Name: Subrato Tapaswi	**Class/Roll No.: D16AD/60**	**Grade:**

Title of Experiment: Backpropagation on a DNN.

Objective of Experiment: The objective of backpropagation in a Deep Neural Network (DNN) with two hidden layers is to adjust the network's weights effectively. It does this by using calculated gradients. Backpropagation takes the error from the output layer and sends it backward to the hidden layers. This fine-tuning process makes the model better at reducing the gap between what it predicts and what's actually true. It helps the network become more skilled at understanding and representing intricate patterns in the data. This leads to higher accuracy and better performance on new, unseen data.

Outcome of Experiment: Backpropagation in a Deep Neural Network with two hidden layers adjusts the network's weights to make its predictions closer to the actual results. This makes the model more accurate, helps it understand complicated patterns in data, and makes it better at handling new situations. It's like giving the network a tool to learn and make better decisions.

Problem Statement: Implement a backpropagation algorithm to train a DNN with at least 2 hidden layers.

Description / Theory:

Backpropagation:

Backpropagation is a crucial part of training neural networks. It's like adjusting the knobs on a machine based on the mistakes it made in the last round. This fine-tuning process helps lower errors and makes the model better at handling new situations, increasing its reliability.

In simple terms, backpropagation is short for "backward propagation of errors." It's a standard way to teach neural networks. It figures out how much each knob (weight) affects the overall performance by calculating the gradients (slopes) of a mathematical function that measures how wrong the model's predictions are.

***Working*:**

The Backpropagation algorithm in a neural network calculates how much each weight affects the overall error using the chain rule. It's more efficient than calculating everything at once. Backpropagation finds the gradients but doesn't tell us how to use them. It's a generalization of the delta rule.

To train a neural network, we have two phases:

1. Forward Pass: We start by sending our data through the network from the input layer to the output layer. This helps us make predictions and measure how wrong they are. The difference between the predictions and the correct answers is our network error. This whole process is called forward propagation.

2. Backward Pass: In this phase, we reverse the flow. We begin by passing the error backward from the output layer to the input layer, going through any hidden layers in between. This is why it's called backpropagation. The backpropagation algorithm is a series of steps that use the error to adjust the network's weights and reduce the error.

Algorithm/ Pseudo Code / Flowchart (whichever is applicable)

Inputs X, arrive through the preconnected path
Input is modeled using real weights W. The weights are usually randomly selected.
Calculate the output for every neuron from the input layer, to the hidden layers, to the output layer.
Calculate the error in the outputs

Error = Actual Output – Desired Output

Travel back from the output layer to the hidden layer to adjust the weights such that the error is decreased.

Program :

1. **Programs on Basic programming constructs like branching and looping.**

```
def forward(self, X):
    self.hidden1_output = sigmoid(np.dot(X, self.weights_input_hidden1) + self.bias_hidden1)
    self.hidden2_output = sigmoid(np.dot(self.hidden1_output, self.weights_hidden1_hidden2) + self.bias_hidden2)
    self.output = sigmoid(np.dot(self.hidden2_output, self.weights_hidden2_output) + self.bias_output)
    return self.output

def backward(self, X, y, output):
    error = y - output
    d_output = error * sigmoid_derivative(output)

    error_hidden2 = d_output.dot(self.weights_hidden2_output.T)
    d_hidden2 = error_hidden2 * sigmoid_derivative(self.hidden2_output)

    error_hidden1 = d_hidden2.dot(self.weights_hidden1_hidden2.T)
    d_hidden1 = error_hidden1 * sigmoid_derivative(self.hidden1_output)

    # Update weights and biases
    self.weights_hidden2_output += self.hidden2_output.T.dot(d_output)
    self.bias_output += np.sum(d_output, axis=0, keepdims=True)

    self.weights_hidden1_hidden2 += self.hidden1_output.T.dot(d_hidden2)
    self.bias_hidden2 += np.sum(d_hidden2, axis=0, keepdims=True)

    self.weights_input_hidden1 += X.T.dot(d_hidden1)
    self.bias_hidden1 += np.sum(d_hidden1, axis=0, keepdims=True)
```

Output Screenshots :

```
# Test the trained model
test_output = model.forward(X)
print("Test Output:")
print(test_output)
```

```
Epoch 0, Loss: 0.3620
Epoch 100, Loss: 0.2486
Epoch 200, Loss: 0.2473
Epoch 300, Loss: 0.2439
Epoch 400, Loss: 0.2331
Epoch 500, Loss: 0.2036
Epoch 600, Loss: 0.1750
Epoch 700, Loss: 0.0717
Epoch 800, Loss: 0.0126
Epoch 900, Loss: 0.0057
Test Output:
[[0.049074  ]
 [0.94135678]
 [0.94127049]
 [0.06946351]]
```

Results and Discussions: Backpropagation is like a math wizard in the background, using techniques like gradients and the chain rule.

In summary, when you run backpropagation in a Deep Neural Network with two hidden layers, it's like fine-tuning the model's inner workings to make it better at predicting. This makes the network really good at understanding complex data patterns, leading to higher accuracy and better performance on different tasks. It's like giving the network superpowers to handle various challenges.