PRACTICAL 5 - IMPLEMENTING THE BACKPROPAGATION ALGORITHM TO UPDATE THE WEIGHTS OF AN MLP

AIM:

- Understand and implement the backpropagation algorithm, which is a widely used method for training multi-layer neural networks.
- Involves building an MLP with one or more hidden layers, and then using backpropagation to update the weights of the network to minimize the error between the predicted and actual outputs on a given dataset.
- Gain a better understanding of how backpropagation works and how it can be used to improve the performance of neural networks on classification and regression tasks.

SOFTWARE:

Anaconda3 2021.11 | Spyder 5.1.5 | Python 3.9.7 – 64-Bit

THEORETICAL BACKGROUND:

The backpropagation algorithm is used to update the weights of a multi-layer neural network to minimize the difference between the predicted and actual outputs on a given dataset. The algorithm works by propagating the error back through the network from the output layer to the input layer and then adjusting the weights in each layer based on the error signals.

The backpropagation algorithm involves two phases: forward propagation and backward propagation. In the forward propagation phase, the input values are fed into the network, and the activations of the neurons in each layer are computed using a set of weights that are initialized randomly. The activations of the neurons in the output layer are then compared with the target values, and the difference between them is computed as the error signal.

In the backward propagation phase, the error signal is propagated backwards through the network to compute the weight updates for each layer. The algorithm uses the chain rule of calculus to compute the derivative of the error with respect to each weight in the network. The weight updates are then computed by multiplying the error derivative with the activation values of the neurons in the layer.

One of the challenges in implementing the backpropagation algorithm is dealing with the problem of vanishing gradients. This occurs when the gradients of the error concerning the weights become very small in deeper layers of the network, making it difficult to update the weights and causing the network to converge slowly. To overcome this issue, various techniques have been developed, such as using non-linear activation functions, weight initialization schemes, and regularization techniques.

RAW PRACTICAL CODE:

1. PYTHON CODE

```
# Import necessary libraries
import numpy as np

# Define the MLP class and its methods
class MLP:
    # Constructor function to initialize model parameters
    def __init__(self, input_size, hidden_size, output_size, learning_rate):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.learning_rate = learning_rate

        # Randomly initialize weights and biases for hidden and output layers
        self.h_weights = np.random.randn(input_size, hidden_size)
        self.h_bias = np.random.randn(hidden_size)
        self.o_weights = np.random.randn(hidden_size, output_size)
        self.o_bias = np.random.randn(output_size)

    # Activation functions
    def linear(self, x):
        return x

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

    def sigmoid(self, x):
        return 1/(1+np.exp(-x))

    def tanh(self, x):
        return np.tanh(x)

    def relu(self, x):
        return np.maximum(0, x)

    def elu(self, x, alpha=1.0):
        return np.where(x >= 0, x, alpha * (np.exp(x) - 1))

    def softplus(self, x):
        return np.log(1 + np.exp(x))

    # Forward propagation function
    def forward(self, X):
        h_output = np.dot(X, self.h_weights) + self.h_bias
        h_activation = self.sigmoid(h_output) # apply activation function to
hidden layer output
```

```
        o_output = np.dot(h_activation, self.o_weights) + self.o_bias
        o_activation = self.sigmoid(o_output) # apply activation function to
output layer output

        return o_activation, h_activation

    # Backward propagation function
    def backward(self, X, y, output, h_activation):
        error = (y - output)
        o_delta = error * self.sigmoid_derivative(output) # calculate output
delta
        hidden_error = np.dot(o_delta, self.o_weights.T)
        hidden_delta = hidden_error * self.sigmoid_derivative(h_activation) #
calculate hidden layer delta
        # update weights and biases for output and hidden layers
        self.o_weights += self.learning_rate * np.dot(h_activation.T, o_delta)
        self.o_bias += self.learning_rate * np.sum(o_delta, axis=0)
        self.h_weights += self.learning_rate * np.dot(X.T, hidden_delta)
        self.h_bias += self.learning_rate * np.sum(hidden_delta, axis=0)

    # Derivative of the sigmoid function
    def sigmoid_derivative(self, x):
        return x * (1 - x)

    # Training function
    def train(self, X, y, epochs):
        for epoch in range(epochs):
            output, hidden_layer = self.forward(X)
            self.backward(X, y, output, hidden_layer)

    # Prediction function
    def predict(self, X):
        return self.forward(X)

# Generate random input features and labels
X = np.random.randn(1000, 4)
y = np.random.randn(1000, 1)

# Create an MLP model object with specified parameters
mlp = MLP(input_size=4, hidden_size=5, output_size=1, learning_rate=0.1)

# Train MLP model on generated data
mlp.train(X, y, epochs=100)

# Test the trained MLP model by predicting output given input features
print(mlp.predict(X))
```

2. CODE EXPLANATION

The code defines a basic Multi-Layer Perceptron (MLP) class in Python programming language that can be used for training and predicting outputs on user-defined input data. The class contains methods for forward and backward propagation along with activation functions like sigmoid, tanh, relu, etc. It initializes the MLP's parameters like learning rate, the number of neurons in each hidden layer, weights, biases, activation functions, etc. using the constructor method.

The forward propagation function takes input features, applies weights & biases to the neurons in the hidden layer(s), applies an activation function to the output of hidden layer(s), and then passes it through another set of weights and biases to get the final output. Whereas the backward propagation function calculates the error between the actual and predicted output, computes deltas for both layers and updates weights and biases accordingly.

The training loop iterates over defined epochs and calls forward and backward propagation functions over each epoch to update weights to find a best-fit model. Lastly, predict method is called to predict the output given input features for evaluating model performance.

Overall, this code provides a simple implementation of MLP that can be further improved by adding validation, early stopping, optimizers or other techniques.

3. CODE COMPLETION

The code provided defines a class `MLP` that implements a Multi-Layer Perceptron model. The model takes in input data of shape (n_samples, n_features) and trains the network to predict an output of shape (n_samples, 1).

The `MLP` class has the following methods:

- `__init__(self, input_size, hidden_size, output_size, learning_rate)`: This method initializes the weights and biases for the hidden and output layers randomly with values drawn from a normal distribution using NumPy's `np.random.randn()` function. The input_size parameter is the number of features in the input data, hidden_size is the number of neurons in the hidden layer, output_size is the number of neurons in the output layer and learning_rate is the step size at each iteration while optimizing the parameters.
- `linear(self, x)`: This method returns the input as is without any transformation.
- `step(self, x)`: This method returns binary outputs based on whether the input is above or below zero.
- `sigmoid(self, x)`: This method applies the sigmoid activation function.
- `tanh(self, x)`: This method applies the hyperbolic tangent activation function.
- `relu(self, x)`: This method applies the Rectified Linear Unit activation function.
- `elu(self, x, alpha=1.0)`: This method applies the Exponential Linear Units activation function.
- `softplus(self, x)`: This method applies the Softplus activation function.

The `MLP` class also has the following additional methods which implement the training, forward propagation, backward propagation and prediction steps of the MLP algorithm:

- `forward(self, X)`: This method computes the forward pass of the MLP by computing the dot product between the input data and the hidden layer weights, adding the bias term, applying the non-linear activation function on this output and then computing the dot product between this output and the output layer weights, adding the bias term and finally applying the non-linear activation function. The method returns the output of the MLP as well as the output of the hidden layer.
- `backward(self, X, y, output, h_activation)`: This method computes the derivative of the loss with respect to the weights and biases in both layers using the backpropagation algorithm and updates these parameters accordingly. Specifically, it first computes the error between the predicted output and the true output, calculates the delta for the output layer and then propagates this delta backwards to compute the delta for the hidden layer. Finally, the method updates the weights and biases in both layers using the computed deltas and the input data.
- `sigmoid_derivative(self, x)`: This method returns the derivative of the sigmoid activation function as implemented in the `sigmoid()` method.
- `train(self, X, y, epochs)`: This method trains the MLP by repeatedly calling the `forward()` and `backward()` methods for a given number of `epochs`, updating the weights and biases at each iteration.
- `predict(self, X)`: This method uses the trained MLP model to make predictions on new input data `X`.

Finally, the code instantiates an `MLP` object with the specified architecture, trains it on randomly generated input features and labels, and then predicts the output of the trained model on the same input features. Note that for the proper execution of the code, you need to import necessary libraries like numpy in the first line to avoid possible errors.

4. <u>CODE BUGS</u>
    - The sigmoid_derivative method is not being called with necessary inputs in the backward method.
    - In the constructor method, the random initialization of biases must be based on the number of neurons in respective layers.
    - There is no validation set to evaluate model performance.
    - The training function doesn't have any stopping criterion.
    - The MLP class should include methods for saving and loading models.
    - The prediction method only returns the output probability of the last layer but doesn't decide a class or regression value.

FINAL PRACTICAL CODE:

1. PYTHON CODE

```
import numpy as np

# Define activation functions
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def sigmoid_derivative(z):
    return z * (1 - z)

# Define propagation functions with activation parameter
def forward_propagation(X, W1, b1, W2, b2, activation=sigmoid):
    Z1 = np.dot(X, W1) + b1
    A1 = activation(Z1)
    Z2 = np.dot(A1, W2) + b2
    A2 = activation(Z2)

    cache = {"Z1": Z1,
             "A1": A1,
             "Z2": Z2,
             "A2": A2}

    return A2, cache

def backward_propagation(X, Y, cache, W1, b1, W2, b2, learning_rate=0.1,
activation_derivative=sigmoid_derivative):
    m = X.shape[0]

    # Retrieve cached data from the forward propagation
    A1 = cache['A1']
    A2 = cache['A2']

    # Compute gradients
    dZ2 = (A2 - Y) * activation_derivative(A2)
    dW2 = np.dot(A1.T, dZ2) / m
    db2 = np.sum(dZ2, axis=0, keepdims=True) / m

    dZ1 = np.dot(dZ2, W2.T) * activation_derivative(A1)
    dW1 = np.dot(X.T, dZ1) / m
    db1 = np.sum(dZ1, axis=0, keepdims=True) / m

    # Update weights and biases using the learning rate and gradients
    W1_new = W1 - learning_rate * dW1
    b1_new = b1 - learning_rate * db1
    W2_new = W2 - learning_rate * dW2
    b2_new = b2 - learning_rate * db2
```

```
    return W1_new, b1_new, W2_new, b2_new

# Define the training function with the hidden_size parameter
def train(X, Y, hidden_size=4, iterations=10000, print_loss=False,
activation=sigmoid, activation_derivative=sigmoid_derivative):
    input_size = X.shape[1] # Number of input nodes
    output_size = Y.shape[1] # Number of output nodes

    # Initialize the weights and biases
    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))

    for i in range(iterations):
        # Forward propagation
        output, cache = forward_propagation(X, W1, b1, W2, b2, activation)

        # Backward propagation
        W1_new, b1_new, W2_new, b2_new = backward_propagation(X, Y, cache, W1,
b1, W2, b2, activation_derivative=activation_derivative)

        # Print loss every 1000 iterations
        if print_loss and i % 1000 == 0:
            print("Loss after iteration ", i, ": ", np.mean(np.abs(output -
Y)))

        # Update weights and biases
        W1, b1, W2, b2 = W1_new, b1_new, W2_new, b2_new

    return W1, b1, W2, b2

# Test the model with example data
X = np.array([[0, 0, 1],
              [0, 1, 1],
              [1, 0, 1],
              [1, 1, 1]])

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

W1, b1, W2, b2 = train(X, Y, iterations=10000, print_loss=True)

# Predict the output using trained weights and biases
output, _ = forward_propagation(X, W1, b1, W2, b2)
```

```
print("\nOutput: \n", output)
```

2. CODE EXPLANATION

The above code is a neural network implementation in Python using the NumPy library. It defines functions for forward and backward propagation, as well as training the model. The activation function used is sigmoid, which takes an input value and maps it to a value between 0 and 1, making it useful for binary classification problems. The sigmoid_derivative function returns the derivative of the sigmoid function.

The forward_propagation function takes in inputs (X), weights (W1 and W2) and biases (b1 and b2) as arguments and calculates the output using the sigmoid activation function. The results are stored in a dictionary called "cache" which is returned along with the final output value.

The backward_propagation function is responsible for calculating the gradients and updating the weights and biases based on those gradients. It takes in the inputs (X), expected outputs (Y), cache from forward propagation, current weights and biases, learning rate and the derivative of the activation function. It then computes the gradients using the chain rule and updates the weights and biases using these calculated gradients.

The training function takes in training data (X and Y) and the number of hidden layer neurons (hidden_size) among other things, and returns the optimized weights and biases. The function initializes weights and biases and then alternates between forward propagation and backward propagation to calculate gradients and update weights/biases.

The process is repeated for the specified number of iterations. Lastly, test data is provided to the trained model, which predicts the outputs using the optimized weights and biases obtained from training.

3. CODE COMPLETION

The code is a basic implementation of a neural network with a single hidden layer using the sigmoid activation function. It trains on input data to output certain target values. The code defines functions for forward propagation, backpropagation, and training the model. It also tests the model using sample data and prints the predicted output.

4. CODE BUGS

I couldn't find any issues with the code. However, I suggest one improvement, which is using a more efficient optimization algorithm like Adam instead of updating weights and biases using the learning rate and gradients.

CONCLUSION:

First, the implementation of the backpropagation algorithm provides an understanding of how neural networks can be trained to minimize the error between predicted and actual outputs. By propagating error signals back through the network, the algorithm updates the weights of the network in each layer to reduce the error and improve the accuracy of predictions.

Second, the implementation of the backpropagation algorithm highlights the importance of selecting appropriate activation functions, weight initialization schemes, and regularization techniques to improve the performance of the network. For example, using non-linear activation functions can help overcome the problem of vanishing gradients and improve the convergence speed of the network.

Third, the implementation of the backpropagation algorithm allows for the comparison of the performance of an MLP with that of a single-layer perceptron. By training and testing both types of networks on the same dataset, it is possible to evaluate the improvement in accuracy achieved by the MLP due to its ability to model non-linear relationships between the input and output variables.

Finally, the implementation of the backpropagation algorithm provides insights into the limitations of neural networks, including the risk of overfitting to the training data and the need for careful tuning of hyperparameters such as the learning rate, batch size, and the number of hidden units in the network. Through experimentation with these hyperparameters, it is possible to achieve a well-performing network that generalizes well to new data.