Aim: Implement Artificial Neural Network training process in Python using Forward Propagation and Back Propagation.

Software for Python: Jupyter Notebook

Theory:

Forward Propagation

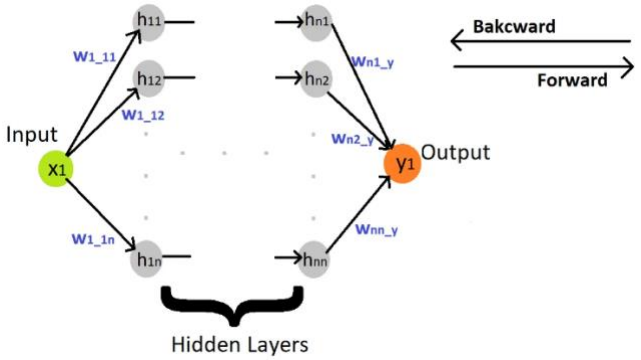

Fig.1-Forward feed in an example neural network structure

In the Neural Network, our journey starting from the input to the output is called the forward direction. The weights entering each node are multiplied with the available value (if input is x feature value, if hidden layer is the value from the sum of previous multiplications entering that node) and bias is added. This multiplication operation is called the "dot product", and that's because we're dealing with vectors.

$$z = W^1.x + b_1$$

$$h = \emptyset(z)$$

$$o = W^2.h + b_2$$

Now we can move on to our numerical example. Let's go through a simple example and focus on the operations rather than the complexity of the model. There are two input neurons (node), two hidden neurons, and two output neurons. In addition to these, the hidden layer and the output layer contain two biases. Let's use the "sigmoid" activation function in the hidden layer.

i is abbreviated to represent the input layer
h is abbreviated to represent the hidden layer
o is abbreviated to represent the output layer.

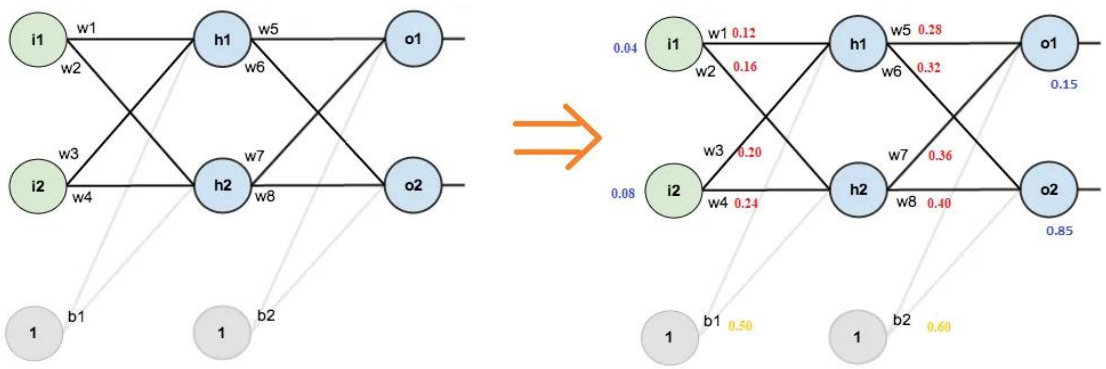

Fig.2-Example with parameter values

Let's calculate h1 first. While doing this, it is seen that there are i1, i2 and b1 nodes entering the h1 node. Therefore:

$$h_1 = w_1 * i_1 + w_3 * i_2 + b_1 * 1 = 0.12 * 0.04 + 0.2 * 0.08 + 0.50 * 1 = 0.5208$$

$$out_{h_1} = \frac{1}{1 + e^{-h_1}} = \frac{1}{1 + e^{-0.5208}} = 0.6273$$

Then let's find h2:

$$h_2 = (w_2 * i_1) + (w_4 * i_2) + (b_1 * 1) = 0.16 * 0.05 + 0.24 * 0.1 + 0.5 * 1 = 0.5256$$

$$out_{h_2} = \frac{1}{1+e^{-h_2}} = \frac{1}{1+e^{-0.584}} = 0.6284$$

Now we have calculated our hidden layers. The queue is in the output layer. Let's start with o1 first:

$$o_1 = (w_5 * out_{h_1}) + (w_7 * out_{h_2}) + (b_2 * 1) =$$

$$(0.28 * 0.6273) + (0.36 * 0.6304) + 0.6 * 1 = 0.9350$$

$$out_{o_1} = \frac{1}{1+e^{-o_1}} = \frac{1}{1+e^{-1}} = 0.7181$$

Finally, let's calculate o2:

$$o_2 = w_6 * out_{h_1} + w_8 * out_{h_2} + b_2 * 1 =$$

$$(0.32 * 0.6273) + (0.40 * 0.6284) + 0.6 * 1 = 0.9768$$

$$out_{o_2} = \frac{1}{1+e^{-o_2}} = \frac{1}{1+e^{-1.1084}} = 0.7265$$

Since we have calculated our outputs, it is time to find the loss function value. Let's say our target values are both 0.15:

$$E_{o_1} = \frac{1}{2}(\hat{y} - out_{o_1})^2 = \frac{1}{2}(0.15 - 0.7181)^2 = 0.1614$$

$$E_{o_1} = \frac{1}{2}(\hat{y} - out_{o_2})^2 = \frac{1}{2}(0.85 - 0.7265)^2 = 0.0076$$

$$E_{total} = E_{o_1} + E_{o_2} = 0.1688 + 0.0076 = 0.169$$

E_{total} is the total loss of forward feed.

Backward Propagation

This algorithm is called backpropagation because it tries to reduce errors from output to input. It looks for the minimum value of the error function in the weight field using a technique called gradient descent.

With this method, we will try to reduce the error by changing the weight and bias values. Since we will do a single step and show the weight update, let's take the learning rate as a big value like 0.5 to make a little bigger progress and start back propagation. Let's choose w5 as the node whose weight we want to update and let's do the operations on it. (All weights are updated with this method, but I chose w5 for illustration)

Backpropagation is the core of neural network training. It is a method of adjusting the weights of a neural network based on the loss value obtained in the previous epoch. Correctly adjusting the weights allows us to reduce the error rate and increase its generalization, making the model reliable.

Instead of going directly to the formula, let's look at the above network again. This will guide us in a way to implement Chain Rule. The value taken from Loss is given back as input and affects o1's sigmoid state, which in turn affects o1 before sigmoid and o1, w5 as the last link of the chain:

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o_1}} * \frac{\partial out_{o_1}}{\partial o_1} * \frac{\partial o_1}{\partial w_5}$$

Let's consider all three derivative operations above one by one. The formulas of these variables are the same as in forward propagation, but it will be enough to take the derivative and substitute it:

$$E_{total} = \frac{1}{2}.(y_{o1} - out_{o1})^2 + \frac{1}{2}.(y_{o2} - out_{o2})^2$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = 2.\left(\frac{1}{2}\right).(y_{o1} - out_{o1})^{2-1}.(-1) + 0 = -(y_{o1} - out_{o1}) = -(0.15 - 0.7181) = 0.5681$$

Since we took the derivative with respect to o1, the part with o2 behaved like a constant and was equal to 0.

$$o_1 = w5 * out_{h1} + w6 * out_{h2} + b_2$$

$$\frac{\partial o_1}{\partial w5} = 1 * out_{h1} * w5^{(1-1)} + 0 + 0 = out_{h1} = 0.6273$$

→ Again, the parts with o2 were equal to zero.

Now that we have the derivatives, we can apply the chain rule by multiplying them:

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o_1}} * \frac{\partial out_{o_1}}{\partial o_1} * \frac{\partial o_1}{\partial w_5}$$

$$\frac{\partial E_{total}}{\partial w_5} = -(y_{o1} - out_{o1}) * out_{o1}(1 - out_{o1}) * out_{h1} = 0.5681 * 0.2024 * 0.6273 = 0.0721$$

As a final step, we can now update the w5 weight. While doing this, we multiply the derivative we found from the previous weight with a fixed learning rate that we determine:

$$w_5^+ = w_5 - \eta.\left(\frac{\partial E_{total}}{\partial w5}\right) = 0.28 - 0.5.(0.0721) = 0.24395$$

Updated w5 weight

Procedure:

This code defines a simple neural network using the sigmoid activation function and backpropagation for training.

Here's how it works:

First, the code defines the sigmoid activation function and its derivative using the numpy library. The sigmoid function maps any input to a value between 0 and 1, which is useful for creating a binary output for classification problems.

Then, the code defines a NeuralNetwork class. The initialization method takes the input and output data as arguments, and initializes the weights randomly. The feedforward method computes the output of the network given the input and current weights, while the backprop method updates the weights based on the difference between the predicted output and the actual output.

Next, the code defines the training data as X (input) and y (output). X is a 4x3 matrix representing four training examples, each with three input features. y is a 4x1 matrix representing the target output for each training example.

Finally, the code trains the neural network using a for loop that calls the feedforward and backprop methods repeatedly for a fixed number of iterations (in this case, 1500). After training, the code prints the final output of the network.

Note that this code implements a very simple neural network with only one hidden layer and four hidden units. For more complex problems, you may need to adjust the number of layers and units, as well as the activation function and optimization algorithm.

Conclusion: Thus, we have studied how to implement Artificial Neural Network training process in Python using Forward Propagation and Back Propagation.

Program Code and Output

In [6]:
```
import numpy as np

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

# Define the derivative of sigmoid function
def sigmoid_derivative(x):
    return x * (1 - x)

# Define the neural network class
class NeuralNetwork:

    def __init__(self, x, y):
        self.input = x
        self.weights1 = np.random.rand(self.input.shape[1],4)
        self.weights2 = np.random.rand(4,1)
        self.y = y
        self.output = np.zeros(self.y.shape)

    def feedforward(self):
        self.layer1 = sigmoid(np.dot(self.input, self.weights1))
        self.output = sigmoid(np.dot(self.layer1, self.weights2))

    def backprop(self):
        d_weights2 = np.dot(self.layer1.T, (2*(self.y - self.output) * sigmoid_derivative(self.output)))
        d_weights1 = np.dot(self.input.T,  (np.dot(2*(self.y - self.output) * sigmoid_derivative(self.output),
                                                   self.weights2.T) * sigmoid_derivative(self.layer1)))
        self.weights1 += d_weights1
        self.weights2 += d_weights2

# Define the training data
X = np.array([[0,0,1],[0,1,1],[1,0,1],[1,1,1]])
y = np.array([[0],[1],[1],[0]])

# Initialize the neural network
nn = NeuralNetwork(X,y)

# Train the neural network
for i in range(1500):
    nn.feedforward()
    nn.backprop()

# Print the output after training
print(nn.output)
```

```
print(nn.output)
```

```
[[0.01030782]
 [0.96569   ]
 [0.96580405]
 [0.04033994]]
```

In [ ]: