Chapter 1
Why We Need Complex-Valued Neural Networks?

"Why is my verse so barren of new pride,
So far from variation or quick change?
Why with the time do I not glance aside
To new-found methods and to compounds strange?"

William Shakespeare, Sonnet 76

This chapter is introductory. A brief observation of neurons and neural networks is given in Section 1.1. We explain what is a neuron, what is a neural network, what are linearly separable and non-linearly separable input/output mappings. How a neuron learns is considered in Section 1.2, where Hebbian learning, the perceptron, and the error-correction learning rule are presented. In Section 1.3, we consider a multilayer feedforward neural network and essentials of backpropagation learning. The Hopfield and cellular neural networks are also presented. Complex-valued neural networks, their naturalness and necessity are observed in Section 1.4. It is shown that a single complex-valued neuron can learn non-linearly separable input/output mappings and is much more functional than a single real-valued neuron. Historical observation of complex-valued neural networks and the state of the art in this area are also presented. Some concluding remarks will be given in Section 1.5.

1.1 Neurons and Neural Networks: Basic Foundations and Historical View

1.1.1 What Is a Neural Network?

As we have clearly mentioned, this book is devoted to complex-valued neural networks, even only to those of them that are based on multi-valued neurons. However, it should not be correct, if we will start immediately from complex-valued neurons and neural networks. To understand, why complex-valued neurons were introduced and to understand that motivation, which was behind their introduction, it is important to observe what a neural network is. It is also important to have a good imagination about those solutions that existed in neural networks that time when the first complex-valued neuron was proposed and about state of the art

I. Aizenberg: Complex-Valued Neural Networks with Multi-Valued Neurons, SCI 353, pp. 1–53.
springerlink.com

in neural networks, to understand why complex-valued neurons are even more important today. It is also important to understand those limitations that are specific for real-valued neurons and neural networks. This will lead us to much clearer understanding of the importance of complex-valued neurons and the necessity of their appearance for overcoming limitations and disadvantages of their real-valued counterparts.

So let us start from the brief historical overview.

What an artificial neural network is? Among different definitions, which the reader can find in many different books, we suggest to use the following given in [1] by Igor Aleksander and Helen Morton, and in [2] by Simon Haykin.

Definition 1.1. A *neural network* is a massively parallel distributed processor that has a natural propensity for storing experimental knowledge and making it available for use. It means that: 1) Knowledge is acquired by the network through a learning process; 2) The strength of the interconnections between neurons is implemented by means of the synaptic weights used to store the knowledge.

Let us consider in more detail what stands behind this definition. It is essential that an artificial neural network is a massively parallel distributed processor whose basic processing elements are artificial neurons. The most important property of any artificial neural network and of its basic element, an artificial neuron, is their ability to learn from their environment. *Learning* is defined in [2] as a process by which the free parameters of a neural network (or of a single neuron) are adapted through a continuing process of simulation by the environment in which the network (the neuron) is embedded. This means that both a single artificial neuron and an artificial[1] neural network are *intelligent systems*. They do not perform computations according to the pre-defined externally loaded program, but they *learn* from their environment formed by learning samples that are united in a *learning set*. Once the learning process is completed, they are able to generalize relying on that knowledge, which was obtained during the learning process. The quality of this generalization is completely based on that knowledge, which was obtained during the learning process.

Compared to biological neural networks, artificial neural networks are "neural" in the sense that they have been inspired by neuroscience, but they are not true models of biological or cognitive phenomena. The important conclusion about artificial neural networks, which is done by Jacek Zurada in [3], states that typical neural network architectures are more related to mathematical and/or statistical techniques, such as non-parametric pattern classifiers, clustering algorithms, nonlinear filters, and statistical regression models.

In contrast to algorithmic approaches usually tailored to tasks at hand, neural networks offer a wide palette of versatile modeling techniques applicable to a large class of problems. Here, learning in data-rich environments leads to models of specific tasks. Through learning from specific data with rather general

[1] We will omit further the word "artificial" keeping in mind that across this book we have deal with artificial neurons and artificial neural networks. Wherever it will be needed, when a biological neuron will be considered, we will add the word "biological".

neural network architectures, neurocomputing techniques can produce problem-specific solutions [3].

1.1.2 The Neuron

We just told that there are many equivalent definitions of a neural network. However, it is quite difficult to find a strict definition of a neuron. We may say that an artificial neuron is on the one hand, an abstract model of a biological neuron, but on the other hand, it is an intelligent information processing element, which can learn and can produce the output in response to its inputs. As a result of learning process, the neuron forms a set of *weights* corresponding to its inputs. Then by weighting summation of the inputs and transformation of the weighted sum of input signals using an *activation* (*transfer*) function it produces the output.

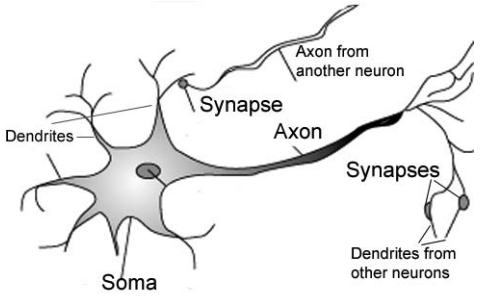


Fig. 1.1 A schematic model of a biological neuron

This is really quite similar to what a biological neuron is doing. Let us consider its schematic model (see Fig. 1.1). Indeed, a biological neuron receives input signals thorough its *dendrites* that are connected to *axons* (which transmit output signals) of other neurons via *synapses* where the input signals are being weighted by the synaptic weights. Then the biological neuron performs a weighting summation of inputs in *soma* where it also produces the output, which it transmits to the dendrites of other neurons through the synaptic connections.

The first artificial neuron model was proposed by W. McCulloch and W. Pitts in 1943 [4]. They tried to create a mathematical model of neural information processing as it was considered that time. A common view was that a neuron receives some input signals $x_1,...,x_n$ that can be excitatory ("1") or inhibitory ("-1"), calculates the weighted sum of inputs $z = w_1x_1 + ... + w_nx_n$ and then produces the excitatory output ("1") if the weighted sum of inputs exceeds some predetermined threshold value and the inhibitory output ("-1") if it does not. For many years, it is a commonly known fact that a biological neuron is much more sophisticated from the signal processing view point. It is not a discrete binary processing element, its inputs and outputs are continuous, etc. Thus, the McCulloch-Pitts model as a model of a biological neuron is very schematic and it just approaches a basic idea of neural information processing. Nevertheless it is difficult to overestimate the importance of this model. First of all, it is historically the first model of a neuron. Secondly, this model was important for understanding of learning mechanisms that we will consider below. Thirdly, all later neural models are based on the same approach that

was in the McCulloch-Pitts model: weighted summation of inputs followed by the transfer function applied to the weighted sum to produce the output.

Let us take a closer look at the McCulloch-Pitts model. As we have mentioned, in this model the neuron is a binary processing element. It receives binary inputs $x_1,...,x_n$ taken their values from the set {-1, 1} and produces the binary output belonging to the same set. The weights $w_1,...,w_n$ can be arbitrary real numbers and therefore the weighted sum $z = w_1x_1 + ... + w_nx_n$ can also be an arbitrary real number. The neuron output $f\left(x_1,...,x_n\right)$ is determined as follows:

$$f\left(x_1,...,x_n\right) = \begin{cases} 1, \text{if } z \geq \Theta \\ -1, \text{if } z < \Theta, \end{cases}$$

where Θ is the pre-determined threshold. The last equation can be transformed if the threshold will be included to the weighted sum as a "free weight" $w_0 = -\Theta$, which is often also called a *bias* and the weighted sum will be transformed accordingly ($z = w_0 + w_1x_1 + ... + w_nx_n$):

$$f\left(x_1,...,x_n\right) = \begin{cases} 1, \text{if } z \geq 0 \\ -1, \text{if } z < 0. \end{cases}$$

This is the same as

$$f\left(x_1,...,x_n\right) = \text{sgn}\left(z\right), \tag{1.1}$$

where *sgn* is a standard sign function, which is equal to 1 when its argument is non-negative and to -1 otherwise (see Fig. 1.2). Thus, function *sgn* in (1.1) is an *activation function*. It is usually referred to as the *threshold activation function*. The McCulloch-Pitts neuron is also often called the *threshold element* or the *threshold neuron*. [5]. These names were especially popular in 1960s – 1970s.

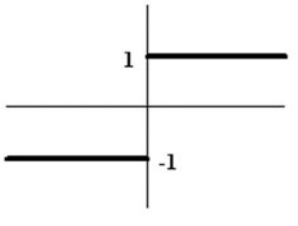


Fig. 1.2 Threshold Activation Function

It is important to mention that function *sign* is nonlinear. Hence, the first neuron was a nonlinear processing element. This property is very important. All popular activation functions that are used in neurons are nonlinear. It will not be the overestimation, if we will say that the functionality of a neuron is mainly (if not completely) determined by its *activation function*.

Let us consider now the most general model of a neuron, which is commonly used today (see Fig. 1.3). A neuron has *n* inputs $x_1,...,x_n$ and weights $w_1,...,w_n$

corresponding to these inputs. It also has a "free weight" (bias) w_0, which does not correspond to any input. All together weights form an (*n*+1)-dimensional *weighting vector* $(w_0, w_1, ..., w_n)$. There is a pre-determined activation function $\varphi(z)$ associated with a neuron. It generates the neuron output limiting it to some reasonable (permissible) range. The neural processing consists of two steps. The first step is the calculation of the weighted sum of neuron inputs

$$z = w_0 + w_1 x_1 + ... + w_n x_n .$$

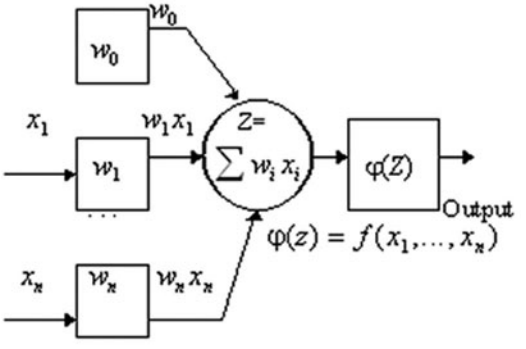


Fig. 1.3 A general model of a neuron

The second step is the calculation of the value of the activation function $\varphi(z)$ for the value z of the weighted sum. This value of the activation function forms the output of the neuron. If input/output mapping is described by some function $f(x_1, ..., x_n)$, then

$$f(x_1, ..., x_n) = \varphi(z) = \varphi(w_0 + w_1 x_1 + ... + w_n x_n). \tag{1.2}$$

Initially only binary neuron inputs and output were considered. Typically, they were taken from the set $E_2 = \{1, -1\}$ or (rarely) from the set $K_2 = \{0, 1\}$[2]. It is important to mention that it is very easy to move from one of these alphabets to another one. For example, if $y \in K_2$ then $x = 1 - 2y \in E_2$, and if $x \in E_2$ then $y = -(x-1)/2 \in K_2$, respectively. Hence, $0 \leftrightarrow 1$, $1 \leftrightarrow -1$. As for the weights, they were taken from the set $\mathbb{R}$ of real numbers. Therefore, the weighted sum in this case is also real and an activation function is a function of a real variable. We may say that mathematically the threshold neuron implements a mapping $f(x_1, ..., x_n): E_2^n \to E_2$.

[2] In [4], in the original McCulloch-Pitts model, a classical Boolean alphabet $K_2 = \{0, 1\}$ was used. However, especially for the learning purpose, the bipolar alphabet $E_2 = \{1, -1\}$ is much more suitable. We will consider a bit later, why it is better to use the bipolar alphabet for learning.

1.1.3 Linear Separability and Non-linear Separability: XOR Problem

If the neuron performs a mapping $f(x_1,...,x_n): E_2^n \to E_2$, this means that $f(x_1,...,x_n)$ is a Boolean function. If the function *sgn* is used as the activation function of a neuron (thus, if it is the threshold neuron), then this Boolean function is called and commonly referred to as a *threshold* (*linearly separable*) Boolean function. Linear separability means that there exists an *n*-dimensional hyperplane determined by the corresponding weights (it is evident that the equation $z = w_0 + w_1 x_1 + ... + w_n x_n$ determines a hyperplane in an *n*-dimensional space) and separating 1s of this function from its -1s (or 0s from 1s if the classical Boolean alphabet $K_2 = \{0,1\}$ is used). It is very easy to show this geometrically for *n*=2. Let us consider the function $f(x_1, x_2) = x_1$ or x_2, the disjunction of the two Boolean variables. A table of values of this function is shown in Table 1.1.

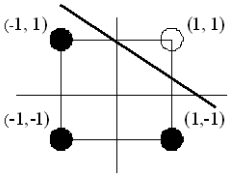
Fig. 1.4a demonstrates a geometrical interpretation of this function. It also shows what a linear separability is. There is a line, which separates a single "1" value of this function from three "-1" values. It is also clear that there exist infinite amount of such lines. In 1960s study of threshold Boolean functions was very popular.

Table 1.1 Values of function $f(x_1, x_2) = x_1$ or x_2

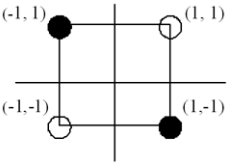
x_1	x_2	$f(x_1, x_2) = x_1$ or x_2
1	1	1
1	-1	-1
-1	1	-1
-1	-1	-1

We can mention at least two comprehensive monographs devoted to this subject [6, 7]. However, the number of threshold or linearly separable Boolean functions is very small. While for *n*=2 there are 14 threshold functions out of 16 and for *n*=3 there are 104 threshold functions out of 256, for *n*=4 there are just about 2000 threshold functions out of 65536. For $n > 4$, the ratio of the number of threshold Boolean functions of *n* variables to 2^{2^n} (the number of all Boolean functions of *n* variables) approaches 0.

While threshold Boolean functions can be implemented using a single threshold neuron, other functions that are not threshold cannot. May be the most typical and the most popular example of such a function is XOR problem



(a) $f(x_1, x_2) = x_1$ or x_2 is a linearly separable function. There exists a line, which separates 1 value of this function (a transparent circle) from its -1s (filled circles)



(b) $f(x_1, x_2) = x_1$ xor x_2 is a non-linearly separable function. There is no way to find a line, which separates 1s value of this function (transparent circles) from its -1s (filled circles)

Fig. 1.4

(the Exclusive OR) $f(x_1, x_2) = x_1$ xor x_2, mod 2 sum of the two Boolean variables. This function is non-linearly separable. Let us take a look at the table of values of this function (see Table 1.2) and its graphical representation (see Fig. 1.4b). Geometrically, this problem belongs to the classification of the points in the hypercube, as any problem described by the Boolean function (see Fig. 1.4). Each point in the hypercube is either in class "1" or class "-1". In the case of XOR problem the input patterns (1, 1) and (-1, -1) that are in class "1" are at the opposite corners of the square (2D hypercube). On the other hand, the input patterns (1, -1) and (-1, 1) are also at the opposite corners of the same square, but they are in class "-1". It is clear from this that the function XOR is non-linearly separable, because there is no way to draw a line, which can separate two “1” values of this function from its two “-1” values, which is clearly seen from Fig. 1.4b. Since such a line does not exist, there are no weights using which XOR function can be implemented using a single threshold neuron.

Table 1.2 Values of function $f(x_1, x_2) = x_1$ xor x_2

x_1	x_2	$f(x_1, x_2) = x_1$ xor x_2
1	1	1
1	-1	-1
-1	1	-1
-1	-1	1

The existence of non-linearly separable problems was a starting point for neural networks design and likely the XOR problem stimulated creation of the first multilayer neural network. We will consider this network in Section 1.3. However, the most important for us will be the fact that XOR problem can be easily solved using a single complex-valued neuron. We will show this solution in Section 1.4.

1.2 Learning: Basic Fundamentals

1.2.1 Hebbian Learning

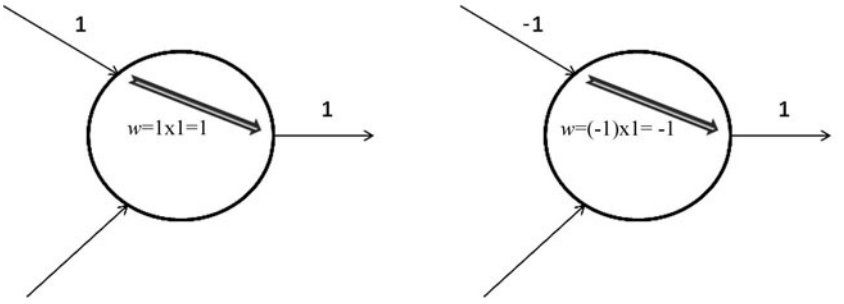
We told from the beginning that the main property of both a single neuron and any neural network is their ability to learn from their environment. How a neuron learns? The first model of the learning process was developed by Donald Hebb in 1949 [8]. He considered how biological neurons learn. As we have already mentioned, biological neurons are connected to each other through synaptic connections: axon of one neuron is connected to dendrites of other ones through synapses (Fig. 1.1). To represent the Hebbian model of learning, which is commonly referred to as *Hebbian learning*, let us cite D. Hebb's fundamental book [8] directly. The idea of the Hebbian learning is as follows ([8], p. 70).

"The general idea is … that any two cells or systems of cells that are repeatedly active at the same time will tend to become 'associated', so that activity in one facilitates activity in the other."

The mechanism of Hebbian learning is the following ([8], p. 63).

"When one cell repeatedly assists in firing another, the axon of the first cell develops synaptic knobs (or enlarges them if they already exist) in contact with the soma of the second cell."

Let us "translate" this idea and mechanism into the language of the threshold neuron. In this language, "1" that is a "positive" signal, means excitation, and "-1" that is a "negative" signal, means inhibition. When the neuron "fires" and produces "1" in its output, this means that weights have to help this neuron to "fire". For example, if the neuron receives a "positive" signal ("1") from some input, then the corresponding weight passing this signal can be obtained by multiplication of the desired output "1" by the input "1" (see Fig. 1.5a). Thus, the weight is equal to 1 and the "positive" input signal will contribute to the positive output of the neuron. Indeed, to produce a "positive" output, according to (1) the weighted sum must be positive. On the contrary, if the neuron "fires", but from some input it receives a "negative" (inhibitory) signal, the corresponding weight has to invert this signal, to make its contribution to the weighted sum and the neuron output positive. Again, the simplest way to achieve this, is to multiply the desired output "1" by the input "-1" (see Fig. 1.5b). The corresponding weight will be equal to -1 and when multiplied by the input, will produce a positive contribution $(-1)\cdot(-1)=1$ to the weighted sum and output. Respectively, if the neuron does not "fire" and has to produce a "negative" inhibitory output ("-1"), the weights have to help to inhibit the neuron and to produce a negative weighted sum. The weights should be found in the same way: by multiplication of the desired output "-1" by the corresponding input value. If the input is $"-1"$ (inhibitory), then the weight $(-1)\cdot(-1)=1$ just passes it (see Fig. 1.5c). If the input is "1" (excitatory), the weight $(-1)\cdot 1=-1$ inverts it (see Fig. 1.5d).



(a) the neuron "fires" and a "firing" input is passed to the output by the positive weight

(b) the neuron "fires" and an "inhibitory" input is inverted by the negative weight

(c) the neuron "inhibits" and an "inhibitory" input is passed to the output by the positive weight

(d) the neuron "inhibits" and a "firing" input is inverted by the negative weight

Fig. 1.5 Calculation of the weight using the Hebb rule for one of the neuron's inputs and for a single learning sample: the weight is equal to the product of the desired output and input value

To obtain a bias w_0, we just need to multiply the desired output by 1, which can be considered as a "virtual constant input" corresponding to this weight.

It is clear that if the threshold neuron has to learn only a single learning sample, then this Hebb rule always produces the weighting vector implementing the corresponding input/output mapping. However, learning from a single learning sample is not interesting, because it is trivial. What about multiple learning samples? In this case, the weights can be found by generalization of the rule for a single learning sample, which we have just described. This generalization leads us to the following representation of the Hebbian learning rule for a single threshold neuron. Let us have N n-dimensional learning samples (this means that our neuron has n inputs $x_1,...,x_n$). Let f be an N-dimensional vector-column[3] of output

[3] Here and hereafter we will use a notation $\mathrm{f} = (f_1,...,f_n)^T$ for a vector-column, while a notation $F = (f_1,...,f_n)$ will be used for a vector-row.

values. Let $\mathbf{x}_1,...,\mathbf{x}_n$ be *N*-dimensional vectors of all possible values of inputs $x_1^j,...,x_n^j, j=1,...,N$.

Then according to the Hebbian learning rule the weights $w_1,...,w_n$ should be calculated as dot products of vector f and vectors $\mathbf{x}_1,...,\mathbf{x}_n$, respectively. The weight w_0 should be calculated as a dot product of vector f and an *N*-dimensional vector-constant $\mathbf{x}_0=(1,1,...,1)^T$:

$$w_i=(\mathbf{f},\mathbf{x}_i), i=0,...,n, \tag{1.3}$$

where $(\mathbf{a},\mathbf{b})=a_1\overline{b}_1+...+a_n\overline{b}_n$ is the dot product of vector-columns $\mathbf{a}=(a_1,...,a_n)^T$ and $\mathbf{b}=(b_1,...,b_n)^T$ in the unitary space ("bar" is a symbol of complex conjugation, in the real space it should simply be ignored).

It can also be suggested to normalize the weights obtained by (1.3):

$$w_i=\frac{1}{N}(\mathbf{f},\mathbf{x}_i), i=0,...,n. \tag{1.4}$$

Let us check how rule (1.4) works.

Example 1.1 Let us learn using this rule the OR problem $f(x_1,x_2)=x_1$ or x_2, which is linearly separable and which we have already considered for illustration of the linear separability (Table 1.1, Fig. 1.4a). Let us use rule (1.4) to obtain the weights. From Table 1.1, we have $\mathbf{f}=(1,-1,-1,-1)^T;\mathbf{x}_0=(1,1,1,1)^T;\mathbf{x}_1=(1,1,-1,-1)^T;\mathbf{x}_2=(1,-1,1,-1)^T$. Then, applying Hebbian learning rule (1.4), we obtain the following weights $w_0=(\mathbf{f},\mathbf{x}_0)=-0.5; w_1=(\mathbf{f},\mathbf{x}_1)=0.5; w_2=(\mathbf{f},\mathbf{x}_2)=0.5$. Let us now check the results of this learning and apply the weighting vector $W=(-0.5,0.5,0.5)$ to all four possible binary inputs of the threshold neuron. The results are summarized in Table 1.3. We see that the weighting vector, which we obtained learning the OR function using the Hebbian learning rule really implements the OR function using the threshold neuron.

The reader for whom neural networks is a new subject may say "Hurrah! It so simple and beautiful!" It is really simple and beautiful, but unfortunately just a minority of all threshold Boolean functions of more than two variables can be learned in this way.

It is also clear that neither of non-threshold Boolean functions can be learned by the threshold neuron using rule (1.4) (non-threshold functions cannot be learned by the threshold neuron at all). By the way, different non-threshold Boolean functions may have the same weighting vectors obtained by rule (1.4). If we apply rule (1.4) to such a non-threshold Boolean function, like XOR (see Table 1.2 and Fig. 1.4b), which is symmetric (self-dual (or odd, in other words) or even), we get the zero weighting vector (0, …, 0).

Table 1.3 Threshold neuron implements $f(x_1, x_2) = x_1$ or x_2 function with the weighting vector (-0.5, 0.5, 0.5) obtained by Hebbian learning rule (1.4)

x_1	x_2	$z = w_0 + w_1 x_1 + w_2 x_2$	$\mathrm{sgn}(z)$	$f(x_1, x_2) = x_1$ or x_2
1	1	0.5	1	1
1	-1	-0.5	-1	-1
-1	1	-0.5	-1	-1
-1	-1	-1.5	-1	-1

The following natural questions can now be asked by the reader. How those threshold Boolean functions that cannot be learned using the Hebb rule, can be learned? What about multiple-valued and continuous input/output mappings, is it possible to learn them? If the Hebb rule has a limited capability, is it useful? The answer to the first question will be given right in the next Section. Several answers to the second question will be given throughout this book. The third question can be answered right now. The importance of Hebbian learning is very high, and not only because D. Hebb for the first time explained mechanisms of associations developing during the learning process. A vector obtained using the Hebb rule, even if it does not implement the corresponding input/output mapping, can often be a very good first approximation of the weighting vector because it often can "draft" a border between classes when solving pattern recognition and classification problems. In [6] it was suggested to call a vector obtained by (1.4) for a Boolean function the *characteristic vector* of that function. Later the same notion was considered for multiple-valued functions and the Hebb rule was used to learn them using the multi-valued neuron. We will consider this aspect of Hebbian learning later when we will consider multi-valued neurons and their applications (Chapters 2-6).

Using the Hebbian learning it is possible to develop associations between the desired outputs and those inputs that stimulate these outputs. However, the Hebbian learning cannot correct the errors if those weights obtained by the Hibbian rule still do not implement the corresponding input/output mapping. To be able to correct the errors (to adjust the weights in such a way that the error will be minimized or eliminated), it is necessary to use the error-correction learning rule.

1.2.2 Perceptron and Error-Correction Learning

The *perceptron* is historically the first artificial neural network. The perceptron was suggested in 1958 by Frank Rosenblatt in [9] as "a hypothetical nervous

system", and as the illustration of "some of the fundamental properties of intelligent systems in general".

Today we may say that the Rosenblatt's perceptron as it was defined in the seminal paper [9] is the simplest feedforward neural network (it will be considered in Section 1.3), but in 1958 when F. Rosenblatt published his paper, this today's most popular kind of a neural network was not invented yet. The perceptron was suggested as a network consisted of three types of elements that can simulate recognition of visual patterns (F. Rosenblatt demonstrated the perceptron's ability to recognize English typed letters). The perceptron in its original concept contained three types of units (Fig. 1.6): *S*-units (sensory) for collecting the input information and recoding it into the form appropriate for *A*-units (associate units), and *R*-units (responses). While *S*-units are just sensors (like eye retina) and *R*-units are just responsible for reproduction of the information in terms suitable for its understanding, *A*-units are the neurons, for example the ones with the threshold activation function (later a sigmoid activation was suggested, we will also consider it below). Thus, *A*-units form a single layer feedforward neural network. All connections among units were usually built at random.

The main idea behind the perceptron was to simulate a process of pattern recognition. At that time when the perceptron concept was suggested, classification was considered only as a binary problem (a two-class classification problem), and the perceptron was primarily used as a binary classifier. Thus, each neuron (each A-unit) performed only input/output mappings $f(x_1,...,x_n): E_2^n \to E_2$ (or $f(x_1,...,x_n): K_2^n \to K_2$ depending on which Boolean alphabet was used). Later it was suggested to consider a more general case when neuron (perceptron) inputs are real numbers from some bounded set $T \subset \mathbb{R}$ (often the case of $T = [0,1]$ is considered). Thus, if $x_i \in T, i = 1,...,n$ a mapping performed by the neuron becomes $f(x_1,...,x_n): T^n \to E_2$.

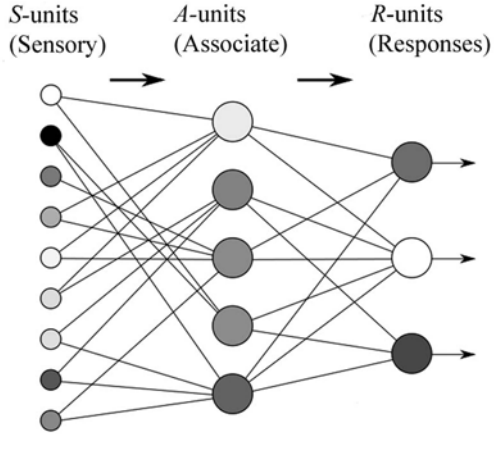


Fig. 1.6 The Perceptron

One of the main achievements of the perceptron era was the error-correction learning concept first suggested by F. Rosenblatt in [10] and then developed and deeply presented in his monograph [11]. Since in the perceptron all its *A*-units learn separately and independently, we may consider the error-correction learning rule with regard to a single neuron. We will derive the error-correction learning

rule in the same way as it was done by its inventor. We have to mention that this learning rule will be very important for us when we will consider its generalization for complex-valued neurons and neural networks.

Let us consider the threshold neuron with activation function (1.1). Suppose a neuron has to learn some input/output mapping $f(x_1,...,x_n): E_2^n \to E_2$. This input/output mapping could represent, for example, some binary classification problem. Thus, there are two classes of objects described by *n*-dimensional real-valued vectors. The purpose of the learning process in this case is to train a neuron to classify patterns labeling them as belonging to the first or the second class. Let $d_i \in E_2$ be the desired output for the *i*th learning sample. This means that the input/output mapping has to map a vector $(x_1,...,x_n)$ to some desired output *d*. Suppose we have *N* learning samples that form a learning set $(x_1^i,...,x_n^i) \to d_i, i = 1,...,N$. Let us have some weighting vector $W = (w_0, w_1,..., w_n)$ (the weights can be generated, for example, by a random number generator). Let y be the actual output of the neuron $y = \mathrm{sgn}(w_0 + w_1x_1 + ... + w_nx_n)$ and it does not coincide with the desired output *d*. This forms the error

$$\delta = d - y. \tag{1.5}$$

Evidently, the goal of the learning process should be the elimination or minimization of this error through the adjustment of the weights by adding to them the adjustment term Δw

$$\tilde{w}_i = w_i + \Delta w_i, i = 0,1,...,n. \tag{1.6}$$

We expect that once the weights will be adjusted, our neuron should produce the desired output

$$d = \mathrm{sgn}(\tilde{w}_0 + \tilde{w}_1x_1 + ... + \tilde{w}_nx_n). \tag{1.7}$$

Taking into account (1.5) and (1.6), (1.7) can be transformed as follows

$$\begin{aligned} &d = \delta + y = \\ &\mathrm{sgn}\big((w_0 + \Delta w_0) + (w_1 + \Delta w_1)x_1 + ... + (w_n + \Delta w_n)x_n\big). \end{aligned} \tag{1.8}$$

Then we obtain from (1.8) the following

$$\begin{aligned} &\delta + y = \\ &\mathrm{sgn}\big((w_0 + w_1x_1 + ... + w_nx_n) + (\Delta w_0 + \Delta w_1x_1 + ... + \Delta w_nx_n)\big). \end{aligned} \tag{1.9}$$

Since the neuron's output is binary and it can be equal only to 1 or -1, according to (1.5) we have the following two cases for the error

$$\delta = \begin{cases} 2, \text{if } d = 1, y = -1 \\ -2, \text{if } d = -1, y = 1. \end{cases} \tag{1.10}$$

Let us consider the first case from (1.10), $d = 1, y = -1, \delta = 2$. Substituting these values to (1.9), we obtain the following

$$\begin{aligned} &\delta + y = 2 - 1 = 1 = \\ &\operatorname{sgn}\left(\left(w_0 + w_1 x_1 + \ldots + w_n x_n\right) + \left(\Delta w_0 + \Delta w_1 x_1 + \ldots + \Delta w_n x_n\right)\right). \end{aligned} \tag{1.11}$$

It follows from the last equation that

$$0 \le \left(w_0 + w_1 x_1 + \ldots + w_n x_n\right) + \left(\Delta w_0 + \Delta w_1 x_1 + \ldots + \Delta w_n x_n\right),$$

and (since $w_0 + w_1 x_1 + w_n x_n < 0$ because $y = -1$)

$$\begin{aligned} &0 < \left(\Delta w_0 + \Delta w_1 x_1 + \ldots + \Delta w_n x_n\right), \\ &\left|w_0 + w_1 x_1 + \ldots + w_n x_n\right| \le \left|\Delta w_0 + \Delta w_1 x_1 + \ldots + \Delta w_n x_n\right| \end{aligned}. \tag{1.12}$$

Let us set

$$\Delta w_0 = \alpha\delta; \Delta w_i = \alpha\delta x_i, i = 1, \ldots, n, \tag{1.13}$$

where $\alpha > 0$ is some constant, which is called a *learning rate*. Then

$$\begin{aligned} &\Delta w_0 + \Delta w_1 x_1 + \ldots + \Delta w_n x_n = \\ &\alpha\delta + \alpha\delta x_1 x_1 + \ldots + \alpha\delta x_n x_n = \alpha\delta(n+1). \end{aligned} \tag{1.14}$$

It is important to mention that $x_i x_i = x_i^2 = 1; i = 1, \ldots, n$ in (14) because since we consider the threshold neuron with binary inputs, $x_i \in E_2 = \{1, -1\}$. We will see later that it is more difficult to use (1.13) if the neuron inputs are not binary. We also will see later that this difficulty does not exist for the error-correction learning rule for the multi-valued neuron, which will be considered in Section 3.3.

Since $\alpha > 0, \delta = 2 > 0$, then $\alpha\delta(n+1) > 0$ and the 1st inequality from (1.12) holds. However, it is always possible to find a learning rate $\alpha > 0$ such that the 2nd inequality from (1.12) also holds. This means that for the first case in (1.10) the learning rule based on (1.6) and (1.13) guarantees that (1.11) is true and the neuron produces the correct result after the weights are adjusted.

Let us consider the second case in (1.10). $d = -1, y = 1, \delta = -2$. Substituting these values to (1.9), we obtain the following

$$\begin{aligned} &\delta + y = -2 + 1 = -1 = \\ &\operatorname{sgn}\left(\left(w_0 + w_1 x_1 + \ldots + w_n x_n\right) + \left(\Delta w_0 + \Delta w_1 x_1 + \ldots + \Delta w_n x_n\right)\right). \end{aligned} \tag{1.15}$$

It follows from the last equation that

$$\left(w_0 + w_1 x_1 + ... + w_n x_n\right) + \left(\Delta w_0 + \Delta w_1 x_1 + ... + \Delta w_n x_n\right) < 0,$$

and (since $w_0 + w_1 x_1 + w_n x_n \geq 0$ because $y = 1$)

$$\begin{aligned} &\left(\Delta w_0 + \Delta w_1 x_1 + ... + \Delta w_n x_n\right) < 0, \\ &\left|w_0 + w_1 x_1 + ... + w_n x_n\right| \leq \left|\Delta w_0 + \Delta w_1 x_1 + ... + \Delta w_n x_n\right|. \end{aligned} \tag{1.16}$$

Let us again use (1.6) and (1.13) to adjust the weights. We again obtain (1.14). Since $\alpha > 0, \delta = -2 < 0$, then $\alpha\delta(n+1) < 0$ and the 1st inequality from (1.16) holds. However, it is always possible to find such learning rate $\alpha > 0$ that the 2nd inequality from (1.16) also holds. This means that for the second case in (1.10) the learning rule based on (1.6) and (1.13) guarantees that (1.15) is true and the neuron produces the correct result after the weights are adjusted. Since for both cases in (1.10) the learning rule based on (1.6) and (1.13) works, then this rule always leads to the desired neuron output after the weights are corrected. We can merge (1.6) and (1.13) into

$$\begin{aligned} &\tilde{w}_0 = w_0 + \alpha\delta; \\ &\tilde{w}_i = w_i + \alpha\delta x_i, i = 1, ..., n, \end{aligned} \tag{1.17}$$

where δ is the error calculated according to (1.5) and $\alpha > 0$ is a learning rate. Equations (1.17) present the *error-correction learning rule*. After the weights are corrected according to (1.17), we obtain for the updated weighted sum the following expression

$$\begin{aligned} &\tilde{z} = \tilde{w}_0 + \tilde{w}_1 x_1 + ... + \tilde{w}_n x_n = \\ &(w_0 + \alpha\delta) + (w_1 + \alpha\delta x_1) x_1 + ... + (w_1 + \alpha\delta x_n) x_n = \\ &\underbrace{w_0 + w_1 x_1 + ... + w_n x_n}_{z} + \alpha\delta(n+1) = z + \alpha\delta(n+1). \end{aligned} \tag{1.18}$$

Since as we saw, δ in (1.18) has a sign, which is always opposite to the one of z, then it is always possible to choose $\alpha > 0$ such that $\operatorname{sgn}(\tilde{z}) = -\operatorname{sgn}(z)$. If $x_i \in T, i = 1, ..., n$, where $T \subset \mathbb{R}$ and $f\left(x_1, ..., x_n\right) : T^n \to E_2$, then instead of (1.18) we obtain

$$\begin{aligned} &\tilde{z} = \tilde{w}_0 + \tilde{w}_1 x_1 + ... + \tilde{w}_n x_n = \\ &(w_0 + \alpha\delta) + (w_1 + \alpha\delta x_1) x_1 + ... + (w_1 + \alpha\delta x_n) x_n = \\ &\underbrace{w_0 + w_1 x_1 + ... + w_n x_n}_{z} + \alpha\delta\left(1 + x_1^2 + ... + x_n^2\right) = \\ &z + \alpha\delta\left(1 + x_1^2 + ... + x_n^2\right). \end{aligned} \tag{1.19}$$

Like in (1.18), δ in (1.19) has a sign, which is always opposite to the one of z. Since $\alpha > 0$ and $1 + x_1^2 + ... + x_n^2 > 0$, it is again possible to choose α such that (1.19) holds. However, it is necessary to be more careful choosing α here than for the binary input case. While in (1.18) α does not depend on the inputs, in (1.19) it does. We will consider later, In Section 3.3, the error-correction learning rule for the multi-valued neuron and we will see that this problem exists there neither for the discrete multiple-valued inputs/output nor for the continuous ones.

1.2.3 Learning Algorithm

Definition 1.2. A *learning algorithm* is the iterative process of the adjustments of the weights using a learning rule. Suppose we need to learn some learning set containing N learning samples $\left(x_1^i, ..., x_n^i\right) \to d_i, i = 1, ..., N$. One iteration of the learning process consists of the consecutive checking for all learning samples whether (1.2) holds for the current learning sample. If so, the next learning sample should be checked. If not, the weights should be adjusted according to a learning rule. The initial weights can be chosen randomly. This process should continue either until (1.2) holds for all the learning samples or until some additional criterion is satisfied.

When the learning process is successfully finished, we say that it has *converged* or converged to a weighting vector. Thus, *convergence* of the learning process means its successful completion. No-convergence means that the corresponding input/output mapping cannot be learned.

A *learning iteration* (*learning epoch*) is a pass over all the learning samples $\left(x_1^i, ..., x_n^i\right) \to d_i, i = 1, ..., N$.

If the learning process continues until (1.2) holds for all the learning samples, we say that the learning process converges with the zero error. If errors for some learning samples are acceptable, as it was mentioned, some additional criterion for stopping the learning process should be used. The most popular additional criterion is the mean square error/root mean square error criterion. In this case, the learning process continues until either of this errors drops below some predetermined acceptable threshold value. This works as follows. Let $\delta_i, i = 1, ..., N$ be the error for the *i*th learning sample. Then the mean square error (MSE) over all learning samples is

$$MSE = \frac{1}{N} \sum_{i=1}^{N} \delta_i^2 , \tag{1.20}$$

and the root mean square error (RMSE) is

$$RMSE = \sqrt{MSE} = \sqrt{\frac{1}{N} \sum_{i=1}^{N} \delta_i^2} . \tag{1.21}$$

If either of (1.20) or (1.21) is used, then a learning iteration starts from computation of MSE (RMSE). The learning process should continue until MSE (RMSE) drops below some pre-determined reasonable threshold value.

Another approach to the learning is the error minimization. In this case, the learning algorithm is considered as an optimization problem and it is reduced to the minimization of the error functional. The error is considered as the function of the weights. But in fact, the error is a composite function

$$\delta(W) = \delta(\varphi(z)) = d - \varphi(z) = d - (w_0 + w_1 x_1 + ... + w_n x_n),$$

where $\varphi(z)$ is the activation function. Actually, this approach is the most popular, but since minimization of the error functional using optimization methods requires differentiability of an activation function, it cannot be applied to the threshold neuron whose activation function $\mathrm{sgn}(z)$ is not differentiable. It is widely used for sigmoidal neurons and neural networks based on them. We will observe them in Section 1.3.

Now we have to discuss the convergence of the learning algorithm for a single threshold neuron based on the error-correction rule (1.17). The first proof of the *perceptron convergence theorem* was given by F. Rosenblatt in [10]. It is important to mention that F. Rosenblatt considered only binary inputs. In its most comprehensive form, this convergence theorem states that if the given input/output mapping can be learned and learning samples appear in an arbitrary order, but with a condition that each of them is repeated in the *learning sequence* within some finite time interval, then the learning process converges starting from an arbitrary weighting vector after a finite number of iterations.

This theorem, however, did not clarify the question which input/output mappings can be learned using the perceptron and which cannot.

A more general case of this theorem was considered by A. Novikoff in [12]. He introduced a notion of a *linearly separable set*. The learning set $(x_1^i, ..., x_n^i) \to d_i, i = 1, ..., N; x_j^i \in T \subset \mathbb{R}, j = 1, ..., n; i = 1, ..., N$ is called linearly separable if there exist a positive constant s and a weighting vector W such that the following condition holds

$$d_i (w_0 + w_1 x_1 + ... + w_n x_n) > s, i = 1, ..., N.$$

This means that the weighted sum multiplied by the desired output must be greater than some positive constant for all the learning samples. *Novikoff's convergence theorem states that the learning algorithm converges after a finite number of iterations if the learning set is linearly separable*. The idea behind the Novikoff's proof is to show that the assumption that the learning process does not converge after a finite number of iterations contradicts to the linear separability of the learning set. Novikoff showed that the amount of changes to the initial weighting vector is bounded by $(2M / s)^2$, where M is the maximum norm of an input vector.

Since the norm is always a finite non-negative real number, the number of iterations in the learning algorithm is also finite.

We will see later that this approach used by Novikoff to prove the convergence of the error-correction learning algorithm for the threshold neuron also works to prove the convergence of the learning algorithm for the multi-valued neuron (Section 3.3) and a multilayer neural network based on multi-valued neurons (Chapter 4).

1.2.4 Examples of Application of the Learning Algorithm Based on the Error-Correction Rule

Let us consider how learning rule (1.17) can be used to train the threshold neuron using the learning algorithm, which was just defined.

Example 1.2. Let us consider again the OR problem $f(x_1, x_2) = x_1 \text{ or } x_2$ (Table 1.1, Fig. 1.4a), which we have already considered above. Our learning set contains four learning samples (see Table 1.1). Let us start the learning process from the weighting vector $W = (1,1,1)$.

Iteration 1.

1) Inputs (1, 1). The weighted sum is equal to $z = 1 + 1 \cdot 1 + 1 \cdot 1 = 3$; $\varphi(z) = \text{sgn}(z) = \text{sgn}(3) = 1$. Since $f(1,1) = 1$, no further correction of the weights is needed.

2) Inputs (1, -1). The weighted sum is equal to $z = 1 + 1 \cdot 1 + 1 \cdot (-1) = 1$; $\varphi(z) = \text{sgn}(z) = \text{sgn}(1) = 1$. Since $f(1,-1) = -1$, we have to correct the weights. According to (1.5) $\delta = -1 - 1 = -2$. Let $\alpha = 1$ in (1.17). Then we have to correct the weights according to (1.17):

$$\tilde{w}_0 = 1 - 2 = -1;\ \tilde{w}_1 = 1 + (-2) \cdot 1 = -1;\ \tilde{w}_2 = 1 + (-2) \cdot (-1) = 3.$$

Thus, $\tilde{W} = (-1,-1,3)$.

The weighted sum after the correction is equal to $z = -1 + (-1) \cdot 1 + 3 \cdot (-1) = -5$; $\varphi(z) = \text{sgn}(z) = \text{sgn}(-5) = -1$. Since $f(1,-1) = -1$, no further correction of the weights is needed.

3) Inputs (-1, 1). The weighted sum is equal to $z = -1 + (-1) \cdot (-1) + 3 \cdot 1 = 3$; $\varphi(z) = \text{sgn}(z) = \text{sgn}(3) = 1$. Since $f(-1,1) = -1$, we have to correct the weights. According to (17) $\delta = -1 - 1 = -2$. Let $\alpha = 1$ in (1.17). Then we have to correct the weights according to (1.17):

$$\tilde{w}_0 = -1-2 = -3;\ \tilde{w}_1 = -1+(-2)\cdot(-1) = 1;\ \tilde{w}_2 = 3+(-2)\cdot 1 = 1.$$

Thus, $\tilde{W} = (-3,1,1)$.

The weighted sum after the correction is equal to $z = -3+1\cdot(-1)+1\cdot 1 = -3$; $\varphi(z) = \mathrm{sgn}(z) = \mathrm{sgn}(-3) = -1$. Since $f(-1,1) = -1$, no further correction of the weights is needed.

4) Inputs (-1, -1). The weighted sum is equal to $z = -3+1\cdot(-1)+1\cdot(-1) = -5$; $\varphi(z) = \mathrm{sgn}(z) = \mathrm{sgn}(-5) = -1$. Since $f(-1,-1) = -1$, no further correction of the weights is needed.

Iteration 2.

1) Inputs (1, 1). The weighted sum is equal to $z = -3+1\cdot 1+1\cdot 1 = -1$; $\varphi(z) = \mathrm{sgn}(z) = \mathrm{sgn}(-1) = -1$. Since $f(1,1) = 1$, we have to correct the weights. According to (1.17) $\delta = 1-(-1) = 2$. Let $\alpha = 1$ in (1.17). Then we have to correct the weights according to (1.17):

$$\tilde{w}_0 = -3+2 = -1;\ \tilde{w}_1 = 1+2\cdot 1 = 3;\ \tilde{w}_2 = 1+2\cdot 1 = 3.$$

Thus, $\tilde{W} = (-1,3,3)$.

The weighted sum after the correction is equal to $z = -1+3\cdot 1+3\cdot 1 = 5$; $\varphi(z) = \mathrm{sgn}(z) = \mathrm{sgn}(5) = 1$. Since $f(1,1) = 1$, no further correction of the weights is needed.

2) Inputs (1, -1). The weighted sum is equal to $z = -1+3\cdot 1+3\cdot(-1) = -1$; $\varphi(z) = \mathrm{sgn}(z) = \mathrm{sgn}(-1) = -1$. Since $f(1,-1) = -1$, no further correction of the weights is needed.

3) Inputs (-1, 1). The weighted sum is equal to $z = -1+3\cdot(-1)+3\cdot 1 = -1$; $\varphi(z) = \mathrm{sgn}(z) = \mathrm{sgn}(-1) = -1$. Since $f(-1,1) = -1$, no further correction of the weights is needed.

4) Inputs (-1, -1). The weighted sum is equal to $z = -1+3\cdot(-1)+3\cdot(-1) = -7$; $\varphi(z) = \mathrm{sgn}(z) = \mathrm{sgn}(-7) = -1$. Since $f(-1,-1) = -1$, no further correction of the weights is needed.

This means that the iterative process converged after two iterations, there are no errors for all the samples from the learning set, and this learning set presented by the OR function $f(x_1, x_2) = x_1$ or x_2 of the two variables is learned. Therefore, the OR function can be implemented with the threshold neuron using the weighting vector $\tilde{W} = (-1,3,3)$ obtained as the result of the learning process.

Table 1.4 Learning process for the function $f(x_1, x_2) = \overline{x}_1 \,\&\, x_2$

x_1	x_2	$z = w_0 + w_1 x_1 + w_2 x_2$	$\mathrm{sgn}(z)$	$f(x_1, x_2) = \overline{x}_1 \,\&\, x_2$	δ
$W = (1,1,1)$					
Iteration 1					
1	1	$z = 1+1\cdot 1+1\cdot 1 = 3$	1	1	0
1	-1	$z = 1+1\cdot 1+1\cdot(-1) = 1$	1	-1	-2
$\tilde{w}_0 = 1-2 = -1;\ \tilde{w}_1 = 1+(-2)\cdot 1 = -1;\ \tilde{w}_2 = 1+(-2)\cdot(-1) = 3$ $W = (-1,-1,3)$					
-1	1	$z = -1+(-1)\cdot(-1)+3\cdot 1 = 3$	1	1	0
-1	-1	$z = -1+(-1)\cdot(-1)+3\cdot(-1) = -3$	-1	1	2
$\tilde{w}_0 = -1+2 = 1;\ \tilde{w}_1 = -1+2\cdot(-1) = -3;\ \tilde{w}_2 = -1+2\cdot(-1) = -3$ $\tilde{W} = (1,-3,-3)$					
Iteration 2					
1	1	$z = 1+(-3)\cdot 1+(-3)\cdot 1 = -5$	-1	1	2
$\tilde{w}_0 = 1+2 = 3;\ \tilde{w}_1 = -3+2\cdot 1 = -1;\ \tilde{w}_2 = -3+2\cdot 1 = -1$ $\tilde{W} = (3,-1,-1)$					
1	-1	$z = 3+(-1)\cdot 1+(-1)\cdot(-1) = 3$	1	-1	-2
$\tilde{w}_0 = 3-2 = 1;\ \tilde{w}_1 = -1+(-2)\cdot 1 = -3;\ \tilde{w}_2 = -1+(-2)\cdot(-1) = 3$ $\tilde{W} = (1,-3,3)$					
-1	1	$z = 1+(-3)\cdot(-1)+3\cdot 1 = 7$	1	1	0
-1	-1	$z = 1+(-3)\cdot(-1)+3\cdot(-1) = 1$	1	1	0
Iteration 3					
1	1	$z = 1+(-3)\cdot 1+3\cdot 1 = 1$	1	1	0
1	-1	$z = 1+(-3)\cdot 1+3\cdot(-1) = -5$	-1	-1	0
-1	1	$z = 1+(-3)\cdot(-1)+3\cdot 1 = 7$	1	1	0
-1	-1	$z = 1+(-3)\cdot(-1)+3\cdot(-1) = 1$	1	1	0

Example 1.3. Let us learn the function $f(x_1, x_2) = \overline{x}_1 \,\&\, x_2$ ($\overline{x}$ means the negation of the Boolean variable x, in the alphabet $\{1, -1\}$ $\overline{x} = -x$) using the threshold neuron.

#	x_1	x_2	$\overline{x}_1 \,\&\, x_2$
1)	1	1	1
2)	1	-1	-1
3)	-1	1	1
4)	-1	-1	1

Table 1.4 shows the function values and the entire learning set containing four input vectors and four values of the function, respectively. We start the learning process from the same weighting vector $W = (1,1,1)$ as in Example 1.2. We hope that so detailed explanations as were

given in Example 1.2 will not be needed now. Thus, the iterative process has converged after three iterations, there are no errors for all the elements from the learning set, and our input/output mapping described by the Boolean function $f(x_1, x_2) = \overline{x}_1 \,\&\, x_2$ is implemented on the threshold neuron using the weighting vector $\widetilde{W} = (1,-3,3)$ obtained as the result of the learning process.

1.2.5 Limitation of the Perceptron. Minsky's and Papert's Work

In 1969, M. Minsky and S. Papert published their famous book [13] in which they proved that the perceptron cannot learn non-linearly separable input/output mappings. Particularly, they showed that, for example the XOR problem is unsolvable using the perceptron. Probably from that time the XOR problem is a favorite problem, which is used to demonstrate why we need multilayer neural networks - to learn such problems as XOR. This resulted in a significant decline in interest to neurons and neural networks in 1970s.

We will show later that this problem is the simplest possible problem, which can be solved by a single multi-valued neuron with a periodic activation function (Section 1.4 and Chapter 5).

Thus, a principal limitation of the perceptron is its impossibility to learn non-linearly separable input/output mappings. This limitation causes significant lack of the functionality and reduces a potential area of applications because the most of real-world pattern recognition and classification problems are non-linearly separable.

The next significant limitation of the perceptron is its binary output. Thus, the perceptron can be used neither for solving multi-class classification problems (where the number of classes to be classified is greater than two) nor problems with a continuous output.

In this book, starting from Section 1.4 and thereafter we will show how these limitations can easily be overcome with complex-valued neurons. Non-linearly separable binary problems and multiple-valued problems (including the nonlinearly-separable ones) can be learned using a single multi-valued neuron.

But first, to conclude our observation of neurons and neural networks, let us consider the most popular topologies of neural networks, which were proposed in 1980s and which are now successfully used in complex-valued neural networks.

1.3 Neural Networks: Popular Topologies

1.3.1 XOR Problem: Solution Using a Feedforward Neural Network

As we have seen, the perceptron cannot learn non-linearly separable problems (input/output mappings). In the third part of his book [11], F. Rosenblatt proposed an idea of multilayer perceptron containing more than one layer of *A*-units (see Fig. 1.6). He projected that this neural network will be more functional and will be able to learn non-linearly separable problems. However, no learning algorithm for this network was proposed that time. In [13], M. Minsky and S. Papert presented their skeptical view on the "multilayer perceptron". They did not hope that it will be more efficient than the classical single layer perceptron, probably because there was still no learning algorithm for a multilayer neural network.

However, the existence of non-linearly separable problems was a great stimulus to develop new solutions. We will see starting from Section 1.4 how easily many of them can be solved using the multi-valued neuron. But first let us again take a historical view. A two-layer neural network containing three neurons in total, which can solve the XOR problem, is described, for example, in [2], where the paper [14] is cited as a source of this solution. We are not sure that this solution was presented for the first time definitely in [14]; most probably it was known earlier. It is quite difficult to discover today who found this solution first. Nevertheless, let us consider it here.

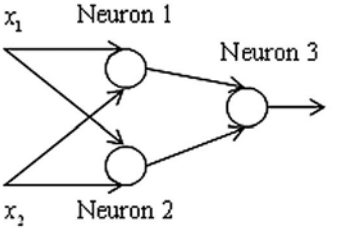
To solve the XOR problem within a "threshold basis" (using the threshold neuron), it is necessary to build a network from threshold neurons. Let us consider a network from three neurons (see Fig. 1.7a). This network contains the input layer, which distributes the input signals x_1 and x_2, one hidden layer containing Neurons 1 and 2 and one output layer containing a single Neuron 3. This is the simplest possible non-trivial *multilayer feedforward neural network* (MLF). It is the simplest possible network because it contains a minimum amount of layers and neurons to be non-trivial (two layers including one hidden layer and one output layer, two neurons in the hidden layer, and one neuron in the output layer). A network is trivial if it contains just a single hidden neuron and a single output neuron. This network is called *feedforward* because there are no feedback connections there, all signals are transmitted through the network in a strictly feedforward manner.

Let us remind that the function XOR may be presented in the full disjunctive normal form as follows:

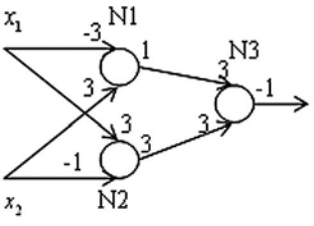
$$x_1 \oplus x_2 = x_1\overline{x}_2 \vee \overline{x}_1 x_2 = f_1(x_1, x_2) \vee f_2(x_1, x_2),$$

where $f_2(x_1, x_2) = \overline{x}_1 x_2$ is that function whose learning and implementation using the threshold neuron was considered in Example 1.3. Let us also remind that learning and implementation of the OR function, which connects functions

$f_1(x_1, x_2)$ and $f_2(x_1, x_2)$ was considered in Example 1.2. Notice that function $f_1(x_1, x_2) = x_1\overline{x}_2$ may be obtained by changing the order of variables in function $f_2(x_1, x_2) = \overline{x}_1 x_2$. It was shown in [6] that if some Boolean function is threshold, than any function obtained from the first one by the permutation of its variables is also threshold and its weighting vector can be obtained by the permutation of the weights in the weighting vector of the first function corresponding to the permutation of the variables.



(a) a two layer neural network with two inputs, with one hidden layer containing two neurons, and the output layer containing a single neuron



(b) a two layer neural network with two inputs, with one hidden layer containing two neurons, and the output layer containing a single neuron. The weights that solve the XOR problem are assigned to the neurons

Fig. 1.7 Simple neural networks

The weight w_0 remains unchanged. Therefore a weighting vector W_{f_1} for $f_1(x_1, x_2) = x_1\overline{x}_2$ may be obtained from the one for $f_2(x_1, x_2) = \overline{x}_1 x_2$ by reordering the weights w_1 and w_2. Since, as we found in Example 1.3 for function $f_2(x_1, x_2) = \overline{x}_1 x_2$, $W_{f_2} = (1, -3, 3)$, the weighting vector $W_{f_1} = (1, 3, -3)$ implements function $f_2(x_1, x_2) = \overline{x}_1 x_2$ using a single threshold neuron. It is easy to check that this weighting vector gives a correct realization of the function.

This means that if Neuron 1 implements function $f_1(x_1, x_2)$, Neuron 2 implements function $f_2(x_1, x_2)$, and Neuron 3 implements the OR function, then the network presented in Fig. 1.7b implements the XOR function. Let us consider how it works. Thus, Neuron 1 operates with the weighting vector $\tilde{W} = (1, 3, -3)$, Neuron 2 operates with the weighting vector $\tilde{W} = (1, -3, 3)$, and Neuron 3 operates with the weighting vector $\tilde{W} = (-1, 3, 3)$ (see Fig. 1.7b). The network works in the following way. There are no neurons in the input layer. It just distributes the

input signals among the hidden layer neurons. The input signals x_1 and x_2 are accepted in parallel from the input layer by both neurons from the hidden layer (N1 and N2). Their outputs are coming to the corresponding inputs of the single neuron in output layer (N3). The output of this neuron is the output of the entire network. The results are summarized in Table 1.5 (z is the weighted sum of the inputs). For all three neurons their weighted sums and outputs are shown. To be convinced that the network implements definitely the XOR function, its actual values are shown in the last column of Table 1.5.

Table 1.5 Implementation of the XOR function using a neural network presented in Fig. 1.7b

Inputs		**Neuron 1** $\tilde{W} = (1,3,-3)$		**Neuron 2** $\tilde{W} = (1,-3,3)$		**Neuron 3** $\tilde{W} = (-1,3,3)$		x_1 XOR x_2
x_1	x_2	Z	sgn(z) **output**	Z	sgn(z) **output**	Z	sgn(z) **output**	
1	**1**	1	**1**	1	**1**	5	**1**	**1**
1	**-1**	7	**1**	-5	**-1**	-1	**-1**	**-1**
-1	**1**	-5	**-1**	7	**1**	-1	**-1**	**-1**
-1	**-1**	1	**1**	1	**1**	5	**1**	**1**

1.3.2 Popular Real-Valued Activation Functions

As we see, a multilayer feedforward neural network (MLF) has a higher functionality compared to the perceptron. It can implement non-linearly separable input/output mappings, while the perceptron cannot. Considering in the previous section how MLF may solve the XOR problem, we have not passed this problem through a learning algorithm; we just have synthesized the solution. However, the most wonderful property of MLF is its learning algorithm. MLF was first proposed in [15] by D.E. Rumelhart, G.E. Hilton, and R.J. Williams. They also described in the same paper the backpropagation learning algorithm. It is important to mention that a seminal idea behind the error backpropagation and its use to train a feedforward neural network belongs to Paul Werbos. He developed these ideas in his Harvard Ph. D. dissertation in 1974 and later he included it as a part in his book [16] (Chapters 1-6).

It is also important to mention that starting from mid 1980s, especially from the moment when D. Rumelhart and his co-authors introduced MLF, threshold neurons as basic neurons for building neural networks have moved to the background. It became much more interesting to learn and implement using neural networks continuous and multi-valued input/output mappings described by functions $f(x_1,...,x_n): T^n \to T, T \subset \mathbb{R}$, which was impossible using a hard-limited threshold activation function $\mathrm{sgn}(z)$.

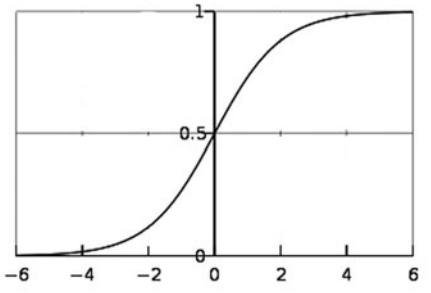


Fig. 1.8 Logistic function

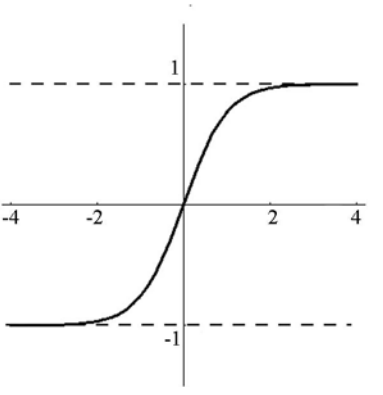


Fig. 1.9 tanh function

Typically, there have been considered $T=[0,1]$ or $T=[-1,1]$. Respectively, new activation functions became very popular from mid 1980s. The most popular of them is a *sigmoid activation function*. It has two forms – the logistic function and the hyperbolic tangent function. Logistic function is as follows

$$\varphi(z)=\frac{1}{1+e^{-\alpha z}}, \tag{1.22}$$

(see Fig. 1.8), where α is a slope parameter. The curve in Fig. 1.8 got its name "sigmoid" from Pierre François Verhulst (in 1844 or 1845) who studied the population growth described by (1.22). Evidently, the range of function (1.22) is $]0,1[$, the function approaches 0 when $z\to-\infty$ and approaches 1 when $z\to\infty$ (actually, the logistic function approaches its bounds with significantly smaller values of its argument as it is seen from Fig. 1.8). To obtain a sigmoid curve with the range $]-1,1[$, the hyperbolic tangent function

$$\tanh\alpha z=\frac{\sinh\alpha z}{\cosh\alpha z}=\frac{e^{\alpha z}-e^{-\alpha z}}{e^{\alpha z}+e^{-\alpha z}}, \tag{1.23}$$

should be used. The shape of function (1.23) is identical to the one of function (1.22) (see Fig. 1.9) with only distinction that the $\tanh$ function cross not the line y=0.5, but the horizontal axis at the origin and it is bounded from the bottom by the line y= -1. α in (1.23) is again a slope parameter and its role is identical to the one in (1.22). It is clear that if $\alpha\to\infty$ in (1.23), then $\tanh\alpha z$ approaches $\mathrm{sgn}(z)$ (compare Fig. 1.2 and Fig. 1.9).

Why definitely sigmoid activation functions (1.22) and (1.23) became so popular? There are at least two reasons. The first reason is that on the one hand, they easily limit the range of the neuron output, but on the other hand, they drastically increase the neuron's functionality making it possible to learn continuous and multiple-valued discrete input/output mappings. Secondly, they are increasing (we will see a little bit later that this is important to develop a computational model for approximation, which follows from the Kolmogorov's theorem [17]). Their specific nonlinearity can be used for approximation of other highly nonlinear functions. Finally, they are differentiable (which is important for the learning purposes; as it is well known and as we will see, the differentiability is critical for that backpropagation learning technique developed in [15, 16]. We will also see later (Chapter 4) that it will not be needed for the backpropagation learning algorithm for a feedforward network based on multi-valued neurons).

Another popular type of an activation function, which is used in real-valued neurons and neural networks, is radial basis function (RBF) first introduced by M.J.D. Powell [18] in 1985. RBF is a real-valued function whose value depends only on the distance from the origin $\varphi(z) = \varphi(\|z\|)$ or on the distance from some pre-determined other point c, called a *center*, so that $\varphi(z,c) = \varphi(\|z-c\|)$, where $\|\ \|$ is the norm in the corresponding space. There are different functions that satisfy this property. Perhaps, the most popular of them, which is used in neural networks and machine learning is the Gaussian RBF $\varphi(r) = e^{-\left(\frac{r}{\alpha}\right)^2}$ [2, 5], where $r = z - c$ (c is the corresponding center), $\alpha > 0$ is a parameter.

1.3.3 Multilayer Feedforward Neural Network (MLF) and Its Backpropagation Learning

Let us consider in more detail a network with perhaps the most popular topology, namely a multilayer feedforward neural network (MLF), also widely referred to as a multilayer perceptron (MLP) [15, 2]. We will also consider the basic principles of the backpropagation learning algorithm for this network.

Typically, an MLF consists of a set of sensory units (source nodes – analogues of *S*-units in the perceptron) that constitute the *input* layer (which distributes input signals among the first hidden layer neurons), one or more *hidden* layers of neurons, and an *output* layer of neurons. We have already considered a simple example of such a network, which solves the XOR problem. The input signals progresses through the network in a *forward* direction, on a layer-by-layer basis. An important property of MLF is its full connection architecture: the outputs of *all* neurons in a specified layer are connected to the corresponding inputs of *all* neurons of the following layer (for example, the output of a neuron *ij* (the *i*th neuron from the *j*th layer) is connected to the *i*th input of all neurons from the j+1st layer).

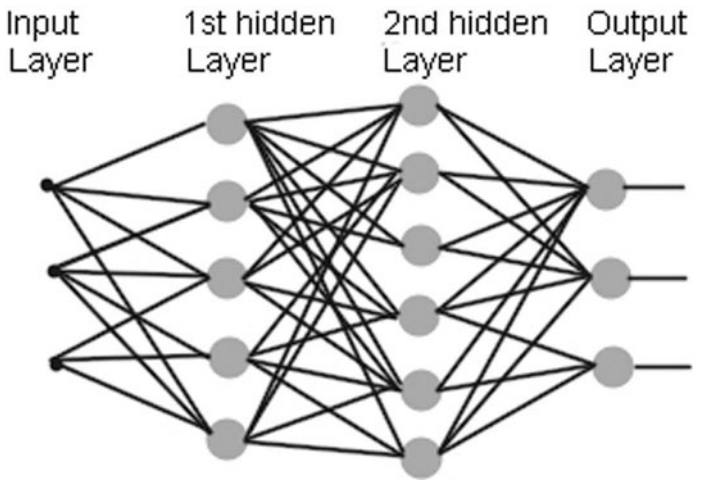


Fig. 1.10 3-5-6-3 MLF – Multilayer Feedforward Neural Network. It has 3 inputs, the 1st hidden layer containing 5 neurons, the 2nd hidden layer containing 6 neurons and the output layer containing 3 neurons

This means a full connection between consecutive layers (see Fig. 1.10). To specify a network topology, the notation $n-n_1-...-n_i-...-n_s-n_o$ is used. Here n is the number of network inputs, $n_i, i=1,...,s$ is the number of neurons in the ith hidden layer, s is the number of hidden layers, and n_o is the number of neurons in the output layer.

This architecture is the result of a "universal approximator" computing model based on the famous Kolmogorov's Theorem [17]. This theorem states the following. There exist fixed (universal) increasing continuous functions $h_{ij}(x)$ on $I=[0,1]$ such that each continuous function of n variables $f(x_1,...,x_n)$ on I^n can be written in the form

$$f(x_1,...,x_n)=\sum_{j=1}^{2n+1} g_j\left(\sum_{i=1}^{n} h_i(x_i)\right), \tag{1.24}$$

where $g_j, j=1,...,n$ are some properly chosen continuous functions of one variable.

This result states that any multivariate continuous function can be represented by the superposition of a small number of univariate continuous functions. It is clear that in terms of feedforward neural networks equation (1.24) describes a three layer feedforward neural network whose first two layers contain n and $2n+1$

neurons, respectively, and implement functions $h_i, i = 1,...,n$ and $g_j, j = 1,...,2n+1$, respectively. The output layer of this network contains a single neuron with the linear activation function (its output is equal to the weighted sum; according to (1.20) all weights of the output neuron are equal to 1 except $w_0 = 0$, there are no weighting coefficients in a front of $g_j, j = 1,...,2n+1$). It is well known that a multilayer feedforward neural network is a universal approximator (for the first time this was clearly proven in [19] and [20]).

However, the Kolmogorov's Theorem, being very important, is a typical "existence theorem". It justifies only the existence of the solution. It does not show a mechanism for finding functions $h_i, i = 1,...,n$ and $g_j, j = 1,...,2n+1$. To approach that solution, which exists according to the Kolmogorov's Theorem, a feedforward neural network has to learn that function $f(x_1,...,x_n)$, which we want to approximate. To implement the learning process, the backpropagation learning algorithm was suggested. A problem, which is necessary to solve, implementing the learning process for a feedforward neural network, is finding the hidden neurons errors. While the exact errors of output neurons can be easily calculated as the differences between the desired and actual outputs, for all the hidden neurons their desired outputs are unknown and therefore there is no straightforward way to calculate their errors. But without the errors it is not possible to adjust the weights.

The basic idea behind a backpropagation learning algorithm is sequential propagation of the errors of the neurons from the output layer through all the layers from the "right hand" side to the "left hand" side up to the first hidden layer (see Fig. 1.10), in order to calculate the errors of all other neurons. The heuristic idea is to share the errors of output neurons, which can be calculated because their desired outputs are known (unlike the ones of the hidden neurons), with all the hidden neurons.

Basically, the entire learning process consists of two passes through all the different layers of the network: a forward pass and a backward pass. In the forward pass, the inputs are propagated from the input layer of the network to the first hidden layer and then, layer by layer, output signals from the hidden neurons are propagated to the corresponding inputs of the following layer neurons. Finally, a set of outputs is produced as the actual response of the network. Evidently, during the forward pass the synaptic weights of the network are all fixed. During the backward pass first the errors of all the neurons are calculated and then the weights of all the neurons are all adjusted in accordance with the learning rule. One complete iteration (epoch) of the learning process consists of a forward pass and a backward pass.

Although the error backpropagation algorithm for MLF is well known, we would like to include its derivation here. In our opinion, this is important for the following two reasons. The first reason is to simplify perception of this book for

those readers (first of all for students) who are not the experts in neural networks and just make their first steps in this area. The second and even more important reason is to compare this derivation and the backpropagation learning algorithm for MLF with the ones for a complex-valued multilayer feedforward neural network based on multi-valued neurons, which will be considered in Chapter 4. This comparison will be very important for understanding of significant advantages of complex-valued neural networks.

In the derivation of the MLF backpropagation learning algorithm we mostly will follow here [2] and [5].

It is important to mention that the backpropagation learning algorithm is based on the generalization of the error-correction learning rule for the case of MLF. Specifically, the actual response of the network is subtracted from a desired response to produce an error signal. This error signal is then propagated backward through the network, against the direction of synaptic connections – hence the name "*backpropagation*". The weights are adjusted so as to make the actual output of the network move closer to the desired output. A common property of a major part of real-valued feedforward neural networks is the use of sigmoid activation functions for its neurons. Let us use namely logistic function (1.22).

Let us consider a multilayer neural network with traditional feedforward architecture (see Fig. 1.10), when the outputs of neurons of the input and hidden layers are connected to the corresponding inputs of the neurons from the following layer. Let us suppose that the network contains one input layer, *m*-1 hidden layers and one output layer. We will use here the following notations.

Let

D_{km} - be a desired output of the *k*th neuron from the output (*m*th) layer

Y_{km} - be the actual output of the *k*th neuron from the output (*m*th) layer.

Then a global error of the network related to the *k*th neuron of the output (*m*th) layer can be calculated as follows:

$$\delta^*_{km} = D_{km} - Y_{km} \text{ - error for the } k\text{th neuron from output } (m\text{th}) \text{ layer.} \qquad (1.25)$$

δ^*_{km} denotes here and further a global error of the network. We have to distinguish it from the local errors δ_{km} of the particular output neurons because each output neuron contributes to the global error equally with the hidden neurons.

The learning algorithm for the classical MLF is derived from the consideration that the global error of the network in terms of the *mean square error* (MSE) must be minimized. The functional of the error may be defined as follows:

$$E = \frac{1}{N}\sum_{s=1}^{N} E_s \,, \qquad (1.26)$$

where E denotes *MSE*, *N* is the total number of samples (patterns) in the learning set and E_s denotes the square error of the network for the *s*th pattern;

$E_s = \left(D_s - Y_s\right)^2 = \left(\delta_s^*\right)^2, s = 1,...,N$ for a single output neuron and $E_s = \frac{1}{N_m}\sum_{k=1}^{N_m}\left(D_{k_s} - Y_{k_s}\right)^2 = \frac{1}{N_m}\sum_{k=1}^{N_m}\left(\delta_{k_s}^*\right)^2; s = 1,...,N$ for N_m output neurons. For simplicity, but without loss of generality, we can consider minimization of a *square error* (*SE*) function instead of minimization the MSE function (1.26).

The square error is defined as follows:

$$E = \frac{1}{2}\sum_{k=1}^{N_m}(\delta_{km}^*)^2 , \tag{1.27}$$

where N_m indicates the number of output neurons,

$$\delta_{km}^* = D_{k_s} - Y_{k_s}, s = 1,...,N , \tag{1.28}$$

m is the output layer index, and the factor $\frac{1}{2}$ is used so as to simplify subsequent derivations resulting from the minimization of E. The error function (1.27) is a function of the weights. Indeed, it strictly depends on all the network weights. It is a principal assumption that the error depends not only on the weights of the neurons at the output layer, but on all neurons of the network.

Thus, a problem of learning can be reduced to finding a global minimum of (1.27) as a function of weights. In these terms, this is the optimization problem.

The backpropagation is used to calculate the gradient of the error of the network with respect to the network's modifiable weights. This gradient is then used in a gradient descent algorithm to find such weights that minimize the error. Thus, the minimization of the error function (1.27) (as well, as (1.26)) is reduced to the search for those weights for all the neurons that ensure a minimal error.

To ensure movement to the global minimum on each iteration, the correction of the weights of all the neurons has to be organized in such a way that each weight w_i has to be corrected by an amount Δw_i, which must be proportional to the partial derivative $\frac{\partial E}{\partial w_i}$ of the error function *E*(*W*) with respect to the weights [2].

For the next analysis, the following notation will be used. Let w_i^{kj} denote the weight corresponding to the *i*th input of the *k*th neuron at the *j*th layer. Furthermore let z_{kj}, y_{kj} and $Y_{kj} = y_{kj}\left(z_{kj}\right)$ represent the weighted sum (of the input signals), the activation function value, and the output value of the *k*th neuron at the *j*th layer, respectively. Let N_j be the number of neurons in the *j*th layer (notice that this means that neurons of the j+1st layer have exactly N_j inputs.) Finally,

recall that $x_1,...,x_n$ denote the inputs to the network (and as such, also the inputs to the neurons of the first layer.)

Then, taking into account that $E(W)=E\left(y\left(z(W)\right)\right)$ and applying the chain rule for the differentiation, we obtain for the k^{th} neuron at the output (m^{th}) layer

$$\frac{\partial E(W)}{\partial w_i^{km}}=\frac{\partial E(W)}{\partial y_{km}}\frac{\partial y_{km}}{\partial z_{km}}\frac{\partial z_{km}}{\partial w_i^{km}},\quad i=0,1,...,N_{m-1},$$

where

$$\frac{\partial E(W)}{\partial y_{km}}=\frac{\partial}{\partial y_{km}}\left(\frac{1}{2}\sum_k(\delta_{km}^*)^2\right)=\frac{1}{2}\sum_k\frac{\partial}{\partial y_{km}}(\delta_{km}^*)^2=$$

$$=\frac{1}{2}\frac{\partial}{\partial y_{km}}(\delta_{km}^*)^2=\delta_{km}^*\frac{\partial}{\partial y_{km}}(\delta_{km}^*)=\delta_{km}^*\frac{\partial}{\partial y_{km}}\frac{1}{N}\sum_{s=1}^{N}\left(D_{km_s}-Y_{km_s}\right)=-\delta_{km}^*;$$

$$\frac{\partial y_{km}}{\partial z_{km}}=y_{km}'(z_{km}),$$

and

$$\frac{\partial z_{km}}{\partial w_i^{km}}=\frac{\partial}{\partial w_i^{km}}\left(w_0^{km}+w_1^{km}Y_{1,m-1}+...+w_{N_{m-1}}^{km}Y_{N_{m-1},m-1}\right)=Y_{i,m-1},$$

$i=0,\ 1,\ ...,\ N_{m-1}.$

Then we obtain the following:

$$\frac{\partial E(W)}{\partial w_i^{km}}=\frac{\partial E(W)}{\partial y_{km}}\frac{\partial y_{km}}{\partial z_{km}}\frac{\partial z_{km}}{\partial w_i^{km}}=-\delta_{km}^*y_{km}'\left(z_{km}\right)Y_{i,m-1},\quad i=0,\ 1,\ ...,\ N_{m-1};$$

where $Y_{0,m-1}\equiv 1$. Finally, we obtain now the following

$$\Delta w_i^{km}=-\beta\frac{\partial E(W)}{\partial w_i^{km}}=\begin{cases}\beta\delta_{km}^*y_{km}'\left(z_{km}\right)Y_{i,m-1} & i=1,...,\ N_{m-1}\\ \beta\delta_{km}^*y_{km}'\left(z_{km}\right) & i=0,\end{cases}\tag{1.29}$$

where $\beta>0$ is a learning rate.

The part of the rate of change of the square error $E(W)$ with respect to the input weight of a neuron, which is independent of the value of the corresponding input signal to that neuron, is called the *local error* (or simply the error) of that neuron. Accordingly, the local error of the kth neuron of the output layer, denoted by δ_{km}, is given by

$$\delta_{km} = y'_{km}\left(z_{km}\right)\cdot\delta^{*}_{km}; k = 1,...,N_m\,. \tag{1.30}$$

It is important that we differ local errors of output neurons presented by (1.30) from the global errors of the network presented by (1.28) and taken from the same neurons. Respectively, taking into account (1.30), we can transform (1.29) as follows:

$$\Delta w_i^{km} = -\beta\frac{\partial E(W)}{\partial w_i^{km}} = \begin{cases}\beta\delta_{km}Y_{i,m-1} & i = 1,...,\ N_{m-1}\\ \beta\delta_{km} & i = 0,\end{cases} \tag{1.31}$$

Let us now find the hidden neurons errors. To find them, we have to backpropagate the output neurons errors (1.30) to the hidden layers. To propagate the output neurons errors to the neurons of all hidden layers, a sequential error backpropagation through the network from the *m*th layer to the *m*-1st one, from the *m*-1st one to the m-2nd one, ..., from the 3rd one to the 2nd one, and from the 2nd one to the 1st one has to be done. When the error is propagated from the layer *j*+1 to the layer *j*, the local error of each neuron of the *j*+1st layer is multiplied by the weight of the path connecting the corresponding input of this neuron at the *j*+1st layer with the corresponding output of the neuron at the *j*th layer. For example, the error $\delta_{i,j+1}$ of the *i*th neuron at the *j*+1st layer is propagated to the *k*th neuron at the *j*th layer, multiplying $\delta_{i,j+1}$ with $w_k^{i,j+1}$, namely the weight corresponding to the *k*th input of the *i*th neuron at the *j*+1st layer. This analysis leads to the following expression for the error of the *k*th neuron from the *j*th layer:

$$\delta_{kj} = y'_{kj}\left(z_{kj}\right)\sum_{k=1}^{N_{j+1}}\delta_{i,j+1}w_k^{i,j+1}; k = 1,...,N_j\,. \tag{1.32}$$

It should be mentioned that equations (1.29)-(1.32) are obtained for the general case, without the connection with some specific activation function. Since we agreed above that we use a logistic function (1.22) in our MLF, a derivative of this function is the following (let us take for simplicity, but without loss of generality, $\alpha = 1$ in (1.22)):

$$y'(z) = \varphi'(z) = \left(\frac{1}{1+e^{-z}}\right)' = \left((1+e^{-z})^{-1}\right)' = -(1+e^{-z})^{-2}\cdot(-e^{-z}) =$$

$$= \frac{e^{-z}}{(1+e^{-z})(1+e^{-z})} = y(z)\frac{e^{-z}}{(1+e^{-z})} = y(z)\left(1-y(z)\right)$$

because

$$1-y(z) = 1-\frac{1}{1+e^{-z}} = \frac{1+e^{-z}-1}{1+e^{-z}} = \frac{e^{-z}}{1+e^{-z}}\,.$$

Thus $y'(z) = \varphi'(z) = y(z)\big(1 - y(z)\big)$ and substituting this to (1.32) we obtain the equation for the error of the MLF hidden neurons (the *k*th neuron from the *j*th layer) with the *logistic activation function*:

$$\delta_{kj} = y_{kj}\left(z_{kj}\right) \cdot \big(1 - y_{kj}(z_{kj})\big) \sum_{i=1}^{N_{j+1}} \delta_{i,j+1} w_k^{i,j+1}; k = 1,..., N_j . \quad (1.33)$$

Once all the errors are known, (1.30) determine the output neurons errors and (1.33) determine the hidden neurons errors, and all the weights can be easily adjusted by adding the adjusting term Δw to the corresponding weight. For the output neurons this term was already derived and it is shown in (1.31). For all the hidden neurons it can be derived in the same way and it is equal for the first hidden layer neurons to

$$\Delta w_i^{k1} = -\beta \frac{\partial E(W)}{\partial w_i^{k1}} = \begin{cases} \beta \delta_{k1} x_i, & i = 1,..., n \\ \beta \delta_{k1} & i = 0, \end{cases} \quad (1.34)$$

where $x_1,..., x_n$ are the network inputs, n is the number of them, and β is a learning rate. For the rest of hidden neurons

$$\Delta w_i^{kj} = -\beta \frac{\partial E(W)}{\partial w_i^{kj}} = \begin{cases} \beta \delta_{kj} Y_{i,j-1}, & i = 1,..., N_{m-1} \\ \beta \delta_{kj}, & i = 0, \end{cases} \quad (1.35)$$

$$j = 2,..., m-1.$$

All the network weights can now be adjusted taking into account (1.31), (1.34) and (1.35) as follows (we consider $N_0 = n$ - the number of "neurons" in the first layer is equal to the number of network inputs, there are *m*-1 hidden layers in the network and the *m*th layer is the output one). Thus, for the *k*th neuron in the *j*th layer we have

$$\tilde{w}_i^{kj} = w_i^{kj} + \Delta w_i^{kj}; i = 0,..., N_{j-1}; k = 1,..., N_j; j = 1,..., m . \quad (1.36)$$

Thus, the derivation and description of the MLF learning algorithm with the error backpropagation is completed. In practice, the learning process should continue either until MSE or RMSE drops below some reasonable pre-defined minimum or until some pre-determined number of learning iterations is exceeded.

It is important to mention that this learning algorithm was really revolutionary. It opened absolutely new opportunities for using neural networks for solving classification and prediction problems that are described by non-linearly separable discrete and continuous functions.

However, we have to point out some specific limitations and disadvantages of this algorithm.

1) The backpropagation learning algorithm for MLF is developed as a method of solving the optimization problem. Its target is to find a global minimum of the error function. As all other such optimization methods, it suffers from a "local minima" phenomenon (see Fig. 1.11).

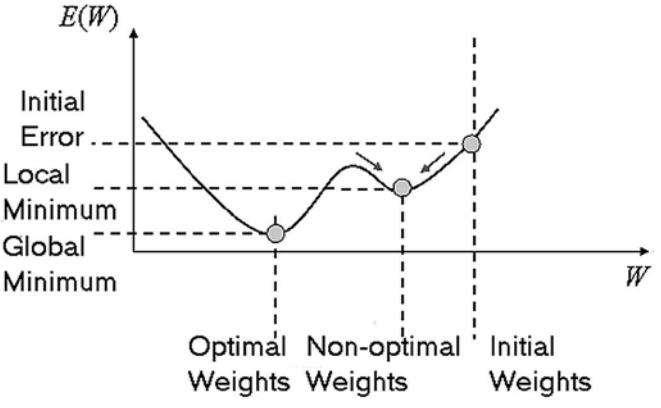


Fig. 1.11 A "local minima" phenomenon. The learning process may get stuck in a local minimum area. To reach a global minimum, it is necessary to jump over a local minimum using a proper learning rate.

The error function may have many local minima points. A gradient descent method, which is used in the MLF backpropagation learning algorithm, may lead the learning process to the closest local minimum where the learning process may get stuck. This is a serious problem and it has no regular solution. The only method of how to jump over a local minimum is to "play" with the learning rate β in (1.31), (1.34), and (1.35) increasing a step of learning. There are many recommendations on how to do that; however all of them are not universal and cannot guarantee that a global minimum of the error function will be reached.

2) Since the MLF backpropagation learning is reduced to solving the optimization problem, an activation function, which is used in MLF neurons, must be differentiable. This is a limitation, because, for example, discrete-valued activation functions cannot be used with this learning algorithm at all, since they are not differentiable. This complicates using MLF as a multi-class classifier and typically it is used just for two-class classification. In this case, the right "half" of the sigmoid activation function is truncated to "1" and the left half to "-1" or 0. For example, for functions (1.22) and (1.23) this means, respectively,

$$\hat{\varphi}_1(z) = \begin{cases} 1, \varphi(z) = \dfrac{1}{1+e^{-\alpha z}} \geq 0.5 \\ 0, \varphi(z) = \dfrac{1}{1+e^{-\alpha z}} < 0.5, \end{cases}$$

$$\hat{\varphi}_2(z) = \begin{cases} 1, \tanh \alpha z = \dfrac{e^{\alpha z} - e^{-\alpha z}}{e^{\alpha z} + e^{-\alpha z}} \geq 0 \\ -1, \tanh \alpha z = \dfrac{e^{\alpha z} - e^{-\alpha z}}{e^{\alpha z} + e^{-\alpha z}} < 0. \end{cases}$$

3) Sigmoid functions (1.22) and (1.23) are nonlinear, but their flexibility for approximation of highly nonlinear functions with multiple irregular jumps is limited. Hence, if we need to learn highly nonlinear functions, it is often necessary to extend a network by more hidden neurons.

4) Extension of a network leads to complications during the learning process. The more hidden neurons are in the network, the more is level of heuristics in the backpropagation algorithm. Indeed, the hidden neurons desired outputs and the exact errors are never known. The hidden layer errors can be calculated only on the base of the backpropagation learning algorithm, which is based on the heuristic assumption on the dependence of the error of each neuron on the errors of those neurons to which this neuron is connected. Increasing of the total number of weights in the network leads to complications in solving the optimization problem of the error functional minimization.

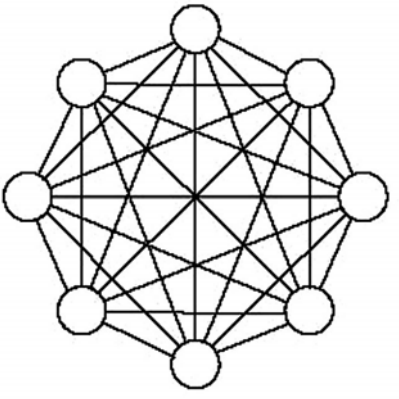
These remarks are important for us. When we will consider a backpropagation learning algorithm for the complex-valued multilayer feedforward neural network based on multi-valued neurons (Chapter 4), we will see that this network and its learning algorithm do not suffer from the mentioned disadvantages and limitations.

1.3.4 Hopfield Neural Network

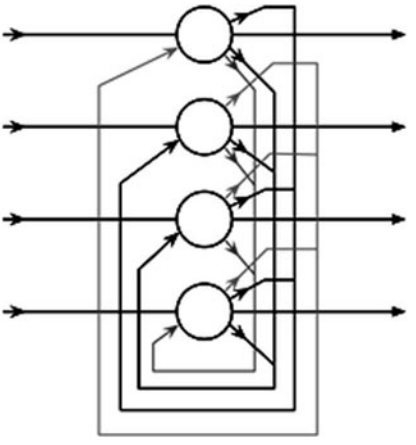
In 1982, John Hopfield proposed a fully connected recurrent neural network with feedback links [21]. The *Hopfield Neural Network* is a multiple-loop feedback neural network, which can be used first of all as an associative memory. All the neurons in this network are connected to all other neurons except to themselves that is there are no self-feedbacks in the network (see Fig. 1.12). Thus, the Hopfield network is a *fully connected neural network*. Initially, J. Hopfield proposed to use the binary threshold neurons with activation function (1.1) as the basic ones in this network.

The weight w_{ij} corresponds to the synaptic connection of the *i*th neuron and the *j*th neuron. It is important that in the Hopfield network, for the *i*th and *j*th neurons $w_{ij} = w_{ji}$. Since there is no self-connection, $w_{ii} = 0$. The network works cyclically updating the states of the neurons. The output of the *j*th neuron at cycle $t+1$ is

$$s_j(t+1) = \varphi\left(w_0^j + \sum_{i \neq j} w_{ij} s_i(t) \right). \tag{1.37}$$



a) The Hopfield neural network with 8 neurons



b) The Hopfield neural network with 4 neurons[4]

Fig. 1.12 Hopfield Neural Network

A main idea behind the Hopfield net is to use it as the *associative memory* (content-addressable memory). Initially this idea was suggested by Teuvo Kohonen in [22], but D. Hopfield comprehensively developed it in his seminal work [21], which was a great stimulus for the further development of neural networks after a "skeptical period" in 1970s caused by the M. Minsky's and S. Papert's analysis of limited capabilities of the perceptron [13]. The associative memory may learn patterns (for, example, if we want to store n x m images in the associative memory, we should take the n x m Hopfield network whose each neuron learns the intensity values in the corresponding pixels; in this case, there is a one-to-one correspondence between a set of pixels and a set of neurons). The Hebbian learning rule (1.3) or (1.4) can be effectively used for learning. After the learning process is completed, the associative memory may retrieve those patterns, which were learned, even from their fragments or from distorted (noisy or corrupted) patterns. The retrieval process is iterative and recurrent as it is seen from (1.37) (t is the number of cycle-iteration). D. Hopfield showed in [21] that this retrieval process always converges. A set of states of all the neurons on the tth cycle is called a *state of the network*. The network state on tth cycle is the network input for the t+1st cycle. The network is characterized by its *energy* corresponding to the current state. The energy is determined [21] as

$$E_t = -\frac{1}{2}\sum_i \sum_j w_{ij} s_i(t) s_j(t) + \sum_i w_0^i s_i(t). \tag{1.38}$$

Updating its states during the retrieval process, the network converges to the local minimum of the energy function (1.38), which is a stable state of the network.

[4] This picture is taken from Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/File:Hopfield-net.png

Once the network reaches its stable state, the retrieval process should be stopped. In practical implementation, the retrieval process should continue either until some pre-determined minimum of the energy function (1.38) is reached or until MSE or RMSE between the states on cycle t and t+1 drop below some pre-determined minimum.

In [23], D. Hopfield generalized all principles that he developed in [21] for a binary network with threshold neurons for a network with neurons with a continuous monotonic increasing and bounded activation function (for example, a sigmoid function) and with continuous states.

It is important to mention that the Hopfield neural network not only is the first comprehensively developed recurrent neural network. It also stimulated active research in areas of neural networks and dynamical systems in general. It is also worth to mention that the Hopfield network with continuous real-valued neurons suffers from disadvantages and limitations similar to the ones for MLF. For example, it is difficult to use such a network to store gray-scale images with 256 or more gray levels because local minima of the energy function are all located close to the corners of a unitary hypercube. Thus, a stable state of the network tends to a binary state. In Chapter 6, we will observe complex-valued associative memories based on networks with multi-valued neurons that do not suffer from these disadvantages.

1.3.5 Cellular Neural Network

The Hopfield neural network as we have seen is a fully connected network. The MLF is a network with full feedforward connections among adjacent layers neurons. We have also seen that the Hopfield network is a recurrent network. It updates its states iteratively until a stable state is reached. In 1988, Leon Chua and Lin Yang proposed another recurrent network with local connections [24] where each neuron is connected just with neurons from its closest neighborhood. They called it the *cellular neural network* (CNN). One of the initial ideas behind this network topology was to use it for image processing purposes. Since the correlation and respectively a mutual dependence between image pixels in any local n x m window is high, the idea was to create a recurrent neural network containing the same amount of neurons as the amount of pixels in an image to be processed. Local connections between the neurons could be used for implementation of various spatial domain filters, edge detectors, etc.

For example, CNN with 3x3 local connections is shown in Fig. 1.13. This network contains $N \times M$ neurons and it is very suitable for processing $N \times M$ images. The output of each neuron is connected to the corresponding inputs of 8 neurons closest to the given neuron (all neurons from a 3x3 neighborhood of a given neuron) and only to them, while outputs of these 8 adjacent neurons are connected to the corresponding inputs of a given neuron and there are only inputs of a given neuron. Unlike the Hopfield net, CNN allows a feedback connection, so each neuron may have one input receiving a signal from its own output. CNN is a recurrent network. Like the Hopfield network, it updates its states iteratively until a stable

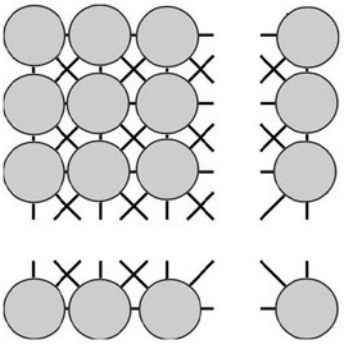


Fig. 1.13 Cellular Neural Network with 3x3 local connections

state is reached. CNN can be a binary network with the threshold neurons, but it can be based also on neurons with other activation functions, which makes it possible to implement different linear (using a piecewise linear activation function) and nonlinear (using nonlinear activation functions) filters.

Unlike it is in the Hopfield network, weights in CNN are not symmetric. Each neuron is indexed by two indexes-coordinates. The weight corresponding to the *i*th input of the *kj*th neuron is denoted w_i^{kj}. As we told, the network works cyclically updating the states of the neurons. The output of the *kj*th neuron at cycle $t+1$ is

$$s_{kj}(t+1) = \varphi\left(w_0^{kj} + \sum_i w_i^{kj} s_{rp}(t) \right); \quad k-d+2 \le r \le k+d-2,\ j-d+2 \le p \le j+d-2, \tag{1.39}$$

where φ is an activation function and d is the closest neighborhood size (for example, for a 3x3 local window $d=3$). In the CNN community, a very popular topic is mutual influence of a given neuron and those neurons connected to it. This is important for investigation of the stability of the network. In the context of this book, it will be enough for us to consider just equation (1.39), which determines the output of each neuron. The most interesting for us will be CNN based on multi-valued neurons, which can be successfully used as an associative memory (see Chapter 6, Section 6.3), significantly increasing the CNN functionality.

1.4 Introduction to Complex-Valued Neurons and Neural Networks

1.4.1 Why We Need Them?

We have already mentioned that complex numbers are absolutely natural, as well as real numbers. From this point of view, complex-valued neurons are natural too.

But additionally there are at least three very significant reasons for using complex-valued neurons and neural networks. These reasons are:

1) Unlike a single real-valued neuron, a single complex-valued neuron may learn non-linearly separable problems (a great variety of them) in that initial n-dimensional space where they are defined, without any nonlinear projection to a higher dimensional space (very popular kernel-based techniques, and the most popular and powerful of them – the support vector machines (SVM)[5] proposed by Vladimir Vapnik [25, 26] are based on this approach). Thus, a complex-valued neuron is much more functional than a real-valued one.

2) Many real-world problems, especially in signal processing, can be described properly only in the frequency domain where complex numbers are as natural as integer numbers in counting. In the frequency domain, it is essential to treat the amplitude and phase properly. But there is no way to have deal with the phase phenomenon without complex numbers. If we want to analyze any process, in which phase is involved, we should definitely use complex numbers and tools that are suitable for working with them. If we treate the phase as just real numbers belonging to the interval $[0,2\pi[$ or $[-\pi,\pi[$, then we make a great mistake, because in this way the physical nature of the phase is completely eliminated.

3) Since the functionality of a single complex-valued neuron is higher than the one of a single real-valued neuron, the functionality of complex-valued neural networks is also higher than the functionality of their real-valued counterparts. A smaller complex-valued neural network can learn faster and generalize better than a real-valued neural network. This is true for feedforward complex-valued networks and for Hopfield-like complex-valued networks. More functional neurons connected into a network ensure that this network also is more functional than its real-valued counterpart. We will see below (Chapter 4) that, for example, a feedforward multilayer neural network with multi-valued neurons (MLMVN) completely outperforms MLF. Even smaller MLMVN learns faster and generalizes better than larger MLF. Moreover, there are many problems, which MLF is not able to solve successfully, while MLMVN can. We will also see that a Hopfield-like neural network with multi-valued neurons can store much more patterns and has better retrieval rate as an associative memory, than a classical Hopfield network (Chapter 6, Section 6.3). Moreover, we will also see that just partially connected neural network with multi-valued neurons can also be used as a very powerful associative memory.

However, it is important for better understanding of the foregoing Chapters, to consider right now the first two of three mentioned reasons in more detail.

[5] While we presented in detail the most important classical neural network techniques, we do not present here in detail the SVM essentials. We believe that the interested reader can easily find many sources where SVM are described in detail. This book is devoted to complex-valued neural networks, but at least so far no complex-valued SVM were considered. However, we will compare a number of CVNN techniques presented in this book with SVM in terms of number of parameters they employ and generalization capability.

1.4.2 Higher Functionality

We have briefly observed what a neuron is, and what a neural network is. We have also observed not all, but the most important turning-points in real-valued artificial neurons and neural networks. We have mentioned several times that real-valued neurons and real-valued neural networks have some specific limitations. May be the most important of these limitations is impossibility of a single real-valued neuron to learn non-linearly separable input/output mappings in that initial linear *n*-dimensional space where the corresponding input/output mapping is defined. The classical example of such a problem, which cannot be learned by a single real-valued neuron due to its non-linear separability, is XOR as we have seen.

Table 1.6 Threshold neuron implements $f(x_1, x_2) = x_1 \text{ xor } x_2$ function with the weighting vector (0, 1, 1, 2) in 3-dimensional space (x_1, x_2, x_1x_2)

x_1	x_2	x_1x_2	$z = w_0 + w_1x_1 + w_2x_2 + w_3x_1x_2$	$\text{sgn}(z)$	$f(x_1, x_2) = x_1 \text{ xor } x_2$
1	1	1	4	1	1
1	-1	-1	-2	-1	-1
-1	1	-1	-2	-1	-1
-1	-1	1	0	1	-1

The reader may notice that the XOR problem can be learned using a single real-valued threshold neuron if the initial 2-dimensional space E_2^2 where it is defined, will be nonlinearly extended to the 3-dimensional space by adding to the two inputs x_1 and x_2 a nonlinear (quadratic) third input x_1x_2, which is determined by the product of the two initial inputs [27]. Indeed, let us consider the space (x_1, x_2, x_1x_2), which is obtained from E_2^2 by adding a quadratic term and, for example, the weighting vector $W = (0,1,1,2)$ [6]. This solution is shown in Table 1.6.

Actually, this solution confirms the Cover's theorem [28] on the separability of patterns, which states that a pattern classification problem is more likely to be linearly separable in a high dimensional feature space when nonlinearly projected into a high dimensional space. In fact, all kernel-based machine learning techniques including SVM are based on this approach. If some problem is non-linearly separable in that initial *n*-dimensional space where it is defined (for example, some classification problem described by some *n* features), it can be projected nonlinearly into a higher dimensional space where it becomes linearly separable. We have to understand that any feedforward neural network is also doing the same. It extends the

[6] We could also use here the weighting vector $W = (0,0,0,1)$.

initial space (if it contains more neurons in any hidden layer than network inputs) or at least transforms it nonlinearly into another space. When we considered in Section 1.3 solution of the XOR problem using MLF (see Fig. 1.7 and Table 1.5) containing three neurons and two layers, we nonlinearly projected initial space (x_1, x_2) into another (functional) space $(f_1(x_1), f_2(x_2))$ using the first layer neurons where the problem becames linearly separable using the third (output) neuron. This transformation is in fact nonlinear because it is implemented through a nonlinear activation function of neurons.

Nevertheless, is it possible to learn non-linearly separable problems using a single neuron without the extension or transformation of the initial space? The answer is "Yes!" It is just necessary to move to the complex domain!

In all neurons and neural networks that we have considered so far weights and inputs are real and weighted sums are real, respectively. Let us consider now complex-valued weights. Thus, weights can be arbitrary complex numbers. Inputs and outputs will still be real. Moreover, lest us consider even a narrow case of binary inputs and outputs. So, our input/output mapping is described by the function $f(x_1,...,x_n): E_2^n \to E_2$, which is a Boolean function. However, since our weights are complex ($w_i \in \mathbb{C}, i = 0,1,...,n$) and inputs are real $x_i \in E_2 = \{1,-1\}$, a weighted sum is definitely complex $w_0 + w_1 x_1 + ... + w_n x_n = z \in \mathbb{C}$. This means, that an activation function must be a function from $\mathbb{C}$ to E_2. Let us define the following activation function

$$\varphi(z) = \begin{cases} 1, \text{ if } 0 \le \arg z < \pi/2 \text{ or } \pi \le \arg z < 3\pi/2 \\ -1, \text{ if } \pi/2 \le \arg z < \pi \text{ or } 3\pi/2 \le \arg z < 2\pi, \end{cases} \tag{1.40}$$

where $\arg z$ is the argument of the complex number z in the range $[0, 2\pi[$. Evidently $\varphi(z)$ maps $\mathbb{C}$ to E_2, so $\varphi(z): \mathbb{C} \to E_2$. Activation function (1.40) divides the complex plane into 4 sectors (see Fig. 1.14) that coincide with the quarters of the complex plane formed by its separation with real and imaginary axes. Depending on $\arg z$, $\varphi(z)$ is equal to 1 in the 0th and the 2nd sectors (the 1st and the 3rd quarters) and to -1 in the 1st and the 3rd sectors (the 2nd and the 4th quarters).

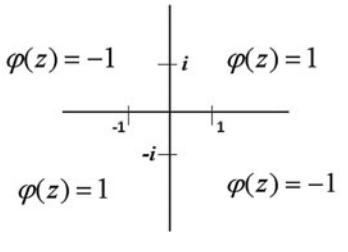


Fig. 1.14 Activation function (1.40)

Let us return to the most popular classical example of non-linearly separable problem – XOR. Let us show that a single neuron with the activation function (1.40) can easily implement the non-linearly separable XOR function without any extension of the original

Table 1.7 A complex-valued neuron with the activation function (1.40) implements the $f(x_1, x_2) = x_1 \text{ xor } x_2$ function with the weighting vector (0, *i*, 1) in the original 2-dimensional space

x_1	x_2	$z = w_0 + w_1 x_1 + w_2 x_2$	$\arg(z)$	$\varphi(z)$	$f(x_1, x_2) = x_1 \text{ xor } x_2$
1	1	$i+1$	$\pi/4$	1	1
1	-1	$i-1$	$3\pi/4$	-1	-1
-1	1	$-i+1$	$5\pi/4$	-1	-1
-1	-1	$-i-1$	$7\pi/4$	1	-1

2-dimensional space. Let us take the weighting vector $W = (0, i, 1)$ (*i* is an imaginary unity). The results are shown in Table 1.7.

These results shows that the XOR problem, which was for many years, on the one hand, a stumbling block in neurons theory [13] and, on the other hand, was a main argument for necessity of neural networks due to a limited functionality of a single neuron, can in fact be easily solved using a single neuron! But what is the most important – this is a single neuron with the complex-valued weights! This solution was for the first time shown by the author of this book in 1985 [29] and then it was deeply theoretically justified by him in [30].

The ability of a single neuron with complex-valued weights to solve non-linearly separable problems like XOR clearly shows that a single complex-valued neuron has a higher functionality than a single real-valued neuron. This is a crucial point!

We will show later (Chapter 5) why those problems that are non-linearly separable in the space $\mathbb{R}^n$ (or its subspace) can be linearly separable in the space $\mathbb{C}^n$ or its subspace. We will see there that problems like XOR and Parity *n* (*n*-input XOR or mod 2 sum of *n* variables) are likely the simplest non-linearly separable problems that can be learned by a single complex-valued neuron. We will also show that activation function (1.40) is a particular case of the 2-valued periodic activation function, which determines a universal binary neuron (UBN), which in turn is a particular case of the multi-valued neuron with a periodic activation function.

1.4.3 Importance of Phase and Its Proper Treatment

We have already mentioned that there are many engineering problems in the modern world where complex-valued signals and functions of complex variables are involved and where they are unavoidable. Thus, to employ neural networks for their analysis the use of complex-valued neural networks is natural.

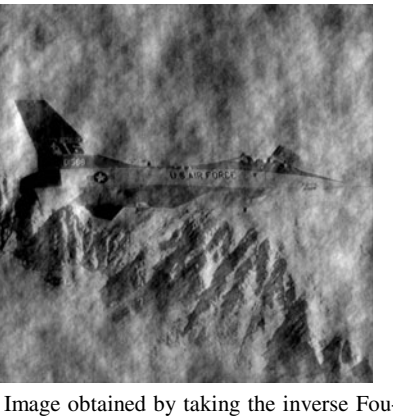
However, even in the analysis of real-valued signals (for example, images or audio signals) one of the most efficient approaches is the frequency domain analysis, which immediately involves complex numbers. In fact, analyzing signal properties in the frequency domain, we see that each signal is characterized by magnitude and phase that carry different information about the signal. A fundamental result showing the crucial importance of phase and its proper treatment was

presented in 1981 by Alan Oppenheim and Jae Lim [31]. They have considered, particularly, the importance of phase in images. They have shown that the information about all the edges, shapes, and, respectively, about all the objects located in an image, is completely contained in phase. Magnitude contains just the information about the contrast, about contribution of certain frequencies in the formation of an image, about the noisy component in the image, but not about what is located there. Thus, *phase is much more informative and important for image understanding and interpretation and for image recognition*, respectively.

(a) Original image "Lena"

(b) Original image "Airplane"

(c) Image obtained by taking the inverse Fourier transform from the synthesized spectrum (magnitude of the "Airplane" original spectrum and phase of the "Lena" original spectrum)



(d) Image obtained by taking the inverse Fourier transform from the synthesized spectrum (magnitude of the "Lena" original spectrum and phase of the "Airplane" original spectrum)

Fig. 1.15 The importance of phase

These properties can be easily confirmed by the experiments that are illustrated in Fig. 1.15. Let us take two well known test images[7] "Lena" (Fig. 1.15a) and

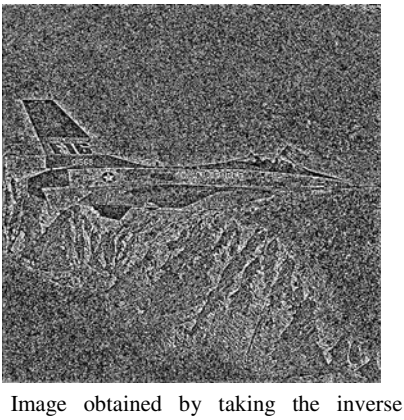
[7] These test images have been downloaded from the University of Sothern California test image database "The USC-SIPI Image Database", http://sipi.usc.edu/database/

"Airplane" (Fig. 1.15b). Let us take their Fourier transform and then swap magnitudes and phases of their Fourier spectra.

Thus, we synthesize one spectrum from magnitude of the "Airplane" spectrum and phase of the "Lena" spectrum and another one from phase of the "Airplane" spectrum and magnitude of "Lena" spectrum. Let us now take the inverse Fourier transform from both synthesized spectra. The results are shown in Fig. 1.15c and Fig. 1.15d, respectively. It is very clearly visible that definitely those images were restored whose phases were used in the corresponding synthesized spectra. In Fig. 1.15c we see just the "Lena" image, while in Fig. 1.15d we see just the "Airplane" image. There is no single trace of those images whose magnitudes were used in the synthesized Fourier spectra from which images in Fig. 1.15c and Fig. 1.15d have been obtained.

Another interesting experiment is illustrated in Fig. 1.16. We took the Fourier spectra of the same original images "Lena" (Fig. 1.15a) and "Airplane" (Fig. 1.15b). Then magnitudes in both spectra were replaced by the constant 1, while phases were preserved. Thus, magnitudes became "unitary". Then we took the inverse Fourier transform from these modified spectra with "unitary" magnitude. The results are shown in Fig. 1.16a ("Lena") and Fig. 1.16b ("Airplane"). It is clearly seen that all edges, shapes, and even the smallest details from the original images are preserved. Since images in Fig. 1.16 were obtained just from phase (magnitude was eliminated by setting all its values to 1), this confirms that all information about the edges, shapes, objects and their orientation is contained only in phase.

a) Image obtained by taking the inverse Fourier transform from the synthesized spectrum ("unitary" magnitude (constant 1) and phase of the "Lena" original spectrum)



b) Image obtained by taking the inverse Fourier transform from the synthesized spectrum ("unitary" magnitude (constant 1) and phase of the "Airplane" original spectrum)

Fig. 1.16 The importance of phase

These wonderful properties of phase are determined by its physical nature. The Fourier transform express any signal in terms of the sum of its projections onto a set of basic functions that represent those electromagnetic waves, which form this

signal. Hence, the Fourier transform is the decomposition of a signal by these basic functions that are defined as

$$e^{i2\pi ut} = \cos\left(2\pi ut\right) + i\sin\left(2\pi ut\right), \tag{1.41}$$

or in the discrete case

$$e^{i2\pi uk} = \cos\left(\frac{2\pi}{n}uk\right) + i\sin\left(\frac{2\pi}{n}uk\right); u,k = 0,1,...,n-1, \tag{1.42}$$

where u is the corresponding frequency. The Fourier spectrum of the continuous signal $f\left(t\right)$ is

$$F\left(u\right) = \int f\left(t\right)e^{-i2\pi ut} = \left|F\left(u\right)\right|e^{i\varphi(u)}, \tag{1.43}$$

where $\left|F\left(u\right)\right|$ is magnitude and $\varphi\left(u\right)$ is phase. For the discrete signal $f\left(k\right), k = 0,1,...,n-1$, equation (1.43) is transformed as follows

$$F\left(u\right) = \sum_{k=0}^{n-1} f\left(k\right)e^{-i2\pi uk} = \left|F\left(\omega\right)\right|e^{i\varphi(u)}; u = 0,1,...,n-1, \tag{1.44}$$

where each $\left|F\left(u\right)\right|e^{i\varphi(u)}, u = 0,1,...,n-1$ is referred to as a spectral coefficient or a decomposition coefficient. $\left|F\left(u\right)\right|$ is the absolute value (magnitude) of the uth spectral coefficient and $\varphi\left(u\right) = \arg F\left(u\right)$ is the argument (phase) of this spectral coefficient. To reconstruct a signal from (1.43), we have to perform the inverse Fourier transform

$$f\left(t\right) = \frac{1}{2\pi}\int F\left(\omega\right)e^{i2\pi ut}. \tag{1.45}$$

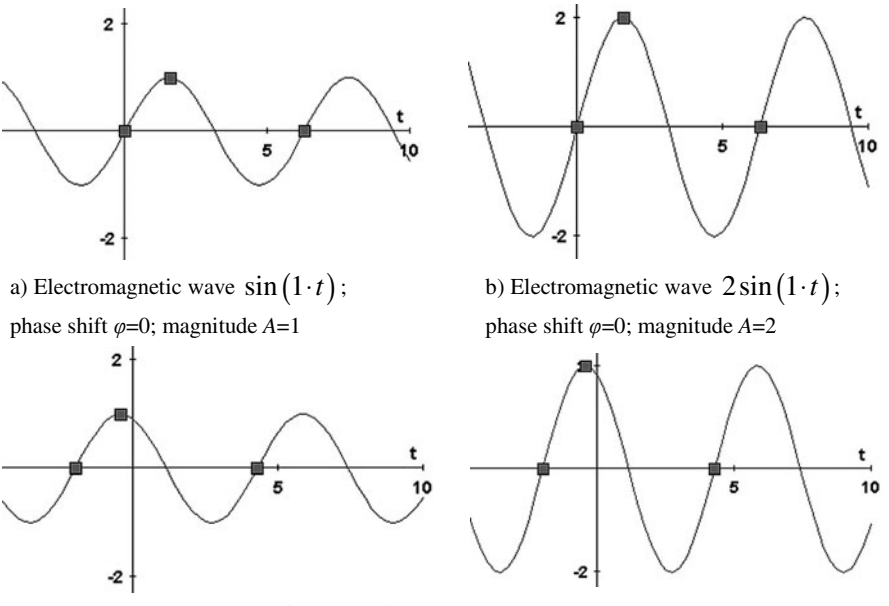
To reconstruct a signal from (1.44) in the discrete case, we have to perform the inverse Fourier transform – to find a sum of basic functions (waves) (1.42) with the coefficients (1.44):

$$f\left(k\right) = \frac{1}{n}\sum_{u=0}^{n-1} F\left(\omega\right)e^{i2\pi uk}; k = 0,1,...,n-1. \tag{1.46}$$

In (1.41) and (1.42) that are the basic functions of the Fourier transform, the electromagnetic waves corresponding to all frequencies have a zero phase shift. Let us set $2\pi u = \omega$ in (1.41). Then the corresponding basic function of the Fourier transform is $e^{iut} = \cos\left(ut\right) + i\sin\left(ut\right)$. Respectively, (1.45) can be written as follows

$$f(t) = \frac{1}{2\pi} \int F(\omega) e^{i\omega t} \,. \tag{1.47}$$

Let us take a look at Fig. 1.17a. It shows a sinusoidal wave $\sin(2\pi u t)$ for $u = 1$. According to (1.46) and (1.47), after this sinusoidal wave is multiplied with the Fourier spectral coefficient $F(u)$, its absolute value (magnitude) is equal



a) Electromagnetic wave $\sin(1 \cdot t)$; phase shift φ=0; magnitude A=1

b) Electromagnetic wave $2\sin(1 \cdot t)$; phase shift φ=0; magnitude A=2

c) Electromagnetic wave $\sin((1 \cdot t) + 2)$; phase shift φ=2; magnitude A=1

d) Electromagnetic wave $2\sin((1 \cdot t) + 2)$; phase shift φ=2; magnitude A=2

Fig. 1.17 A role of phase and magnitude in the Fourier transform. Phase in a Fourier transform coefficient shows the phase shift for the electromagnetic wave with the frequency corresponding to the given Fourier transform coefficient. The phase shift is a carrier of information about a signal concentrated in the wave with the corresponding frequency. Magnitude in a Fourier transform coefficient just shows the intensity (the "weight") of the wave corresponding to the given frequency in the formation of a signal[8]

[8] To create these pictures, we used a wonderful tool located at http://www.ugrad.math.ubc.ca/coursedoc/math100/notes/trig/phase.html (this is a site of University of British Columbia, Canada)

to $A\left|e^{i\omega t}\right| = A\left|\cos(\omega t) + i\sin(\omega t)\right|$ where $A = \left|F(u)\right|$. Fig. 1.17b shows a sinusoidal wave $A\sin(2\pi ut)$ for $u = 1$ and $A = \left|F(u)\right| = 2$. The phase shift of these both electromagnetic waves is equal to 0. Thus, if a sinusoidal wave has a basic form $A\sin(\omega t + \varphi)$, then waves in Fig. 1.17a and Fig. 1.17b have $\varphi = 0$. It follows from (1.47) that $F(u)e^{i2\pi ut} = \left|F(u)\right|e^{i(\omega t + \arg(F(u)))}$ because the argument of the product of two complex numbers is equal to the sum of multipliers' arguments, while the magnitude of the product is equal to the product of magnitudes (take into account that $\left|e^{i2\pi ut}\right| = 1$). Fig. 1.17c and Fig. 1.17d show sinusoidal waves $A\sin(\omega t + \varphi)$ with the phase shift $\varphi = 2$. These waves could be obtained by multiplication of the "standard" wave $\sin(\omega t)$ by such $F(u)$ that $\arg F(u) = 2$. For the sinusoidal wave in Fig. 1.17c, $A = \left|F(u)\right| = 1$, while for the one in Fig. 1.17d, $A = \left|F(u)\right| = 2$. Hence, the sinusoidal waves in Fig. 1.17a, b have the same phase shifts and different magnitudes, and the sinusoidal waves in Fig. 1.17c, d have the same phase shifts and different magnitudes.

As we see, *the phase shift is nothing else than phase of the Fourier transform coefficient* corresponding to a wave with the certain frequency. This shift determines the contribution of this wave to the shape of a signal after its reconstruction from the Fourier transform while magnitude plays only subsidiary role.

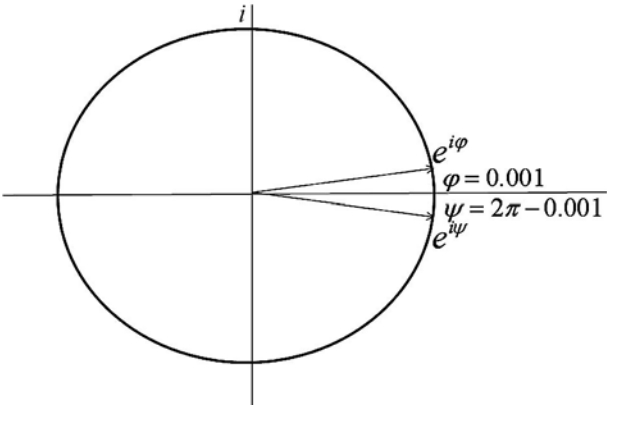


Fig. 1.18 Importance of proper treatment of phase as an angle

Hopefully, it is clear now why all shapes and even all details of images in Fig. 1.16 were successfully reconstructed from the Fourier transform whose magnitude was completely eliminated and replaced by the constant 1. It is also clear now how images in Fig. 1.15 were reconstructed just from the original phases of their Fourier spectra, while their magnitudes were swapped.

This means that phase is a very important carrier of information about those objects that are presented by a signal. This information can be used, for example, for solving image recognition problems (we will consider this later, in Chapter 6). However, first of all it is absolutely important to treat properly phase and that information, which is concentrated in phase. We have to treat the phase φ only as an angular value determining the complex number $e^{i\varphi}$ located on the unit circle. Any attempt to work with phases as with formal real numbers located either in interval $[0,2\pi[$ or $[-\pi,\pi[$ without taking into account that they are angles that are in turn arguments of complex numbers, completely eliminates a physical nature of phase. If we do not treat phases properly (as arguments of complex numbers), then the information, which is contained in phase, is completely distorted.

For example, if we do not care of the nature of phase, we may treat numbers $\varphi = 0.001$ and $\psi = 2\pi - 0.001 = 6.282$ as such located in the opposite ends of the interval $[0,2\pi[$. In this case, their formal difference is 6.282-0.001=6.281. But in fact, these numbers determine angles that are very close to each other, and the difference between them is just 0.002 radian. Respectively, these two phases determine two points on the unit circle $e^{i\varphi}$ and $e^{i\psi}$ that are located very close to each other (see Fig. 1.18).

Thus, to treat phases properly, they have to be considered only as arguments of complex numbers. To work only with that information concentrated in phase, it is enough to consider phases as arguments determining complex numbers located on the unit circle. In this case, we do not care of magnitude (like in the example presented in Fig. 1.16). We will see below (Chapter 2) that this is definitely the case of a multi-valued neuron whose inputs and output are always located on the unit circle.

Hence, to analyze phase and the information contained in phase, using neural networks, it is absolutely important to use complex-valued neurons.

1.4.4 Complex-Valued Neural Networks: Brief Historical Observation and State of the Art

Before we will move to the detailed consideration of multi-valued neurons, neural networks based on them, their learning algorithms and their applications, let us present a brief historical overview of complex-valued neural networks and state of the art in this area.

The first historically known complex-valued activation function was suggested in 1971 by Naum Aizenberg and his co-authors Yuriy Ivaskiv and Dmitriy Pospelov in [32]. Thus, complex-valued neural networks start their history form this seminal paper. A main idea behind this paper was to develop a model of multiple-valued threshold logic, to be able to learn and implement multiple-valued functions using a neural element similarly to learning and implementation of Boolean threshold functions using a neuron with the threshold activation function. Moreover, according to this new model, Boolean threshold logic should be just a

particular case of multiple-valued threshold logic. We will consider this model in detail in Chapter 2. Now we just want to outline a basic approach.

As we have already seen, in neural networks the two-valued alphabet $E_2 = \{1, -1\}$ is usually used instead of the traditional Boolean alphabet $K_2 = \{0, 1\}$. This can easily be explained by two factors. First of all, unlike in K_2, in E_2 values of two-valued logic are normalized, their absolute value is equal to 1. Secondly, we have seen that, for example, in the error-correction learning rule (1.17), $x_i \in E_2$ is a very important multiplicative term participating in the adjustment of the weight $w_i, i = 1, ..., n$. If it was possible that $x_i = 0$, then the error-correction learning rule (1.17) could not be derived in that form, in which it exists.

In the classical multiple-valued (*k*-valued) logic, the truth values are traditionally encoded by integers from the alphabet $K = \{0, 1, ..., k-1\}$. They are not normalized. If we want to have them normalized, evidently, this problem can be solved neither within the set of integer numbers nor the set of real numbers for $k > 2$. However, Naum Aizenberg suggested a wonderful idea to jump to the field of complex numbers and to encode the values of *k*-valued logic by the *k*th roots of unity (see Fig. 1.19). Since there are exactly *k* *k*th roots of unity, it is always possible and very easy to build a one-to-one correspondence between the set $K = \{0, 1, ..., k-1\}$ and the set $E_k = \{1, \varepsilon_k, \varepsilon_k^2, ..., \varepsilon_k^{k-1}\}$, where $\varepsilon_k = e^{i2\pi/k}$ is the primitive k^{th} root of unity (*i* is an imaginary unity). We will consider later in detail, (Chapter 2, Section 2.1) a mathematical background behind this idea. Unlike in the set *K*, in the set E_k the values of *k*-valued logic are normalized – their absolute values are equal to 1. Particularly, for two-valued logic, $E_2 = \{1, -1\}$, which corresponds to $K_2 = \{0, 1\}$, and we obtain a well known model of Boolean logic in the alphabet $E_2 = \{1, -1\}$.

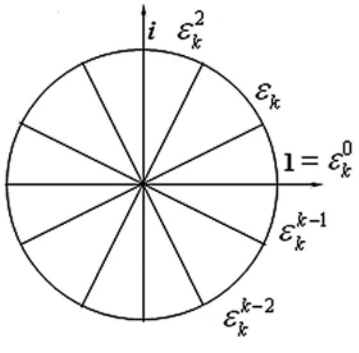
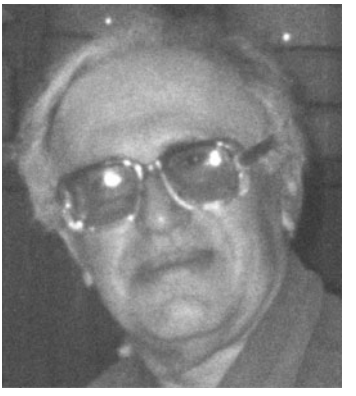


Fig. 1.19 Model of *k*-valued logic over the field of complex numbers. Values of *k*-valued logic are encoded by the *k*th roots of unity

Thus, in multiple-valued logic over the field of complex numbers, a multiple-valued (*k*-valued) function of *n* variables becomes $f(x_1, ..., x_n): E_k^n \to E_k$. Naum Aizenberg and his co-authors suggested in [32] the following activation function, which they called



Naum Nisonovich Aizenberg (1928-2002)

Seminal ideas in the area of complex-valued neural networks and in multi-valued neurons were proposed and developed by Professor Naum N. Aizenberg. He was born in Kiev (Ukraine, that time USSR). From 1953 to 1998 he was with Uzhgorod National University (USSR until 1991 and then Ukraine) where he has started as a part time teaching assistant and then became a Professor. For a number of years he was a Chair of the Department of Cybernetics. His first love in research was Algebra, which formed a solid background for his further work in Computer Science and Engineering. His main result in Algebra is solution of the problem of computation of the wreath products of the finite groups. In early 1970s he developed a theory of multiple-valued threshold logic over the field of complex numbers, which became a background for complex-valued neural networks. He also developed an algebraic theory of signal processing in an arbitrary basis. His important accomplishment is also a theory of prime tests, which found many applications in Pattern Recognition. His 11 Ph.D. students got their Ph.D. degrees under his supervision. He retired in 1998 after he got a damaging heart attack. The same year he moved from Ukraine to Israel. Even being seriously ill, he continued his research as far as possible, collaborating with other colleagues. His last paper has been published right after he passed in 2002…

CSIGN[9] (keeping in mind that this is a specific generalization of the sgn function for the multiple-valued case)

$$\mathrm{CSIGN}(z) = \varepsilon_k^j, 2\pi j / k \le \arg z < 2\pi (j+1) / k \tag{1.48}$$

Function (1.48) divides complex plane into k equal sectors (see Fig. 1.19). We will consider it and its properties in detail in Chapter 2. Now we can say that it follows form (1.48) that if the complex number z is located in the sector j, then

[9] In the later work, where the multi-valued neuron was introduced, N. Aizenberg himself suggested to use another notation for the function CSIGN. Since in terms of logic this function is multiple-valued predicate, he suggested to use just a letter P ("Predicate") for its notation considering that the initial CSIGN was not successful, because in fact a complex number does not have a sign. We will use the notation P throughout the book except this section.

$\mathrm{CSIGN}(z) = \varepsilon_k^j$. Then a notion of multiple-valued threshold function was introduced in [32]. A function $f(x_1,...,x_n): E_k^n \to E_k$ is called a *multiple-valued threshold function* if there exist such n+1 complex numbers (weights) $w_0, w_1,..., w_n$ that for all $(x_1,...,x_n)$ from the domain of the function f

$$\mathrm{CSIGN}(w_0 + w_1 x_1 + ... + w_n x_n) = f(x_1,...,x_n). \tag{1.49}$$

Paper [32] was then followed by two papers [33, 34] by N. Aizenberg and co-authors where a multi-valued threshold element was introduced as a processing element implementing (1.49) and, respectively, implementing a multiple-valued threshold function. A learning algorithm for this element was also introduced in [34]. By the way, papers [33, 34] originally published only in Russian (as well as [32]) are available now in English [35, 36] (the English version of the journal Cybernetics and Systems Analysis (previously Cybernetics) is published by Springer from late 1990s, and all the earlier journal issues are translated into English too and they are available online from the Springer website[10]). Papers [32-35] were followed in 1977 by the monograph [37] (also published only in Russian) by N. Aizenberg and Yu. Ivaskiv. In [37], all theoretical aspects of multiple-valued threshold logic over the field of complex numbers, multi-valued threshold elements, and their learning were comprehensively observed. It is important to mention that a word "neuron" was not used in those publications, but it is absolutely clear that a *multi-valued threshold element* is nothing else than the discrete multi-valued neuron formally named a neuron in 1992 [38] by N. Aizenberg and the author of this book.

It is difficult to overestimate the importance of the seminal publications [32-34, 37]. For the first time, a neural element introduced there, could learn multiple-valued input/output mappings $E_k^n \to E_k$ and $O^n \to E_k$ (O is a set of points on the unit circle). This means that it was possible to use it for solving, for example, multi-class classification problems where the number of classes is greater than 2. Unfortunately, published only in Russian, these important results were unavailable to the international research community for many years. In 1988 (17 years later (!) after paper [32] was published) A. Noest even "re-invented" activation function (1.48) calling a neuron with this activation function a "phasor neuron" [39]. But in fact, this activation function was proposed in 1971 and we believe that A. Noest simply was not familiar with [32].

Since Chapters 2-6 of this book are completely devoted to multi-valued neurons and neural networks based on them, we will observe all publications devoted to MVN and MVN-based neural networks later as the corresponding topics will be deeply considered. However, we would like to observe briefly now other important works on complex-valued neural networks, not related to MVN.

[10] http://www.springer.com/mathematics/applications/journal/10559

Starting from early 1990s complex-valued neural networks became a very rapidly developing area. In 1991 and 1992, independently on each other, H. Leung and S. Haykin [40], and G. Georgiou and C. Koutsougeras [41], respectively, generalized the MLF backpropagation learning algorithms for the complex-valued case. They considered complex weights and complex-valued generalization of the sigmoid activation function and showed that complex backpropagation algorithm converges better than the real one.

Important contributions to CVNN are done by Akira Hirose He is the author of the fundamental monograph [42] with a detailed observation of the state of the art in the field, and the editor of the book [43] with a great collection of papers devoted to different aspects of complex-valued neural networks. He also was one of the first authors who considered a concept of fully-complex neural networks [44] and continuous complex-valued backpropagation [45].

Other interesting contributions to CVNN are done by Tohru Nitta. He has edited a recently published book on CVNN [46]. He also developed the original approach to complex backpropagation [47], and he is probably the first author who considered a quaternion neuron [48].

Very interesting results on application of complex-valued neural networks in nonlinear filtering are obtained by Danilo Mandic and under his supervision. Just a few of his and his co-authors important contributions are recently published fundamental monograph [49] and papers on different aspects of filtering [50] and prediction [51].

Important contributions to learning algorithms for complex-valued neural networks are done by Simone Fiori. We should mention here among others his generalization of Hebbian Learning for complex-valued neurons [52, 53] and original optimization method, which could be used for learning in complex-valued neural networks [54].

We should also mention recently published works by Md. F. Amin and his co-authors [55, 56] on solving classification problems using complex-valued neural networks.

It is also important to mention here interesting works by Sven Buchholz and his co-authors on neural computations in Clifford algebras where complex-valued and quaternion neurons are involved [57]. They also recently developed a concept of quaternionic feedforward neural networks [57, 58].

1.5 Concluding Remarks to Chapter 1

In this introductory Chapter, we have briefly considered a history of artificial neurons and neural networks. We have observed such turning-point classical solutions and concepts as the McCulloch-Pitts neuron, Hebbian learning, the Rosenblatt's perceptron, error-correction learning, a multilayer feedforward neural network, backpropagation learning, and linear separability/non-linear separability. We have paid a special attention to those specific limitations that characterize real-valued neural networks. This is first of all impossibility of a single real-valued neuron to learn non-linearly separable problems. This is also strict dependence of the backpropagation learning algorithm on the differentiability of an activation function.

This is also absence of some regular approach to representation of multiple-valued discrete input/output mappings.

We have shown that moving to the complex domain it is possible to overcome at least some of these disadvantages. For example, we have shown how a classical non-linearly separable problem XOR can be easily solved using a single complex-valued neuron without the extension of that 2-dimensional space where it is defined. We have also shown that complex-valued neurons can be extremely important for a proper treatment of phase, which in fact contains much more significant information about the objects presented by the corresponding signals.

We briefly presented the first historically known complex-valued activation function, which makes it possible to represent multiple-valued discrete input/output mappings.

We have also observed recent contributions in complex-valued neural networks. We have mentioned here just recent and perhaps the most cited works. Nevertheless, it follows from this observation that complex-valued neural networks have become increasingly popular. The reader may find many other papers devoted to different aspects of CVNN. Just, for example, take a look at [43, 46] where very good collections of papers are presented. There were also many interesting presentations in a number of special sessions on complex-valued neural networks organized just during last several years (IJCNN-2006, ICANN-2007, IJCNN-2008, IJCNN-2009, and IJCNN-2010). As the reader may see, there are different specific types of complex-valued neurons and complex-valued activation functions. Their common great advantage is that using complex-valued inputs/outputs, weights and activation functions, it is possible to improve the functionality of a single neuron and of a neural network, to improve their performance, and to reduce the training time (we will see later, for example, how simpler and more efficient is learning of MVN and MVN-based neural networks).

We hope that the reader is well prepared now to move to the main part of this book where we will present in detail the multi-valued neuron, its learning, and neural networks based on multi-valued neurons. We will also consider a number of examples and applications that will show great advantages of the multi-valued neuron with complex-valued weights and complex-valued activation function over its real-valued counterparts.