Logic Gates Implementation Via Single Layer Perceptron And Artificial Neural Networks

Amir Mahmood Mousavi _ 3971242026

***Abstract*-- Newly developed paradigms of artificial neural networks have strongly contributed to the discovery understanding and utilization of potential functional similarities between human and artificial information processing systems . this article focuses on one of the first ANN that has been used , a type of artificial neuron called a perceptron.**

This article aims to provide a comprehensive tutorial and survey about single layer perceptron and its problems and trying to find a solution for the problems .

***Index Terms*—perceptron , neural networks – single layer perceptron , NAND gate , logic gates , machine learning .**

I. Introduction

Artificial Neural Networks are a programming paradigm that seek to emulate the microstructure of the brain, and are used extensively in artificial intelligence problems from simple pattern-recognition tasks, to advanced symbolic manipulation. The Multilayer Perceptron is an example of an artificial neural network that is used extensively for the solution of a number of different problems, including pattern recognition and interpolation. It is a development of the Perceptron neural network model, that was originally developed in the early 1960s but found to have serious limitations.
In this article we will discuss logic gates implementation and single layer perceptron limitations along its solution .

II. Perceptron

Perceptrons were developed in the 1950s and 1960s by the scientist Frank Rosenblatt, inspired by earlier work by Warren McCulloch and Walter Pitts. Today, it's more common to use other models of artificial neuron called the sigmoid neuron . but to understand why sigmoid neurons are defined the way they are, it's worth taking the time to first understand perceptrons .
A perceptron takes several binary inputs ,and produces a single binary output . A way you can think about the perceptron is that it's a device that makes decisions by weighing up evidence
In the example shown the perceptron has three inputs, I1,I2,I3. In general it could
have more or fewer inputs. Rosenblatt proposed a simple rule to compute the output. He
introduced weights ,w1,w2,. . ., real numbers expressing the importance of the respective inputs to the output. The neuron's output, 0 or 1, is determined by whether the weighted

sum ∑jwjIj is less than or greater than some threshold value. Just like the weights, the threshold is a real number which is a parameter of the neuron. To put it in more precise coding terms , take a look at Fig . 1.

$$\text{if } \sum_i w_i I_i \geq t \text{ then } y = 1$$

$$\text{else (if } \sum_i w_i I_i < t \text{) then } y = 0$$

Fig . 1 .The output node has a "threshold" t.
If summed input ≥ t, then it "fires" (output y = 1).
Else (summed input < t) it doesn't fire (output y = 0).

After that we can move the threshold to the other side of the inequality, and to replace it by what's known as the perceptron's bias, b=−threshold. Using the bias instead of the threshold, the perceptron rule can be written :

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

Fig . 2 .

III. Perceptron learning rule

We don't have to design these networks. We could have *learnt* those weights and thresholds, by showing it the correct answers we want it to generate.
Let input x = (I_1, I_2, .., I_n) where each I_i = 0 or 1. And let output y = 0 or 1.

Algorithm is repeat forever:

- Given input x = (I_1, I_2, .., I_n). Perceptron produces output y. We are told correct output **O**.
- If we had wrong answer, change w_i's and t, otherwise do nothing.

Motivation for weight change rule:

a) If output y=0, and "correct" output **O**=1, then y is not large enough, so reduce threshold and increase w_i's. This will increase the output. (Note inputs cannot be

negative, so high positive weights means high positive summed input.)

Note: Only need to increase w_i's along the input lines that are active, i.e. where I_i=1. If I_i=0 for this exemplar, then the weight w_i had no effect on the error this time, so it is pointless to change it (it may be functioning perfectly well for other inputs).

b) Similarly, if y=1, **O**=0, output y is too large, so increase threshold and reduce w_i's (where I_i=1). This will decrease the output.
c) If y=1, **O**=1, or y=0, **O**=0, no change in weights or thresholds.

Hence algorithm is repeat forever:

a) Given input x = (I_1, I_2, .., I_n). Perceptron produces output y. We are told correct output **O**.
b) For all i:

$$w_i := w_i + C\,(\mathbf{O}\text{-}y)\,I_i$$

c) $t := t - C\,(\mathbf{O}\text{-}y)$

where C is some (positive) learning rate. Note the threshold is learnt as well as the weights. If **O**=y there is no change in weights or thresholds. If I_i=0 there is no change in w_i
The sequence of exemplars presented may go like this:

input x	network fires or not (y)	told correct answer (**O**)	what to do with w's	what to do with t
(0,1,0,0)	0	1	increase	reduce
(1,0,0,0)	0	0	no change	no change
(0,1,1,1)	1	0	reduce	increase
(1,0,1,0)	1	0	reduce	increase
(1,1,1,1)	0	1	increase	reduce
(0,1,0,0)	1	1	no change	no change
....	....	....	....	....

.

IV. NAND GATE

I've described perceptrons as a method for weighing evidence to make decisions. Another
way perceptrons can be used is to compute the elementary logical functions we usually think
of as underlying computation, functions such as AND, OR, and NAND . For example, suppose
we have a perceptron with two inputs, each with weight –2, and an overall bias of 3. Here's
our perceptron:

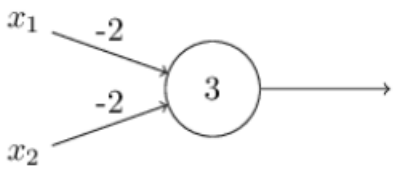
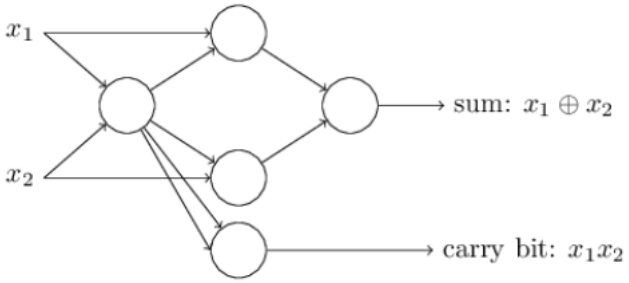


Fig . 4 .

Then we see that input 00 produces output 1, since$(-2)*0+(-2)*0+3=3$ is positive.
Here, I've introduced the $*$ symbol to make the multiplications explicit. Similar calculations
show that the inputs 01 and 10 produce output 1. But the input 11 produces output 0, since$(-2)*1+(-2)*1+3=-1$ is negative. And so our perceptron implements a NAND gate.

The NAND example shows that we can use perceptrons to compute simple logical
functions. In fact, we can use networks of perceptrons to compute any logical function at all. The reason is that the NAND gate is universal for computation, that is, we can build any computation up out of NAND gates. For example, we can use NAND gates to build a circuit which adds two bits. to understand how it works , take a look at Fig . 5 .



.

V. IMPLEMENTATION OF SINGLE LAYER PERCEPTRON

For the code, we'll start off by importing numpy and math . Numpy is a library with built in mathematical functions that is great for doing matrix multiplies and scientific programming. (Learn more about numpy here: http://www.numpy.org/) . we have set up randomly the initial weights .

Then we'll create a perceptron function that will act as the perceptron in the way told before. The function will take in a weight matrix, the bias and the dataset matrix. We multiply the dataset and the weights from the input and then add the output of the matrix multiply to the bias . "1/(1+np.exp(-model)" represents the activation function, passing the model into a sigmoid function. After creating the perceptron we need to fill in the inputs for it for that purpose we use nested loops.

```
for i in range(epochs):
    j = 0
    for x in train_data:
        res = w0 * x[0] + w1 * x[1] + w2 * x[2]

        if (res >= 0):
            act = 1
        else:
            act = 0
```

```
        # act = 1/(1+math.exp(-x))

        err = op[j] - act


        w0 = w0 + (alpha * x[0] * err)
        w1 = w1 + (alpha * x[1] * err)
        w2 = w2 + (alpha * x[2] * err)

        j = j + 1
```

After that we have a test phase to evaluate the accuracy of the final weights , we have generated 15 random input to test it :

```
test_data = [[0.98, 1], [0.01, 0.97], [0.77, 0.99], [0.912,
1.002], [0.88, 0.11], [0.82, 0.9], [0.8, 1], [0.02,
0.01],
             [0.21, 0.99], [0.11, 0.2], [0.79, 1], [0.11,
1.02], [0.98, 0.87], [0.2, 1.3], [0.2, 0.003]]

test_op = [0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1]  #
NAND
```

then we start testing and measuring the accuracy .
here we have some result from running the code :

```
Initial weights :
w0 =  0.469104811134157 w1 =  -0.33901645352207854 w2 =  0.7299839265839326
epoch : 100
error =  0

Final weights :
w0 =  0.669104811134157 w1 =  -0.5390164535220785 w2 =  -0.2700160734160675

x1 :  0.02 ,x2 :  0.01
Predicted :  1 ,Actual :  1

x1 :  0.21 ,x2 :  0.99
Predicted :  1 ,Actual :  1

x1 :  0.11 ,x2 :  0.2
Predicted :  1 ,Actual :  1

x1 :  0.79 ,x2 :  1
Predicted :  0 ,Actual :  0

x1 :  0.11 ,x2 :  1.02
Predicted :  1 ,Actual :  1

x1 :  0.98 ,x2 :  0.87
Predicted :  0 ,Actual :  0

x1 :  0.2 ,x2 :  1.3
Predicted :  1 ,Actual :  1

x1 :  0.2 ,x2 :  0.003
Predicted :  1 ,Actual :  1

errors  :  0

Accuracy :  100.0 %
```

Fig . 6 .

VI. Perceptron For XOR

XOR is where if one is 1 and other is 0 but not both.

Need:
$1.w_1 + 0.w_2$ cause a fire, i.e. >= t
$0.w_1 + 1.w_2$ >= t
$0.w_1 + 0.w_2$ doesn't fire, i.e. < t
$1.w_1 + 1.w_2$ also doesn't fire, < t

w_1 >= t
w_2 >= t
0 < t
w_1+w_2 < t
Contradiction.

Note: We need all 4 inequalities for the contradiction. If weights negative, e.g. weights = -4 and t= -5, then weights can be greater than t yet adding them is less than t, but t > 0 stops this.

A "single-layer" perceptron can't implement XOR. The reason is because the classes in XOR are not linearly separable. You cannot draw a straight line to separate the points (0,0),(1,1) from the points (0,1),(1,0). To put it in more precise coding terms , take a look at Fig . 6.

```
Initial weights :
w0 =  0.2965280083918306 w1 =  -2.2846196461852335 w2 =  -0.9825638209192572
epoch : 100
error =  -1
```

as you could see , the error is always -1

Fig . 7.

In the same way we even can prove that XNOR has the same problem . To find the solution Led us to invention of multi-layer networks . (we can by using multi-layer networks (AND & OR) , implement the XOR gate . it's shown in Fig . 8.)

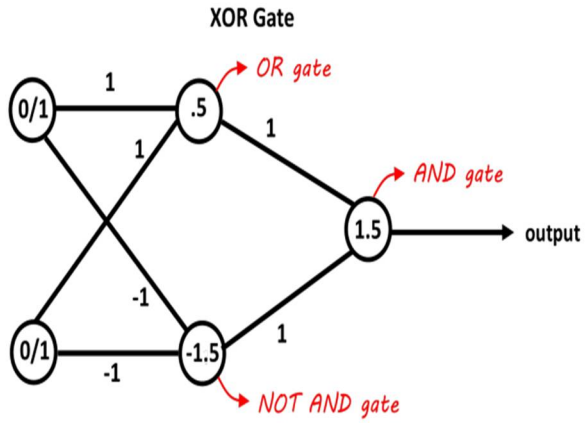


Fig . 8.

VII. Conclusion

The computational universality of perceptrons is simultaneously reassuring and disappointing. It's reassuring because it tells us that networks of perceptron can be as powerful aany other computing device. But it's also disappointing, because it makes it seem as though perceptrons are merely a new type of NAND gate. However , It turns out that we can devise learning algorithms which can automatically tune the weights and biases of a network of artificial neurons. This tuning happens in response to external stimuli, without direct intervention by a programmer. These learning algorithms enable us to use artificial neurons in a way which is radically different to conventional logic gates. Instead of explicitly laying out a circuit of NAND and other gates, our neural networks can simply learn to solve problems, sometimes problems where it would be extremely difficult to directly design a conventional circuit .

VIII. References

[1] http://static.latexstudio.net/article/2018/0912/neuralnetworksanddeeplearning.pdf

[2] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.608.2530&rep=rep1&type=pdf, 1993, pp. 123–135.

[3] https://computing.dcu.ie/~humphrys/Notes/Neural/single.neural.html.

[4] https://towardsdatascience.com/neural-representation-of-logic-gates-df044ec922bc

[5] https://stackoverflow.com

[6] http://lab.fs.uni-lj.si/lasin/wp/IMIT_files/neural/nn04_mlp_xor/

[7] https://www.youtube.com/watch?v=5rAhHGimTOU

[8] https://www.youtube.com/watch?v=kNPGXgzxoHw