2. Perceptron and Learning

INTRODUCTION

As seen in the previous report, the MCP neuron represents an abstraction of the biological neuron to a simple input/output processing system. By linking together multiple MCPs, distributed in layers, we attain the so called feed-forward neural network (FFNN). FFNNs (Fig. 1) with opportune neuron distribution and activation functions, have been proved to show general processing abilities, typical of complex non-linear systems, as for example the ability to classify input patterns into classes, to compress high dimensional input data into low dimensional output data, to approximate continuous functions to cite few.
It is said that complex FFNNs show increasing emergent properties that are not easily predictable from the analysis of the behavior of a pool of neurons taken singularly.
The simplest FFNN network has been termed "Perceptron". The elementary idea underlying the Perceptron is that, using a unique computational basis (MCP neuron) linked into a network, one can achieve in principle very different behaviors (functions).
The simplest Perceptron is endowed with an input layer with N signals and an output layer with 1 single MCP neuron characterized by the signum activation function. It was shown that this Perceptron is able to produce a linear separation (binary) of the input space (McCulloch and Pitts, 1943). This means that for an N-dimensional input, the output neuron creates a hyper-plane in the input space that separates the input signals into two classes. In 1958, Rosenblatt disclosed the concept of parameter learning for this kind of ANN according to the concept of stimulus-response with reinforcement using a supervisor. This approach considers the error correction principle. The network parameters (weights and thresholds) undergo update by iteratively minimizing the difference of the predicted network output and a reference (desired network output), which is provided by the supervisor. In 1960, Widrow and Hoff extended error correction approach proposing a gradient-based technique named Delta rule. This new approach was shows to be more reliable than Rosenblatt approach, requiring differentiable activation functions though. In 1969, Minsky and Papert demonstrated that the basic Perceptron (no hidden layers) could only classify linearly separable input patterns, acknowledging that more complex layered networks, with non-linear activation functions, would be required. However, the learning based on Delta rule was unable to train multilayer networks. This finding had a negative impact of the potentiality of feed-forward networks and the research in this field suffered a setback. In 1986, Rumelhart proposed a new methodology to train multi-layer Perceptron networks based on the innovative concept of back-propagation of the error throughout the network demonstrating a general learning ability. This renewed the interest in ANN as classification systems, spreading in the following years the application of FFNN to several application fields.

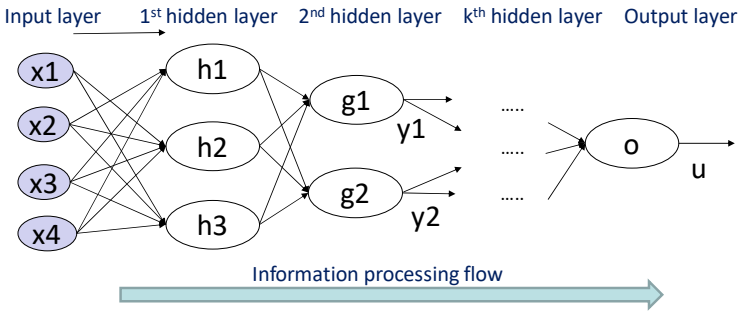


Figure 1. Feed-forward neural network.

LEARNING AND NEURAL PLASTICITY

One general definition of learning is: "The process by which an activity/ability is either originated or modified by reacting to a situation, unless the characteristics of the activity change cannot be explained on the basis of innate response tendencies, current organism states." According to this principle, we can assert that learning is not a pre-programmed skill (it needs external stimuli) or can be explained by looking at the topology of the neural structure (neural cells and current interconnections).
At neural level, learning means basically increasing of the efficiency of the neural signal transmission. We know that the signal exchange from one neuron to another neuron occurs at synaptic level as the electrical signal running along the axon is transmitted between the axon end of the input neuron and the dentric spine of the output neuron by means of a molecular exchange at such an interface. Therefore, learning

corresponds to act on the signal transmission by improving the efficiency of the synaptic exchange. This means that activating the sequence of two neurons will require less and less energy as the learning increases (Fig. 3). The concept of learning strongly relies on neural plasticity.

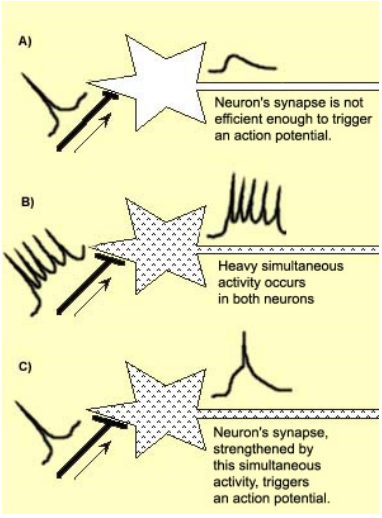


Figure 3. Process of neural plasticity

Neural plasticity just refers to the ability of the neurons to make changes in the efficiency of their synaptic transmission by increasing the protein connections. The process has direct effects on specializing the function of neuron populations. For example, when studying we activate certain memory cells that are connected in between in unique ways. Over time, the neurons that "remember" a set of names (e.g. roman emperors) are activated over and over again through electrical impulses. These impulses cause the cell body or nucleus to manufacture and place certain proteins in the membranes that "cement" the pathways between the cells, creating memories. Recurrent activation binds the cells more closely and allows us to remember what we have studied. Over time, neurons increase coordination in between. It is said that a coherent network has been created. Typically, a coherent network of neurons carries out a function. In general, any learning process, performed by repetitions throughout time, is termed "reinforcement learning".

Neurons are living units that grow, change and adapt over time. When the list of "roman emperors" names is not reinforced over time, the earlier protein connections are progressively lost along with the memories. The proteins that were made to build the intercellular links degrade and are not replaced. Over time, one will probably lose the memory of such names because you do not continually reinforce the information but you will keep the memory of your best friend because you use this information every day and these critical intercellular links and memories are continually renewed.

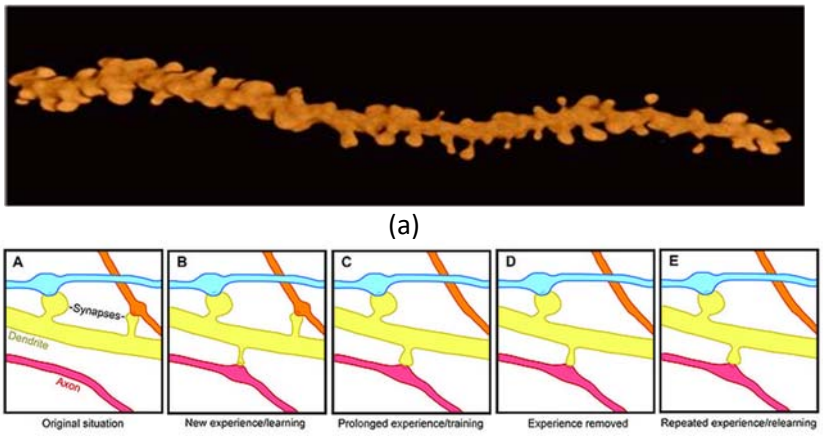


Figure 4. Dendrite with dendritic spines. (a). Simulation of the formation of a new neural pathway (b).

This kind of plasticity is usually called "long-term" plasticity as the neurons require a wide temporal span to establish and reinforce the interconnections. Short-term plasticity is in contrast a mechanism that change, even dramatically, the function of a population of neurons under various agent effects. For example, endogenous agents (neurotransmitters as for instance the serotonin) act as neuromodulator to temporarily deactivate or activate neural pathways. Exogenous agents as drugs, anesthetics and stimulants can exert intense effects on network functions. When the effects of agents is concluded, the population of neurons can take back its original function.

It has been verified that cells, transferring electrical impulses, boost receiving dendrites to grow, over time, towards the stimulus and eventually create new pathways for transmission of that message. These microscopic receivers are called dendritic spines (Fig. 4). The growth of new dendritic spines (synaptogenesis) is the important mechanism that occurs for example as a consequence of stroke rehabilitation when the patient is relearning to walk, talk or reacquire any type of information that was lost (usually in long-term memory) because of stroke. Activating a new path to process information or electrical activity takes place over time. The amount of time depends on the severity of damage, the location of the damage in the brain, and the ability of the patient to participate actively in the rehabilitation procedure

necessary to build the new electrical pathways. This is the process commonly known as ”rewiring the brain”.
Dendritic spines are very "plastic", that is, spines change significantly in shape, volume, and number in small time steps. The majority of spines change their shape within seconds to minutes. Spine plasticity applies to motivation, learning, and memory. In particular, long-term memory is mediated in part by the growth of new dendritic spines (or the enlargement of pre-existing spines) to reinforce a particular neural pathway. Because dendritic spines are plastic structures whose lifespan is influenced by input activity, spine dynamics may play an important role in the maintenance of memory over a lifetime.
Age-dependent changes in the rate of spine turnover suggest that spine stability affects the developmental learning. In youth, dendritic spine turnover is relatively high and produces a net loss of spines, with the rate of the elimination of spines surpassing the rate of the formation of spines. This high rate of spine turnover may characterize critical periods of development and reflect learning capacity in youth—different cortical areas exhibit differing levels of synaptic turnover during development, possibly reflecting varying critical periods for specific brain regions. In adult age, however, most spines remain persistent, and the half-life of spines increases. This stabilization occurs due to a developmentally regulated slow-down of spine elimination, a process that may underlie the stabilization of memories in maturity.
Modifications in dendritic spine stability induced by experience also refer to spine changes as a mechanism involved in the preservation of long-term memories, despite the fact that it is unclear how sensorial experience affects neural circuitry. Two different models may describe the effect of experience on structural plasticity. Experience and activity can promote the discrete development of relevant synaptic connections storing significant information for learning. On the other hand, synaptic connections may be formed in excess, and experience and activity may lead to the pruning of extraneous synaptic connections.
Skill training has been shown to lead to the formation and stabilization of new spines while undermining old spines, suggesting that the learning of a new skill involves a rewiring process of neural circuits. Since the extent of dendritic spine remodeling correlates with the success of learning, this suggests a crucial role of synaptic structural plasticity in memory formation. In addition, changes in spine stability and strengthening occur rapidly and have been observed within hours after training (lab animal model experiments). Conversely, while enrichment and training are related to increase in spine formation and stability, long-term sensory deprivation leads to an increase in the rate of spine elimination and therefore impacts long-term neural circuitry. Upon restoring sensory experience after deprivation in adolescence, spine elimination is accelerated, suggesting that experience plays an important role in the net loss of spines during development.
Research in neurological disorders and injuries shed further light on the nature and significance of spine turnover. After stroke, a noticeable increase in structural plasticity occurs near the trauma site, and a five- to eightfold increase from control rates in spine turnover has been observed. Dendrites disintegrate and reassemble rapidly during ischemia (e.g. due to stroke), with survivors showing an increase in dendritic spine turnover. While a net loss of spines is observed in Alzheimer's disease and cases of mental retardation, cocaine and amphetamine use have been linked to increases in dendritic branching and spine density in the prefrontal cortex and the nucleus accumbens. Because significant changes in spine density occur in various brain diseases, this suggests a balanced state of spine dynamics in normal circumstances, which may be susceptible to disequilibrium under varying pathological conditions. There is also some evidence for loss of dendritic spines as a consequence of aging. One study using mice has noted a correlation between age-related reductions in spine densities in the hippocampus that and age-dependent declines in hippocampal learning and memory.

BOOLEAN NEURAL NETWORKS AS CLASSIFIER TOOLS

The original application of ANN to the pattern classification problem encompassed the attempt to model boolean operators to represent logic functions like NOT, AND, OR. The simplest network, utilized to this aim, was the Perceptron, i.e. a FFNN composed of MCP neurons with step or signum activation function and threshold T (Fig. 2).
It can be easily seen that, using a single MCP neuron with signum activation function, it is possible to obtain the logic port NOT. Assuming that $u = sgn(w*x-T)$, setting $(w=-1)$ and $(-1<T<0)$, an input $x = 1$ will lead to an

output u = 0. Conversely, an input x = 0 will lead to an output u = 1. Now let assume that the neuron has two input x_1 and x_2 so that $u = sgn(w_1*x_1 + w_2*x_2-T)$. If we set ($w_1 = 1$), ($w_2 = 1$) and ($0<T<1$), then the MCP is behaving like an OR port. Contrary, ($w_1 = 1$), ($w_2 = 1$) and ($1<T<2$) allow to differentiate the MCP to an AND port.

As depicted in figures 3a and 3b, the equation $w_1*x_1 + w_2*x_2-T=0$ was plot as a line in the input space (x_1,x_2) for the OR and AND ports with S=0.5 and T=1.5, respectively. Such a line is called decision boundary and divides the input space into two regions. The upper and lower regions are the positive and the negative neuron outputs for all the four input combinations. By construction, the weight vector is always orthogonal to the decision boundary.

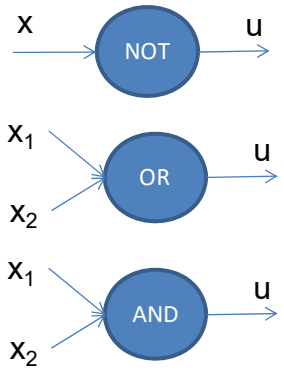


Figure 2. Logic ports.

Let now consider the XOR port. The XOR should create a separation among both input sets (0,0) and (1,1), and the other two input sets (0,1) and (1,0) (Fig. 3c). It is clear however that a single decision boundary cannot represent such a classification.

The intuitive approach would be to consider two output neurons u_1 and u_2, able to generate two independent decision boundaries. With two output neurons we can generate four different classes (u1=0, u2=0), (u1=1, u2=0), (u1=0, u2=1), and (u1=1, u2=1) (Fig. 4a). Let consider the first and the last classes as the target of our representation. The correct classifier should associate the class (u1=1, u2=1) to both (0,0) and (1,1) inputs and the class (u1=0, u2=0) to both (1,0) and (0,1) inputs. It can be shown that no combination of parameters w and S is successful for classifying the input pattern accordingly with a single layer of neurons. Consequently, a single-layer MCP network cannot represent all the Boolean operators. One should conversely consider to add one (Fig. 4b) or more layers (Fig. 4c) to the original network.

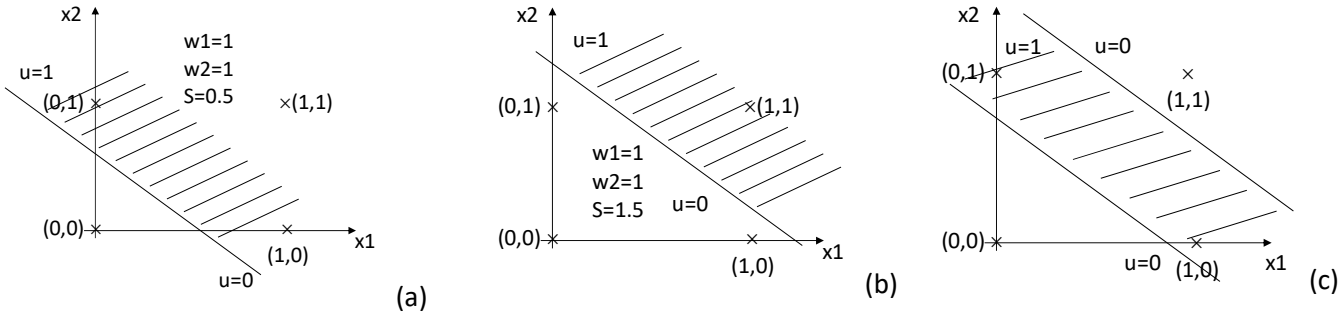


Figure 3. OR (a) and AND (b) port classifiers. Attempting the XOR classification with two output neurons.

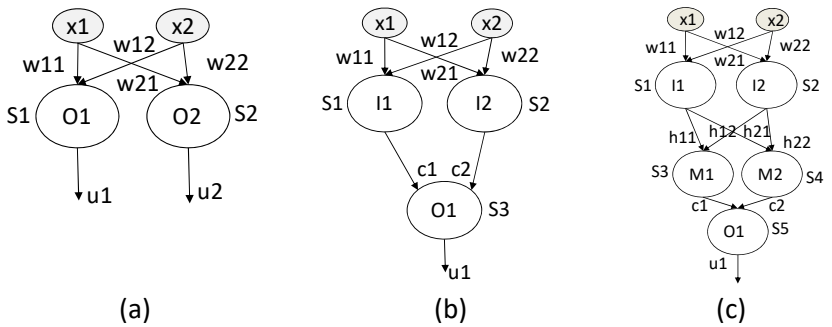


Figure 4. two-, three- and four-layer Perceptrons.

One feasible solution for the XOR port can be represented by the ANN in figure 5 considering the following relation:

$$A \underline{\cup} B = (A \cup B) \cap (\sim A \cup \sim B) = (A \cup B) \cap \sim(A \cap B)$$

which is the joint between the OR and negation of the AND of the two input. The process can be split sequentially then in three steps as:

u1 = sgn(w11*x1 + w12*x2 – T1) with 1<T1<2 (**AND**) w11 = 1 w12 = 1
u2 = sgn(w21*x1 + w22*x2 – T2) with 0<T2<1 (**OR**) w21 = 1 w22 = 1

v1 = sgn(h1*u1 – T3) with -1<T3<0 (**NOT**) h1 = -1
v2 = sgn(k2*u2 – T4) with 0<T4<1 (**EYE**) k2 = 1

o = sgn(m1*v1 + m2*v2 – T5) with 1<T5<2 (**AND**) m1 = 1 m2 = 1

Notice that, in order to enable layer consistency, an identity neuron is mandatory between the 2nd and 3rd layer. This heuristic solution actually do apply to this particular operator. Alternatively, it was shown that any particular Boolean operator can be learned by an MCP network with an opportune learning procedure.

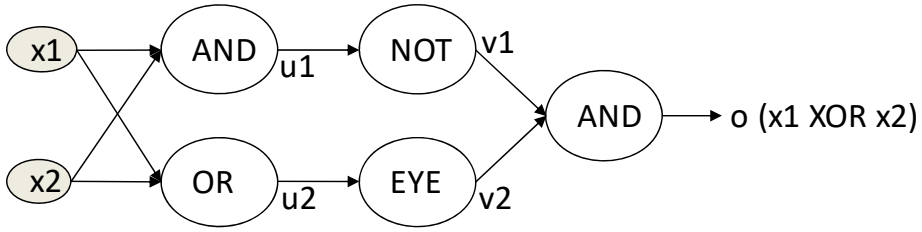
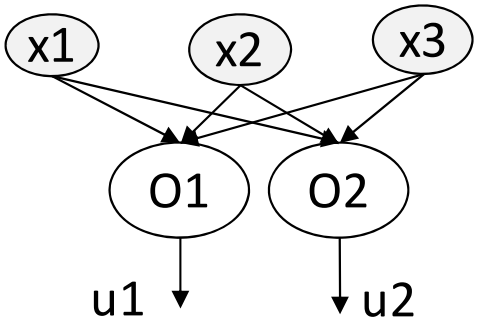


Figure 5. Example of a XOR port implemented by a multi-layer Perceptron.

SUPERVISED LEARNING

As we defined earlier, “Learning is a process by which an activity is either originated or modified by reacting to a situation, unless the characteristics of the activity change cannot be explained on the basis of innate response tendencies, current organism states.” This definition highlights the intrinsic property of the learning process to create new abilities of the system that could not be envisioned by simply looking at the system structure. In the ANN paradigm, learning at network level means to specialize the network to a specific task. Given that the network topology (architecture and activation functions) is given, the ability to

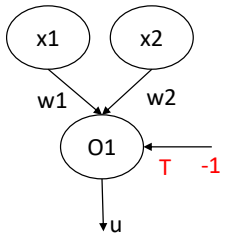


Network		Pattern
output	$u^{(1)}_1, u^{(1)}_2$	1: $(x^{(1)}_1, x^{(1)}_2, x^{(1)}_3)$
reference	$t^{(1)}_1, t^{(1)}_2$	
output	$u^{(2)}_1, u^{(2)}_2$	2: $(x^{(2)}_1, x^{(2)}_2, x^{(2)}_3)$
reference	$t^{(2)}_1, t^{(2)}_2$	
		…
output	$u^{(k)}_1, u^{(k)}_2$	k: $(x^{(k)}_1, x^{(k)}_2, x^{(k)}_3)$
reference	$t^{(k)}_1, t^{(k)}_2$	
		…
output	$u^{(R)}_1, u^{(R)}_2$	R: $(x^{(R)}_1, x^{(R)}_2, x^{(R)}_3)$
reference	$T^{(R)}_1, t^{(R)}_2$	

Figure 8. Example of a Perceptron with M=3 input and N=2 output neurons and a training dataset composed by R different patterns. The signal $u^{(k)}_i$ is the output of the i^{th} neuron in correspondence of the k^{th} pattern. The signal $t^{(k)}_i$ is the reference value of the output of the i^{th} neuron in correspondence of the k^{th} pattern.

perform a task is provided by a specific combination of synaptic weights and neuron thresholds. From now on, we are considering the network training as the process that learns such a combination. In order to address in a compact way the weight computation, the neuron threshold T can be easily regarded as a

weight, associated to a steady input equal to -1 (Fig. 9). We want to remark that a simple change in the notation will allow us to use the same approach in the learning for both neural weights and thresholds. The trick part is here to assume that the threshold is an additional weight of the neuron associated to a virtual constant input equal to -1 (Fig. 9). In the light of this change, from now on, we call *P-T* action potential and we name it with *P*.



$$P = \sum_{j=1}^{2} w_j x_j - T = w_1 x_1 + w_2 x_2 + T(-1)$$

$$P = \sum_{j=1}^{3} w_j x_j = w_1 x_1 + w_2 x_2 + w_3 x_3 \quad with \quad w_3 = \mathrm{T} \; and \; x_3 = -1$$

Figure 9. The threshold can be regarded as additional weight of the neuron associated to a virtual constant input equal to -1.

One of the main mechanisms to train a FFNN is called supervised learning. In the supervision paradigm, a number of different input examples, collected in the training dataset, is to be given along with a number of corresponding desired outputs (output reference dataset). Therefore, any element in the training dataset is a pattern having a corresponding reference (nominal) output. The supervised learning exploits the so called "error signal", which is computed as the difference between the reference output and the output predicted by the network given that actual weight vector. The supervised learning rests on the minimization of such an error signal with respect to the synaptic weights.

PERCEPTRON LEARNING RULE (BINARY UNITS)

Let consider a perceptron with the signum function as neural activation. The perceptron features M input lines and N output neurons. The training dataset is composed by *R* different patterns (cfr. Fig. 8). The corresponding network is intended as a classifier. The perceptron learning rule by Rosenblatt (1962) is based on an incremental procedure for arriving at a solution $\mathbf{w}^*$ starting from an initial arbitrary weight vector $\mathbf{w}_{start}$ by taking into account the difference between the target and the network output.
The perceptron learning rule corrects, iteratively across the patterns in the training set, the weight vector if and only if a misclassification occurs. Let consider a single output neuron with m inputs and *k* training patterns, the perceptron rule can be written as:

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \Delta\mathbf{w}^{(k)} \quad \text{with} \quad \Delta\mathbf{w}^{(k)} = \eta\left(t^{(k)} - u^{(k)}\right)\mathbf{x}^{(k)}$$

where $t^{(k)}$ and $u^{(k)} \in \{-1, +1\}$, $\mathbf{x}^{(k)} = \left[x_1 \; x_2 \quad x_n \; 1\right]^{(k)}$, $\mathbf{w}^{(k)} = \left[w_1 \; w_2 \quad w_n \; S\right]^{(k)}$ and η is a positive factor, called learning rate, lower than 1. The meaning of this rule is that each pattern is trying to reorient the weight vector either towards itself if the error signal is +1 or opposite to itself if the error signal is -1. If the error signal is 0 then no correction should be given to the weight vector.

If e = +1 then $\mathbf{w}^{(new)} = \mathbf{w}^{(old)} + \mathbf{x}^{(k)}$

If e = -1 then $\mathbf{w}^{(new)} = \mathbf{w}^{(old)} - \mathbf{x}^{(k)}$

If e = 0 then $\mathbf{w}^{(new)} = \mathbf{w}^{(old)}$

In this approach, any pattern contributes to a direct update of the weights. This learning mechanism is called "on-line updating" implying that each pattern error contributes sequentially to the weight updating

(the patterns selection can be random). As any other learning algorithm, there are two main issues to consider: (1) the existence of solutions and (2) convergence of the algorithm to the desired solutions (if they exist). In the case of the classifier perceptron, it is clear that a solution vector (i.e., a vector w* which correctly classifies the training set) exists if and only if the given training set is linearly separable. Designing an appropriate perceptron, to correctly classify the training set, amounts to determine the weight vector **w*** such that the following relations are satisfied:

$$\begin{cases} \mathbf{x}^{T(k)}\mathbf{w}^* > 0 \ \text{ if } t^{(k)} = +1 \\ \mathbf{x}^{T(k)}\mathbf{w}^* < 0 \ \text{ if } t^{(k)} = -1 \end{cases}$$

As earlier stated, the perceptron learning rule corrects the weight vector if and only if a misclassification, indicated by

$$\mathbf{z}^{T(k)}\mathbf{w}^{(k)} \le 0 \ \text{ with } \ \mathbf{z}^{(k)} = \begin{cases} +\mathbf{x}^{(k)} \ \text{ if } t^{(k)} = +1 \\ -\mathbf{x}^{(k)} \ \text{ if } t^{(k)} = -1 \end{cases}, \text{ occurs.}$$

This rule geometrically can be thought of as a way to orient the decision boundary in such a way that the scalar product of weight vector with all the patterns into the first class is positive whereas it is negative with all the patterns into the second class.

Let us try to verify the rule from the following example. Let us consider the following simple Perceptron, encompassing two input lines and one single output neuron, aiming at classify the input in two classes (Fig. 10). In order to ensure generalization over the activation function we here will consider the "step" function. For this choice, the above equation will modify in:

$$\begin{cases} \mathbf{x}^{T(k)}\mathbf{w}^* > 0 \text{ if } t^{(k)} = 1 \\ \mathbf{x}^{T(k)}\mathbf{w}^* < 0 \text{ if } t^{(k)} = 0 \end{cases}$$

and the rule will change in:

$$\mathbf{z}^{T(k)}\mathbf{w}^{(k)} \le 0 \text{ with } \mathbf{z}^{(k)} = \begin{cases} +\mathbf{x}^{(k)} \text{ if } t^{(k)} = 1 \\ -\mathbf{x}^{(k)} \text{ if } t^{(k)} = 0 \end{cases}$$

As we claimed earlier, we are assuming that the neuron threshold T can be thought of as a weight for a virtual steady signal (-1) entering the neuron. Three input patterns are given in the training dataset along with the corresponding nominal output set $\{t_i\}$.

Being $\mathbf{w}_{start}$ the initial randomly selected weight vector, we can obtain the decision boundary in the input plane (x_1,x_2) as the implicit line equation $P = w_1x_1+w_2x_2+w_3(-1) = 0$ ($w_3 \equiv T$) or in the explicit form as:

$x_2 = -w_1x_1/w_2 + w_3/w_2$ with slope $-w_1/w_2$

By construction, featuring a slope of $w_2/w1$, the weight vector $\mathbf{w}=[\ w_1\ w_2]$ is orthogonal to the decision boundary. Now we can apply the perceptron rule using the first pattern $\mathbf{x}^{(1)}=[1\ 2\ -1]$ (setting $\mathbf{w}_0 = \mathbf{w}_{start}$) as (Fig. 11):

$u1 = step(w_1x_1+w_2x_2+w_3(-1)) = 0$
$\mathbf{w}_1 = \mathbf{w}_0 + (t_1 - u_1) * \mathbf{x}^{(1)} = [\ 2.0\ 1.2\ -0.5]$

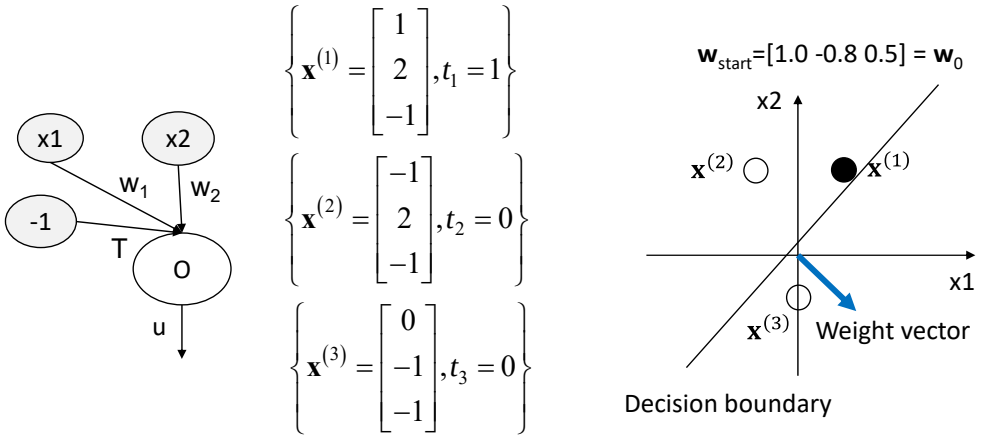


Figure 10. Single neuron perceptron with 2 input lines. Training dataset. Graphic representation of the actual decision boundary.

We can easily notice that the new weight vector is the result of the geometric composition between the old weight vector and the vector $\mathbf{x}^{(1)}$, positive or negative according to the signal error. Iterating the updating step across all the patterns we could expect to reach convergence so that all the input patterns are correctly classified, meaning that the classification is right and any other training step does not sensibly change the weights. This cannot be true in general. It can be easily understood that the classification can be done if and only if the two classes are linearly separated. In order to address this issue, we will extend the perceptron to continuous activation functions first, and then increase the number of layers of the FFNN.

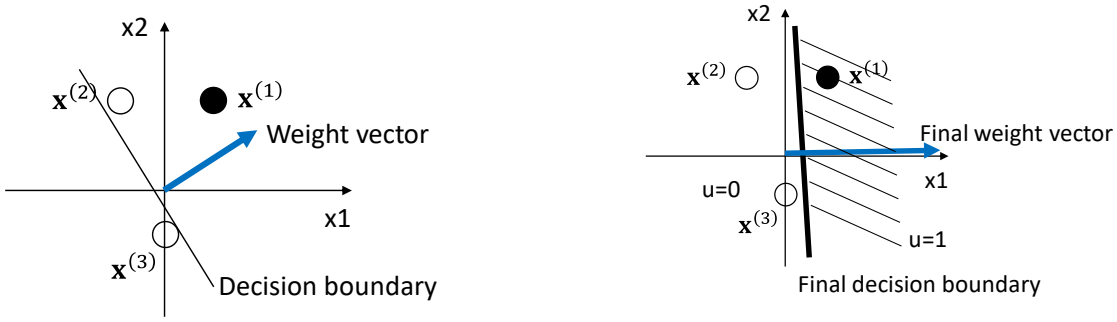


Figure 11. Perceptron rule in action. (Left Panel) Updating the weight vector **w** using the first pattern **p1** allows to changing the orientation of **w** according to **p1** and the corresponding error $e=(t^{(1)}-u^{(1)})$. If the dataset is linearly separable the learning converges to a reliable solution.

PRINCIPLE OF THE DELTA LEARNING RULE (LINEAR AND NON-LINEAR UNITS)

Let consider now the same perceptron of the former example (M input lines, N output neurons, training dataset composed by *R* different patterns) with generic linear/non-linear activation functions like the sigmoid. The error signal for a particular pattern *k* in the training set can be computed as a squared value of difference between the reference value and the neuron output as:

$$E^{(k)} = \sum_{i=1}^{N} \frac{1}{2}\left(t_i^{(k)} - u_i^{(k)}\right)^2$$

where signals $u^{(k)}_i$ and $t^{(k)}_i$ are the output and the reference value of the i^{th} neuron in correspondence of the k^{th} pattern. Assuming R patterns in the training dataset, the overall error signal is the summation of the error signals for each pattern as:

$$E = \sum_{k=1}^{R} E^{(k)} = \sum_{k=1}^{R}\sum_{i=1}^{N} \frac{1}{2}\left(t_i^{(k)} - u_i^{(k)}\right)^2$$

where square makes error positive and penalizes large errors more. Again, we can expect that:

$$\Delta w_{ij} = f\left(E\right)$$

The delta rule for ANN training extended this concept assumes that:

1) the variation Δw_{ij} is to be negative whether E increases with the increase of w_{ij};
2) the variation Δw_{ij} is to be positive whether E increases with the decrease of w_{ij}.

These two updating rules are straightforward when the Gradient descent method is taken into account and can be summarized with the differential form:

$$\Delta w_{ij} \approx \partial E / \partial w_{ij} .$$

In the general formulation, gradient descent is an iterative optimization algorithm that approaches a local minimum of a function by taking steps proportional to the negative of the gradient of the function as the current point. Therefore, it is implemented by calculating the derivative (gradient) of the Cost Function E with respect to the weights, and then change each weight by a small increment in the negative (opposite) direction to the gradient so that:

$$w_{new} = w_{old} + \Delta w$$
$$\Delta w = \eta\left(-\frac{\partial E}{\partial w}\right)$$

where $0 < \eta \leq 1$ is again the learning rate (to be chosen a priori) that now is modulating the amplitude of the gradient vector. The Delta rule for updating the weight of the neuron *i* for the input *j*, given the *r* pattern in the training set, reads as

$$\left(\Delta w_{ij}\right)^{(k)} = \eta\left(t_i^{(k)} - u_i^{(k)}\right) f'\left(P_i^{(k)}\right) x_j^{(k)}$$

where $f'\left(P_i^{(k)}\right)$ is the derivative of the activation function (Table 1) of the neuron *i* computed in correspondence of the actual action potential. One can easily argue that both Heaviside and signum activation functions, being not differentiable in the classical sense, cannot be used into a network undergoing training based on Delta rule. Being iterative, w_{ij}, including the contributes of all the patterns, is updated starting from an initial guess step-by-step as:

$$w_{ij}\left(h+1\right) = w_{ij}\left(h\right) + \Delta w_{ij}\left(h\right)$$

where *h* is the iteration step, until a stop criterion is met (e.g. maximum number of iterations, error threshold, gradient error or even a combination of them).

Linear	Logistic	Hyperbolic tangent
$f\left(P_i\right) = aP_i$ $f'\left(P_i\right) = a$	$f\left(P_i\right) = \frac{1}{1+e^{-P_i}}$ $f'\left(P_i\right) = f\left(P_i\right)\left(1 - f\left(P_i\right)\right)$	$f\left(P_i\right) = \tanh\left(P_i\right) = \frac{e^{P_i} - e^{-P_i}}{e^{P_i} + e^{-P_i}}$ $f'\left(P_i\right) = 1 - \left(\tanh\left(P_i\right)\right)^2$

Table 1. Main activation functions along with derivatives.

DERIVING THE DELTA LEARNING RULE

The derivation of the Delta rule can be carried out by using the chaining of the differential terms. By taking into consideration the contribute of any generic pattern, the differential of the error E with respect the weight w_{ij} reads as:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial\left(\frac{1}{2}\left(t_i - u_i\right)^2\right)}{\partial w_{ij}}$$

that can be chained by ∂u_i leading to:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial\left(\frac{1}{2}\left(t_i - u_i\right)^2\right)}{\partial w_{ij}} = \frac{\partial\left(\frac{1}{2}\left(t_i - u_i\right)^2\right)}{\partial u_i}\frac{\partial u_i}{\partial w_{ij}} =$$

The first differential term is equal to $-\left(t_i - u_i\right)$ whereas the second term can be further chained by ∂P_i as to attain $-\left(t_i - u_i\right)\frac{\partial u_i}{\partial P_i}\frac{\partial P_i}{\partial w_{ij}}$. This can be simplified in $-\left(t_i - u_i\right) f'\left(P_i\right)\frac{\partial P_i}{\partial w_{ij}}$ and, as $P_j = \sum_{j=1}^{M} w_{ij} x_j$, further in $-\left(t_i - u_i\right) f'\left(P_i\right)\frac{\partial\left(\sum_{j=1}^{M} w_{ij} x_j\right)}{\partial w_{ij}}$ with $\frac{\partial\left(\sum_{j=1}^{M} w_{ij} x_j\right)}{\partial w_{ij}} = x_j$

As a final point, $\frac{\partial E}{\partial w_{ij}} = \frac{\partial\left(\frac{1}{2}\left(t_i - u_i\right)^2\right)}{\partial w_{ij}} = -\left(t_i - u_i\right) f'\left(P_i\right) x_j$ meaning that the gradient-based increment of the w_{ij}, weight of the neuron i corresponding to the input j, depends on the difference between the target value and the predict value of the neuron i, the derivative of the activation function of the neuron *i* in correspondence of the action potential, the input *j*.

NOTES ON DELTA RULE

Local minima

As the Delta rule is gradient-based, the starting guess (initial values of the weights) is critical and the method intrinsically suffers from local minima. In general, different weight vector solutions may exist able to solve the network task, despite of providing such vectors technically suboptimal solutions.

Dependence on the activation function

The rule is dependent on the derivative activation function so that the selection of the activation function (differentiable) can change the property of the convergence. Sigmoidal activation functions, which are C_0 and C_1, are usually used. Nonetheless, one should keep the initial network weights small enough so that the sigmoids are not saturated during convergence. The derivative of sigmoidal function tends to zero as it when action potential in the saturation region of the sigmoid. This means that the weight updates are very small or even null and the learning can no longer take place. In such a case, the delta rule cannot correct by itself. There are two heuristic countermeasures namely the target off-set and sigmoid prime off-set. The first strategy uses targets of 0.1 and 0.9 instead of 0 and 1. In this case the sigmoid will no longer saturate and the learning will no longer get stuck. Off-setting the targets also has the effect of stopping the network

weights growing too large. In the second strategy, a small off-set (e.g. 0.1) is added to the sigmoid derivative so that it is no longer zero when the sigmoid saturates.

Weight updating

Two main strategies exists for weight update: "online" and "batch". In the online strategy, the weight update, computed for one pattern *k* as:

$$w_{ij} = w_{ij} + \left(\Delta w_{ij}\right)^{(k)} \text{ with } \left(\Delta w_{ij}\right)^{(k)} = \eta\left(t_i^{(k)} - u_i^{(k)}\right) f'\left(P_i^{(k)}\right) x_j^{(k)},$$

determines the re-computation of all the network output signals using this new weight. The weight update for next pattern *k*+1 therefore reads as:

$$w_{ij} = w_{ij} + \left(\Delta w_{ij}\right)^{(k+1)} \text{ with } \left(\Delta w_{ij}\right)^{(k+1)} = \eta\left(t_i^{(k+1)} - u_i^{(k+1)}\right) f'\left(P_i^{(k+1)}\right) x_j^{(k+1)}$$

where $u_i^{(k+1)}$ depends on w_{ij} which has been updated using $\left(\Delta w_{ij}\right)^{(k)}$. Differently from the on-line updating strategy, the batch updating strategy involves the accumulation of all the pattern errors before computing Δw_{ij}. The update rule changes therefore in:

$$\Delta w_{ij} = \eta \sum_{k=1}^{R} \left(t_i^{(k)} - u_i^{(k)}\right) f'\left(P_i^{(k)}\right) x_j^{(k)}$$

In the batch mode, the batch size can be altered from the standard size *R,* arbitrarily choosing a number of patterns *S* even with *S*<<*R*. In such a case, in one iteration there will n=*R*/*S* + 1 steps with the last step featuring R-n*S patterns.

One iteration

$$\Delta w_{ij}^{(1)} = \eta \sum_{k=1}^{S} \left(t_i^{(k)} - u_i^{(k)}\right) f'\left(P_i^{(k)}\right) x_j^{(k)} \qquad w_{ij} = w_{ij} + \Delta w_{ij}^{(1)}$$

$$\Delta w_{ij}^{(2)} = \eta \sum_{k=S+1}^{2*S} \left(t_i^{(k)} - u_i^{(k)}\right) f'\left(P_i^{(k)}\right) x_j^{(k)} \qquad w_{ij} = w_{ij} + \Delta w_{ij}^{(2)}$$

…

$$\Delta w_{ij}^{(n)} = \eta \sum_{k=(n-1)S+1}^{n*S} \left(t_i^{(k)} - u_i^{(k)}\right) f'\left(P_i^{(k)}\right) x_j^{(k)} \quad w_{ij} = w_{ij} + \Delta w_{ij}^{(n)}$$

$$\Delta w_{ij}^{(n+1)} = \eta \sum_{k=n*S+1}^{R-n*S} \left(t_i^{(k)} - u_i^{(k)}\right) f'\left(P_i^{(k)}\right) x_j^{(k)} \quad w_{ij} = w_{ij} + \Delta w_{ij}^{(n+1)}$$

Learning rate

The selection of learning rate η can be critical as well, affecting both the quality and the speed of the convergence. Too small values guarantee resolution in the detection of the minimum but makes very slow the estimation. The gradient descent weight changes depending on the gradient of the error function. Consequently, if the error function has flat spots, the learning algorithm can take a long time to pass through them. Higher value can help the estimation to converge in such flat spots of the error function but conversely can prevent convergence in sharp regions of the error function. On-line learning does not perform true gradient descent, and the individual weight changes can be rather erratic. Normally a much

lower learning rate η will be necessary than for batch learning. However, overall the learning is often much quicker. This is particularly true if there is a lot of redundancy in the training data, i.e. many training patterns containing similar information.

Cost function

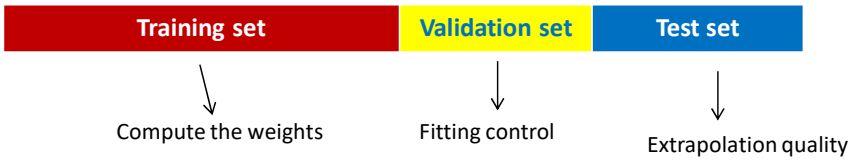
We have seen the error function $E^{(k)} = \sum_{i=1}^{N} \frac{1}{2}\left(t_i^{(k)} - u_i^{(k)}\right)^2$ (k is the index of the k^{th} pattern in the training set) is minimized in the Delta rule approach. However, we do not have any control on the amplitude of the estimated weights. In general, an undue weight increase could correspond to instability of the optimization even preventing converge to the minimum. In order to overcome this issue, the optimization can exploit regularization. This approach rests on a modification of the error function by introducing a regularization factor that account for weight amplitude. Typically the term $||w||^2$ (the squared norm of the weights), composed with the output network error allows keeping weights small as much as possible. The error function modifies therefore in:

$$f = \eta \sum_{i=1}^{N} \frac{1}{2}\left(t_i^{(k)} - u_i^{(k)}\right)^2 + \beta \|w\|^2$$

where β is called regularization rate (heuristically chosen), which modulate the effect of the regularization factor.

Training data

According to the traditional approach in machine learning, the dataset used for network training can be split in three subsets namely training, validation and test. According to the next picture, the training set is used for weight updating, the validation set is used to online control fitting quality during the training, whereas the test set is used to evaluate the extrapolation accuracy of the trained network. As small residual error on the training set does not guarantee good generalization properties, the validation set can be exploited to stop the training when the error on that set overcomes a predefined threshold, this reducing the effect of overfitting.



Some additional heuristic rules need to be discussed. First, training data should be representative for the target task avoiding too many examples of one type at the expense of another type. Second, if one class of pattern is easy to learn, having large numbers of patterns from that class in the training set will only slow down the over-all learning process. Last, for real-value inputs, rescaling the range using zero mean and std (data normalization) is a suitable way to reduce the risk of sigmoid saturation preventing minimization stuck, premature convergence or undue increase of the estimated weights.

Concluding remarks

The main drawback of the Delta rule is that it cannot be used to train multilayer networks, which are mandatory to create non-linear decision boundaries. Actually, the delta rule requires the nominal output of all the neurons but this is not possible for the output of the neurons in the hidden layers of the multi-layer FFNN. We will see in the next how to overcome this issue by the extension of the Delta rule to the backpropagation of the error.