Name : ABID ALI
Student_No : 2019380141

18.19 Construct by hand a neural network that computes the XOR function of two inputs.
Make sure to specify what sort of units you are using.

Answer: XOR (in fact any Boolean function) is easiest to construct using step-function units. Because XOR is not linearly separable, we will need a hidden layer. It turns out that just one hidden node suffices. To design the network, we can think of the XOR function as OR with the AND case (both inputs on) ruled out. Thus the hidden layer computes AND, while the output layer computes OR but weights the output of the hidden node negatively. The network shown in Figure S18.2 does the trick.

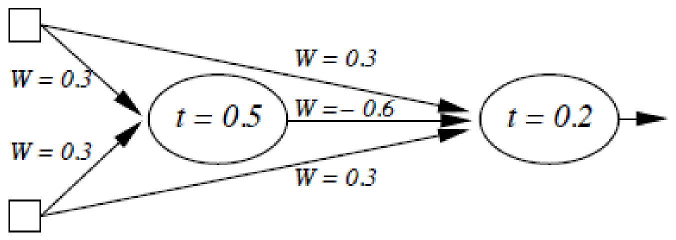


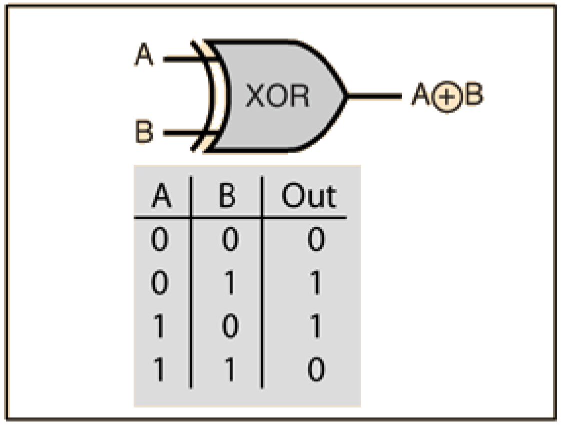
Figure S18.2 A network of step-function neurons that computes the XOR function.

Exercise 1: Compute a NN to realize function XOR based on Perceptrons.

Answer:

We conclude that a single perceptron with a Heaviside activation function can implement each one of the fundamental logical functions: NOT, AND and OR. They are called fundamental because any logical function, no matter how complex, can be obtained by a combination of those three. We can infer that**, if we appropriately connect the three perceptrons we just built, we can implement any logical function!**

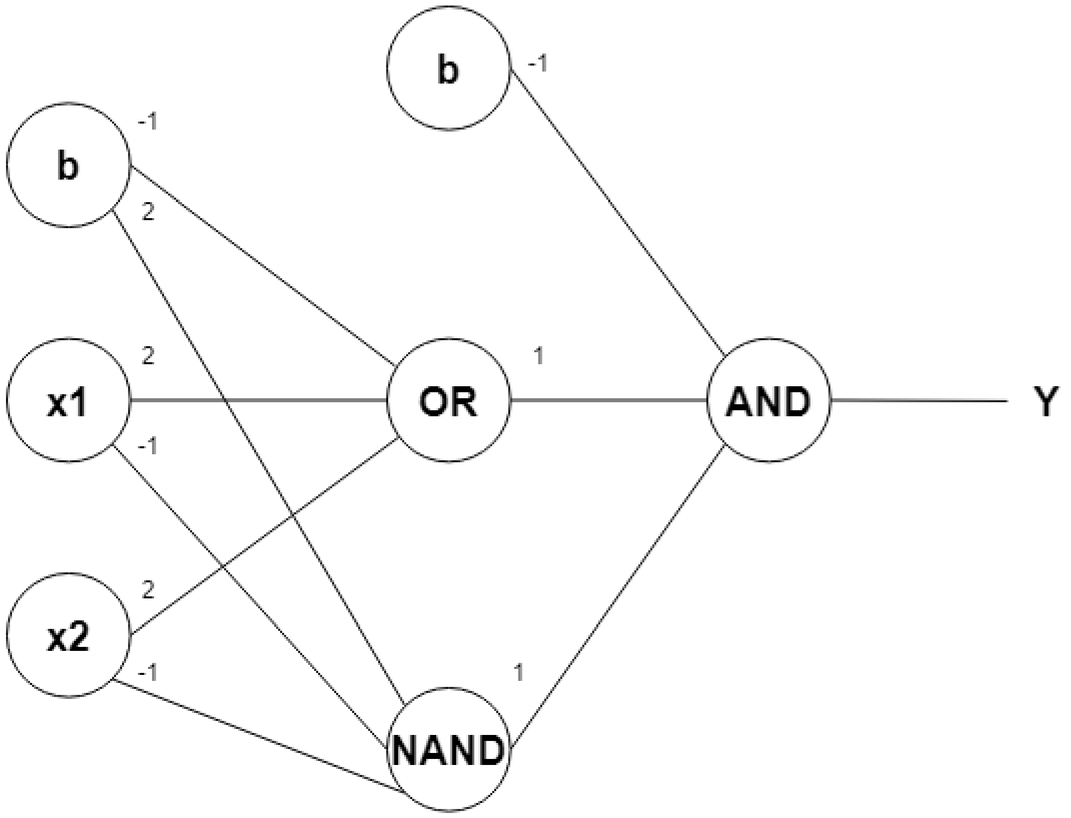
A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0



A	B	Out
0	0	0
0	1	1
1	0	1
1	1	0

The boolean representation of an XOR gate is;
x1x`2 + x`1x2
We first simplify the boolean expression
x`1x2 + x1x`2 + x`1x1 + x`2x2
x1(x`1 + x`2) + x2(x`1 + x`2)
(x1 + x2)(x`1 + x`2)
(x1 + x2)(x1x2)`

From the simplified expression, we can say that the XOR gate consists of an OR gate (x1 + x2), a NAND gate (-x1-x2+1) and an AND gate (x1+x2–1.5).
This means we will have to combine 2 perceptrons:
OR (2x1+2x2–1)
NAND (-x1-x2+2)
AND (x1+x2–1)



```
def
XOR_net(x):
            gate_1 = AND_percep(x)
            gate_2 = NOT_percep(gate_1)
            gate_3 = OR_percep(x)
            new_x = np.array([gate_2, gate_3])
            output = AND_percep(new_x)
            return output
          print("XOR({}, {}) = {}".format(1, 1, XOR_net(example1)))
          print("XOR({}, {}) = {}".format(1, 0, XOR_net(example2)))
          print("XOR({}, {}) = {}".format(0, 1, XOR_net(example3)))
          print("XOR({}, {}) = {}".format(0, 0, XOR_net(example4)))
```

And the output is:

```
XOR(1, 1) = 0
XOR(1, 0) = 1
```

XOR(0, 1) = 1
XOR(0, 0) = 0

In conclusion, this is just a custom method of achieving this, there are many other ways and values you could use in order to achieve Logic gates using perceptrons.
For example;
AND (20x1+20x2–30)
OR (20x1+20x2–10)
NOT (-20x1+10)
This will still work.

Exercise 2: Prove the formula of BP Algorithm based.
Answer:

Backpropagation (backward propagation) is an important mathematical tool for improving the accuracy of predictions in data mining and machine learning. Essentially, backpropagation is an algorithm used to calculate derivatives quickly.

The backpropagation equations provide us with a way of computing
the gradient of the cost function. Let's explicitly write this out in the form of an algorithm:

1. **Input x**: Set the corresponding activation a^1 for the input layer.
2. **Feedforward:** For each l=2,3 ,….L compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$.
3. **Output error δ^L** : Compute the vector $\delta^L = \nabla aC \odot \sigma'(z^l)$.
4. **Backpropagate the error:** For each l=L-1, L-2,…..2, compute $\boldsymbol{\delta^l} = ((w^{l+1})^T \boldsymbol{\delta^{l+1}}) \odot (\sigma')(z^l)$.
5. **Output:**The gradient of the cost function is given by $\frac{\partial C}{\partial w^l jk} = a^{l-1} k\ \delta^l\ j$ **and** $\frac{\partial C}{\partial b^l j}\ \delta^L\ j$.

Examining the algorithm you can see why it's called backpropagation. We compute the error vectors backward,
starting from the final layer. It may seem peculiar that we're going
through the network backward. But if you think about the proof ofbackpropagation, the b ackward movement is a consequence of the
fact that the cost is a function of outputs from the network. To
understand how the cost varies with earlier weights and biases weneed to repeatedly appl y the chain rule, working backward through the layers to obtain usable expressions.