Introduction to computational models

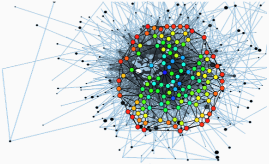
Tema 1. Backpropagation algorithm

Pedro Antonio Gutiérrez (pagutierrez@uco.es)
César Hervás Martínez (chervas@uco.es)

9th September 2021

Departamento de Informática y Análisis Numérico
Universidad de Córdoba.

Contents



Introduction
Artificial neuron
Neural networks
Learning
Classification
ANNs using Weka software
Conclusions

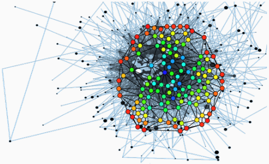
Summary

Objectives of this topic

Introduction to artificial neural networks (ANNs):

- Motivation for the use of this kind of models.
- Basic model: the artificial neuron or the simple perceptron.
- More advanced model: ANN or multilayer perceptron.
- Training algorithm: backpropagation or gradient descent.
- Limitations of the basic algorithm and mechanisms to avoid them.
- Adaptations of the algorithm and the models for tackling classification problems.
- Use of ANNs in Weka.
- Advantages/disadvantages of ANNs.

Contents



Introduction
Artificial neuron
Neural networks
Learning
Classification
ANNs using Weka software
Conclusions

The brain as a computational device

- Animals are able to react for adapting to external and internal changes, using their nervous system to implement these behaviours.
- An adequate model of simulation of the nervous system should be able to produce similar responses and behaviours in an artificial system.
- The nervous system is made up of relatively simple units, the neurons $\rightarrow$ we can draw inspiration from the behaviour and functioning of the brain.

The brain as a computational device

How is our brain able to work so well?

- Massive parallelism: the brain is a signal or information processing system that is composed of a large number of simple processing elements, called neurons.
- Connexionism: the brain is a highly interconnected neural system, so that the state of a neuron affects the input of a large number of other connected neurons according to their weights or links.
- Distributed associative memory: the storage of information in the brain is concentrated in the synaptic connections of the neural network, or more precisely, in the patterns of these connections and in their strength (**weights**).

The brain as a computational device

- Motivation: learning algorithms developed over centuries cannot address the complexity of real problems.
- However, the human brain is the most sophisticated computer for solving extremely complex problems.
  - Reasonable size: 10^{11} neurons (neural cells), where only a small portion is used.
  - Simple processing nodes: the cells do not contain much information.
  - Massively parallel: each region of the brain performs specific tasks.
  - Fault tolerant y robust: information is mainly saved in the connections between neurons, which can be regenerated.

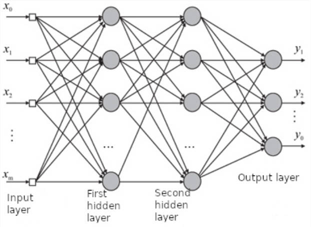
Comparing the brain against a computer

	Computer	Human brain
Computing units	1 CPU, 10^5 gates	10^{11} neurons
Storage units	10^9 RAM bits	10^{14} synapses
Time per cycle	10^{-8} seconds	10^{-3} seconds
Bandwidth	10^{22} bits / second	10^{28} bits / second

Even if the computer is a million times faster than the brain in computing speed, a brain ends up being a million times faster than the computer in bandwidth (thanks to the large amount of computing elements).

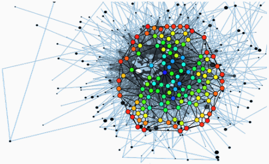
- Recognizing a face:
    - Brain $< 1s$.
    - Computer: billions of cycles.

Artificial neural networks



- In most classification/regression problems, formalising a decision rule is very complex (or impossible).
- An artificial neural network is a non-linear classification/regression model, capable of accumulating knowledge (learning) about its coefficients and structure, using an algorithm (backpropagation algorithm).
- After the learning process, a network is able to approximate a continuous function, which is supposed to be our decision rule.

Contents



Introduction

Artificial neuron

Neural networks

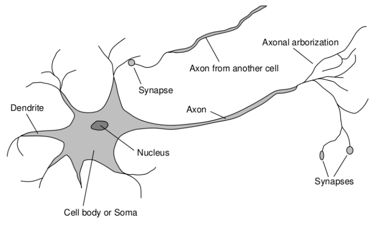
Learning

Classification

ANNs using Weka software

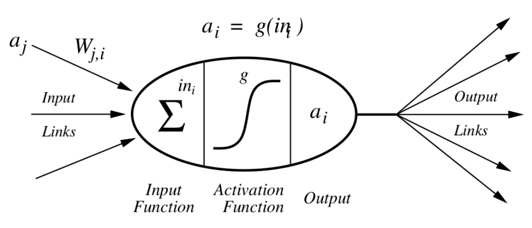
Conclusions

Biological neuron



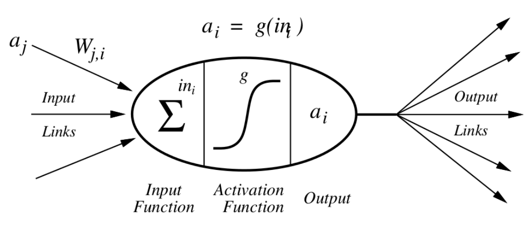
- A neuron does not do anything until the influence of its inputs reaches a certain level.
- The neuron produces an output in the form of a pulse that starts in the nucleus, goes down the axon and ends in its branches.
- It is either triggered or does nothing: "all-or-nothing" device.
- The output causes the excitation or inhibition of other neurons.

Artificial neuron (simple perceptron)



- Artificial neurons are nodes connected to other nodes by links.
- Each link is associated with a weight.
- The weight determines the nature (excitatory $+$ or inhibitory $-$) and the strength (absolute value) of the influence between nodes.
- If the influence of all input links is high enough, the node is activated.

Artificial neuron (simple perceptron)



- Each *i*-th node has several input and output connections, each with its own weight.
- The output is a function of the weighted sum of the inputs.

Artificial neuron (simple perceptron)

- Each input link to the i neuron provides it with a a_j activation value coming from another neuron.
- Often, the input function is the weighted sum of these trigger values:

$$in_i(a_1, \ldots, a_{n_i}) \;=\; \sum_{j=1}^{n_i} W_{j,i}\, a_j$$

- The output of the neuron is the result of applying the activation to the result of the input function:

$$out_i = g(in_i) = g\left(\sum_{j=1}^{n_i} W_{j,i}\, a_j\right)$$

Artificial neuron (simple perceptron)

In vector form:

$$out_i = f(\mathrm{x}, \mathrm{w}),$$

where x is the vector of inputs to the neuron, w is the vector of synaptic weights and the function f is the composition of the input function and the activation function:

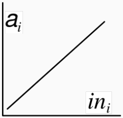
$$f(\mathrm{x}, \mathrm{w}) = g(in(\mathrm{x}, \mathrm{w})).$$

For additive neurons, we can use the scalar product:

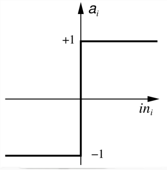
$$f(\mathrm{x}, \mathrm{w}) = g(\mathrm{x} \cdot \mathrm{w}).$$

Activation functions $g(\cdot)$

- linear function: $linear(x) = x$

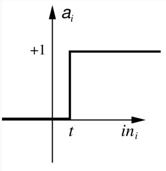


- sign function: $sign(x) = \begin{cases} +1, & \text{if } x \geq 0 \\ -1, & \text{if } x < 0 \end{cases}$



- Threshold or step function:

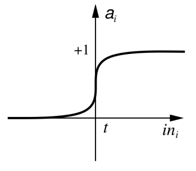
$$step_t(x) = \frac{sign(x-t)+1}{2} = \begin{cases} 1, & \text{if } x \geq t \\ 0, & \text{if } x < t \end{cases}$$



Activation functions $g(\cdot)$

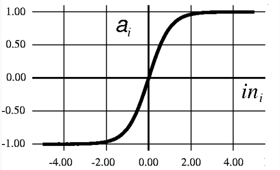
- Sigmoid function (logistic):

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



- Hyperbolic tangent function:

$$\tanh(x) = \frac{1 - e^{-2(x-t)}}{1 + e^{-2(x-t)}}$$



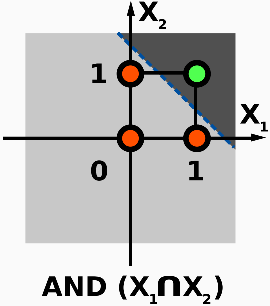
Simulation of boolean functions

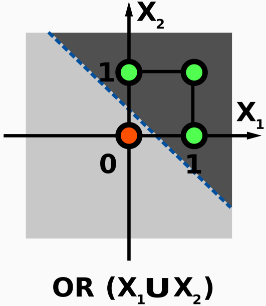
- The output of a threshold neuron is binary, while the inputs can be binary or continuous.
- If the inputs are binary, a threshold function implements a Boolean function.
- The Boolean alphabet $\{1, -1\}$ is normally used instead of $\{0, 1\}$. The correspondence with the classical Boolean alphabet $\{0.1\}$ can be established by:

$$0 \to 1; 1 \to -1; y \in \{0, 1\}, x \in \{1, -1\} \Rightarrow x = 1 - 2y = (-1)^y$$

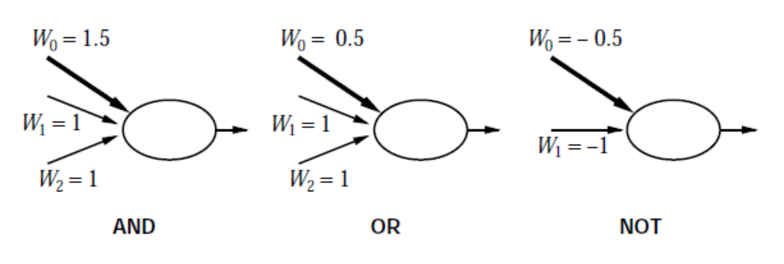
Simulation of boolean functions

- Simulating simple logical functions by means of a neuron.
- For And and Or, we only need one neuron with the function *step* and the correctly selected weights:



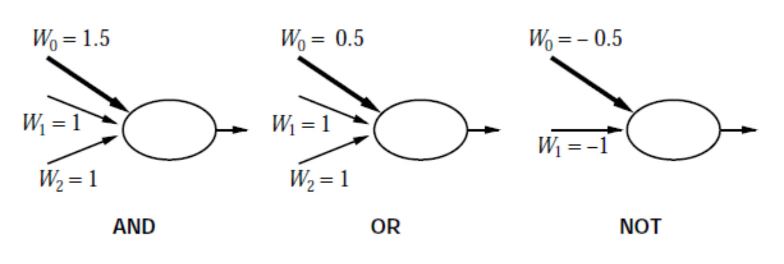


Artificial neuron



$$and(0,0) = step_{1.5}(1 \cdot 0 + 1 \cdot 0) = step_{1.5}(0) = 0$$
$$and(0,1) = step_{1.5}(1 \cdot 0 + 1 \cdot 1) = step_{1.5}(1) = 0$$
$$and(1,0) = step_{1.5}(1 \cdot 1 + 1 \cdot 0) = step_{1.5}(1) = 0$$
$$and(1,1) = step_{1.5}(1 \cdot 1 + 1 \cdot 1) = step_{1.5}(2) = 1$$

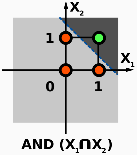
Artificial neuron

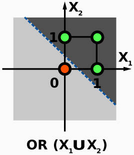


$$or(0,0) = step_{0.5}(1 \cdot 0 + 1 \cdot 0) = step_{0.5}(0) = 0$$
$$or(0,1) = step_{0.5}(1 \cdot 0 + 1 \cdot 1) = step_{0.5}(1) = 1$$
$$or(1,0) = step_{0.5}(1 \cdot 1 + 1 \cdot 0) = step_{0.5}(1) = 1$$
$$or(1,1) = step_{0.5}(1 \cdot 1 + 1 \cdot 1) = step_{0.5}(2) = 1$$

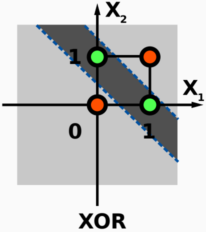
Artificial neuron

- These problems can be solved with a neuron because they can be linearly separable:

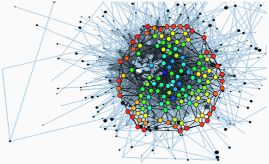




- Problem: a single neuron is not capable of solving more complex problems, e.g. XOR.



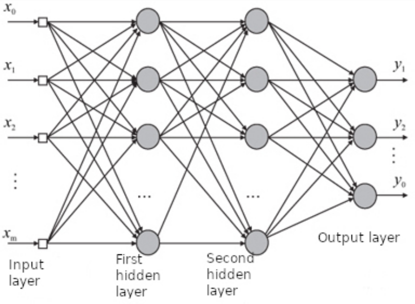
Contents



Introduction
Artificial neuron
Neural networks
Learning
Classification
ANNs using Weka software
Conclusions

Artificial neural networks

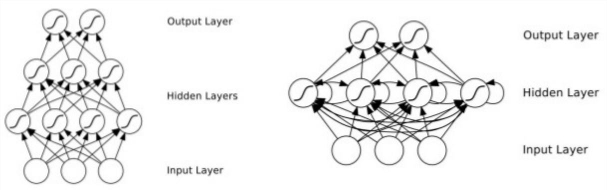
- An Artificial Neural Network (ANN) is a graph of nodes (or neurons) connected by links.
- The neurons are organized in layers, so that the outputs of the neurons in one layer serve as inputs for the neurons in the next layer.
- A popular example is the *MultiLayer Perceptron* (MLP).



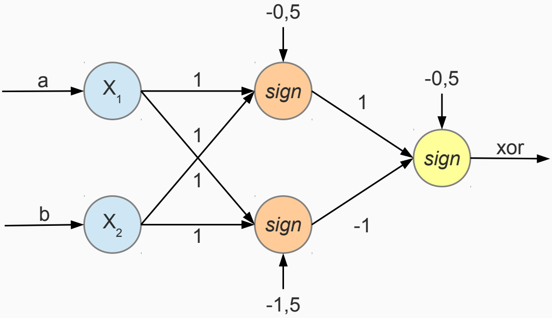
Artificial neural networks

Types or ANNs according to their organisation:

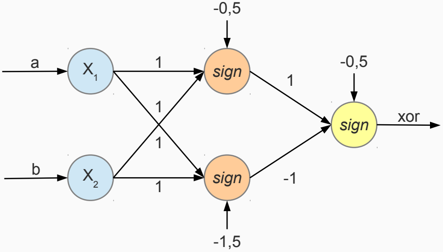
- Feedforward neural networks: neurons in one layer only link up with neurons in the next layer.
  - Sometimes, links that skip layers are established (*skip layer connections*): the input layer connected to the first layer and to all the hidden layers.
- Recurrent ANNs: neurons in a layer can be connected to each other in loops (used for time series modelling, natural language processing…).



- The XOR problem can be solved with a single-layer ANNs and two neurons:

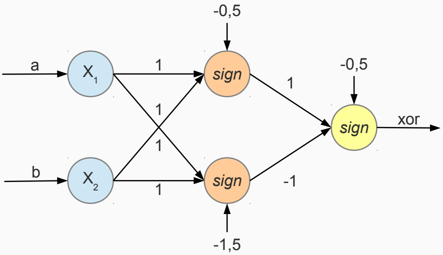


Artificial neural networks



$$xor(0,0) = step_{0.5}(step_{0.5}(1 \cdot 0 + 1 \cdot 0) - step_{1.5}(1 \cdot 0 + 1 \cdot 0))$$
$$= step_{0.5}(step_{0.5}(0) - step_{1.5}(0)) = step_{0.5}(0 - 0) = step_{0.5}(0) = 0$$
$$xor(0,1) = step_{0.5}(step_{0.5}(1 \cdot 0 + 1 \cdot 1) - step_{1.5}(1 \cdot 0 + 1 \cdot 1))$$
$$= step_{0.5}(step_{0.5}(1) - step_{1.5}(1)) = step_{0.5}(1 - 0) = step_{0.5}(1) = 1$$

Artificial neural networks

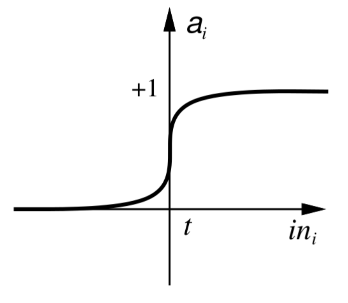


$$xor(1, 0) = step_{0.5}(step_{0.5}(1 \cdot 1 + 1 \cdot 0) - step_{1.5}(1 \cdot 1 + 1 \cdot 0))$$
$$= step_{0.5}(step_{0.5}(1) - step_{1.5}(1)) = step_{0.5}(1 - 0) = step_{0.5}(1) = 1$$
$$xor(1, 1) = step_{0.5}(step_{0.5}(1 \cdot 1 + 1 \cdot 1) - step_{1.5}(1 \cdot 1 + 1 \cdot 1))$$
$$= step_{0.5}(step_{0.5}(2) - step_{1.5}(2)) = step_{0.5}(1 - 1) = step_{0.5}(0) = 0$$

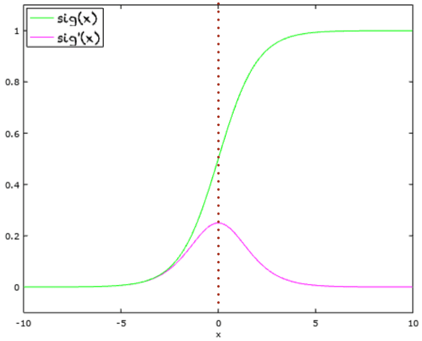
Artificial neural networks

- The hidden layer projects the data to a space where the task to be performed is easier.
- If we want to optimise the weights of the network, the *step* function is not suitable. why?.
- We use an approximation, the sigmoid function (with bias):

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



Properties of the sigmoid function



Plot of σ(x) and its derivate σ'(x)

Domain: (-∞, +∞)

Range: (0, +1)

σ(0) = 0.5

Other properties

$$\sigma(x) = 1 - \sigma(-x)$$

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

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

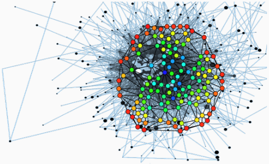
Source: `https://machinelearningmastery.com/a-gentle-introduction-to-sigmoid-function/`

- Very important property of the derivative of $\sigma(x)$:

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

$$\begin{aligned}
\sigma'(x) &= (-1)\cdot\left(1+e^{-x}\right)^{-2}\left(1+e^{-x}\right)' = \\
&= (-1)\cdot\left(1+e^{-x}\right)^{-2}\cdot\left(0+\left(e^{-x}\right)'\right) \\
&= (-1)\cdot\left(1+e^{-x}\right)^{-2}\cdot\left(e^{-x}\right)\cdot(-x)' \\
&= (-1)\cdot\left(1+e^{-x}\right)^{-2}\cdot\left(e^{-x}\right)\cdot(-1) = \frac{e^{-x}}{(1+e^{-x})^2} \\
&= \frac{1}{(1+e^{-x})}\frac{e^{-x}}{(1+e^{-x})} = \sigma(x)\frac{1+e^{-x}-1}{(1+e^{-x})} \\
&= \sigma(x)\left(\frac{(1+e^{-x})}{(1+e^{-x})} - \frac{1}{(1+e^{-x})}\right) = \sigma(x)\cdot(1-\sigma(x))
\end{aligned}$$

Contents



Introduction

Artificial neuron

Neural networks

Learning

Classification

ANNs using Weka software

Conclusions

Backpropagation algorithm

- *Feedforward* neural networks: forward propagation to obtain the outputs.
- *Backpropagation* algorithm: backwards propagation to obtain the error and the derivatives and to adjust weights.
- Given a training set, we want to adjust the weights of the connections so that the error made in classifying or regressing on that set is minimal.
- The general idea is to minimise training error and the mathematical procedure is based on obtain the derivatives of a cost function.
- Error backpropagation algorithm or gradient descent.

Backpropagation algorithm

- Notation:
  - Training patterns:
    - Input vector: $\mathrm{x} = (x_1, \ldots, x_k)$.
    - Target vector: $\mathrm{d} = (d_1, \ldots, d_J)$.
  - Architecture of the neural network: $\{n_0 : n_1 : \ldots : n_H\}$
    - n_h if the number of neurons of the h-th layer.
    - $n_0 = k$, $n_H = J$.
    - $H - 1$ hidden layers.
  - Weights of the network. For each layer h, without considering input layer:
    - Matrix with an input weight vector for each neuron: $\mathrm{W}^h = (\mathrm{w}_1^h, \ldots, \mathrm{w}_{n_h}^h)$.
    - j-th neuron weight vector of h-th layer (including bias): $\mathrm{w}_j^h = (w_{j0}^h, w_{j1}^h, \ldots, w_{jn_{(h-1)}}^h)$.
  - Output of the network: $\mathrm{o} = (o_1, \ldots, o_J)$.

Backpropagation algorithm

First, let's consider prediction problems (regression) with one or more variables to predict.
Let us consider that all the neurons, except those in the input layer, will be of the sigmoids:

- If the neurons have bias, their formula will be:

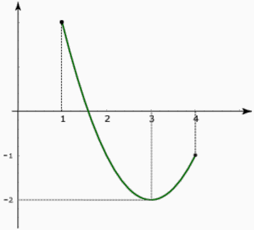
$$out_j^h = \frac{1}{1 + \exp(-w_{j0}^h - \sum_{i=1}^{n_{h-1}} w_{ji}^h out_i^{h-1})}$$

- If they do not have a bias:

$$out_j^h = \frac{1}{1 + \exp(-\sum_{i=1}^{n_{h-1}} w_{ji}^h out_i^{h-1})}$$

Backpropagation algorithm

- Underlying idea:
  - Minimize the function $f(x) = x^2 - 6x + 7$.



  - Derivative function $f'(x) = 2x - 6$
  - Since the function is very simple (one single global minimum) and only depends on one variable (x), we can equal to zero this derivative and we obtain the minimum:

$$2x - 6 = 0 \rightarrow x = 6/2 = 3 \quad (1)$$

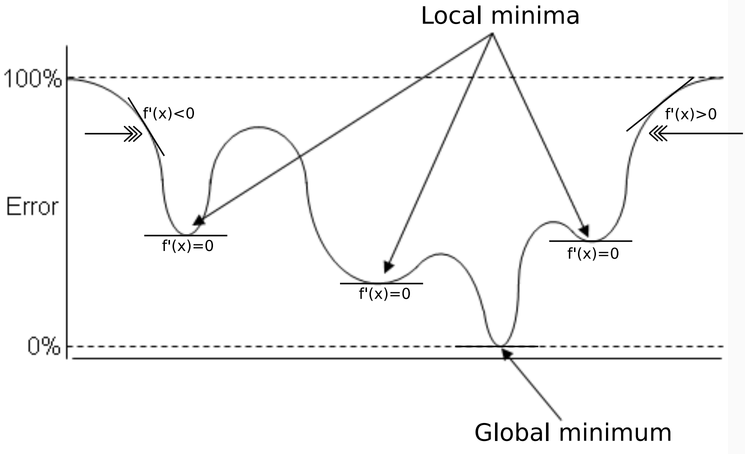
Backpropagation algorithm

- The philosophy is the same, we are going to try to minimise the error made by the neural network, taking the weights of the network as variables. For a specific pattern, the mean square error (we consider regression problems) is:

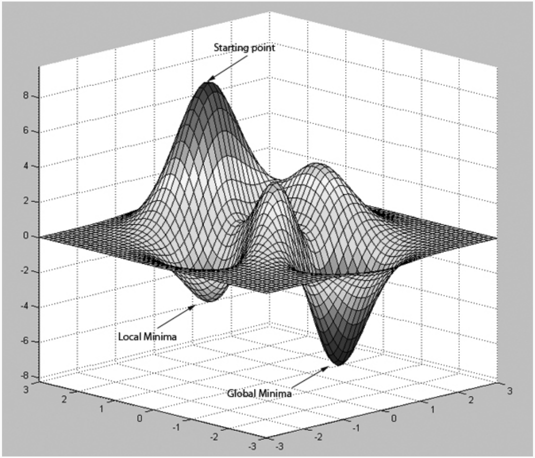
$$E = \frac{1}{J}\sum_{i=1}^{J}(d_i - o_i)^2 \tag{2}$$

- The value of d_i is fixed (according to the training pattern) and is given by the researcher, tutor or decision-maker (supervised learning), but the value of o_i depends on the weights.
- We carry out an iterative process, where, given a value for the current weights, we move those weights trying to minimize E.
- We evaluate the derivative in the current point (weights):
    - If $f'(x) > 0$, we decrease the value of x.
    - If $f'(x) < 0$, we increase the value of x.

Backpropagation algorithm



- Let's imagine that the network has only two weights, then, according to the value of the weights, we can represent their error surface:



Backpropagation algorithm

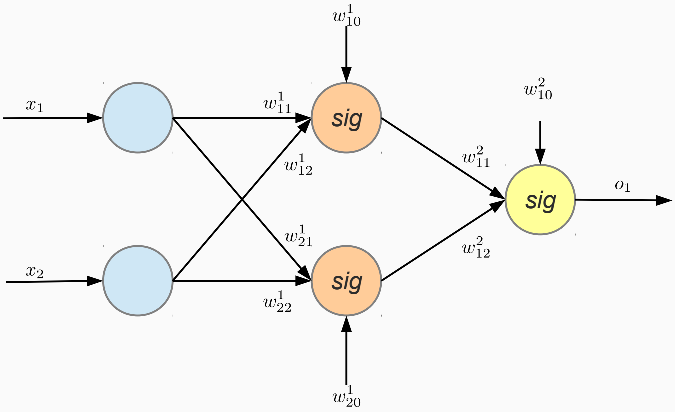
- In the case of having multiple weights, we need a vector of derivatives, where each component is the derivative of the error with respect to each of the weights.
- This is known as the gradient vector.

$$\nabla E = \left\{ \frac{\partial E}{\partial w_{10}^{1}}, \frac{\partial E}{\partial w_{11}^{1}}, \ldots, \frac{\partial E}{\partial w_{n_1 k}^{1}}, \frac{\partial E}{\partial w_{10}^{2}}, , \ldots, \frac{\partial E}{\partial w_{J n_{(H-1)}}^{H}} \right\}$$

- The structure of layers in the ANN means that these derivatives can be recursively calculated.

Backpropagation algorithm

- Let's calculate the derivatives for a simple example and then see how they are calculated in a general way.



Backpropagation algorithm

Phase 1: Forward propagation.

- We call out_j^h to the output of the j-th neuron in the h-th layer.
- Given two input values x_1 and x_2, calculate the output of each neuron.
  - First layer:

$$out_1^0 = x_1; out_2^0 = x_2$$

  - Second layer:

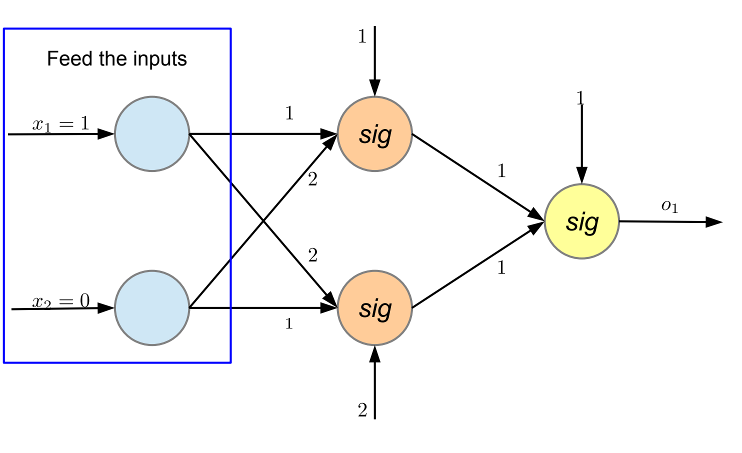
$$out_1^1 = \sigma(w_{10}^1 + w_{11}^1 out_1^0 + w_{12}^1 out_2^0) = \sigma(w_{10}^1 + w_{11}^1 x_1 + w_{12}^1 x_2);$$
$$out_2^1 = \sigma(w_{20}^1 + w_{21}^1 out_1^0 + w_{11}^1 out_2^0) = \sigma(w_{20}^1 + w_{21}^1 x_1 + w_{22}^1 x_2);$$

  - Third layer:

$$out_1^2 = o_1 = \sigma(w_{10}^2 + w_{11}^2 out_1^1 + w_{12}^2 out_2^1)$$

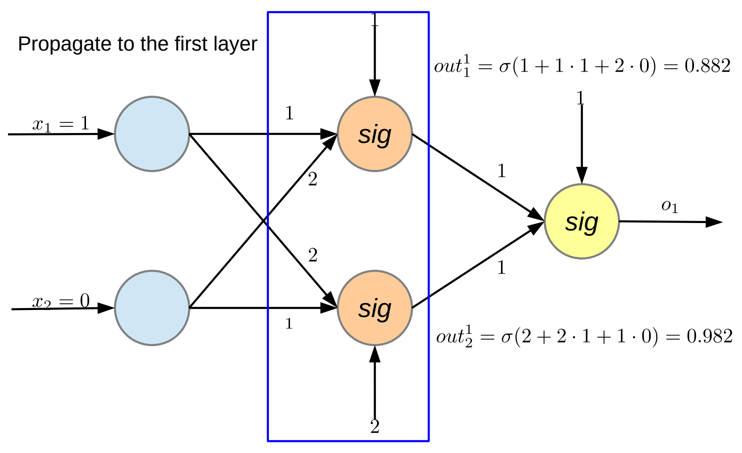
Backpropagation algorithm

Phase 1: Forward propagation.

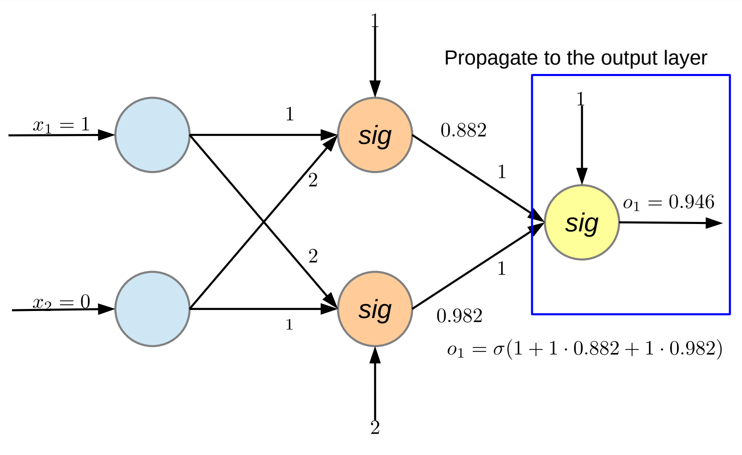


Backpropagation algorithm

Phase 1: Forward propagation.



Phase 1: Forward propagation.



Phase 2: Calculation of the error and derivatives.

- We obtain the value of error made by the network:

$$E = (d_1 - o_1)^2$$

- We now find the derivative of that error with respect each of the weights:

$$\nabla E = \left\{ \frac{\partial E}{\partial w^1_{10}}, \frac{\partial E}{\partial w^1_{11}}, \frac{\partial E}{\partial w^1_{12}}, \frac{\partial E}{\partial w^1_{20}}, \frac{\partial E}{\partial w^1_{21}}, \frac{\partial E}{\partial w^1_{22}}, \frac{\partial E}{\partial w^2_{10}}, \frac{\partial E}{\partial w^2_{11}}, \frac{\partial E}{\partial w^2_{12}} \right\}$$

- From the expression $(d_1 - o_1)^2$, the weights only influence o_1 (o_1 is a function of each of the weights). In these cases, the chain rule allows us to perform these derivatives recursively (the following is true for all the weights):

$$\frac{\partial E}{\partial w^2_{10}} = \frac{\partial E}{\partial o_1}\frac{\partial o_1}{\partial w^2_{10}} = -2(d_1 - o_1)\frac{\partial o_1}{\partial w^2_{10}}$$

Backpropagation algorithm

- We call net_j^h to the weighted sum of inputs of the j-th neuron in the h-th layer, i.e. the output before applying the sigmoid activation function:

$$out_j^h = \sigma(net_j^h)$$

Backpropagation algorithm

- Remember that:

$$o_1 = \sigma(net_1^2)$$
$$\sigma(x)' = \sigma(x)(1 - \sigma(x))$$

- We continue finding the derivative of o_1:

$$\frac{\partial E}{\partial w_{10}^2} = -2(d_1 - o_1)\frac{\partial o_1}{\partial w_{10}^2} = -2(d_1 - o_1) \cdot o_1(1 - o_1)\frac{\partial net_1^2}{\partial w_{10}^2}$$
$$\frac{\partial E}{\partial w_{11}^2} = -2(d_1 - o_1)\frac{\partial o_1}{\partial w_{11}^2} = -2(d_1 - o_1) \cdot o_1(1 - o_1)\frac{\partial net_1^2}{\partial w_{11}^2}$$
$$\frac{\partial E}{\partial w_{12}^2} = -2(d_1 - o_1)\frac{\partial o_1}{\partial w_{12}^2} = -2(d_1 - o_1) \cdot o_1(1 - o_1)\frac{\partial net_1^2}{\partial w_{12}^2}$$

Backpropagation algorithm

- And now we can write the complete derivatives for the output layer weights (w_{1j}^2):

$$
\begin{aligned}
net_1^2 &= w_{10}^2 + w_{11}^2 out_1^1 + w_{12}^2 out_2^1 \\
\frac{\partial E}{\partial w_{10}^2} &= -2(d_1 - o_1)o_1(1 - o_1)\frac{\partial net_1^2}{\partial w_{10}^2} = \\
&= -2(d_1 - o_1)o_1(1 - o_1)1 \\
\frac{\partial E}{\partial w_{11}^2} &= -2(d_1 - o_1)o_1(1 - o_1)\frac{\partial net_1^2}{\partial w_{11}^2} = \\
&= -2(d_1 - o_1)o_1(1 - o_1)out_1^1 \\
\frac{\partial E}{\partial w_{12}^2} &= -2(d_1 - o_1)o_1(1 - o_1)\frac{\partial net_1^2}{\partial w_{12}^2} = \\
&= -2(d_1 - o_1)o_1(1 - o_1)out_2^1
\end{aligned}
$$

Backpropagation algorithm

- We now analyse the derivatives of the hidden layer weights (w^1_{1i} and w^1_{2i}):
  - The weights w^1_{1i} influence on out^1_1.
  - The weights w^1_{2i} influence on out^1_2.
- Therefore, its derivative will be similar to the other derivatives, but when we reach $\frac{\partial net^2_1}{\partial w^1_{ji}}$ we will have to process the rest of the derivative.
- For the weight w^1_{10} (bias of the first neuron on the first layer):

$$
\begin{aligned}
net^2_1 = & \quad w^2_{10} + w^2_{11} out^1_1 + w^2_{12} out^1_2 \\
\frac{\partial E}{\partial w^1_{10}} = & \quad -2(d_1 - o_1) o_1 (1 - o_1) \frac{\partial net^2_1}{\partial w^2_{10}} = \\
= & \quad -2(d_1 - o_1) o_1 (1 - o_1) w^2_{11} \frac{\partial out^1_1}{\partial w^2_{10}}
\end{aligned}
$$

Backpropagation algorithm

- Remember that:

$$out_1^1 = \sigma(net_1^1)$$
$$net_1^1 = w_{10}^1 + w_{11}^1 x_1 + w_{12}^1 x_2$$
$$\sigma(x)' = \sigma(x)(1 - \sigma(x))$$

- We continue obtaining the derivative with respect out_1^1:

$$\frac{\partial E}{\partial w_{10}^1} = -2(d_1 - o_1)o_1(1 - o_1)w_{11}^2 \frac{\partial out_1^1}{\partial w_{10}^2} =$$
$$= -2(d_1 - o_1)o_1(1 - o_1)w_{11}^2 out_1^1(1 - out_1^1)\frac{\partial net_1^1}{\partial w_{10}^2} =$$
$$= -2(d_1 - o_1)o_1(1 - o_1)w_{11}^2 out_1^1(1 - out_1^1)1$$

Backpropagation algorithm

- We repeat this process for all the weights in the hidden layer

$$\frac{\partial E}{\partial w_{10}^1} = -2(d_1 - o_1)o_1(1 - o_1)w_{11}^2 out_1^1(1 - out_1^1)1$$

$$\frac{\partial E}{\partial w_{11}^1} = -2(d_1 - o_1)o_1(1 - o_1)w_{11}^2 out_1^1(1 - out_1^1)x_1$$

$$\frac{\partial E}{\partial w_{12}^1} = -2(d_1 - o_1)o_1(1 - o_1)w_{11}^2 out_1^1(1 - out_1^1)x_2$$

$$\frac{\partial E}{\partial w_{20}^1} = -2(d_1 - o_1)o_1(1 - o_1)w_{12}^2 out_2^1(1 - out_2^1)1$$

$$\frac{\partial E}{\partial w_{21}^1} = -2(d_1 - o_1)o_1(1 - o_1)w_{12}^2 out_2^1(1 - out_2^1)x_1$$

$$\frac{\partial E}{\partial w_{22}^1} = -2(d_1 - o_1)o_1(1 - o_1)w_{12}^2 out_2^1(1 - out_2^1)x_2$$

Backpropagation algorithm

- To recapitulate:
    - Output layer:

$$\frac{\partial E}{\partial w_{ji}^2} = \begin{cases} -2(d_1 - o_1)o_1(1 - o_1)1, & \text{if } i = 0, \\ -2(d_1 - o_1)o_1(1 - o_1)out_i^1, & \text{if } i \neq 0. \end{cases}$$

    - Hidden layer:

$$\frac{\partial E}{\partial w_{ji}^1} = \begin{cases} -2(d_1 - o_1)o_1(1 - o_1)w_{1j}^2 out_j^1(1 - out_j^1)1, & \text{if } i = 0, \\ -2(d_1 - o_1)o_1(1 - o_1)w_{1j}^2 out_j^1(1 - out_j^1)x_i, & \text{if } i \neq 0. \end{cases}$$

Backpropagation algorithm

- Note that there are common parts:
  - Output layer:

$$\frac{\partial E}{\partial w_{ji}^2} = \begin{cases} -2(d_1 - o_1)o_1(1 - o_1)1, & \text{if } i = 0, \\ -2(d_1 - o_1)o_1(1 - o_1)out_i^1, & \text{if } i \neq 0. \end{cases}$$

  - Hidden layer:

$$\frac{\partial E}{\partial w_{ji}^1} = \begin{cases} -2(d_1 - o_1)o_1(1 - o_1)w_{1j}^2 out_j^1(1 - out_j^1)1, & \text{if } i = 0, \\ -2(d_1 - o_1)o_1(1 - o_1)w_{1j}^2 out_j^1(1 - out_j^1)x_i, & \text{if } i \neq 0. \end{cases}$$

Backpropagation algorithm

- Many parts are common, the calculation of derivatives can be done recursively.
- We call δ_j^h to the derivative of the error with respect to the j-th neuron of the h-th layer ("how responsible is that neuron of the error?").

$$
\begin{aligned}
\delta_1^2 &= -2(d_1 - o_1)o_1(1 - o_1) \\
\delta_1^1 &= w_{11}^2 \delta_1^2 out_1^1 (1 - out_1^1) \\
\delta_2^1 &= w_{12}^2 \delta_1^2 out_2^1 (1 - out_2^1)
\end{aligned}
$$

- For the purpose of updating weights, the constant (2) can be ignored.

Backpropagation algorithm

- We redefine the derivatives according to these values δ_j^h:
  - Output layer:

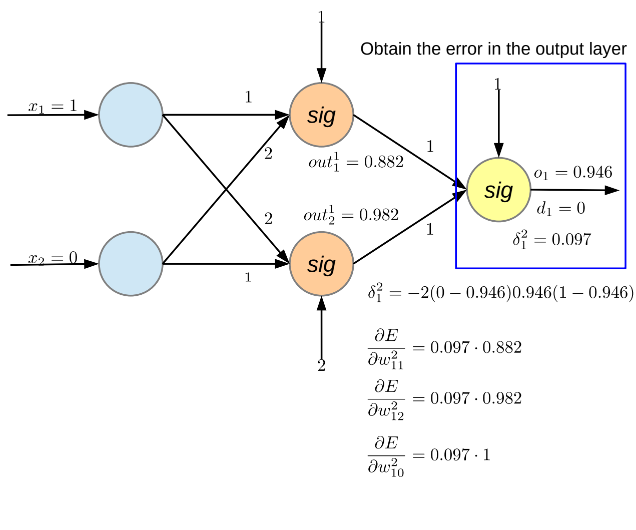
$$\frac{\partial E}{\partial w_{ji}^2} = \begin{cases} \delta_j^2 1, & \text{if } i = 0, \\ \delta_j^2 out_i^1, & \text{if } i \neq 0. \end{cases}$$

  - Hidden layer:

$$\frac{\partial E}{\partial w_{ji}^1} = \begin{cases} \delta_j^1 1, & \text{if } i = 0, \\ \delta_j^1 x_i, & \text{if } i \neq 0. \end{cases}$$

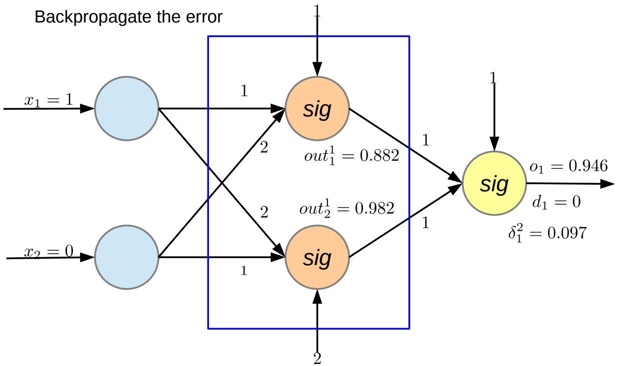
Backpropagation algorithm

Phase 2: Backpropagation.



Backpropagation algorithm

Phase 2: Backpropagation.



$$\delta_1^1 = 1 \cdot 0.097 \cdot 0.882 \cdot (1 - 0.882) = 0.0101$$

$$\delta_2^1 = 1 \cdot 0.097 \cdot 0.982 \cdot (1 - 0.982) = 0.0017$$

$$\frac{\partial E}{\partial w_{10}^1} = 0.0101 \cdot 1 \quad \frac{\partial E}{\partial w_{11}^1} = 0.0101 \cdot 1 \quad \frac{\partial E}{\partial w_{12}^1} = 0.0101 \cdot 0$$

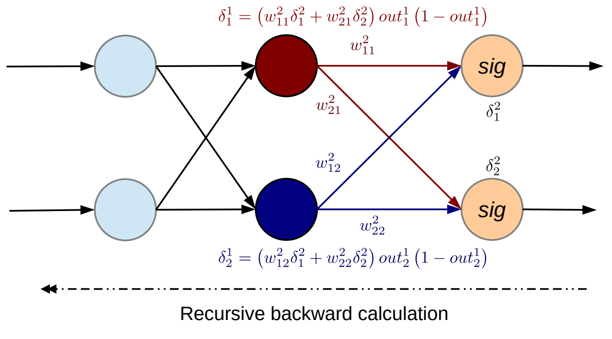
$$\frac{\partial E}{\partial w_{20}^1} = 0.0017 \cdot 1 \quad \frac{\partial E}{\partial w_{21}^1} = 0.0017 \cdot 1 \quad \frac{\partial E}{\partial w_{22}^1} = 0.0017 \cdot 0$$

Backpropagation algorithm

Phase 2: Backpropagation.

If there are several neurons in the next layer, each δ_j^h receives its value from the neurons it is connected to:

$$\delta_j^h \leftarrow \left(\sum_{i=1}^{n_{h+1}} w_{ij}^{h+1} \delta_i^{h+1} \right) \cdot out_j^h \cdot (1 - out_j^h)$$



Backpropagation algorithm

Phase 3: Weight updating.

- Once we have obtained the gradient vector, we must update the weights.
- The value of the derivative itself is used (above all, its sign), multiplied by a constant *eta* (η) which controls that the steps taken are not too small or too large (learning rate).
- General equation:

$$w_{ji}^h = w_{ji}^h - \eta \Delta w_{ji}^h$$

$$\Delta w_{ji}^h = \frac{\partial E}{\partial w_{ji}^h} = \begin{cases} \delta_j^h \cdot 1, & \text{if } i = 0 \\ \delta_j^h \cdot out_i^{h-1}, & \text{if } i \neq 0 \end{cases}$$

Backpropagation algorithm

Phase 3: Weight updating.

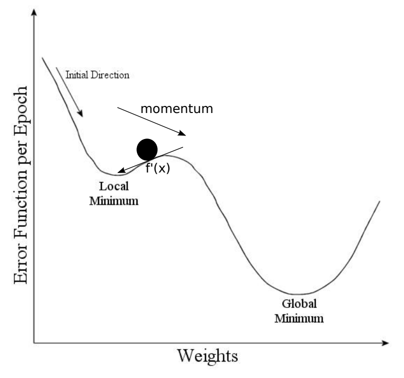
- Adjusting the value of *eta* is difficult:
    - If it is too big, we can cause oscillations.
    - If it is too small, we will need many iterations.
- We will use the concept of momentum, to improve convergence:
    - The previous changes should influence the current direction of movement:

$$w_{ji}^h = w_{ji}^h - \eta \Delta w_{ji}^h - \mu \left(\eta \Delta w_{ji}^h (t-1) \right)$$

    - Thus, the weights that start to move in a certain direction tend to move in that direction
    - The parameter *mu* (μ) controls the effect of the moment.

Backpropagation algorithm

- The idea is that in an example like the one below, the momentum makes it possible to escape from the local optimum.
- Even if the derivative tells you to go left, the "*inertia*" can lead you to go to the right:



Backpropagation algorithm

- So far, we have seen how to adjust the weights according to the error made in a single pattern.
- There are different ways to adapt the network according to the way the training information (patterns) is used.
  - Off-line learning (*batch*):
    - Each time we update the weights of the model connections, we consider all the training patterns.
    - One iteration is usually called *epoch*.
    - Expensive if there is a lot of data.
  - On-line learning:
    - For each training pattern, we perform a weight update.
    - Problem of “forgetting” “old” patterns.
  - Intermediate methods $\Rightarrow$ updating every p patterns (*mini batches*).

    $\Rightarrow$ Problem: you have to define the *batch* size.

Backpropagation algorithm

On-line backpropagation

Start

1. $w_{ji}^{h} \leftarrow U[-1,1]$ *// Random values between -1 and $+1$*
2. **Repeat**
   2.1 **For** each pattern with inputs x and outputs d
      2.1.1 $\Delta w_{ji}^{h} \leftarrow 0$ *// Changes will be applied for each pattern*
      2.1.2 $out_j^0 \leftarrow x_j$ *// Feed inputs*
      2.1.3 forwardPropagation() *// Forward propagation ($\Rightarrow\Rightarrow$)*
      2.1.4 backPropagation() *// Error backpropagation ($\Leftarrow\Leftarrow$)*
      2.1.5 accumulateChange() *// Obtain the weight update*
      2.1.6 weightAdjustment() *// Apply the calculated update*
      End for
   Until (StopCondition)
3. **Return** weight matrices.

End

Backpropagation algorithm

Off-line backpropagation

Start

1. $w_{ji}^{h} \leftarrow U[-1,1]$ *// Random values between* -1 *and* $+1$
2. **Repeat**
   - 2.1 $\Delta w_{ji}^{h} \leftarrow 0$ *// Changes will be applied at the end*
   - 2.2 **For** each pattern with inputs x and outputs d
     - 2.2.1 $out_{j}^{0} \leftarrow x_j$ *// Feed inputs*
     - 2.2.2 forwardPropagation() *// Forward propagation (⇒⇒)*
     - 2.2.3 backPropagation() *// Error backpropagation (⇐⇐)*
     - 2.2.4 accumulateChange() *// Obtain the weight update*

     End for
   - 2.3 weightAdjustment() *// Apply the calculated update*

   Until (StopCondition)
3. **Return** weight matrices.

End

Backpropagation algorithm: functions

Input function: weighted sum.

Activation function: $g(x)$.

- Sigmoid: $g(x) = \frac{1}{1+\exp(-x)}$, $g'(x) = g(x)\,(1 - g(x))$.
- Hyperbolic tangent: $g(x) = \frac{1-\exp(-2x)}{1+\exp(-2x)}$,
  $g'(x) = \left(1 - (g(x))^2\right)$
  Demonstrate.

Backpropagation algorithm: functions

<u>forwardPropagation()</u>

Start

1. **For** h from 1 to H *// For each layer (⇒⇒)*
   1.1 **For** j from 1 to n_h *// For each neuron of layer h*
      1.1.1 $net_j^h \leftarrow w_{j0}^h + \sum_{i=1}^{n_{h-1}} w_{ji}^h out_i^{h-1}$
      1.1.2 $out_j^h \leftarrow g\left(net_j^h\right)$
      End For

   End For

End

Backpropagation algorithm: functions

backPropagation()

Start

1. **For** j from 1 to n_H *// For each output neuron*
   1.1 $\delta_j^H \leftarrow -\left(d_j - out_j^H\right) \cdot g'(net_j^H)$ *// We have eliminated the constant (2), the result should be similar*

   End For
2. **For** h from $H-1$ to 1 *// For each layer (⇐⇐)*
   2.1 **For** j from 1 to n_h *// For each neuron in layer h*
      2.1.1 $\delta_j^h \leftarrow \left(\sum_{i=1}^{n_{h+1}} w_{ij}^{h+1}\delta_i^{h+1}\right) \cdot g'(net_j^h)$ *// Navigate all neurons in layer $h+1$ connected with neuron j*

      End For

   End For

End

accumulateChange()

Start

1. **For** h from 1 to H *// For each layer ($\Rightarrow\Rightarrow$)*
   - 1.1 **For** j from 1 to n_h *// For each neuron of layer h*
     - 1.1.1 **For** i from 1 to n_{h-1} *// For each neuron of layer $h-1$*
       $\Delta w_{ji}^h \leftarrow \Delta w_{ji}^h + \delta_j^h \cdot out_i^{h-1}$
       End For
     - 1.1.2 $\Delta w_{j0}^h \leftarrow \Delta w_{j0}^h + \delta_j^h \cdot 1$ *// Bias*

     End For

   End For

End

Backpropagation algorithm: functions

weightAdjustment()

Start

1. **For** h from 1 to H *// For each layer ($\Rightarrow\Rightarrow$)*
   1.1 **For** j from 1 to n_h *// For each neuron of layer h*
      1.1.1 **For** i from 1 to n_{h-1} *// For each neuron of layer $h-1$*
      $w_{ji}^{h} \leftarrow w_{ji}^{h} - \eta \Delta w_{ji}^{h} - \mu \left(\eta \Delta w_{ji}^{h} (t-1) \right)$
      End For
      1.1.2 $w_{j0}^{h} \leftarrow w_{j0}^{h} - \eta \Delta w_{j0}^{h} - \mu \left(\eta \Delta w_{j0}^{h} (t-1) \right)$ *// Bias*
   End For

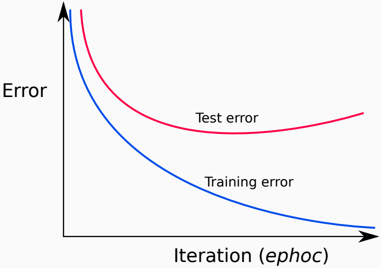
   End For

End

Backpropagation algorithm: problems

- The complexity of the algorithm is polynomial depending on the number of weights in the network.
- The architecture of the network (number of layers and number of nodes in each layer), together with the typology of the nodes (type of activation functions to be considered), are decisive parameters of the algorithm that must be searched for by trial and error or by cross validation.
- The initialisation of the weights can lead to the algorithm being trapped in local minima.
  - In the *online* version, we can randomize the order of presentation of the patterns
  - We can introduce noise into the training patterns (also prevents over-fitting).

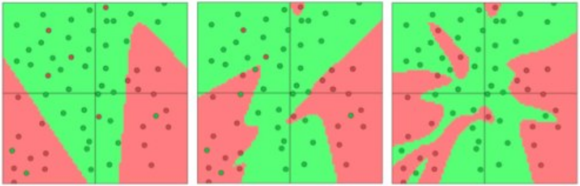
Backpropagation algorithm: problems



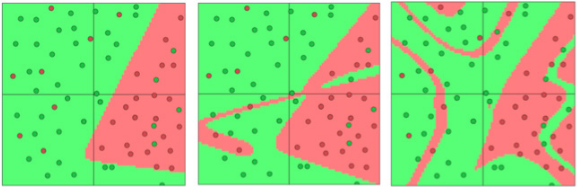
- Over-fitting in neural networks is usually caused by two facts:
    - Too long training (improper stop condition).
    - Too complex networks (many neurons or many layers).

Backpropagation algorithm: problems

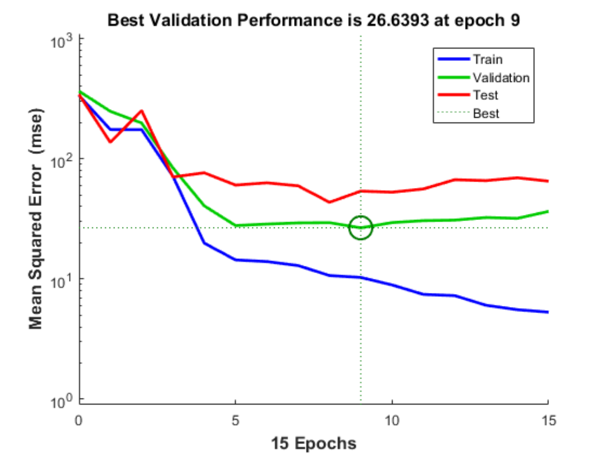
A layer with 3, 6 and 20 neurons:



One, two and four layers, with 3 neurons each:



Early stopping: mechanism to detect when over-fitting is happening.



Backpropagation algorithm: problems

Early stopping: mechanism to detect when over-fitting is happening.

1. We divide the data into three parts: training (e.g. 60%), validation (e.g. 20%) and test (e.g. 20%).
2. Weights are adjusted using the training set.
3. The network is evaluated using the validation set.
4. If the error is decreased more than t (tolerance), return to step 2. Otherwise, stop training.
5. Evaluate the model using the test set.

Backpropagation algorithm: problems

Regularization: mechanism to avoid over-fitting (minimizes the magnitude of the weights).

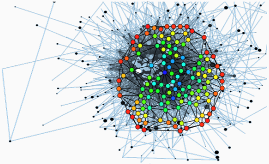
- L2 regularization:

$$E = \frac{1}{J}\sum_{i=1}^{J}(d_i - o_i)^2 + \lambda \sum_h \sum_j \sum_i \left(w_{ji}^h\right)^2 \tag{3}$$

- Modifying the derivatives is direct:

$$\frac{\partial E}{\partial w_{ji}^h} = \begin{cases} \delta_j^h \cdot 1 + 2\lambda w_{ji}^h, & \text{if } i = 0 \\ \delta_j^h \cdot out_i^{h-1} + 2\lambda w_{ji}^h, & \text{if } i \neq 0 \end{cases}$$

Contents



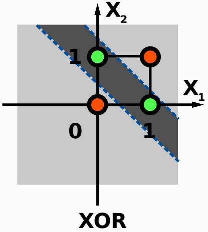
Introduction
Artificial neuron
Neural networks
Learning
Classification
ANNs using Weka software
Conclusions

Classification

- Motivation: We often encounter real-world problems where the aim is "*categorising*" or "*classifying*" according to a set of characteristics.
- We need a predictive model that, from a database, is capable of obtaining the value of a categorical or nominal variable.
- For example:
    - Eye colour: $\{\text{blue}, \text{green}, \text{brown}\}$.
    - Success or failure of a treatment: $\{\text{yes}, \text{no}\}$.
    - Presence of cancer in a picture of an organ: $\{\text{yes}, \text{no}\}$.

Classification

- The XOR problem is also a problem of classification:



- Actually, classification is an intrinsically non-linear task because we try to put things that are not the same into the same group, i.e. a difference is the vector of inputs does not cause a difference in the output of the model (the category is the same).

Representation of class values

- Representation of class values (eye colour):

Type	Blue class	Green class	Brown class
Integer values	1	2	3

- Using this representation, we could use the multi-layer percetron for regression, so that the predicted class would be the integer closest to the value predicted by the model.
- Disadvantages:
    - It has been assumed that there is an order between classes.
    - A distance has been assumed between each of the classes.

Representation of class values

- Representation 1-of-J, where J is the number of classes.
    - For each pattern, we will have a vector of J elements, where the i-th element will be equal to 1 if the pattern belongs to that class and to 0 if it does not.
    - That is, the vector will contain 0 in all positions except the position that corresponds to the correct class (in which there will be a 1).
- Representation of class values (eye colour):

Type	Blue class	Green class	Brown class
1-of-J	$\{1,0,0\}$	$\{0,1,0\}$	$\{0,0,1\}$

- Using this representation, the multilayer perceptron will model each of the J binary variables separately.
- One output neuron per class (model three variables at a time).

Representation of class values

- Predicted class: neuron with the maximum output value.
- If we use a sigmoid function in the output layer, we ensure that the predicted values will be between 0 and 1.

d_1	d_2	d_3	o_1	o_2	o_3	Predicted class
0	0	1	0.1	0.2	0.8	3
0	0	1	1.0	0.2	0.8	1
0	1	0	0.2	0.9	0.1	2

- Problem: inconsistencies, as the variables are being modelled independently.
- Solution: incorporating a probabilistic meaning into the outputs.

Probabilistic interpretation

- If we think about it, the 1-of-J representation can be seen as a probabilistic representation of a series of events:
  - J classes: $\{C_1, C_2, \ldots, C_J\}$.
  - J events: the pattern belongs to each of the classes of the problem, i.e. "$\mathrm{x} \in C_1$", "$\mathrm{x} \in C_2$", ..., "$\mathrm{x} \in C_J$".
  - The desired output (d) is to predict that the pattern belongs to the correct class with the highest probability (1-of-J output):

$$d_j = \begin{cases} 1 & \text{if } \mathrm{x} \in C_j \\ 0 & \text{if } \mathrm{x} \notin C_j \end{cases} \tag{4}$$

  - In this way, what we model and predict is the probability of belonging to each of the classes:

$$o_j = \hat{P}(\mathrm{x} \in C_j | \mathrm{x}) \tag{5}$$

The softmax function

- Now we need the outputs of the neural network (o) to be "consistent", from a probabilistic point of view:

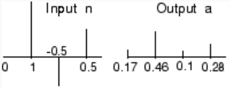
$$\sum_{j=1}^{J} o_j = 1 \tag{6}$$

- To do this, we can use the softmax function, which is a normalization function that ensures that the outputs will be between 0 and 1 and that their sum will be 1:

$$net_j^H = w_{j0} + \sum_{i=1}^{n_{H-1}} w_{ji} out_i^{H-1}, \tag{7}$$

$$out_j^H = \frac{\exp(net_j^H)}{\sum_{l=1}^{n_H} \exp(net_l^H)} \tag{8}$$

The softmax function



$$o_j = out_j^H = \frac{\exp(net_j^H)}{\sum_{l=1}^{n_H} \exp(net_l^H)} \tag{9}$$

- It is a plausible approximation (from the biological point of view) and derivable to the maximum function.
- The exponential (exp) ensures that we will deal with positive amounts and "exaggerate" the outputs a lot, so that the result looks like the maximum function (a 1 for the maximum and a 0 for the rest).

The softmax function

- The denominator, $\sum_{l=1}^{n_H} \exp(net_l^H)$, normalise these positive amounts, since it is the sum of all of them. In this way, we achieve the fulfillment of the probability calculation axiom:

$$\sum_{j=1}^{k} o_j = 1 \tag{10}$$

- It therefore produces probabilistically correct outcomes.
- The predicted class will be the index of the output neuron with the highest value:

$$C(\mathrm{x}) = \arg\max_j o_j \tag{11}$$

The softmax function

net_1^H	net_2^H	net_3^H	$e^{net_1^H}$	$e^{net_2^H}$	$e^{net_3^H}$	$\sum_{l=1}^{n_H} e^{net_l^H}$
-1	0	3	0.368	1.000	20.086	21.454
1	4	-1	2.718	54.598	0.368	57.684
0.4	0	3	1.492	1.000	20.086	22.578

out_1^H	out_2^H	out_3^H	Predicted class
0.017	0.047	0.936	3
0.047	0.947	0.006	2
0.066	0.044	0.890	3

The softmax function: derivatives

$$o_j = out_j^H = \frac{\exp(net_j^H)}{\sum_{l=1}^{n_H} \exp(net_l^H)} \tag{12}$$

- To simplify, let's take $net_j = n_j$ and ignore the limits of the summation:

$$o_j = \frac{e^{n_j}}{\sum_l e^{n_l}} \tag{13}$$

- We want to calculate $\frac{\partial o_j}{\partial n_i}$.
- When calculating derivatives, all the n_l are part of the output of each neuron.

The softmax function: derivatives

We distinguish two cases:

1. $i = j$, i.e. $\frac{\partial o_j}{\partial n_j}$.
2. $i \neq j$, i.e. $\frac{\partial o_i}{\partial n_j}$

First case ($i = j$). Derivative of the output (o_j) with respect something which is behind neuron j:

$$
\begin{aligned}
\frac{\partial o_j}{\partial n_j} = & \qquad \tfrac{\partial}{\partial n_j} \tfrac{e^{n_j}}{\sum_l e^{n_l}} = \tfrac{\partial}{\partial n_j} \left(\textstyle\sum_l e^{n_l}\right)^{-1} e^{n_j} = \\
= & \quad \left((-1) \left(\textstyle\sum_l e^{n_l}\right)^{-2} e^{n_j} \right) e^{n_j} + \left(\textstyle\sum_l e^{n_l}\right)^{-1} e^{n_j} = \\
= & \quad -\frac{\left(e^{n_j}\right)^2}{\left(\sum_l e^{n_l}\right)^2} + \tfrac{e^{n_j}}{\sum_l e^{n_l}} = -o_j^2 + o_j = o_j(1 - o_j)
\end{aligned}
$$

The softmax function: derivatives

Second case ($i \neq j$). Derivative of the output (o_j) with respect something which is behind a neuron i that is not j:

$$
\begin{aligned}
\frac{\partial o_j}{\partial n_{i|i\neq j}} &= \tfrac{\partial}{\partial n_i}\tfrac{e^{n_j}}{\sum_l e^{n_l}} = e^{n_j}\tfrac{\partial}{\partial n_i}(\textstyle\sum_l e^{n_l})^{-1} = \\
&= e^{n_j}\left((-1)(\textstyle\sum_l e^{n_l})^{-2} e^{n_i}\right) = \\
&= -\tfrac{e^{n_j}}{(\sum_l e^{n_l})} \cdot \tfrac{e^{n_i}}{(\sum_l e^{n_l})} = -o_j o_i
\end{aligned}
$$

Both can be summarised in the next way:

$$\frac{\partial o_j}{\partial n_i} = o_j(I(i=j) - o_i)$$

where $I(cond)$ will be 1 if *cond* is true and 0 otherwise.

The softmax function: derivatives

- For a *softmax* neuron, the δ_j^h value must be obtained by adding the derivatives with respect all the net_j^h:

$$\delta_j^h = \sum_{i=1}^{n_h} out_j^h (I(i=j) - out_i^h) \tag{14}$$

where out_j^h is the *softmax* transformation:

$$out_j^h = \frac{\exp(net_j^h)}{\sum_{l=1}^{n_h} \exp(net_l^h)} \tag{15}$$

Performance measure: *CCR*

- In a classification task, our aim should be that the classifier almost always gets the class right.
- *Correctly Classified Ratio* or percentage of well classified patterns:

$$CCR = 100 \times \frac{1}{N} \sum_{p=1}^{N} \left(I(y_p = y_p^*) \right) \quad (16)$$

  - N: number of patterns.
  - y_p: desired class for pattern p, $y_p = \arg\max_o d_{po}$.
    - Index of the maximum value of vector d_p.
  - y_p^* class obtained for class p, $y_p^* = \arg\max_o o_{po}$.
    - Index of the maximum value of vector o_p or the output neuron that gets the maximum probability for the pattern p.

- We could train the MLP by trying to maximize this amount, but there is a problem:
  - To obtain y_p e y_p^*, we have to apply the arg max function, which is not derivable.
  - Moreover, *CCR* improves in steps, which would not allow us to gradually adjust the weights $\Rightarrow$ Difficult convergence.
- Again, we can use the *MSE* as error function (to be minimised) using the 1-of-J coding for the outputs and the probabilities predicted by the *softmax* function:

$$MSE = \frac{1}{N}\sum_{p=1}^{N}\left(\frac{1}{J}\sum_{o=1}^{J}(d_{po} - o_{po})^2\right) \tag{17}$$

Performance measure: cross-entropy

- The mean square error (MSE) is not the natural error function when we have probabilistic outputs, since it equally treats any error difference.
- For classification problems, we should penalise more the errors made for the correct class ($d_j = 1$) than for the incorrect one ($d_j = 0$).
- Cross entropy ($-\ln$ likelihood) is more suitable for classification problems because it compares the two probability distributions:

$$L = -\frac{1}{N \cdot J} \sum_{p=1}^{N} \left(\sum_{o=1}^{J} d_{po} \ln(o_{po}) \right) \quad (18)$$

Performance measure: cross-entropy

- Given the training set, training a classification algorithm by minimizing this error function involves estimating the parameters that maximize the likelihood of my coefficients (*maximum likelihood estimation*).
- The derivative is obtained in a similar way (for a single pattern and ignoring the constant $\frac{1}{J}$):

$$L = -\sum_{l=1}^{J} d_l \ln(o_l)$$

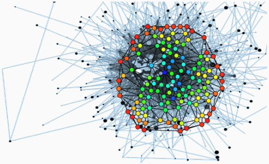
$$(\ln(x))' = \frac{1}{x}$$

$$\frac{\partial L}{\partial w_{ji}^h} = -\sum_{l=1}^{J} \frac{\partial L}{\partial o_l} \frac{\partial o_l}{\partial w_{ji}^h} = -\sum_{l=1}^{J} \left(\frac{d_l}{o_l}\right) \frac{\partial o_l}{\partial w_{ji}^h}$$

Summary of calculation of δ_j^h

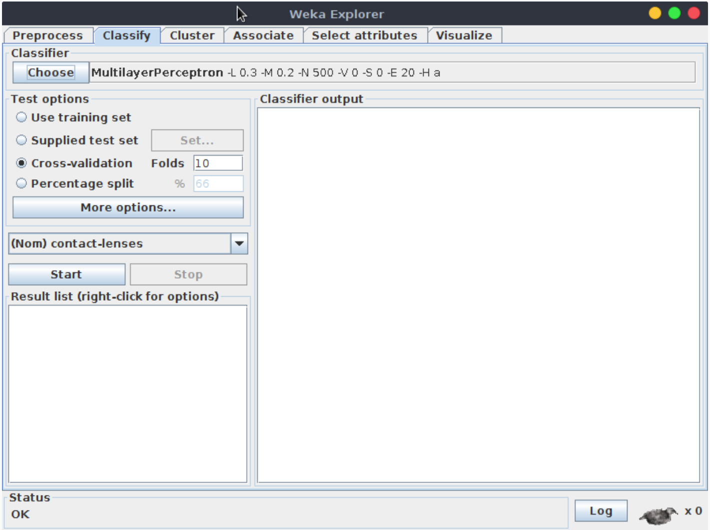
- Derivatives for sigmoid or hyperbolic tangent neurons
  - Output layer:
    - *MSE*:
      $\delta_j^H \leftarrow -\left(d_j - out_j^H\right) \cdot g'(net_j^H)$
    - Cross-entropy:
      $\delta_j^H \leftarrow -\left(d_j / out_j^H\right) \cdot g'(net_j^H)$
  - Hidden layers:
    $\delta_j^h \leftarrow \left(\sum_{i=1}^{n_{h+1}} w_{ij}^{h+1} \delta_i^{h+1}\right) \cdot g'(net_j^h)$
- *softmax* neurons:
  - Output layer:
    - Error *MSE*:
      $\delta_j^H \leftarrow -\sum_{i=1}^{n_H} \left(\left(d_i - out_i^H\right) \cdot out_j^H (I(i=j) - out_i^H)\right)$
    - Cross-entropy:
      $\delta_j^H \leftarrow -\sum_{i=1}^{n_H} \left(\left(d_i / out_i^H\right) \cdot out_j^H (I(i=j) - out_i^H)\right)$

Contents

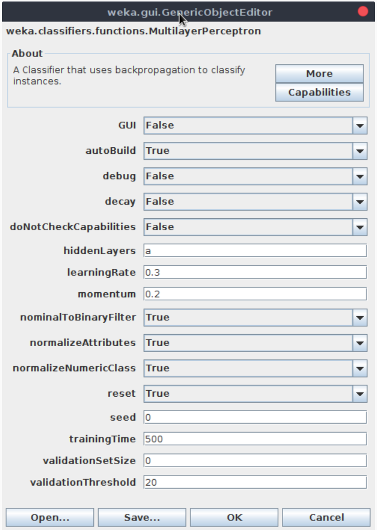


Introduction
Artificial neuron
Neural networks
Learning
Classification
ANNs using Weka software
Conclusions

functions → MultilayerPerceptron



Parámetros MultilayerPerceptron



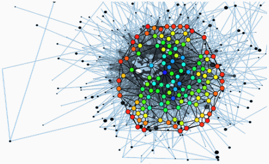
Parámetros MultilayerPerceptron

- `GUI`: allows to use an interactive interface in the construction of the network (if we use the GUI, `autoBuild` builds us the network).
- `decay`: decreases the learning factor as the epochs increase.
- `hiddenLayers`: network architecture.
  - `20`: a hidden layer of 20 neurons.
  - `4,4`: two hidden layers of four neurons.
  - Some heuristics included in Weka:
    - `a`: (attribs + classes) / 2.
    - `t`: attribs + classes.
- `learningRate`: learning rate (η).
- `momentum`: momentum factor (μ).
- `nominalToBinaryFilter`: converting nominal attributes into binary.
- `normalizeAttributes`: normalize inputs in the range $[-1, 1]$.

Parámetros MultilayerPerceptron

- `normalizeNumericClass`: for regression problems, normalize the variable to be predicted (it may help).
- `reset`: avoid divergence. If during the learning process, the predictions diverge too much from the target values, then we reset the algorithm by lowering the value of the learning rate.
- `seed`: seed for random numbers.
- `trainingTime`: maximum number of epochs.
- `validationSetSize`: if 0, we do not use *early stopping*. If not, this value will be the percentage of the dataset used as validation for *early stopping*.
- `validationThreshold`: number of iterations during which the validation error has to go up for stopping the algorithm.

Contents



Introduction
Artificial neuron
Neural networks
Learning
Classification
ANNs using Weka software
Conclusions

Conclusions

- Advantages:
  - Usually high accuracy rate in prediction.
  - Robustness in the presence of errors (noise, outliers...).
  - Great adaptability: nominal output, numerical, vectors...
  - Efficiency (speed) in the evaluation of new cases.
  - They improve their performance through learning and this can continue after it has been applied to the dataset.
- Disadvantages:
  - They need a lot of time for training.
  - Many parameters. Training is largely trial and error (architecture, learning rate, momentum...).
  - Low interpretability of the model (black box models).
  - It is difficult to incorporate prior knowledge of the domain.
  - Input attributes must be numerical.
  - They can produce overfitting.

Conclusions: a bit of history

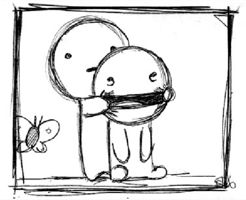
- Year 58: emergence of the simple perceptron, initial enthusiasm.
- 1970s: abandonment of nueral networks (limitations in complex problems).
- 80s and 90s: Boom in neural networks thanks to the multilayer perceptron and the backpropagation algorithm. Use in many fields.
- 90s-2010: its use was superseded by other models (decision trees, support vector machines). Reasons
  - Expensive learning process (in CPU and expert time).
  - Low interpretability.
  - It was not possible to train many-layered networks (gradient vanishing problem when applying the chain rule).
- 2010-Now: reborn of neural networks, "*Deep learning*".

Conclusions: deep learning

- A set of techniques that allow the use of networks with many layers and many neurons.
- Very important applications in computer vision (object recognition, image classification...), natural language processing (sequence translation, summarization...)
- Also driven by the availability of computing resources and the use of specific hardware architecture (GPU-based).
- Example: GoogleNet Inception v4 (convolutional architecture, adapted for images)
  - Neural network trained with 1.28 million images to distinguish between 1000 categories of objects.
  - More than 60 million weights in the network.

¿Preguntas?

¡Gracias!



Bibliografía adicional i

Christopher M. Bishop.
Pattern Recognition and Machine Learning.
Springer, 1st ed. 2006. corr. 2nd printing edition, August 2007.

Fernando Berzal.
Redes Neuronales & Deep Learning.
Edición independiente, 1 edition, 2018.

Christopher M. Bishop.
Neural Networks for Pattern Recognition.
Oxford University Press, USA, 1 edition, January 1996.