**Name: Abhinav Pallayil**
**UID:2019120045**

## Experiment No 4

## Back Propagation Algorithm

| | |
|---|---|
| **Objective:** | To explore the multi-layer perceptron algorithm using back-propagation. |
| **Outcomes:** | 1) Identifying the algorithms for non-linear classification.<br>2) Build the Multi-layer perceptron ML Model for the given data.<br>3) Draw various plots and interpret them.<br>4) Learn to use in built functions from libraries for training the NN. |
| **System Requirements:** | Linux OS with Python and libraries or R or windows with MATLAB. |
| **Theory:** | **How does backpropagation work?**<br>Back-propagation is just a way of propagating the total loss back into the neural network to know how much of the loss every node is responsible for, and subsequently updating the weights in such a way that minimizes the loss by giving the nodes with higher error rates lower weights and vice versa. Back-propagation works by doing the delta calculation step at every unit, back-propagating the loss into the neural net, and finding out what loss every node/unit is responsible for.<br>**Loss function?**<br>In fitting a neural network, backpropagation computes the gradient of the loss function with respect to the weights of the network for a single input–output example, and does so efficiently, unlike a naive direct computation of the gradient with respect to each weight individually. This efficiency makes it feasible to use |

| | |
|---|---|
| | gradient methods for training multilayer networks, updating weights to minimize loss; gradient descent, or variants such as stochastic gradient descent, are commonly used.<br><br>**Why do we need backpropagation?**<br><br>Backpropagation, or backward propagation of errors, is an algorithm that is designed to test for errors working back from output nodes to input nodes. It is an important mathematical tool for improving the accuracy of predictions in data mining and machine learning. Essentially, backpropagation is an algorithm needed to calculate derivatives quickly. |
| **Dataset Numerical:** | Input data is the truth table of AND Logic Gate & the network is two inputs layers, two hidden layers & two output layers network. |
| **Code:** | (see below) |

```
import numpy as np
def sigmoid (x):
  return 1/(1 + np.exp(-x))
def sigmoid_derivative(x):
  return x * (1 - x)

#Input datasets
inputs = np.array([[0,0],[0,1],[1,0],[1,1]])
expected_output = np.array([[0,0],[0,0],[0,0],[1,1]])
epochs = int(input("Enter your number of epochs: ")) #50000
lr = float(input("Enter your learning rate: ")) #0.2
inputLayerNeurons, hiddenLayerNeurons, outputLayerNeurons = 2,2,2

#Random weights and bias initialization
w1 = float(input("Enter your w1 weight: ")) #0.15
w2 = float(input("Enter your w2 weight: ")) #0.20
w3 = float(input("Enter your w3 weight: ")) #0.25
w4 = float(input("Enter your w4 weight: ")) #0.30
w5 = float(input("Enter your w5 weight: ")) #0.40
w6 = float(input("Enter your w6 weight: ")) #0.45
w7 = float(input("Enter your w7 weight: ")) #0.50
w8 = float(input("Enter your w8 weight: ")) #0.55
b1 = float(input("Enter your b1 weight: ")) #0.35
b2 = float(input("Enter your b2 weight: ")) #0.6
```

```
b3 = float(input("Enter your b3 weight: ")) #0.05
b4 = float(input("Enter your b4 weight: ")) #0.1

hidden_weights = np.array([[w1,w2],[w3,w4]])
hidden_bias =np.array([[b3,b4]])
output_weights = np.array([[w5,w6],[w7,w8]])
output_bias = np.array([[b1,b2]])

print("\nInitial hidden weights: ",end='')
print(*hidden_weights)
print("Initial hidden biases: ",end='')
print(*hidden_bias)
print("Initial output weights: ",end='')
print(*output_weights)
print("Initial output biases: ",end='')
print(*output_bias)

#Training algorithm
for _ in range(epochs):
  #Forward Propagation
  hidden_layer_activation = np.dot(inputs,hidden_weights)
  hidden_layer_activation += hidden_bias
  hidden_layer_output = sigmoid(hidden_layer_activation)
  output_layer_activation = np.dot(hidden_layer_output,output_weights)
  output_layer_activation += output_bias
  predicted_output = sigmoid(output_layer_activation)

  #Backpropagation
  error = expected_output - predicted_output
  d_predicted_output = error * sigmoid_derivative(predicted_output)
  error_hidden_layer = d_predicted_output.dot(output_weights.T)
  d_hidden_layer = error_hidden_layer * sigmoid_derivative(hidden_layer_output)

  #Updating Weights and Biases
  output_weights += hidden_layer_output.T.dot(d_predicted_output) * lr
  output_bias += np.sum(d_predicted_output,axis=0,keepdims=True) * lr
  hidden_weights += inputs.T.dot(d_hidden_layer) * lr
  hidden_bias += np.sum(d_hidden_layer,axis=0,keepdims=True) * lr

print("\nFinal hidden weights: ",end='')
print(*hidden_weights)
```

```
print("Final hidden bias: ",end='')
print(*hidden_bias)
print("Final output weights: ",end='')
print(*output_weights)
print("Final output bias: ",end='')
print(*output_bias)
print(f"\nOutput from neural network after {epochs} epochs: ",end='')
print(*predicted_output)
print(f"Error from neural network after {epochs} epochs: ",end='')
print(*error)
```

**Output:**

```
import numpy as np
def sigmoid (x):
  return 1/(1 + np.exp(-x))
def sigmoid_derivative(x):
  return x * (1 - x)
```

```
#Input datasets
inputs = np.array([[0,0],[0,1],[1,0],[1,1]])
expected_output = np.array([[0,0],[0,0],[0,0],[1,1]])
epochs = int(input("Enter your number of epochs: ")) #50000
lr = float(input("Enter your learning rate: ")) #0.2
inputLayerNeurons, hiddenLayerNeurons, outputLayerNeurons = 2,2,2
```

```
Enter your number of epochs: 50000
Enter your learning rate: 0.2
```

```
#Random weights and bias initialization
w1 = float(input("Enter your w1 weight: "))#0.15
w2 = float(input("Enter your w2 weight: "))#0.20
w3 = float(input("Enter your w3 weight: "))#0.25
w4 = float(input("Enter your w4 weight: "))#0.30
w5 = float(input("Enter your w5 weight: "))#0.40
w6 = float(input("Enter your w6 weight: "))#0.45
w7 = float(input("Enter your w7 weight: "))#0.50
w8 = float(input("Enter your w8 weight: "))#0.55
b1 = float(input("Enter your b1 weight: "))#0.35
b2 = float(input("Enter your b2 weight: "))#0.6
b3 = float(input("Enter your b3 weight: "))#0.05
b4 = float(input("Enter your b4 weight: "))#0.1

hidden_weights = np.array([[w1,w2],[w3,w4]])
hidden_bias =np.array([[b3,b4]])
output_weights = np.array([[w5,w6],[w7,w8]])
output_bias = np.array([[b1,b2]])
```

```
Enter your w1 weight: 0.15
Enter your w2 weight: 0.20
Enter your w3 weight: 0.25
Enter your w4 weight: 0.30
Enter your w5 weight: 0.40
Enter your w6 weight: 0.45
Enter your w7 weight: 0.50
Enter your w8 weight: 0.55
Enter your b1 weight: 0.35
Enter your b2 weight: 0.6
Enter your b3 weight: 0.05
Enter your b4 weight: 0.1
```

```
print("\nInitial hidden weights: ",end='')
print(*hidden_weights)
print("Initial hidden biases: ",end='')
print(*hidden_bias)
print("Initial output weights: ",end='')
print(*output_weights)
print("Initial output biases: ",end='')
print(*output_bias)
```

```
Initial hidden weights: [0.15 0.2 ] [0.25 0.3 ]
Initial hidden biases: [0.05 0.1 ]
Initial output weights: [0.4  0.45] [0.5  0.55]
Initial output biases: [0.35 0.6 ]
```

```python
    #Training algorithm
    for _ in range(epochs):
      #Forward Propagation
      hidden_layer_activation = np.dot(inputs,hidden_weights)
      hidden_layer_activation += hidden_bias
      hidden_layer_output = sigmoid(hidden_layer_activation)
      output_layer_activation = np.dot(hidden_layer_output,output_weights)
      output_layer_activation += output_bias
      predicted_output = sigmoid(output_layer_activation)

      #Backpropagation
      error = expected_output - predicted_output
      d_predicted_output = error * sigmoid_derivative(predicted_output)
      error_hidden_layer = d_predicted_output.dot(output_weights.T)
      d_hidden_layer = error_hidden_layer * sigmoid_derivative(hidden_layer_output)

      #Updating Weights and Biases
      output_weights += hidden_layer_output.T.dot(d_predicted_output) * lr
      output_bias += np.sum(d_predicted_output,axis=0,keepdims=True) * lr
      hidden_weights += inputs.T.dot(d_hidden_layer) * lr
      hidden_bias += np.sum(d_hidden_layer,axis=0,keepdims=True) * lr
```

```python
print("\nFinal hidden weights: ",end='')
print(*hidden_weights)
print("Final hidden bias: ",end='')
print(*hidden_bias)
print("Final output weights: ",end='')
print(*output_weights)
print("Final output bias: ",end='')
print(*output_bias)
print(f"\nOutput from neural network after {epochs} epochs: ",end='')
print(*predicted_output)
print(f"Error from neural network after {epochs} epochs: ",end='')
print(*error)
```

```
Final hidden weights: [-3.99866756  3.73808099] [-3.98790167  3.74829186]
Final hidden bias: [ 5.7242418  -5.33843355]
Final output weights: [-6.80112327 -6.85500816] [6.60702122 6.55033182]
Final output bias: [-0.49136222 -0.43575545]

Output from neural network after 50000 epochs: [0.00071769 0.00071886] [0.00573839 0.00573948] [0.00573782 0.0057398 ] [0.99170497 0.99170301]
Error from neural network after 50000 epochs: [-0.00071769 -0.00071886] [-0.00573839 -0.00573948] [-0.00573782 -0.0057398 ] [0.00829503 0.00829699]
```

**Interpretation:**

My interpretation for the above experiment is as follows:
1) As the number of epochs increases, the error reduces and came to near zero.
2) Hence, the output is closer to the target.

**Conclusion:**

By doing this experiment, I got to know about the following things:
1) I learned the implementation of back propagation algorithm.
2) As the number of epochs increases, the error reduces and came to near zero. Therefore, the output is closer to the target.