

1 Introduction to Perceptron Learning Algorithm

We don't learn how the brain works by studying neurons, the same way that just by studying transistors we do not know how a computer works. We know the brain does processing but we don't know how it works. The bottleneck to understand brain is probably that we do not have the right abstractions to understand it.

McCulloch and Pitts developed a computational model of a biological neuron in 1943. The McCulloch-Pitts neural model is also known as linear threshold unit/gate. It is a neuron with a set of inputs and one output.

1.1 McCulloch-Pitts neurons vs biological neurons

similarities:

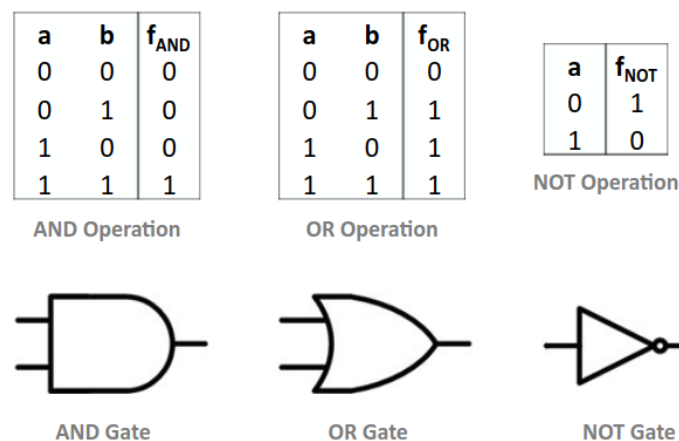
- Both can be active or inactive.
- The input/output is directed.
- The activation of a neuron is dependent on a weighted function of other neurons.

differences:

- Real neurons exist in continuous time, whereas McCulloch-Pitts neurons operate in discrete time.
- Real neurons have degrees of activation, not just on/off.
- The activation as a function of the inputs of real neuron is typically not linear or threshold linear.

1.2 (Basic) Digital Logic

Gates are processing units.



Circuits are a combination of gates.

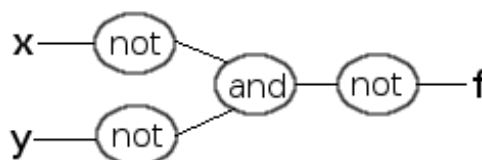


Figure 1: A basic circuit implementing an OR gate.

The circuit in Figure 1 produces the same output as an OR gate. With NOT and AND gates we can build an OR gate, but with AND and OR we can't build a NOT gate.

XOR gates = exclusive OR They are exclusive in the case of two inputs. For more than two, XOR counts the number of "active" (1's) inputs and returns 0 for even and 1 for odd.

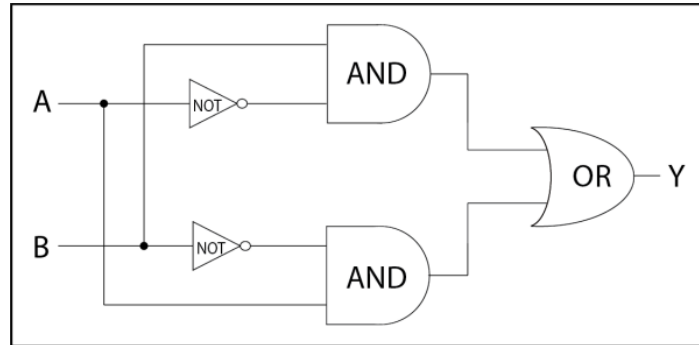


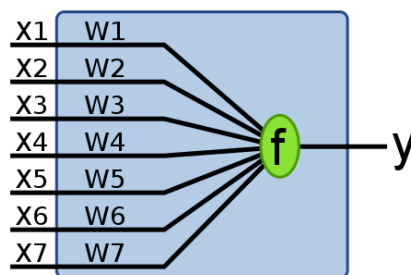
Figure 2: XOR gate built out other gates. Image from <https://blog.digilentinc.com/building-logic-gates-with-transistors/>

A table with N inputs has 2^N rows.

Other gates

Input		Output						
A	B	AND	OR	NOT	XOR	NAND	NOR	XNOR
0	0	0	0	1	0	1	1	1
0	1	0	1	1	1	1	0	0
1	0	0	1	0	1	1	0	0
1	1	1	1	0	0	0	0	1

1.3 Linear Threshold (LT) Unit/Gates (Perceptron)



This model represents a neuron with inputs x and one output y . Weights w determine the influence of the inputs. f is a function determining the output: if the influence of all the inputs combined cross a threshold, then the neuron become active. Active state: $\sum_i (w_i \cdot x_i) \geq \theta$. Otherwise, the neuron is inactive.

We add a bias input as $-\theta$ so that $w_0 + \sum_i (w_i \cdot x_i) \geq 0$ activates the neuron. As a digital abstraction, we consider that the activity of a neuron can be described as 0 or 1. Where 0 is inactive and 1 is active.

Neurons can learn by adjusting their weights, they can have thousand of inputs. For a desired behavior there are many possible weight vectors that can work (if any can work). This model can create AND/OR/NOT-Gates.

1.3.1 Threshold function

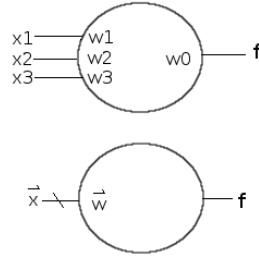


Figure 3: Same unit in two representations

Both units in Figure 3 are the same. The one in the bottom is a simplified version, where the inputs x and the weights w are represented as vectors. The bias term is added to the weight vector and a new input x_0 is added with a fixed value of 1.

$$f(x_1, x_2, x_3) = \theta(w_1 \times x_1 + w_2 \times x_2 + w_3 \times x_3 + w_0)$$

$$f(\vec{x}) = \theta(\vec{w} \times \vec{x} + w_0)$$

$$\theta(x) = \begin{cases} 0, & \text{if } x < 0 \\ 1, & \text{if } x \geq 0 \end{cases}$$

1.3.2 XOR impossibility with LT/Perceptrons

To compute XOR with LT, it is required that:

- $w_0 < 0$
- $x_2 \times w_2 + w_0 \geq 0$
- $x_1 \times w_1 + w_0 \geq 0$
- $x_1 \times w_1 + x_2 \times w_2 + w_0 < 0$.

This causes a contradiction because $x_1 \times w_1 + x_2 \times w_2 + 2 \times w_0 < 0$ and $x_1 \times w_1 + x_2 \times w_2 + 2 \times w_0 \geq 0$ when adding up the constraints.

XOR is not a linear combination of the inputs. Although these units can't model the XOR, they are still powerful. You can't calculate it with a single unit, but with a combination of them, it is possible. The same way, one cannot compute the XOR function using one single OR, NOT or AND gate. In fact, we need three gates¹ to compute the XOR function, see Figure 2.

In the end, these model of neuron units are more efficient than digital logic gates, however, these new analog units run into precision problems. For implementing the XOR function, McCulloch and Pitts units allow exponentially smaller circuits.

1.4 Perceptron Learning Algorithm

x1	x2	f
0	0	1
0	1	0
1	0	1
1	1	1

Can we set w_0, w_1, w_2 so that this unit computes the above f ? We start with random weights, let's say all zero. With this weights, doesn't matter the values of x_1 and x_2 , the result will always be 1. And for the second case ($x_1 = 0, x_2 = 1, f = 0$) we produce a wrong output.

¹when counting gates, we do not count NOT gates

So, we reduce w_0 and w_2 because they contribute to the sum ($0 \times w_1 + 1 \times w_2 + w_0$). We reduce the weights if the sum should go down and we increase the weights if the sum should go up. We iterate this step until convergence.

We can consider the bias as a weight with input value always one ($x_0 = 1$), thus, we can write:

$$f(\vec{x}) = \theta(\vec{w} \times \vec{x} + w_0) = \theta(\vec{w} \times \vec{x})$$

Recall $\vec{w} \times \vec{x} = |\vec{w}| \times |\vec{x}| \times \cos \alpha$. So, if $\alpha < 90$ deg, $\vec{w} \times \vec{x}$ is greather than zero and if $\alpha > 90$ deg, $\vec{w} \times \vec{x} < 0$.

This way, we compute a similarity between the weight vector and the input vector.

The weights of the perceptron units can be seen as the components of a normal vector to the hyperplane that separates the classes (1 and 0). In fact, the length of the normal vector doesnt matter, we are looking for its direction.

Convergence Suppose there is a solution \vec{w}^* , i.e., the data is linearly separable. Pick any solution \vec{w} , for instance, starting with all weights as zero, if this solution already satisfies our conditions, we are done. Otherwise, we pick an arbitrary missclassified point and update \vec{w} . Each step makes progress in the \vec{w}^* direction, because additions to \vec{w} are always ≤ 90 deg. The magnitude of $\vec{w}^* \times \vec{w}$ increases linearly. Since \vec{w}^* doesn't change, there is a maximum growth from \vec{w} to achieve the solution. By contradiction in the limit of infinite steps, we can say that the algorithm converges, i.e., if there is a solution and it takes infinite steps to achieve it, this is a contradiction.

Algorithm

- Choose random initial weights.
- Calculate output for given input.
- If the output is not the expected value, then $e = d - c$, where d is the desired output and c the current output.
- Change the weight of inputs and bias by $\Delta w_i = e \cdot \alpha \cdot x_i$. For the bias, always use $x = 1$.

1.5 Converting real weights to integer

Any unit can be converted into one with integer weights.

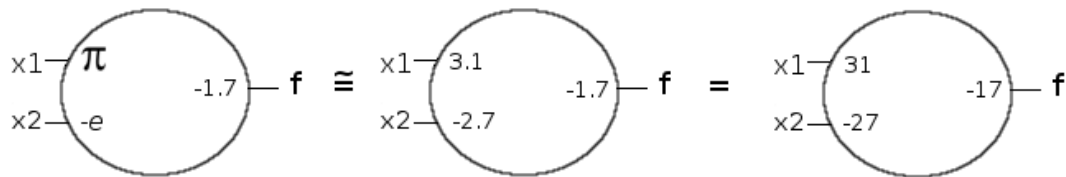


Figure 4: Converting real weights to integer

Figure 4 shows the approach to convert real weights to integer. The approximation (images left and center) can be problematic if x in $\theta(x)$ is equal to zero (in the real setup), because it can be shifted in uthe approximation. In the second step (images center and right) there is no problem. By multiplying the weights by a factor of 10, we do not change the value of x in $\theta(x)$.

One way to solve the above problem (approximation) is to shift the bias term a bit.

1. Find the largest possible negative sum (closest to zero)
2. Adjust bias (increase) if necessary, so no sum is zero
3. Replace weights with rational approximations more precise than sum closest to zero
4. Scale up weights to be integers

In the first two steps, we avoid have a sum of zero. On the last two, we transform the weights to integer.

So far, the difference of this unit and a real neuron is that neurons can adjust their own weight (learning) and these units are behaving as gates.