QUESTION:

Perform the feed-forward pass, prediction of output, calculation of loss function, and weight-bias adjustments with the Multi-layer Perceptron with input, output, and initial weight & bias vectors which are also given below. The network must go through one single epoch of feed-forward pass and backpropagation to make predictions and adjust the weights & biases by following the gradient descent optimization strategy that reduces the loss function (mean square error). You are required to report the value of the loss function and the adjusted values of weight & bias vectors after the first epoch. The learning rate is 0.1 and the activation function is sigmoid.

Weight Vector $\mathbf{W}_{ij}^{1}$ for $1 \leq i \leq 2$ and $1 \leq j \leq 3$

i/j	**1**	**2**	**3**
1	-0.2	-1.4	0.6
2	-0.1	0.9	1.2

Bias Vector $\mathbf{B}_{i}^{1}$ for $1 \leq i \leq 3$

i		
1	2	3
-1.4	-0.8	0.2

Weight Vector $\mathbf{W}_{ij}^{2}$ for $1 \leq i \leq 3$ and $j=1$

j/i	**1**	**2**	**3**
1	0.2	-1.1	0.5

Bias $\mathbf{B}_{1}^{2}$

$\mathbf{B}_{1}^{2} = -0.6$

Dataset

x1	**x2**	**y**
-0.3	-1.5	0.1
-1.7	0.7	-0.3

PYTHON CODE:

```
# Import numpy library
import numpy as np

# Define the sigmoid activation function using NumPy’s built-in exp() function
def sigmoid(x):
    """
    Calculate the sigmoid function given an input x.
    Args:
        x: Input value(s) to feed into the sigmoid function.
    Returns:
        Result of the sigmoid function applied to the input x.
    """
    return 1 / (1 + np.exp(-x))

# Define the derivative of the sigmoid activation function
def sigmoid_derivative(x):
    """
    Calculate the derivative of the sigmoid function given an input x.
    Args:
        x: Input value(s) to feed into the derivative of the sigmoid function.
    Returns:
        Result of the derivative of the sigmoid function applied to the input
x.
    """
    return sigmoid(x) * (1 - sigmoid(x))

# Define the learning rate
learning_rate = 0.1

# Define the input data
X = np.array([[-0.3, -1.5], [-1.7, 0.7]])

# Define the output data
Y = np.array([[0.1], [-0.3]])

# Define the weight and bias vectors for each layer of the neural network
using a dictionary
params = {
    'W1': np.array([[-0.2, -1.4, 0.6], [-0.1, 0.9, 1.2]]),
    'B1': np.array([[-1.4], [-0.8], [0.2]]),
    'W2': np.array([[0.2], [-1.1], [0.5]]),
    'B2': np.array([[-0.6]])
}

# Define batch size for batch processing
batch_size = 1
```

```
# Perform the feed-forward pass to get predictions
Z1 = np.dot(X, params['W1']) + params['B1'].T
A1 = sigmoid(Z1)
Z2 = np.dot(A1, params['W2']) + params['B2']
Y_hat = sigmoid(Z2)

# Calculate the mean squared error loss function
error = Y - Y_hat
mse_loss = np.mean(error ** 2)

# Perform backpropagation and weight-bias adjustments to optimize the neural
network using batch processing
num_samples = X.shape[0]
for i in range(0, num_samples, batch_size):

    # Compute gradients
    dZ2 = error[i:i+batch_size] * sigmoid_derivative(Z2[i:i+batch_size])
    dW2 = np.dot(A1[i:i+batch_size].T, dZ2)
    dB2 = np.sum(dZ2, axis=0, keepdims=True)
    dZ1 = np.dot(dZ2, params['W2'].T) * sigmoid_derivative(Z1[i:i+batch_size])
    dW1 = np.dot(X[i:i+batch_size].T, dZ1)
    dB1 = np.sum(dZ1, axis=0, keepdims=True)

    # Update weights and biases
    params['W2'] -= learning_rate * dW2
    params['B2'] -= learning_rate * dB2
    params['W1'] -= learning_rate * dW1
    params['B1'] -= learning_rate * dB1.T

# Report the value of the loss function and the adjusted values of weight and
bias vectors
print("\nMean Square Error Loss: ", mse_loss)
print("\nAdjusted W1: \n", params['W1'])
print("\nAdjusted B1: \n", params['B1'])
print("\nAdjusted W2: \n", params['W2'])
print("\nAdjusted B2: \n", params['B2'])
```

CODE EXPLANATION:

The code is implementing a simple neural network using the sigmoid activation function and batch processing to optimize its parameters. Here's a detailed explanation of each step:

- The numpy library is imported to use its functionality for numerical calculations.
- The `sigmoid` function is defined as taking an input `x` and returning its output after applying the sigmoid transformation. The sigmoid function maps any real value to a value between 0 and 1, making it useful in modelling binary classification problems.

- The `sigmoid_derivative` function is defined as taking the output from the `sigmoid` function as `x` and returning its derivative. This function is used during the backpropagation phase of training the neural network.
- A learning rate of 0.1 is set as a hyperparameter to control how much the weights and biases are adjusted during training.
- The input data (`X`) and target output data (`Y`) are defined as NumPy arrays.
- The weight (`W1`, `W2`) and bias (`B1`, `B2`) vectors for each layer of the neural network are defined as dictionaries containing numpy arrays.
- A batch size of 1 is defined for batch processing to train the model faster.
- The forward propagation phase begins by calculating the dot product of the input data with the weights of the first hidden layer (`W1`). The bias vector for the same layer is added to the result. Then, these values are passed through the sigmoid activation function (`A1`). The same process is repeated for the second (output) layer to obtain the predicted output of the model (`Y_hat`).
- The mean squared error loss function is calculated, which measures the average squared difference between the predicted output and the actual output. This will be used as a metric to evaluate the performance of the model.
- The backward propagation phase begins, where the gradients are computed, and the weights and biases are adjusted. The derivatives concerning the error are calculated for each layer, followed by the gradients of the weights and biases for each layer. These gradients are used to update the weights and biases using the learning rate set earlier.
- The final value of the loss function and the adjusted values of weight and bias vectors are reported.

Overall, this code implements a simple neural network that can learn from a small dataset using batch processing and adjust its parameters to minimize the mean squared error loss. However, depending on the problem at hand, more complex neural network architectures and other optimization algorithms may be required for better performance.

CODE COMPLETION:

This is a simple neural network with one hidden layer that performs batch processing to adjust its weights and biases using backpropagation. The output should show the mean squared error loss and the adjusted weight and bias values for each layer of the neural network.

CODE BUGS:

The code seems to be syntactically correct and should run without errors. However, there could be some issues with the neural network's design and implementation that may affect its performance. For instance:

- The current neural network only has one hidden layer with a few nodes. Increasing the number of nodes or layers could potentially increase its accuracy.
- The learning rate is arbitrarily set to 0.1. This value may not work well for all types of problems, and choosing the right learning rate can significantly affect the model's training time and results.

- The dataset used here is very small, which may not accurately represent real-world data. Using more extensive and diverse datasets can improve the model's ability to generalize.
- The batch size used in this code is 1, which means the weights and biases are updated after every data point is processed. Using larger batch sizes can help the model converge faster and provide better results.

Overall, these are just recommendations, and the optimal neural network architecture heavily depends on the specific problem being addressed.