# XOR Gate- Neural Networks

Madhumitha S

May 6, 2020

## 1 Introduction

The aim of this project is to build a neural network capable of learning the logic behind the XOR gate and then producing the correct output when inputs are provided.
The XOR gate (pronounced as Exclusive OR) is a logic gate that gives a true output (1 or HIGH) when the number of true inputs is odd. Implements an exclusive or; that is, a true output results if one, and only one, of the inputs to the gate is true.
The truth table for the XOR gate, is as follows-

| A | B | A XOR B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

## 2 Need for Neural Network

Previously, we had worked on an algorithm for implementing a single perceptron. The question now is: **Why do we need a neural network to model the XOR gate?** Won't one perceptron do?
Trying with one perceptron-

```
import numpy as np

class Perceptron(object):        #creating a class

    #function to initialise parameters
    def __init__(self, no_of_inputs, epochs=2000,
        learning_rate=0.2):
        self.epochs = epochs
        self.learning_rate = learning_rate
        self.weights = np.zeros(no_of_inputs + 1)
```

```
    #Sigmoid function which is used as activation function
    def sigmoid(self, x):
        return 1/(1 + np.exp(-x))

    #Sigmoid derivative (gradient descent)
    def sigmoid_derivative(self, x):
        return x * (1-x)

    #prediction function while training
    def predict_train(self, inputs):
        z = np.dot(inputs, self.weights[1:]) + self.weights[0]
        activation = self.sigmoid(z)
        return activation

    #training the machine
    def train(self, X, y):
        for _ in range(self.epochs):
            prediction = self.predict_train(X)
            error = (y_train - prediction) *
                self.sigmoid_derivative(prediction)
            self.weights[1:] += self.learning_rate *
                np.dot(X_train.T, error)
            self.weights[0] += self.learning_rate *
                np.sum(y_train-prediction, axis=0, keepdims=True)

    #to predict output of user entered inputs
    def predict_test(self, inputs):
        probability = self.predict_train(inputs)
        if probability >= 0.5:
            output = 1
        else:
            output = 0
        print('Prediction- ',probability)
        print('Output- ', output)
```

---

```
#Running the code
X_train = np.array((0, 0, 0, 1, 1, 0, 1, 1)).reshape(4,2)
y_train = np.array([0, 1, 1, 0])

perceptron = Perceptron(2)
perceptron.train(X_train, y_train)

for i in range(4):
    inputs = []
    for i in range(2):
        inputs.append(int(input('Enter input {}: '.format(i+1))))
```

```
X_test = np.array(inputs)
perceptron.predict_test(X_test)
```

Output-
Enter input 1: 0
Enter input 2: 0
Prediction- 0.5
Output- 1
Enter input 1: 0
Enter input 2: 1
Prediction- 0.5
Output- 1
Enter input 1: 1
Enter input 2: 0
Prediction- 0.5
Output- 1
Enter input 1: 1
Enter input 2: 1
Prediction- 0.5
Output- 1

We notice that the machine predicts the output to be 1 in every case (the activation in each case in 0.5). This is not the case with XOR gates. Hence the need for a network with hidden layers.

# 3 Model

The neural network model used for the XOR gate is-

- 2 neurons in the input layer
- One hidden layer with 2 neurons
- Output layer with 1 neuron

Activation function used- sigmoid function.

$$\sigma = \frac{1}{1 + e^{-x}}$$

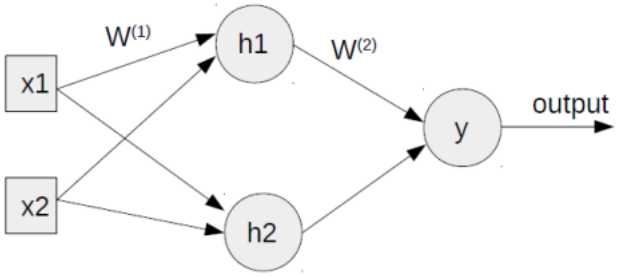


Figure 1: XOR gate

# 4 Code

We begin with initialising values for learning rate (step size while moving towards minimum of loss function), epochs (no.of cycles through the training set). Also, assigning random weights and biases to begin with.

1. Forward pass
2. Backpropagation, updating weights and biases

## 4.1 Forward Pass

It involves traversing through all neurons from first layer to the last, and calculating the output from the input.

Input layer to hidden layer-

$$\begin{bmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} W_{11}x_1 + W_{12}x_2 + b_1 \\ W_{21}x_1 + W_{22}x_2 + b_2 \end{bmatrix} \tag{1}$$

Activation-

$$\begin{bmatrix} \sigma(W_{11}x_1 + W_{12}x_2 + b_1) \\ \sigma(W_{21}x_1 + W_{22}x_2 + b_2) \end{bmatrix} = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} \tag{2}$$

Hidden layer to output layer-

$$\begin{bmatrix} W_1 & W_2 \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} + b = \begin{bmatrix} W_1 z_1 + w_2 x_2 + b \end{bmatrix} \tag{3}$$

Predicted output-

$$\begin{bmatrix} \sigma(W_1 z_1 + w_2 x_2 + b) \end{bmatrix} \tag{4}$$

```
hidden_layer_activation = np.dot(X_train, hidden_weights) +
    hidden_bias
hidden_layer_output = self.sigmoid(hidden_layer_activation)

output_layer_activation = np.dot(hidden_layer_output,
    output_weights) + output_bias
predicted_output = self.sigmoid(output_layer_activation)
```

## 4.2 Backpropagation

It involves traversing through the network from the last layer (output) to the first and calculating the loss/cost, using gradient descent algorithm to minimize loss and updating the weights and biases accordingly.
The idea is to find the combination of weights for which the loss is minimum.

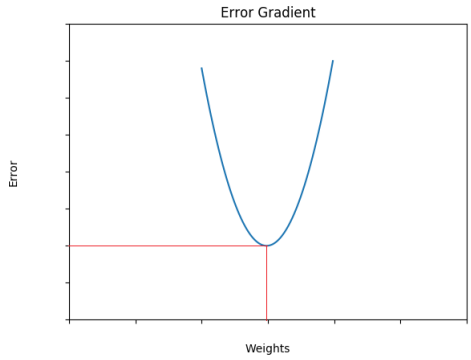


Figure 2: Loss vs Weights graph

```
 #Backpropagation
error = y_train - predicted_output
d_predicted_output = error *
    self.sigmoid_derivative(predicted_output)

error_hidden_layer = np.dot(d_predicted_output, output_weights.T)
d_hidden_layer = error_hidden_layer *
    self.sigmoid_derivative(hidden_layer_output)

#Updating weights and biases
output_weights += self.learning_rate *
    np.dot(hidden_layer_output.T, d_predicted_output)
output_bias += self.learning_rate * np.sum(d_predicted_output,
    axis=0,
keepdims=True)
```

```
 hidden_weights += self.learning_rate * np.dot(X_train.T,
     d_hidden_layer)
hidden_bias += self.learning_rate * np.sum(d_hidden_layer,
    axis=0,keepdims=True)
```

# 5 Final Code

```
import numpy as np


class Network(object):

    def __init__(self, no_of_inputs, epochs=10000,
        learning_rate=0.1):
        self.epochs = epochs
        self.learning_rate = learning_rate


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

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


    def train(self, X, y):
        #Initializing size of layers
        inputNeurons = 2
        hiddenNeurons = 2
        outputNeurons = 1

        #Random initialisation of weights and biases of hidden and
            output layer
        hidden_weights = np.random.uniform(size=(inputNeurons,
            hiddenNeurons))
        hidden_bias = np.random.uniform(size=(1, hiddenNeurons))

        output_weights = np.random.uniform(size=(hiddenNeurons,
            outputNeurons))
        output_bias = np.random.uniform(size=(1, outputNeurons))


        #Training model
        for i in range(self.epochs):
            #Forward pass
```

```
            hidden_layer_activation = np.dot(X_train,
                hidden_weights) + hidden_bias
            hidden_layer_output =
                self.sigmoid(hidden_layer_activation)

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

            #Backpropagation
            error = y_train - predicted_output
            d_predicted_output = error *
                self.sigmoid_derivative(predicted_output)

            error_hidden_layer = np.dot(d_predicted_output,
                output_weights.T)
            d_hidden_layer = error_hidden_layer *
                self.sigmoid_derivative(hidden_layer_output)

            #Updating weights and biases
            output_weights += self.learning_rate *
                np.dot(hidden_layer_output.T, d_predicted_output)
            output_bias += self.learning_rate *
                np.sum(d_predicted_output, axis=0, keepdims=True)

            hidden_weights += self.learning_rate * np.dot(X_train.T,
                d_hidden_layer)
            hidden_bias += self.learning_rate *
                np.sum(d_hidden_layer, axis=0, keepdims=True)

        return hidden_weights, hidden_bias, output_weights,
            output_bias

    def predict(self, inputs):
        hidden_weights, hidden_bias, output_weights, output_bias =
            self.train(X_train, y_train)

        hidden_layer_activation = np.dot(inputs, hidden_weights) +
            hidden_bias
        hidden_layer_output = self.sigmoid(hidden_layer_activation)

        output_layer_activation = np.dot(hidden_layer_output,
            output_weights) + output_bias
        predicted_output = self.sigmoid(output_layer_activation)
        if predicted_output >= 0.5:
            return 1
        else:
            return 0
```

## 6 Conclusion

This project helped gaining an understanding on how an Artificial Neural Network(ANN) works and emphasised the need for networks of perceptrons for deep learning in machines. Since we coded from scratch, without using any libraries (other than numpy), we also learned some of the math behind it, which definitely helped understanding Neural networks better.

## References


[1] 3 blue 1 brown Youtube Neural networks series- https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1$_6$7000$Dx_ZCJB-$ $3pi$

[2] https://towardsdatascience.com

[3] Wikipedia page for XOR gate- https://en.wikipedia.org/wiki/XOR$_gate$