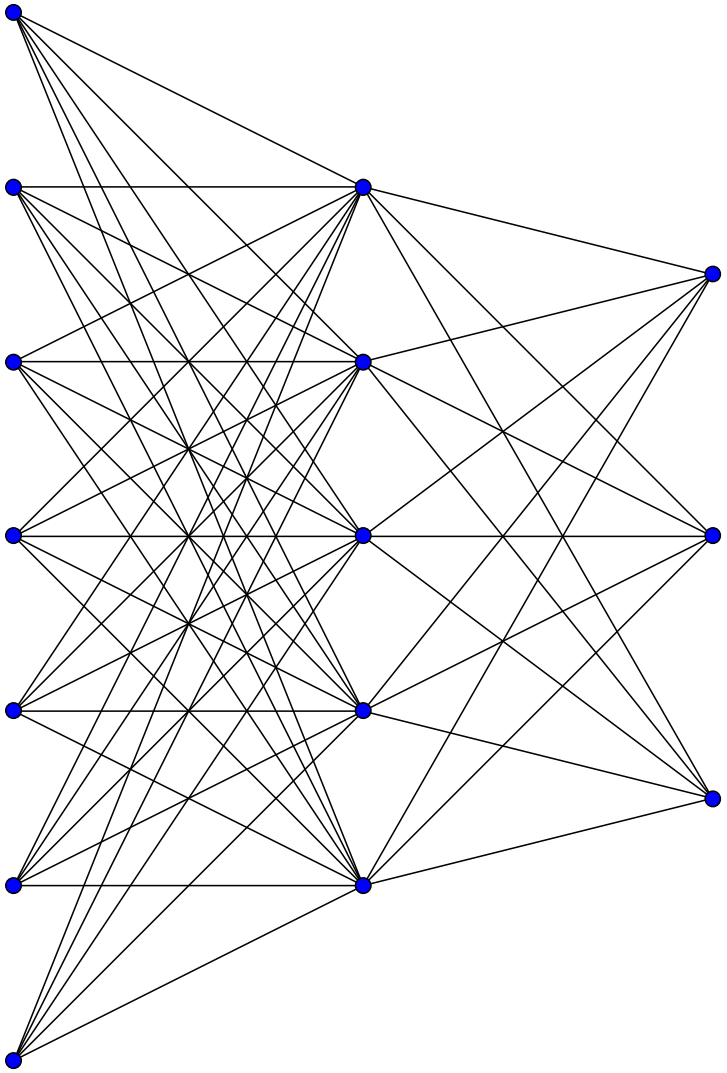
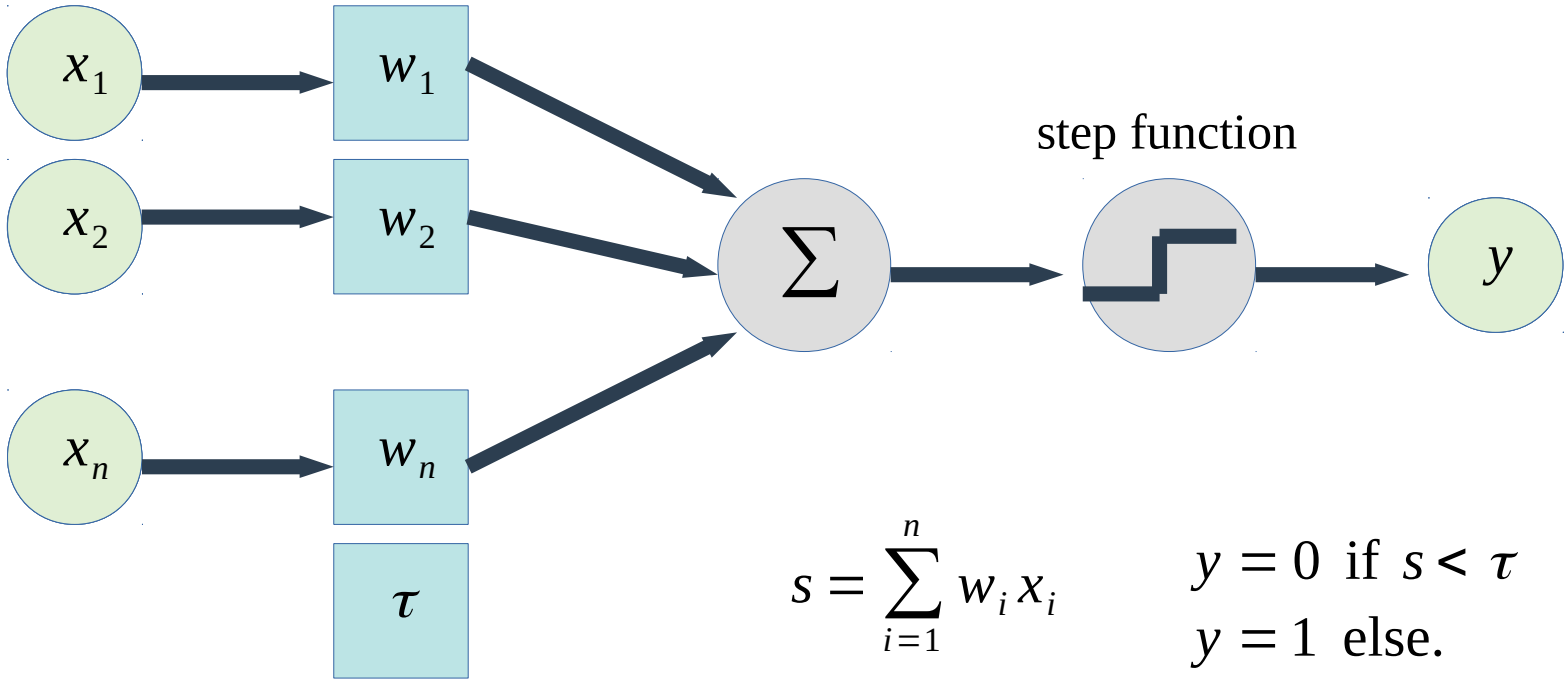
Neural networks



Perceptron:

Given a set of binary inputs $x_1, \ldots, x_n$, we want to produce a single binary output y.
For each x_i is assigned a weight w_i, and a threshold τ is fixed.



$$s = \sum_{i=1}^{n} w_i x_i$$

$y = 0$ if $s < \tau$
$y = 1$ else.

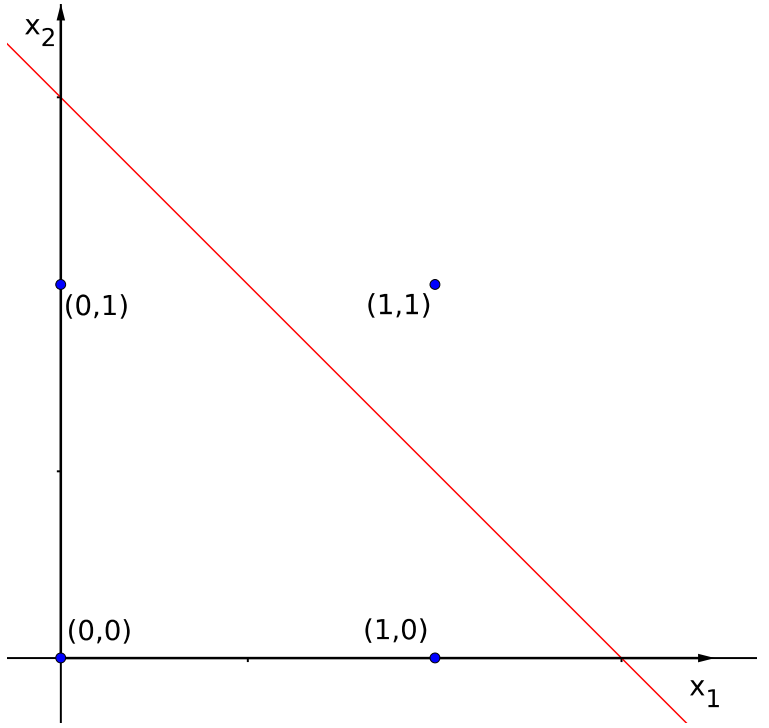
Now for the system to output correct answers, the parameters $w_1, \ldots, w_n$ and τ need to be learned.

Example - the logical gate AND:

For any binary input x_1 and x_2, we want the perceptron to output $x_1 \wedge x_2$.

There is an infinite number of tuples (w_1, w_2, τ) such that $w_1 x_1 + w_2 x_2 \geq \tau$ if and only if $x_1 = 1$ and $x_2 = 1$. Finding a solution here, is equivalent to finding a line of the euclidean plane, such that the points $(0,0), (0,1), (1,0)$ are not in the same half-plane as $(1,1)$. One possible solution is:

$(w_1, w_2, \tau) = (0.2, 0.2, 0.3)$.

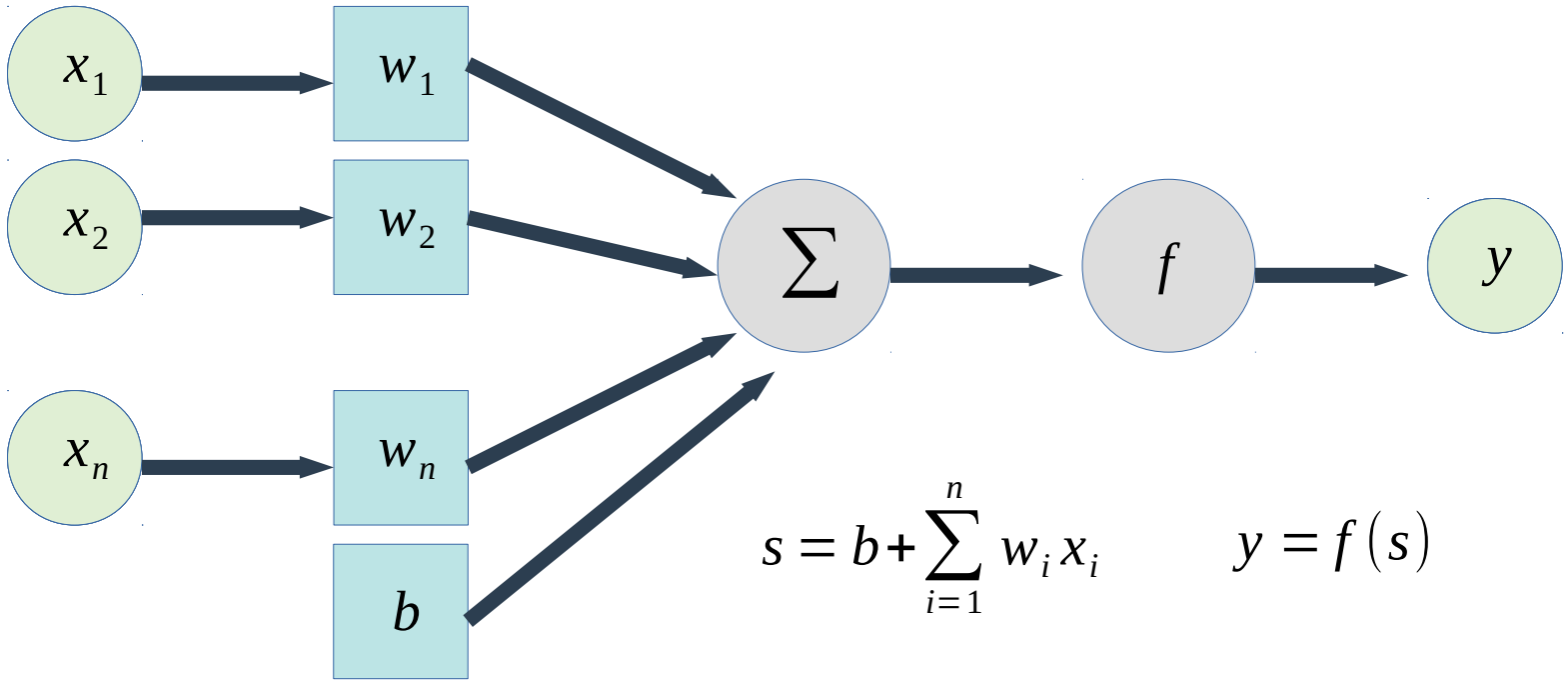


Exercice: show that the logical gate XOR cannot be learned by a perceptron.

Artificial neuron:

An artificial neuron is an improvement over the perceptron: it uses a continuous function instead of the step function. This function is called activation.

Finally, the threshold τ is replaced by a term b called the bias:

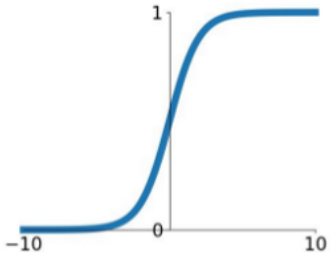


A neuron's inputs and output should be between 0 and 1.

Activation functions:

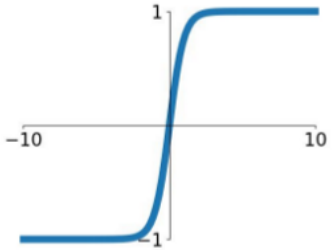
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



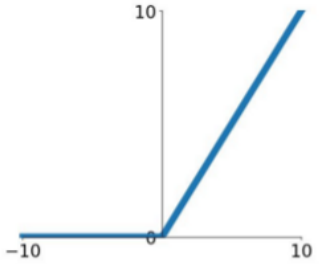
tanh

$$\tanh(x)$$



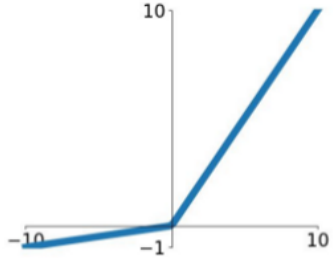
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

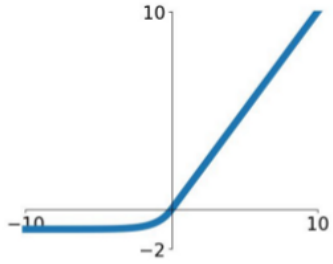


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



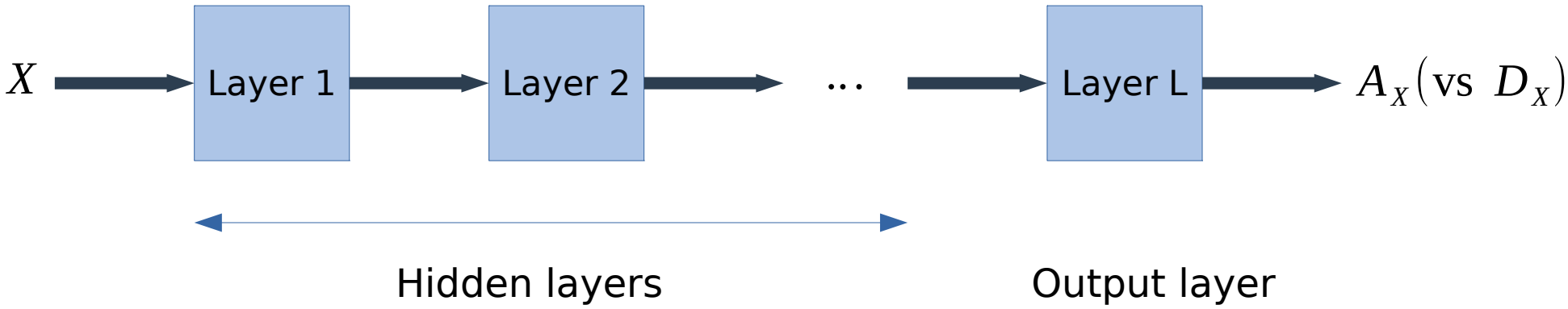
Improvements:

- Use several neurons in order to be able to answer non-binary questions.

- Use several layers of neurons, in hope that the network will be able to generalize from layers to layers.

Structure of a neural network:

L layers. For any input X, the network outputs a vector A_X for answer. During the learning phase, a given input X will be labelled with a desired answer D_X, which will be compared to A_X. The vector D_X should have the same size as A_X, and be filled with 0 and 1.



The output of each layer is the input of the next one.

Structure of the *l*-th layer:

X^{l-1}: l-th input vector

W^l: l-th weight matrix

B^l: l-th bias vector

S^l: l-th sum vector

G^l: l-th gradient vector

f_l: l-th activation funtion

X^l: l-th output vector

Sizes: B^l, S^l, G^l, X^l: $1 \times n_l$

W^l: $n_l \times n_{l-1}$

Where: $X^0 = X$, the network's input.

$X^L = A_X$, the network's output.

Feedforward:

For a given input X, one can propagate said input through the network by doing for every layer:

$$S^l = X^{l-1}(W^l)^t + B^l$$

$$X^l = f_l(S^l)$$

Once the learning is done, only this action will be performed on new inputs.

Weights initialization:

For a fast convergence, initial weights should not be too large. For every layer l, it is advised to initialize each weight with a normal distribution of mean 0 and standard deviation $\frac{1}{\sqrt{n_{l-1}}}$. If a normal distribution generator is not available, an uniform distribution on the interval $[\frac{-1}{\sqrt{n_{l-1}}}, \frac{1}{\sqrt{n_{l-1}}}]$ may also be used.

Finally, biases can either be initialized with the previous distributions, or be set to 0.

Loss function:

The loss function is a map $\mathscr{L} : \mathbb{R}^{n_0} \to \mathbb{R}_+$, which is used for the learning phase.

For any input X it returns a positive value, with: $\mathscr{L}(X) = 0 \Leftrightarrow A_X = D_X$.

The total loss will be the sum of every input's loss from the learning dataset $\mathbb{D}$:

$$\mathscr{L}_T = \sum_{X \in \mathbb{D}} \mathscr{L}(X)$$

During the learning, we will try to minimize the total loss.

There is several loss functions which can be chosen, for example:

Quadratic loss: $\mathscr{L}(X) = \frac{1}{2} \| A_X - D_X \|_2^2 = \frac{1}{2} \sum_{i=1}^{n_L} (a_i - d_i)^2$

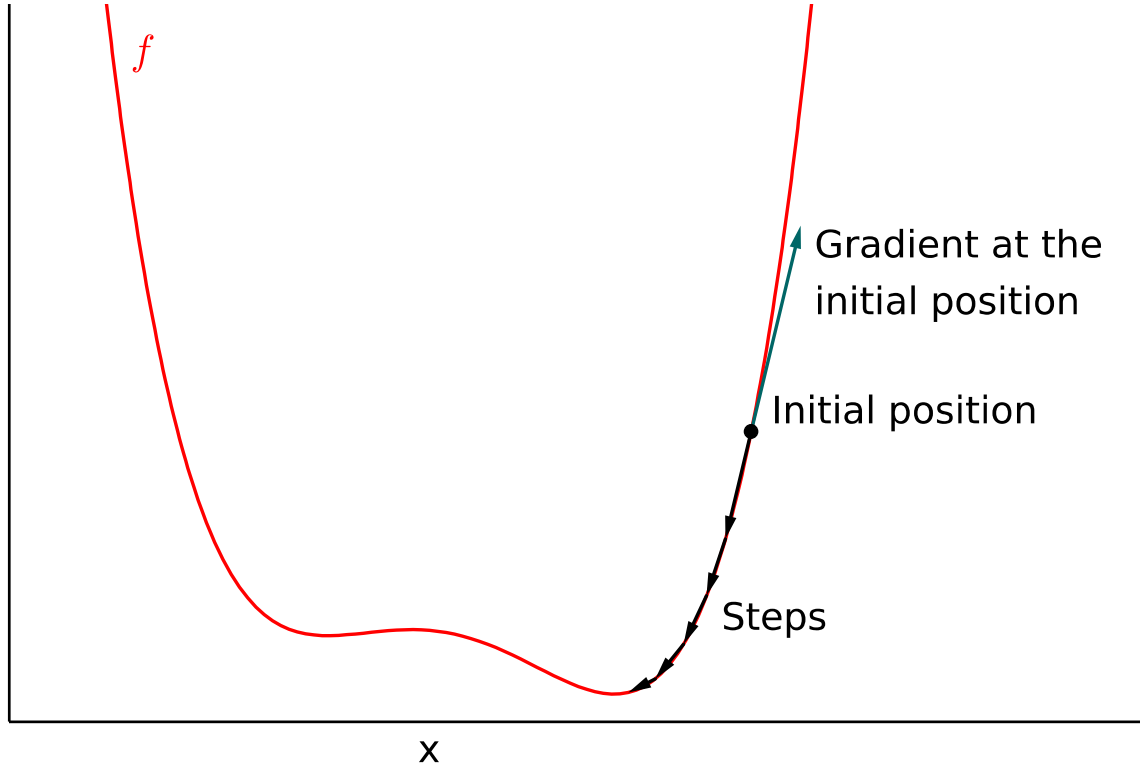
Cross entropy loss: $\mathscr{L}(X) = -D_X \odot \ln(A_X) - (1 - D_X) \odot \ln(1 - A_X)$

where $\odot$ designate the Hadamard product.

Gradient descent:

Gradient descent is a method used to find the minimum of a differentiable function $f : \mathbb{R}^n \to \mathbb{R}$. At each step, we apply $\Delta x = -\eta \frac{\partial f}{\partial x}$ for some $\eta > 0$.

The smaller η is, the slower but more precise is the convergence.



Backpropagation:

We will use the stochastic gradient descent: for each input X of the dataset, gradient descent is applied on every layer's weights and biases, for the loss $\mathscr{L}(X)$ and a learning rate $\eta > 0$. That is:

$$\Delta W^l = -\eta \frac{\partial \mathscr{L}(X)}{\partial W^l} \qquad \Delta B^l = -\eta \frac{\partial \mathscr{L}(X)}{\partial B^l}$$

One can show by computation that this leads to:

$$\Delta W^l = -\eta \left(G^l\right)^t X^{l-1} \qquad \Delta B^l = -\eta G^l$$

where $G^l := \frac{\partial \mathscr{L}(X)}{\partial S^l}$. For every $l < L$, we can compute G^l recursively:

$$G^l = f_l'(S^l) \odot (G^{l+1} W^{l+1})$$

And G^L depends on the choice of loss funtion:

$G^L = f_l'(S^L) \odot (A_X - D_X)$ for the quadratic loss.

$G^L = A_X - D_X$ for the cross entropy loss.