Artificial Intelligence

academic year 2025/2026

Giorgio Fumera, Ambra Demontis

Pattern Recognition and Applications Lab
Department of Electrical and Electronic Engineering
University of Cagliari (Italy)

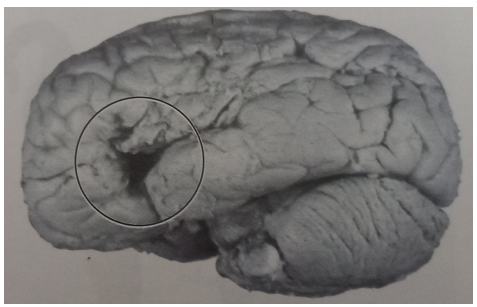
Machine learning

Artificial Neural Networks

Artificial Neural Networks: historical notes

19th-early20th cent.: Inspired by findings in neuroanatomy (structure) and neurophysiology (electrical activity).

- Paul Broca (1861): Different regions of the brain control different functionalities.



Brain of a person who had lost the capability to talk.
Source: *Corsi, 1991 Fig III*

- Thanks to the Golgi's stain and the electronic microscope (around 1930): The neuron is the fundamental unit that constitutes the brain.

Source: *Bear F. M., et al., M. Neuroscience. Exploring the brain.*

Artificial Neural Networks: historical notes

- 1943: McCulloch and Pitts' model of neurons as "logic units"
- 1949: Hebb's model of changes in "synaptic strength" and *cell assemblies* as the origin of adaptation, learning and "thinking"
- 1957: Rosenblatts' **perceptron**:
  - "training procedures" for adjusting connection weights
  - applications to pattern recognition: "error correction" training procedure (perceptron learning algorithm)

Artificial Neural Networks: historical notes

- 1970s: limits of perceptrons are pointed out (low expressiveness, lack of mathematical rigor), causing a drop of interest in neural networks (M.L. Minsky and S.A. Papert, *Perceptrons*, MIT Press, 1969)
- Mid–1980s: "renaissance" of neural networks, or the **connectionist** approach
  - an efficient learning algorithm: **back-propagation**
  - **theoretical support** from statistics and computational learning theory
  - seminal works:
    - D.E. Rumelhart, J.L. McClelland (Eds.), *Parallel Distributed Processing*, MIT Press, 1986
    - D.E. Rumelhart, G.E. Hinton, R.J. Williams, *Learning representations by back-propagating errors*, Nature 323, 533-536, 1986

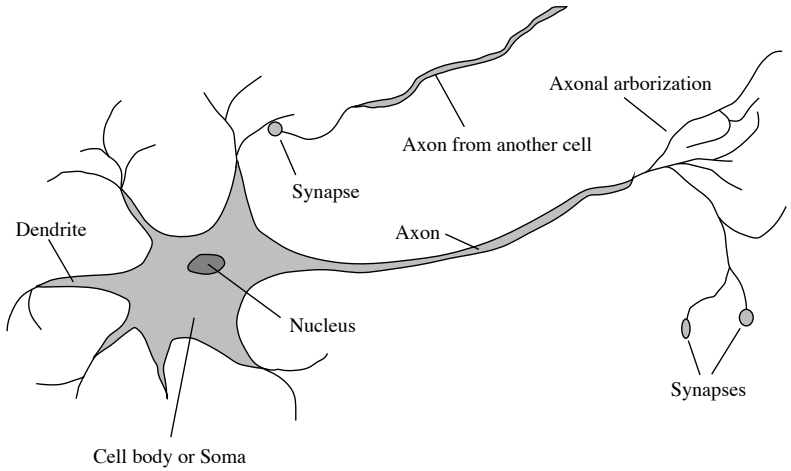
Artificial Neural Networks: historical notes

Since the 1990s:

- different kinds of ANNs:
  - Feed-Forward Multi-Layer networks
  - Radial Basis networks
  - recurrent networks: Hopfield networks, associative memories
  - Boltzmann machines
- applications in several fields: computer vision, pattern recognition, control systems, etc.
- main current trend: **deep** neural networks, and in particular **convolutional** networks, originally devised for *image* processing, **transformers**, employed for *natural language processing*
- Outstanding performance thanks to:
  - Large datasets
  - Powerful GPUs

Brains and neurons: some facts

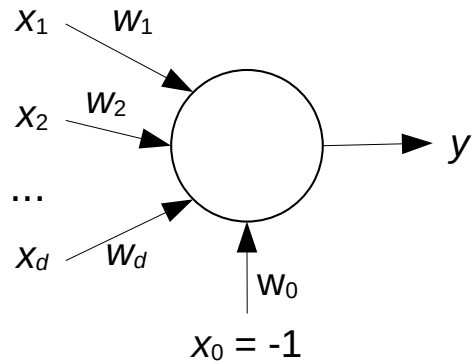
Basic elements: nerve cells called **neurons**



- 10^{11} neurons, 10^{15} connections between them (in total)
- apparently simple neuron behavior: "firing" (up to 10^3 Hz) in response to specific **patterns** of input signals, modulated by **excitatory** and **inhibitory** connections
- massive parallelism
- robustness to "noise"
- learning capability in response to external stimuli: novel connections between neurons, changing connection "strength"

McCulloch and Pitts' model of neurons

First mathematical model of neuron's behaviour: the "logic unit" model by McCulloch and Pitts (1943)



- **input** signals: $x_1, \ldots, x_d \in \{0, 1\}$, plus a **fictitious** input $x_0 = -1$
- connection **weights** $w_0, \ldots, w_d \in \mathbb{R}$: their values and signs model the excitatory or inhibitory effect
- perceptron **input**: **weighted sum** of its input signals:

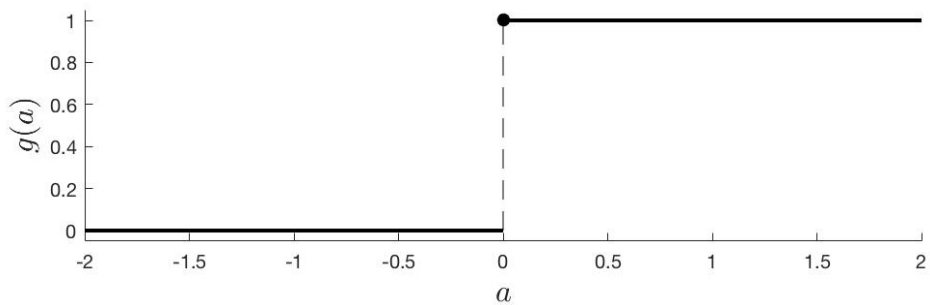
$$a(\mathbf{x}, \mathbf{w}) = \sum_{i=0}^{d} w_i x_i,$$

  where $\mathbf{x} = (x_1, \ldots, x_d)$, $x_0 = -1$, $\mathbf{w} = (w_0, \ldots, w_d)$
- **output** signal $y \in \{0, 1\}$, called **activation**: $y = g(a)$, where g is called **activation function**

McCulloch and Pitts' model of neurons

The activation function is defined as the **Heaviside (step) function**, or threshold function:

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



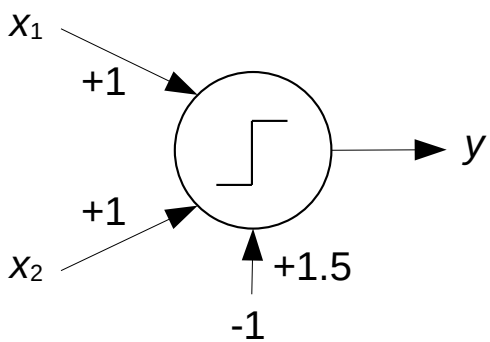
In words, the neuron "**fires**" (it outputs a 1), if the weighted sum of its input signals exceeds a threshold given by the bias weight, i.e., if $\sum_{i=1}^{d} w_i x_i \geq w_0$, otherwise it does not "fire" (it outputs a 0).

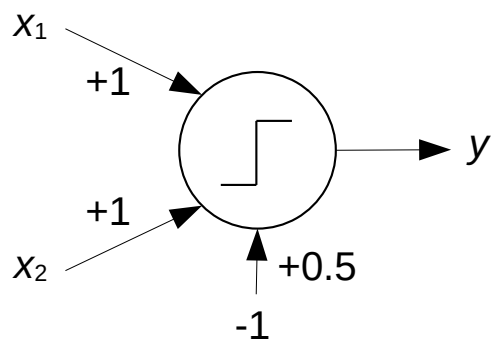
McCulloch and Pitts' model of neurons

For instance, McCulloch and Pitts' model with $d = 2$ inputs x_1 and x_2 can behave as an `AND` or `OR` logic gate:

- `AND`: one can set, e.g., $w_1 = w_2 = 1$, and $w_0 = 1.5$: this way the activation $w_1 x_1 + w_2 x_2 - w_0$ is non-negative (thus $y = 1$), only when $x_1 = x_2 = 1$ (see the figure below on the left)
- `OR`: w_1 and w_2 can still be set to 1, whereas w_0 can be set, e.g., to 0.5: (see the figure below on the right)

Note however that there are **infinite** combinations of values of the connection weights producing the **same** results (e.g., $w_1 = 3$, $w_2 = 2$ and $w_0 = 4$ for the `AND` logic gate).





The idea of *artificial* neural networks

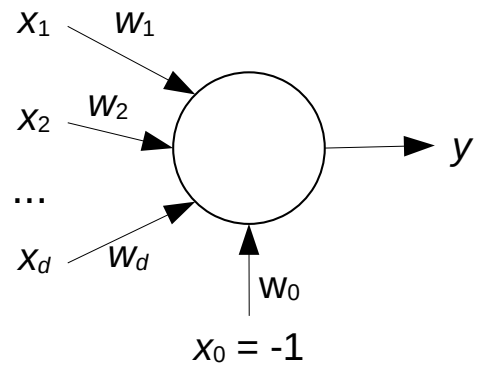
Neurons in human and animal brains are organised in complex structures. The job of some structures is to **perceive** or **recognise** "patterns" in sensory inputs (e.g., enabling sight from signals coming from cells in the retina), and to "learn" to recognise **novel** patterns by changing their connections or connection strength.

McCulloch and Pitts' model inspired the idea of building hardware or software structures, named **artificial neural networks**, emulating some aspects of brain organisation and workings, in particular the capability of "learning" to perform a recognition task by modifying their structure or parameters based on the analysis of **examples** of the desired input-output behaviour.

McCulloch and Pitts' model of **real** neurons turned out to be oversimplified, but it proved to be useful as a basic building block for **artificial** neural networks in AI. More accurate models are still being developed by neuroscientists, but their complexity is too high for AI applications.

The *perceptron*

One of the first attempts of defining an artificial neuron capable of performing **pattern recognition** tasks was the **perceptron** by Frank Rosenblatt (1957), inspired by McCulloch and Pitts' model of neuron.



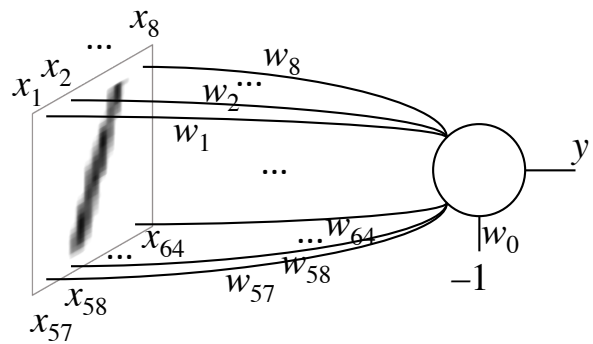
The only difference from McCulloch and Pitts' model is that the inputs can be **real** numbers, instead of the binary values $\{0, 1\}$.

Since the perceptron's output can take only two values (0 and 1), it can be used to recognise **two** different classes of input values.

The *perceptron*

For instance, a perceptron could be used for image recognition tasks involving two classes, e.g., in OCR, to discriminate between images of "zeros" and of "ones".

Considering gray-level images of size 8×8 pixels, with 8 bits per pixel (i.e., 256 grey levels in $\{0, \ldots, 255\}$), the values of the 64 pixels can be used as the perceptron's inputs $x_1, \ldots, x_{64}$.

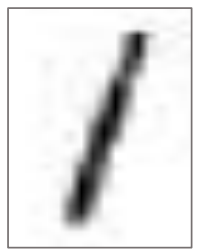


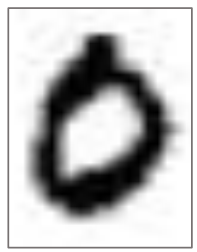
After choosing the desired values of the perceptron output (e.g., $y = 1$ when the input image represents a "one", and $y = 0$ otherwise), one should find suitable values of the connection weights $w_0, w_1, \ldots, w_{64}$ capable of producing the corresponding input–output behaviour.

The *perceptron*

An alternative choice for the input values is to use a set of measures (attributes or features) computed from the input image, which are deemed to be discriminant between the classes of interest.

As a simple example, one can consider that "zeros" are likely to be made up of a larger number of pixels and to be wider than "ones", as in the figures on the right.



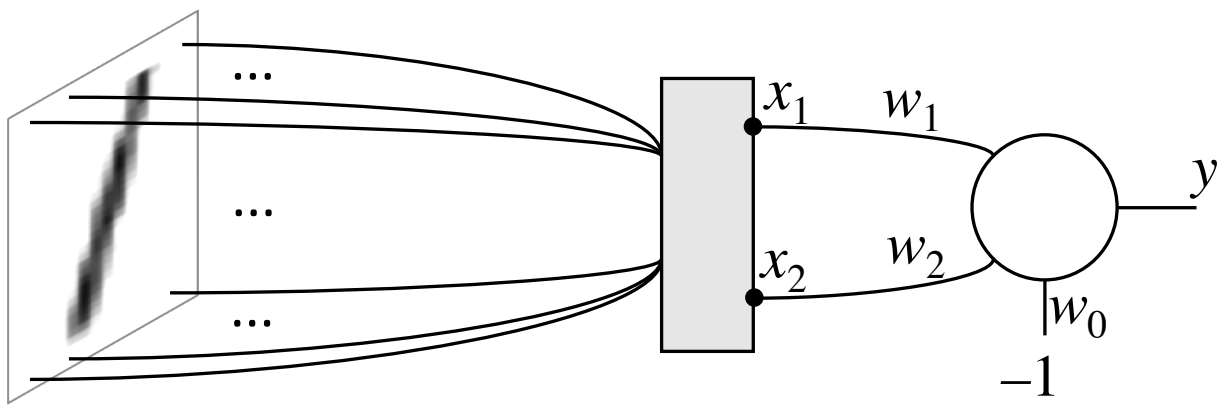


Accordingly, one may use:

- x_1: the **fraction of foreground pixels**, i.e., the number of pixels whose value exceeds a given binarisation threshold, divided by the total number of pixels, 64
- x_2: the **relative character "width"**, i.e., the number of image columns that contain at least one foreground pixel divided by the total number of columns, 8

The *perceptron*

The computation of x_1 and x_2 can be represented by a **feature extraction** module acting on the input image:



The perceptron as a *learning machine*

As mentioned above, experience and learning produce changes (among other things) on the **connection strength** between neurons in the human brain.

Can the perceptron **emulate** this behaviour?

In other words, can the **connection weights** of a perceptron be modified by a **learning algorithm** to perform a recognition (supervised classification) task involving two categories of inputs, based on a **training set** of **labelled** examples, whose **attributes** correspond to the perceptron's **inputs**?

The perceptron learning algorithm

A learning algorithm for the perceptron was devised by F. Rosenblatt in 1960.

Its goal is to find the connection weights that **minimise** the number of misclassifications on a given training set $\mathcal{T}$, possibly achieving zero errors, i.e., **consistency** with $\mathcal{T}$.

Basically, the perceptron learning algorithm starts by **randomly** setting the connection weights, then it iterates several times over the training examples, and whenever an example is found to be **misclassified** by the **current** perceptron, the connection weights are **updated** in an **attempt** to correctly classify it, until some stopping criterion is met.

The decision regions of a perceptron

To better understand how the perceptron learning algorithm works, it is useful to look at the kind of **decision regions** produced by a perceptron in attribute (input) space, for **fixed** values of the connection weights.

To this aim, remember that the perceptron **activation function** is defined as:

$$y = \begin{cases} 1, & \text{if } \sum_{i=0}^{d} w_i x_i \geq 0 \\ 0, & \text{if } \sum_{i=0}^{d} w_i x_i < 0 \end{cases}$$

Where $x_0 = -1$.

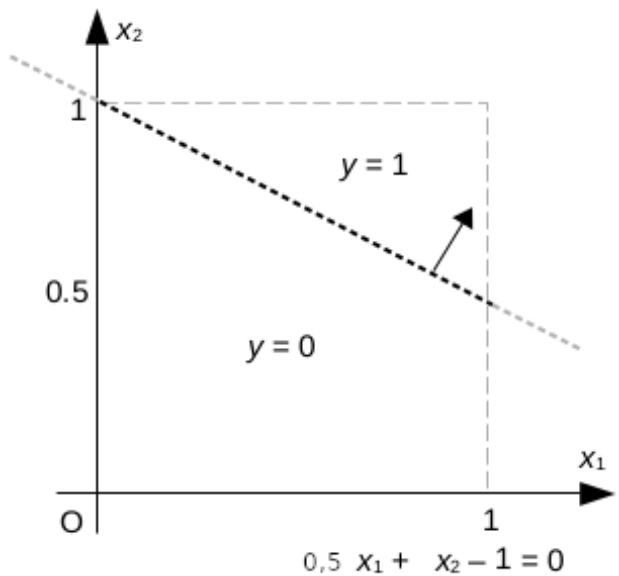
This corresponds to a **linear discriminant function** in attribute space, i.e., a **hyperplane** whose equation is $\sum_{i=0}^{d} w_i x_i = 0$.

The decision regions of a perceptron

As an example, consider a classification problem with two attributes ($d = 2$) with values in $[0, 1]$, and a perceptron with connection weights $w_1 = 0,5$, $w_2 = 1$, $w_0 = 1$, whose output is defined by:

$$y = \begin{cases} 1, & \text{if } 0,5x_1 + x_2 - 1 \geq 0 \\ 0, & \text{if } 0,5x_1 + x_2 - 1 < 0 \end{cases}$$

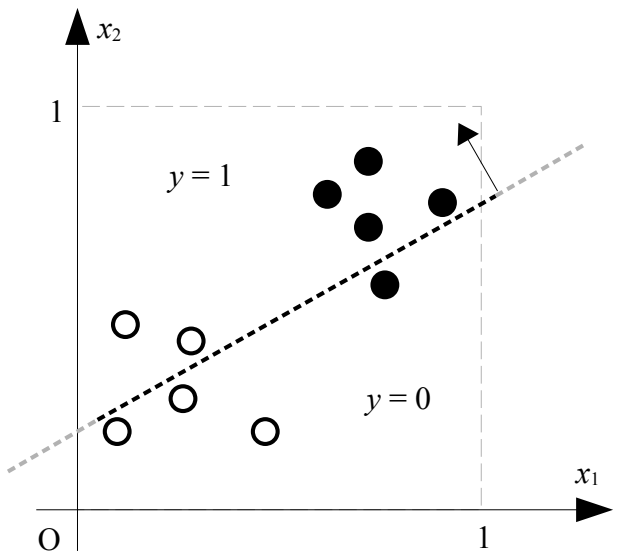
The corresponding decision regions are shown in the figure on the right, where the small arrow points toward the region where the perceptron output equals 1.



The decision regions of a perceptron

For instance, consider the two possible features mentioned above for a two-class handwritten digit recognition task, "0" vs "1": **fraction of foreground pixels** (x_1) and **relative character "width"** (x_2).
The figure below is an example of a training set made up of five instances (images) from each class ($\circ$ = "1", $\bullet$ = "0"), plotted in the feature space, and the decision regions produced by a perceptron with **random** weight values.

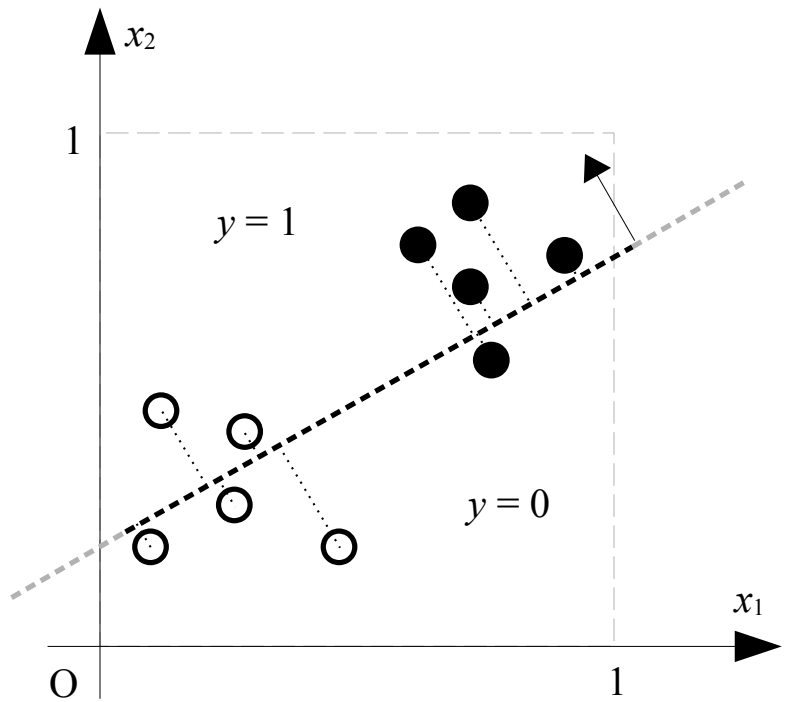
Assuming that the desired perceptron output is 1 for the class "1" and 0 for the class "0", this perceptron misclassifies three instances of class "1" and four instances of class "0".



The decision regions of a perceptron

Since the boundary of the decision regions implemented by a perceptron is the hyperplane h defined by $w_1x_1 + w_2x_2 + \ldots + w_dx_d - w_0 = 0$, for any given point $\mathbf{x} = (x_1, \ldots, x_d)$ in the feature space the absolute value of the the **input**, $|a(\mathbf{x}, \mathbf{w})| = |w_1x_1 + w_2x_2 + \ldots + w_dx_d - w_0|$, is **proportional** to the distance of $\mathbf{x}$ to h.

The figure on the right shows the same plot of the previous example, with the distance of each training instance to the decision boundary highlighted.



The *error function* of the perceptron learning algorithm

The weight update rule of the perceptron learning algorithm is based on the idea of defining an **error function** *that evaluates the extent to which a misclassified training example is "far" from being correctly classified.*

The error function is defined in such a way that it is **positive** and **proportional** to the **distance** in feature space of a misclassified example $\mathbf{x}$ to the perceptron's decision boundary: the underlying intuition is that the larger the distance, the larger the amount by which the connection weights have to be modified to get $\mathbf{x}$ correctly classified.

The *error function* of the perceptron learning algorithm

The **error function**, for a given **misclassified** example $\mathbf{x}$ and for given values of the connection weights $\mathbf{w} = w_0, \ldots, w_d$, is defined as:

$$E(\mathbf{x}, \mathbf{w}) = -t \times a(\mathbf{x}, \mathbf{w}) = -t \times (w_1 x_1 + w_2 x_2 + \ldots + w_d x_d - w_0) \ ,$$

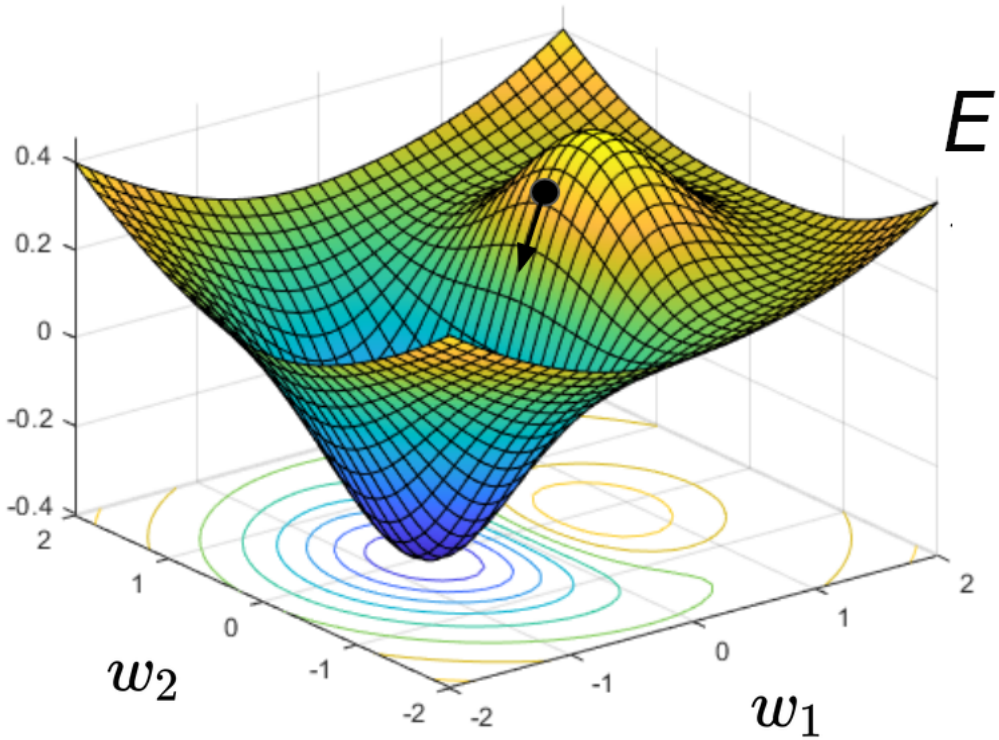
where $t = +1$ if the desired perceptron output for $\mathbf{x}$ is 1, and $t = -1$ if the desired output is 0.

It is easy to see that, if $\mathbf{x}$ is misclassified, then $E(\mathbf{x}, \mathbf{w}) > 0$:

- if $t = +1$ and $\mathbf{x}$ is misclassified, then the input $a(\mathbf{x}, \mathbf{w})$ is negative, and therefore $E(\mathbf{x}, \mathbf{w}) = -t \times a(\mathbf{x}, \mathbf{w}) > 0$
- if $t = -1$ and $\mathbf{x}$ is misclassified, then the input $a(\mathbf{x}, \mathbf{w})$ is positive, and therefore $E(\mathbf{x}, \mathbf{w}) = -t \times a(\mathbf{x}, \mathbf{w}) > 0$

The *error function* of the perceptron learning algorithm

To **reduce** the error function for a misclassified example, the **gradient descent** approach is used:



The *error function* of the perceptron learning algorithm

To **reduce** the error function for a misclassified example, the **gradient descent** approach is used:

$$w_i \leftarrow w_i - \eta \frac{\partial E(\mathbf{x}, \mathbf{w})}{\partial w_i}, \quad i = 0, \ldots, d \ ,$$

where η is an **arbitrary**, **positive** constant, usually called **learning rate**.

From the definition of the error function, it is easy to see that its partial derivatives with respect to the connection weights are given by:

$$\frac{\partial E(\mathbf{x}, \mathbf{w})}{\partial w_i} = \begin{cases} -t \times x_i, & i = 1, \ldots, d \\ t, & i = 0 \end{cases}$$

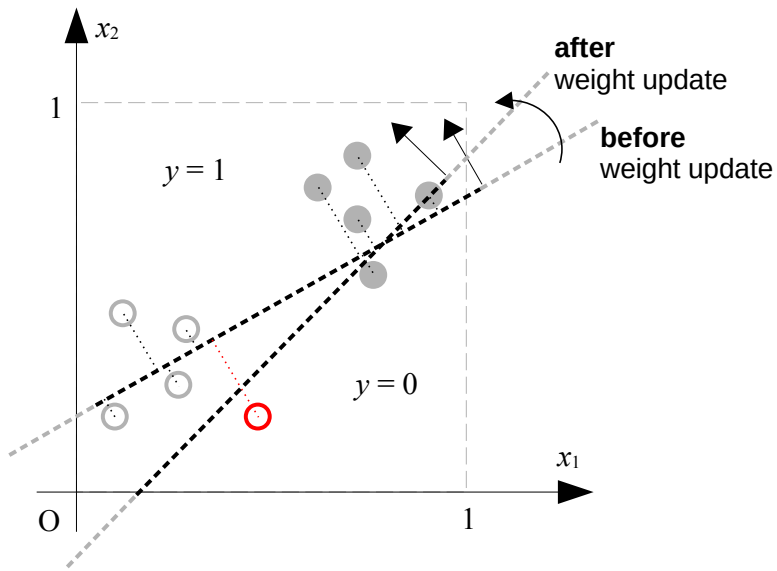
The **weight update rule** is therefore:

$$\begin{aligned} w_i &\leftarrow w_i + \eta x_i t, \quad i = 1, \ldots, d \\ w_0 &\leftarrow w_0 - \eta t \end{aligned}$$

Pseudo-code of the perceptron learning algorithm

Note that the above weight update rule "moves" the perceptron decision boundary in feature space toward correctly classifying a previously misclassified example **x**, but does **not** guarantee that **x** gets correctly classified after a **single** weight update.

The figure on the right shows an example of how the decision boundary may change after the weight update corresponding to the misclassified example highlighted in red:



For the above reason, the perceptron learning algorithm **repeatedly** iterates over the whole training set $\mathcal{T}$, updating the connection weights for each misclassified example.

Pseudo-code of the perceptron learning algorithm

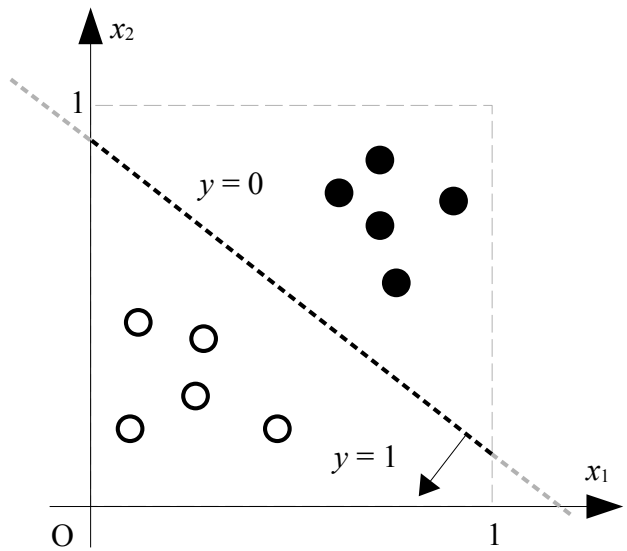
function PERCEPTRON-LEARNING ($\mathcal{T}$)
returns weight values **w**
    randomly choose the initial weight values **w**
    repeat
        for each $(\mathbf{x}, t) \in \mathcal{T}$ **do**
            if $E(\mathbf{x}; \mathbf{w}) > 0$
            then $w_i \leftarrow w_i - \eta \frac{\partial E(\mathbf{x};\mathbf{w})}{\partial w_i}, \ i = 0, \ldots, d$
        end for
    until a stopping condition is satisfied
    return w

Each `for` loop over training examples is named **epoch**.

The perceptron learning algorithm: convergence

If the examples of the two classes in the training set are **linearly separable**, then the perceptron learning algorithm **always** converges to a **consistent** solution after a **finite** number of epochs, for **any** $\eta > 0$ (F. Rosenblatt, 1960).

An example of a possible solution provided by the perceptron learning algorithm for linearly separable classes:

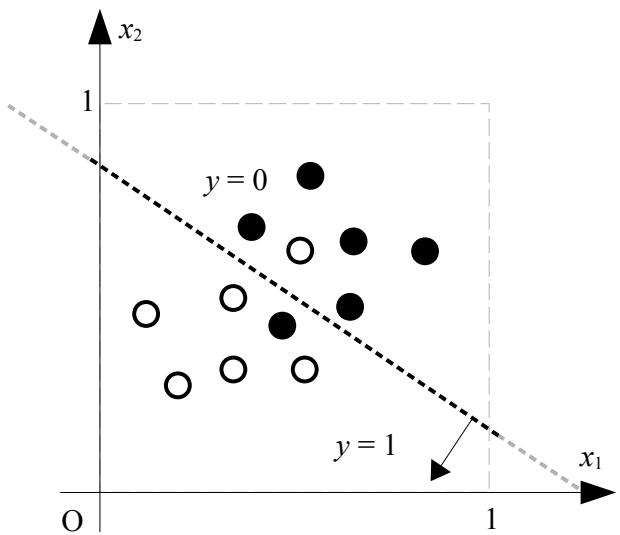


Since convergence is guaranteed (for linearly separable classes) for any $\eta > 0$, usually $\eta = 1$ is chosen.

The perceptron learning algorithm: convergence

If $\mathcal{T}$ is **not** linearly separable, the perceptron learning algorithm proceeds toward a solution "close" to the minimum possible number of misclassified examples (**without** guaranteeing the actual minimum), but **does not** converge to a fixed solution: in this case the number of misclassified examples tends to remain constant or to slightly "oscillate" around a given value.

An example of a possible solution provided by the perceptron learning algorithm for a **non-linearly separable** training set:



The perceptron learning algorithm: convergence

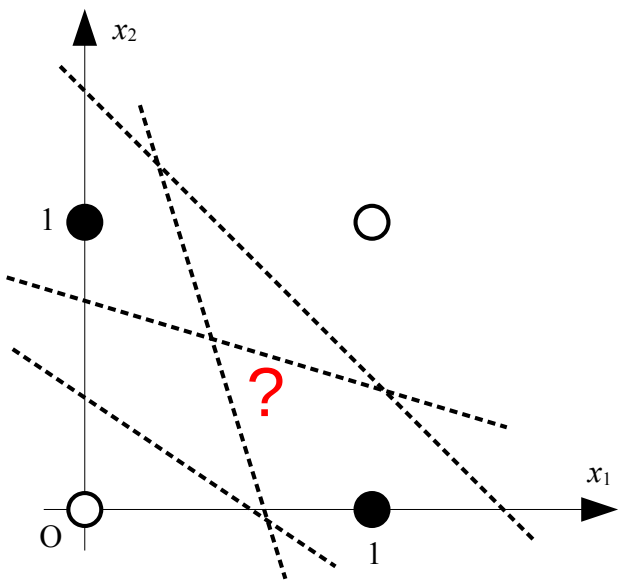
In practice, it is not possible to determine **in advance** whether a given $\mathcal{T}$ is linearly separable, especially if the number of attributes and of training examples is very high.

Accordingly, as the **stopping condition** of the perceptron learning algorithm, one can consider setting a maximum number of epochs or detecting that the number of misclassified examples remains almost constant for a certain number of consecutive epochs.

Limitations of the perceptron

We have seen that a perceptron can represent Boolean functions such as `AND` and `OR`.

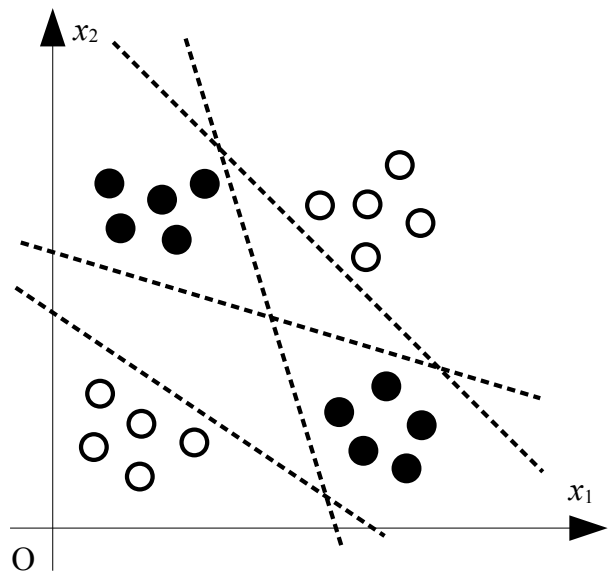
However, it cannot represent a Boolean function like `XOR`. This can be graphically proven as shown below, taking into account that no **line** can separate points $(1, 0)$ and $(0, 1)$, where the perceptron output should equal 1, from points $(0, 0)$ and $(1, 1)$, where the perceptron output should equal 0:



Limitations of the perceptron

In the 1970's the perceptron was found to have a too limited expressive capability for many real-world classification tasks, whose class distribution in attribute space can be **highly non-linear**: in this case the **linear** class boundary implemented by a perceptron cannot guarantee an acceptably small misclassification rate.

A toy-example for two classes with a "XOR-like" distribution in attribute space: it is not difficult to see that **any** linear boundary will misclassify **at least** one fourth of the training examples (e.g., all the examples in one of the four clusters):

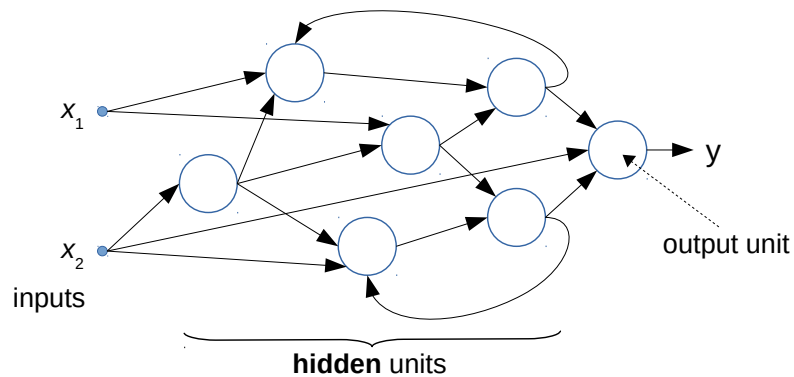


Perceptron networks

Neurons in the human brain are organized into complex and highly interconnected networks, which enables them to produce very complex "input-output" behaviors.

To mimic them, **artificial neural networks** (ANN) made up of interconnected perceptrons (possibly with recurrent connections) may be used: it is not difficult to see that they can represent **non-linear** discriminant functions in attribute space.

An example: a perceptron network with two inputs, one output, several "internal" or **hidden** units (i.e., units whose output is an input signal of other perceptrons), and some **recurrent** connections (bias inputs are not shown, for the sake of simplicity).



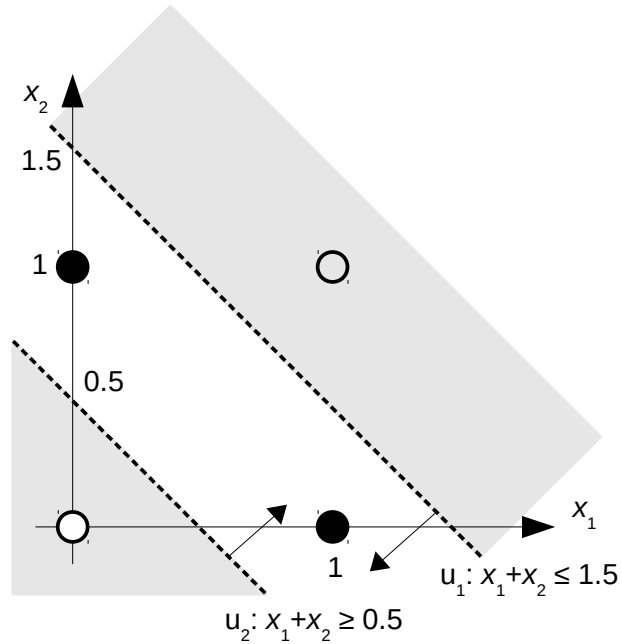
Perceptron networks: an example

To create a deep neural network that correctly classify the training examples shown at page 31 one can first reason about the decision function it should have.

Perceptron networks: an example

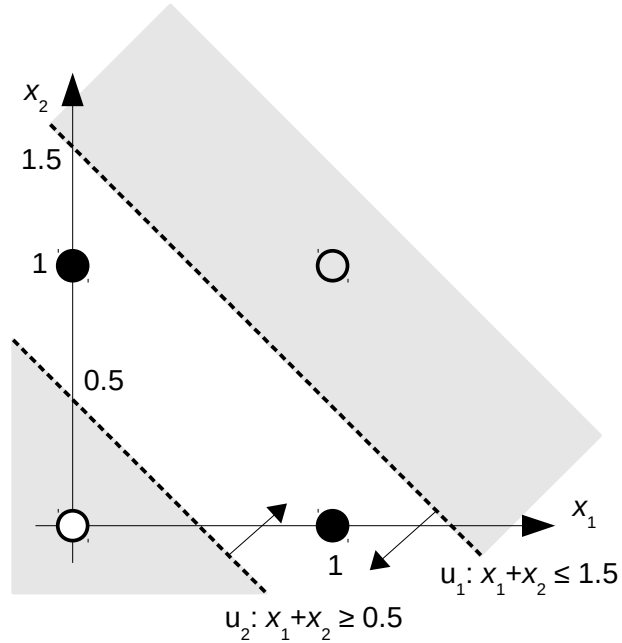
To create a deep neural network that correctly classify the training examples shown at page 31 one can first reason about the decision function it should have.

Then, they can reason about the architecture that can create that decision function.

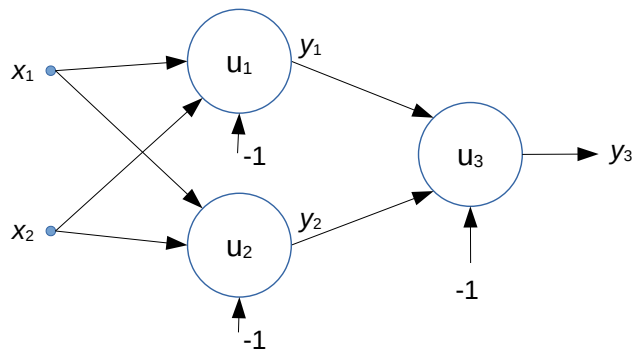


Perceptron networks: an example

To create a deep neural network that correctly classify the training examples shown at page 31 one can first reason about the decision function it should have.

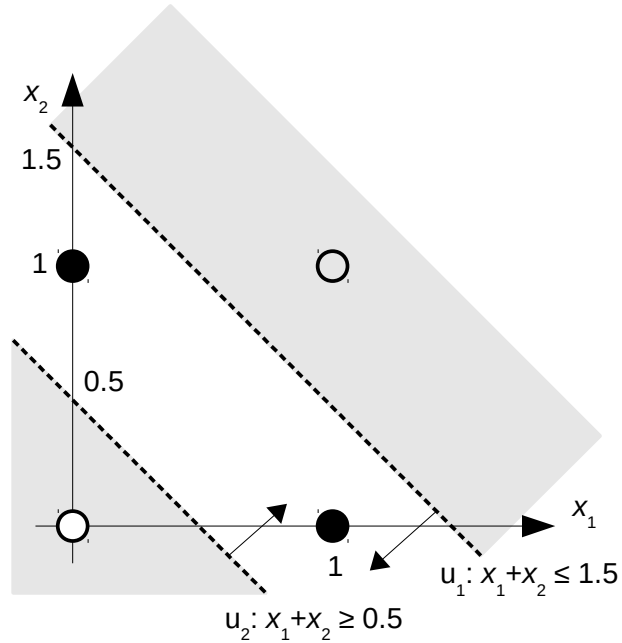


This decision function can be created with the following network architecture:

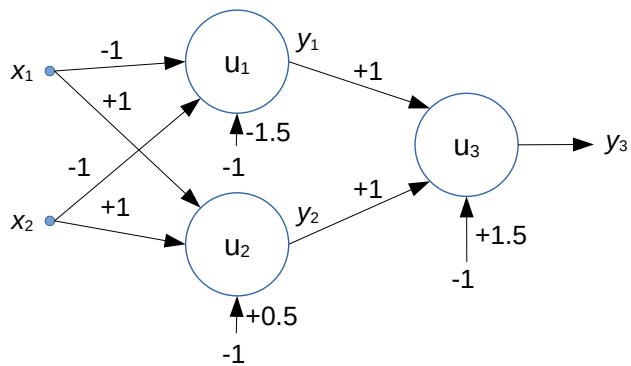


Perceptron networks: an example

Finally, one can set the weight values of each unit such that they implement the linear discriminant functions shown on the right.

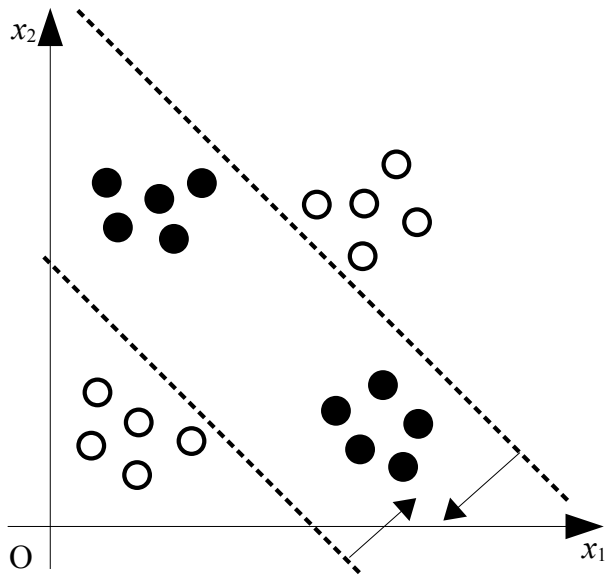


This perceptron network can represent the XOR Boolean function.



Perceptron networks: an example

A similar solution can be adopted to build a two-class classifier using a perceptron network for XOR-like class distributions:



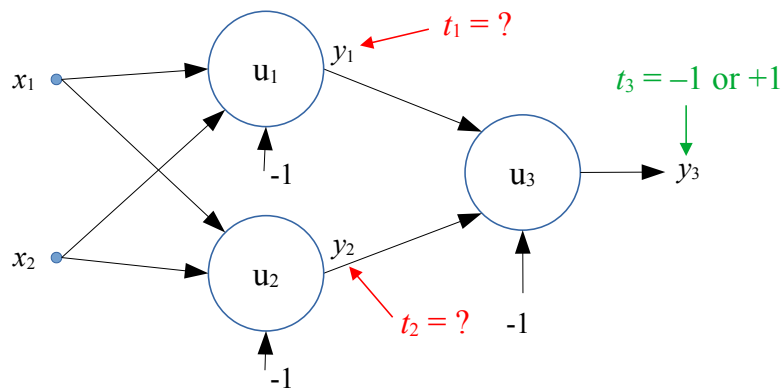
Issues of perceptron networks

Perceptron networks have a higher expressive capability than single perceptrons. However, it is not possible to adopt the **perceptron learning algorithm** to set the connection weights of their units.

Indeed, the **error function** of the perceptron learning algorithm requires the definition of the **desired output** for each training example, represented by $t \in \{-1, +1\}$:

$$E(\mathbf{x}, \mathbf{w}) = -t \times a(\mathbf{x}, \mathbf{w})$$

The desired output can be defined for the **output units** of a network, but there is no way to define it for **hidden units**:



Issues of perceptron networks

Alternative learning algorithms were devised in the 1970s, but they exhibited a too high **computational complexity**.

Another issue is how to define a suitable network **architecture**, i.e., the number of units and the connections between them, for a given application. Can the architecture itself be chosen by the learning algorithm, together with connection weights?

These difficulties contributed to a **drop of interest** in ANNs in the 1970s.

The renaissance of artificial neural networks

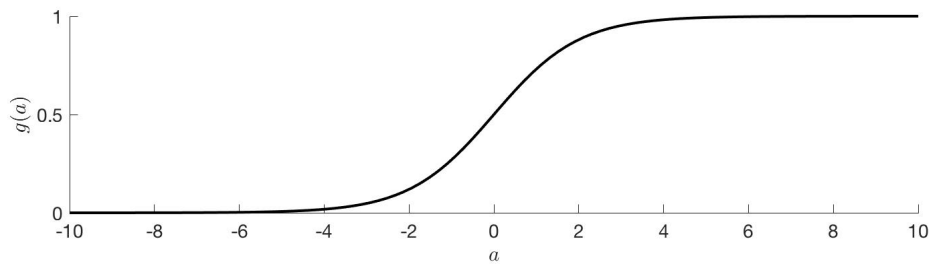
A practical solution to the above issues was found in the 1980s:

- devising learning algorithms capable of defining also the network architecture is too difficult: it is better to use **predefined** architectures
- **efficient** and **effective** learning algorithms were devised for:
  - specific architectures, like **feed-forward** networks
  - **continuous** activation functions, instead of the Heaviside step function: this allows to use the **gradient descent**-like approach also for **hidden** units, without the need of defining a **desired output** for them

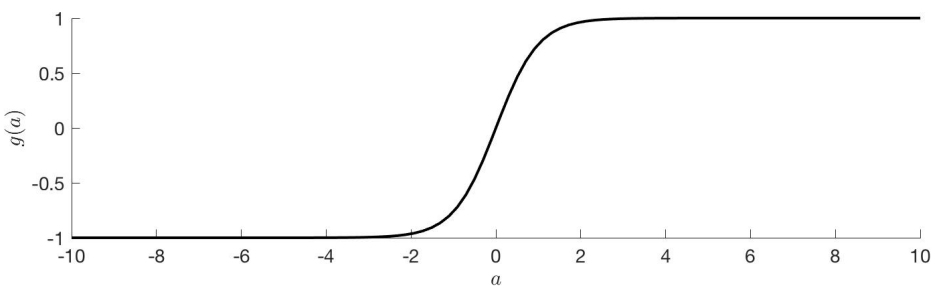
Continuous activation functions

Two widely used, **continuous** activation functions, which can be seen as approximations of the Heaviside function:

- **logistic function** (or "sigmoid"): $g(a) = \frac{1}{1+e^{-a}} \in (0, 1)$



- **hyperbolic tangent**: $g(a) = \tanh(a) = \frac{e^{a}-e^{-a}}{e^{a}+e^{-a}} \in (-1, 1)$



The feed-forward multi-layer architecture

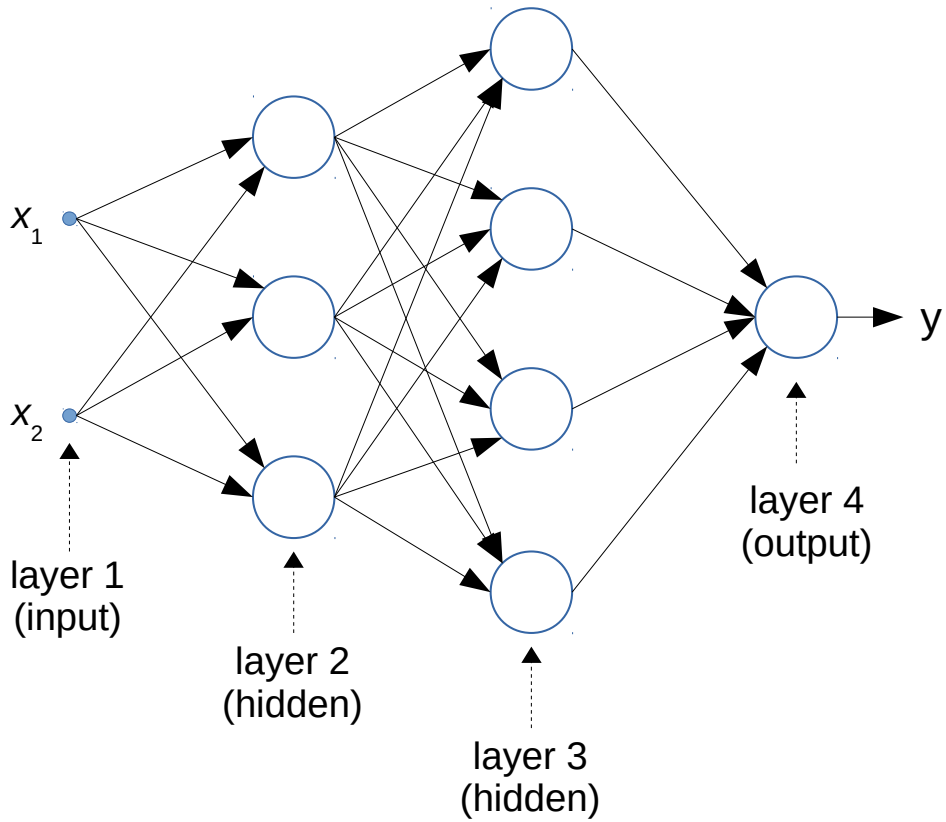
The most widely used architecture for supervised classification and regression problems is the **feed-forward multi-layer** (FF-ML):

- units are arranged into **layers**
  - **input** layer: input (e.g., attribute) values
  - **output** layer
  - one or more **hidden** layers
- **no** recurrent connections (hence the name "feed-forward")
- units of every layer (hidden and output layers) receive **inputs** only from units of the **previous** layer (hence the name "multi-layer")

Usually FF-ML networks are **fully-connected**: every input and hidden unit sends its output to **all** units of the **next** layer.

The feed-forward multi-layer architecture

An example of **fully-connected FF-ML network** with two inputs (x_1 and x_2), two hidden layers of three and four units each, and one output unit:



FF-ML networks as supervised classifiers

When FF-ML networks are used as supervised classifiers, the number of **output** units and their **desired outputs** are usually defined as follows:

- **two-class** problems: a **single** output unit whose desired output is $t \in \{0, +1\}$ (when the logistic activation function is used) or $t \in \{-1, +1\}$ (for the tanh activation function)
- **multi-class** problems ($m > 2$ classes): m output units $u_1, \ldots, u_m$, whose target values for training examples of the k-th class are $t_k = +1$, and $t_i = 0$ (or -1) for $i \neq k$ (**one-hot** encoding)

After training, the class label of any new instance $\mathbf{x}$ is defined as follows:

- **two-class** problems: label $+1$, if $y(\mathbf{x}) \geq 0.5$ (logistic function) or $y(\mathbf{x}) \geq 0$ (tanh); label 0 (or -1) otherwise
- **multi-class** problems: the label k^* corresponding to the **highest** output value: $k^* = \arg\max_{k=1,\ldots,m} y_k(\mathbf{x})$

Expressive capability of FF-ML networks

It has been shown that a FF-ML network with a **sufficient** number of hidden units with logistic or tanh activation function, and output units with **linear** activation function $g(a) = a$, can represent:

- any Boolean function, using **one** hidden layer
- any bounded and **continuous** function, with arbitrarily small approximation error, using **one** hidden layer
- any bounded, **discontinuous** function, with arbitrarily small approximation error, using **two** hidden layers

However, in the worst case an **exponential** number of hidden units is required, with respect to the number of inputs.

Expressive capability of FF-ML networks

In practical classification and regression problems it is not possible to determine beforehand the most suitable FF-ML architecture (i.e., the number of hidden layers and of hidden units), since the "target" function (i.e., the relationship between attribute values and the class label for classification, or the function to approximate for regression) is **unknown**, and only a finite set of examples is available.

Therefore, an iterative **trial-and-error** design approach is usually adopted, according to the Occam's razor principle (simpler hypotheses consistent with the observations are preferable to more complex ones):

- start with a **simple**, "small" FF-ML network, e.g., one hidden layer and a few hidden units
- if the (estimated) generalization capability is not satisfactory, consider a slightly more complex architecture (e.g., increase the number of hidden units, or add a hidden layer)

The *back-propagation* learning algorithm

An efficient learning algorithm was devised in the 1980s for FF ANNs with continuous and derivable activation function, including FF-ML ANNs: **back-propagation**.

Similarly to the perceptron learning algorithm, also the back-propagation algorithm is based on:

- defining an **error function** (or **loss function**) to evaluate the difference between the network **output** and the desired (target) output, for each training example
- a **gradient descent**-like procedure to reduce the value of the error function, starting by **random** initial weight values, by **updating** the connection weights through repeated **epochs** over the training set

Back-propagation: the error function

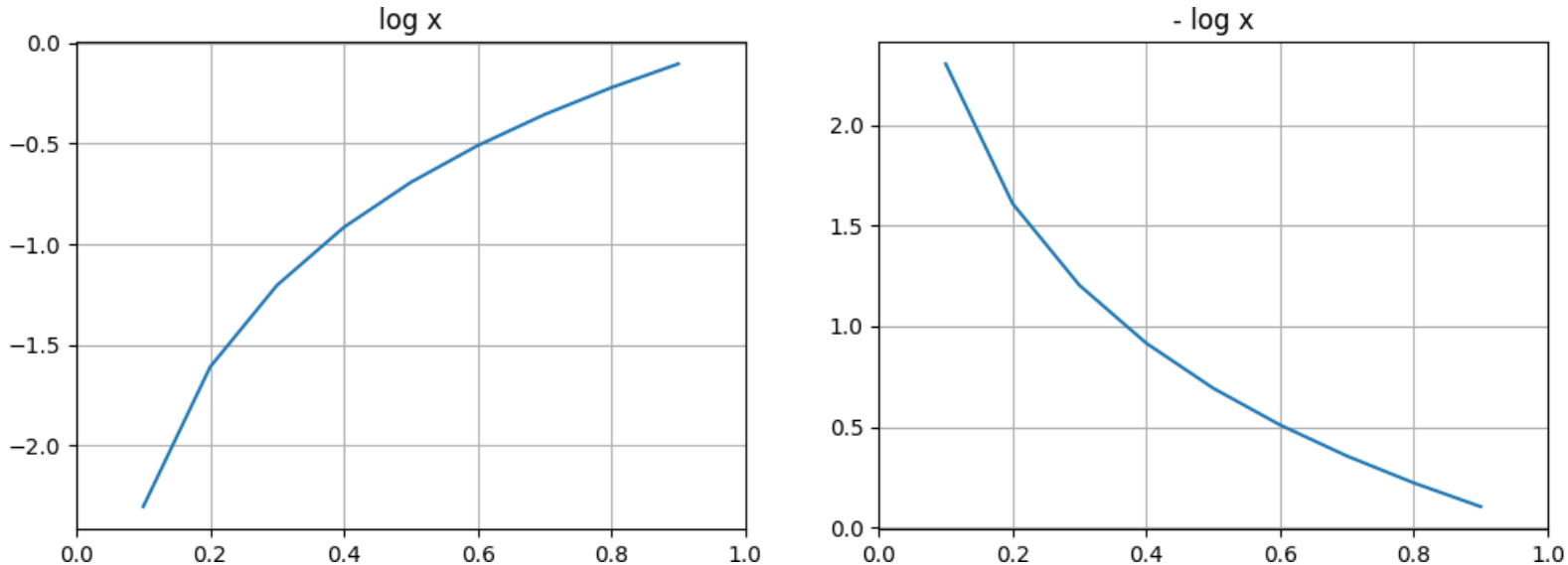
For **regression** problems, given a network with m output units (to be used to represent a function of m values), denoting with $\mathbf{y} = (y_1, \ldots, y_m)$ the vector of its output values, the typical error function for a training example with attribute vector $\mathbf{x}$ and desired outputs $\mathbf{t} = (t_1, \ldots, t_m)$ is the **squared error**:

$$E(\mathbf{y}, \mathbf{t}) = \frac{1}{2} \sum_{k=1}^{m} (t_k - y_k)^2$$

For **classification** problems the **cross-entropy** is preferable (for reasons rooted in the statistical setting of supervised classification problems):

$$E(\mathbf{y}, \mathbf{t}) = \begin{cases} -(t \log y + (1-t) \log(1-y)), & \text{for } m = 2 \text{ classes} \\ -\sum_{k=1}^{m} t_k \log y_k, & \text{for } m > 2 \text{ classes} \end{cases}$$

Back-propagation: minimising the loss function



Note that if the sample is classified with a confidence of 1, the value of -log 1 is equal to zero.

Back-propagation: minimising the loss function

Denoting the values of the connection weights by a vector $\mathbf{w}$, the goal of the back-propagation algorithm is to find the $\mathbf{w}^*$ that minimises the error function over the **whole** training set $\mathcal{T}$:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \sum_{(\mathbf{x},\mathbf{t})\in\mathcal{T}} E(\mathbf{y},\mathbf{t}) \ ,$$

where the network output $\mathbf{y}$ is a function of both $\mathbf{x}$ and $\mathbf{w}$.
Similarly to the perceptron learning algorithm, this problem **cannot** be solved analytically. A **gradient descent**-like approach is therefore used, which consists of updating **all** the connection weights, after computing the network output for each single training example $(\mathbf{x},\mathbf{t})$:

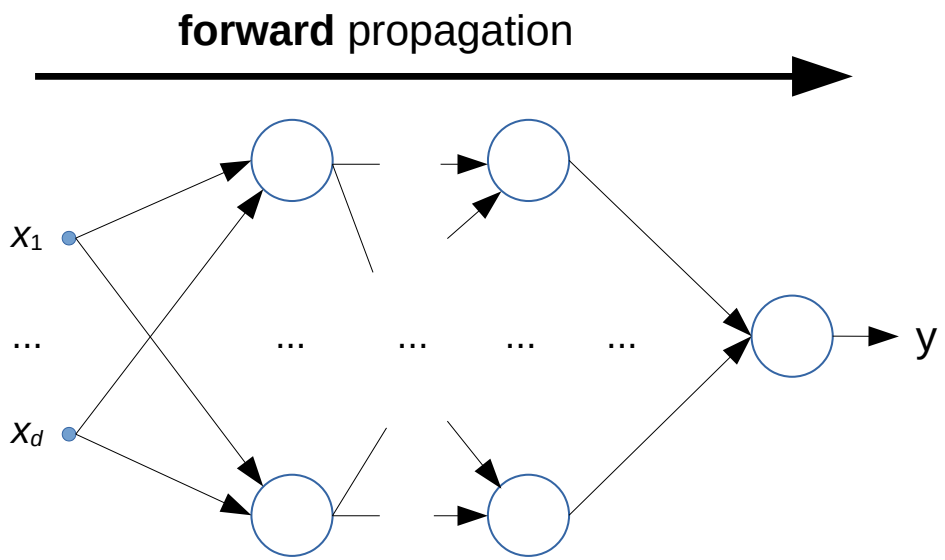
$$w \leftarrow w - \eta \frac{\partial E(\mathbf{y},\mathbf{t})}{\partial w}, \quad \text{for each } w \in \mathbf{w}$$

In the following, the back-propagation algorithm is described for a network with a **single** output unit, for the sake of simplicity.

Back-propagation algorithm: the *forward* pass

Given the **current** connection weights, for a **given** training example $(\mathbf{x}, \mathbf{t})$ it is first necessary to compute the network output $\mathbf{y}$.

This step is known as **forward propagation**, since the outputs of the units in the **first hidden layer** must be computed first, then the outputs of the units of the **second hidden layer** and so on, until the output unit:

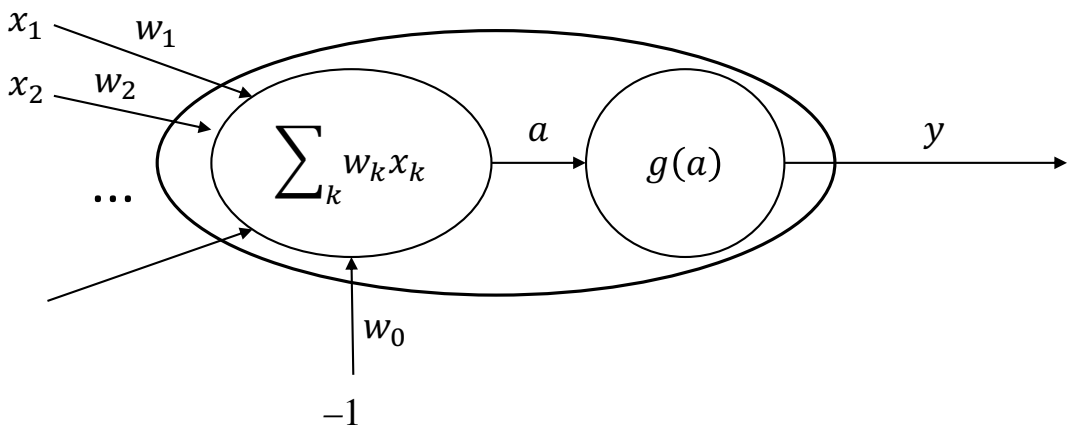


Back-propagation algorithm: the *forward* pass

The computations of the forward propagation step for any **single** unit is made up of two steps:

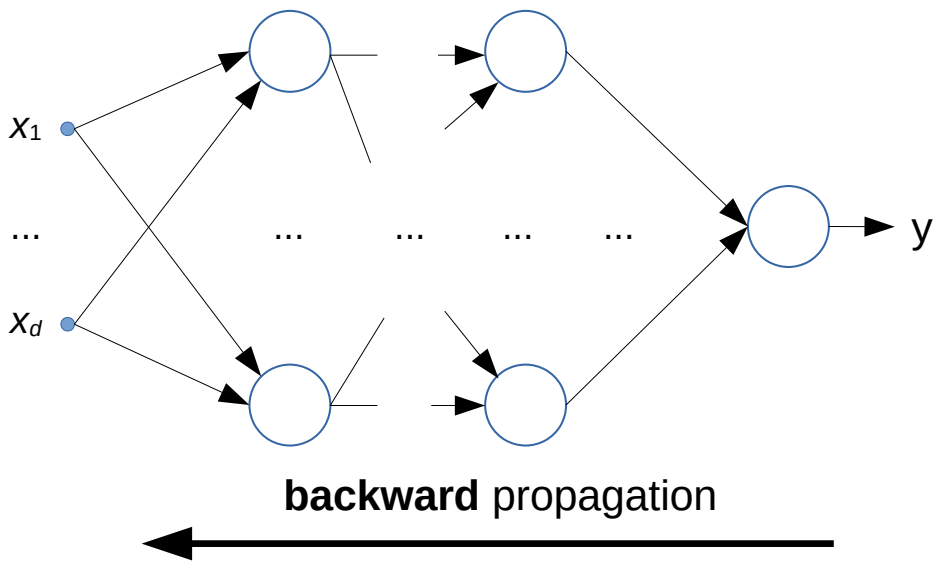
1. compute the unit's **input** a (weighted sum of its input values)
2. compute the unit's **output** y through the **activation function**, $y = g(a)$

For a generic unit the above steps can be represented as follows:



Back-propagation algorithm: the *backward* pass

Once the network output for a **single** training example has been computed, the partial derivatives of the error function can be computed starting from the connection weights of the **output** unit, then proceeding **backward** with the ones of the **last** layer of hidden units, and so on until the **first** layer of hidden units: hence the name of **back-propagation**.



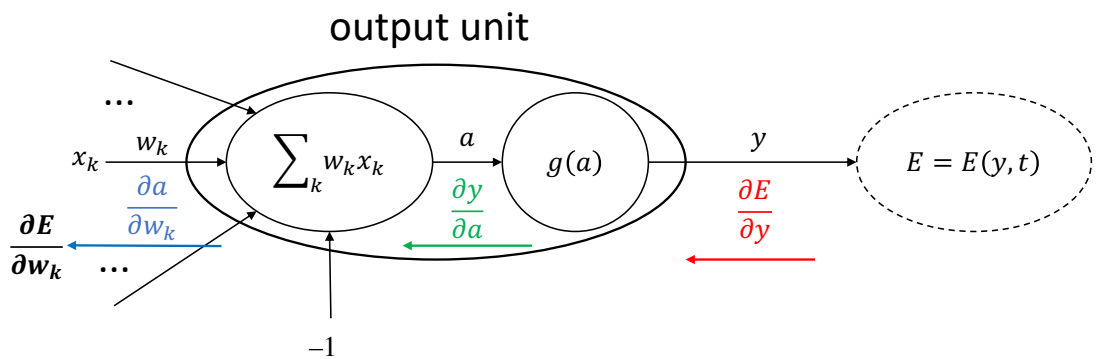
Back-propagation algorithm: partial derivatives

Consider first the **output** unit. The error function is computed on its output, y, and can be written as a **composition** of functions:

$$E(y,t) = E(g(a),t) = E\left(g\left(\sum_k w_k x_k\right), t\right)$$

The partial derivative for the weight of any input connection, w_k, can therefore be computed using the well known **chain rule**:

$$\frac{\partial E}{\partial w_k} = \frac{\partial E}{\partial y}\frac{\partial y}{\partial a}\frac{\partial a}{\partial w_k}$$



Back-propagation algorithm: partial derivatives

For instance, if $E(y, t)$ is defined as the **squared error**, $\frac{1}{2}(t-y)^2$, and the **logistic** activation function is used, $g(a) = (1+e^{-a})^{-1}$, the first two terms above can be easily computed as follows:

$$\begin{aligned} \frac{\partial E}{\partial y} &= -(t-y) \\ \frac{\partial y}{\partial a} &= \frac{\mathrm{d}g(a)}{\mathrm{d}a} = \frac{e^{-a}}{(1+e^{-a})^2} = g(a)[1-g(a)] = y(1-y) \end{aligned}$$

It is also easy to see that the third term is given by:

$$\frac{\partial a}{\partial w_k} = x_k$$

Back-propagation algorithm: partial derivatives

Therefore, for the **output** unit one obtains:

$$\frac{\partial E}{\partial w_k} = -(t - y)y(1 - y)x_k$$

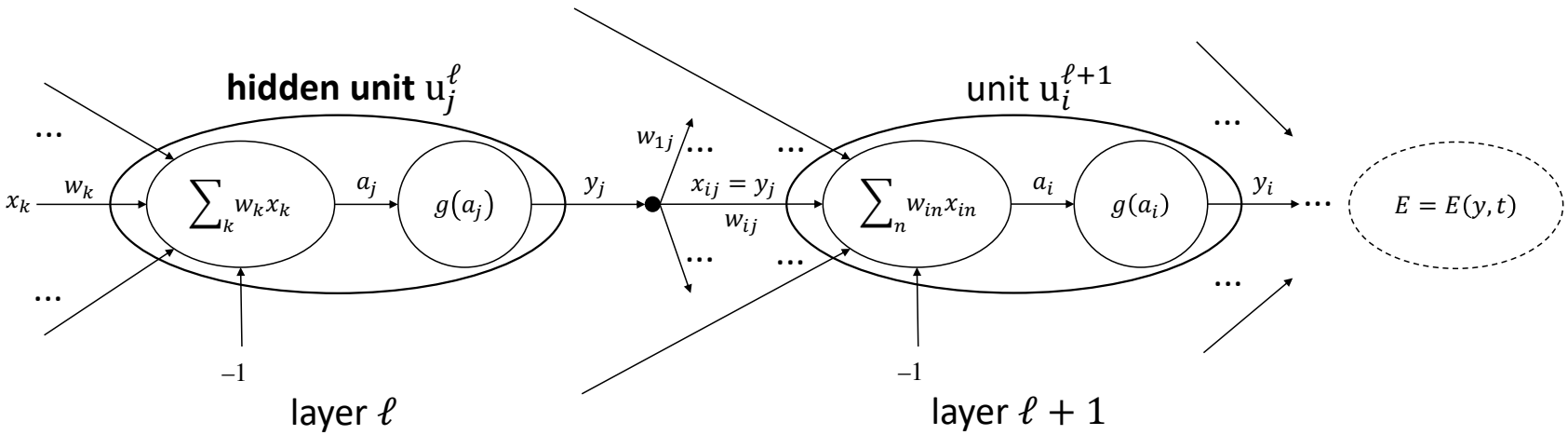
Note that **all** the above terms are **known**, after the **forward propagation** step.

Accordingly, the weight updates of the input connections to the **output** unit can be computed immediately.

Back-propagation algorithm: partial derivatives

Consider now any **hidden unit** of the ℓ-th layer, u_j^ℓ.

The error function depends on such a unit through its output y_j, which in turn is the input of each unit $\mathrm{u}_i^{\ell+1}$ of the **next** layer (either a hidden or the output layer):



Back-propagation algorithm: partial derivatives

The partial derivative for the weight of any input connection to the hidden unit u_j^ℓ can be computed similarly to the output unit:

$$\frac{\partial E}{\partial w_k} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial a_j} \frac{\partial a_j}{\partial w_k}$$

As shown above, the last two terms can be computed as follows:

$$\begin{aligned} \frac{\partial y_j}{\partial a_j} &= y_j(1 - y_j) \\ \frac{\partial a_j}{\partial w_k} &= x_k \end{aligned}$$

Back-propagation algorithm: partial derivatives

The first term $\partial E/\partial y_j$ can be computed taking into account that $E(y,t)$ depends on y_j through the **outputs** of **each** unit $u_i^{\ell+1}$ of the **next** layer, and therefore it can be rewritten by applying the **chain rule** as follows:

$$\frac{\partial E}{\partial y_j} = \sum_{u_i^{\ell+1}} \frac{\partial E}{\partial y_i} \frac{\partial y_i}{\partial y_j}$$

The second term can in turn be computed using the **chain rule**:

$$\frac{\partial y_i}{\partial y_j} = \frac{\partial y_i}{\partial a_i} \frac{\partial a_i}{\partial y_j} = y_i(1-y_i)w_{ij}$$

Back-propagation algorithm: partial derivatives

One finally obtains that, for a hidden unit u_j^ℓ:

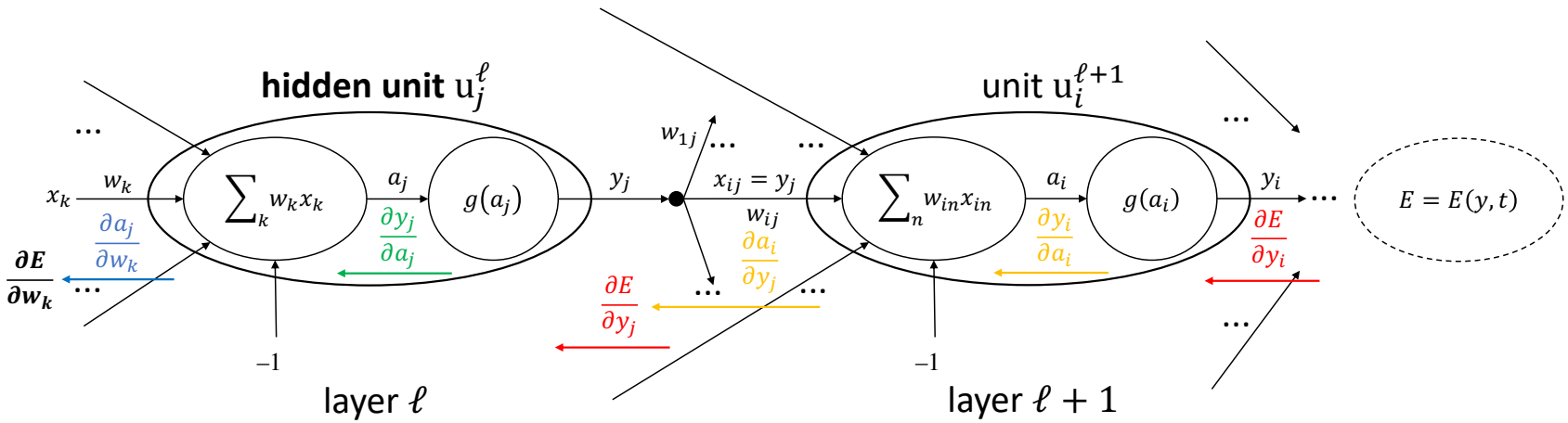
$$\frac{\partial E}{\partial w_k} = \left[\sum_{u_i^{\ell+1}} \frac{\partial E}{\partial y_i} y_i(1-y_i) w_{ij} \right] y_j(1-y_j) x_k$$

- **all** the above terms **except** for $\partial E / \partial y_i$ are **known** from the **forward pass**
- the term $\partial E / \partial y_i$ is known from the **backward pass** carried out for all the units $u_i^{\ell+1}$ of the **next** layer

This shows that the computation of the partial derivatives $\partial E / \partial w$ has to proceed **backwards**, layer by layer, starting from the units of the output layer.

Back-propagation algorithm: backward pass

The figure below shows the different steps of the computation of the partial derivative $\partial E / \partial w_k$.



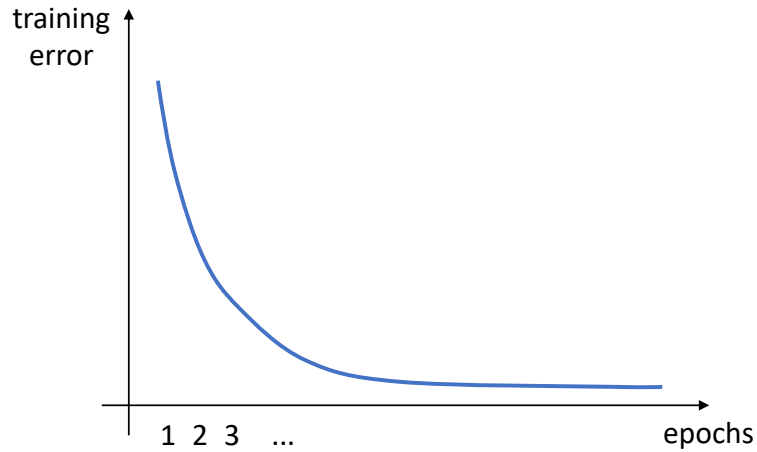
Pseudo-code of the back-propagation learning algorithm

function BACK-PROPAGATION ($\mathcal{T}$)
returns weight values **w**
    randomly choose the initial weight values **w**
    repeat
        for each $(\mathbf{x}, \mathbf{t}) \in \mathcal{T}$ **do**
            compute the network output **y** (forward-propagation)
            compute the error function $E(\mathbf{y}, \mathbf{t})$
            weight update: $w \leftarrow w - \eta \frac{\partial E(\mathbf{y},\mathbf{t})}{\partial w}$ (back-propagation)
        end for
    until a stopping condition is satisfied
    return w

Convergence of the back-propagation learning algorithm

The error functions used for the back-propagation learning algorithm have many **local minima**: in this case, starting from **random** weight values, the gradient descent-like approach **does not** guarantee to reach the **global minimum**.

Usually the error function decreases over the first epochs, then reaches a plateau, corresponding to a (local) minimum of the error function.



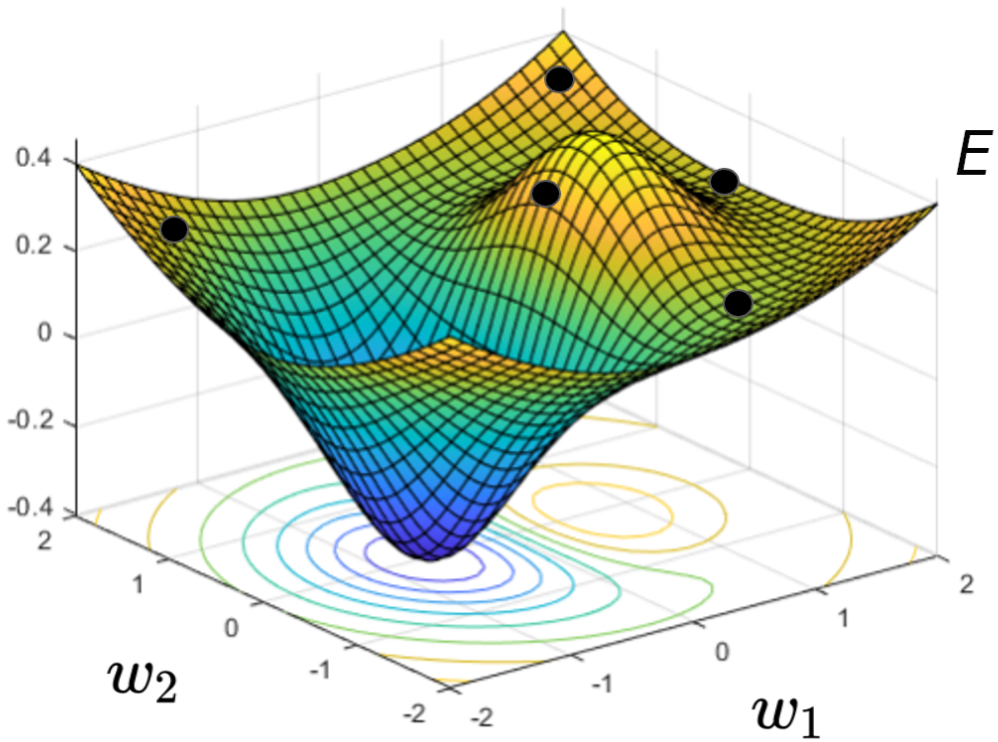
The **stopping condition** can therefore be defined in terms of

- carrying out a predefined (sufficiently high) number of epochs, or
- detecting that the error function remains almost constant over a certain number of consecutive epochs

The **convergence speed** depends also on the value of the constant η. Typical values are in the range from $\eta = 10^{-4}$ to $\eta = 10$.

Dealing with local minima of the error function

To mitigate the issue of local minima, a **multi-start** strategy can be used: the back-propagation algorithm is run **several times** starting from **different** random weights; then the weight values which provide the minimum training error across the runs are chosen.



Attribute normalisation

The back-propagation learning algorithm usually benefits from **normalised** attribute values, e.g., linearly re-scaling each attribute into the range $[0, 1]$ (**min-max** scaling) or to zero mean and unit variance:

- **min-max** scaling:

$$x_i' = \frac{x_i - x_{i,\min}}{x_{i,\max} - x_{i,\min}}, \ i = 1, \ldots, d$$

  where $x_{i,\min}$ and $x_{i,\max}$ are the minimum and maximum values of the i-th attribute across training examples

- zero mean and unit variance:

$$x_i' = \frac{x_i - \mu_i}{\sigma_i}, \ i = 1, \ldots, d$$

  where μ_i and σ_i are the mean and standard deviation of the i-th attribute evaluated on training examples

Dealing with over-fitting

ANNs can incur **over-fitting**, as all machine learning models, due to several possible factors:

- model **complexity**, which can be roughly evaluated in terms of the **number of connection weights**, that in turn depends on the number of hidden layers and of hidden units
- **number of training examples**
- **number of attributes**

These factors are **interlinked** with each other, i.e., a **relatively high** number of attributes with respect to the number of training examples, and a **relatively high** number of connection weights with respect to the number of training examples is **more likely** to lead to over-fitting.

Dealing with over-fitting

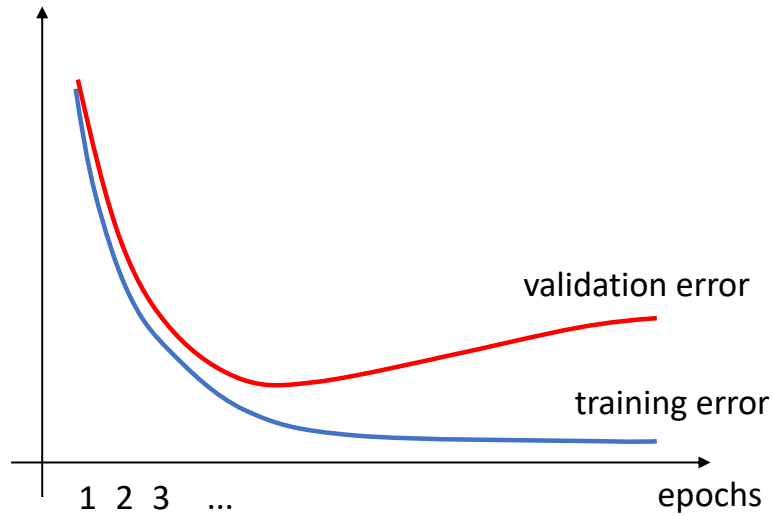
In ANNs over-fitting can be dealt with in several ways.

- choosing the network **architecture** by **trial-and-error**:
  - starting with a **relatively** simple architecture, e.g., one hidden layer with "few" hidden units (depending on the number of attributes and of training examples)
  - evaluating increasingly complex architectures, until the estimated generalisation capability is satisfactory
- **regularisation**: setting constraints on the network weights through the addition of specific **penalty terms** to the error function, to favour **"simpler"** decision boundaries (e.g., **weight decay**, to avoid **too large** weights in absolute value)

Dealing with over-fitting

Another strategy, which can be used together with the trial-and-error approach for model choice, is to monitor the error function during the training epochs, on a **distinct** set of labelled examples than the ones used for training, named **validation set**.

Whereas the **training error** usually decreases as the number of epochs increases, the **validation error** may start increasing after some epochs: in this case the resulting ANN is likely to over-fit.



To prevent or to mitigate over-fitting, the back-propagation algorithm can be stopped when the **validation error** starts increasing, and the weight values corresponding to the minimum **validation error** can be used, instead of the ones leading to the minimum **training error** – this technique is called **early stopping**.

Software libraries for Artificial neural networks

scikit-learn: an open source Python library for machine learning, including FF-ML ANNs (and DTs):

`https://scikit-learn.org`

playground: a web application that visualises the decision regions produced during the execution of the back-propagation algorithm by FF-ML ANNs with up to 6 hidden layers and up to 8 hidden units per layer, on four different two-class toy problems with two real-valued inputs:

`https://playground.tensorflow.org`

Artificial neural networks vs Decision trees

ANNs and DTs are two examples of different machine learning models.

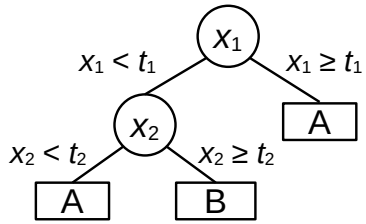
They have different characteristics in terms of:

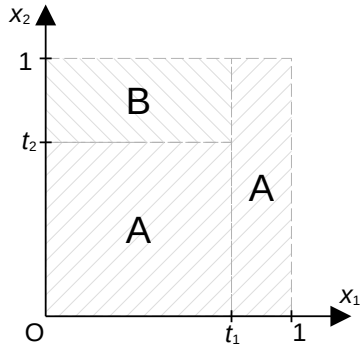
- "shape" of class boundaries in attribute space
- generalisation capability
- **interpretability**

Artificial neural networks vs Decision trees

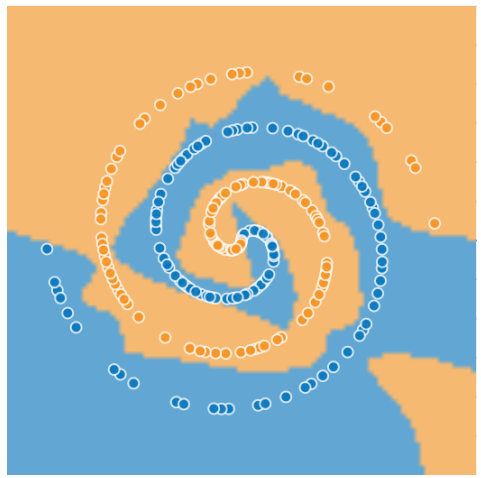
"Shape" of class boundaries in attribute space:

As previously shown, DTs produce **axis-parallel** splits of the attribute space.





ANNs can produce more flexible class boundaries, instead, depending on the number of hidden layers and of hidden units.
The example on the right shows the decision regions produced by an ANN with **two** hidden layers of **eight** units each, on a two-class problem with two attributes, with intertwined spiral-shaped class distributions.



taken from `https://playground.tensorflow.org`

Artificial neural networks vs Decision trees

Generalization capability:

- ANNs are usually more **robust** than DTs to **noise** on attribute values and on class labels, which may be due to imprecise attribute measurements or to errors in manual labelling of training examples
- ANNs (with a suitable architecture) often achieve a higher generalisation capability than DTs

It is however well known that **no** machine learning model can outperform **all** other models on **all** classification or regression tasks: **model selection** should therefore be carefully carried out, taking into account the characteristics of the task at hand.

Artificial neural networks vs Decision trees

Interpretability of a machine learning model refers to the possibility for its users to understand its outputs (predictions), e.g., how they have been produced from inputs.

Interpretability has become a key issue for the widespread adoption of machine learning models, especially in sensitive application scenarios such as medical diagnosis.

- DTs are in principle interpretable **by construction**, in terms of high-level `IF-THEN` rules, although rules associated to large and deep DTs are likely to be very difficult to understand
- the input-output relationship represented by ANNs has no analogous high-level interpretation, instead, since the output values depend in a **distributed** way on **all** the connection weights; for this reason models like ANNs are called **"black boxes"**

Beyond Artificial neural networks

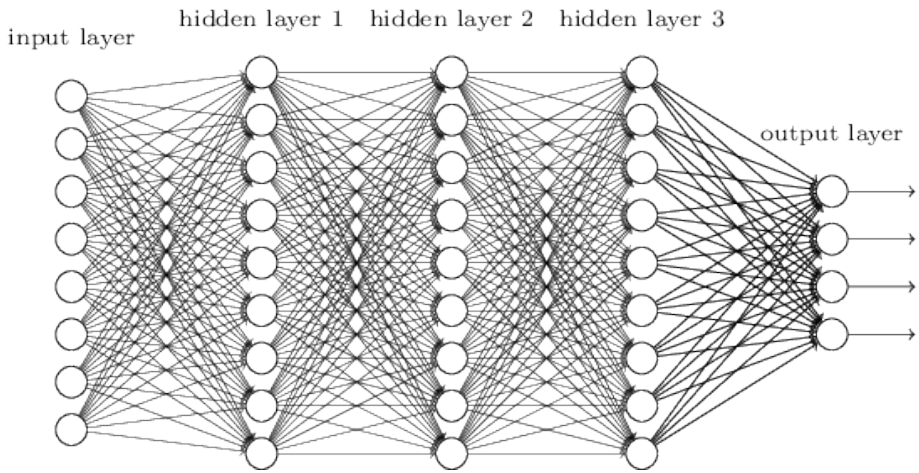
The learning algorithms of many machine learning models, e.g., Support Vector Machines and **deep learning** models, are based on a similar approach as back-propagation:

- defining a **loss function** that evaluates how "far" the model prediction is from the desired one
- iteratively updating the model's parameters to minimise the loss function on training examples, through a **gradient descent**-like procedure

Deep neural networks

Deep neural networks (DNNs) are a recent, very popular extension of ANNs (but early ideas date back to the 1970s).

Basically, they are multi-layer networks with **many** hidden layers. An example of a (not so deep) DNN:



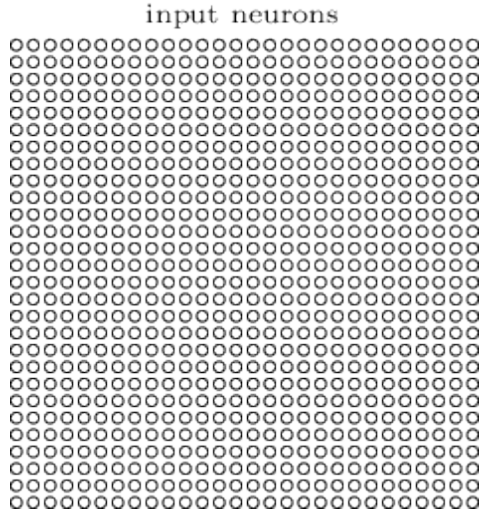
taken from `http://neuralnetworksanddeeplearning.com`

Ad hoc activation functions and modifications of the back-propagation learning algorithm have been devised to avoid drawbacks emerging in DNNs, in particular, **very slow convergence**.

Deep neural networks

DNNs are widely employed for **computer vision** tasks, for which specialized architectures have been devised, named **convolutional neural networks** (CNNs).

In computer vision tasks the CNN input is a **raw image**, e.g., a 2D **array** of pixel values, and the units of the **first hidden layers** are arranged into 2D arrays as well, to take into account the **spatial adjacency** between pixels:



taken from `http://neuralnetworksanddeeplearning.com`

Deep neural networks

The 2D-shaped hidden layers of CNNs carry out two specific kinds of image processing operations:

- **convolution** layers carry out specific image **filtering** operations: their connection weights are set by the **learning algorithm**
- **pooling** layers carry out a **downsampling** operation on the output of convolution layers, using **predefined** connection weights

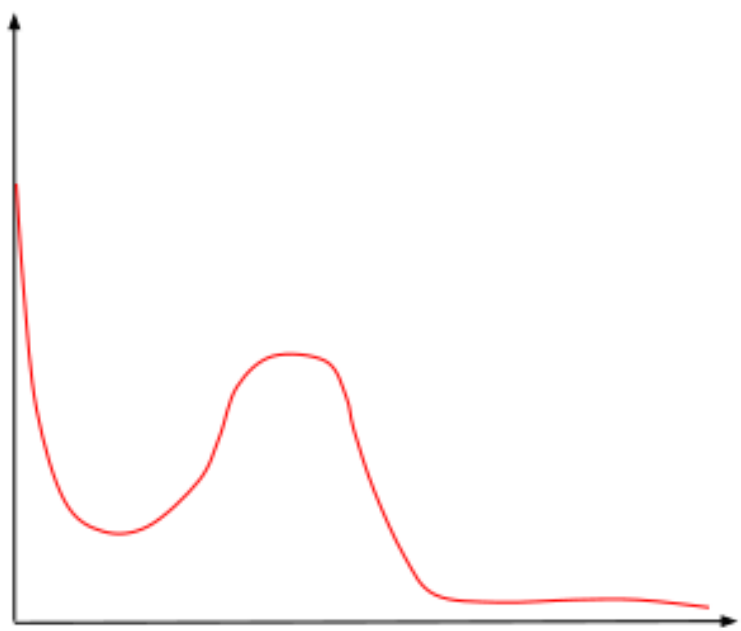
The units of convolutional and pooling layers are **not** fully connected.

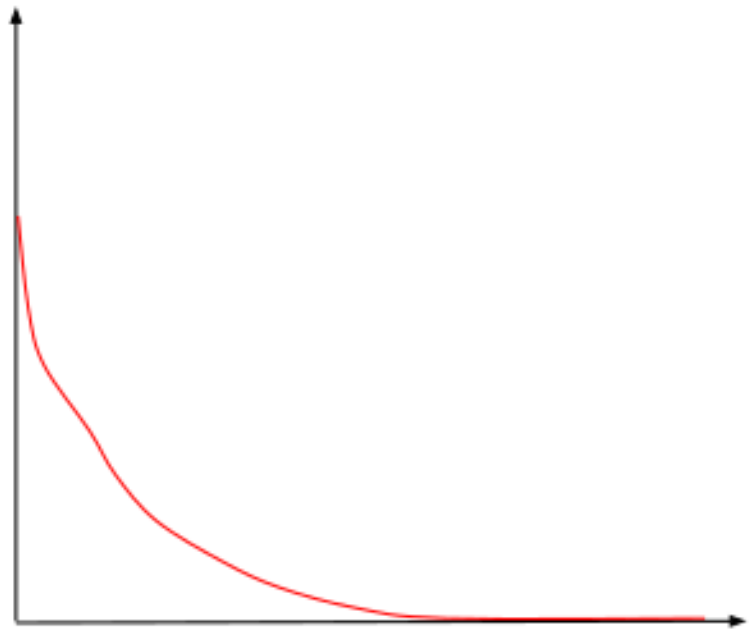
The **upper layers** consist instead of a standard, **fully-connected ML-FF** sub-network, typically made up of one hidden layer and the output layer.

A key difference between standard ("shallow") and deep networks is that the latter **do not** require the **manual** definition of image features (attributes), but directly operate on **raw images**: their convolutional and pooling layers can therefore be seen as **automatic feature extractors**.

Validation Error in Deep Neural Networks

In deep neural networks, the trend of the validation error is often the following:





Deep neural network resources

Some introductory and advanced readings on DNNs (and on "shallow" NNs as well):

- `http://neuralnetworksanddeeplearning.com` (suggested as a very gentle introduction to ANNs and DNNs)
- `http://deeplearning.stanford.edu/tutorial/`
- `http://www.deeplearningbook.org/`
- `http://deeplearning.net/`