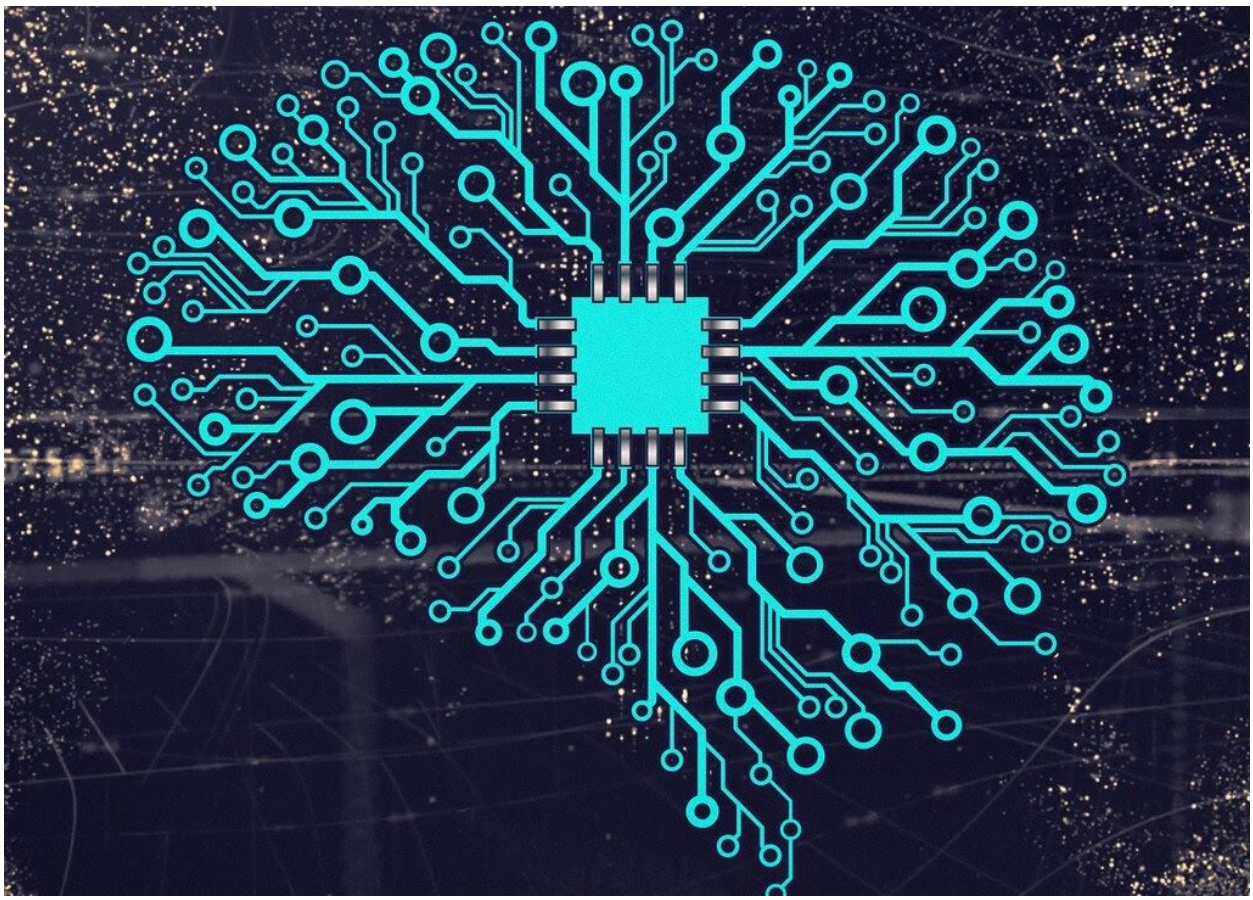
Neural Networks

Implementing a XOR gate

Report by: Divyanshi Kamra



Introduction

This is a report about how to implement a XOR gate while working with neural networks.This report assumes that you know how to implement a perceptron.

First let's know what is a XOR gate,

XOR (exclusive OR gate):

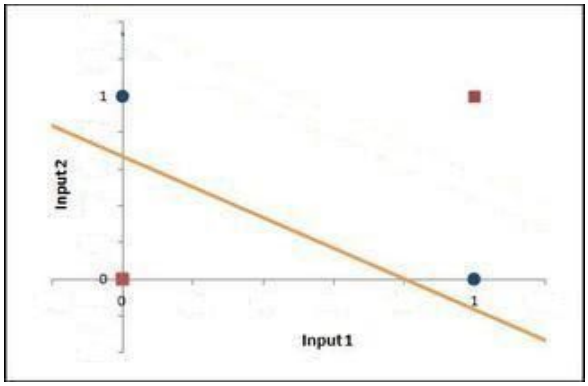
An XOR (exclusive OR gate) is a digital logic gate that gives a true output only when both its inputs differ from each other. The truth table for an XOR gate is shown below:

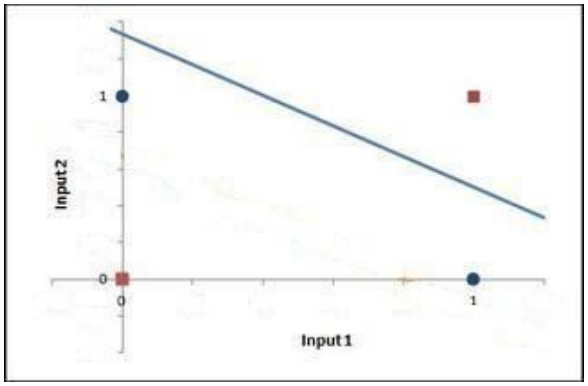
X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	0

Truth Table for XOR

Using neural networks to implement logic gates:

The goal of the neural network is to classify the input patterns according to the above truth table. A perceptron can classify inputs which are linearly separable using a line/hyperplane. Let's plot the input of the XOR gate and try to classify it.

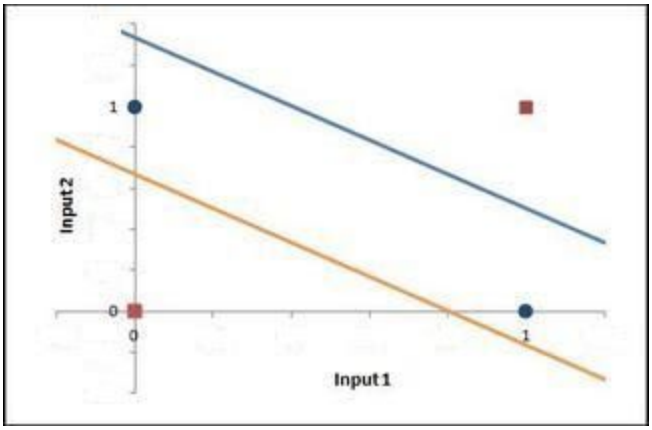




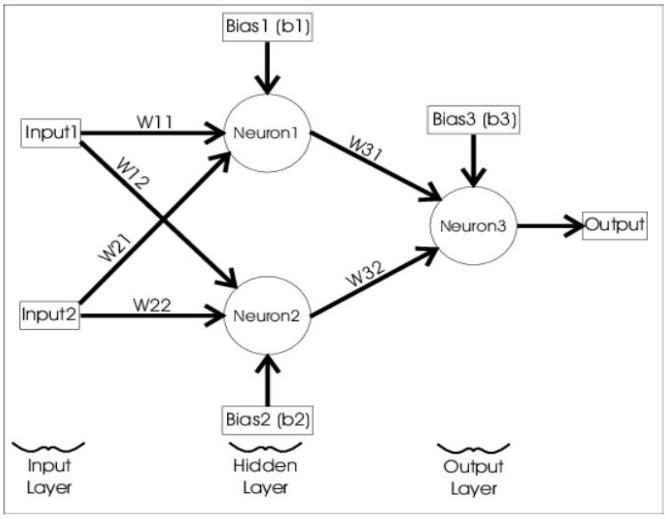
From the above images, we can see that the input of a XOR gate is not linearly separable. Thus, we need to improve our model.

Improvising our model:

We visualize that a possible solution to separate the XOR inputs is

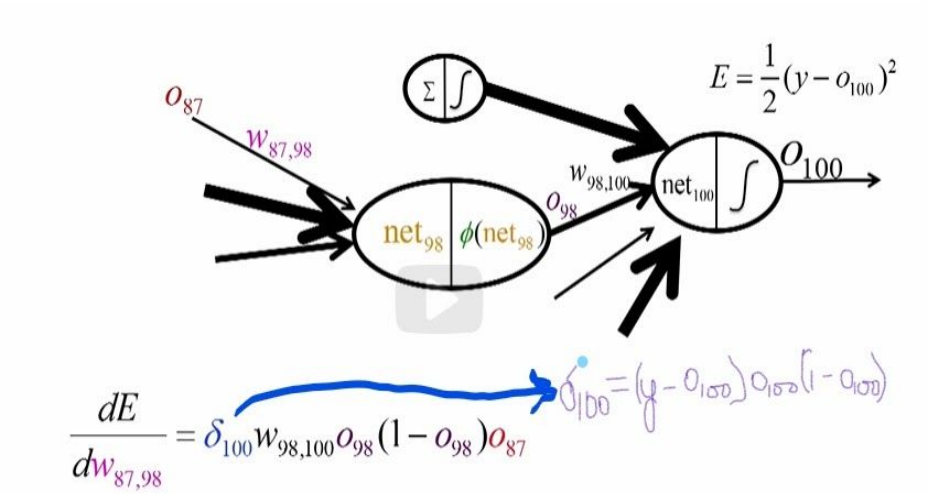


Thus we can make two hyperplanes and take their intersection to classify our inputs correctly. Thus, we use **Hidden Neurons** which take the output of input neurons as neuron1 and neuron2 and run the calculations again and decide final input by taking the intersection of output of both neuron1 and neuron2.



The Neural Network Model to solve the XOR Logic (from: https://stopsmokingaids.me/)

ERROR CALCULATION:



Points to be implemented in code:

1. Take the XOR inputs as training data and make activation functions..

```
#taking inputs of XOR gate as training data
X_train=np.array([[0,0],[0,1],[1,0],[1,1]])
# .T is to take transpose of the matrix
y_train=np.array([0,1,1,0]).T
y_train=y_train.reshape(4,1)
```

```
def func(t):
    return 1/(1+np.exp(-t))

def derivative(t):
    return t*(1-t)
```

2. Initialize the weights randomly.

```
def initialize(input_neurons=2,hidden_neurons=2,output_neurons=1):
    np.random.seed(0)
    weights_inh=np.random.random((input_neurons,hidden_neurons))
    weights_hio=np.random.random((hidden_neurons,output_neurons))
    return weights_inh,weights_hio

weights_ih,weights_ho=initialize(2,2,1)
```

3. Propagate the data forward by taking dot product of weights and the values and calculate final output.

```
#forward propagation
hidden_input=func(np.dot(X_train,weights_ih))
predicted_output=func(np.dot(hidden_input,weights_ho))
```

4. Calculate error and update the weights using gradient descent.

```
#error calculation
abs_error= np.subtract(y_train,predicted_output)
hidden_layer_error= derivative(predicted_output)*(abs_error)
d_hidden= np.dot(predicted_output,weights_ho.T)
input_layer_error= derivative(hidden_input)*d_hidden
```

```
#updating weights
lr=0.5
weights_ih += np.dot(X_train.T,input_layer_error)*lr
weights_ho += np.dot(hidden_input.T,hidden_layer_error)*lr
```

5. Propagate forward,backward, forward,... continue doing this until the error is minimal.

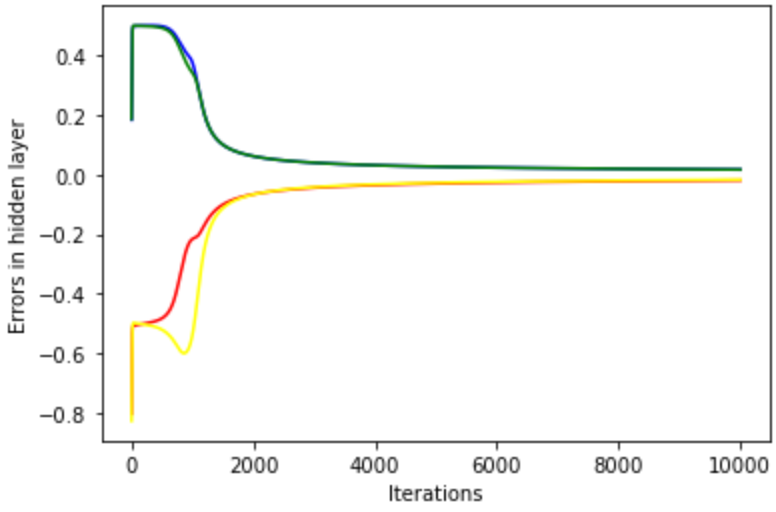
At this point, after implementing the code I realised that I need to use bias for better results; rather the number of iterations need to be increased since the code takes a very long time to learn. So i added it in the final code.

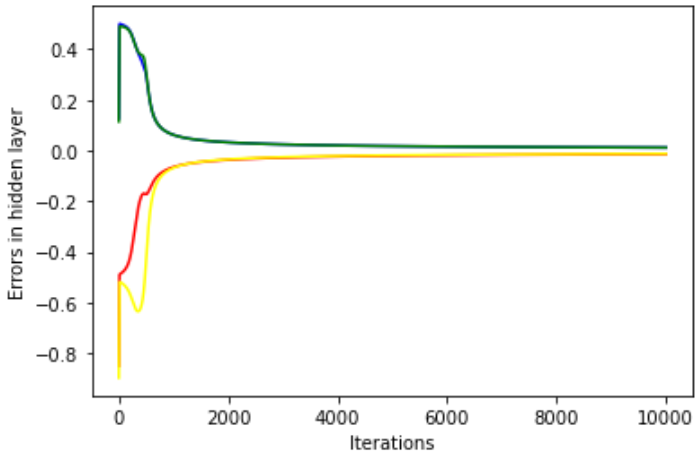
Complete code:

https://github.com/divyanshikamra/Intelligent-Agents/blob/master/Neural/XOR.py

Some inferences from the data:

I tried to visualize the relationship between error and learning rate. The graphs for 0.5 and 0.9 learning rates respectively are shown below:





Larger the learning rate, the error decreases sooner(quite obvious xD)

Sources helpful for visualizing:

1. **edx : DAT275 course on ML :**(that's the place i've learnt all the algo's from)
2. **3 Blue 1 Brown videos on neural networks**
3. **https://www.youtube.com/watch?v=kNPGXgzxoHw**

   (helped me to realise the need of hidden neurons)