AI for Astrophysics: Simple Neural Networks

David Cornu

LUX, Observatoire de Paris, PSL

Doctoral course – ED AAIF 2025/2026

The brain as model

« There is a fantastic existence proof that learning is possible, which is the bag of water and electricity (together with a few trace chemicals) sitting between your ears [...] which is the squishy thing that your skull protects »

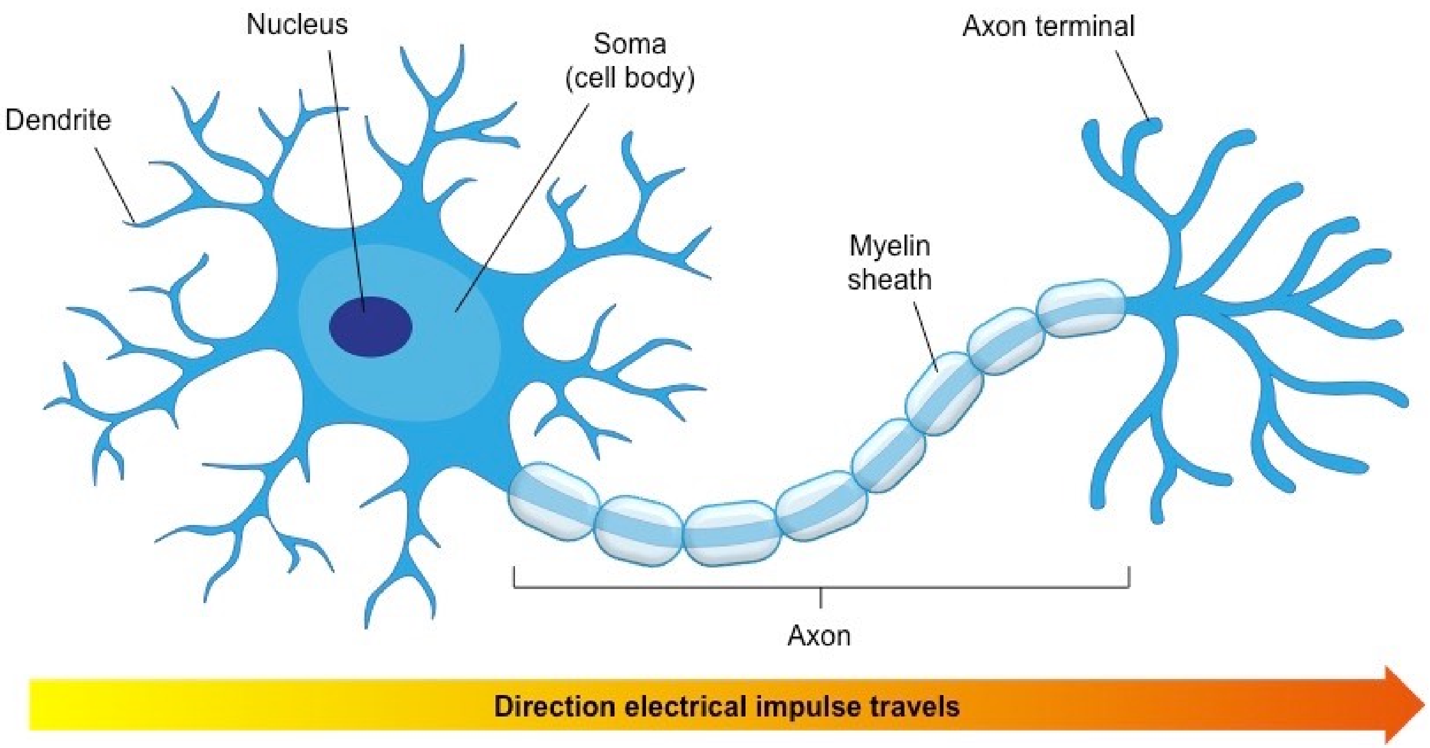
**Stephen Marsland*

The brain does exactly what we want for data analysis:

- Extract complex information in a compressed form
- Deal with noisy and/or inconsistent data
- Work in highly-dimensional spaces
- Give the appropriate answer most of the time
- Provide results very quickly
- Remain robust through aging (neuron loss)

The biological neuron

Elementary brick of a biological brain (10^{11} in the human brain).
The idea will be to use it as a model to emulate learning capabilities.



Perform the sum of various electrochemical input signals. If this total signal is sufficient, it sends a new signal through its axon to transfer information (toward other neurons).

Biological Neural Networks

A connection between two neurons is called a synapse (10^{14} in the human brain).
A neuron is a binary compute unit that either “fire” or “not-fire” in response to a signal.

➡ **In this view, a brain is a massively parallel super-computer of 10^{11} processing units.**

A simplified view of how it learns

The synapses represent the “strength” of the connection between two neurons.

Learning = modifying this connection (either positive or negative and changing its intensity)

➡ **This is called plasticity**

The Hebb law defines a simplified rule for learning:

If two neurons “fire” at the same time, there must be some **correlation** between them, and their connection must be strengthened.

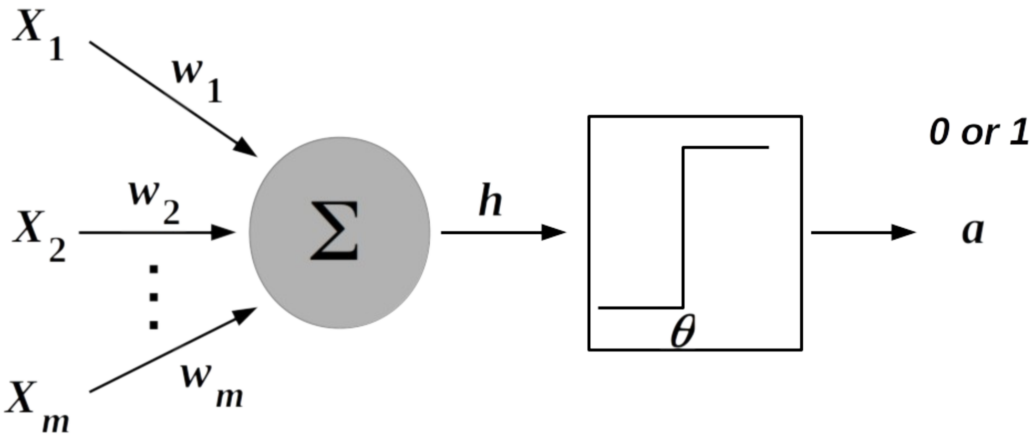
➡ **This is called conditioning**

**These rules are not enough to train a neural system but illustrate the processes that occur in the biological brain.*

To create an algorithm from these biological concepts, one first need to create a mathematical model.

Model of a Neuron

Mathematical model from McCulloch and Pitts, inspired by the biological neuron.

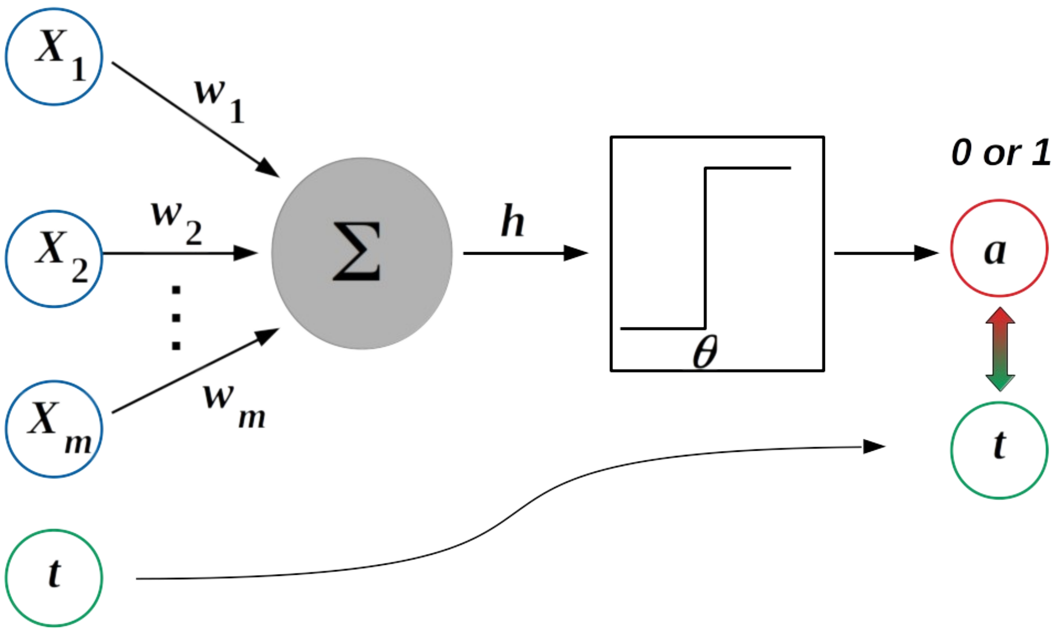


$$h = \sum_{i=1}^{m} X_i w_i \qquad a = g(h) = \begin{cases} 1 & \text{if} \quad h > \theta \\ 0 & \text{if} \quad h \leq \theta \end{cases}$$

Its main components are:

- An input vector X_i that represents the dimensions of a given object
- A set of weights w_i that links the various input dimensions to the neuron
- A sum function h that defines how these weights are combined with the input dimensions
- An activation function $g(h)$ that defines if the network should remain in a "0" state or should be activated to a "1" state, depending on the results of the sum.

Training a Neuron



$$h = \sum_{i=1}^{m} X_i w_i \quad a = g(h) = \begin{cases} 1 & \text{if} \quad h > \theta \\ 0 & \text{if} \quad h \leq \theta \end{cases}$$

The learning process for this model is supervised
→ Each input vector is associated to a target value

So what is the purpose of the neuron if the expected value is already known ?

Generalization

Find patterns in the data distribution in the parameter space so it can perform prediction on vectors with unseen (but close) values.

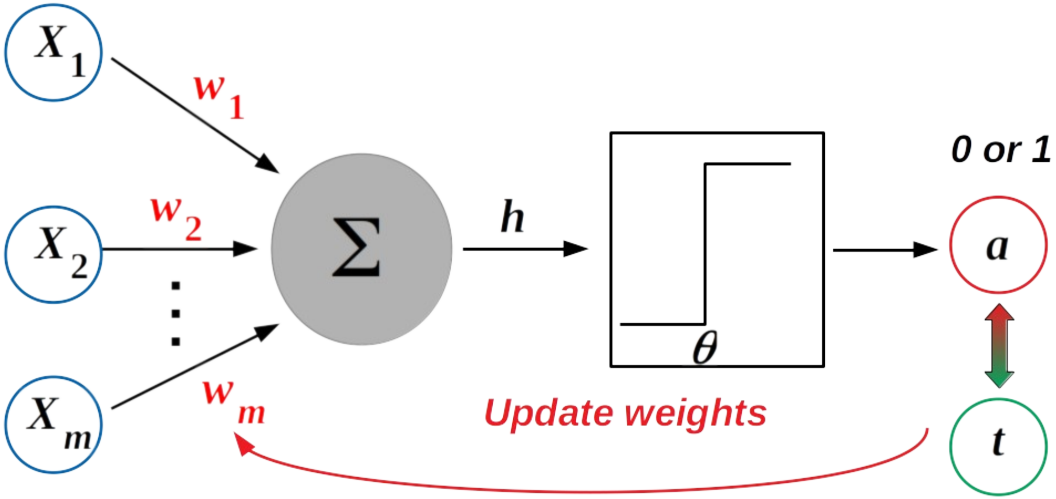
In practice, such a simple neuron optimizes a linear separation in the parameter space (**hyperplane**).

How does this model learn ?

Which parameters can or cannot be modified ?

The input, output, and targets are fixed, so the learning relies on modifying the **weights and the activation threshold**.

Training a Neuron



$$\omega_i \leftarrow \omega_i - \eta\ (a - t)\ x_i$$

$$h = \sum_{i=1}^{m} X_i w_i \qquad a = g(h) = \begin{cases} 1 & \text{if} \quad h > \theta \\ 0 & \text{if} \quad h \leq \theta \end{cases}$$

How to proceed when the output does not correspond to the target ?

There are *m* weights w_i associated with the network corresponding to the input dimensions.

How to modify the weights?

- If the output state is 1 while it should be 0, the weights need to be lowered
- If the output state is 0 while it should be 1, the weights need to be increased

To quantify this modification, one needs to choose an error function *E.*

$$E = 0.5 \times (a - t)^2$$

In the end, the weight update is defined as proportional to the input value, to the derivative of the error function for each dimension, and scaled by a learning rate* factor.

$$\omega_i \leftarrow \omega_i - \eta\ (a - t)\ x_i$$

**Typically between 0.1 and 0.4, details will be given later.*

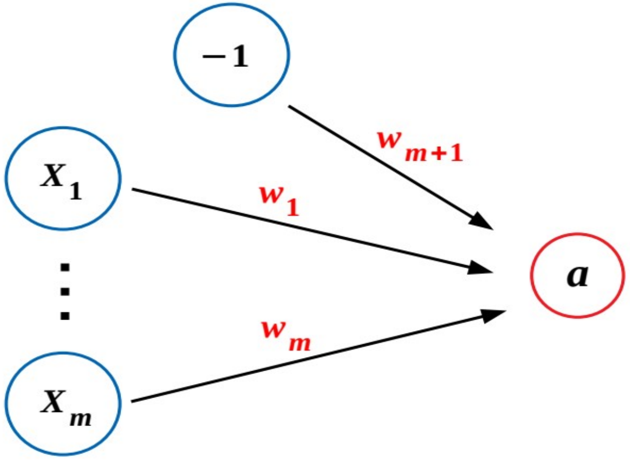
The bias node

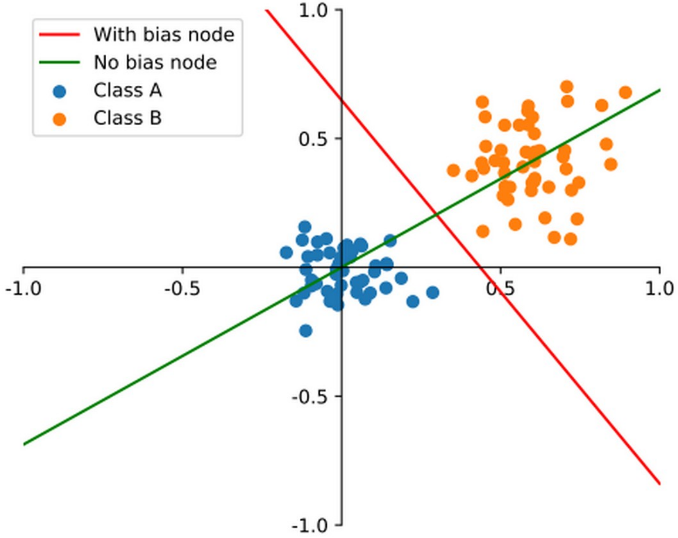
Problem with the previous formalism

The linear separation presents a fixed *f(0) = 0 point.*
The weight correction is also 0, regardless of the input value.

Solution

Add a bias input node that acts as an additional input with a constant value (usually -1). It has its own **variable weight** so it can learn the shift of the intercept position. The size of both the input vector and the weights vector is now *m+1*.





To plot the found linear separation of a neuron, we need to find where in the parameter space the neuron changes state, which corresponds to:

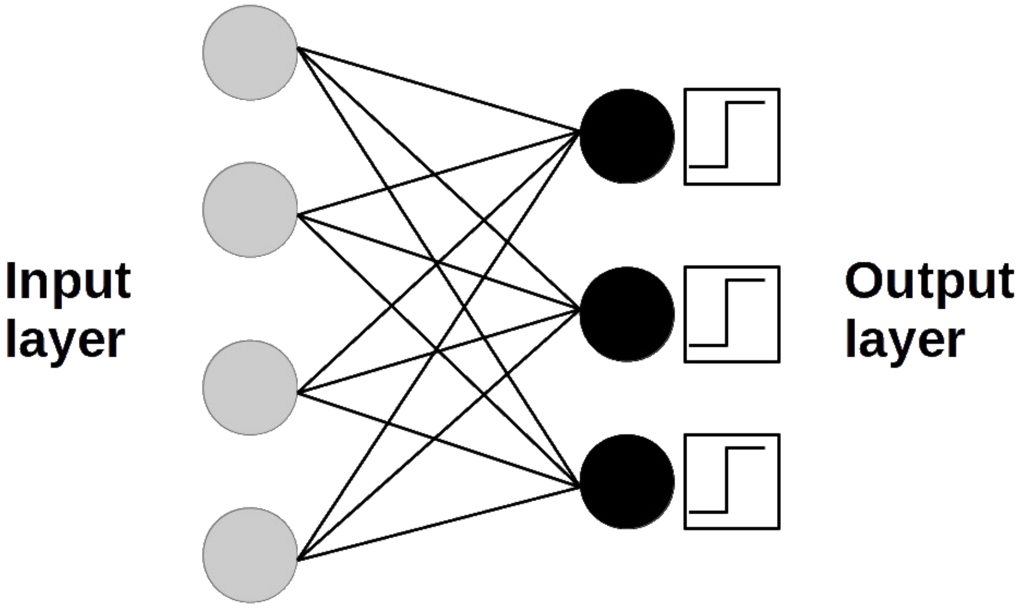
$$\sum_{i=1}^{m} X_i w_i = h = 0$$

For a 2D parameter space it results in:

$$X_1 = (W_b - X_0 W_0)/W_1$$

Note that the direction of activation is orthogonal to the linear separation.

The Perceptron algorithm



A single neuron only performs a linear separation, which is insufficient for many applications.

The simplest way to combine neurons on a single problem is to **stack them independently**. Each neuron is connected to the input vector with its own set of weights.

The training procedure is identical, but this time there is an index *j* to represent the neurons, and the weights are now in the form of a 2D matrix.

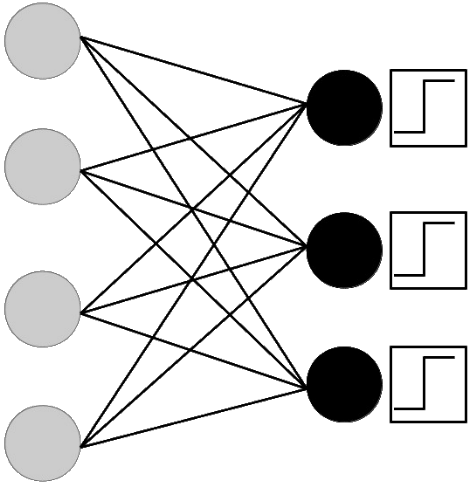
$$h_j = \sum_{i=1}^{m+1} x_i \omega_{ij} \qquad a_j = g(h_j) = \begin{cases} 1 & \text{if} \quad h_j > \theta \\ 0 & \text{if} \quad h_j \leq \theta \end{cases}$$

$$\omega_{ij} \leftarrow \omega_{ij} - \eta \left(a_j - t_j \right) \times x_i$$

The combination of this training procedure and this neuron connection scheme is called the single layer “Perceptron” (Rosenblatt 1958).

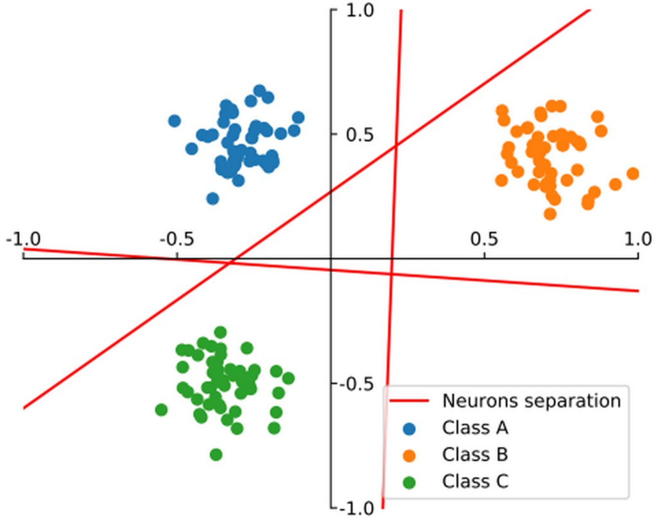
The Perceptron algorithm

Input layer



Output layer

C1	C2	C3
1	0	0
0	1	0
0	0	1



How to have multiple neurons work on the same problem ?

1) Encode the output vector (e.g in binary format)
2) Have one neuron per output class and target a format with a 1 only in the appropriate class, e.g [1,0,0]-[0,1,0]-[0,0,1]

**Warning: Since each neuron only performs a linear separation, each class must be linearly separable from all the others.*

The Perceptron algorithm

- **Initialization**
  - Set the starting weights to small random values (positive and negative). Can be drawn from a uniform or Gaussian distribution centered on zero.

- **Training**
  - For a given number of steps T, or until the output is “correct”
    - ➔ For each input vector
      - Compute the activation of each neuron *j* using the activation function *g :*

$$a_j = g\left(\sum_{i=0}^{m+1} w_{ij} x_i\right) = \begin{cases} 1 & \text{if} \quad \sum_{i=0}^{m+1} w_{ij} x_i > 0 \\ 0 & \text{if} \quad \sum_{i=0}^{m+1} w_{ij} x_i \leq 0 \end{cases}$$

      - Update each weight individually using :

$$\omega_{ij} \leftarrow \omega_{ij} - \eta\left(a_j - t_j\right) \times x_i$$

- **Inference**
  - Compute the final activation of each neuron *j* for each input vector to test

First application : logical gates

OR		A	
		0	1
B	0	0	1
	1	1	1

AND		A	
		0	1
B	0	0	0
	1	0	1

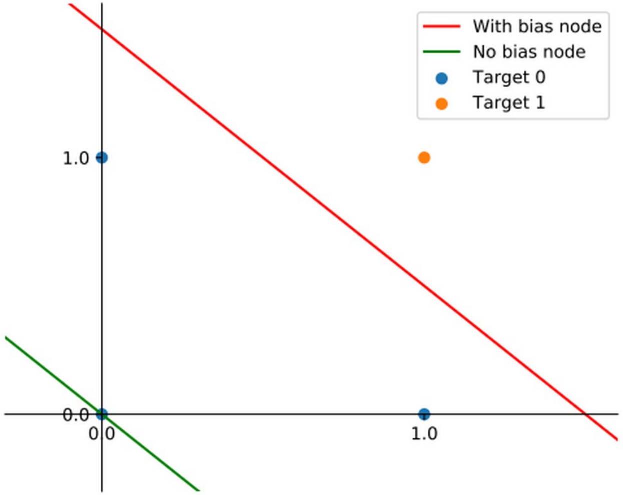
XOR		A	
		0	1
B	0	0	1
	1	1	0

NOR		A	
		0	1
B	0	1	0
	1	0	0

NAND		A	
		0	1
B	0	1	1
	1	1	0

A NOT $\bar{A}$

A	
0	1
1	0



A straightforward application of this algorithm is to make it learn a logical door, here the OR or AND gate. For this simple application, the Perceptron has a two-dimensional input vector and 4 possibilities as input-target pairs.

The output is made of a single binary neuron to predict either the 0 or 1 state of the gate.

As a first exercise:

- Write a simple program that declares an array with all the possible inputs and the associated targets.
- Identify all the necessary variables and initialize the weights with small values.
- Try to train this neuron over a few iterations following the Perceptron algorithm.

Do not forget the bias node!

- Display the output at each iteration to watch the network converge toward a stable solution.

Why is the **XOR gate** problematic with this algorithm?

How to solve the problem?