XOR with Neural Networks : An overview

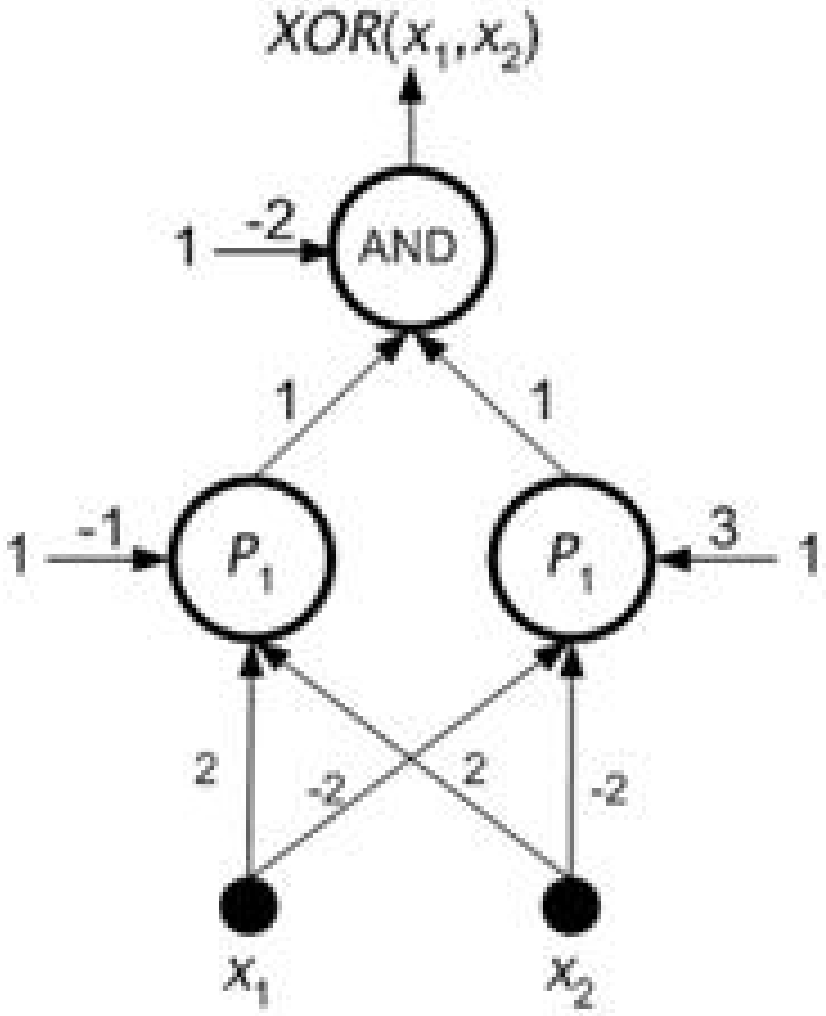
Tushar Nandy

The *XOR* Problem:

1. Input = $\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix}$
2. Output = $\begin{bmatrix} 1 & 1 & 0 & 0 \end{bmatrix}$
3. Percpetrons using **sigmoids** endup with: $\begin{bmatrix} 0.5 & 0.5 & 0.5 & 0.5 \end{bmatrix}$
4. Since I used a **tanh** activation function, my output was different.

Hidden Neurons to the rescue:

The architecture:



1. **Defining my activation funtion:**

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

```
def f(x):
    return np.tanh(x)

def grad(y):
    return 1 - y**2
```

2. **Defining my weights and biases:**

```
n_hn = 2
np.random.seed(9)
weights_1 = 2 * np.random.random((n_hn,2)) - 1
weights_2 = 2 * np.random.random((1,n_hn)) - 1

np.random.seed(54)
bias_1 = 2 * np.random.random((n_hn,1)) - 1
bias_2 = 2 * np.random.random((1,1)) - 1
```

3. **Forward propagation from input to hidden layer:**

$$\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} w_{11}x_1 + w_{12}x_2 \\ w_{21}x_1 + w_{22}x_2 \end{bmatrix} (+ \begin{bmatrix} b_1 \\ b_2 \end{bmatrix})$$

4. **Activation of Hidden Neurons:**

$$\begin{bmatrix} f(w_{11}x_1 + w_{12}x_2 + b_1) \\ f(w_{21}x_1 + w_{22}x_2 + b_2) \end{bmatrix} = \begin{bmatrix} Z_1 \\ Z_2 \end{bmatrix}$$

```
intermediate_layer = f(np.dot(weights_1, input) + bias_1)
```

6. **From hidden layer to output:**

$$\begin{bmatrix} w_{o1} & w_{o2} \end{bmatrix} \times \begin{bmatrix} Z_1 \\ Z_2 \end{bmatrix} = \begin{bmatrix} w_{01}Z_1 + w_{02}Z_2 \end{bmatrix} (+ \begin{bmatrix} b_0 \end{bmatrix})$$

7. **Final output after activation:**

$$\mathrm{Output} = \mathrm{f}(\mathrm{w}_{01}\mathrm{Z}_1 + \mathrm{w}_{02}\mathrm{Z}_2 + \mathrm{b}_0)$$

```
output = f(np.dot(weights_2, intermediate_layer) + bias_2)
```

8. **Ok, Now is the time for back propogation:**

   Don’t worry, won’t burden you with all those scary partial derivates to cross check (I know you’ll skip anyway XD) so here’s the snippet:

```
lr = 0.01

# error

E = expected_output - output

# updating the 2nd fold weights and bias
```

```
weights_2 += np.dot(E * grad(output), intermediate_layer.T) * lr
bias_2 += np.sum(E * grad(output), axis=1, keepdims=True) * lr

# updating the 1st fold weights and biases

grad_hidden = weights_2.T * grad(intermediate_layer)
weights_1 += np.dot(E * grad(output) * grad_hidden), input.T) * lr
bias_1 += np.sum(E * grad(output) * grad_hidden), axis=1, keepdims=True) * lr
```

9. **WTF is a learning rate, Tushar?**
   Imagine that our cost function is a baby, whom we want to guide towards a place called “the global minima”. Learning rates define the step-length of this kiddo. Ideally between 0.01 and 0.0001, a large learning rate will mean a faster stride accompanied with the risk of ‘overshooting’ and a low rate would mean slower and smaller steps which would require more epochs to reach. How did i choose mine? The ans is **hit-and-trial**.

Helpful Sources:

1. **3blue1brown: Neural Network Series**
   The coding train guy kept referring to 3b1b time and again. This made me ponder that if I give it some time (5 hrs. LOL. XD) I would be able to code on my own
2. **python Numpy docs: How to find sum of rows in a matrix**
   This is the stuff that I use in my biases
   ==========