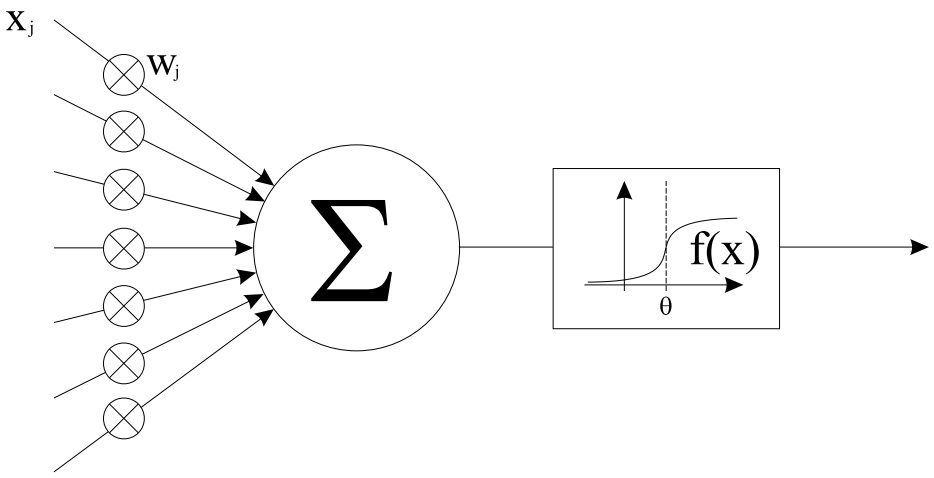
Neural Networks:

In principle, neural networks can compute any *computable function* (i.e. they can do everything a digital computer does), under some assumptions.

In practice, NNs are especially useful for classification and function approximation/mapping problems which are tolerant of some imprecision, which have lots of training data available, but to which hard and fast rules cannot easily be applied.

The basic unit is the **neuron**, or "Threshold Logic Unit":

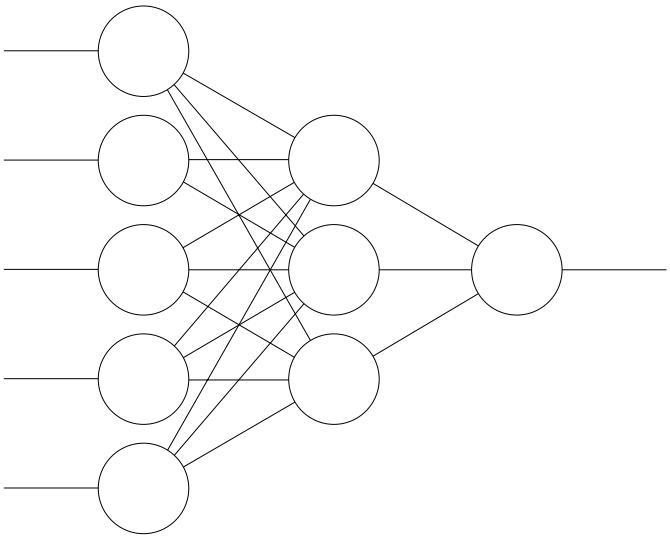


The simplest NN is the "perceptron", constituted by a single neuron.

NNs can have any topology suitable to the problem to which are tailored (bi- or three-dimensional, layered or unlayered...)

Multi-Layer Perceptrons:

The MLP is an example of "feedforward" network, i.e. the information flows only in one direction, without cycles.



MLP's structure consists of:

- one **"input layer"**: neurons have one single input each from the environment
- one **"output layer"**: neurons communicate their outputs to the user
- one or more **"hidden layers"**: neurons have as input the outputs of the preceding layer and give their outputs as inputs of the following layer

Supervised learning:

For every event i the user provides both the inputs $\vec{I_i}$ and the desired output T_i.

With the current weights (random, at the beginning) the NN associates $\vec{I_i} \to O_i$, and a **distance** or **error** $d(O_i - T_i)$ is computed.

Usually, $d(O_i - T_i) = |O_i - T_i|$ or $d(O_i - T_i) = (O_i - T_i)^2$

Errors are then **propagated back** through the NN, and the weights are modified attempting to have a smaller error at the next iteration.

Examples:

- Forecasts: the NN tries to guess the next values of the input variables, $T_i = I_{i+1}$.
- Fitting: $T_i = f(I_i)$, f unknown. NN can approximate with arbitrary accuracy any continuous (and quasi-continuous) function.
- Classification: f is discrete (e.g. signal$\to 1$, background$\to 0$).

How to train a NN (or anything else) to classify things:

- **Show examples of both "signal" and "background"**

  (to teach what a dog is, show dogs and non-dogs).

- **Examples should be as various as possible for every class**

  (people in isolated amazonic tribes aren't able to recognize white men as humans, since they are too different from any other human being that they know).

- **In principle, examples should be as many as possible...**

- **...but many examples $\Rightarrow$ many learning iterations.**

  If available learning time is not ∞, a balance has to be found.

- **Don't iterate too much**

  to avoid "overlearning" or "overfitting" (see next transparency).

Overfitting:

If the training has gone too far the weights can be optimally adapted to the data set used to learn but the network can have poor performance on any other analogous data set.

.....FIGURE.....

Solution: use two data sets.

- *Training sample*: the examples shown.
- *Test sample*: at every learning iteration, the NN is applied to the test sample and $d(O_i - T_i)$ is computed.

Learning is stopped at the minimum of $d(O_i - T_i)$ for the test sample. This is the point of maximum generalization for the NN.

Training algorithm:

$d(O_i - T_i)$ is treated as a function of the synaptic weights of all the neurons.

Training is a minimization problem in the space of the weights.

The most popular training algorithms are variants of *Standard backpropagation*, which is an NN implementation of the **heavy ball method**:

the minimum is looked for by following the gradient of the "error surface", $\frac{\partial E}{\partial w_{ij}}$

Minimization algorithms are designed to avoid, as far as possible,

- to be trapped in a local ("false") minimum;
- to skip the absolute ("true") minimum.

The success of an algorithm depends on the features of the particular error surface given by the problem.

RPROP (Resilient Propagation):

RPROP was chosen both for its performances in minimum finding and for its speed.

RPROP belongs to the family of Backpropagation algorithms:

$$\Delta w_{ij} = \eta \delta_i o_j$$

o_j is an output of unit j and an input to i

δ_i is a function of the outputs of the preceding layer's neurons:

- if i is an output neuron: $\delta_i = o_i(1 - o_i)E \quad (E = d(O - T))$
- if i is a hidden neuron: $\delta_i = o_i(1 - o_i) \sum_k \delta_k w_{ki}$

η is the **learning rate**.

The special feature of RPROP is the way to update η. Only the direction of $\frac{\partial \vec{E}}{\partial w_{ij}}$ is used.

It begins with an initial small update value, and then:

- increases, if $\frac{\partial \vec{E}}{\partial w_{ij}}(t)$ and $\frac{\partial \vec{E}}{\partial w_{ij}}(t-1)$ have angle $< 180°$;
- decreases otherwise.

How many hidden neurons?

Question: what is the optimal number of hidden neurons for a given problem?
Answer: nobody knows.

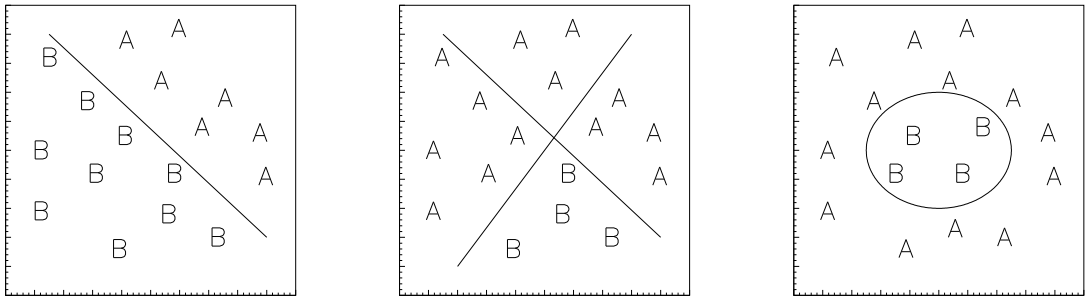
NN literature is full of "rules of thumb", but their limits of application are often tight or/and mostly unknown.

The recommended way to proceed is by trial and error:
beginning with only one unit per layer, the number of units is increased until the performance of the NN ceases to improve.

When the number of units is too large, the number of connections, and consequently the number of weights to variate, makes the training very slow and a larger training sample becomes necessary to avoid over-fitting.

How many hidden layers?

The first question is:
do we need hidden layers?



(a) (b) (c)

- (a) **Linearly separable problem:**
  finding a single hyper-plane separating A/B.

  A **perceptron** is able to do that. Output ≈ 0 on the "A" side, ≈ 1 on the other side, and intermediate values in the proximity of the separating line.

- (b) **Non-linearly separable problem:**
  no straight line can properly separate the two categories of events.

  The introduction of a hidden layer overcomes the problem.

- (c) **Non-linearly separable problem with closed surface:**
  a second hidden layer can greatly improve the performances of the NN.