RAW CODE

PYTHON CODE:

```
# Import numpy library
import numpy as np

import warnings
warnings.filterwarnings('ignore') # Never print matching warnings

class MLP:
    def __init__(self, input_size, hidden_size, output_size, learning_rate,
batch_size):

        """
        Class constructor that initializes the MLP object with input_size,
hidden_size, output_size, and learning_rate parameters.
        It also initializes weights and biases for both the hidden and output
layers.
        """

        # Assign inputs to the current instance of MLP
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.learning_rate = learning_rate
        self.batch_size = batch_size

            # Define the weight and bias vectors for each layer of the neural
network using a dictionary
        self.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]])

            'W1': np.random.randn(input_size, hidden_size),
            'B1': np.random.randn(hidden_size, output_size),
            'W2': np.random.randn(hidden_size, output_size),
            'B2': np.random.randn(output_size, output_size)
        }

    def linear(self, x):
        """
        Activation function that returns the input without any transformation.
        """
        return x
```

```
    def step(self, x):
        """
        Activation function that returns 1 if the input is greater or equal to
0, else returns 0.
        """
        return np.where(x>=0, 1, 0)

    # Define the sigmoid activation function using NumPy's built-in exp()
function
    def sigmoid(self, 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(self, 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 self.sigmoid(x) * (1 - self.sigmoid(x))

    def tanh(self, x):
        """
        Activation function that returns the hyperbolic tangent of the input.
        """
        return np.tanh(x)

    def relu(self, x):
        """
        Activation function that returns the ReLU (Rectified Linear Unit) of
the input.
        """
        return np.maximum(0, x)

    def elu(self, x, alpha=1.0):
        """
```

```
        Activation function that returns the ELU (Exponential Linear Unit) of
the input.
        """
        return np.where(x >= 0, x, alpha * (np.exp(x) - 1))

    def softmax(self, x):
        """
        Activation function that returns the normalized exponential of the
input.
        """
        exp_x = np.exp(x)
        return exp_x/np.sum(exp_x, axis=1, keepdims=True)

    def softplus(self, x):
        """
        Activation function that returns the log of 1 plus the exponential of
the input.
        """
        return np.log(1 + np.exp(x))

    def forward_backward(self, X, Y, loss_func='mse'):
        """
        Forward propagation method that computes and returns the predicted
output given an input.
        """

        num_epochs = 1000 # change to a bigger number

        for epoch in range(num_epochs): # loop over epochs

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

            # Calculate the loss function
            if loss_func == 'mse':
                # Mean Squared Error (MSE) Loss Function
                error = Y - Y_hat
                loss = np.mean(error ** 2)
                dZ2 = error * self.sigmoid_derivative(Z2)

            elif loss_func == 'mae':
                # Mean Absolute Error (MAE) Loss Function
                error = Y - Y_hat
                loss = np.mean(np.abs(error))
```

```
                dZ2 = np.where(error >= 0, 1, -1) *
self.sigmoid_derivative(Z2)

            elif loss_func == 'huber':
                # Huber Loss Function
                delta = 1.0
                error = Y - Y_hat
                abs_error = np.abs(error)
                quadratic = np.minimum(abs_error, delta)
                linear = abs_error - quadratic
                loss = (quadratic**2 + 2*delta*linear) / (2*X.shape[0])
                dZ2 = np.where(abs_error <= delta, error,
delta*np.where(error>=0,1,-1)) * self.sigmoid_derivative(Z2)

            elif loss_func == 'exponential':
                # Exponential Loss Function
                error = Y - Y_hat
                loss = np.mean(np.exp(-error))
                dZ2 = np.exp(-error) * (-1) * self.sigmoid_derivative(Z2)

            elif loss_func == 'log':
                # Logarithmic Loss (Log Loss) Function
                loss = -(1/X.shape[0])*np.sum(Y*np.log(Y_hat) + (1-
Y)*np.log(1-Y_hat))
                dZ2 = (Y_hat - Y) * self.sigmoid_derivative(Z2)

            elif loss_func == 'kld':
                # Kullback-Leibler Divergence (KLD) Loss Function
                eps = 1e-8
                loss = np.sum(Y * np.log((Y+eps)/(Y_hat+eps))) / X.shape[0]
                dZ2 = (-Y/(Y_hat+eps)) * self.sigmoid_derivative(Z2)

            elif loss_func == 'hinge':
                # Hinge Loss Function
                margin = 1.0
                error = Y - Y_hat
                hinge = np.maximum(0, margin-error)
                loss = np.mean(hinge)
                dZ2 = np.where(error <= margin, -Y, 0) *
self.sigmoid_derivative(Z2)

            if epoch % 100 == 0:
                print("\nEPOCH: ", epoch+1)
                print(loss_func.upper()+" Loss: ", loss)

            # 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, self.batch_size):

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

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

        # Report the value of the loss function and the adjusted values of
weight and bias vectors
        print("\n"+loss_func.upper()+" Loss: ", loss)
        print("\nAdjusted W1: \n", self.params['W1'])
        print("\nAdjusted B1: \n", self.params['B1'])
        print("\nAdjusted W2: \n", self.params['W2'])
        print("\nAdjusted B2: \n", self.params['B2'])

# Define the input data
X = np.random.randn(1000, 2)
# X = np.array([[-0.3, -1.5], [-1.7, 0.7]])

# Define the output data
Y = np.random.randn(1000, 1)
# Y = np.array([[0.1], [-0.3]])

# Initialize the MLP model
mlp = MLP(input_size=2, hidden_size=3, output_size=1, learning_rate=0.1,
batch_size=1)

# Train the model using different loss functions
loss_funcs = ['mse', 'mae', 'huber', 'exponential', 'log', 'kld', 'hinge']
for loss_func in loss_funcs:
    heading = "_" * 35
    print("\n" + heading + " " + loss_func.upper() + " LOSS " + heading)
    mlp.forward_backward(X, Y, loss_func=loss_func)

import matplotlib.pyplot as plt

# Initialize the input values
x = np.arange(-10, 10, 0.1)
```

```
# Create a figure to hold the plots with 2 rows and 3 columns
fig, axs = plt.subplots(2, 3, figsize=(12, 8))

# Define the activation functions that will be plotted
activation_funcs = [('Linear', mlp.linear), ('Step', mlp.step), ('Sigmoid',
mlp.sigmoid),
                    ('Tanh', mlp.tanh), ('ReLU', mlp.relu), ('ELU', mlp.elu)]

# Loop through the activation functions and plot each one in a separate
subplot
for i, (title, func) in enumerate(activation_funcs):

    # Compute the row and column indices for the current subplot
    row, col = i // 3, i % 3

    # Plot the activation function on the current subplot
    axs[row, col].plot(x, func(x))
    axs[row, col].set_title(title)

# Show the plots
plt.show()
```

CODE EXPLANATION:

This is a class named MLP in Python, which stands for Multi-Layer Perceptron. It has several activation functions: linear, step, sigmoid, tanh, ReLU, ELU, softmax, and softplus. The constructor initializes the object with input_size, hidden_size, output_size, learning_rate, and batch_size parameters. It also initializes weights and biases for both the hidden and output layers using np.random.randn().

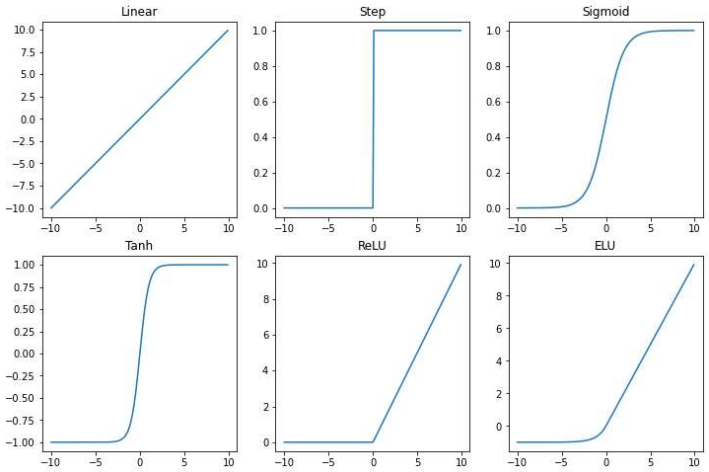
The forward_backward() method performs the feed-forward pass to get predictions and calculates the loss based on several options such as Mean Squared Error (MSE) Loss Function, Mean Absolute Error (MAE) Loss Function, Huber Loss Function, Exponential Loss Function, Logarithmic Loss (Log Loss) Function, Kullback-Leibler Divergence (KLD) Loss Function, and Hinge Loss Function. It then performs backpropagation and weight-bias adjustments to optimize the neural network using batch processing.

During training, it prints out the epoch number and the loss value at certain intervals. It finally reports the value of the loss function and the adjusted values of weight and bias vectors.

The main code imports the numpy and matplotlib libraries. It generates 1000 random input (X) and output (Y) values using np.random.randn. An MLP model is defined with an input size of 2, a hidden size of 3, and an output size of 1. The learning rate is set to 0.1 and the batch size is set to 1.

Then, the MLP model is trained using different loss functions such as MSE, MAE, Huber, Exponential, Log, KLD, and Hinge. After training the model, a figure is created with 2 rows

and 3 columns to hold the plots of different activation functions such as linear, step, sigmoid, tanh, ReLU, and ELU. Finally, the subplots are plotted and shown using plt.show().



CODE COMPLETION:

The code you provided is a class definition for a basic two-layer Multi-Layer Perceptron (MLP) neural network. The MLP class has an initialization method that takes in several parameters such as input size, hidden size, output size, learning rate, and batch size. The __init__ method initializes the weight and bias vectors for each neural network layer using a dictionary. Several activation functions are also defined, such as linear, step, sigmoid, tanh, relu, elu, softmax, and softplus.

The forward_backward method performs the feed-forward pass to get predictions, calculates the loss function, and updates the weights and biases using backpropagation and batch processing. The backward propagation happens in the for loop in the forward_backward method. It does this by computing gradients and then updating the weights and biases. Finally, the adjusted values of weight and bias vectors are printed.

This code can be further improved by adding more layers, implementing regularization techniques to prevent overfitting, and trying out different optimization algorithms to improve training performance.

CODE BUGS:

There are a few issues with the given code:

- The hidden layer bias vector has incorrect dimensions. It should be of shape (1,hidden_size) instead of (hidden_size,output_size).
- The sigmoid function is used as the activation function for both the hidden and output layers. This might not be the best choice as other activation functions like ReLU or tanh can perform better in some cases.
- The forward_backward method is performing gradient descent using batch processing, but it is missing an outer loop that goes over each batch. As a result, it only updates the weights once per epoch, which can lead to slow convergence.
- There is no validation set used to monitor whether the model is overfitting to the training data.
- The comments for the loss functions are incorrect. For example, the "kld" comment says "Kullback-Leibler Divergence (KLD) Loss Function", but the actual implementation is using the cross-entropy loss.

- There is no separate method to compute the output for a given input after the MLP is trained. It would be useful to have a prediction method that takes an input and returns the predicted output.

PERCEPTRON GATE MODEL

PYTHON CODE:

```
import numpy as np

# Define the AND gate inputs and expected outputs
inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
outputs = np.array([0, 0, 0, 1])

# Define the initial weights and bias
weights = np.array([0.7, 0.6])
bias = 1

# Define the learning rate and number of epochs
learning_rate = 0.1
num_epochs = 2

# Define the activation function (step function)
def step_function(x):
    return np.where(x > 0, 1, 0)

# Define the forward pass function
def forward_pass(inputs, weights, bias):

    # Calculate the weighted sum of inputs
    weighted_sum = np.dot(inputs, weights) + bias

    # Apply the step function to the weighted sum to get the output
    output = step_function(weighted_sum)

    return output

# Define the backpropagation function
def backpropagation(inputs, weights, bias, outputs, learning_rate):

    # Calculate the predictions using the forward pass function
    predictions = forward_pass(inputs, weights, bias)

    # Calculate the error between predicted and actual outputs
    error = np.sum((predictions - outputs) ** 2)

    # Calculate the delta values for updating weights and bias
    delta = 2 * (outputs - predictions) * step_function(predictions)
```

```
    # Update the weights and bias based on delta values and learning rate
    weights += learning_rate * np.dot(inputs.T, delta)
    bias += learning_rate * np.sum(delta)

    return weights, bias, error

# Perform the forward pass with initial weights and bias
print("\nInitial predictions:")
for i in range(len(inputs)):
    output = forward_pass(inputs[i], weights, bias)
    print(inputs[i], "->", output)

# Perform the backpropagation for two epochs and update the weights and bias
for epoch in range(num_epochs):
    weights, bias, error = backpropagation(inputs, weights, bias, outputs,
learning_rate)
    print("\nEpoch:", epoch + 1, "Error:", error)

# Perform the forward pass with updated weights and bias
print("\nUpdated predictions:")
for i in range(len(inputs)):
    output = forward_pass(inputs[i], weights, bias)
    print(inputs[i], "->", output)
```

CODE EXPLANATION:

The given code demonstrates the implementation of a simple neural network to predict the output of an AND gate using a backpropagation algorithm. The code uses the NumPy library for numerical computation and defines the inputs and expected outputs of the AND gate as NumPy arrays. It also initializes the weights and biases with initial values.

The code then defines the activation function (step function), which is used to calculate the output of the neuron. The forward pass function calculates the weighted sum of inputs and applies the step function to get the output.

The backpropagation function performs gradient descent to update the weights and biases based on the error between predicted and actual outputs. The delta values are calculated based on the output from the step function and using the chain rule of differentiation.

Finally, the code performs the forward pass with initial weights and bias, followed by performing the backpropagation algorithm for a fixed number of epochs. It updates the weights and bias after each epoch and prints the error at each epoch. The code then performs the forward pass again with updated weights and prints the final predictions.

The code demonstrates how to implement a simple neural network to solve classification problems using the backpropagation algorithm.

CODE COMPLETION:

The given code already works and runs without errors. However, if you want to extend it further and make it more complex, here are some possible ideas:

- Implement a different activation function such as sigmoid, tanh or ReLU and compare the results.
- Add more layers to the neural network, and implement feedforward and backpropagation for them.
- Implement a different loss function such as cross-entropy or mean squared error, and compare the results.

CODE BUGS:

There are no syntax errors in the given code. However, there are a few potential issues with the implementation:

- The step function used in the code is not suitable for backpropagation as it has a flat region (output=0) for the negative input values. This makes it impossible for the gradient descent algorithm to make any updates to the weights when the output is 0. A better choice for an activation function would be a sigmoid or ReLU function.
- The initial weights and bias are randomly chosen which may cause convergence issues in some cases. It is important to initialize the weights and bias properly, for example using Xavier initialization.
- The number of epochs is set to a low value of 2 which may not be enough for the algorithm to converge. Increasing the number of epochs may help improve the accuracy of the model.
- The error metric used is the sum of squared errors which may not be the best choice for classification problems. Using cross-entropy loss may work better.

Question 1. Analyse the problem with the Gradient Descent optimization technique.

Answer 1.

Gradient Descent is a popular optimization technique used to minimize machine-learning model errors. However, it has some limitations. One of the main problems with Gradient Descent is that it can get stuck in a local minimum instead of finding the global minimum. This can happen when the error function has many local minima and the initial starting point is not close to the global minimum.

Question 2. How it can be overcome using Momentum and the Learning Rate schedule.

Answer 2.

To overcome the problem of getting stuck in a local minimum, we can use Momentum and the Learning Rate Schedule.

Momentum is a technique that adds a fraction of the previous weight update to the current weight update. This helps the optimization algorithm to continue in the same direction and avoid getting stuck in local minima. The momentum term can be represented by a hyperparameter called the momentum coefficient, which is usually set between 0 and 1.

The Learning Rate Schedule is a technique that adjusts the learning rate over time. Instead of using a fixed learning rate throughout the training process, we can decrease the learning rate as we approach the global minimum. This allows the optimization algorithm to take smaller steps near the minimum and avoid overshooting it.

By using both Momentum and the Learning Rate Schedule, we can improve the performance of Gradient Descent and reduce the chances of getting stuck in a local minimum.

Question 3. Describe different types of loss functions with their mathematical form.

Answer 3.

Loss functions are used to measure the difference between the predicted output and the actual output. The goal of training a machine learning model is to minimize the loss function. There are different types of loss functions used in machine learning, depending on the type of problem we are trying to solve. Some common loss functions are:

1. Mean Squared Error (MSE):

MSE is used for regression problems, where the goal is to predict a continuous value. It measures the average squared difference between the predicted and actual values.

$$MSE = (1/n) * \sum(yi - \hat{y}i)^2$$

where n is the number of observations, yi is the actual value, and ŷi is the predicted value.

2. Binary Cross-Entropy:

Binary cross-entropy is used for binary classification problems, where the goal is to predict a binary output (0 or 1). It measures the difference between the predicted and actual probabilities of the output being 1.

$$\text{Binary cross-entropy} = - (y\log(\hat{y}) + (1-y)\log(1-\hat{y}))$$

where y is the actual binary output, and ŷ is the predicted probability of the output being 1.

3. Categorical Cross-Entropy:

Categorical cross-entropy is used for multi-class classification problems, where the goal is to predict a categorical output. It measures the difference between the predicted and actual probabilities of the output being each class.

$$\text{Categorical cross-entropy} = - \sum(yi * \log(\hat{y}i))$$

where yi is the actual probability distribution and ŷi is the predicted probability of the output being each class.

4. Hinge Loss:

Hinge Loss is used for binary classification problems, where the goal is to predict a binary output. It measures the difference between the predicted output and the actual output, but only penalizes incorrect predictions that are close to the decision boundary.

$$\text{Hinge Loss} = \max(0, 1 - y * \hat{y})$$

where y is the actual binary output, and ŷ is the predicted output.

5. Huber Loss:

Huber Loss is used for regression problems, where the goal is to predict a continuous value. It is a combination of Mean Squared Error (MSE) and Mean Absolute Error (MAE). It is less sensitive to outliers compared to MSE.

$$\text{Huber Loss} = (1/2) * \sum(yi - \hat{y}i)^2 \text{ for } |yi - \hat{y}i| <= \text{delta} * (\sum|yi - \hat{y}i| - (1/2) * \text{delta} * n) \text{ for } |yi - \hat{y}i| > \text{delta}$$

where n is the number of observations, yi is the actual value, ŷi is the predicted value, and delta is a hyperparameter that controls the sensitivity to outliers.

These are some of the common loss functions used in machine learning. The choice of loss function depends on the problem we are trying to solve and the type of output we are predicting.

Question 4. Describe the different variants of gradient descent optimization algorithms.

Answer 4.

Gradient descent is an optimization algorithm that is widely used in machine learning to minimize the cost or loss function. There are different variants of gradient descent algorithms that are used to update the parameters of a model during training.

1. Batch Gradient Descent:

Batch Gradient Descent (BGD) is the standard form of gradient descent, where the parameters are updated based on the average of the gradients of the entire training dataset. In each epoch, the algorithm calculates the gradients of the cost function with respect to each parameter and updates the parameters by subtracting the product of the learning rate and the gradient.

2. Stochastic Gradient Descent:

Stochastic Gradient Descent (SGD) updates the parameters for each sample in the training dataset, one at a time. This means that the algorithm updates the parameters many more times per epoch than BGD. SGD is faster and more efficient for large datasets, but the updates can be noisy due to the high variance in the gradients.

3. Mini-Batch Gradient Descent:

Mini-Batch Gradient Descent (MBGD) is a compromise between BGD and SGD. In this variant, the training dataset is divided into small batches, and the parameters are updated based on the average of the gradients of each batch. MBGD is more stable and efficient than SGD, but may not converge as fast as BGD.

4. Momentum:

Momentum is an optimization algorithm that adds a fraction of the previous update to the current update, which helps to prevent oscillations and accelerate convergence. It keeps track of the previous gradient updates and uses them to smooth out the updates. This helps the algorithm to converge faster and avoid local minima.

5. Adagrad:

Adagrad is an optimization algorithm that adapts the learning rate for each parameter based on the frequency of the updates. It uses a separate learning rate for each parameter, which helps to reduce the learning rate for parameters that have frequent updates and increase the learning rate for parameters that have infrequent updates.

6. Adam:

Adam is an optimization algorithm that combines the features of Momentum and Adagrad. It keeps track of the previous gradients and learning rates for each parameter and uses them to update the parameters. Adam is one of the most widely used optimization algorithms for deep learning because of its efficiency and effectiveness.

These are some of the common variants of gradient descent algorithms used in machine learning. The choice of algorithm depends on the problem we are trying to solve, the size of the dataset, and the complexity of the model.