K. J. Somaiya College of Engineering, Mumbai-77
(A Constituent College of Somaiya Vidyavihar University)
Department of Computer Engineering

Batch:C-2 Roll No.:16010122323

Experiment No. : 07

Grade: AA / AB / BB / BC / CC / CD /DD

Signature of the Staff In-charge with date

Title: Error Back propagation algorithm

Objective: To Write a program to implement error back propagation algorithm in MLP.

Expected Outcome of Experiment:

CO2 : Analyze various neural network architectures

Books/ Journals/ Websites referred:

Pre Lab/ Prior Concepts:

Back-propagation training algorithm :

Back-propagation is a training algorithm for neural networks, based on gradient descent.

1. **Forward Propagation: Input data passes through the network to compute predicted output.**
2. **Compute Loss: Calculate error between predicted and actual output using a loss function (e.g., MSE or cross-entropy).**

Back-propagation:

- **Compute gradients of the loss with respect to each weight and bias using the chain rule.**
- **Error is propagated backward from output to input layers.**

3. **Update Weights: Adjust weights and biases using gradient descent.**
4. **Repeat: The process continues for multiple iterations (epochs) until the error is minimized.**

Network Architecture :

Network architecture refers to the structure of an artificial neural network, defining how neurons are arranged in layers and how these layers interact. The main components of a neural network architecture include:

1. Input Layer:

- Receives the raw data (features) to be processed.
- Number of neurons in the input layer corresponds to the number of features in the data.

2. Hidden Layers:

- Layers between the input and output that perform computations and feature transformations.
- Neurons in hidden layers apply weights, biases, and activation functions (e.g., ReLU, sigmoid).
- More hidden layers form a deep neural network (DNN).

3. Output Layer:

- Produces the final result (predictions).
- The number of neurons corresponds to the output needed (e.g., 1 for binary classification, multiple for multi-class).

4. Connections (Weights and Biases):

- Each connection between neurons has a weight and bias that are adjusted during training.

5. Activation Functions:

- Non-linear functions applied to each neuron's output to introduce non-linearity (e.g., ReLU, tanh, sigmoid).

Algorithm: (Backpropogation):

1. **Initialize** weights and biases randomly for the network.
2. **Forward Propagation**:

- Input data passes through the network layer by layer.
- Compute weighted sum of inputs, apply activation functions, and generate the output.

3. **Compute Loss**:

- Compare the predicted output with the actual target using a loss function (e.g., mean squared error, cross-entropy).

4. **Back-propagation of Error**:

- **Step 1**: Calculate the gradient of the loss with respect to the output (derivative of loss function).
- **Step 2**: Propagate error backward through the layers, calculating the gradient for each weight and bias using the chain rule.

5. **Update Weights and Biases**:

- Adjust weights and biases using gradient descent:

- New weight = Old weight - (Learning rate * Gradient of weight).
- New bias = Old bias - (Learning rate * Gradient of bias).

6. **Repeat**:

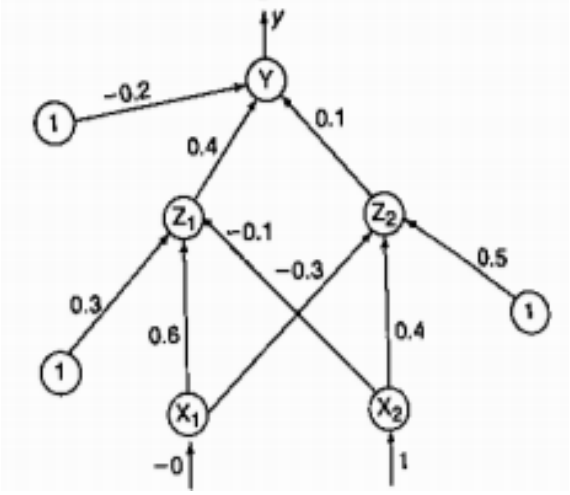
- Repeat the process (forward pass, back-propagation, weight update) for multiple iterations (epochs) until the loss is minimized.

7. **Convergence**:

- Continue training until the model reaches a satisfactory error level or after a set number of epochs.

Implementation Details :

To implement EBPT algorithm to find the new weights for the network given for given input output pair for certain no. of epochs.



It is presented with the input pattern [0, 1] and the target output is 1.

Use a learning rare a = 0.25 and binary sigmoidal activation function.

Program:

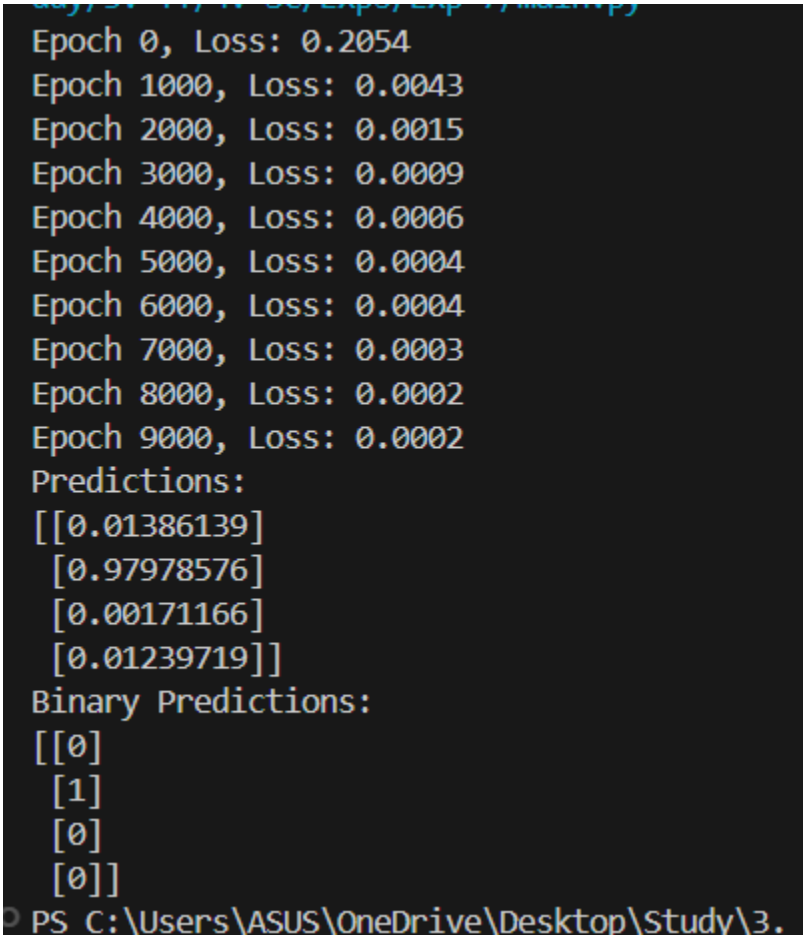
```
import numpy as np


class NeuralNetwork:

    def __init__(self, input_dim, hidden_dim, output_dim, lr=0.1):

        # Initialize weights using Xavier initialization

        self.W1 = np.random.randn(input_dim, hidden_dim) * np.sqrt(1. /
input_dim)

        self.b1 = np.zeros((1, hidden_dim))

        self.W2 = np.random.randn(hidden_dim, output_dim) * np.sqrt(1. /
hidden_dim)

        self.b2 = np.zeros((1, output_dim))

        self.learning_rate = lr
```

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

    def sigmoid_grad(self, z):
        return z * (1 - z)

    def forward_propagation(self, X):
        # Compute hidden layer activation
        self.z1 = np.dot(X, self.W1) + self.b1
        self.a1 = self.sigmoid(self.z1)
        # Compute output layer activation
        self.z2 = np.dot(self.a1, self.W2) + self.b2
        self.a2 = self.sigmoid(self.z2)
        return self.a2

    def backward_propagation(self, X, y):
        m = y.shape[0]
        # Error in output layer
        output_error = self.a2 - y
        output_delta = output_error * self.sigmoid_grad(self.a2)
        # Error in hidden layer
        hidden_error = output_delta.dot(self.W2.T)
```

```
        hidden_delta = hidden_error * self.sigmoid_grad(self.a1)

        # Update weights and biases

        self.W2 -= self.a1.T.dot(output_delta) * self.learning_rate

        self.b2 -= np.sum(output_delta, axis=0, keepdims=True) *
self.learning_rate

        self.W1 -= X.T.dot(hidden_delta) * self.learning_rate

        self.b1 -= np.sum(hidden_delta, axis=0, keepdims=True) *
self.learning_rate



    def train(self, X, y, epochs=10000):

        for epoch in range(epochs):

            # Perform forward and backward passes

            self.forward_propagation(X)

            self.backward_propagation(X, y)

            if epoch % 1000 == 0:

                # Calculate mean squared error

                loss = np.mean((self.a2 - y) ** 2)

                print(f'Epoch {epoch}, Loss: {loss:.4f}')



    def predict(self, X):

        return self.forward_propagation(X)


if __name__ == "__main__":

    #input data
```

```
X = np.array([[0, 0],
              [0, 1],
              [1, 0],
              [1, 1]])
# Expected output
y = np.array([[0], [1], [0], [0]])


# Initialize the network
nn = NeuralNetwork(input_dim=2, hidden_dim=2, output_dim=1, lr=0.25)


# Training
nn.train(X, y, epochs=10000)


# Predictions after training
print("Predictions:")
predictions = nn.predict(X)
print(predictions)


# Binary predictions (threshold at 0.5)
binary_preds = (predictions > 0.5).astype(int)
print("Binary Predictions:")
print(binary_preds)
```

output:



Conclusion:.

Hence,we are able to execute and perform Error Back propagation algorithm.

Post Lab Descriptive Questions :

1. What is meant by local and global minima?

ANS:-

1. Local Minima:

a) A **local minimum** is a point on the error surface where the function value (error or loss) is lower than its neighboring points but might not be the lowest possible value.

b) At a local minimum, small changes in the input do not reduce the error, but it's possible that the error could be lower at some other point far away from the current location.
c) **Example**: Imagine a bumpy surface with several valleys. A local minimum is a valley that is not the deepest but is surrounded by higher terrain.
d) **Global Minima:**
e) A **global minimum** is the point on the error surface where the function value (error or loss) is the lowest possible. It represents the absolute minimum value that can be achieved by the cost function.
f) No other point on the surface has a lower value than the global minimum.
g) **Example**: On the same bumpy surface, the global minimum is the lowest valley, representing the point where the error is minimized the most.

2. What are factors that can improve the convergence of learning in Back propagation learning algorithm

ANS :-

a) **Learning Rate**: Choose an optimal rate or use adaptive methods (Adam, RMSprop).
b) **Momentum**: Helps prevent oscillations and accelerates learning.
c) **Weight Initialization**: Use Xavier or He initialization for balanced start.
d) **Batch Size**: Mini-batch gradient descent balances speed and accuracy.
e) **Normalization**: Use feature scaling or batch normalization for faster, stable training.
f) **Regularization**: Apply L2 regularization or dropout to prevent overfitting.
g) **Gradient Clipping**: Prevents exploding gradients for stability.
h) **Advanced Optimizers**: Use Adam, RMSprop for adaptive learning rates.
i) **Activation Functions**: ReLU or Leaky ReLU reduce vanishing gradient problems.
j) **Early Stopping**: Stops training when validation performance degrades.
k) **Shuffle Data**: Avoids model bias and helps escape local minima.
l) **Cross-Validation**: Helps tune hyperparameters for faster convergence.