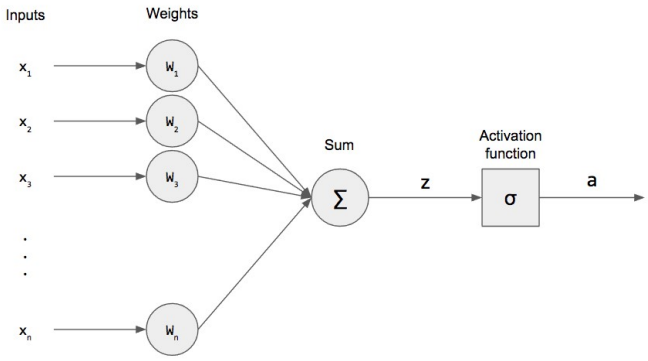
Neural Representation of AND, OR, NOT, XOR and XNOR Logic Gates (Perceptron Algorithm)

Stanley Obumneme Dukor Follow
Nov 12, 2018 · 7 min read



While taking the Udacity Pytorch Course by facebook, I found it difficult understanding how the Perceptron works with Logic gates (AND, OR, NOT, and so on). I decided to check online resources, but as of the time of writing this, there was really no explanation on how to go about it. So after personal readings, I finally understood how to go about it, which is the reason for this medium post.

First, we need to know that the Perceptron algorithm states that:

> *Prediction (y`) = 1 if Wx+b >= 0 and 0 if Wx+b<0*

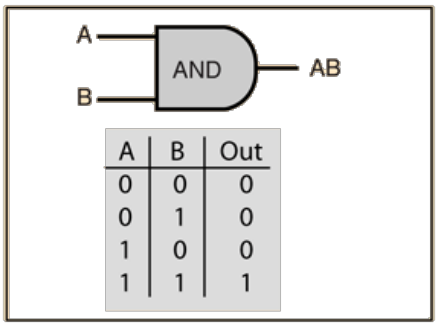
Also, the steps in this method are very similar to how Neural Networks learn, which is as follows;

- Initialize weight values and bias
- Forward Propagate
- Check the error
- Backpropagate and Adjust weights and bias
- Repeat for all training examples

Now that we know the steps, let's get up and running:

AND Gate

From our knowledge of logic gates, we know that an AND logic table is given by the diagram below



AND Gate

The question is, what are the weights and bias for the AND perceptron?

First, we need to understand that the output of an AND gate is 1 only if both inputs (in this case, x1 and x2) are 1. So, following the steps listed above;

Row 1

- From w1x1+w2x2+b, initializing w1, w2, as 1 and b as -1, we get;

> *x1(1)+x2(1)-1*

- Passing the first row of the AND logic table (x1=0, x2=0), we get;

> *0+0–1 = -1*

- From the Perceptron rule, if Wx+b<0, then y` =0. Therefore, this row is correct, and no need for Backpropagation.

Row 2

- Passing (x1=0 and x2=1), we get;

> *0+1–1 = 0*

- From the Perceptron rule, if Wx+b >= 0, then y` =1. This row is incorrect, as the output is 0 for the AND gate.
- So we want values that will make the combination of x1=0 and x2=1 to give y` a value of 0. If we change b to -1.5, we have;

> *0+1–1.5 = -0.5*

- From the Perceptron rule, this works (for both row 1, row 2 and 3).

Row 4

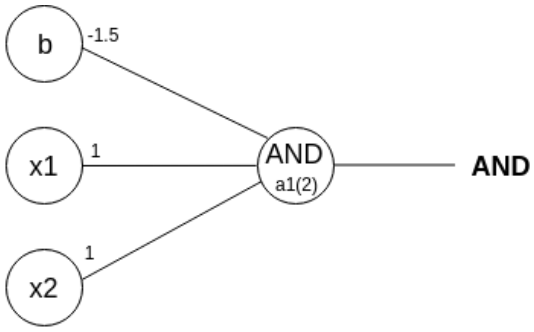
- Passing (x1=1 and x2=1), we get;

> *1+1–1.5 = 0.5*

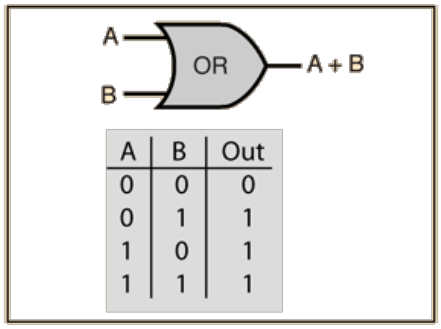
- Again, from the perceptron rule, this is still valid.

Therefore, we can conclude that the model to achieve an AND gate, using the Perceptron algorithm is;

> *x1+x2–1.5*



OR Gate



OR Gate

From the diagram, the OR gate is 0 only if both inputs are 0.

Row 1

- From w1x1+w2x2+b, initializing w1, w2, as 1 and b as -1, we get;

> *x1(1)+x2(1)-1*

- Passing the first row of the OR logic table (x1=0, x2=0), we get;

> *0+0–1 = -1*

- From the Perceptron rule, if Wx+b<0, then y` =0. Therefore, this row is correct.

Row 2

- Passing (x1=0 and x2=1), we get;

> *0+1–1 = 0*

- From the Perceptron rule, if Wx+b >= 0, then y` =1. This row is again, correct (for both row 1, row 2 and 3).

Row 4

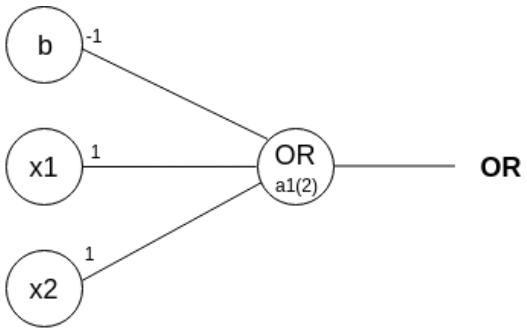
- Passing (x1=1 and x2=1), we get;

> *1+1–1 = 1*

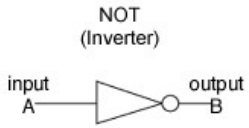
- Again, from the perceptron rule, this is still valid. Quite Easy!

Therefore, we can conclude that the model to achieve an OR gate, using the Perceptron algorithm is;

> *x1+x2–1*



NOT Gate



A	B
0	1
1	0

NOT Gate

From the diagram, the output of a NOT gate is the inverse of a single input. So, following the steps listed above;

Row 1

- From w1x1+w2x2+b, initializing w1 as 1, w2 as 0 (since single input), and b as -1, we get;

> *x1(1)-1*

- Passing the first row of the NOT logic table (x1=0), we get;

> *0–1 = -1*

- From the Perceptron rule, if Wx+b<0, then y` =0. This row is incorrect, as the output is 1 for the NOT gate.
- So we want values that will make input x1=0 to give y` a value of 1. If we change b to 1, we have;

> *0+1 = 1*

- From the Perceptron rule, this works.

Row 2

- Passing (x1=1), we get;

> *1+1 = 2*

- From the Perceptron rule, if Wx+b >= 0, then y` =1. This row is so incorrect, as the output is 0 for the NOT gate.
- So we want values that will make input x1=1 to give y` a value of 0. If we change w1 to -1, we have;

> *-1+1 = 0*

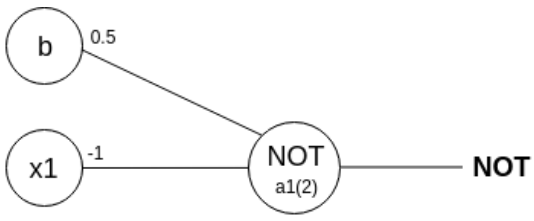
- From the Perceptron rule, this still wouldn't work, but it's better than the previous value. Changing b to -0.5

> *-1+0.5=-0.5*

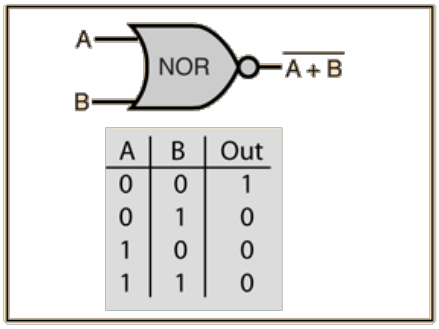
- That's it, it works (for both row 1 and row 2)

Therefore, we can conclude that the model to achieve a NOT gate, using the Perceptron algorithm is;

> *-x1+0.5*



NOR Gate



NOR Gate

From the diagram, the NOR gate is 1 only if both inputs are 0.

Row 1

- From w1x1+w2x2+b, initializing w1 and w2 as 1, and b as -1, we get;

> *x1(1)+x2(1)-1*

- Passing the first row of the NOR logic table (x1=0, x2=0), we get;

> *0+0-1 = -1*

- From the Perceptron rule, if Wx+b<0, then y` =0. This row is incorrect, as the output is 1 for the NOR gate.
- So we want values that will make input x1=0 and x2 = 0 to give y` a value of 1. If we change b to 1, we have;

> *0+0+1 = 1*

- From the Perceptron rule, this works.

Row 2

- Passing (x1=0, x2=1), we get;

> *0+1+1 = 2*

- From the Perceptron rule, if Wx+b >= 0, then y` =1. This row is incorrect, as the output is 0 for the NOR gate.
- So we want values that will make input x1=0 and x2 = 1 to give y` a value of 0. If we change w1 to -1 and w2 to -1, we have;

> *0-1+1 = 0*

- From the Perceptron rule, this still doesn't work because if Wx+b >= 0, then y` =1. Let's adjust the value of b to 0.5.

> *0–1+0.5 = -0.5*

- Cool, this works (for both row 2 and row 3).

Row 4

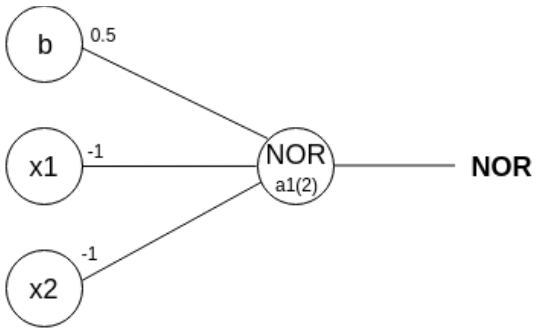
- Passing (x1=1, x2=1), we get;

> *-1-1+0.5 = -1.5*

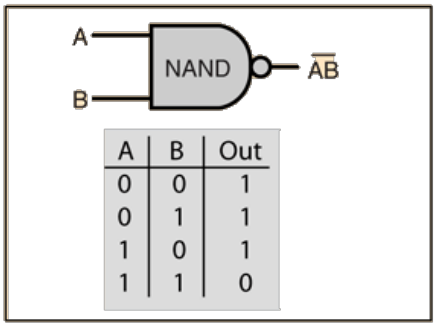
- From the Perceptron rule, this still works.

Therefore, we can conclude that the model to achieve a NOR gate, using the Perceptron algorithm is;

> *-x1-x2+0.5*



NAND Gate



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

From the diagram, the NAND gate is 0 only if both inputs are 1.

Row 1

- From w1x1+w2x2+b, initializing w1 and w2 as 1, and b as -1, we get;

> *x1(1)+x2(1)-1*

- Passing the first row of the NAND logic table (x1=0, x2=0), we get;

> *0+0-1 = -1*

- From the Perceptron rule, if Wx+b<0, then y` =0. This row is incorrect, as the output is 1 for the NAND gate.
- So we want values that will make input x1=0 and x2 = 0 to give y` a value of 1. If we change b to 1, we have;

> *0+0+1 = 1*

- From the Perceptron rule, this works.

Row 2

- Passing (x1=0, x2=1), we get;

> *0+1+1 = 2*

- From the Perceptron rule, if Wx+b >= 0, then y` =1. This row is also correct (for both row 2 and row 3).

Row 4

- Passing (x1=1, x2=1), we get;

> *1+1+1 = 3*

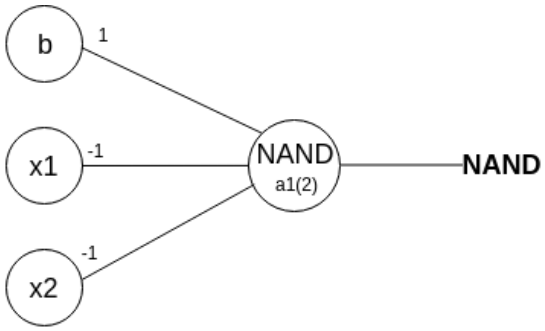
- This is not the expected output, as the output is 0 for a NAND combination of x1=1 and x2=1.
- Changing values of w1 and w2 to -1, we get;

> *-1-1+1 = -1*

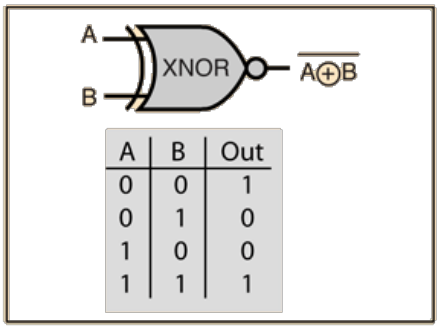
- It works.

Therefore, we can conclude that the model to achieve a NAND gate, using the Perceptron algorithm is;

> *-x1-x2+1*



XNOR Gate



XNOR Gate

Now that we are done with the necessary basic logic gates, we can combine them to give an XNOR gate.

The boolean representation of an XNOR gate is;

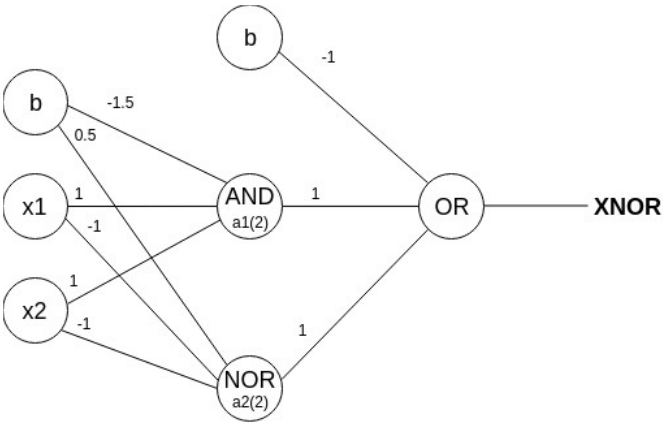
> *x1x2 + x1 `x2 `*

Where ‘`' means inverse.

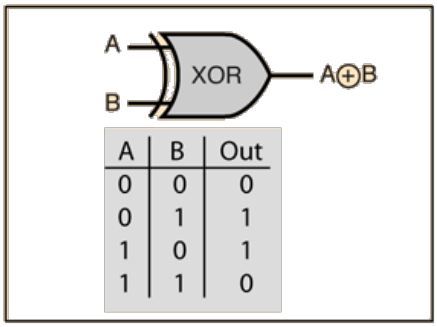
From the expression, we can say that the XNOR gate consists of an AND gate (x1x2), a NOR gate (x1`x2`), and an OR gate.

This means we will have to combine 3 perceptrons:

- AND (x1+x2–1.5)
- NOR (-x1-x2+0.5)
- OR (x1+x2–1)



XOR Gate



XOR Gate

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)*

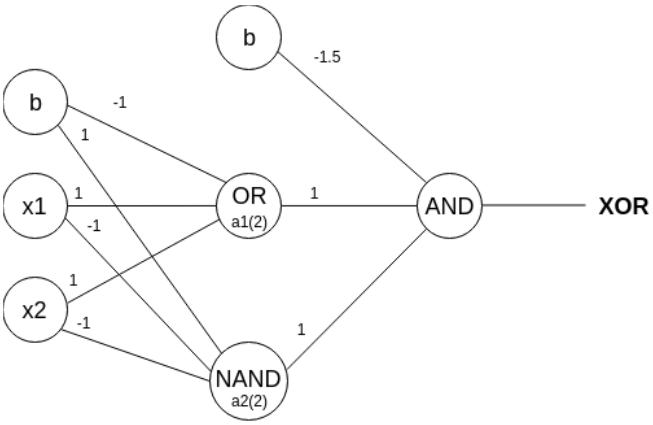
> *(x1 + x2)(x`1 + x`2)*

> *(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 (x1+x2–1)
- NAND (-x1-x2+1)
- AND (x1+x2–1.5)



CONCLUSION

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.

Thank you….

Udacity, Facebook Research