Chapter 3
Threshold Logic Units

The description of biological neural networks in the preceding chapter makes it natural to model neurons as **threshold logic units**: if a neuron receives enough excitatory input that is not compensated by equally strong inhibitory input, it becomes active and sends a signal to other neurons. Such a model was already examined very early in much detail by McCulloch and Pitts (1943). As a consequence, threshold logic units are also known as **McCulloch–Pitts neurons**. Another name which is commonly used for a threshold logic unit is **perceptron**, even though the processing units that Rosenblatt (1958, 1962) called "perceptrons" are actually somewhat more complex than simple threshold logic units.[1]

3.1 Definition and Examples

Definition 3.1 A **threshold logic unit** is a simple processing unit for real-valued numbers with n inputs $x_1, \ldots, x_n$ and one output y. The unit as a whole possesses a **threshold** θ. To each input x_i a **weight** w_i is assigned. A threshold logic unit computes the function

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

The inputs are often combined into an input vector $\mathbf{x} = (x_1, \ldots, x_n)$ and the weights into a weight vector $\mathbf{w} = (w_1, \ldots, w_n)$. With the help of the scalar product, the condition tested by a threshold logic unit may then also be written as $\mathbf{wx} \geq \theta$.

We depict a threshold logic unit as shown in Fig. 3.1. That is, we draw it as a circle, in which the threshold θ is recorded. Each input is drawn as an arrow that points to the circle and that is labeled with the weight of the input. The output of the threshold logic unit is shown as an arrow that points away from the circle.

[1] In a perceptron there is, besides the actual threshold logic unit, an input layer that executes additional operations on the input signals. However, this input layer consists of immutable functional elements and therefore is often neglected.

R. Kruse et al., *Computational Intelligence*, Texts in Computer Science,
DOI 10.1007/978-1-4471-5013-8_3, © Springer-Verlag London 2013

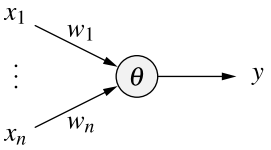
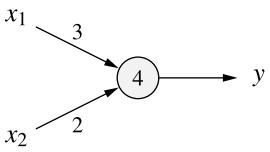


Fig. 3.1 Representation of a threshold logic unit

To illustrate how threshold logic units work and to demonstrate their capabilities, we consider a couple of simple examples. Figure 3.2 shows on the left a threshold logic unit with two inputs x_1 and x_2, which carry the weights $w_1 = 3$ and $w_2 = 2$, respectively. The threshold is $\theta = 4$. If we assume that the input variables can only have values 0 and 1, we obtain the table shown in Fig. 3.2 on the right. Clearly, this threshold logic unit computes the conjunction of its inputs: only if both inputs are active (that is, equal to 1), it becomes active itself and outputs a 1.

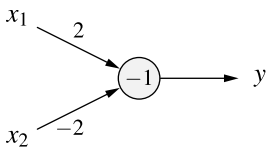
Figure 3.3 shows another threshold logic unit with two inputs, which differs from the one shown in Fig. 3.2 by a negative threshold $\theta = -1$ and one negative weight $w_2 = -2$. Due to the negative threshold, it is active (that is, outputs a 1) even if both inputs are inactive (that is, are equal to 0). Intuitively, the negative weight corresponds to an inhibitory synapse: if the corresponding input becomes active (that is, equal to 1), the threshold logic unit is deactivated and its output becomes 0. We can also observe here that positive weights correspond to excitatory synapses: even if the input x_2 inhibits the threshold logic unit, (that is, if $x_2 = 1$), it can become active, namely if it is "excited" by an active input x_1 (that is, by $x_1 = 1$). In summary, this threshold logic unit computes the function show in the table in Fig. 3.3 on the right, that is, the implication $y = x_2 \rightarrow x_1$.

An example for a threshold logic unit with three inputs is shown in Fig. 3.4 on the left. This threshold logic unit already computes a fairly complex function, namely the logical expression $y = (x_1 \wedge \overline{x_2}) \vee (x_1 \wedge x_3) \vee (\overline{x_2} \wedge x_3)$. The truth table of this function and the computations that are carried out by the threshold logic unit for the



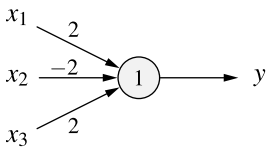
x_1	x_2	$3x_1 + 2x_2$	y
0	0	0	0
1	0	3	0
0	1	2	0
1	1	5	1

Fig. 3.2 A threshold logic unit for the conjunction $x_1 \wedge x_2$



x_1	x_2	$2x_1 - 2x_2$	y
0	0	0	1
1	0	2	1
0	1	-2	0
1	1	0	1

Fig. 3.3 A threshold logic unit for the implication $x_2 \rightarrow x_1$



x_1	x_2	x_3	$\sum_i w_i x_i$	y
0	0	0	0	0
1	0	0	2	1
0	1	0	−2	0
1	1	0	0	0
0	0	1	2	1
1	0	1	4	1
0	1	1	0	0
1	1	1	2	1

Fig. 3.4 A threshold logic unit for $(x_1 \wedge \overline{x_2}) \vee (x_1 \wedge x_3) \vee (\overline{x_2} \wedge x_3)$

different input vectors are shown in Fig. 3.4 on the right. This and the preceding threshold logic unit may lead us to presume that negations (in logical expressions) are (often) represented by negative weights.

3.2 Geometric Interpretation

The condition that is tested by a threshold logic unit in order to decide whether it should output a 0 or a 1 is very similar to the equation of a straight line (cf. Sect. 10.1). Indeed, the computation of a threshold logic unit can easily be interpreted geometrically if we turn this condition into a line, plane or hyperplane equation, that is, if we consider the equation

$$\sum_{i=1}^{n} w_i x_i = \theta \quad \text{or} \quad \sum_{i=1}^{n} w_i x_i - \theta = 0.$$

(Note that the line equation differs from the actual condition in that is uses "=" instead of "≥." This is taken care of below, where the inequality is reinstated.) The resulting geometric interpretation is illustrated in Figs. 3.5, 3.6 and 3.8.

Figure 3.5 repeats on the left the threshold logic unit for the conjunction considered above. In the diagram on the right, the input space of this threshold logic unit is shown. The input vectors, which are listed in the table in Fig. 3.2 on the right, are marked according to the output of the threshold logic unit: a filled circle indicates that the threshold logic unit yields output 1 for this point, while an empty circle indicates that it yields output 0. In addition, the diagram shows the straight line described by the equation $3x_1 + 2x_2 = 4$, which corresponds to the **decision border** of the threshold logic unit. It is easy to verify that the threshold logic unit yields output 1 for all points to the right of this line and output 0 for all points to the left of it, even if we allow for other input values than 0 and 1.

On which side the output of the threshold logic unit is 1 and on which it is 0 can also be read from the line equation: it is well known that the coefficients of x_1 and x_2 are the elements of a normal vector of the line (cf. also Sect. 10.1, which

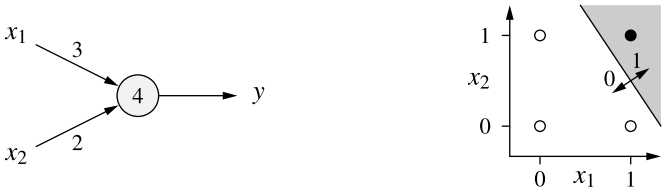


Fig. 3.5 Geometry of the threshold logic unit for $x_1 \wedge x_2$. The *straight line* shown in the right diagram has the equation $3x_1 + 2x_2 = 4$; for the *gray half-plane* it is $3x_1 + 2x_2 \geq 4$

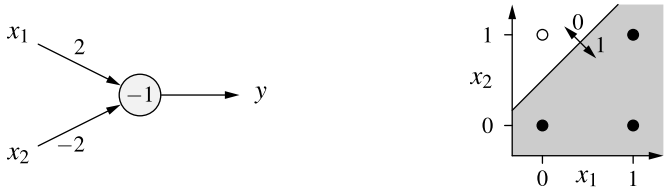


Fig. 3.6 Geometry of the threshold logic unit for $x_2 \rightarrow x_1$: The *straight line* shown in the right diagram has the equation $2x_1 - 2x_2 = -1$; for the *gray half-plane* it is $2x_1 - 2x_2 \geq -1$

collects some important facts about straight lines and their equations.) The side of the line to which this normal vector points if it is attached to a point on the line is the side on which the output is 1. Indeed, the normal vector $\mathbf{n} = (3, 2)$ that can be read from the equation $3x_1 + 2x_2 = 4$ points to the right top and thus to the side on which the point $(1, 1)$ is located. Hence the threshold logic unit yields output 1 in the **half-plane** $3x_1 + 2x_2 \geq 4$, which is shown in gray in Fig. 3.5.

Analogously, Fig. 3.6 shows the threshold logic unit computing the implication $x_2 \rightarrow x_1$ and its inputs space. The straight line drawn into this input space corresponds to its decision border: it separates the points of the input space for which the output is 0 from those for which the output is 1. Since the normal vector $\mathbf{n} = (2, -2)$ that can be read from the equation $2x_1 - 2x_2 = -1$ points to the bottom right, the output is 1 for all points below the line and 0 for all points above it. This coincides with the computations listed in the table in Fig. 3.3, which are represented by filled and empty circles in the diagram in Fig. 3.6. Generally, the threshold logic unit yields output 1 for points in the gray half-plane shown in Fig. 3.6.

Naturally the computations of threshold logic units with more than two inputs can be interpreted geometrically as well. However, due to the limited spatial imagination of humans, we have to confine ourselves to threshold logic units with no more than three inputs. With three inputs, the **separating line** turns into a **separating plane**. We illustrate this by depicting the input space of a threshold logic unit with three inputs as a unit cube as it is shown in Fig. 3.7. With this diagram, let us reconsider the example of a threshold logic unit with three inputs studied in the preceding section, which is repeated in Fig. 3.8. Into the unit cube on the right of this figure the plane with the equation $2x_1 - 2x_2 + 2x_3 = 1$ is drawn in gray, which corresponds to the decision rule of this threshold logic unit. In addition, all input vectors, for which

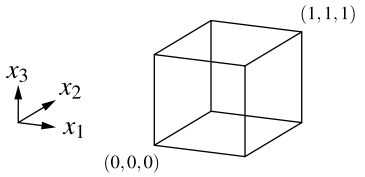


Fig. 3.7 Graphical representation of ternary Boolean functions

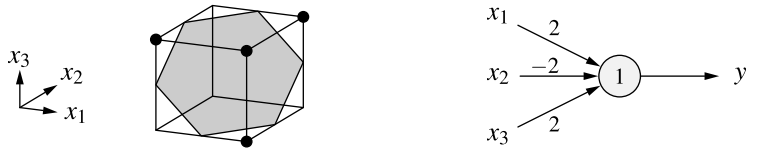


Fig. 3.8 Geometry of the threshold logic unit for the function $(x_1 \wedge \overline{x_2}) \vee (x_1 \wedge x_3) \vee (\overline{x_2} \wedge x_3)$: The plane shown in the left diagram is described by the equation $2x_1 - 2x_2 + 2x_3 = 1$

the table in Fig. 3.4 lists an output of 1, are marked with a filled circle. For all other corners of the unit cube, the output is 0. Like in the two-dimensional case we can again read the side of the plane on which the output is 1 from the normal vector of the plane: from the plane equation we derive the normal vector $\mathbf{n} = (2, -2, 2)$, which points out of the drawing plane to the top right.

3.3 Limitations

The examples studied in the preceding section—especially the threshold logic unit with three inputs—may lead us to presume that threshold logic units are fairly powerful processing units. Unfortunately, though, *single* threshold logic units are severely limited in their expressive and computational power. We know from the geometric interpretation of their computations that threshold logic units can represent only functions that are, as one says, **linearly separable**, that is, functions for which the points with output 1 can be separated from the points with output 0 by a linear function—that is, by a line, plane or hyperplane.

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

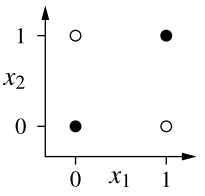


Fig. 3.9 The biimplication problem: there is no separating straight line

Table 3.1 The number of all Boolean functions of n inputs and the number of them that are linearly separable (Widner 1960 cited according to Zell 1996)

inputs	Boolean functions	linearly separable functions
1	$2^{2^1} = 4$	4
2	$2^{2^2} = 16$	14
3	$2^{2^3} = 256$	104
4	$2^{2^4} = 65536$	1774
5	$2^{2^5} \approx 4.3 \cdot 10^9$	94572
6	$2^{2^6} \approx 1.8 \cdot 10^{19}$	$5.0 \cdot 10^6$

Unfortunately, though, not all functions are linearly separable. A very simple example of a function that is not linearly separable is the biimplication (that is, $x_1 \leftrightarrow x_2$), the truth table of which is shown in Fig. 3.9 on the left. From the graphical representation of this function, which is shown in the same figure on the right, we can already see that there is no separating line. As a consequence, there cannot be any threshold logic unit computing this function.

The formal proof is executed by the common method of *reductio ad absurdum* (proof by contradiction). We assume that there exists a threshold logic unit with weights w_1 and w_2 and threshold θ that computes the biimplication. Then it is

$$\begin{aligned}
&\text{due to } (0,0) \mapsto 1: && 0 && \geq \theta, && (1)\\
&\text{due to } (1,0) \mapsto 0: && w_1 && < \theta, && (2)\\
&\text{due to } (0,1) \mapsto 0: && w_2 && < \theta, && (3)\\
&\text{due to } (1,1) \mapsto 1: && w_1 + w_2 && \geq \theta. && (4)
\end{aligned}$$

From (2) and (3) it follows $w_1 + w_2 < 2\theta$, which together with (4) yields $2\theta > \theta$, or $\theta > 0$. However, this contradicts (1). Therefore, there is no threshold logic unit that computes the biimplication.

The fact that only linearly separable functions can be represented may appear to be, at first sight, a small and bearable restriction, since only two of the 16 possible Boolean functions of two variables are *not* linearly separable (namely the biimplication and the exclusive or). However, if the number of inputs is increased, the fraction of all Boolean functions that are linearly separable drops rapidly (see Table 3.1). For a larger number of inputs, (single) threshold logic units can therefore compute "almost no" Boolean functions (relative to all possible ones).

3.4 Networks of Threshold Logic Units

As demonstrated in the preceding section, threshold logic units are severely limited. However, up to now we only considered *single* threshold logic units. The powers of threshold logic units can be increased considerably if we combine several threshold logic units, that is, if we consider networks of threshold logic units.

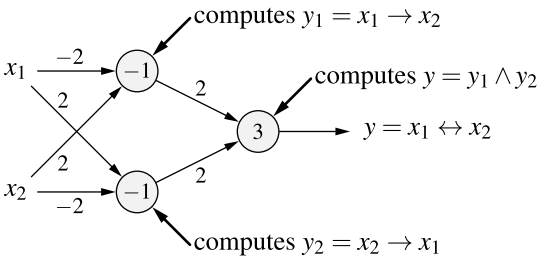


Fig. 3.10 Combining several threshold logic units

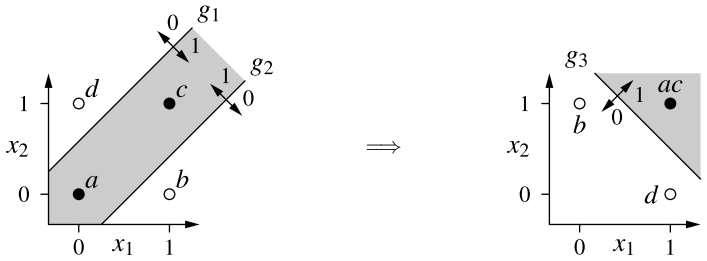


Fig. 3.11 Geometric interpretation of combining multiple threshold logic units into a network to compute the biimplication

As an example, we consider a possible solution for the biimplication problem with the help of three threshold logic units that are organized into two layers. This solution exploits the logical equivalence

$$x_1 \leftrightarrow x_2 \equiv (x_1 \rightarrow x_2) \wedge (x_2 \rightarrow x_1),$$

which divides the biimplication into three functions. From Figs. 3.3 and 3.6, we already know that the implication $x_2 \rightarrow x_1$ is linearly separable. In the implication $x_1 \rightarrow x_2$, the variables are merely exchanged, so it is linearly separable as well. Finally, we know from Figs. 3.2 and 3.5 that the conjunction of two Boolean variables is linearly separable. As a consequence, we only have to connect the corresponding threshold logic units, see Fig. 3.10. Thus we obtain a network with two layers, corresponding to the nested structure of the logical expression.

Intuitively, the two threshold logic units on the left (first layer) compute new Boolean coordinates y_1 and y_2 for the input vectors, so that the transformed input vectors in the input space of the threshold logic unit on the right (second layer) become linearly separable. This is illustrated geometrically with the two diagrams shown in Fig. 3.11. The separating line g_1 corresponds to the upper threshold logic unit and describes the implication $y_1 = x_1 \rightarrow x_2$: for all points above this line the output is 1, for all points below it the output is 0. The separating line g_2 belongs to the lower threshold logic unit and describes the implication $y_2 = x_2 \rightarrow x_1$: for all points above the line the output is 0, for all points below it the output is 1.

The threshold logic units on the left assign the new coordinates $(y_1, y_2) = (0, 1)$ to the input vector $b \mathrel{\hat{=}} (x_1, x_2) = (1, 0)$ and the new coordinates $(y_1, y_2) = (1, 0)$ to the input vector $d \mathrel{\hat{=}} (x_1, x_2) = (0, 1)$, while they assign the coordinates $(y_1, y_2) = (1, 1)$ to both the input vector $a \mathrel{\hat{=}} (x_1, x_2) = (0, 0)$ and the input vector $c \mathrel{\hat{=}} (x_1, x_2) = (1, 1)$ (see Fig. 3.11 on the right). After this transformation, the input vectors for which the output is 1 can easily be separated from those for which the output is 0, for instance, by the line g_3 that is shown in the diagram in Fig. 3.11 on the right.

It can be shown that all Boolean functions with an arbitrary number of inputs can be computed by networks of threshold logic units, simply by exploiting logical equivalences to divide these functions in such a way that all occurring sub-functions are linearly separable. With the help of the disjunctive normal form (or, analogously, the conjunctive normal form), one can even show that the networks only need to have two layers, regardless of the Boolean function to represent:

Algorithm 3.1 (Representation of Boolean Functions)
Let $y = f(x_1, \ldots, x_n)$ be a Boolean function of n variables.

1. Represent the Boolean function $f(x_1, \ldots, x_n)$ in disjunctive normal form. That is, determine $D_f = K_1 \vee \cdots \vee K_m$, where all K_j are conjunctions of n literals, that is, $K_j = l_{j1} \wedge \cdots \wedge l_{jn}$ with $l_{ji} = x_i$ (positive literal) or $l_{ji} = \neg x_i$ (negative literal).
2. For each conjunction K_j of the disjunctive normal form create one neuron (with n inputs—one input for each of the variables) where

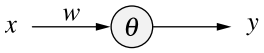
$$w_{ji} = \begin{cases} 2 & \text{if } l_{ji} = x_i, \\ -2 & \text{if } l_{ji} = \neg x_i, \end{cases} \quad \text{and} \quad \theta_j = n - 1 + \frac{1}{2} \sum_{i=1}^{n} w_{ji}.$$

3. Create one output neuron (with m inputs—one input for each of the neurons created in step 2) where

$$w_{(n+1)k} = 2, \quad k = 1, \ldots, m, \quad \text{and} \quad \theta_{n+1} = 1.$$

In the network constructed in this way, every neuron created in step 2 computes a conjunction and the output neuron computes their disjunction.

Intuitively, each neuron in the first layer describes a hyperplane that separates the corner of the hypercube, for which the conjunction is 1, from the rest of the unit hypercube. The equation of this hyperplane is easy to determine: its normal vector points from the center of the unit hypercube to the corner that is cut off and thus is 1 in all components in which the position vector of the corner has value 1, and it is -1 in all components in which the position vector of the corner has the value 0. (As an illustration consider the three-dimensional case.) We multiply this normal vector with 2 in order to obtain an integer threshold. The threshold has to be determined in such a way that it is exceeded only if all inputs that carry a weight of 2 are 1 and all other inputs are 0. The formula stated in step 2 yields such a value.



x	y
0	1
1	0

Fig. 3.12 A threshold logic unit with a single input and training examples for the negation

To compute the disjunction of the outputs of the neurons created in step 2, we have to separate, in an m-dimensional unit hypercube of the conjunctions, the corner $(0, \ldots, 0)$, for which the output is 0, from all other corners, for which the output is 1. This can be achieved, for instance, with the hyperplane that possesses the normal vector $(1, \ldots, 1)$ and the support vector $(\frac{1}{2}, 0, \ldots, 0)$. (As an illustration consider the three-dimensional case again.) The parameters of the output neuron stated in step 3 are then simply read from the corresponding equation.

3.5 Training the Parameters

By interpreting the computations of a threshold logic unit geometrically, as shown in Sect. 3.2, we have (at least for functions with 2 and 3 variables) a simple method to find, for a given linearly separable function, a threshold logic unit that computes it: we determine a line, plane or hyperplane that separates the points for which the output is 1 from those for which the output is 0. From the equation describing this line, plane or hyperplane we can then easily read the weights and the threshold.

However, this methods becomes difficult and finally infeasible if the function to compute has more than three arguments, because we cannot imagine the input space of such a function. Furthermore it is impossible to automate this methods, because we find a suitable separating line or plane by "visual inspection" of the point sets to separate. This "visual inspection" cannot be mimicked directly by a computer. In order to be able to determine the parameters of a threshold logic unit with a computer, so that it computes a given function, we need a different approach. The principle consists in starting with randomly chosen values for the weights and the threshold and then changing these values step by step until the desired function is computed. The slow, stepwise adaptation of the weights and the threshold is also called **learning** or—in order to avoid confusions with the much more complex human learning process—the **training** of the threshold logic units.

In order to find a method to adapt the weightsand the threshold, we start from the following consideration: depending on the values of the weights and the threshold, the computation of the threshold logic unit will be more or less correct. Therefore, we can define an error function $e(w_1, \ldots, w_n, \theta)$, which states how well, for given weights and threshold, the computed function coincides with the desired one. Our objective is, of course, to determine the weights and the threshold in such a way that the error vanishes, that is, that the error function becomes 0. To achieve this, we try to reduce the value of the error function in every step.

We illustrate this procedure with the help of a very simply example, namely a threshold logic unit with only one input. The parameters of this unit are to be

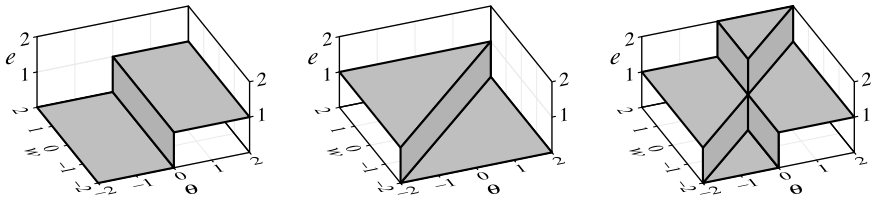


Fig. 3.13 Error of computing the negation w.r.t. the threshold

determined in such a way that it computes the negation. Such a threshold logic unit is shown in Fig. 3.12 together with the two training examples for the negation: if the input is 0, the output should be 1, if the input is 1, the output should be 0.

The error function we define first, as it appears to be natural, as the absolute value of the difference between the desired and the actual output. This function is shown in Fig. 3.13. The left diagram shows the error for the input $x = 0$, for which an output of 1 is desired. Since the threshold logic unit computes a 1 if $xw \geq \theta$, the error is 0 for a negative threshold and 1 for a positive threshold. (Obviously, the weight does not have any influence, because it is multiplied with the input, which is 0.) The middle diagram shows the error for the input $x = 1$, for which an output of 0 is desired. Here both the weight and the threshold have an influence. If the weights are less than the threshold, we have $xw < \theta$ and thus the output and consequently the error is 0. The diagram on the right shows the sum of these individual errors.

From the right diagram, a human can now easily read how the weight and the threshold have to chosen so that the threshold logic unit computes the negation: the values of these parameters must lie in the triangle in the lower left of the w–θ plane, in which the error is 0. However, it is not yet possible to automatically adapt the parameters with this function, because the global "visual inspection" of the error function, which a human relies on to find the solution, cannot be mimicked directly in a computer. Rather we would have to be able to read from the shape of the function at the point, which is given by the current weight and the current threshold, in which directions we have to change the weight and the threshold so that the error is reduced. With this error function, however, this is impossible, because it consists of "plateaus" or "terraces." In "almost all" points (the "edges" of the error function are the only exceptions) the error stays the same in all directions.[2]

In order to circumvent this problem, we modify the error function. Where the threshold logic unit produces the wrong output, we consider how far the threshold is exceeded (for a desired output of 0) or how far it is underrun (for a desired output of 1). Intuitively, we may say that the computation is "the more wrong" the farther the threshold is exceeded for a desired output of 0, or the farther the threshold is underrun for a desired output of 1. The modified error function is shown in Fig. 3.14.

[2]The somewhat imprecise notion "almost all points" can be made mathematically precise by drawing on measure theory: the set of points at which the error function changes has measure 0.

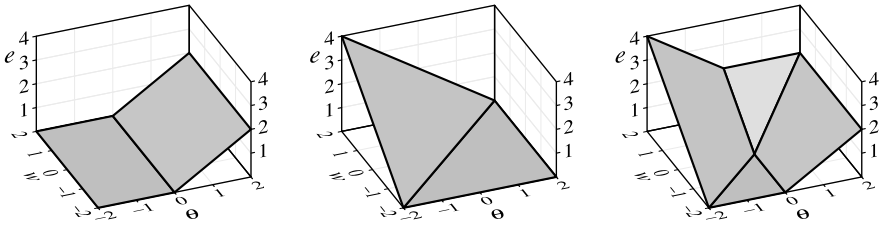


Fig. 3.14 Error of computing the negation w.r.t. how far the threshold is exceeded or underrun

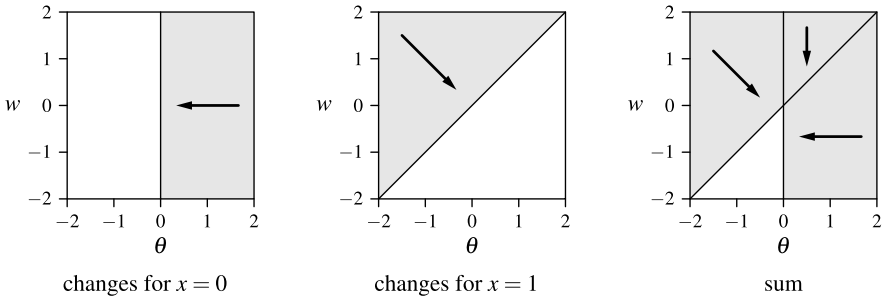


Fig. 3.15 Directions of the weight and threshold changes

Again the left diagram shows the error for the input $x = 0$, the middle diagram the error for the input $x = 1$ and the right diagram the sum of these individual errors.

If a threshold logic unit now produces a wrong output, we adapt the weight and the threshold in such a way that the error is reduced. That is, we try to "descent in the error landscape". With the modified error function this is possible, because we can read from it "locally" (that is, without a visual inspection of the whole error function, but merely by looking at the shape of the error function at the point that is given by the current values of the weight and the threshold) in which directions we have to change the weight and the threshold: we simply move in the direction in which the error function has the strongest downward slope. Intuitively, we follow the common scout advice how to find water: always go downhill. The directions that result from this rule are shown schematically in Fig. 3.15. The arrows indicate how the weight and the threshold should be adapted in different regions of the parameter space. In those regions, in which no arrows are drawn, weight and threshold are left unchanged, because there is no error.

The adaptation rules that are shown in Fig. 3.15 can be applied in two different ways. In the first place, we may consider the inputs $x = 0$ and $x = 1$ alternatingly and may adapt the weight and the threshold according to the corresponding rules. That is, first we adapt the weight and the threshold according to the left diagram, then we adapt them according to the middle diagram, then we adapt them again according to the left diagram and so forth until the error vanishes. This way of

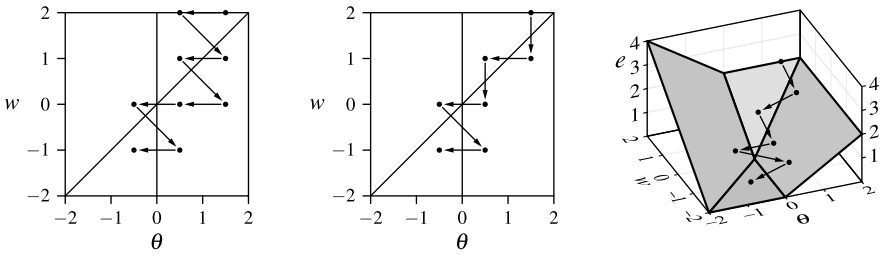


Fig. 3.16 Training processes with initial values $\theta = \frac{3}{2}$, $w = 2$ and learning rate 1

Fig. 3.17 Learned threshold logic unit for the negation and its geometric interpretation

training a neural network is called **online learning** or **online training**, since with every training example that becomes available, a training step can be carried out.

The second option consists in not applying the changes immediately after every training example, but aggregating them over all training examples. Only at the end of a **(learning/training) epoch**, that is, after all training examples have been traversed, the aggregated changes are applied. Then the training examples are traversed again and at the end the weight and the threshold are adapted and so forth until the error vanishes. This way of training is called **batch learning** or **batch training**, since all training examples have to be available together (in a *batch*), and corresponds to adapting weight and threshold according to the diagram on the right.

Figure 3.16 shows the training processes for the initial values $\theta = \frac{3}{2}$ and $w = 2$. Both online training (left) and batch training (middle) use a **learning rate** of 1. The learning rate states by how much the weight and the threshold are changed, and thus how "fast" the training is. (However, the learning rate should also not be chosen arbitrarily large, see Chap. 5.) If the learning rate is 1, the weight and the threshold are increased or reduced by 1. In order to illustrate the "descent in the error landscape", the batch training is repeated in a three-dimensional diagram in Fig. 3.16 on the right. The final, fully trained threshold logic unit (with $\theta = -\frac{1}{2}$ and $w = -1$) is shown, together with its geometric interpretation, in Fig. 3.17.

In this simple example, we derived the adaptation rules directly from a visual inspection of the error function. An alternative way of obtaining these adaptation rules are the following considerations: if a threshold logic unit produces an output of 1 instead of a desired 0, then the threshold is too small and/or the weights are too large. Hence, we should increase the threshold a bit and reduce the weights. Of course, the latter is reasonable only if the corresponding input is 1, as otherwise the weight has no influence on the output. Contrariwise, if a threshold logic unit produces an output of 0 instead of a desired 1, then the threshold is too large and/or

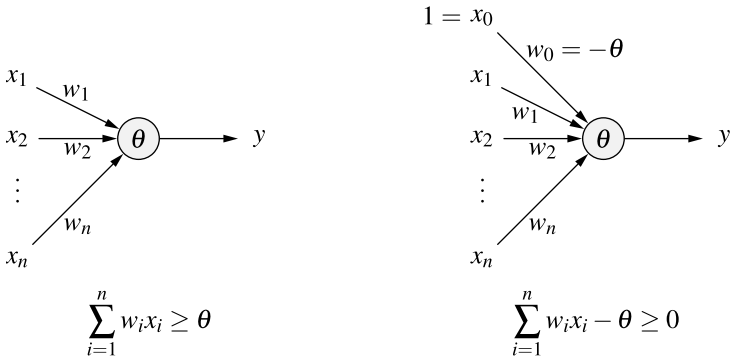


Fig. 3.18 Turning the threshold into a weight

the weights are too small. Hence, the threshold should be reduced and the weights should be increased (provided, of course, that the corresponding input is 1).

For our simple threshold logic unit, the changes shown in Fig. 3.15 have exactly these effects. However, the considerations above have the advantage that they can be applied to threshold logic units with more than one input. Therefore, we can define the following general training method for threshold logic units:

Definition 3.2 Let $\mathbf{x} = (x_1, \ldots, x_n)$ be an input vector of a threshold logic unit, o the desired output for this input vector and y the actual output of the threshold logic unit. If $y \neq o$, then, in order to reduce the error, the threshold θ and the weight vector $\mathbf{w} = (w_1, \ldots, w_n)$ are adapted as follows:

$$\begin{aligned} \theta^{(\text{new})} &= \theta^{(\text{old})} + \Delta\theta \quad \text{with } \Delta\theta = -\eta(o - y), \\ \forall i \in \{1, \ldots, n\} : w_i^{(\text{new})} &= w_i^{(\text{old})} + \Delta w_i \quad \text{with } \Delta w_i = \eta(o - y)x_i, \end{aligned}$$

where η is a parameter that is called **learning rate**. It determines the severity of the weight and threshold changes. This method is called the **delta rule** or **Widrow–Hoff procedure** (Widrow and Hoff 1960).

In this definition, we have to distinguish between an adaptation of the threshold and adaptations of the weights, because the directions of these changes are opposite to each other (opposite signs for $\eta(t - y)$ and $\eta(t - y)x_i$, respectively). However, we can unify the adaptation rules by turning the threshold into a weight. The rationale of this transformation is illustrated in Fig. 3.18: the threshold is fixed to 0 and to compensate this reduction in the number of parameters, an additional (imaginary) input x_0 is introduced, which has a fixed value of 1. This input is weighted with the negated threshold. The two threshold logic units are clearly equivalent, since the left one tests the condition $\sum_{i=1}^{n} w_i x_i \geq \theta$, the right one tests $\sum_{i=1}^{n} w_i x_i - \theta \geq 0$, in order to determine the output. Note that the same effect is produced if the additional input has the fixed value -1 and is weighted with the threshold directly. This alternative way of turning the threshold into a weight is also very common.

Regardless of how the threshold is turned into a weight, we obtain the same adaptation directions for all parameters. With a negated threshold: if the output is 1 instead of 0, then both the w_i as well as $-\theta$ should be reduced. If the output is 0 instead of 1, both the w_i as well as $-\theta$ should be increased. On the other hand, with an unnegated θ and a fixed input of -1, we obtain a uniform rule, because the needed negative sign is produced by the input. Therefore, we can determine the adaptation direction of all parameters by simply subtracting the actual from the desired output.

Thus, we can formulate the delta rule as follows: Let $\mathbf{x} = (x_0 = 1, x_1, \ldots, x_n)$ be an extended input vector of a threshold logic unit (note the additional input $x_0 = 1$), o the desired output for this input vector and y the actual output of the threshold logic unit. If $y \neq o$, then, in order to reduce the error, the extended weight vector $\mathbf{w} = (w_0 = -\theta, w_1, \ldots, w_n)$ (note the added weight $w_0 = -\theta$) is adapted as follows:

$$\forall i \in \{0, 1, \ldots, n\}: \quad w_i^{(\text{new})} = w_i^{(\text{old})} + \Delta w_i \quad \text{with } \Delta w_i = \eta(o - y)x_i.$$

We point out this possibility here, because it can often be used, for example, to simplify derivations (see, for instance, Sect. 5.4). However, for the sake of clarity, we maintain the distinction of weight and threshold for the remainder of this chapter.

With the help of the delta rule, we can now state two algorithms for training a threshold logic unit: an online version and a batch version. In order to formulate these algorithms, we assume that we are given a set $L = \{(\mathbf{x}_1, o_1), \ldots, (\mathbf{x}_m, o_m)\}$ of training examples, each of which consist of an input vector $\mathbf{x}_i \in \mathbb{R}^n$ and the desired output $o_i \in \{0, 1\}$ for this input vector, $i = 1, \ldots, m$. Furthermore, let arbitrary initial weights $\mathbf{w}$ and an arbitrary initial threshold θ be given (e.g., chosen randomly). We consider online training first:

Algorithm 3.2 (Online Training of a Threshold Logic Unit)

```
procedure online_training (var w, var θ, L, η);
var y, e;                                (* output, sum of errors *)
begin
  repeat
    e := 0;                              (* initialize the sum of errors *)
    for all (x, o) ∈ L do begin          (* traverse the examples *)
      if (wx ≥ θ) then  y := 1;          (* compute the output of *)
                  else  y := 0;          (* the threshold logic unit *)
      if (y ≠ o) then begin              (* if the output is wrong *)
        θ := θ − η(o − y);               (* adapt the threshold *)
        w := w + η(o − y)x;              (* and the weights *)
        e := e + |o − y|;                (* sum the errors *)
      end;
    end;
  until (e ≤ 0);                        (* repeat the computations *)
end;                                     (* until the error vanishes *)
```

Obviously, this algorithm repeatedly applies the delta rule until the sum of errors over all training examples vanishes. Note that in this algorithm the weight adaptation is written in vector form, which, however, is clearly equivalent to an adaptation of the individual weights. Let us now turn to the batch version:

Algorithm 3.3 (Batch Training of a Threshold Logic Unit)

```
procedure batch_training (var w, var θ, L, η);
var y, e,                                  (* output, sum of errors *)
    θ_c, w_c;                              (* aggregated changes *)
begin
  repeat
    e := 0; θ_c := 0; w_c := 0;            (* initializations *)
    for all (x, o) ∈ L do begin            (* traverse the examples *)
      if (wx ≥ θ) then y := 1;             (* compute the output of *)
                  else y := 0;             (* the threshold logic unit *)
      if (y ≠ o) then begin                (* if the output is wrong *)
        θ_c := θ_c − η(o − y);             (* sum the threshold and *)
        w_c := w_c + η(o − y)x;            (* the weight changes *)
        e   := e   +  |o − y|;             (* sum the errors *)
      end;
    end;
    θ := θ + θ_c;                          (* adapt the threshold *)
    w := w + w_c;                          (* and the weights *)
  until (e ≤ 0);                           (* repeat the computations *)
end;                                       (* until the error vanishes *)
```

In this algorithm, the delta rule is applied in a modified form, since for each traversal of the training examples the same threshold and the same weights are used. If the output is wrong the computed changes are not applied directly, but summed in the variables θ_c and $\mathbf{w}_c$. Only after all training examples have been visited, the threshold and the weights are adapted with the help of these variables.

To illustrate the operation of these two algorithms, Table 3.2 shows the online training of the simple threshold logic unit considered above, which is to be trained in such a way that it computes the negation. Like in Fig. 3.16 on page 26 the initial values are $\theta = \frac{3}{2}$ and $w = 3$. It is easy to check that the online training shown here in tabular form corresponds exactly to the one shown graphically in Fig. 3.16 on the left. Analogously, Table 3.3 shows batch training. It corresponds exactly to the procedure depicted in Fig. 3.16 in the middle or on the right. Again, the fully trained threshold logic unit with the same parameters is depicted, together with its geometric interpretation, in Fig. 3.19.

As another example, we consider a threshold logic unit with two inputs that is to be trained in such a way that it computes the conjunction of its inputs. Such a threshold logic unit is shown, together with the corresponding training examples, in Fig. 3.20. For this example, we only consider online training. The corresponding training procedure for the initial values $\theta = w_1 = w_2 = 0$ with learning rate 1 is shown in Table 3.4. As for the negation, the training is successful and finally yields

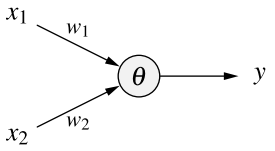
Table 3.2 Online training of a threshold logic unit for the negation with initial values $\theta = \frac{3}{2}$, $w = 2$ and learning rate 1

epoch	x	o	**xw**	y	e	$\Delta\theta$	Δw	θ	w
								1.5	2
1	0	1	−1.5	0	1	−1	0	0.5	2
	1	0	1.5	1	−1	1	−1	1.5	1
2	0	1	−1.5	0	1	−1	0	0.5	1
	1	0	0.5	1	−1	1	−1	1.5	0
3	0	1	−1.5	0	1	−1	0	0.5	0
	1	0	0.5	0	0	0	0	0.5	0
4	0	1	−0.5	0	1	−1	0	−0.5	0
	1	0	0.5	1	−1	1	−1	0.5	−1
5	0	1	−0.5	0	1	−1	0	−0.5	−1
	1	0	−0.5	0	0	0	0	−0.5	−1
6	0	1	0.5	1	0	0	0	−0.5	−1
	1	0	−0.5	0	0	0	0	−0.5	−1

Table 3.3 Batch training of a threshold logic unit for the negation with initial values $\theta = \frac{3}{2}$, $w = 2$ and learning rate 1

epoch	x	o	**xw**	y	e	$\Delta\theta$	Δw	θ	w
								1.5	2
1	0	1	−1.5	0	1	−1	0		
	1	0	0.5	1	−1	1	−1	1.5	1
2	0	1	−1.5	0	1	−1	0		
	1	0	−0.5	0	0	0	0	0.5	1
3	0	1	−0.5	0	1	−1	0		
	1	0	0.5	1	−1	1	−1	0.5	0
4	0	1	−0.5	0	1	−1	0		
	1	0	−0.5	0	0	0	0	−0.5	0
5	0	1	0.5	1	0	0	0		
	1	0	0.5	1	−1	1	−1	0.5	−1
6	0	1	−0.5	0	1	−1	0		
	1	0	−1.5	0	0	0	0	−0.5	−1
7	0	1	0.5	1	0	0	0		
	1	0	−0.5	0	0	0	0	−0.5	−1

Fig. 3.19 Learned threshold logic unit for the negation and its geometric interpretation



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

Fig. 3.20 A threshold logic unit with two inputs and training examples for the conjunction $y = x_1 \wedge x_2$

Table 3.4 Training of a threshold logic unit for the conjunction

Epoch	x_1	x_2	o	**xw**	y	e	$\Delta\theta$	Δw_1	Δw_2	θ	w_1	w_2
										0	0	0
1	0	0	0	0	1	−1	1	0	0	1	0	0
	0	1	0	−1	0	0	0	0	0	1	0	0
	1	0	0	−1	0	0	0	0	0	1	0	0
	1	1	1	−1	0	1	−1	1	1	0	1	1
2	0	0	0	0	1	−1	1	0	0	1	1	1
	0	1	0	0	1	−1	1	0	−1	2	1	0
	1	0	0	−1	0	0	0	0	0	2	1	0
	1	1	1	−1	0	1	−1	1	1	1	2	1
3	0	0	0	−1	0	0	0	0	0	1	2	1
	0	1	0	0	1	−1	1	0	−1	2	2	0
	1	0	0	0	1	−1	1	−1	0	3	1	0
	1	1	1	−2	0	1	−1	1	1	2	2	1
4	0	0	0	−2	0	0	0	0	0	2	2	1
	0	1	0	−1	0	0	0	0	0	2	2	1
	1	0	0	0	1	−1	1	−1	0	3	1	1
	1	1	1	−1	0	1	−1	1	1	2	2	2
5	0	0	0	−2	0	0	0	0	0	2	2	2
	0	1	0	0	1	−1	1	0	−1	3	2	1
	1	0	0	−1	0	0	0	0	0	3	2	1
	1	1	1	0	1	0	0	0	0	3	2	1
6	0	0	0	−3	0	0	0	0	0	3	2	1
	0	1	0	−2	0	0	0	0	0	3	2	1
	1	0	0	−1	0	0	0	0	0	3	2	1
	1	1	1	0	1	0	0	0	0	3	2	1

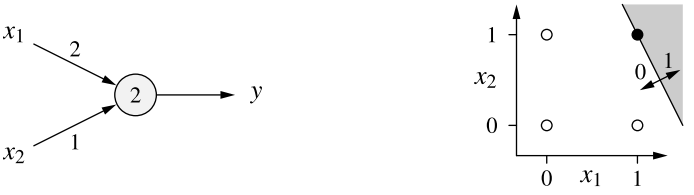


Fig. 3.21 Geometry of the learned threshold logic unit for $x_1 \wedge x_2$. The *straight line* shown on the right is described by the equation $2x_1 + x_2 = 3$

Table 3.5 Training of a threshold logic unit for the biimplication

epoch	x_1	x_2	o	$\mathbf{xw}$	y	e	$\Delta\theta$	Δw_1	Δw_2	θ	w_1	w_2
										0	0	0
1	0	0	1	0	1	0	0	0	0	0	0	0
	0	1	0	0	1	−1	1	0	−1	1	0	−1
	1	0	0	−1	0	0	0	0	0	1	0	−1
	1	1	1	−2	0	1	−1	1	1	0	1	0
2	0	0	1	0	1	0	0	0	0	0	1	0
	0	1	0	0	1	−1	1	0	−1	1	1	−1
	1	0	0	0	1	−1	1	−1	0	2	0	−1
	1	1	1	−3	0	1	−1	1	1	1	1	0
3	0	0	1	0	1	0	0	0	0	0	1	0
	0	1	0	0	1	−1	1	0	−1	1	1	−1
	1	0	0	0	1	−1	1	−1	0	2	0	−1
	1	1	1	−3	0	1	−1	1	1	1	1	0

the threshold $\theta = 3$ and the weights $w_1 = 2$ and $w_2 = 1$. The resulting threshold logic unit is shown, together with its geometric interpretation, in Fig. 3.21. Note that this threshold logic unit indeed computes the conjunction, even though the point $(1, 1)$ lies on the separating line, because it yields output 1 not only for points to the right of the line, but also for all points *on* the line.

After we have seen two examples of successful training, we naturally face the question whether Algorithms 3.2 and 3.3 always achieve their objective. As a first step, we can assert that these algorithms do not terminate if the function to be learned is *not* linearly separable. This is illustrated in Table 3.5 with the help of the online training procedure for the biimplication. Epochs 2 and 3 are clearly identical and will thus be repeated indefinitely, without a solution ever being found. However, this is not surprising, since the training procedure terminates only if the sum of the errors over all training examples vanishes. Since we know from Sect. 3.3 that there is no threshold logic unit that computes the biimplication, the error can never vanish and thus the algorithm cannot terminate.

For linearly separable functions, however, that is, for functions that can actually be computed by a threshold logic unit, it is guaranteed that the algorithms find a solution. That is, the following theorem holds.

Theorem 3.1 (Convergence theorem for the Delta Rule) *Let* $L = \{(\mathbf{x}_1, o_1), \ldots (\mathbf{x}_m, o_m)\}$ *be a set of training examples, each consisting of an input vector* $\mathbf{x}_i \in \mathbb{R}^n$ *and the desired output* $o_i \in \{0, 1\}$ *for this input vector. Furthermore, let* $L_0 = \{(\mathbf{x}, o) \in L \mid o = 0\}$ *and* $L_1 = \{(\mathbf{x}, o) \in L \mid o = 1\}$. *If* L_0 *and* L_1 *are linearly separable, that is, if there exist* $\mathbf{w} \in \mathbb{R}^n$ *and* $\theta \in \mathbb{R}$ *such that*

$$\forall (\mathbf{x}, 0) \in L_0 : \ \mathbf{wx} < \theta \quad \text{and}$$
$$\forall (\mathbf{x}, 1) \in L_1 : \ \mathbf{wx} \geq \theta,$$

then the Algorithms 3.2 and 3.3 terminate.

Proof The proof, which we do not want to spell out here, can be found, for example, in Rojas (1996) or in Nauck et al. (1997). □

Since both algorithms terminate only when the error vanishes, the computed values for the threshold and the weights are a solution of the learning problem.

3.6 Variants

All examples that we considered up to now referred to logical functions and we encoded *false* as 0 and *true* as 1. However, this encoding has the disadvantage that with an input of *false* the corresponding weight cannot be changed, because the formula for the weight change contains the input as a factor (see Definition 3.2 on page 27). This disadvantage can, in certain situations, slow down training unnecessarily, since a weight can only be adapted if the corresponding input is *true*.

To avoid this problem, the ADALINE model (ADAptive LINear Element) relies on the encoding *false* $\hat{=}$ -1 and *true* $\hat{=}$ $+1$. Thus an input of *false* also leads, provided the output is wrong, to a weight adaptation. Indeed, the delta rule was originally developed for the ADALINE model (Widrow and Hoff 1960), so that strictly we may only speak of the **delta rule** or the **Widrow–Hoff procedure** if the ADALINE model is employed. Although the procedure is equally applicable (and has the same convergence properties) for the encoding *false* $\hat{=}$ 0 and *true* $\hat{=}$ 1 (see the preceding section), it is sometimes called **error correction procedure** to avoid confusion (Nilsson 1965, 1998). We ignore this distinction here, because it is due to historical rather than conceptual reasons.

3.7 Training Networks

After simple neuro-computers had been used successfully for pattern recognition tasks at the end of the 1950s (for example, Rosenblatt 1958), the simple and fast delta rule training method had been developed by Widrow and Hoff (1960) and the perceptron convergence theorem (corresponds to the convergence theorem for the delta rule) had been proven by Rosenblatt (1962), great expectations were placed in the development of (artificial) neural networks. This started the "first bloom" of neural network research, in which it was believed that one had already discovered the core principles underlying systems that are able to learn.

Only after Minsky and Papert (1969) carried out a careful mathematical analysis of the perceptron and pointed out in all clarity that threshold logic units can compute only linearly separable functions, the limitations of the models and procedures used at the time were properly recognized. Although it was already known from the early works of McCulloch and Pitts (1943), that the limitations of the expressive and computational power can be lifted by using *networks* of threshold logic

units (since, for example, such networks can compute arbitrary Boolean functions), training methods were confined to *single* threshold logic units.

Unfortunately, transferring the training procedures to networks of threshold logic units turned out to be a surprisingly difficult problem. For example, the delta rule derives the weight adaptation from the difference between the actual and the desired output (see Definition 3.2 on page 27). However, a desired output is available only for the neuron that yields the output of the network as a whole. For all other threshold logic units, which carry out some kind of preprocessing and transmit their outputs only to other threshold logic units, no such desired output can be given. As an example, consider the biimplication problem and the structure of the network that we proposed as a solution for this problem (Fig. 3.10 on page 21): the training examples do not state desired outputs for the two threshold logic units on the left. One of the main reasons for this is that the necessary coordinate transformation is not uniquely determined: separating lines in the input space may just as well be placed in a completely different way (for example, perpendicular to the bisectrix) or one may direct the normal vectors somewhat differently.

As a consequence (artificial), neural networks were seen as a "research dead end" and the so-called "dark age" of neural network research began. Only when the training procedure of **error backpropagation** was developed, the area was revived. This procedure was described first in Werbos (1974), but did not receive any attention. Only when Rumelhart et al. (1986a, 1986b) independently developed the method again and advertised it in the research community, the modern age ("second bloom") of (artificial) neural network began, which last to the present day.

We consider error backpropagation only in Chap. 5, since it cannot be applied directly to threshold logic units. It requires that the activation of a neuron does not jump at a crisply defined threshold from 0 to 1, but that the activation rises slowly, according to a differentiable function. For networks consisting of pure threshold logic units, still no training method is known.

References

J.A. Anderson and E. Rosenfeld. *Neurocomputing: Foundations of Research*. MIT Press, Cambridge, MA, USA, 1988

M.A. Boden, ed. *The Philosophy of Artificial Intelligence*. Oxford University Press, Oxford, United Kingdom, 1990

W.S. McCulloch. *Embodiments of Mind*. MIT Press, Cambridge, MA, USA, 1965

W.S. McCulloch and W.H. Pitts. A Logical Calculus of the Ideas Immanent in Nervous Activity. *Bulletin of Mathematical Biophysics* 5:115–133, 1943, USA. Reprinted in McCulloch (1965), 19–39, in Anderson and Rosenfeld (1988), 18–28, and in Boden (1990), 22–39

L.M. Minsky and S. Papert. *Perceptrons*. MIT Press, Cambridge, MA, USA, 1969

D. Nauck, F. Klawonn, and R. Kruse. *Foundations of Neuro-Fuzzy Systems*. J. Wiley & Sons, Chichester, United Kingdom, 1997

N.J. Nilsson. *Learning Machines: The Foundations of Trainable Pattern-Classifying Systems*. McGraw-Hill, New York, NY, 1965

N.J. Nilsson. *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann, San Francisco, CA, USA, 1998

R. Rojas. *Theorie der neuronalen Netze—Eine systematische Einführung*. Springer-Verlag, Berlin, Germany, 1996

F. Rosenblatt. The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. *Psychological Review* 65:386–408, 1958, USA

F. Rosenblatt. *Principles of Neurodynamics*. Spartan Books, New York, NY, USA, 1962

D.E. Rumelhart and J.L. McClelland, eds. *Parallel Distributed Processing: Explorations in the Microstructures of Cognition, Vol. 1: Foundations*, 1986

D.E. Rumelhart, G.E. Hinton and R.J. Williams. Learning Internal Representations by Error Propagation. In Rumelhart and McClelland (1986), 318–362, 1986a

D.E. Rumelhart, G.E. Hinton and R.J. Williams. Learning Representations by Back-Propagating Errors. *Nature* 323:533–536, 1986b

P.D. Wasserman. *Neural Computing: Theory and Practice*. Van Nostrand Reinhold, New York, NY, USA, 1989

P.J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. Ph.D. Thesis, Harvard University, Cambridge, MA, USA, 1974

R.O. Widner. Single State Logic. *AIEE Fall General Meeting*, 1960. Reprinted in Wasserman (1989)

B. Widrow and M.E. Hoff. Adaptive Switching Circuits. *IRE WESCON Convention Record*, 96–104. Institute of Radio Engineers, New York, NY, USA, 1960

A. Zell. *Simulation Neuronaler Netze*. Addison-Wesley, Stuttgart, Germany, 1996