Contents

1. Neural Networks

1.1. Biological foundations

Neurons are first and foremost studied by neurobiology and neurophyisiology. The interest of artificial intelligence is to mimic the way biological neurons work, so that the same model can be applied to non-living beings. In particular, the interest is to study the way living beings collect information through senses, the way they process this collected information and the way they learn from experience.

Neurons have a a core in the form of the nucleus that receives information from other neurons collected information. When the nucleus receives a sufficient aumount of stimulation, it releases back information on nearby neurons. The connection between the simulated neuron and the stimulating one is called **synapsis**; an excited neuron induces the synapsis to release chemicals called **neurotransmitters**, received from the **dendrites** of the receiving neuron.

If a neuron receives enough stimulation from its dendrites, it decides to send in turn a signal to other neurons through an electric signal. The **axon** propagate the electric stimulus from the dendrites to the nucleus. When a neuron sends an electric signal, we say that the neuron *fired*.

A real computer cannot, as is, completely capture the complexity of a real brain.

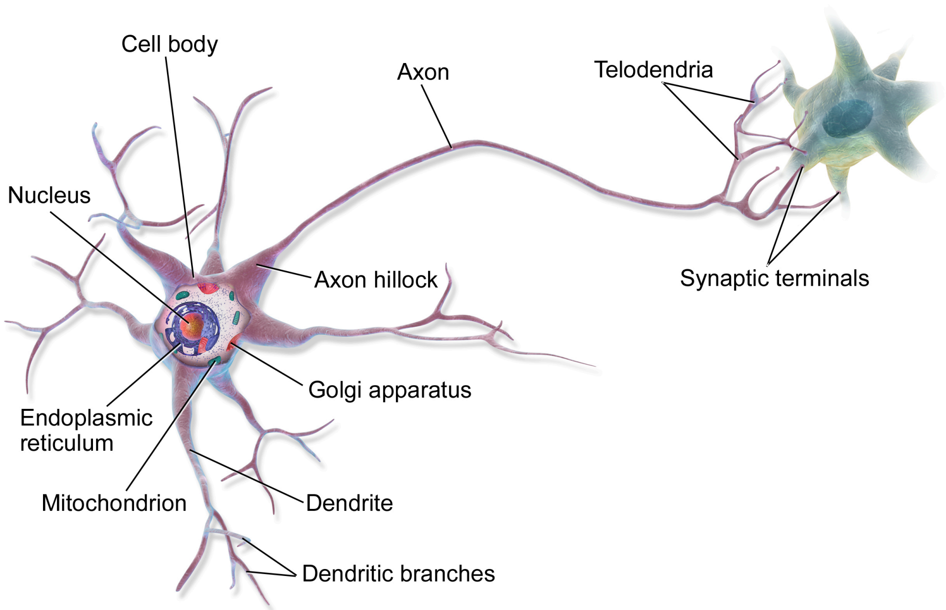


Figure 1: Schematic image of a neuron. By BruceBlaus, CC BY 3.0, via Wikimedia Commons. original image

Advantages of neural networks:

- High parallelism, which entails speedup;
- Fault tolerance, even if a large part of the network is failing the overall network might still work (not always, but close to);
- If some neurons get degraded, we slowly loose our capabilities, but never abruptly. Failing nodes can be phased slowly.

In first approximation, any living being has an input facility (smell, touch, taste), which deliver information to a neuron pool connected to an output. The idea is to have a model that approximates this structure but without the "living being" part.

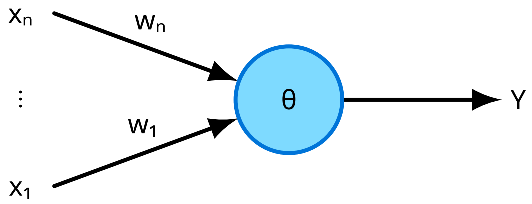
1.2. Threshold Logic Units

A **Threshold Logic Unit** (TLU), also known as **perceptron**[1] or **McCulloch-Pitts neuron** is a mathematical structure that models, in a very simplified way, how neurons operate.

[1]The original definition of perceptron was more refined than a TLU, but the two terms are often used interchangeably.

A TLU has n binary inputs $x_1, x_2, ..., x_n$, each weighted by a weight $w_1, w_2, ..., w_n$, that generates a single binary output y. If the sum of all the inputs multiplied by their weights is a value greater or equal than a given threshold θ, the output y is equal to 1, otherwise is equal to 0.

The analogy between TLUs and biological neurons is straightforward. The output of a TLU is analogous to the firing of a neuron: an output equal to 1 corresponds to the firing of a neuron, whereas an output equal to 0 corresponds to a neuron insufficiently stimulated to be firing.



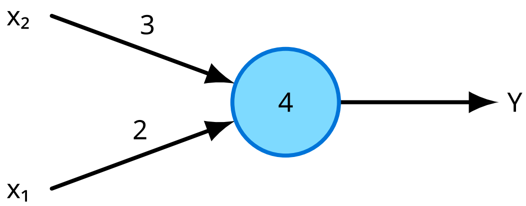
$$y = \begin{cases} 1 \text{ if} \sum_{i=1}^{n} w_i x_i \geq \theta \\ 0 \text{ otherwise} \end{cases}$$

Figure 2: A common way of representing a TLU. The processing unit is drawn as a circle, with the threshold in its center. Inputs are drawn as arrows entering the TLU from the left, with their respective weights above. The output is an arrow exiting the TLU from the right.

The inputs can be collected into a single input vector $\boldsymbol{x} = (x_1, ..., x_n)$ and the weights into a weight vector $\boldsymbol{w} = (w_1, ..., w_n)$. With this formalism, the output y is equal to 1 if $\langle \boldsymbol{w}, \boldsymbol{x} \rangle \geq \theta$, where $\langle \rangle$ denotes the scalar product.

Exercise 1.2.1: Construct a TLU with two inputs whose threshold is 4 and whose weights are $w_1 = 3$ and $w_2 = 2$.

Solution:



x_1	x_2	$3x_1 + 2x_2$	y
0	0	0	0
1	0	3	0
0	1	2	0
1	1	5	1

□

Intuitively, a negative weight corresponds to an inhibitory synapse: if the corresponding input becomes active (that is, equal to 1), it gives a negative contribution to the overall excitation. On the other hand, a positive weight corresponds to an excitatory synapse: if the corresponding input becomes active (that is, equal to 1), it gives a positive contribution to the overall excitation.

Note how the weighted summation that discriminates whether the output of a TLU is 1 or 0 is very similar to an n-dimensional linear function. That is, by substituting the $\geq$ sign with a $=$ sign, it effectively turns into an n dimensional straight line:

$$\sum_{i=1}^{n} w_i x_i = \theta \Rightarrow \sum_{i=1}^{n} w_i x_i - \theta = 0 \Rightarrow w_1 x_1 + w_2 x_2 + ... + w_n x_n - \theta = 0$$

As a matter of fact, the line $\sum_{i=1}^{n} w_i x_i - \theta = 0$ acts as a **decision border**, partitioning the n-dimensional Euclidean hyperplane into two half-planes: one containing n-dimensional points that give an output of 1 when fed the TLU and the other containing points that give an output of 0.

To deduce which of the two regions of space is which, it suffices to inspect the coefficients of the line equation. Indeed, the coefficients $x_1, ..., x_n$ are the elements of a normal vector of the line: the half-plane that contains points that give the TLU an output of 1 is the one to which this vector points to.

Unfortunately, not all linear functions can be represented by a TLU. More formally, two sets of points are called **linearly separable** if there exists a linear function, called **decision function**, that partitions the Euclidean hyperplane into two half-planes, each containing one of the two sets.

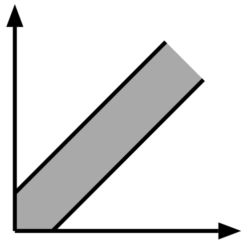
A set of points in the plane is called **convex** if connecting each point of the set with straight lines does not require to go outside of the set. The **convex hull** of a set of points is its the smallest superset that is convex. If two sets of points are both convex and disjoint, they are linearly separable.

A TLU is capable of representing only functions such as these, but for two sets of points a decision function might not exist, and therefore not all sets of points are linearly separable.

Exercise 1.2.2: Is the function $A \leftrightarrow B$ linearly separable?

Solution: No, and it can be proven.

x_1	x_2	y
0	0	1
1	0	0
0	1	0
1	1	1

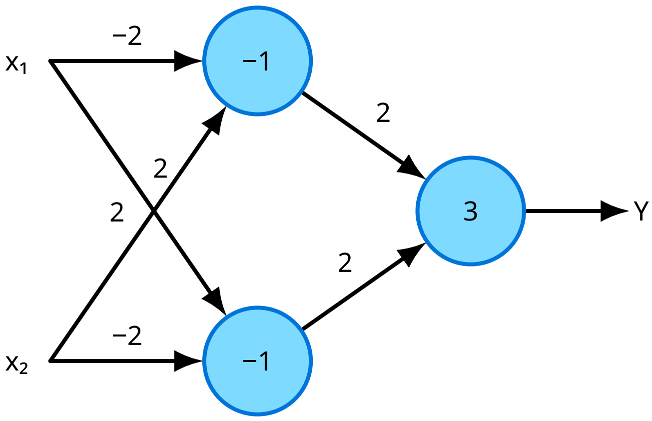


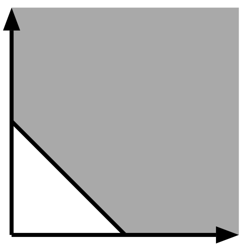
□

Even though single TLPs are fairly limited, it is possible to chain more TLPs together, creating a *network* of threshold logic units. This can be done by breaking down a complicated boolean function into approachable functions, each representable by a TLU, and using the outputs of TLUs as inputs of other TLUs. Since both the inputs and the outputs of a TLP are binary values, this doesn't pose a problem. By applying a coordinate transformation, moving from the original domain to the image domain, the set of points become linearly separable.

Exercise 1.2.3: Is it possible to construct a network of threshold logic units that can represent $A \leftrightarrow B$?

Solution: Yes. Note how $A \leftrightarrow B$ can be rewritten as $(A \rightarrow B) \wedge (B \rightarrow A)$. Each of the three functions (two single implications and one logical conjunction) is linearly separable.





□

It can be shown that all Boolean functions with an arbitrary number of inputs can be computed by networks of TLUs, since any Boolean function can be rearranged in the disjunctive normal form (or conjunctive normal form). A Boolean function in disjunctive normal form is only constituted by a streak of `or` each constituted by `and` (potentially negated), which are all linearly separable.

In particular, a TLU network of two layers will suffice: let $y = f(x_1, ..., x_n)$ be a Boolean function of n variables. It is possible to construct a network of threshold logic units that represents y applying this algorithm:

1. Rewrite the function y in disjunctive normal form:

$$D_f = K_1 \vee ... \vee K_m = (l_{1,1} \wedge ... \wedge l_{1,n}) \vee ... \vee (l_{m,1} \wedge ... \wedge l_{m,n}) = \bigvee_{j=1}^{m} \left(\bigwedge_{i=1}^{n} l_{j,i} \right)$$

   Where each $l_{j,i}$ can be either non-negated (positive literal) or negated (negative literal)
2. For each K_j construct a TLU having n inputs (one input for each variable) and the following weights and threshold:

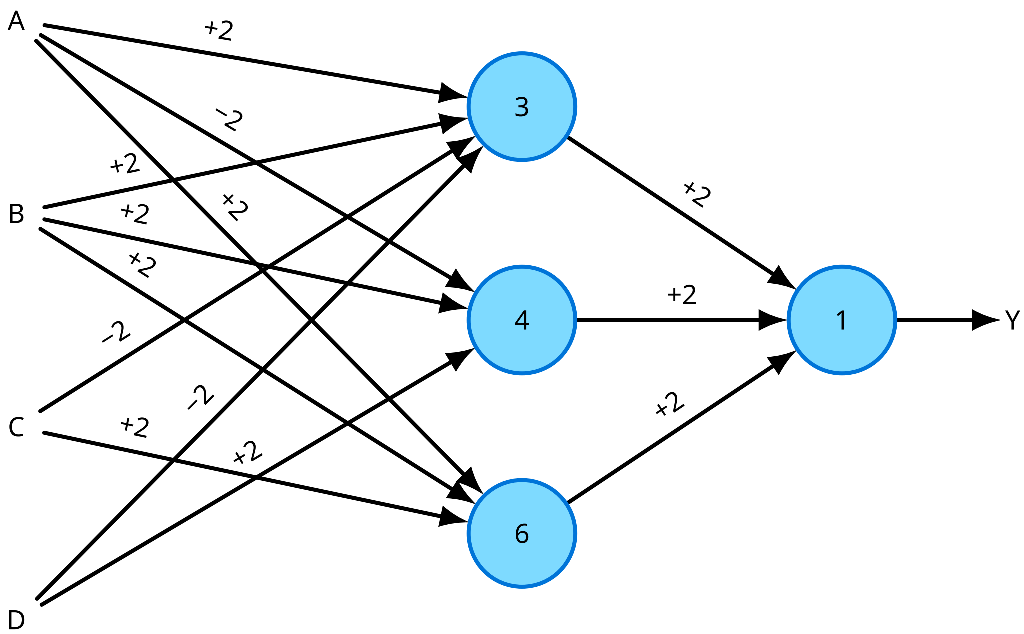
$$w_{j,i} = \begin{cases} +2 \text{ if } l_{j,i} \text{ is a positive literal} \\ -2 \text{ if } l_{j,i} \text{ is a negative literal} \end{cases} \qquad \theta_j = n - 1 + \frac{1}{2} \sum_{i=1}^{n} w_{j,i}$$

3. Create one output neuron, having m inputs (equal to the number of TLUs created in the previous steps), threshold equal to 1 and all weights equal to 2.

Exercise 1.2.4: Construct a TLU network for the boolean function:

$$F(A, B, C, D) = (A \wedge B \wedge C) \vee (\overline{A} \wedge B \wedge D) \vee (A \wedge B \wedge \overline{C} \wedge \overline{D})$$

Solution:



□

1.3. Training TLUs

The aforementioned method for constructing a TLU consists in finding an n-dimensional hyperplane that separates a convex set into two subsets, one containing values for which the TLU outputs 1 and one containing values for which the TLU outputs 0. However, this method is feasible only if the dimension of the sets is small.

First of all, if the dimension of the sets is greater than 3, it's impossible to give it a visual representation. Secondly, this method requires a "visual inspection" of the set to identify the chosen line/plane, meaning that it is hardly possible to encode the process into an algorithm to be fed to a computer, and has to be carried out "by hand" instead. Finally, even if the number of dimensions is small, finding a linear separation for a set can still be tedious.

To construct an automated process that is capable of generate a TLU given a boolean function, a different approach is needed. The idea is to start with randomly generated values for the weights and the threshold of the TLU, trying out the TLU with input data to see if its outputs match the expected outputs, tuning the TLU parameters in accord if this isn't the case and repeating the process until the output of the TLU matches the output of the function. This process of stepwise tuning of the TLU is also referred to as the **training** of the TLU.

To achieve the goal of training a TLU, it is first necessary to quantify "how much" the outputs of the TLU and the outputs of the function to encode differ. This quantification is given by an *error function* $e(w_1, ..., w_n, \theta)$, that taken in input the n weigths $w_1, ..., w_n$ of a TLU and the threshold θ and returns as output a weighted difference between the outputs of the TLU and the outputs of the function. Clearly, when the output of the error function is 0, the original function and the encoded function of the TLU match perfectly. The goal is therefore to reduce the output of the function at any training step of the TLU until it becomes 0.

The most natural way to construct an error function would be to take the absolute value of the difference between the outputs of the function and the outputs of the TLU and summing them up. However, this approach would not be feasible, because it would create a stepwise error function, meaning that, again, only visual inspection would be able to determine how to tune the weights and the threshold of the TLU so that the outputs match. This is due to the fact that stepwise functions are not minimizable, since they are not differentiable everywhere. One could try at random possible combinations of inputs and weights until one is found that zeros the error function, but in general this is not a feasible approach.

A better way to define such a function is to consider instead "how far" the threshold of the TLU is exceeded for each input. This way, it becomes possible to read "locally" where to follow along the shape of the error function by moving, at each step, in the direction of greatest descent, that is, with the direction of the highest slope, even when the overall shape of the function is unknown.

There are two formulations of the training process. The first consists in tuning the TLU with respect to the first input, then tuning the TLU with respect to the second input, and so on until a traning process is undergone for all inputs, then repeating from the first input if necessary: this is referred to as **online training**. The second consists in accumulating all the tunings for each input and applying them all at once at the end of a training cycle: this is referred to as **batch training** and each training cycle is also referred to as an **epoch**.

It is now possible to explicitly formulate an algorithm for the training process of the TLU. First, one should start from this observation: if the output of the TLU is 1 whereas the output of the function is 0, it must mean that the threshold of the TLU is too low and/or the weights of the TLU are too high. Therefore, if this happens, one should raise the threshold and lower the weights. On the other hand, if the output of the TLU is 0 whereas the output of the function is 1, it must mean that the threshold of the TLU is too high and/or the weights of the TLU are too low, and those should be tuned accordingly.

A single training step can be formulated as follows. Let $\boldsymbol{x} = (x_1, ..., x_n)$ be an input vector of a TLU, y the output of the function with $\boldsymbol{x}$ as input and $\hat{y}$ the output of the TLU with $\boldsymbol{x}$ as input. If $\hat{y} \neq y$, then the threshold θ and the weights $\boldsymbol{w} = (w_1, ..., w_n)$ of the TLU can be updated in accord to the following rule, called **delta rule**, or **Widrow-Hoff rule**:

$$\begin{cases} \theta & \leftarrow \theta - \eta(y - \hat{y}) \\ w_i & \leftarrow w_i + \eta(y - \hat{y})x_i, \forall i \in \{1, ..., n\} \end{cases}$$

The parameter η is called **learning rate**, and determines how much the threshold and weights are changed: at every step, they are increased or reduced by a factor of η. It shouldn't be set either too low, because the updates would be very slow, but should be too high either, because the new value of the parameters might jump to another slope of the error function.

The delta rule allows one to write out an algorithm for the training of TLU, both following the batch training paradigm and the online training paradigm. Let $L = ((X_1, y_1), ..., (X_m, y_m))$ be a set of examples used to train the TLU; each example is constituted by an array of binary inputs $X_j = (x_{1,j}, ..., x_{m,j})$ and a binary output y_j. Let $W = (w_1, ..., w_n)$ be a set of randomly chosen initial weights and let θ be a randomly chosen initial threshold. The two algorithms are presented as follows:

```
TLU-TRAIN-ONLINE(W = (w_1, ..., w_n), L = ((X_1, y_1), ..., (X_m, y_m)), θ, η):
let e ← ∞                                    // Error
while (e ≠ 0)                                // Continue until error vanishes
    e ← 0
    for l_i in L do
        let X, y ← l_{i,1}, l_{i,2}          // Unpack
        let ŷ ← 0                            // Evaluate scalar product
        if (Σ_{j=1}^{|X|} X_j · W_j ≥ θ)
            ŷ ← 1
        if (ŷ ≠ y)                           // Test for output mismatch
            e ← e + |y − ŷ|                  // Update error
            θ ← θ − η · (y − ŷ)              // Update threshold
            for w_j in W do
                w_j ← w_j + η · (y − ŷ) · X_j    // Update weights
```

```
TLU-TRAIN-BATCH(W = (w_1, ..., w_n), L = ((X_1, y_1), ..., (X_m, y_m)), θ, η):
let e ← ∞                                    // Error
while (e ≠ 0)                                // Continue until error vanishes
    e ← 0
    let θ* ← 0                               // Partial threshold
    let W* ← (0, ..., 0)                     // Partial weights
    for l_i in L do
        let X, y ← l_{i,1}, l_{i,2}          // Unpack
        let ŷ ← 0                            // Evaluate scalar product
        if (Σ_{j=1}^{|X|} X_j · W_j ≥ θ)
            ŷ ← 1
        if (ŷ ≠ y)                           // Test for output mismatch
            e ← e + |y − ŷ|                  // Update error
            θ* ← θ* − η · (y − ŷ)            // Partially update threshold
            for w_j in W do
                w*_j ← w*_j + η · (y − ŷ) · X_j  // Partially update weights
        θ ← θ + θ*                           // Update threshold
        W ← W + W*                           // Update weights
```

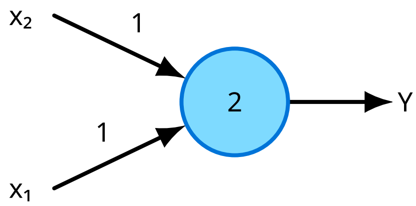
Exercise 1.3.1: Construct a TLU that computes the logical AND between two bits.

Solution:

Let $L = (((0,0),1),((0,1),0),((1,0),0),((1,1),1))$, $W = (0,0)$, $\theta = 0$ and $\eta = 1$. The tables on the left and on the middle denote the training of the TLU, employing online learning and batch learning respectively. On the right, the graphical representation of the TLU obtained from batch learning.

Trial	Weights	θ	Error
0	$(0,0)$	0	∞
1	$(1,1)$	0	2
2	$(2,1)$	1	3
3	$(2,1)$	2	3
4	$(2,2)$	2	2
5	$(2,1)$	3	1
6	$(2,1)$	3	0

Trial	Weights	θ	Error
0	$(0,0)$	0	∞
1	$(-1,-1)$	3	3
2	$(0,0)$	2	1
3	$(1,1)$	1	1
4	$(0,0)$	3	2
5	$(1,1)$	2	1
6	$(1,1)$	2	0



□

The natural question to ask is whether the training process of a TLU always works, that is, if the function encoded in the TLU *converges* to the actual function. Clearly, if the function to be encoded is not lineraly separable, the training process will never converge, since the error function will keep oscillating and never going to 0. However, if the function is linearly separable, the training process does always converge.

Theorem 1.3.1 (Convergence Theorem for the Delta Rule): Let $L = ((X_1, y_1), ..., (X_m, y_m))$ be a set of training examples; each example is constituted by an array of binary inputs and a binary output y_j. Let:

$$L_0 = \{(X, y) \in L \mid y = 0\} \qquad L_1 = \{(X, y) \in L \mid y = 1\}$$

The subsets of L containing all the training examples having output equal to 0 and to 1 respectively. If both L_0 and L_1 are linearly separable, meaning that there exist a vector of weights $W = (w_1, ..., w_n) \in \mathbb{R}^n$ and a threshold $\theta \in \mathbb{R}$ such that:

$$\sum_{j=1}^{n} w_j X_j < \theta,\ \forall (X = (X_1, ..., X_n), 0) \in L_0 \qquad \sum_{j=1}^{n} w_j X_j \geq \theta,\ \forall (X = (X_1, ..., X_n), 1) \in L_1$$

Then, the training process (either batch or online) is guaranteed to terminate.

From this basic formulation, it is possible to look for improvements. First, note how the threshold tuning and the weights tuning are treated separately by the delta rule, since the two updates have opposite signs (negative and positive respectively). However, it is possible to simplify the formula by merging the two expressions into one, turning the threshold into an extra, "special" weight.

To do so, recall that the TLU outputs 1 if $\sum_{i=1}^{n} w_i x_i \geq \theta$ and 0 otherwise. However, this is equivalent to stating that the TLU outputs 1 if $\sum_{i=1}^{n} w_i x_i - \theta \geq 0$ and 0 otherwise. This, in turn, is equivalent to stating that the TLU outputs 1 if $\sum_{i=0}^{n} w_i x_i \geq 0$ and 0 otherwise, where the threshold is now 0 and θ was turned into $w_0 x_0$, a "ficticious" input and a corresponding weight. For the new and old expressions to be equivalent, it suffices to have x_0 always equal to 1 and w_0 equal to $-\theta$ or, equivalently, $x_0 = -1$ and $w_0 = \theta$.

It is now possible to restate the delta rule as follows. Let $\boldsymbol{x} = (x_0 = 1, x_1, ..., x_n)$ be an input vector of a TLU, y the output of the function with $\boldsymbol{x}$ as input and $\hat{y}$ the output of the TLU with $\boldsymbol{x}$ as input. If $\hat{y} \neq y$, then the weights $\boldsymbol{w} = (w_0 = -\theta, w_1, ..., w_n)$ of the TLU can be updated as follows:

$$w_i \leftarrow w_i + \eta(y - \hat{y})x_i, \forall i \in \{0, 1, ..., n\}$$

Once the training process is over, it suffices to turn back w_0 into θ and to remove the input x_0 to obtain the actual formulation of the TLU.

A second improvement deals with the way Boolean functions are encoded. In the original formulation, the value of false is encoded as 0 and the value of true is encoded as 1. The problem of this encoding is that false inputs cannot influence the tuning of the weights, because the sum between weights and zero inputs is zero, slowing the training down. The problem can be circumvented by encoding true as $+1$ and false as -1, so that false inputs also contribute to the training. This is called the **ADALINE model** (**ADAptive LINear Element**).

Having devised a training method for single TLUs, it would be interesting to extend training to networks of TLUs. This would allow one to encode any kind of functions, not just linearly separable functions. Unfortunately, transferring the training process one-to-one from single TLUs to networks of TLUs is not possible. For example, the updates carried out by the delta rule are computed from the difference between the output of the original function and the output of the TLU. However, the tuned output becomes available only to the current TLU, whereas the other TLUs are oblivious to the changes. This means that, to train a network of TLUs, a completely different approach is required.