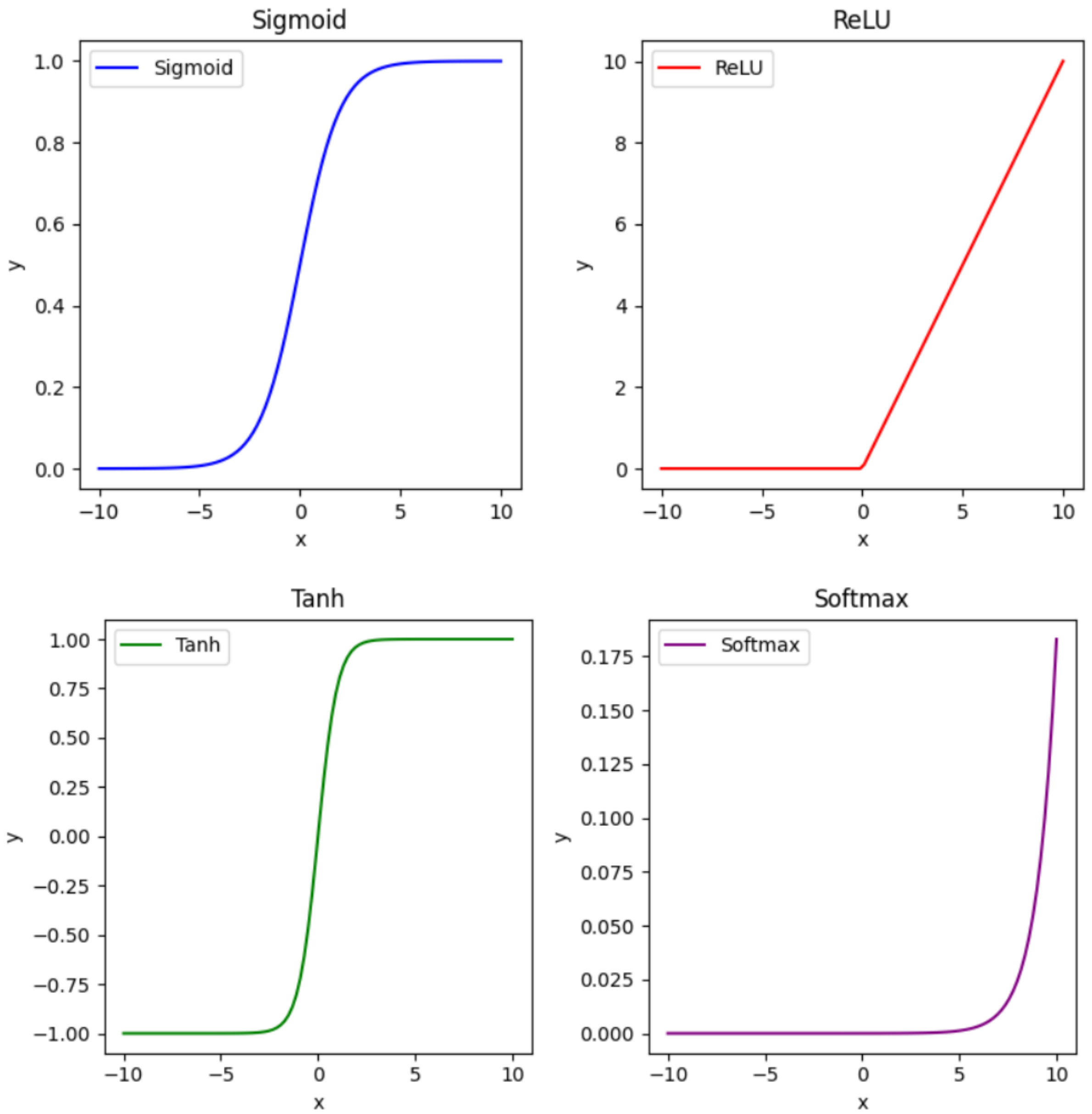
Assignment 1

Roll No.:08
Code:

```
import numpy as np

import matplotlib.pyplot as plt

# Define activation functions

def sigmoid(x):

    return 1 / (1 + np.exp(-x))

def relu(x):

    return np.maximum(0, x)

def tanh(x):

    return np.tanh(x)

def softmax(x):

    exp_x = np.exp(x - np.max(x))  # For numerical stability

    return exp_x / np.sum(exp_x)

# Create x values

x = np.linspace(-10, 10, 100)

# Create a figure with 2x2 subplots

fig, axs = plt.subplots(2, 2, figsize=(8, 8))

# Plot each activation function

axs[0, 0].plot(x, sigmoid(x), label="Sigmoid", color="blue")

axs[0, 0].set_title('Sigmoid')

axs[0, 1].plot(x, relu(x), label="ReLU", color="red")

axs[0, 1].set_title('ReLU')

axs[1, 0].plot(x, tanh(x), label="Tanh", color="green")

axs[1, 0].set_title('Tanh')

axs[1, 1].plot(x, softmax(x), label="Softmax", color="purple")

axs[1, 1].set_title('Softmax')

# Add common axis labels

for ax in axs.flat:
```

```
    ax.set(xlabel='x', ylabel='y')
    ax.legend()  # Add legend
# Adjust layout for better visibility
plt.tight_layout()
plt.show()
```

Output:



Assignment 2

Roll No.:08

Code:

```
import numpy as np

def linear_threshold_gate(dot, T):
    """Returns the binary threshold output"""
    if dot >= T:
        return 1
    else:
        return 0

input_table = np.array([
    [0, 0],  # both no
    [0, 1],  # one no, one yes
    [1, 0],  # one yes, one no
    [1, 1]   # both yes])

print(f'Input Table:\n{input_table}\n')

weights = np.array([1, -1])

dot_products = input_table @ weights

T = 1

for i in range(len(input_table)):
    activation = linear_threshold_gate(dot_products[i], T)
    print(f'Input: {input_table[i]} → Activation: {activation}')
```

Output:

```
Input Table:

[[0 0]
 [0 1]
 [1 0]
 [1 1]]

Input: [0 0] → Activation: 0
Input: [0 1] → Activation: 0
Input: [1 0] → Activation: 1
Input: [1 1] → Activation: 0
```

Assignment 3

Roll No.:08

Code:

```
import numpy as np
class Perceptron:
    def __init__(self, input_size, lr=0.1):
        self.W = np.zeros(input_size + 1)  # Initialize weights including bias
        self.lr = lr  # Learning rate
    def activation_fn(self, x):
        return 1 if x >= 0 else 0
    def predict(self, x):
        x = np.insert(x, 0, 1)  # Insert bias term
        z = self.W.T.dot(x)
        return self.activation_fn(z)
    def train(self, X, Y, epochs):
            for _ in range(epochs):
          for i in range(Y.shape[0]):
            x = X[i]
            y_pred = self.predict(x)
            error = Y[i] - y_pred
            self.W = self.W + self.lr * error * np.insert(x, 0, 1)  # Weight update
X = np.array([ [0, 0, 0, 0, 0, 0, 1, 0, 0, 0],  # 0
    [0, 0, 0, 0, 0, 0, 0, 1, 0, 0],  # 1
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],  # 2
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 1],  # 3
    [0, 0, 0, 0, 0, 0, 1, 1, 0, 0],  # 4
    [0, 0, 0, 0, 0, 0, 1, 0, 1, 0],  # 5
    [0, 0, 0, 0, 0, 0, 1, 1, 1, 0],  # 6
    [0, 0, 0, 0, 0, 0, 1, 1, 1, 1],  # 7
```

```
    [0, 0, 0, 0, 0, 0, 1, 0, 1, 1],  # 8
    [0, 0, 0, 0, 0, 0, 0, 1, 1, 1]   # 9])
Y = np.array([0, 1, 0, 1, 0, 1, 0, 1, 0, 1])  # Even (0) or Odd (1)
perceptron = Perceptron(input_size=10)
perceptron.train(X, Y, epochs=100)
test_X = np.array([
    [0, 0, 0, 0, 0, 0, 1, 0, 0, 0],  # 0
    [0, 0, 0, 0, 0, 0, 0, 1, 0, 0],  # 1
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],  # 2
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 1],  # 3
    [0, 0, 0, 0, 0, 0, 1, 1, 0, 0],  # 4
    [0, 0, 0, 0, 0, 0, 1, 0, 1, 0],  # 5
    [0, 0, 0, 0, 0, 0, 1, 1, 1, 0],  # 6
    [0, 0, 0, 0, 0, 0, 1, 1, 1, 1],  # 7
    [0, 0, 0, 0, 0, 0, 1, 0, 1, 1],  # 8
    [0, 0, 0, 0, 0, 0, 0, 1, 1, 1]   # 9])
print("\nTesting Perceptron on Test Data:")
for i in range(test_X.shape[0]):
    x = test_X[i]
    y = perceptron.predict(x)
    print(f'{x} is {"even" if y == 0 else "odd"}')
```

Output:

```
Testing Perceptron on Test Data:

[0 0 0 0 0 0 1 0 0 0] is even
[0 0 0 0 0 0 0 1 0 0] is odd
[0 0 0 0 0 0 0 0 1 0] is even
[0 0 0 0 0 0 0 0 0 1] is odd
[0 0 0 0 0 0 1 1 0 0] is even
[0 0 0 0 0 0 1 0 1 0] is even
[0 0 0 0 0 0 1 1 1 0] is even
[0 0 0 0 0 0 1 1 1 1] is even
[0 0 0 0 0 0 1 0 1 1] is even
[0 0 0 0 0 0 0 1 1 1] is odd
```

Assignment 4

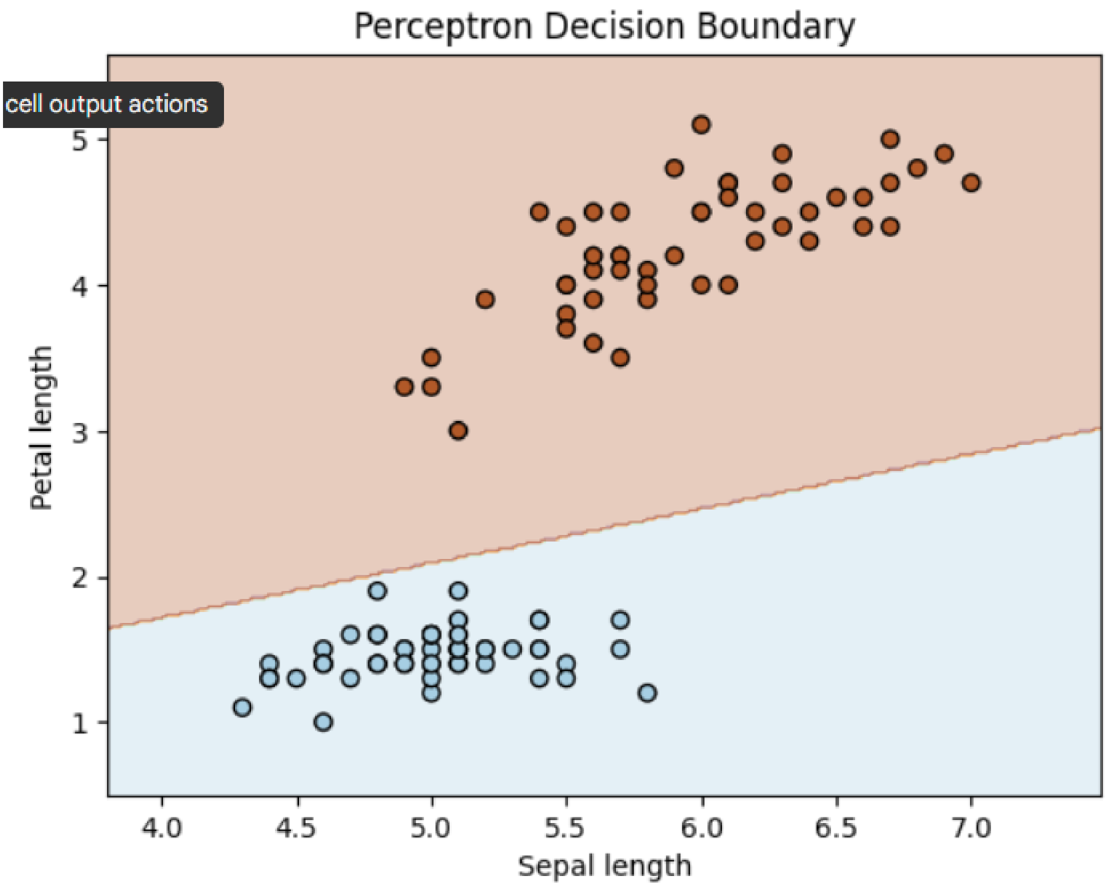
Roll No.:08

Code:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
iris = load_iris()
X = iris.data[:100, [0, 2]]  # Taking first 100 rows (Setosa & Versicolor)
y = iris.target[:100]        # Only first two classes
y = np.where(y == 0, 0, 1)
# Initialize weights and bias
w = np.zeros(2)
b = 0
# Set learning rate and number of epochs
lr = 0.1
epochs = 50
# Define perceptron function
def perceptron(x, w, b):
    z = np.dot(x, w) + b
    return np.where(z >= 0, 1, 0)  # Step activation function
# Train the perceptron
for epoch in range(epochs):
    for i in range(len(X)):
        x = X[i]
        target = y[i]
        output = perceptron(x, w, b)
        error = target - output
        w += lr * error * x  # Update weights
        b += lr * error  # Update bias
```

```
# Plot decision boundary
x_min, x_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5
y_min, y_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02),
                     np.arange(y_min, y_max, 0.02))
Z = perceptron(np.c_[xx.ravel(), yy.ravel()], w, b)
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, cmap=plt.cm.Paired, alpha=0.3)
# Plot data points
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Paired, edgecolors='k')
plt.xlabel('Sepal length')
plt.ylabel('Petal length')
plt.title('Perceptron Decision Boundary')
plt.show()
```

Output:



Assignment 5

Roll No.:08

Code:

```
import numpy as np

# Define pairs of vectors with matching dimensions

x1 = np.array([1, -1, -1])

y1 = np.array([1, 1, 1])

x2 = np.array([-1, 1, 1])  # Modified to match x1's size

y2 = np.array([-1, 1, 1])  # Modified to match y1's size

# Compute weight matrix W

W = np.outer(y1, x1) + np.outer(y2, x2)

# Define BAM function

def bam(x):

    y = np.dot(W, x)

    y = np.where(y >= 0, 1, -1)

    return y

# Test BAM with an input vector

x_test = np.array([1, -1, -1])

y_test = bam(x_test)

# Print output

print("Weight Matrix W:\n", W)

print("Input x:", x_test)

print("Output y:", y_test)
```

Output:

```
Weight Matrix W:

 [[ 2 -2 -2]
 [ 0  0  0]
 [ 0  0  0]]
Input x: [ 1 -1 -1]
Output y: [1 1 1]
```

Assignment 6

Roll No.:08

Code:

```
import numpy as np
class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size, learning_rate=0.1):
        self.W1 = np.random.randn(input_size, hidden_size)
        self.b1 = np.zeros((1, hidden_size))
        self.W2 = np.random.randn(hidden_size, output_size)
        self.b2 = np.zeros((1, output_size))
        self.learning_rate = learning_rate  # Store learning rate
    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))
    def sigmoid_derivative(self, x):
        return x * (1 - x)
    def forward_propagation(self, X):
        self.z1 = np.dot(X, self.W1) + self.b1
        self.a1 = self.sigmoid(self.z1)
        self.z2 = np.dot(self.a1, self.W2) + self.b2
        self.y_hat = self.sigmoid(self.z2)
        return self.y_hat
    def backward_propagation(self, X, y, y_hat):
        self.error = y - y_hat
        self.delta2 = self.error * self.sigmoid_derivative(y_hat)
        self.a1_error = self.delta2.dot(self.W2.T)
        self.delta1 = self.a1_error * self.sigmoid_derivative(self.a1)
        # Corrected weight and bias updates with learning rate
        self.W2 += self.learning_rate * self.a1.T.dot(self.delta2)
        self.b2 += self.learning_rate * np.sum(self.delta2, axis=0, keepdims=True)
```

```
        self.W1 += self.learning_rate * X.T.dot(self.delta1)
        self.b1 += self.learning_rate * np.sum(self.delta1, axis=0, keepdims=True)
    def train(self, X, y, epochs):
        for i in range(epochs):
            y_hat = self.forward_propagation(X)
            self.backward_propagation(X, y, y_hat)
            if i % 1000 == 0:
                print(f"Error at epoch {i}: {np.mean(np.abs(self.error))}")
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])
# Create a neural network with 2 input neurons, 4 hidden neurons, and 1 output neuron
nn = NeuralNetwork(2, 4, 1, learning_rate=0.1)
# Train the neural network for 10,000 epochs
nn.train(X, y, epochs=10000)
# Make predictions on the same dataset
predictions = nn.forward_propagation(X)
# Print predictions
print("\nPredictions:\n", predictions)
```

Output:

```
Error at epoch 0: 0.5143077645843104

Error at epoch 1000: 0.4653217076401508
Error at epoch 2000: 0.3312445504974636
Error at epoch 3000: 0.19017453426027864
Error at epoch 4000: 0.12705700754210805
Error at epoch 5000: 0.09764424873282275
Error at epoch 6000: 0.0808696063311011
Error at epoch 7000: 0.06994465097607569
Error at epoch 8000: 0.06219000560721724
Error at epoch 9000: 0.056350655231263855

Predictions:
 [[0.0372855 ]
 [0.94536041]
 [0.94409925]
 [0.05922075]]
```

Assignment 7

Roll No.:08

Code:

```
import numpy as np
class XORNetwork:
    def __init__(self):
        # Initialize the weights and biases randomly
        self.W1 = np.random.randn(2, 2)
        self.b1 = np.random.randn(2)
        self.W2 = np.random.randn(2, 1)
        self.b2 = np.random.randn(1)
    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))
    def sigmoid_derivative(self, x):
        return x * (1 - x)
    def forward(self, X):
        # Perform the forward pass
        self.z1 = np.dot(X, self.W1) + self.b1
        self.a1 = self.sigmoid(self.z1)
        self.z2 = np.dot(self.a1, self.W2) + self.b2
        self.a2 = self.sigmoid(self.z2)
        return self.a2
    def backward(self, X, y, output):
        # Perform the backward pass
        self.output_error = y - output
        self.output_delta = self.output_error * self.sigmoid_derivative(output)
        self.z1_error = self.output_delta.dot(self.W2.T)
        self.z1_delta = self.z1_error * self.sigmoid_derivative(self.a1)
        self.W1 += X.T.dot(self.z1_delta)
```

```
        self.b1 += np.sum(self.z1_delta, axis=0)
        self.W2 += self.a1.T.dot(self.output_delta)
        self.b2 += np.sum(self.output_delta, axis=0)
    def train(self, X, y, epochs):
        # Train the network for a given number of epochs
        for _ in range(epochs):
            output = self.forward(X)
            self.backward(X, y, output)
    def predict(self, X):
        # Make predictions for a given set of inputs
        return self.forward(X)
# Create a new XOR Network instance
xor_nn = XORNetwork()
# Define the input and output datasets for XOR
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])
# Train the network for 10,000 epochs
xor_nn.train(X, y, epochs=10000)
# Make predictions on the input dataset
predictions = xor_nn.predict(X)
# Print the predictions
print(predictions)
```

Output:

```
[[0.0127298 ]

 [0.98904949]
 [0.98890599]
 [0.01158486]]
```

Assignment 8

Roll No.:08

Code:

```
import numpy as np

# Define sigmoid activation function

def sigmoid(x):

    return 1 / (1 + np.exp(-x))

# Define derivative of sigmoid function

def sigmoid_derivative(x):

    return x * (1 - x)

# Define input dataset

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

# Define output dataset

y = np.array([[0], [1], [1], [0]])

# Define hyperparameters

learning_rate = 0.1

num_epochs = 100000

# Initialize weights randomly with mean 0

hidden_weights = 2 * np.random.random((2, 2)) - 1

output_weights = 2 * np.random.random((2, 1)) - 1

# Train the neural network

for _ in range(num_epochs):

    # Forward propagation

    hidden_layer = sigmoid(np.dot(X, hidden_weights))

    output_layer = sigmoid(np.dot(hidden_layer, output_weights))

    # Backpropagation

    output_error = y - output_layer

    output_delta = output_error * sigmoid_derivative(output_layer)

    hidden_error = output_delta.dot(output_weights.T)
```

```
    hidden_delta = hidden_error * sigmoid_derivative(hidden_layer)

    output_weights += hidden_layer.T.dot(output_delta) * learning_rate

    hidden_weights += X.T.dot(hidden_delta) * learning_rate

# Display input and output

print("Input:")

print(X)

print("Output:")

print(output_layer)
```

Output:

```
Input:

[[0 0]
 [0 1]
 [1 0]
 [1 1]]
Output:
[[0.0448246 ]
 [0.85735638]
 [0.85735614]
 [0.55239002]]
```