Calculation**: Compares predictions against true labels and tracks the best performance.

7. *Plotting the Decision Boundary*

```
# Plot the decision boundary for the best MLP
def plot_decision_boundary(X, T, forward_pass, W1, b1, W2, b2):
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx,   yy   =   np.meshgrid(np.linspace(x_min,   x_max,   200),
np.linspace(y_min, y_max, 200))
    grid = np.c_[xx.ravel(), yy.ravel()]
    _, _, _, probs = forward_pass(grid, W1, b1, W2, b2)
    probs = probs.reshape(xx.shape)
    plt.contourf(xx, yy, probs, levels=[0, 0.5, 1], alpha=0.3, colors=['blue', 'red'])
    plt.scatter(X[:, 0], X[:, 1], c=T, cmap='bwr', edgecolor='k')
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.title(f'Decision Boundary of MLP using Backprop with Accuracy = {best_accuracy*100}%')
    plt.show()

# Plot the decision boundary of the best MLP
plot_decision_boundary(X, T, forward_pass, best_W1, best_b1, best_W2, best_b2)
```

**Explanation:**

- **Decision Boundary Plotting**: Visualizes the decision boundary of the trained MLP by predicting over a mesh grid and contouring regions based on class probabilities.
- **Comparison**: By comparing this plot to the initial boundary, the effect of weight updates through backpropagation is evident, typically showing more refined decision boundaries that better separate the classes.

**Overall Impact of Backpropagation:** The updated MLP achieves higher classification accuracy (from 63% to 84%), demonstrating that backpropagation effectively adjusts the weights to minimize loss and improve decision-making capabilities of the model. This comparison illustrates the critical role of gradient descent in refining the performance of neural networks.

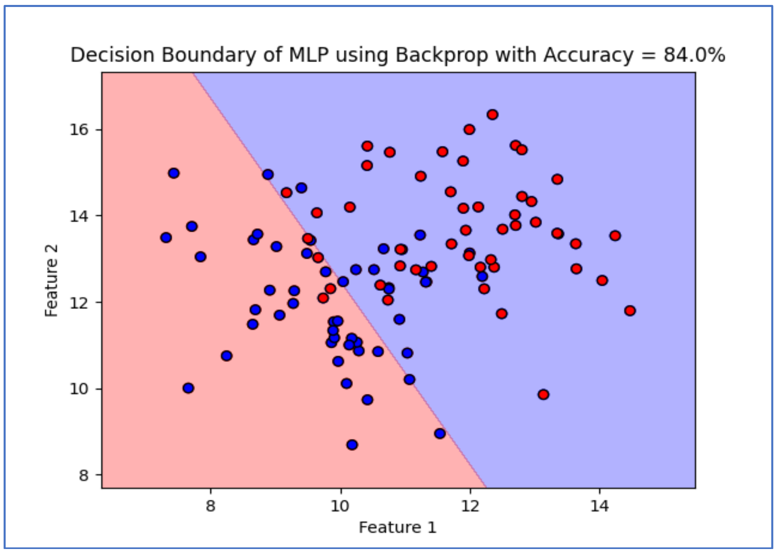


Figure 3. Decision Boundary of the MLP using Backpropagation with 84% Accuracy

## Comparison between Backpropagation and Random Initialization method:

The backpropagation process, coupled with appropriate activation functions and loss calculation, methodically tuned the weights and biases, resulting in a model that adapted its decision boundary to the specific nuances of the given data. This led to a notable improvement in classification accuracy from the initial 63% (random initialization) to 84% (post-backpropagation). Table 1. summarizes how each parameter contributed to the effective learning process:

| Parameter | Initial Method (Random Initialization) | Updated Method (Backpropagation) | Impact on Results |
|---|---|---|---|
| **Weights (W1, W2)** | Random weights lead to arbitrary decision boundaries, resulting in a low accuracy of 63%. | Adjusted weights through backpropagation to minimize loss and improve classification. | Improved weights create better-aligned decision boundaries, boosting accuracy to 84%. |
| **Biases (b1, b2)** | Random biases provide no meaningful contribution, affecting consistency in predictions. | Updated biases adjust activation thresholds, improving model responsiveness. | Proper bias tuning refined decision regions, reducing misclassifications near boundaries. |
| **Activation Functions (tanh, sigmoid)** | Ineffective use without structured updates, limiting the model's ability to capture data patterns. | `tanh` helps model non-linear patterns in the hidden layer; `sigmoid` provides clear output probabilities. | These activations enabled better separation of classes, enhancing decision boundaries. |
| **Loss Function (Binary Cross-Entropy)** | Not used, no guidance for minimizing classification errors. | Directly measures errors and guides updates to reduce misclassification. | Focused error reduction improved the model's ability to classify correctly. |

Table 1. Backpropagation vs. Random Initialization: Performance Summary

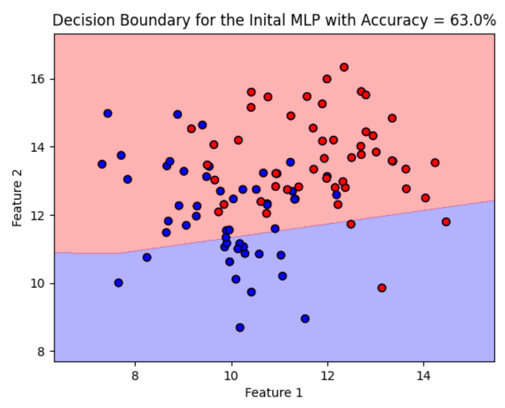
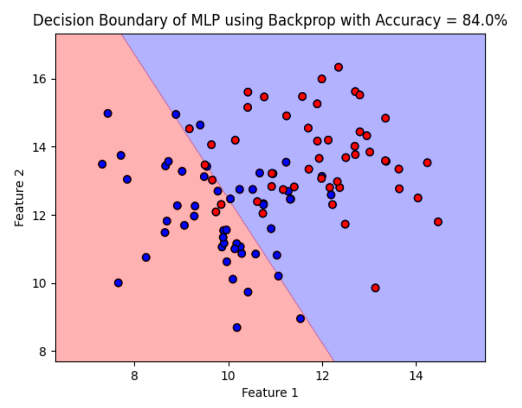
| Decision Boundary with Random Initialization | Decision Boundary with Backpropagation |
|---|---|
|  |  |

Figure 4. Backpropagation vs. Random Initialization: Performance Summary

## PART 2 - Classification with Radial Basis Function (RBF) Network with Random RBF Centres

### *A) Implement an RBF Network with Gaussian function and 3 neurons to classify data*

The RBF (Radial Basis Function) network is a type of artificial neural network that uses radial basis functions as activation functions for its hidden neurons. It is particularly effective for problems like classification.

*1. Gaussian RBF Function and RBF Centres Initialization*

```
import numpy as np
import matplotlib.pyplot as plt

# Load the data from npy files
X1 = np.load('class1.npy')
X2 = np.load('class2.npy')

# Combine data and labels
X = np.concatenate((X1, X2))
T = np.array([0] * len(X1) + [1] * len(X2))  # Ground-truth
labels: 0 for Class 1, 1 for Class 2

# Plot the data
c1 = ['blue'] * len(X1)  # Class 1
c2 = ['red'] * len(X2) # Class 2
color = np.concatenate((c1, c2))
plt.scatter(X[:, 0], X[:, 1], marker='o', c=color)
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Scatter Plot of Loaded Binary Class Data')
plt.show()

# Gaussian RBF function
def rbf(x, centre, sigma=1):
    return np.exp(-np.linalg.norm(x - centre, axis=1) ** 2 /
(2 * sigma ** 2))

# Randomly set RBF centres within the range of the data
```

```
np.random.seed(42)  # For reproducibility
x_min, x_max = X[:, 0].min(), X[:, 0].max()
y_min, y_max = X[:, 1].min(), X[:, 1].max()

centres = np.random.rand(3, 2) * [x_max - x_min, y_max -
y_min] + [x_min, y_min]

# Plot the data with RBF centres
plt.scatter(X[:, 0], X[:, 1], marker='o', c=color)
plt.scatter(centres[:, 0], centres[:, 1], marker='x',
c='green', s=100, label='RBF Centres')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('RBF Centres within Data Range')
plt.legend()
plt.show()
```

Explanation:

- **Gaussian RBF Function:** This function computes the output of each RBF neuron based on the distance of the input data point from the centre. The Gaussian function maps this distance into a range between 0 and 1, with a peak at the centre and decaying as the distance increases. The parameter sigma controls the width of the Gaussian, affecting how spread out the response is.
- **RBF Centres Initialization:** Centres are randomly initialized within the feature range of the data to ensure they are well distributed across the input space. Proper placement of these centres is crucial because they define the "receptive fields" that determine how input points activate the RBF neurons.

  **np.random.rand(3, 2):** This generates a `3x2` matrix of random numbers uniformly distributed between 0 and 1. Each row represents an RBF centre, and each column represents one of the two features (dimensions) in the data. The shape `(3, 2)` means there are 3 centres, each with 2 coordinates (since the data points are two-dimensional).

- **Impact:** The Gaussian RBF function enables the network to handle nonlinear separations by transforming the input space into a feature space where classes are linearly separable. Correctly positioned centres are essential for capturing the data's underlying structure.

2. *Constructing the RBF Matrix*

```python
# Construct the RBF matrix
def construct_rbf_matrix(X, centres, sigma=1):
    G = np.zeros((X.shape[0], centres.shape[0]))
    for i, centre in enumerate(centres):
        G[:, i] = rbf(X, centre, sigma)
    return G

# Construct the RBF matrix for the data
G = construct_rbf_matrix(X, centres, sigma=1)
```

**Explanation:**

- The RBF matrix G is constructed to represent how each input sample is transformed by the RBF layer. Each element of the matrix is the output of the RBF function for a specific input point and a specific RBF centre.
- **Role of the RBF Matrix:** This matrix acts as the hidden layer of the RBF network. It encapsulates how each input point activates the RBF neurons, effectively transforming the input data into a new feature space.
- By converting the input data into a new space where the data points are more distinguishable, the RBF matrix makes it easier for the subsequent layers (or in this case, a single linear output neuron) to separate the classes.

3. *Calculating Weights Using Least Squares Estimation*

```python
# Compute the weights using Least Squares Estimation
W = np.linalg.pinv(G) @ T
```

**Explanation:**

- **Least Squares Estimation:** This method computes the optimal weights connecting the RBF hidden layer to the output neuron by minimizing the squared error between the network's outputs and the actual labels.

- **Working:** By using the pseudo-inverse of the RBF matrix (G), the algorithm finds the weights that best map the transformed inputs to the target outputs. This is a direct solution without iterative optimization, making it quick to compute.
- The accuracy of the RBF network heavily depends on these weights as they directly influence the final classification. Properly calculated weights ensure that the transformed features from the RBF layer are correctly mapped to the output classes.

*4. RBF Network Forward Pass and Classification Accuracy*

```
# Forward pass function for RBF network
def rbf_network(X, centres, W, sigma=1):
    G = construct_rbf_matrix(X, centres, sigma)
    return G @ W

# Predict and calculate classification accuracy
predictions = rbf_network(X, centres, W)
predicted_labels = (predictions > 0.5).astype(int)
accuracy = np.mean(predicted_labels == T)

print(f'RBF Network Classification Accuracy: {accuracy * 
100:.2f}%')
```

**Explanation:**

- **Forward Pass**: In this step, the RBF network generates its predictions by multiplying the RBF matrix G by the learned weights W. The RBF matrix captures the transformed input features, and when combined with the weights, it gives the final output of the network.
- **Classification Accuracy**: To classify the predictions, the output values are compared against a threshold of 0.5. Any value greater than 0.5 is classified as a 1, while values below this threshold are classified as 0. The accuracy is simply the percentage of data points that were correctly classified, comparing these predicted labels to the actual target labels.
- **Accuracy**: A higher accuracy indicates that the RBF network is effectively capturing the underlying patterns in the data. It suggests that the network's transformation of the input features and

subsequent linear mapping to the output is successfully separating the two classes.

*5. Plotting the Decision Boundary of the RBF Network*

```
# Plot the decision boundary of the RBF network
def plot_decision_boundary_rbf(X, T, centres, W, sigma=1):
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, 200),
np.linspace(y_min, y_max, 200))
    grid = np.c_[xx.ravel(), yy.ravel()]
    probs = rbf_network(grid, centres, W,
sigma).reshape(xx.shape)

    plt.contourf(xx, yy, probs, levels=[0, 0.5, 1],
alpha=0.3, colors=['blue', 'red'], linestyles='--')
    plt.contour(xx, yy, probs, levels=[0.5], colors='black',
linewidths=1, linestyles='--')
    plt.scatter(X[:, 0], X[:, 1], c=T, cmap='bwr',
edgecolor='k')
    plt.scatter(centres[:, 0], centres[:, 1], marker='x',
c='green', s=100, label='RBF Centres')
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.title(f'Decision Boundary of Inital RBF Network with
Accuracy = {accuracy*100}%')
    plt.legend()
    plt.show()

# Plot the decision boundary of the RBF network
plot_decision_boundary_rbf(X, T, centres, W, sigma=1)
```

**Explanation:**

- **Visualizing the Decision Boundary**: This function plots the decision boundary of the RBF network. It evaluates the network's output over a grid of input values that span the feature space, allowing us to see how the network separates the two classes.
- **Contour Plot**: The contour plot shows regions corresponding to the two classes, with a boundary at the threshold of 0.5, where the

network is unsure of the classification. The decision boundary line visually separates the two classes, and we can see whether the network is making clean separations.

- **Understanding the Decision Boundary**: This plot gives a visual representation of how well the RBF network distinguishes between classes. A clear and well-defined boundary indicates that the model is effectively capturing the relationship between the features and the class labels.

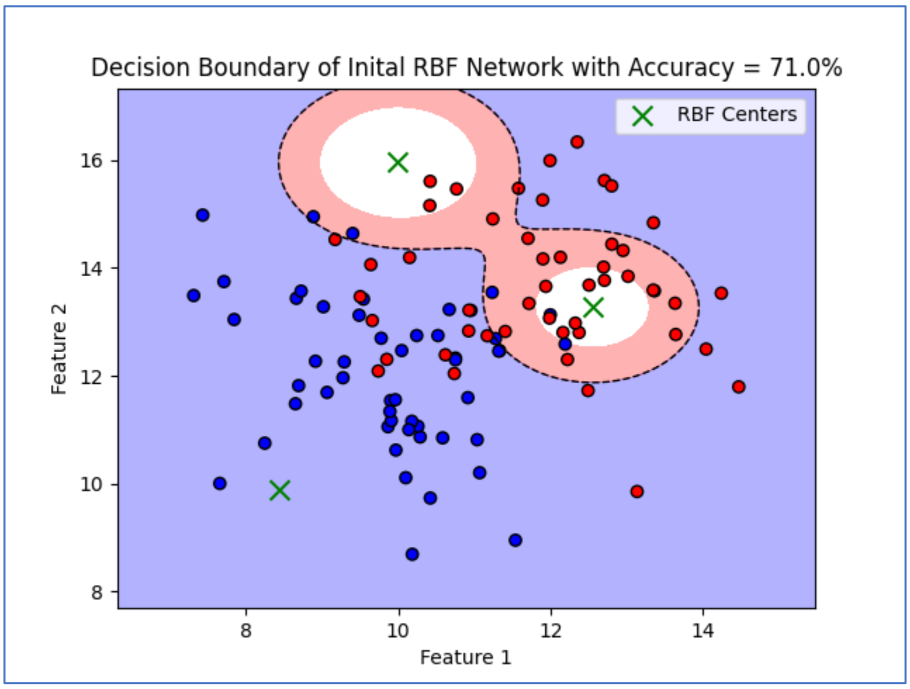


Figure 5. Decision Boundary of the RBF Network using 3 Centres with 71% Accuracy

## Comparison and Discussion:

- **Performance of the RBF Network**: The RBF network achieved a classification accuracy of **71%**, which is a notable improvement over the initial MLP's accuracy of 63%. The ability of the RBF network to model nonlinear relationships using radial basis functions (specifically, Gaussian functions) allows it to capture more complex patterns in the data, leading to better performance.
- **Limitations of the RBF Network**: While the RBF network performs well, its success largely depends on the placement of the RBF centres. If the centres are poorly chosen, the network may struggle to find an optimal decision boundary. Additionally, MLPs trained via backpropagation can outperform RBF networks, especially when the MLP continuously adjusts its weights through iterative learning.

## *B) Implement an RBF Network with Gaussian function and 6 neurons to classify data*

In this part, the RBF network is extended by increasing the number of hidden neurons (RBF centres) from 3 to 6, which significantly impacts the network's classification performance.

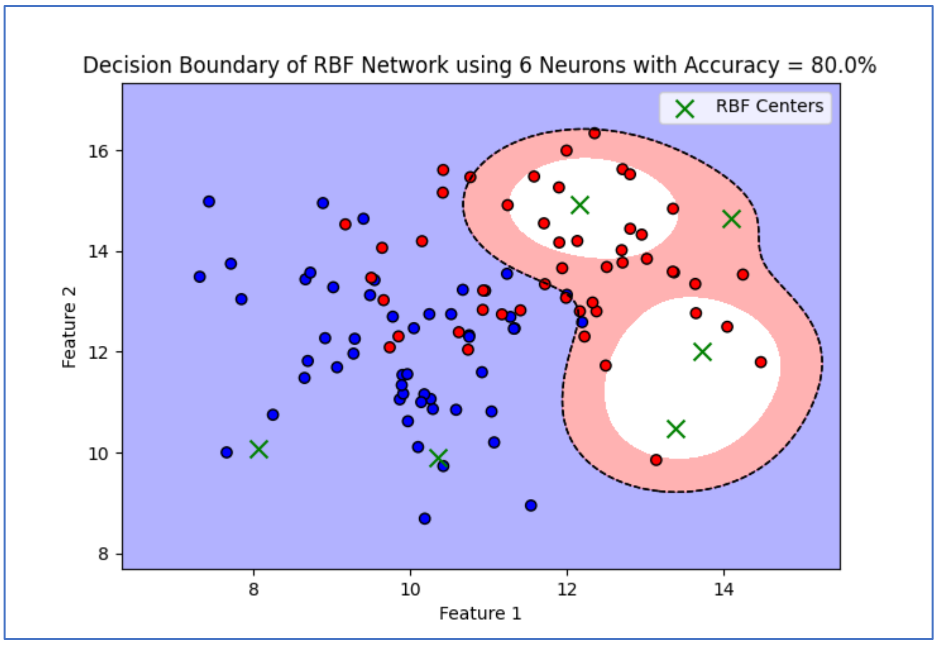


Figure 6. Decision Boundary of the RBF Network using 6 Centres with 80% Accuracy

- **Improvement in Accuracy:** With 6 centres, the network achieves an accuracy of **80%**, compared to 71% with only 3 centres. This improvement shows that adding more RBF centres gives the network more flexibility to model the underlying structure of the data. This improvement demonstrates that additional hidden neurons can better capture the complexity of the data.

**Effect of More Hidden Neurons:**

- **Enhanced Flexibility:** Adding more RBF centres allows the network to better adapt to the data. More centres create smaller, localized regions of influence in the input space, enabling the model to capture finer details and complex patterns in the data.
- **Improved Decision Boundaries:** With 6 centres, the decision boundary becomes more nuanced and accurately reflects the nonlinear separations required for proper classification. This leads to

fewer misclassifications, especially near the boundary between classes.

**Balance Between Complexity and Generalization:** It's important to find the right balance between the number of hidden neurons (RBF centres) and the complexity of the data. With only 3 centres, the network risks underfitting, as it cannot capture all the relevant patterns in the data. However, too many centres might cause overfitting. In this case, 6 centres seem to strike a good balance, resulting in the highest accuracy of 80%.

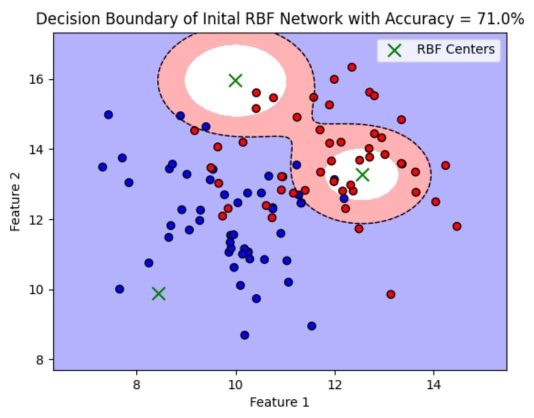
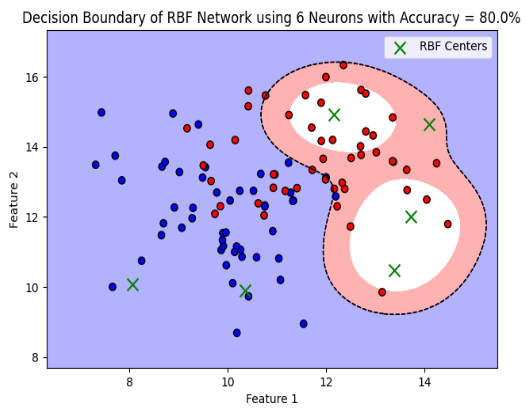
| RBF with 3 Neurons | RBF with 6 Neurons |
|---|---|
|  |  |

Figure 7. Comparison b/w Decision Boundary using three and six RBF Neurons

**Overall Comparison:** The RBF network with 6 centres demonstrates the benefits of additional hidden neurons, clearly outperforming the simpler RBF network with fewer centres and approaching the performance of a well-trained MLP. This highlights the importance of model complexity in achieving higher classification accuracy.