Experiment No. 3

Title - Implement Back-propagation Algorithm

Theory -

Back-propagation is a fundamental concept in training artificial neural networks (ANNs). It's an algorithm that allows the network to learn from its mistakes by adjusting its weights and biases during the training process. Here's a breakdown of how back-propagation works:

- **Forward Propagation**

1. Initialisation: Initialise the weights and biases of the neural network randomly.
2. Forward Pass: During each iteration (epoch) of training, the input data is fed forward through the network.

- Input Layer: Passes the input features through the network.
- Hidden Layers: Each hidden layer calculates a weighted sum of the inputs and applies an activation function (like the sigmoid function) to produce an output.
- Output Layer: Similar to the hidden layers, the output layer calculates a weighted sum and applies an activation function to produce the predicted output.

- **Calculating Error**

3. Compute Error: Compare the predicted output of the network with the actual output (ground truth) to determine the error.

- The error is computed using a chosen error function, often the mean squared error (MSE) or cross-entropy loss.

- **Backward Propagation**

4. Back-propagation: The error is propagated backward through the network to update the weights and biases, minimising the error gradually.

- Output Layer Error: Calculate the error in the output layer by finding the difference between the predicted output and the actual output.
- Output Layer Gradient Descent: Compute the gradient of the error with respect to the weights and biases of the output layer.
- Hidden Layer Error: Propagate this error backward through the network to adjust the weights and biases of the hidden layers.
- Update Weights: Adjust the weights and biases by an amount proportional to the gradient and a predefined learning rate. This step involves minimising the error by descending the gradient in the direction that reduces the error.

- **Repeat**

5. Repeat: Update the weights and biases iteratively, using the updated values to perform another forward and backward pass through the network.

- **Convergence**

6. Stopping Criteria: The training process continues for a defined number of epochs or until a stopping criterion (e.g., a minimum error threshold) is met.

This process continues until the network's performance reaches an acceptable level or until it converges to a point where further training does not significantly improve performance.

■ Mathematical Operations

- **Forward Propagation**

Weighted Sum at Each Neuron:

$$z = \sum_{I=1}^{n} W_i * X_i + b$$

W_i = weights; X_i = inputs; b = bias

Activation Function:

$$a = \sigma(z) = \frac{1}{1 + e^{-z}}$$

$\sigma'(z) = z * (1 - z)$

a = output of the neuron; σ = sigmoid activation; σ'= derivative of sigmoid

- **Back-propagation**

Output Layer Error

$\delta_{output} = (y_{true} - y_{predicted}) * \sigma' (z_{output})$

Hidden Layer Error

$\delta_{hidden} = \delta_{output} * W_{output}^{T} * \sigma' (z_{hidden})$

Weight and Bias Update

$\Delta W = \alpha * \delta * X$

α = Learning rate ; δ = Error; ΔW = change in weights

- **Example Implementation**

In the context of neural networks and back-propagation, let's explore using a neural network to predict the output of a logical OR gate. In the case of logical gates like the OR gate, they take in binary inputs (0 or 1) and produce an output based on specific rules. The OR gate outputs 1 if at least one of its inputs is 1, otherwise, it outputs 0. Using a neural network with back-propagation, we can create a model that, given pairs of binary input values, learns to predict the corresponding OR gate output. The neural network will be trained to approximate the OR gate's behaviour by adjusting its weights and biases during the training process. The goal is to train the neural network so that when presented with different combinations of input values (0 and 1), it can correctly predict the OR gate's output based on the learned patterns in the training data. This application serves as a basic example demonstrating how neural networks, trained using back-propagation, can learn and mimic logical operations like the OR gate, showcasing their ability to approximate and generalise patterns in data.

- **Implementation**

```
import numpy as np

# Define the sigmoid activation function and its derivative
def sigmoid(x):
   return 1 / (1 + np.exp(-x))

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

# Set the input data
X = np.array([[0, 0],[0, 1],[1, 0],[1, 1]])

# Set the true outputs for OR GATE
y_true = np.array([[0],[1],[1],[1]])

# Initialize weights and biases randomly
input_size = 2
hidden_size = 4
output_size = 1
```

```
hidden_weights = np.random.uniform(size=(input_size, hidden_size))
hidden_bias = np.random.uniform(size=(1, hidden_size))
output_weights = np.random.uniform(size=(hidden_size, output_size))
output_bias = np.random.uniform(size=(1, output_size))

learning_rate = 0.1
epochs = 10000

# Training the network
for epoch in range(epochs):
    # Forward propagation
    hidden_layer_input = np.dot(X, hidden_weights) + hidden_bias
    hidden_layer_output = sigmoid(hidden_layer_input)

    output_layer_input = np.dot(hidden_layer_output, output_weights) + output_bias
    predicted_output = sigmoid(output_layer_input)

    # Backpropagation
    output_error = y_true - predicted_output
    output_delta = output_error * sigmoid_derivative(predicted_output)

    hidden_error = output_delta.dot(output_weights.T)
    hidden_delta = hidden_error * sigmoid_derivative(hidden_layer_output)

    # Updating weights and biases
    output_weights += learning_rate * hidden_layer_output.T.dot(output_delta)
    output_bias += learning_rate * np.sum(output_delta, axis=0, keepdims=True)
    hidden_weights += learning_rate * X.T.dot(hidden_delta)
    hidden_bias += learning_rate * np.sum(hidden_delta, axis=0, keepdims=True)

# Testing the trained network
hidden_layer = sigmoid(np.dot(X, hidden_weights) + hidden_bias)
output_layer = sigmoid(np.dot(hidden_layer, output_weights) + output_bias)
rounded_output = np.round(output_layer)
rounded_output = (output_layer > 0.9).astype(int)

print("OR Gate Output:")
print(rounded_output)
```

- Output

```
OR Gate Output:
[[0]
 [1]
 [1]
 [1]]
```

Conclusion

Implementing back-propagation is necessary for the successful training of neural networks, as it enables them to learn from data by iteratively adjusting their parameters to minimise prediction errors.