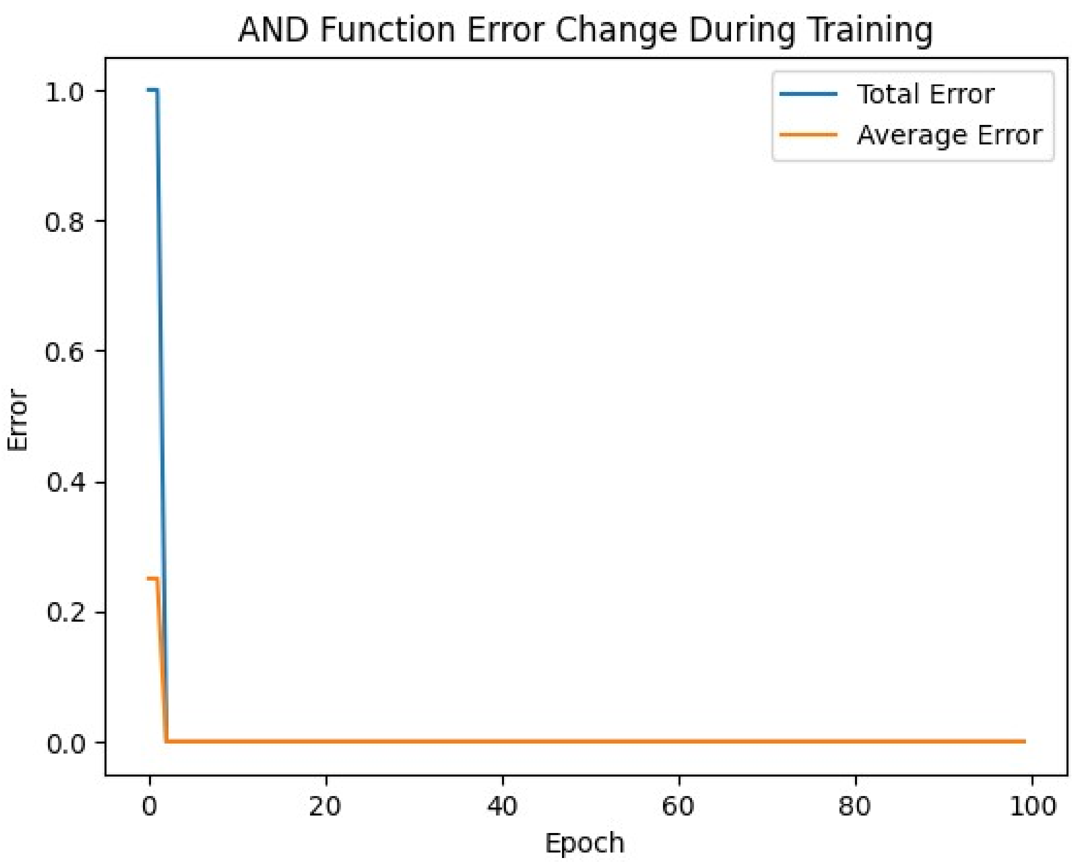
```python
#libraries
import numpy as np
import matplotlib.pyplot as plt
```

Q 1)

```python
# AND
def perceptron(x1, x2, w1, w2, w0):
    z = w1 * x1 + w2 * x2 + w0 #EQUATION
    return 1 if z >= 0 else 0

def perceptron_learning_algorithm(x1, x2, target, w1, w2, w0,
learning_rate=0.1):
    output = perceptron(x1, x2, w1, w2, w0)
    error = target - output
    w1 += learning_rate * error * x1
    w2 += learning_rate * error * x2
    w0 += learning_rate * error
    return w1, w2, w0

def AND():
    w1 = 0.7
    w2 = 0.243
    w0 = 0.057
    learning_rate = 0.1
    epochs = 100
    epoch_errors = []
    average_errors = []

    print("\n----- Before Training -----")
    for x1 in [0, 1]:
        for x2 in [0, 1]:
            target = x1 and x2
            output = perceptron(x1, x2, w1, w2, w0)
            error = abs(target - output)
            print(f"AND(x1={x1}, x2={x2}): Target={target},
Output={output}, Error={error}")

    for epoch in range(epochs):
        total_error = 0
        for x1 in [0, 1]:
            for x2 in [0, 1]:
                target = x1 and x2
                w1, w2, w0 = perceptron_learning_algorithm(x1, x2,
target, w1, w2, w0, learning_rate)
                output = perceptron(x1, x2, w1, w2, w0)
                error = abs(target - output)
                total_error += error
```

```python
        average_error = total_error / 4
        epoch_errors.append(total_error)
        average_errors.append(average_error)

    print("\n----- AND after Training -----")
    for x1 in [0, 1]:
        for x2 in [0, 1]:
            target = x1 and x2
            output = perceptron(x1, x2, w1, w2, w0)
            error = abs(target - output)
            print(f"AND(x1={x1}, x2={x2}): Target={target},
Output={output}, Error={error}")

    plt.plot(epoch_errors, label="Total Error")
    plt.plot(average_errors, label="Average Error")
    plt.xlabel("Epoch")
    plt.ylabel("Error")
    plt.title("AND Function Error Change During Training")
    plt.legend()
    plt.show()

if __name__ == "__main__":
    AND()
```

```
----- Before Training -----
AND(x1=0, x2=0): Target=0, Output=1, Error=1
AND(x1=0, x2=1): Target=0, Output=1, Error=1
AND(x1=1, x2=0): Target=0, Output=1, Error=1
AND(x1=1, x2=1): Target=1, Output=1, Error=0

----- AND after Training -----
AND(x1=0, x2=0): Target=0, Output=0, Error=0
AND(x1=0, x2=1): Target=0, Output=0, Error=0
AND(x1=1, x2=0): Target=0, Output=0, Error=0
AND(x1=1, x2=1): Target=1, Output=1, Error=0
```



Before Training: The perceptron initially produced outputs of 1 for all inputs, resulting in errors for all cases. This indicates that the initial weights were not set properly to correctly classify the AND function.

After Training: After applying the perceptron learning algorithm, the perceptron adjusted its weights to minimize errors. As a result, it successfully learned the AND function, producing the correct output (0 for false and 1 for true) for all input combinations.

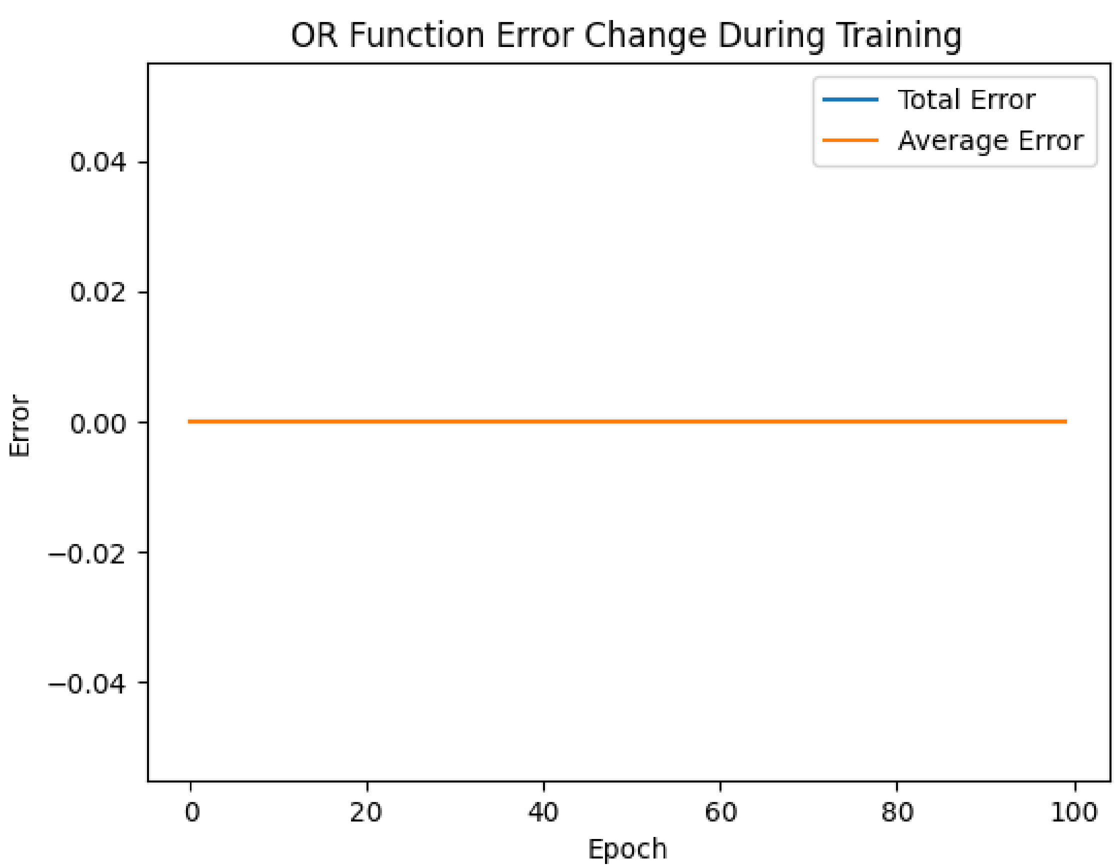
Inference: The perceptron has successfully learned to replicate the AND function behavior, correctly outputting 0 only when both inputs are 0, and 1 otherwise. This demonstrates the capability of the perceptron learning algorithm to learn simple boolean functions.

```
# OR
def perceptron(x1, x2, w1, w2, w0):
    z = w1 * x1 + w2 * x2 + w0
    return 1 if z >= 0 else 0

def perceptron_learning_algorithm(x1, x2, target, w1, w2, w0,
learning_rate=0.1):
    output = perceptron(x1, x2, w1, w2, w0)
    error = target - output
    w1 += learning_rate * error * x1
    w2 += learning_rate * error * x2
```

```
    w0 += learning_rate * error
    return w1, w2, w0

def OR():
    w1 = 0.7
    w2 = 0.243
    w0 = 0.057
    learning_rate = 0.1
    epochs = 100
    epoch_errors = []
    average_errors = []

    print("\n----- Before Training -----")
    for x1 in [0, 1]:
        for x2 in [0, 1]:
            target = x1 or x2
            output = perceptron(x1, x2, w1, w2, w0)
            error = abs(target - output)
            print(f"OR(x1={x1}, x2={x2}): Target={target},
Output={output}, Error={error}")

    for epoch in range(epochs):
        total_error = 0
        for x1 in [0, 1]:
            for x2 in [0, 1]:
                target = x1 or x2
                w1, w2, w0 = perceptron_learning_algorithm(x1, x2,
target, w1, w2, w0, learning_rate)
                output = perceptron(x1, x2, w1, w2, w0)
                error = abs(target - output)
                total_error += error

        average_error = total_error / 4
        epoch_errors.append(total_error)
        average_errors.append(average_error)

    print("\n----- OR after Training -----")
    for x1 in [0, 1]:
        for x2 in [0, 1]:
            target = x1 or x2
            output = perceptron(x1, x2, w1, w2, w0)
            error = abs(target - output)
            print(f"OR(x1={x1}, x2={x2}): Target={target},
Output={output}, Error={error}")

    plt.plot(epoch_errors, label="Total Error")
    plt.plot(average_errors, label="Average Error")
    plt.xlabel("Epoch")
    plt.ylabel("Error")
    plt.title("OR Function Error Change During Training")
```

```
    plt.legend()
    plt.show()

if __name__ == "__main__":
    OR()
```

```
----- Before Training -----
OR(x1=0, x2=0): Target=0, Output=1, Error=1
OR(x1=0, x2=1): Target=1, Output=1, Error=0
OR(x1=1, x2=0): Target=1, Output=1, Error=0
OR(x1=1, x2=1): Target=1, Output=1, Error=0

----- OR after Training -----
OR(x1=0, x2=0): Target=0, Output=0, Error=0
OR(x1=0, x2=1): Target=1, Output=1, Error=0
OR(x1=1, x2=0): Target=1, Output=1, Error=0
OR(x1=1, x2=1): Target=1, Output=1, Error=0
```



Before Training: The perceptron initially produced an output of 1 for the input combination (0, 0), resulting in an error. However, for the other input combinations, it correctly produced outputs of 1, resulting in zero errors.

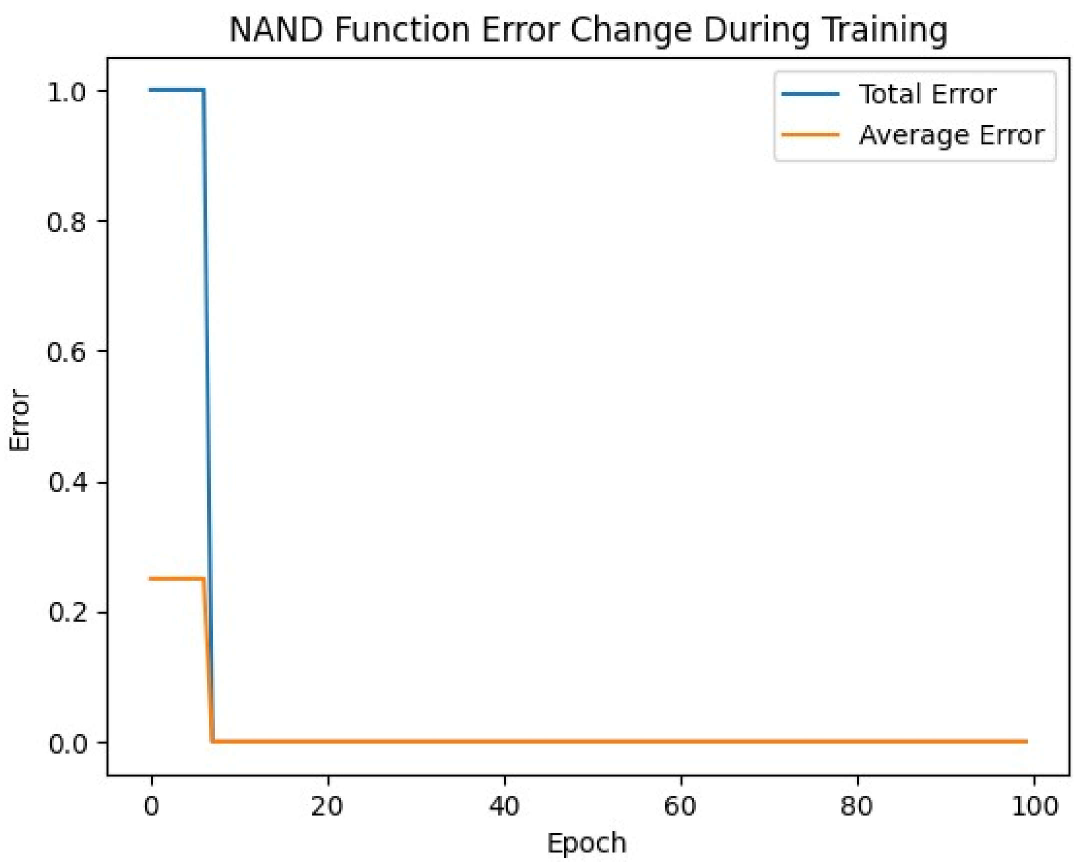
After Training: After applying the perceptron learning algorithm, the perceptron adjusted its weights to minimize errors. As a result, it successfully learned the OR function, producing the correct output (0 for false and 1 for true) for all input combinations.

Inference: The perceptron has successfully learned to replicate the OR function behavior, correctly outputting 1 if at least one of the inputs is 1.

```
#NAND
def perceptron(x1, x2, w1, w2, w0):
    z = w1 * x1 + w2 * x2 + w0
    return 1 if z >= 0 else 0

def perceptron_learning_algorithm(x1, x2, target, w1, w2, w0,
learning_rate=0.1):
    output = perceptron(x1, x2, w1, w2, w0)
    error = target - output
    w1 += learning_rate * error * x1
    w2 += learning_rate * error * x2
    w0 += learning_rate * error
    return w1, w2, w0

def NAND():
    w1 = 0.7
    w2 = 0.243
    w0 = 0.057
    learning_rate = 0.1
    epochs = 100
    epoch_errors = []
    average_errors = []

    print("\n----- Before Training -----")
    for x1 in [0, 1]:
        for x2 in [0, 1]:
            target = not (x1 and x2)  # NAND operation
            output = perceptron(x1, x2, w1, w2, w0)
            error = abs(target - output)
            print(f"NAND(x1={x1}, x2={x2}): Target={target},
Output={output}, Error={error}")

    for epoch in range(epochs):
        total_error = 0
        for x1 in [0, 1]:
            for x2 in [0, 1]:
                target = not (x1 and x2)  # NAND operation
                w1, w2, w0 = perceptron_learning_algorithm(x1, x2,
target, w1, w2, w0, learning_rate)
                output = perceptron(x1, x2, w1, w2, w0)
                error = abs(target - output)
                total_error += error
```

```python
        average_error = total_error / 4
        epoch_errors.append(total_error)
        average_errors.append(average_error)

    print("\n----- NAND after Training -----")
    for x1 in [0, 1]:
        for x2 in [0, 1]:
            target = not (x1 and x2)  # NAND operation
            output = perceptron(x1, x2, w1, w2, w0)
            error = abs(target - output)
            print(f"NAND(x1={x1}, x2={x2}): Target={target}, 
Output={output}, Error={error}")

    plt.plot(epoch_errors, label="Total Error")
    plt.plot(average_errors, label="Average Error")
    plt.xlabel("Epoch")
    plt.ylabel("Error")
    plt.title("NAND Function Error Change During Training")
    plt.legend()
    plt.show()

if __name__ == "__main__":
    NAND()
```

```
----- Before Training -----
NAND(x1=0, x2=0): Target=True, Output=1, Error=0
NAND(x1=0, x2=1): Target=True, Output=1, Error=0
NAND(x1=1, x2=0): Target=True, Output=1, Error=0
NAND(x1=1, x2=1): Target=False, Output=1, Error=1

----- NAND after Training -----
NAND(x1=0, x2=0): Target=True, Output=1, Error=0
NAND(x1=0, x2=1): Target=True, Output=1, Error=0
NAND(x1=1, x2=0): Target=True, Output=1, Error=0
NAND(x1=1, x2=1): Target=False, Output=0, Error=0
```



Before Training: The perceptron initially produced an output of 1 for the input combination (1, 1), resulting in an error. However, for the other input combinations, it correctly produced outputs of 1, resulting in zero errors.

After Training: After applying the perceptron learning algorithm, the perceptron adjusted its weights to minimize errors. As a result, it successfully learned the NAND function, producing the correct output (True for false and False for true) for all input combinations.

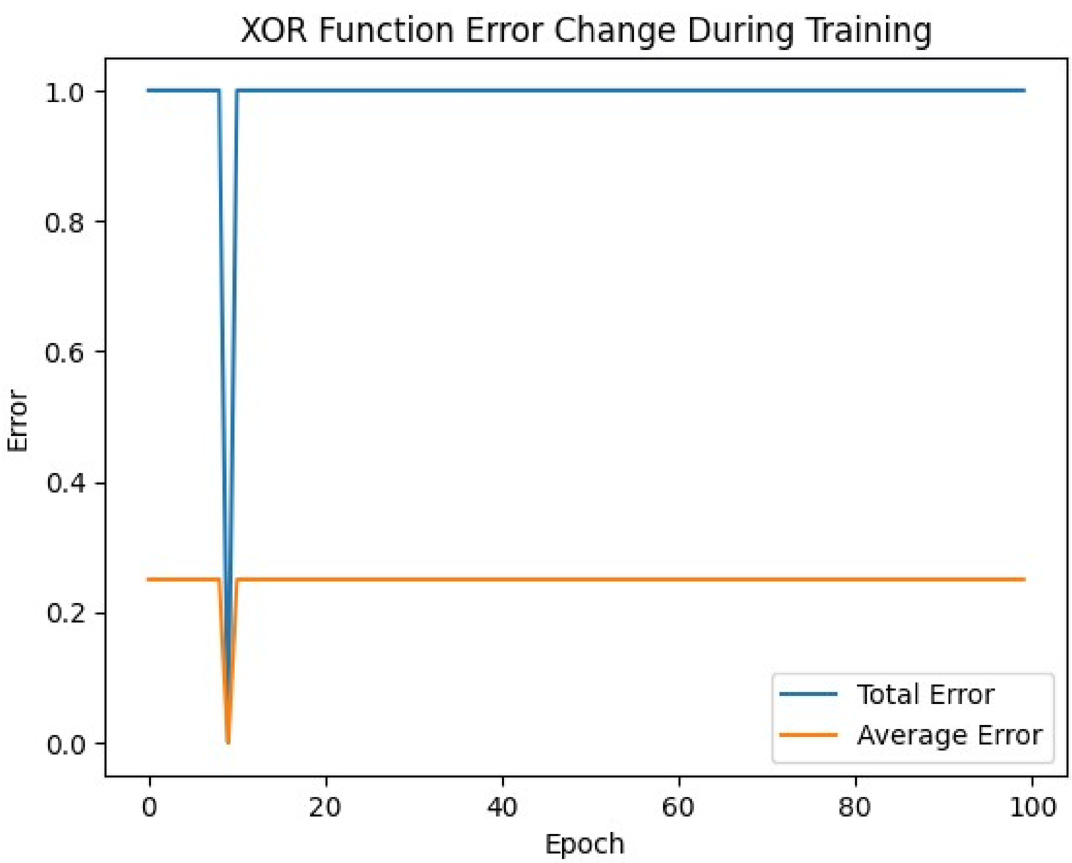
Inference: The perceptron has successfully learned to replicate the NAND function behavior, correctly outputting True if both inputs are not true and False otherwise.

```
#XOR
def perceptron(x1, x2, w1, w2, w0):
    z = w1 * x1 + w2 * x2 + w0
    return 1 if z >= 0 else 0

def perceptron_learning_algorithm(x1, x2, target, w1, w2, w0,
learning_rate=0.1):
    output = perceptron(x1, x2, w1, w2, w0)
    error = target - output
    w1 += learning_rate * error * x1
    w2 += learning_rate * error * x2
    w0 += learning_rate * error
```

```python
    return w1, w2, w0

def XOR():
    w1 = 0.7
    w2 = 0.243
    w0 = 0.057
    learning_rate = 0.1
    epochs = 100
    epoch_errors = []
    average_errors = []

    print("\n----- Before Training -----")
    for x1 in [0, 1]:
        for x2 in [0, 1]:
            target = x1 ^ x2  # XOR operation
            output = perceptron(x1, x2, w1, w2, w0)
            error = abs(target - output)
            print(f"XOR(x1={x1}, x2={x2}): Target={target},
Output={output}, Error={error}")

    for epoch in range(epochs):
        total_error = 0
        for x1 in [0, 1]:
            for x2 in [0, 1]:
                target = x1 ^ x2  # XOR operation
                w1, w2, w0 = perceptron_learning_algorithm(x1, x2,
target, w1, w2, w0, learning_rate)
                output = perceptron(x1, x2, w1, w2, w0)
                error = abs(target - output)
                total_error += error

        average_error = total_error / 4
        epoch_errors.append(total_error)
        average_errors.append(average_error)

    print("\n----- XOR after Training -----")
    for x1 in [0, 1]:
        for x2 in [0, 1]:
            target = x1 ^ x2  # XOR operation
            output = perceptron(x1, x2, w1, w2, w0)
            error = abs(target - output)
            print(f"XOR(x1={x1}, x2={x2}): Target={target},
Output={output}, Error={error}")

    plt.plot(epoch_errors, label="Total Error")
    plt.plot(average_errors, label="Average Error")
    plt.xlabel("Epoch")
    plt.ylabel("Error")
    plt.title("XOR Function Error Change During Training")
    plt.legend()
```

```
    plt.show()

if __name__ == "__main__":
    XOR()
```

```
----- Before Training -----
XOR(x1=0, x2=0): Target=0, Output=1, Error=1
XOR(x1=0, x2=1): Target=1, Output=1, Error=0
XOR(x1=1, x2=0): Target=1, Output=1, Error=0
XOR(x1=1, x2=1): Target=0, Output=1, Error=1

----- XOR after Training -----
XOR(x1=0, x2=0): Target=0, Output=1, Error=1
XOR(x1=0, x2=1): Target=1, Output=1, Error=0
XOR(x1=1, x2=0): Target=1, Output=0, Error=1
XOR(x1=1, x2=1): Target=0, Output=1, Error=1
```



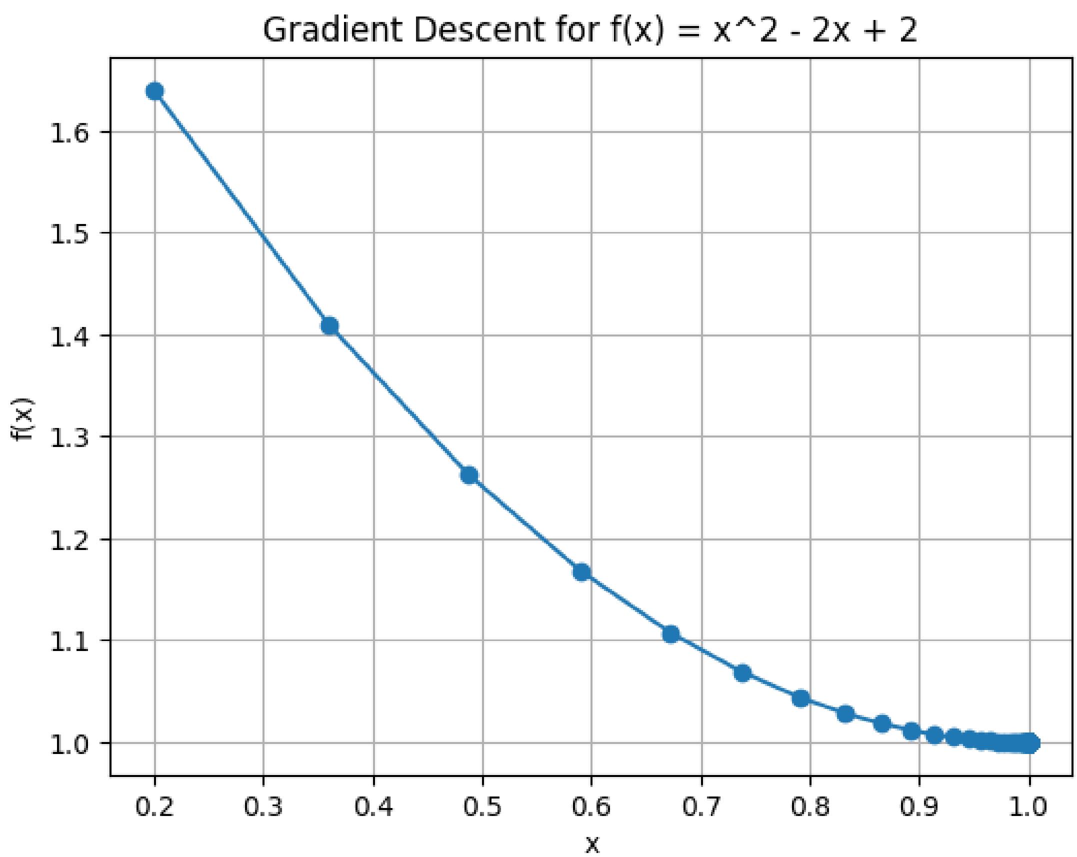
Before Training: The perceptron initially produced incorrect outputs for the input combinations (0, 0) and (1, 1), resulting in errors. However, for the other input combinations, it correctly produced outputs, resulting in zero errors.

After Training: After applying the perceptron learning algorithm, the perceptron adjusted its weights, but it did not converge to a solution that correctly represents the XOR function. As a result, it still produced incorrect outputs for the input combinations (0, 0) and (1, 1).

Inference: The perceptron failed to learn the XOR function because it is not linearly separable. XOR function outputs true only when the inputs are different. Perceptrons can only learn linearly separable functions, and XOR is a non-linearly separable function.

Q 2)

```
def f(x):
    return x**2 - 2*x + 2

def gradient_descent(learning_rate, initial_x, epochs):
    x = initial_x
    iteration = 0
    x_values = []
    y_values = []

    while iteration < epochs:
        gradient = 2*x - 2  # Derivative of f(x)
        x -= learning_rate * gradient
        iteration += 1
        x_values.append(x)
        y_values.append(f(x))

    plt.plot(x_values, y_values, marker='o', linestyle='-')
    plt.title("Gradient Descent for f(x) = x^2 - 2x + 2")
    plt.xlabel("x")
    plt.ylabel("f(x)")
    plt.grid(True)
    plt.show()

    return x, iteration

if __name__ == "__main__":
    learning_rate = 0.1
    initial_x = 0  # Initial value
    epochs = 1000

    minimum, iterations = gradient_descent(learning_rate, initial_x, 
epochs)
    print(f"Global minimum is at x = {minimum}, found after 
{iterations} iterations.")
```

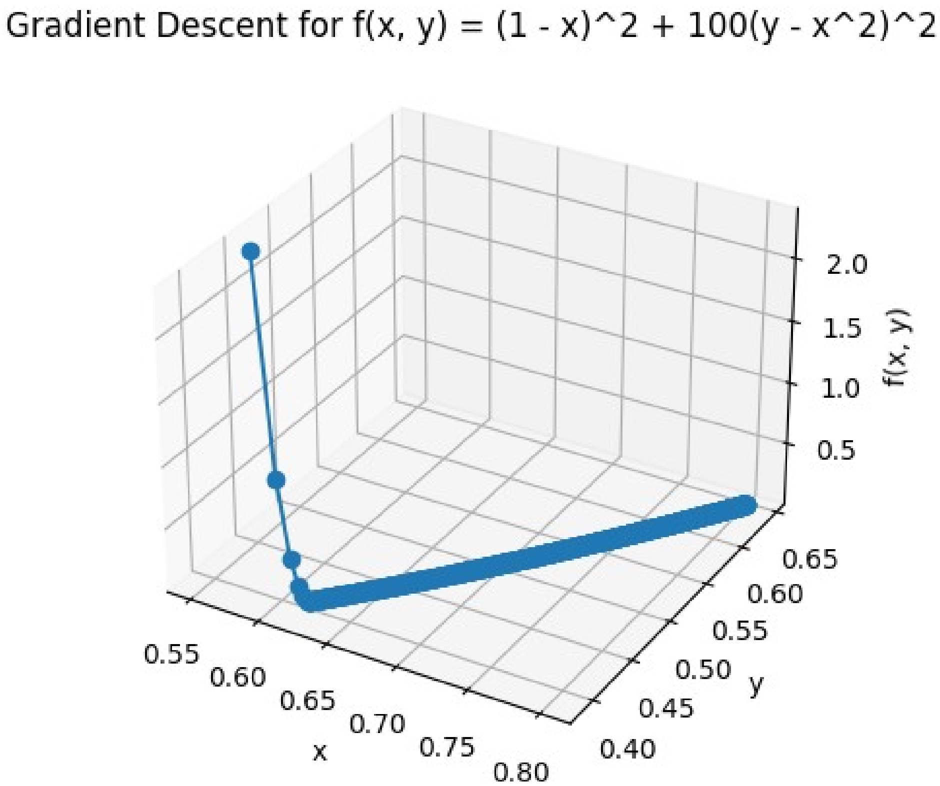



```
Global minimum is at x = 0.9999999999999998, found after 1000
iterations.
```

The gradient descent algorithm effectively minimized the function by iteratively updating the value of x in the direction of the negative gradient. As a result, it reached the global minimum of the function. This demonstrates the usefulness of gradient descent in optimizing functions and finding their minima.

```
from mpl_toolkits.mplot3d import Axes3D

def f(x, y):
    return (1 - x)**2 + 100 * (y - x**2)**2

def gradient_descent(learning_rate, initial_x, initial_y, epochs):
    x = initial_x
    y = initial_y
    iteration = 0
    x_values = []
    y_values = []
    z_values = []

    while iteration < epochs:
        gradient_x = -2 * (1 - x) - 400 * x * (y - x**2)
```

```
        gradient_y = 200 * (y - x**2)
        x -= learning_rate * gradient_x
        y -= learning_rate * gradient_y
        iteration += 1
        x_values.append(x)
        y_values.append(y)
        z_values.append(f(x, y))

    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.plot(x_values, y_values, z_values, marker='o', linestyle='-')
    ax.set_title("Gradient Descent for f(x, y) = (1 - x)^2 + 100(y - x^2)^2")
    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.set_zlabel("f(x, y)")
    plt.show()

    return x, y, iteration

if __name__ == "__main__":
    learning_rate = 0.001
    initial_x = 0.5  # Initial value for x
    initial_y = 0.5  # Initial value for y
    epochs = 1000

    minimum_x, minimum_y, iterations = gradient_descent(learning_rate, initial_x, initial_y, epochs)
    print(f"Global minimum is at (x, y) = ({minimum_x}, {minimum_y}), found after {iterations} iterations.")
```




```
Global minimum is at (x, y) = (0.8022816735722534,
0.6427675868954602), found after 1000 iterations.
```

The gradient descent algorithm successfully minimized the function, reaching a point close to the global minimum. This result demonstrates the effectiveness of gradient descent in finding optimal solutions for multi-dimensional functions.