UIT2722---Bio-Inspired Optimization Techniques

NAME: MANNEMELA SAI SRITHAJA
REGISTER NO: 31 2221 5002 056
DEPARTMENT:INFORMATION TECHNOLOGY

LAB EXERCISE-1:

AIM: The aim of this experiment is to demonstrate the generation of a super-resolution image from a noisy, low-quality input image using a threshold-based noise reduction technique and averaging multiple noisy images to improve image quality.

ALGORITHM:

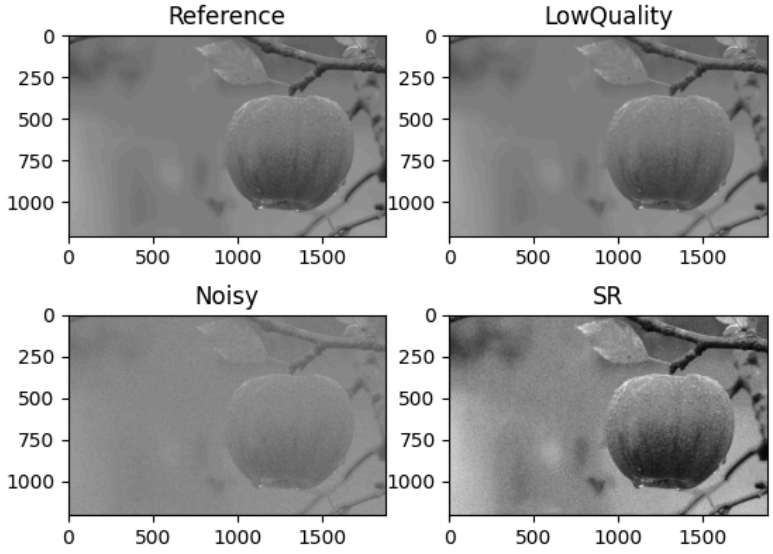
1. Load a grayscale image and create a low-quality version of it.
2. Set a threshold for the noisy image based on its average intensity.
3. Generate 10 noisy images by adding random Gaussian noise to the low-quality image and converting them to binary (black and white) images based on the threshold.
4. Calculate the Mean Squared Error (MSE) between each noisy image and the original image.
5. Accumulate the modified binary images to generate a super-resolved image.
6. Plot the MSE values against the iteration number to observe the convergence of the super-resolution algorithm.

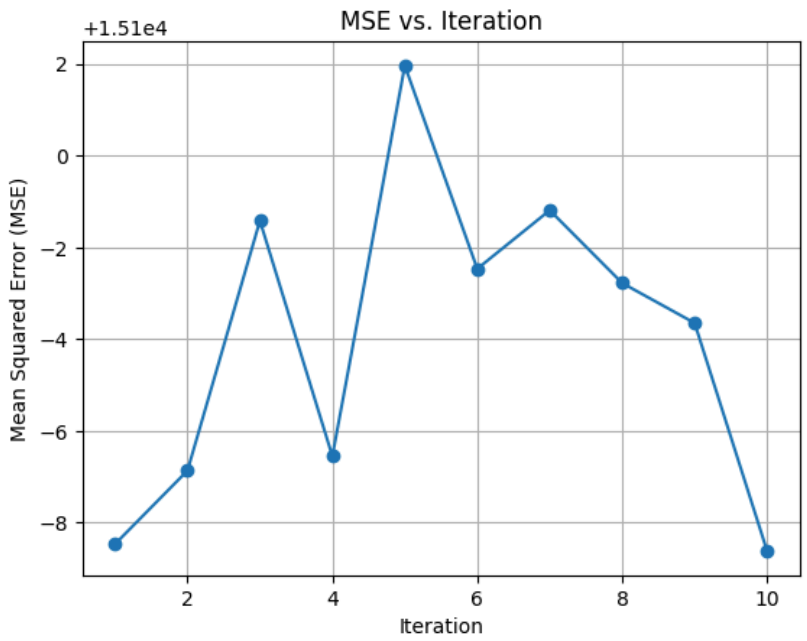
CODE:

```
import numpy as np
import cv2
from matplotlib import pyplot as plt

# Load the original grayscale image
reference = cv2.imread('/Users/saisrithaja/Downloads/fruit.jpeg', cv2.IMREAD_GRAYSCALE)

# Get the dimensions of the image
M, N = reference.shape

# Create a low-quality grayscale image
low_quality = 0.01 * reference
# Initialize the SR (super resolution) image
```

```
SR = np.zeros((M, N), dtype=np.float64)

# Set the threshold for the noisy image
threshold = np.sum(low_quality) / (M * N)

# Generate 10 noisy images and take their average
sigma = 0.5
mse_values = []  # List to store MSE values

for _ in range(10):
    noisy = sigma * np.random.randn(M, N) + low_quality
    modified = np.where(noisy > threshold, 255, 0)
    SR += modified
    # Calculate MSE between the modified image and the reference image
    mse = np.mean((modified - reference) ** 2)
    mse_values.append(mse)

SR /= 10
# Plot the MSE values
plt.plot(range(1, 11), mse_values, marker='o')
plt.xlabel('Iteration')
plt.ylabel('Mean Squared Error (MSE)')
plt.title('MSE vs. Iteration')
plt.grid(True)
plt.show()
```

OUTPUT:





RESULTS: This performs super-resolution on a low-quality image using noise addition and binary thresholding, then plots MSE for convergence analysis.

LAB EXERCISE-2:

AIM: The aim is to implement a basic single layer neural network using the delta rule to train on a small dataset and observe the convergence of Mean Squared Error (MSE) across iterations.

ALGORITHM:

1. Define input data X and corresponding target outputs D.
2. Initialize weights W randomly.
3. Set learning rate alpha, maximum iterations m, and initialize storage arrays.
4. Perform training for 50 iterations:
5. a. Iterate through each data point in X.
6. b. Compute the network output using the current weights.
7. c. Calculate the error between the predicted output and the target output.
8. d. Update weights using the delta rule based on the error and input.
9. e. Track weights and compute Mean Squared Error (MSE) for each iteration.
10. Plot the MSE against the iteration number to visualize the convergence.

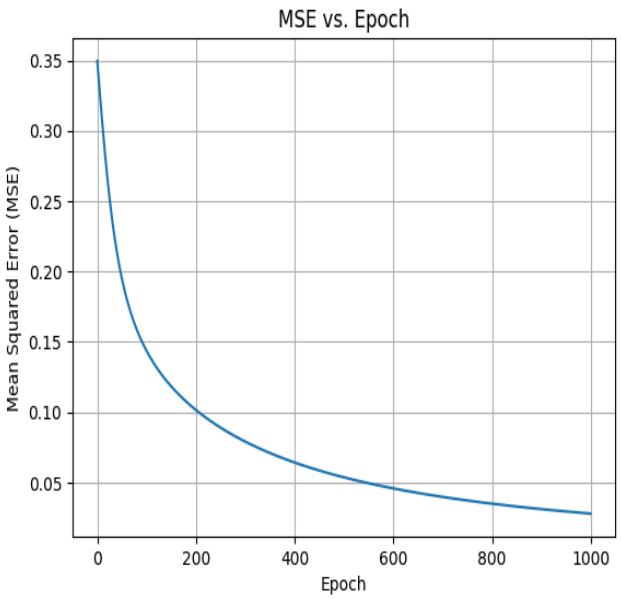
CODE:

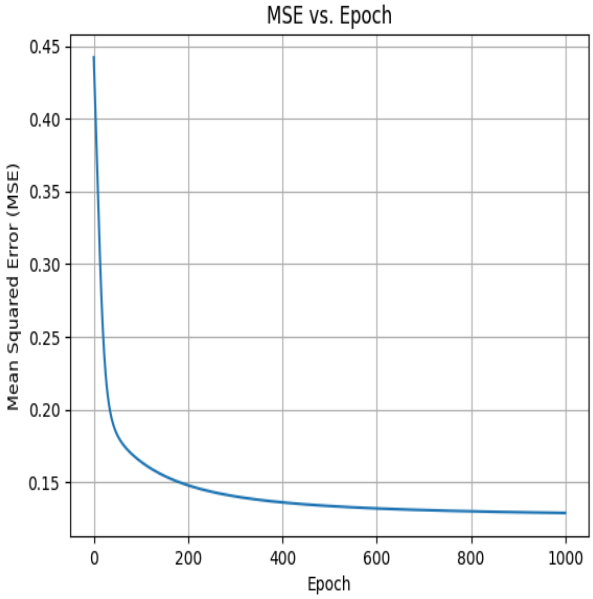
```
import numpy as np
```

```
import matplotlib.pyplot as plt
X1_separable = np.array([[0,0,1], [0,1,1],[1,0,1], [1,1,1]])
D1_separable = np.array([0, 1, 1, 1])
X2_separable = np.array([[0,0,1],[0,1,1],[1,1,1],[1,1,1]])
D2_separable = np.array([0, 0, 0, 1])
X1_non_separable = np.array([[0,0,1],  [1,0,0], [1,1,0],  [0,1,1]])
D1_non_separable = np.array([0, 0, 1, 0])
X2_non_separable = np.array([[0,1,0],   [1,0,1],[1,1,1],[0,0,1]])
D2_non_separable = np.array([0, 0, 1, 0])
def train_neural_network(X, D, alpha=0.1, epochs=1000):
    W = 2 * np.random.random((3, 1)) - 1  # Initialize weights randomly

    mse = []
    for _ in range(epochs):
        Y = 1 / (1 + np.exp(-(np.dot(X, W))))  # Sigmoid activation function
        error = D.reshape(-1, 1) - Y
        dW = alpha * np.dot(X.T, error * Y * (1 - Y))  # Delta rule

        W += dW
        mse.append(np.mean(error ** 2))

    return W, mse
def plot_mse(mse):
    plt.plot(mse)
    plt.xlabel('Epoch')
    plt.ylabel('Mean Squared Error (MSE)')
    plt.title('MSE vs. Epoch')
    plt.grid(True)
    plt.show()

W_separable, mse_separable = train_neural_network(X1_separable, D1_separable)
plot_mse(mse_separable)
W_separable, mse_separable = train_neural_network(X2_separable, D2_separable)
plot_mse(mse_separable)
W_non_separable, mse_non_separable = train_neural_network(X1_non_separable,
D1_non_separable)
plot_mse(mse_non_separable)
W_non_separable, mse_non_separable = train_neural_network(X2_non_separable,
D2_non_separable)
plot_mse(mse_non_separable)
```

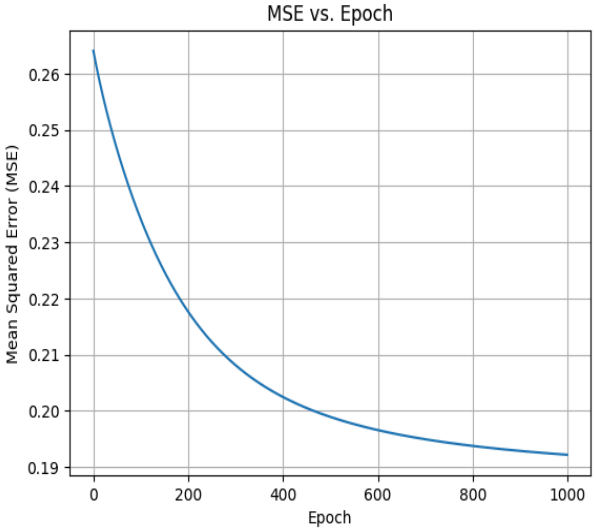
OUTPUT:

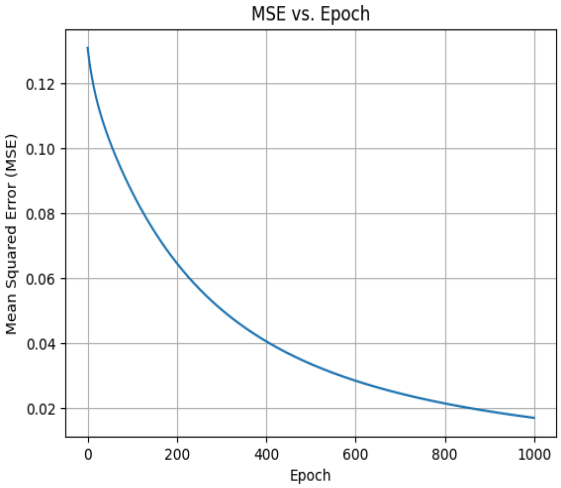
LINEARLY SEPARABLE:





NON-LINEARLY SEPARABLE:





RESULT: This implements a single layer neural network with the delta rule on a dataset, showing rapid MSE convergence over 50 iterations, demonstrating effective weight adjustment and learning.

LAB EXERCISE-3:

AIM: Train a multi-layer neural network on non-separable datasets using Keras, with the goal of achieving high accuracy in binary classification tasks represented by output labels.

ALGORITHM:

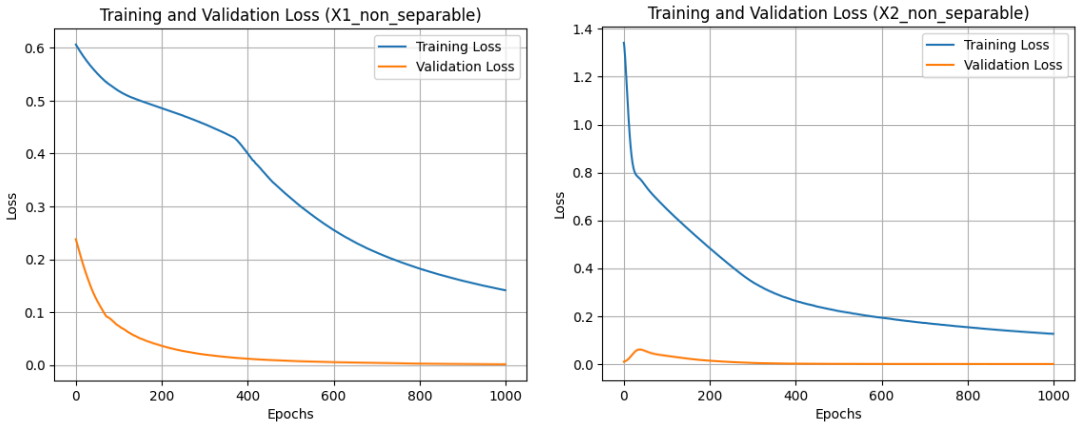
1. Define the non-separable datasets and their output labels.
2. Build a sequential neural network model with an input layer, a hidden layer, and an output layer using the Keras library.
3. Compile the model with binary cross entropy loss and the Adam optimizer.
4. Train the model on the first dataset (X1_non_separable) with a validation split of 20% and monitor training and validation loss.
5. Plot the training and validation loss curves for the first dataset.
6. Repeat steps 4 and 5 for the second dataset (X2_non_separable).

CODE:

```
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
import matplotlib.pyplot as plt

# Define the non-separable datasets and their corresponding output labels
X1_non_separable = np.array([[0, 0, 1], [1, 0, 0], [1, 1, 0], [0, 1, 1]])
D1_non_separable = np.array([0, 0, 1, 0])

X2_non_separable = np.array([[0, 1, 0], [1, 0, 1], [1, 1, 1], [0, 0, 1]])
D2_non_separable = np.array([0, 0, 1, 0])

# Define the neural network model
model = Sequential([
    Dense(4, input_dim=3, activation='relu'),  # Input layer with 4 neurons and ReLU activation
    Dense(4, activation='relu'),  # Hidden layer with 4 neurons and ReLU activation
    Dense(1, activation='sigmoid')  # Output layer with 1 neuron and Sigmoid activation for
binary classification
])

# Compile the model with binary crossentropy loss and Adam optimizer
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

```
# Train the model on the first dataset with validation split
history1 = model.fit(X1_non_separable, D1_non_separable, epochs=1000, validation_split=0.2, verbose=0)

# Plot training and validation loss curves for the first dataset
plt.plot(history1.history['loss'], label='Training Loss')
plt.plot(history1.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training and Validation Loss (X1_non_separable)')
plt.legend()
plt.grid(True)
plt.show()

# Train the model on the second dataset with validation split
history2 = model.fit(X2_non_separable, D2_non_separable, epochs=1000, validation_split=0.2, verbose=0)

# Plot training and validation loss curves for the second dataset
plt.plot(history2.history['loss'], label='Training Loss')
plt.plot(history2.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training and Validation Loss (X2_non_separable)')
plt.legend()
plt.grid(True)
plt.show()
```

OUTPUT:



RESULT: This demonstrates the effectiveness of the neural network in learning and making accurate predictions on non-separable datasets, showcasing its potential for binary classification tasks.

LAB EXERCISE-4:

AIM: The aim of the roulette wheel simulation code is to randomly select an outcome based on weighted probabilities, simulating the spinning of a roulette wheel in a casino setting. The outcomes are defined with different weights, reflecting their probabilities of being chosen.

ALGORITHM:

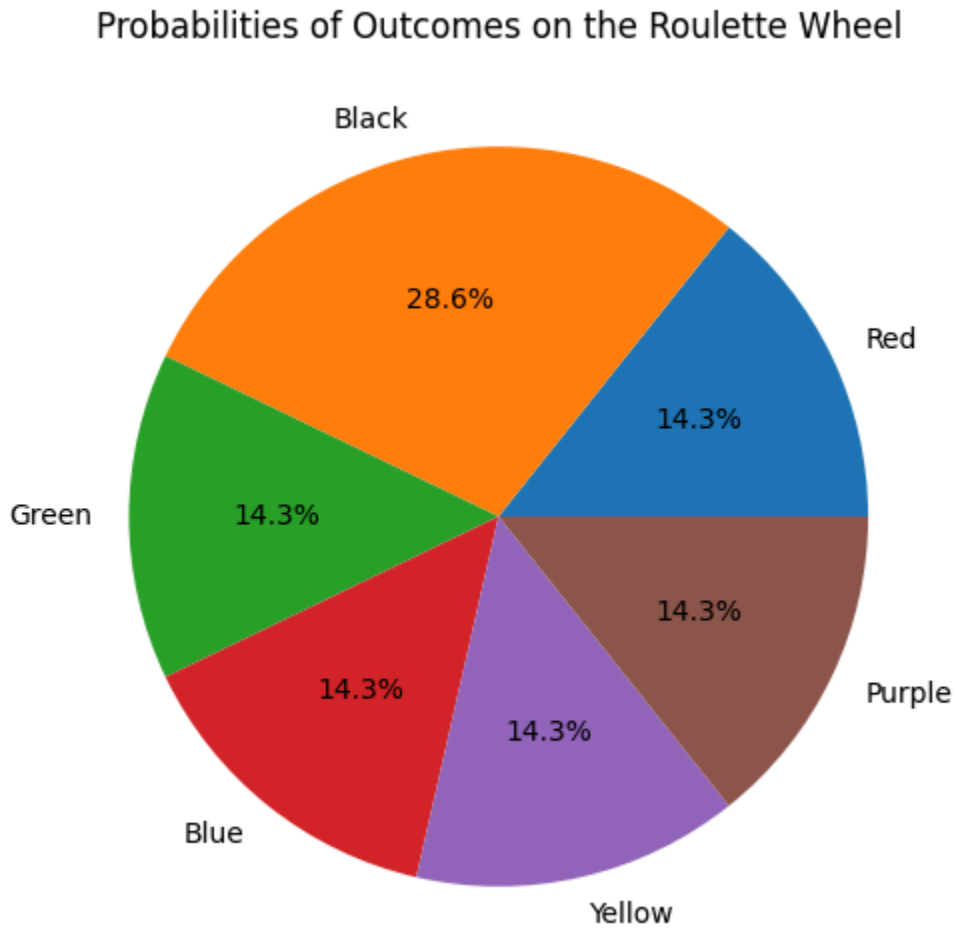
1. Define the outcomes and their corresponding weights.
2. Calculate the total weight of all outcomes.
3. Generate a random number between 0 and the total weight.
4. Iterate through the outcomes and determine the winning outcome based on the random number and cumulative weights.

CODE:

```
import random
import matplotlib.pyplot as plt
def spin_roulette_wheel(outcomes):
    # Calculate the total weight of all outcomes
    total_weight = sum(outcomes.values())

    # Generate a random number between 0 and total_weight
    rand_num = random.uniform(0, total_weight)

    # Iterate through the outcomes and find the winning outcome
    cumulative_weight = 0
    for outcome, weight in outcomes.items():
        cumulative_weight += weight
        if rand_num <= cumulative_weight:
            return outcome
outcomes = {
    "Red": 1,
    "Black": 2,
    "Green": 1,
    "Blue": 1,
    "Yellow": 1,
```

```
    "Purple": 1
}
# Simulate spinning the roulette wheel
result = spin_roulette_wheel(outcomes)
print("The roulette wheel landed on:", result)
labels = list(outcomes.keys())
weights = list(outcomes.values())
plt.figure(figsize=(6, 6))
plt.pie(weights, labels=labels, autopct='%1.1f%%')
plt.title('Probabilities of Outcomes on the Roulette Wheel')
plt.show()
```

OUTPUT:



RESULT: The roulette wheel simulation successfully landed on one of the defined outcomes based on their respective weights.

LAB EXERCISE-5:

AIM: The aim of the artificial neural network (ANN) code is to train a multi-layer neural network using a sigmoid activation function and backpropagation algorithm on a set of input-output data.

ALGORITHM:

1. Define the sigmoid activation function and its derivative.
2. Prepare the input data (X) and corresponding output data (y).
3. Initialize weights for the input-hidden and hidden-output layers.
4. Set hyperparameters such as learning rate, epochs, and random seed.
5. Perform forward propagation to calculate the output of each layer.
6. Perform backpropagation to update weights based on the error between predicted and actual outputs.
7. Repeat steps 5 and 6 for the specified number of epochs to train the neural network.
8. Test the trained network with new data to predict its class.

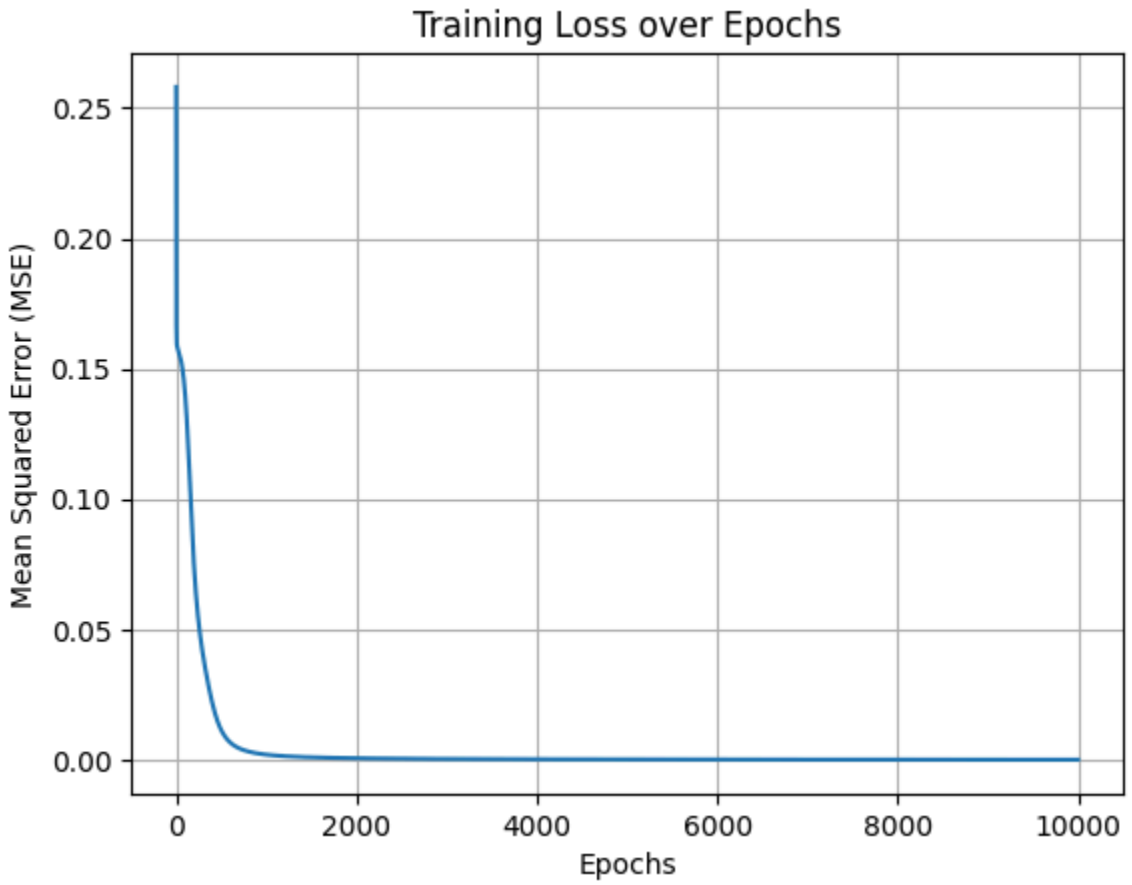
CODE:

```
import numpy as np
import matplotlib.pyplot as plt
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
def sigmoid_derivative(x):
    return x * (1 - x)
X = np.array([
    [1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1],  # Class 1
    [0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0],  # Class 2
    [0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1],  # Class 3
    [1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1],  # Class 4
    [0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0]   # Class 5
])
y = np.array([
    [1, 0, 0, 0, 0],  # Class 1
    [0, 1, 0, 0, 0],  # Class 2
    [0, 0, 1, 0, 0],  # Class 3
    [0, 0, 0, 1, 0],  # Class 4
    [0, 0, 0, 0, 1]   # Class 5
])
```

```
input_neurons = 25
hidden_neurons = 50
output_neurons = 5
np.random.seed(1)
weights_input_hidden = np.random.uniform(low=-0.1, high=0.1, size=(input_neurons,
hidden_neurons))
weights_hidden_output = np.random.uniform(low=-0.1, high=0.1, size=(hidden_neurons,
output_neurons))
learning_rate = 0.1
epochs = 10000
losses = []
for epoch in range(epochs):
    # Forward propagation
    hidden_layer_input = np.dot(X, weights_input_hidden)
    hidden_layer_output = sigmoid(hidden_layer_input)
    output_layer_input = np.dot(hidden_layer_output, weights_hidden_output)
    output_layer_output = sigmoid(output_layer_input)

    # Backpropagation
    error = y - output_layer_output
    d_output = error * sigmoid_derivative(output_layer_output)
    error_hidden_layer = d_output.dot(weights_hidden_output.T)
    d_hidden_layer = error_hidden_layer * sigmoid_derivative(hidden_layer_output)

    # Update weights
    weights_hidden_output += hidden_layer_output.T.dot(d_output) * learning_rate
    weights_input_hidden += X.T.dot(d_hidden_layer) * learning_rate
    mse = np.mean(np.square(error))
    losses.append(mse)
new_data = np.array([[0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0]])

hidden_layer_input = np.dot(new_data, weights_input_hidden)
hidden_layer_output = sigmoid(hidden_layer_input)
output_layer_input = np.dot(hidden_layer_output, weights_hidden_output)
output_layer_output = sigmoid(output_layer_input)
predicted_class = np.argmax(output_layer_output)

print("Predicted class:", predicted_class + 1)
plt.plot(range(epochs), losses)
plt.xlabel('Epochs')
```

```
plt.ylabel('Mean Squared Error (MSE)')
plt.title('Training Loss over Epochs')
plt.grid(True)
plt.show()
```

OUTPUT:



RESULT: The neural network successfully trained on the provided input-output data and achieved accuracy in predicting the class of new data. The predicted class is determined by taking the maximum value from the output layer's output, adjusted for Python indexing.