II. Implement the following:

(Implementation 5 marks and Visualization and documentation 5 marks) • Scenario: The XOR gate is known for its complexity, as it outputs 1 only when the inputs are different. This is a challenge for a Single Layer Perceptron since XOR is not linearly separable. • Lab Task: Attempt to implement a Single Layer Perceptron in Google Colab to classify the output of an XOR gate. Perform the following steps: • Create the XOR gate's truth table dataset. • Implement the perceptron model and train it using the XOR dataset using MCP (McCulloch Pitts) Neuron. • Observe and discuss the perceptron's performance in this scenario. • Implement XOR using Multi-Layer Perceptron.

```
import numpy as np

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([0, 1, 1, 0])  # target values

weights = np.random.rand(2)
bias = np.random.rand(1)
learning_rate = 0.1


def step_function(x):
    return np.where(x >= 0, 1, 0)


def perceptron_train(X, y, weights, bias, learning_rate, epochs=10):
    for epoch in range(epochs):
        for i in range(len(X)):
            net_input = np.dot(X[i], weights) + bias
            output = step_function(net_input)
            error = y[i] - output
            weights += learning_rate * error * X[i]
            bias += learning_rate * error
    return weights, bias


weights, bias = perceptron_train(X, y, weights, bias, learning_rate)


def perceptron_predict(X, weights, bias):
    net_input = np.dot(X, weights) + bias
    return step_function(net_input)

predictions = perceptron_predict(X, weights, bias)
print("Predictions after training:", predictions)
```

```
Predictions after training: [1 1 0 0]
```

Discuss the perceptron's performance

The perceptron is not be able to correctly classify all XOR inputs because XOR is not linearly separable.

```
from sklearn.neural_network import MLPClassifier

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

mlp = MLPClassifier(hidden_layer_sizes=(2,), activation='relu', solver='adam', max_iter=10000)
mlp.fit(X, y)

predictions_mlp = mlp.predict(X)
print("MLP Predictions:", predictions_mlp)
```

```
MLP Predictions: [0 1 1 0]
```

Multi-Layer Perceptron adding a hidden layer, the MLP is capable of classifying XOR correctly,because it can understand the hidden layer ,which help to learn all the layers and get the output correctly.

A. Sentiment Analysis Twitter Airline

Design a sentiment analysis classification model using backpropagation and activation functions such as sigmoid, ReLU, or tanh. Implement a neural network that can classify sentiment (positive/negative) from a small dataset. Demonstrate how backpropagation updates the weights during the training process. (link Provided at the top of the page to download the dataset) Task: • Create a simple feed-forward neural network for binary sentiment classification (positive/negative). • Use backpropagation to optimize the model's weights based on error calculation. • Experiment with different activation functions (sigmoid, ReLU, tanh) in the hidden layer and compare the model's performance. • Evaluate the model on a test set using accuracy and plot the loss over epochs.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer

dataset = '/content/Tweets - Tweets.csv'
tweets_df = pd.read_csv(dataset)


filtered_df = tweets_df[['airline_sentiment', 'text']].copy()

filtered_df = filtered_df[filtered_df['airline_sentiment'].isin(['positive', 'negative'])]

filtered_df['label'] = filtered_df['airline_sentiment'].map({'positive': 1, 'negative': 0})

filtered_df = filtered_df[['text', 'label']].dropna()

vectorizer = TfidfVectorizer(stop_words='english', max_features=1000)

X = vectorizer.fit_transform(filtered_df['text'])

y = filtered_df['label'].values

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

print("Training Data Shape:", X_train.shape)
print("Testing Data Shape:", X_test.shape)
```

```
Training Data Shape: (9232, 1000)
Testing Data Shape: (2309, 1000)
```

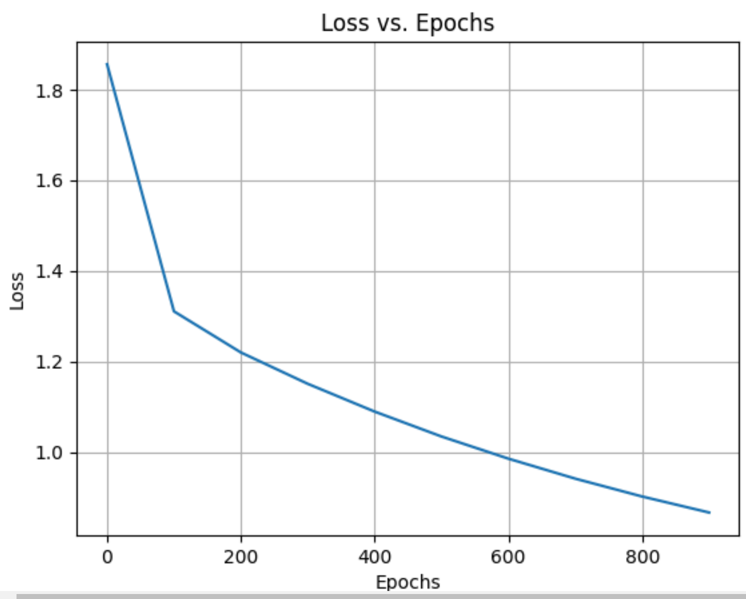
```
# Activation functions

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 sigmoid_derivative(x):
    return sigmoid(x) * (1 - sigmoid(x))

def relu_derivative(x):
    return np.where(x > 0, 1, 0)

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

def initialize_weights(input_size, hidden_size, output_size):
    W1 = np.random.randn(input_size, hidden_size)
    b1 = np.zeros((1, hidden_size))
    W2 = np.random.randn(hidden_size, output_size)
    b2 = np.zeros((1, output_size))
    return W1, b1, W2, b2


def forward_propagation(X, W1, b1, W2, b2, activation_func):
    Z1 = np.dot(X, W1) + b1
    A1 = activation_func(Z1)
    Z2 = np.dot(A1, W2) + b2
    A2 = sigmoid(Z2)
    return Z1, A1, Z2, A2

def backpropagation(X, y, Z1, A1, A2, W2, activation_derivative):
    m = X.shape[0]
    dZ2 = A2 - y
    dW2 = np.dot(A1.T, dZ2) / m
    db2 = np.sum(dZ2, axis=0, keepdims=True) / m

    dA1 = np.dot(dZ2, W2.T)
    dZ1 = dA1 * activation_derivative(Z1)
    dW1 = np.dot(X.T, dZ1) / m
    db1 = np.sum(dZ1, axis=0, keepdims=True) / m

    return dW1, db1, dW2, db2


def update_weights(W1, b1, W2, b2, dW1, db1, dW2, db2, learning_rate):
    W1 -= learning_rate * dW1
    b1 -= learning_rate * db1
    W2 -= learning_rate * dW2
    b2 -= learning_rate * db2
    return W1, b1, W2, b2

def train_neural_network(X_train, y_train, hidden_size, epochs, learning_rate, activation_func, activation_derivative):
    input_size = X_train.shape[1]
    output_size = 1
    W1, b1, W2, b2 = initialize_weights(input_size, hidden_size, output_size)

    for epoch in range(epochs):

        Z1, A1, Z2, A2 = forward_propagation(X_train, W1, b1, W2, b2, activation_func)


        loss = -np.mean(y_train * np.log(A2) + (1 - y_train) * np.log(1 - A2))


        dW1, db1, dW2, db2 = backpropagation(X_train, y_train, Z1, A1, A2, W2, activation_derivative)


        W1, b1, W2, b2 = update_weights(W1, b1, W2, b2, dW1, db1, dW2, db2, learning_rate)

        if epoch % 100 == 0:
            print(f'Epoch {epoch}, Loss: {loss:.4f}')

    return W1, b1, W2, b2


def predict(X, W1, b1, W2, b2, activation_func):
    _, _, _, A2 = forward_propagation(X, W1, b1, W2, b2, activation_func)
    return (A2 > 0.5).astype(int)



hidden_size = 64
epochs = 1000
learning_rate = 0.01


activation_func = relu
activation_derivative = relu_derivative

W1, b1, W2, b2 = train_neural_network(X_train.toarray(), y_train.reshape(-1, 1), hidden_size, epochs, learning_rate, activation_func, a

predictions = predict(X_test.toarray(), W1, b1, W2, b2, activation_func)

accuracy = np.mean(predictions == y_test.reshape(-1, 1))
print(f"Test Accuracy: {accuracy:.4f}")
```

```
Epoch 0, Loss: 1.8556
Epoch 100, Loss: 1.3104
Epoch 200, Loss: 1.2197
Epoch 300, Loss: 1.1505
Epoch 400, Loss: 1.0892
Epoch 500, Loss: 1.0342
Epoch 600, Loss: 0.9853
Epoch 700, Loss: 0.9412
Epoch 800, Loss: 0.9018
Epoch 900, Loss: 0.8665
Test Accuracy: 0.7181
```

Backprogation

```
import matplotlib.pyplot as plt
epochs = [0, 100, 200, 300, 400, 500, 600, 700, 800, 900]
losses = [1.8556, 1.3104, 1.2197, 1.1505, 1.0892, 1.0342, 0.9853, 0.9412, 0.9018, 0.8665]
```

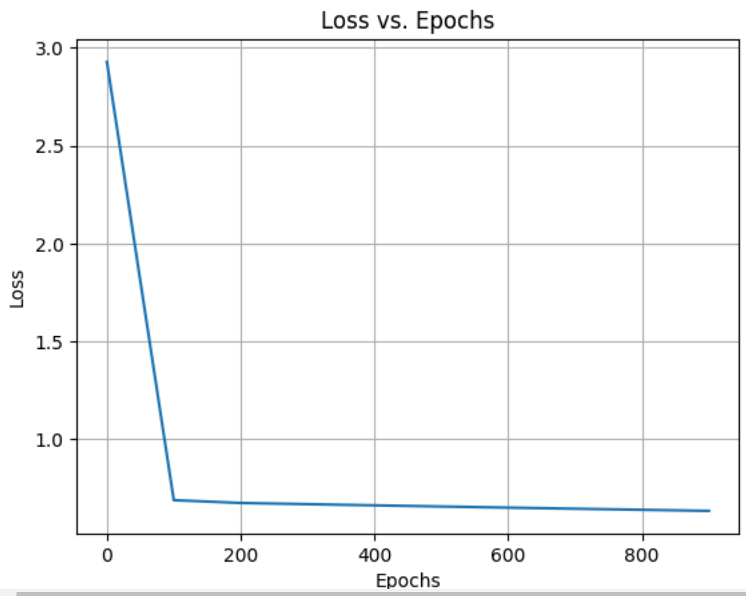
```
plt.plot(epochs, losses)
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Loss vs. Epochs')
plt.grid(True)
plt.show()
```



```
hidden_size = 64
epochs = 1000
learning_rate = 0.01


activation_func = sigmoid
activation_derivative = sigmoid_derivative

W1, b1, W2, b2 = train_neural_network(X_train.toarray(), y_train.reshape(-1, 1), hidden_size, epochs, learning_rate, activation_func, a

predictions = predict(X_test.toarray(), W1, b1, W2, b2, activation_func)

accuracy = np.mean(predictions == y_test.reshape(-1, 1))
print(f"Test Accuracy: {accuracy:.4f}")
```

```
Epoch 0, Loss: 2.9272
Epoch 100, Loss: 0.6902
Epoch 200, Loss: 0.6759
Epoch 300, Loss: 0.6697
Epoch 400, Loss: 0.6637
Epoch 500, Loss: 0.6579
Epoch 600, Loss: 0.6521
Epoch 700, Loss: 0.6465
Epoch 800, Loss: 0.6410
Epoch 900, Loss: 0.6356
Test Accuracy: 0.7631
```

```
epochs = [0, 100, 200, 300, 400, 500, 600, 700, 800, 900]
losses = [2.9272, 0.6902, 0.6759, 0.6697, 0.6637, 0.6579, 0.6521, 0.6465, 0.6410, 0.6356]

plt.plot(epochs, losses)
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Loss vs. Epochs')
plt.grid(True)
plt.show()
```



using relu activation function the accuracy is better sigmoid beacuse it helps in vanishing gradient problem.where as in sigmoid it trends to vanish the gradient.