XOR Function

Ab Waheed Lone
Department of Computer Engineering
Yildiz Technical University
Student No: 18501053

17th October 2019

Another application of Perceptron Learning Algorithm is the Neural implementation of XOR logical function which gives an output 1 when both of the two inputs are different(either 0 or 1)

XOR Logical function		
X_1(First input feature value)	X_2(Second input feature value)	Y(Output activation)
0	0	0
0	1	1
1	0	1
1	1	0

Since XOR logical function is linearly inseparable, model cannot be implemented using single Neuron with logical 0 and 1 as its input values. Therefore more than one neuron are needed to classify the input values into one of the classes.

XOR Model has three different layers.

- Input Layer
- Middle Layer
- Output layer

1)**Input Layer**: Input layer consists of input feature vectors which involves combination of 0's and 1's usually represented by x_i, x_j, x_k etc. where i,j,k=1,2,....n

2) **Middle Layer**: The middle layer consists of several number of neurons for input processing (activation functions) but in case of XOR model, two neurons are used for input training activation. Inputs to both of the neurons are input features from input layer which are usually in terms of 0's and 1's. Single feature

input values are given to both of the neurons. For two neuron XOR model, 2×2 weight matrix is randomly initialized. Summation of the dot product between input values and weight vectors(including bias) is followed by the activation of hidden layer values. Depending on the output expectation and problem domain, different activation functions can be used.Here middle layer activation values are calculated using **Sigmoid** function [-1,1].
Neuron1

$$y1 = x_1 * w_1 + x_2 * w_2 + b$$

Neuron2

$$y2 = x_2 * w_3 + x_1 * w_4 + b$$

Activation1	**z1= Sigmoid(y1)**
Activation2	**z2=Sigmoid(y2)**

3) **Output Layer**
The XOR output layer repeating the same process of taking the summation of dot product between weight matrix(2×1) and activated output values from middle layer. The same activation function (**Sigmoid**) is used to calculate the output value. After this, the error function value is calculated which involves taking the difference between activated output value at output layer and predetermined expected value given during training process. Generally during training phase, the error function values are not equal to zero (observed - actual values), to minimize the value near to zero or equal to zero,back-propagation algorithm is used.
Back-propagation algorithm takes the partial derivatives of error function with respect to all variables including weight vectors through all branches of the neural net. Along every direction of the computational graph of network,partial derivatives are calculated which involves calculating local gradient and upstream gradient (Gradient which comes from nodes which are after the current node in a computational graph). Using the Gradient descent algorithm weights are updated continuously.

In case of logical XOR function,during Back phase (or gradient calculation) partial derivative of **Sigmoid** function is used like we used **Sigmoid** in forward propagation phase.At each node of every layer, Weight adjustments are calculated by doing the dot between training inputs and a term which involves multiplication between error function and derivative of activation function. Then using the gradient descent algorithm, weight updates are made.
In the test phase, the non-learned inputs are provided to the learned model to check the output through the model to get the expected output value near.

Python code for XOR Function

```
import numpy as np

def sigmoid(x):
    return 1/(1+np.exp(-x))


def sigmoid_der(x):
    return sigmoid(x)*(1-sigmoid(x))
class NeuralNet():

    def __init__(self,inputs):
        self.inputs=inputs
        self.l=len(self.inputs)
        self.li=len(self.inputs[0])
        print(self.li)
        print(self.l)
        self.wi=np.random.random((self.li,self.l))
        print(self.wi.shape)
        self.wh=np.random.random((self.l,1))
        print(self.wh.shape)

    def activation(self,inp):
        s1=sigmoid(np.dot(inp,self.wi))
        s2=sigmoid(np.dot(s1,self.wh))
        return s2
    def train(self,inputs,outputs,it):
        for i in range(it):
            l0=inputs
            l1=sigmoid(np.dot(l0,self.wi))
            l2=sigmoid(np.dot(l1,self.wh))

            l2_err=outputs-l2
            l2_delta=np.multiply(l2_err,sigmoid_der(l2))

            l1_err=np.dot(l2_delta,self.wh.T)
            l1_delta=np.multiply(l1_err,sigmoid_der(l1))

            self.wh+=np.dot(l1.T,l2_delta)
            self.wi+=np.dot(l0.T,l1_delta)


inputs=np.array([[0,0],[0,1],[1,0],[1,1]])
outputs=np.array([[0],[1],[1],[0]])
n=NeuralNet(inputs)
print("Before training: ")
print (n.activation(inputs))
n.train(inputs,outputs,10000)
print("after training: ")
print(n.activation(inputs))
```

Figure 1: Testing results