**INSTITUTE OF COMPUTER SCIENCE**

**ACADEMY OF SCIENCES OF THE CZECH REPUBLIC**

# Introduction to Neural Networks

Jiří Šíma

Technical report No. V-755

August 10, 1998

**Institute of Computer Science, Academy of Sciences of the Czech Republic**
**Pod vodárenskou věží 2, 182 07 Prague 8, Czech Republic**
**phone: (+4202) 66 05 30 30  fax: (+4202) 8585789**
**e-mail: sima@uivt.cas.cz**

## Abstract

The presented technical report is a preliminary English translation of selected revised sections from the first part of the book *Theoretical Issues of Neural Networks* [75] by the first author which represents a brief introduction to neural networks. This work does not cover a complete survey of the neural network models but the exposition here is focused more on the original motivations and on the clear technical description of several basic type models. It can be understood as an invitation to a deeper study of this field. Thus, the respective background is prepared for those who have not met this phenomenon yet so that they could appreciate the subsequent theoretical parts of the book. In addition, this can also be profitable for those engineers who want to apply the neural networks in the area of their expertise. The introductory part does not require deeper preliminary knowledge, it contains many pictures and the mathematical formalism is reduced to the lowest degree in the first chapter and it is used only for a technical description of neural network models in the following chapters. We will come back to the formalization of some of these introduced models within their theoretical analysis.

The first chapter makes an effort to describe and clarify the neural network phenomenon. It contains a brief survey of the history of neurocomputing and it explains the neurophysiological motivations which led to the mathematical model of a neuron and neural network. It shows that a particular model of the neural network can be determined by means of the architectural, computational, and adaptive dynamics that describe the evolution of the specific neural network parameters in time. Furthermore, it introduces neurocomputers as an alternative to the classical von Neumann computer architecture and the appropriate areas of their applications are discussed.

The second chapter deals with the classical models of neural networks. First, the historically oldest model — the network of perceptrons is shortly mentioned. Further, the most widely applied model in practice — the multi–layered neural network with the back-propagation learning algorithm, is described in detail. The respective description, besides various variants of this model, contains implementation comments as well. The explanation of the linear model MADALINE, adapted according to the Widrow rule, follows.

The third chapter is concentrated on the neural network models that are exploited as autoassociative or heteroassociative memories. The principles of the adaptation according to Hebb law are explained on the example of the linear associator neural network. The next model is the well–known Hopfield network, motivated by physical theories, which is a representative of the cyclic neural networks. The analog version of this network can be used for heuristic solving of the optimization tasks (e. g. traveling salesman problem). By the physical analogy, a temperature parameter is introduced into the Hopfield network and thus, a stochastic model, the so–called Boltzmann machine is obtained.

The information from this part of the book can be found in an arbitrary monograph or in survey articles concerning neural networks. For its composition we issued namely from the works [16, 24, 26, 27, 35, 36, 45, 73]. This work was supported by GA ČR Grant No. 201/98/0717.

**Keywords**
neural networks, perceptron, backpropagation, MADELINE, associative memory, Hopfield network, Boltzmann machine

# Contents

# Chapter 1

# Neural Network Phenomenon

## 1.1 History of Neurocomputing

The history of the neural network field is often considered to begin with the paper by Warren McCulloch and Walter Pitts from 1943 [50]. In this paper they introduced the first, very simplified, mathematical model of a *neuron* which is the basic cell of the nervous system. Their neuron operated in an all-or-non fashion and thus, the model was discrete, based on binary logic. They showed that an arbitrary logical or arithmetical function could, in principle, be computed by using even simple types of neural networks. However, their model still lacked the capability to learn. This work influenced other researchers. Namely, the founder of cybernetics, Norbert Wiener was inspired by it when he studied similarities of nervous and computer systems. Or the author of an American project of electronic computers, John von Neumann, wrote papers [55, 56] in which he suggested the research of computers that were inspired by brain functioning.

In 1949 Donald Hebb wrote a book entitled *The Organization of Behavior* [25] where he described a learning paradigm that now bears his name. Particularly, Hebb law (see Section 3.1.1) states that the permeability of a *synapse* (inter-neuron interface) increases if there is a presynaptic neuron activity followed closely in time with a postsynaptic activity. This rule is based on the idea that the conditioned reflex which is observed in animals is a property of individual neurons. By using this law, Hebb made an effort to explain some experimental results from psychology. Also his work had an impact on other researchers who, in 1940s and early 1950s, set the stage for later developments. A typical representative of this era was Marvin Minsky who is responsible for some of the first results using neural networks. In 1951 Minsky teamed with Dean Edmonds designed the first (40-neuron) neurocomputer *Snark* [53] with synapses that adjusted their *weights* (measures of synaptic permeabilities) according to the success of performing a specified task (Hebbian learning). The machine was built of tubes, motors, and clutches, and it successfully modeled the behavior of a rat in a maze searching for food. The architecture of Snark later inspired other constructors of neurocomputers.

In 1957 Frank Rosenblatt invented the *perceptron* [66] (see Section 2.1) which generalized the McCulloch-Pitts neuron by supplying this model with a learning algorithm,

i. e. a way to change (generally real) weights iteratively so that a desired task was performed. Rosenblatt was able to prove the convergence theorem stating that this algorithm always finds the weight vector that is consistent with a given training data (providing that it exists), independently on its initial setting after a finite number of steps. This result caused great excitement and hope that such machines could be a basis for artificial intelligence. Rosenblatt also wrote an early book on neurocomputing, *Principles of Neurodynamics* [67].

During the years 1957 to 1958, on the basis of his research Rosenblatt together with Charles Wightman and others [66] constructed and successfully demonstrated a neurocomputer called the *Mark I Perceptron.* Since Rosenblatt's original research interests include pattern recognition, the Mark I Perceptron was designed as a character recognizer. A character was placed on a floodlight-illuminated board and the image of this character was focused upon a $20 \times 20$ array of CdS photoconductors. The intensity of 400 image pixels represented an input to the network of perceptrons whose task was to classify which character is presented (e. g., "A", "B" etc.). The Mark I Perceptron had 512 adjustable weight parameters which were implemented as an $8 \times 8 \times 8$ array of potentiometers. The resistance value associated with each potentiometer which corresponded to a particular weight, was adjusted automatically by a separate motor. This motor was controlled by an analog circuit that implemented the perceptron learning rule. The inputs could arbitrarily be connected to particular perceptrons via a patch panel. Typically, a 'random' connection was used to illustrate the ability of perceptron to learn the desired pattern without a need for precise wiring, in contrast to the conventional programmed computers. Also because of the successful presentation of the Mark I Perceptron, the neurocomputing field which was an alternative to the classical computations realized on the von Neumann computer architecture, became a new subject of intensive research. Therefore, Frank Rosenblatt is considered by many people to be the founder of this new field as we know it today.

Shortly after the perceptron discovery, Bernard Widrow together with his students developed another type of neural computational element which he called *ADALINE* (**ADA**ptive **LIN**ear **E**lement) [82] (see Section 2.3). This model was equipped with a new effective learning rule which is still being successfully used for adaptive signal processing, control systems, etc. Widrow and a young graduate student, Marcian Hoff, mathematically proved that this algorithm converges under certain conditions. The functionality of ADALINE was demonstrated on many simple toy examples. Widrow also founded the first neurocomputer hardware company, *Memistor Corporation* which actually produced and sold neurocomputers and their components during the early to mid 1960s.

During the late 1950s and early 1960s, the successful progress in neurocomputing was confirmed by proposing new models of neural networks, their implementations, and applications. For example, Karl Steinbuch developed a model of binary associative network called the *Learning Matrix* which was applied to problems such as recognizing highly distorted handwritten characters, diagnosing mechanical failures to reduce downtime of machines, and controlling higher processes in production. Or, in 1960, Roger Barron and Lewey Gilstrap founded the first neurocomputing applications company, the *Adaptronics Corporation.* The results of this period are summarized in the 1965 book *Learning Machines* [57] by Nils Nilsson.

In spite of the undoubted success achieved in this period, the neural network field also faced apparent problems. Most of researchers approached to neural networks from an experimental point of view (reminiscent of alchemy) omitting analytical research. Also the excitement of neural network specialists led to a publicity of unjustified statements such as an artificial brain is not far from being developed, etc. This situation discredited the neural network research and discouraged scientists and engineers from being interested in neurocomputing. In addition, the neural network field exhausted itself and a further progress in this area would have required radically new ideas and approaches to be introduced. The best experts left this field and started to deal with related areas of artificial intelligence.

The final episode of this epoch was a campaign led by Marvin Minsky and Seymour Papert who wrote an unpublished technical report which, among otherthings, analyzed the crisis of neural networks. This manuscript, after being extended and revised, was published later in 1969 as an extremely influential book *Perceptrons* [54]. In this work they exploited the known fact that a single perceptron cannot compute some of the simple logical functions such as the exclusive disjunction (XOR), in order to discredit the neural network research. Although this problem can be solved by using a two-layered network, however, at that time no learning algorithm was known for the multilayered perceptron. The authors doubted that such an algorithm would be found since the function of the multilayered neural network is much more complicated than that of the single perceptron. This statement was widely accepted and considered erroneously to be mathematically proved. Thus, Minsky and Papert successfully convinced enough people that further neurocomputing study was pointless which, in turn, had the effect of reallocating the majority of neural network funding into artificial intelligence programs. With this, most of the computer science community left the neural network paradigm for almost 20 years.

In this 'quiet' period from 1967 to 1982 neurocomputing research was carried out either outside of the United States where the book *Perceptrons* had great influence or the majority of neural network results were published under the headings of adaptive signal processing, biological modelling, etc. Still, there were a number of people who continued to develop the neural network theory in the 1970s. A major theme was *associative content-addressable memory* (see Chapter 3), in which different input patterns become associated with one another if sufficiently similar. Also new talented researchers entered the neural network field in this period. Some of them, together with their main research orientation at that time, are listed below.

For example, Shun-Ichi Amari combined biological neural network activity and rigorous mathematical expertise in his studies of the dynamics of randomly connected networks, competitive learning, associative memories, stability of cyclic networks, etc. Or James Anderson formed a linear associative memory which he applied to the recognition, reconstruction and association of visual patterns. Kunihiko Fukushima created the so-called *cognitron* which is a multilayered network for vision. This model was later improved by him (*neocognitron*) and successfully demonstrated for recognizing handwritten numerals that were distorted, shifted, rotated and scaled in many different configurations. One of the most influential neural network researchers, Stephen Grossberg, started to study extensively the psychological and biological processing and the phenomena of human information processing. Grossberg's work has included strict mathematical analysis and emphasized producing neural network paradigms that are self-organizing, self-stabilizing, and self-scaling, that allow direct access to information while operating in real-time. Further, Harry Klopf has been studying the relationship between the psychology of the mind and the biology of the brain since 1969. Also Teuvo Kohonen began his neural network research with randomly connected paradigms in 1971 and quickly focused upon associative memories. His later research led to self-organizing feature maps. Finally, David Willshaw analyzed the self-organization and generalization properties of neural networks. These researchers contributed with their discoveries to the later renaissance of neural networks.

By the early 1980s neurocomputing researchers started to submit their own grant projects oriented towards the development and application of neurocomputers. Thanks to program manager Ira Skurnick, the American grant agency *DARPA* (**D**efense **A**dvanced **R**esearch **P**rojects **A**gency) began funding neurocomputing research in 1983 and within a short time this example has been followed by other organizations supporting basic and applied research. Further credit for the renaissance of neural networks is counted to well-known physicist John Hopfield who started to be interested in neurocomputing at that time. His results were published in two highly readable papers in 1982 and 1984 [31, 32] where he showed connections between some models of neural networks and physical models of magnetic materials. By his invited lectures all over the world hundreds of qualified scientists, mathematicians, and technologists were attracted to neural networks.

In 1986 researchers forming the *PDP* (**P**arallel **D**istributed **P**rocessing) *Research Group* published their results in a two-volume set of books edited by David Rumelhart and James McClelland [69]. Here the paper by Rumelhart, Geoffrey Hinton a Ronald Williams [68] appeared in which they described the learning algorithm *backpropagation* for multilayered neural networks (see Section 2.2). Thus, they solved the problem which seemed to Minsky and Papert in 1960s to be an insuperable obstacle to the exploitation and further development of neural networks. This algorithm represents still the most widely applied method in neural networks. By publishing the above-mentioned proceedings the interest in neurocomputing reached the top. It was later shown that the backpropagation learning algorithm was actually re-discovered since it had been known and published by several researchers in 'quiet' period (e. g. Arthur Bryson a Yu-Chi Ho, 1969 [11]; Paul Werbos, 1974 [81]; David Parker, 1985 [58]), however, that does not alter the case.

The system *NETtalk* (see Paragraph 1.4.2) developed by Terrence Sejnowski and Charles Rosenberg [71] is the well-known example that, in the beginning, illustrated the practical importance of the backpropagation algorithm. This system, after being created in a relatively short time by learning a neural network from examples, successfully converted the English written text to spoken English. It competed with its predecessor, the system *DECtalk* (**D**igital **E**quipment **C**orporation) which contained hundreds of rules created by linguists over decades.

In 1987 the first bigger conference specialized on neural networks in modern times, the *IEEE International Conference on Neural Networks* with 1700 participants was held in San Diego, and the *International Neural Network Society* (*INNS*) has been established. One year later the INNS began to publish its journal *Neural Networks*, followed by *Neural Computation* (1989), *IEEE Transactions on Neural Networks* (1990) and many others. Beginning in 1987, many prestigious universities founded new research institutes and educational programs in neurocomputing. This trend has continued up to now when there are dozens of specialized conferences, journals and projects based on neural networks. It turns out that a wide range of research and investment in neurocomputing may not correspond to the quality of achieved results. Only near future will again justify the vitality of neural network field.

## 1.2 Neurophysiological motivation

The original aim of neural network research represented the effort to understand and model how people think and how the human brain functions. The neurophysiological knowledge made the creation of simplified mathematical models possible which can be exploited in neurocomputing to solve practical tasks from artificial intelligence. This means that the neurophysiology serves here as a source of inspiration and the proposed neural network models are further expanded and refined regardless of whether they model the human brain or not. In spite of that it is helpful to turn back to this analogy for new inspiration or the metaphor can be exploited when describing the properties of a mathematical model.

Therefore it is useful to become familiar with basic neurophysiological knowledge which will help us to understand the original motivations of mathematical models of neural networks. We are not experts in this field and thus, our exposition will be quite superficial. Also the human understanding in this area is far from being complete. One neurophysiologist describes this state with the following hyperbole: 'Since our current understanding of the brain is only limited, everything that is said about the human brain may be considered to be true.' On the other hand we are not looking for an identical copy of the brain but we want to imitate its basic functions, in a similar way which airplanes share with birds and that is mainly the capability to fly.

The human *nervous system* (or generally the nervous system of living organisms) intermediates the relationships between the outward environment and the organism itself as well as among its parts to ensure the corresponding response to external stimuli and internal states of the organism, respectively. This process proceeds by transmitting impulses from particular sensors, so-called *receptors* which enable to receive mechanical, thermal, chemical, and luminous stimuli, to other nervous cells that process these signals and send them to corresponding executive organs, so-called *effectors*. These impulses passing through the *projection channels* where the information is preprocessed, compressed and filtered for the first time, possibly arrive at the *cortex* that is the top controlling center of the nervous system. On the brain surface about six primary, mutually interconnected *projection regions* corresponding approximately to senses may be distinguished where the parallel information processing is performed. The complex information processing which is the basis for a conscious controlling of effector activities, proceeds sequentially in so-called *associative regions*.

A so-called *neuron* is a nervous cell which is the basic functional building element of nervous system. Only the human cortex consists of approximately 13 to 15 billions of neurons which are arranged into a hierarchical structure of six different layers. Moreover, each neuron can be connected with about 5000 of other neurons. The neurons are autonomous cells that are specialized in transmission, processing, and storage of information which is essential for the realization of vital functions of the organism. The structure of a neuron is schematically depicted in Figure 1.1. The neuron is formed

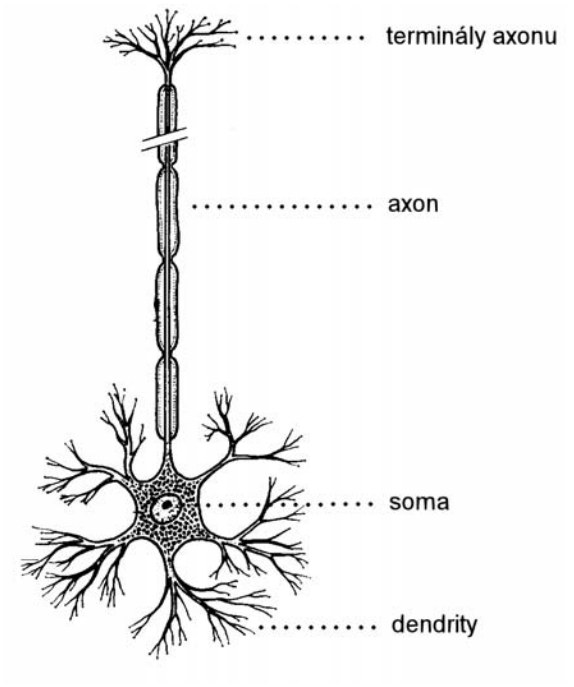

Figure 1.1: Biological neuron.

for signal transmission in such a way that, except its proper body, i. e. the so-called *soma*, it also has the input and output transfer channels, i. e. the *dendrites* and the *axon*, respectively. The axon is branched out into many, so-called *terminals* which are terminated by a membrane to contact the *thorns* of dendrites of other neurons as it is depicted in Figure 1.2. A (chemical) so-called *synapse* serves here as a unique interneuron interface to transfer the information. The degree of synaptic permeability bears all important knowledge during the whole life of the organism. From the functional point of view the synapses are classified in two types: the *excitatory* synapses which enable impulses in the nervous system to be spread and the *inhibitory* ones which cause their attenuation. A memory trace in the nervous system probably arises by encoding the synaptic bindings on the way between the receptor and effector.

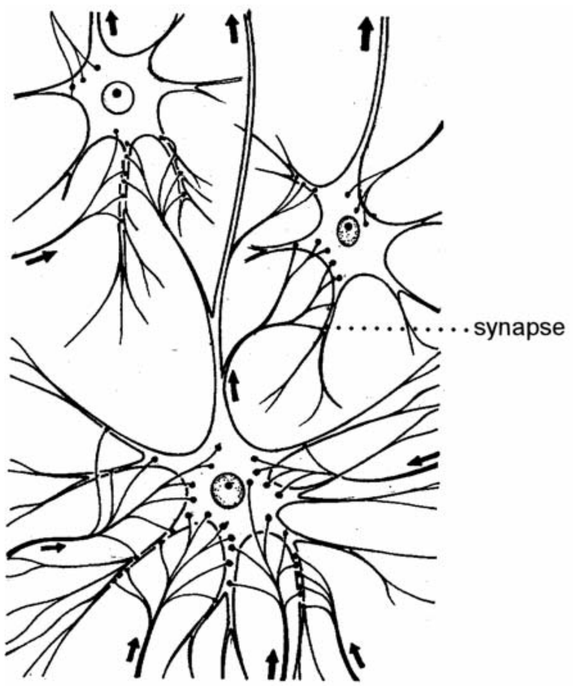

Figure 1.2: Biological neural network.

The information transfer is feasible since the soma and axon are covered with a membrane that is capable to generate an electric impulse under certain circumstances. This impulse is transmitted from the axon to dendrites through the synaptic gates whose permeabilities adjust the respective signal intesities. Then each postsynaptic neuron collects the incoming signals whose postsynaptic intensities, after being summed up, determine its excitation level. If a certain excitation boundary limit, i. e. a so-called *threshold*, is reached, this neuron itself produces an impulse (it fires) and thus, the further propagation of underlying information is ensured. During each signal transit the synaptic permeabilities as well as the thresholds are slightly adapted correspondingly to the signal intensity, e. g. either the firing threshold is being lowered if the transfer is frequent or it is being increased if the neuron has not been stimulated for a longer time. This represents the neuron plasticity, i. e. the neuron capability to learn and adapt to varying environment. Also the inter-neuron connections are subjected to this evolution process during the organism life. This means that during learning new memory traces are established or the synaptic links are broken in the course of forgetting.

The human nervous system has a very complex structure which is still being intensively investigated. However, the above-mentioned oversimplified neurophysiological principles will be sufficient to formulate a mathematical model of neural network.

## 1.3 Mathematical Model of Neural Network

### 1.3.1 Formal Neuron

A *formal neuron* which is obtained by re-formulating a simplified function of biological neuron into a mathematical formalism will be the basis of the mathematical model of neural network. Its schematic structure is shown in Figure 1.3. The formal neuron

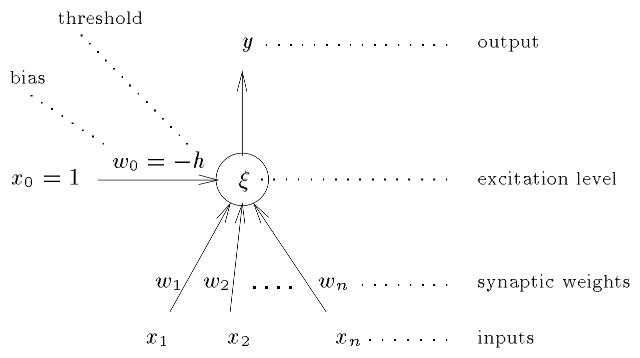

Figure 1.3: Formal neuron.

(shortly, neuron) has $n$, generally real, *inputs* $x_1, \ldots, x_n$ which model the signals coming from dendrites. The inputs are labeled with the corresponding, generally real, synaptic *weights* $w_1, \ldots, w_n$ which measure their permeabilities. According to the neurophysiological motivation some of these synaptic weights may be negative to express their inhibitory character. Then, the weighted sum of input values represents the *excitation level* of the neuron:

$$\xi = \sum_{i=1}^{n} w_i x_i \,. \tag{1.1}$$

The value of excitation level $\xi$, after reaching the so-called *threshold* $h$, induces the *output* $y$ (*state*) of the neuron which models the electric impulse generated by axon. The non-linear grow of output value $y = \sigma(\xi)$ after the threshold excitation level $h$ is achieved, is determined by the so-called *activation* (*transfer*, *squashing*) *function* $\sigma$. The simplest type of activation function is the *hard limiter* which is of the following form:

$$\sigma(\xi) = \begin{cases} 1 & \text{if } \xi \geq h \\ 0 & \text{if } \xi < h \,. \end{cases} \tag{1.2}$$

By a formal manipulation it can be achieved that the function $\sigma$ has zero threshold and the actual threshold with the opposite sign is understood as a further weight, so-called *bias* $w_0 = -h$ of additional formal input $x_0 = 1$ with constant unit value as it is depicted in Figure 1.3. Then, the mathematical formulation of neuron function is given by the following expression:

$$y = \sigma(\xi) = \begin{cases} 1 & \text{if } \xi \geq 0 \\ 0 & \text{if } \xi < 0 \end{cases} \qquad \text{where } \xi = \sum_{i=0}^{n} w_i x_i \,. \tag{1.3}$$

The geometrical interpretation depicted in Figure 1.4 may help us to better understand the function of a single neuron. The $n$ input values of a neuron may be

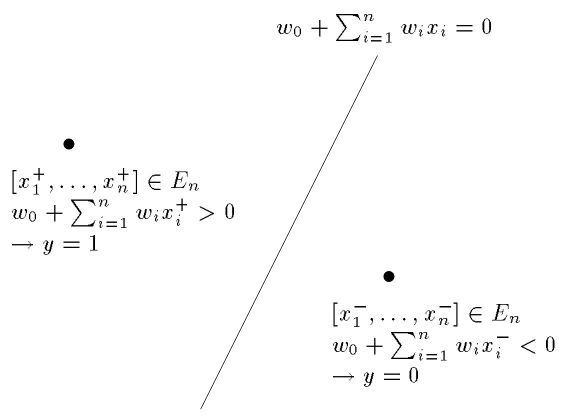
$w_0 + \sum_{i=1}^{n} w_i x_i = 0$
$[x_1^+, \ldots, x_n^+] \in E_n$
$w_0 + \sum_{i=1}^{n} w_i x_i^+ > 0$
$\rightarrow y = 1$
$[x_1^-, \ldots, x_n^-] \in E_n$
$w_0 + \sum_{i=1}^{n} w_i x_i^- < 0$
$\rightarrow y = 0$

Figure 1.4: Geometrical interpretation of neuron function.

interpreted as coordinates of a point in the $n$-dimensional Euclidean, so-called *input space* $E_n$. In this space the equation of a hyperplane (e. g. the straight line in $E_2$, the plane in $E_3$) has the following form:

$$w_0 + \sum_{i=1}^{n} w_i x_i = 0\,. \tag{1.4}$$

This hyperplane disjoins the input space into two halfspaces. The coordinates $[x_1^+, \ldots, x_n^+]$ of points that are situated in one halfspace, meet the following inequality:

$$w_0 + \sum_{i=1}^{n} w_i x_i^+ > 0\,. \tag{1.5}$$

The points $[x_1^-, \ldots, x_n^-]$ from the second, remaining halfspace fulfill the dual inequality with the opposite relation symbol:

$$w_0 + \sum_{i=1}^{n} w_i x_i^- < 0\,. \tag{1.6}$$

Hence, the synaptic weights $w_0, \ldots, w_n$ (including bias) of a neuron may be understood as coefficients of this hyperplane. Clearly, such a neuron classifies the points in the input space (the coordinates of these points represent the neuron inputs) to which from two halfspaces determined by the hyperplane they belong, i. e. the neuron realizes the so-called *dichotomy* of input space. More exactly, the neuron is *active* (i. e. the neuron state is $y = 1$) if its inputs meet condition (1.5) or (1.4), i. e. they represent the coordinates of a point that lies in the first halfspace or on the hyperplane. In the case that this point is located in the second halfspace, the inputs of neuron fulfill the condition (1.6) and the neuron is *passive* (i. e. the neuron state is $y = 0$).

The meaning of the neuron function is illustrated by a motivational metaphorical example from the area of pattern recognition. Consider that a first-former from elementary school learns now to read and recognize characters. For simplistic reasons suppose that one neuron in his brain is responsible for the dichotomy of characters "A" and "B" (compare with the exploitation of Rosenblatt's neurocomputer Mark I Perceptron in Section 1.1). The image of one character is displayed on the array of $n = k \times k$ pixels. The intensities of these image pixels expressed by $n$ reals represent the inputs of the underlying neuron. Thus, in our geometrical interpretation any character corresponds to a point in the $n$-dimensional input space of this neuron. Obviously, during his learning the scholar meets different occurrences of character "A" (e. g. sample character from reader, hand-writing of teacher, etc.) whose shapes may not always be identical but anyway they should be very similar (they represent the same character "A"). The points corresponding to these occurrences of character "A" should not be too distant from one another in the input space (e. g., measured by Euclidean metric) since the image patterns are similar. On the other hand a point that represents character "B" whose image sufficiently differs from the image of character "A" is somewhat distant from the points corresponding to character "A". In other words, the points that represent characters "A" and "B" form two separate clusters in the input space. The purpose of the underlying neuron is to separate these clusters by a hyperplane as it is depicted in Figure 1.5.

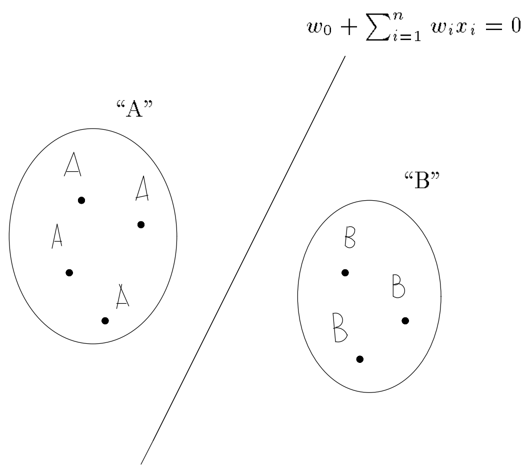

Figure 1.5: The separation of images "A" and "B" by a neuron.

In the above-mentioned metaphorical example the methodical difference in the concept representation in neural networks may be observed. The scholar does not need to remember particular images of all occurrences of a given character. It suffices the synaptic weights of the relevant neuron to be set so that the corresponding hyperplane separates the images of this character from that of remaining ones. At the beginning of school year the weights of the neuron that is responsible for the dichotomy of characters "A" and "B", are probably random since the scholar still cannot read. During his learning when he faces many sample characters the weights of this neuron are being step by step *adapted* so that the corresponding hyperplane begins to separate the cluster of points corresponding to character “A” from the cluster of points representing character “B”.

For example, suppose that now the scholar distinguishes between the sample characters “A” and “B” fairly well. At this stage of learning he meets rather experienced writing of his mother in which the image of character “A” is distorted in such a way that he first reads it as character “B”. In our geometrical interpretation this means that the relevant hyperplane associated with underlying neuron disjoins the input space in such a way that the point representing the deformed image of character “A” is wrongly located in the halfspace corresponding to character “B”. After wrong reading the scholar is corrected by his mother and he adjusts the respective synaptic weights $w_0, \dots w_n$ so that the hyperplane is turned slightly to include the new pattern into the right halfspace. This situation is depicted in Figure 1.6 where the new position of

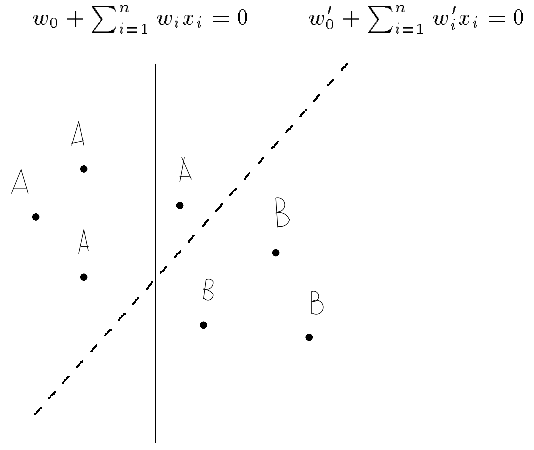
$w_0 + \sum_{i=1}^{n} w_i x_i = 0$
$w_0' + \sum_{i=1}^{n} w_i' x_i = 0$
A
A
A
A
B
B
B

Figure 1.6: The adaptation of weights when the image of “A” is misclassified.

hyperplane (represented by weights $w_0', \dots w_n'$) is marked with a dashed line. In this case, a supervision managed here by mother is needed to correct errors. Sometimes a negative experience itself stimulates learning when the correct object classification is a matter of surviving (e. g. a chicken must learn to differentiate a farmer who brings her feed from a predator who wants to destroy her). The adaptation of weights in the formal neuron models the change of synaptic permeabilities in the biological neuron as well as the rise of memory traces in the nervous system of a living organism.

The above-mentioned motivational example illustrates the basic principles of the mathematical model of a neuron. Obviously, one neuron can solve only very simple tasks. The exclusive disjunction XOR is a typical example of a logical function that cannot be implemented by one neuron (compare with the argument by Minsky and Papert against perceptron in Section 1.1). For instance, consider only two binary inputs (whose values are taken from the set $\{0, 1\}$) and one binary output whose value is 1 if and only if the value of exactly one input is 1 (i. e. $XOR(0,0) = 0$, $XOR(1,0) = 1$, $XOR(0,1) = 1$, $XOR(1,1) = 0$). It follows from Figure 1.7 where all possible inputs

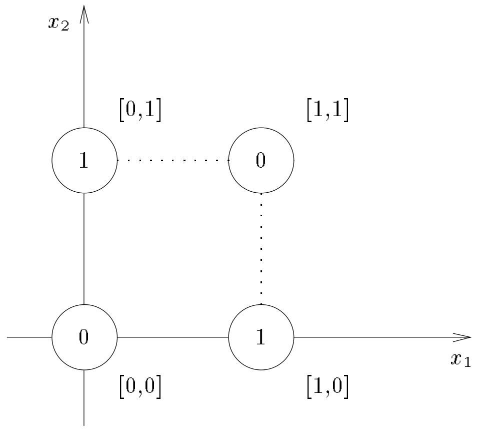

Figure 1.7: Geometrical representation of XOR function.

are depicted in the input space $E_2$ and labeled with the corresponding outputs that there is no hyperplane (straight line) to separate the points corresponding to output value 1 from the points associated with output value 0. This implies that the neurons need to be connected into a network in order to solve more complicated tasks as it is the case of the human nervous system.

### 1.3.2 Neural Network

A *neural network* consists of formal neurons which are connected in such a way that each neuron output further serves as the input of generally more neurons similarly as the axon terminals of a biological neuron are connected via synaptic bindings with dendrites of other neurons. The number of neurons and the way how they are interconnected determine the so-called *architecture* (*topology*) of neural network. Regarding their purpose, the *input*, *working* (*hidden*, *intermediate*), and *output* neurons may be distinguished in the network. By the oversimplified neurophysiological analogy, the input and output neurons represent the receptors and effectors, respectively, and the connected working neurons create the corresponding channels between them to propagate the respective signals. These channels are called paths in the mathematical model. The signal propagation and information processing along a network path is realized by changing the states of neurons on this path. The states of all neurons in the network form the *state* of the neural network and the synaptic weights associated with all connections represent the so-called *configuration* of the neural network.

The neural network evolves in time, namely, the interconnections as well as the neuron states are being changed, and the weights are being adapted. In the context of updating these network attributes in time, it is useful to split the global *dynamics* of neural network into three dynamics and consider three *modes* (*phases*) of network operation: *architectural* (topology change), *computational* (state change), and *adaptive* (configuration change). This classification does not correspond to neurophysiological reality since in the nervous system all respective changes proceed simultaneously. The above-introduced dynamics of neural network are usually specified by an initial condition and by a mathematical equation or rule that determines the evolution of a particular network characteristic (topology, state, configuration) in time. The updates controlled by these rules are performed in the corresponding operational modes of neural network.

By a concretization of the introduced dynamics various *models* of neural networks are obtained which are suitable to solve specific tasks. This means that in order to specify a particular neural network model it suffices to define its architectural, computational and adaptive dynamics. In the following exposition general principles and various types of these three dynamics are described which represent the basis for the taxonomy of neural network models. Also several typical examples are mentioned which will later help us to describe the well-known neural network models.

## Architectural dynamics

The architectural dynamics specifies the network topology and its possible change. The architecture update usually applies within the framework of an adaptive mode in such a way that the network is supplied with additional neurons and connections when it is needed. However, in most cases the architectural dynamics assumes a fixed neural network topology which is not changed anymore.

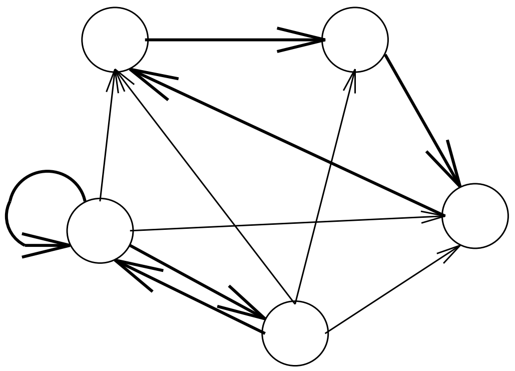

Figure 1.8: Example of cyclic architecture.

Basically, two types of architectures are distinguished: *cyclic* (*recurrent*) and *acyclic* (*feedforward*) network. In the cyclic topology there exists a group of neurons in the network which are connected into a ring, so-called *cycle*. This means that in this group of neurons the output of the first neuron represents the input of the second neuron whose output is again the input for the third neuron, etc. as far as the output of the last neuron in this group is the input of the first neuron. The simplest cycle is a *feedback* of the neuron whose output serves simultaneously as its input. The maximum number of cycles is contained in the so-called *complete topology* in which the output of each neuron represents the input for all neurons. An example of a general cyclic neural network is depicted in Figure 1.8 where all the cycles are indicated. On the contrary the feedforward neural networks do not contain any cycle and all paths lead in one direction. An example of an acyclic neural network is in Figure 1.9 where the longest path is marked.

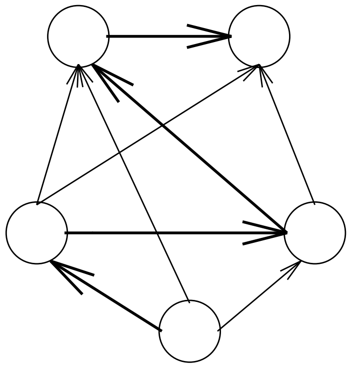

Figure 1.9: Example of acyclic architecture.

The neurons in the feedforward networks can always be disjointly split into so-called *layers* which are ordered (e. g. arranged one over another) so that the connections among neurons lead only from lower layers to upper ones and generally, they may skip one or more layers. Especially, in a so-called *multilayered neural network*, the zero (lower), so-called *input* layer consists of input neurons while the last (upper), so-called *output* layer is composed of output neurons. The remaining, so-called *hidden* (*intermediate*) layers contain hidden neurons. As it has been already suggested the layers are counted starting from zero that corresponds to the input layer which is then not included in the number of network layers (e. g. a two-layered neural network consists of input, one hidden, and output layer). In the topology of a multilayered network each neuron in one layer is connected to all neurons in the next layer (possibly missing connections between two consecutive layers might be implicitly interpreted as connections with zero weights). Therefore, the multilayered architecture can be specified only by the numbers of neurons in particular layers, typically hyphened in the order from input to output layer. Also any path in such a network leads from the input layer to the output one while containing exactly one neuron from each layer. An example of a three-layered neural network 3–4–3–2 with an indicated path is in Figures 1.10 which, besides the input and output layers, is composed of two hidden layers.

## Computational dynamics

The computational dynamics specifies the network *initial state* and a rule for its updates in time providing that the network topology and configuration are fixed. At the

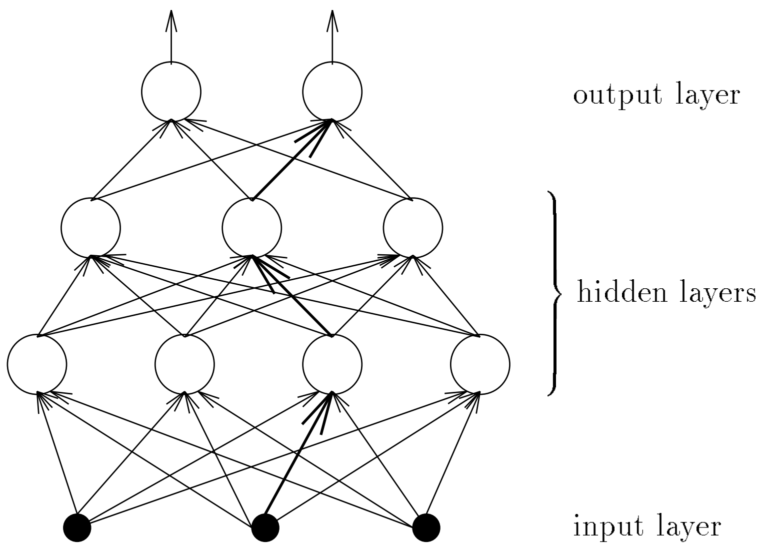

Figure 1.10: Example of architecture of multilayered neural network 3–4–3–2.

beginning of computational mode the states of input neurons are assigned to the so-called *network input* and the remaining neurons find themselves in the above-mentioned initial state. All potential network inputs and states form the so-called *input* and *state space* of neural network, respectively. After initializing the network state, a proper computation is performed. Generally, a continuous-time evolution of neural network state is considered in the so-called *continuous model* in which the network state is a (continuous) function of time that is usually described by a differential equation within computational dynamics. However, in most cases a discrete computational time is assumed, i. e. at the beginning the network finds itself at time 0 and the network state is updated only at time $1, 2, 3, \ldots$. At each such time step one neuron (during so-called *sequential* computation) or more neurons (during so-called *parallel* computation) are selected according to a given rule of computational dynamics. Then, each of these neurons collects its inputs, i. e. the outputs of incident neurons, and *updates* (changes) its state with respect to them. According to whether the neurons change their states independently on each other or their updating is centrally controlled, the *asynchronous* and *synchronous* models of neural networks, respectively, are distinguished. The states of output neurons which are generally being varied in time, represent the *output* of neural network, i. e. the result of computation. Usually, the computational dynamics is considered so that the network output is constant after a while and thus, the neural network, under computational mode, implements a function in the input space, i. e. for each network input exactly one output is computed. This so-called *neural network function* is specified by the computational dynamics whose equations are parametrized by the topology and configuration that are fixed during the computational mode. Obviously, the neural network is exploited for proper computations in the computational mode.

The computational dynamics also determines the function of particular neurons whose formal form (mathematical formula) is usually the same for all (non-input) neurons in the network (so-called *homogeneous* neural network). By now, we have considered only the function given by equation (1.3) which has been inspired by a biological neuron operation. However, in neural network models various neuron functions are in general use which may not correspond to any neurophysiological pattern but they are designed only by mathematical invention or even motivated by other theories (e. g. physics). For example, instead of weighted sum (1.1) a polynomial in several indeterminates (inputs) is exploited in so-called *higher-order neural networks*. Or sometimes the excitation level corresponds formally to the distance between the input and respective weight vector, etc. Also the transfer function is often approximated by a continuous (or differentiable) activation function or replaced by a completely different function. For example, the so-called *sigmoid activation functions* create a special class of transfer functions. This class includes *hard limiter* (1.7), *piecewise-linear (saturated-linear) function* (1.8), *standard sigmoid (logistic function)* (1.9), *hyperbolic tangent* (1.10), etc.

$$\sigma(\xi) = \begin{cases} 1 & \xi \geq 0 \\ 0 & \xi < 0 \end{cases} \quad \textit{hard limiter} \tag{1.7}$$

$$\sigma(\xi) = \begin{cases} 1 & \xi > 1 \\ \xi & 0 \leq \xi \leq 1 \\ 0 & \xi < 0 \end{cases} \quad \textit{piecewise-linear function} \tag{1.8}$$

$$\sigma(\xi) = \frac{1}{1+e^{-\xi}} \quad \textit{standard (logistic) sigmoid} \tag{1.9}$$

$$\sigma(\xi) = \tanh\left(\frac{1}{2}\xi\right) = \frac{1-e^{-\xi}}{1+e^{-\xi}} \quad \textit{hyperbolic tangent} \tag{1.10}$$

The graphs of these sigmoid functions are drawn in Figure 1.11. Depending on whether the neuron function is discrete or continuous, the *discrete* and *analog* models of neural networks, respectively, are distinguished.

The above-mentioned principles will be illustrated through an example of a neural network whose computational dynamics will be described in more detail and also its geometrical interpretation will be outlined similarly to the neuron in Paragraph 1.3.1. A fixed architecture of a multilayered neural network is considered (see Figure 1.10). Further, suppose that each connection in this network is labeled with synaptic weights, i. e. a configuration of the multilayered network is given. At the beginning, the states of input layer neurons are assigned to a generally real network input and the remaining (hidden and output) neurons are passive. The computation further proceeds at discrete time steps. At time 1 the states of neurons from the first (hidden) layer are updated according to equation (1.3). This means that a neuron from this layer collects its inputs from the input neurons, computes its excitation level as a weighted sum of these inputs and its state (output) determines from the sign of this sum by applying the transfer function (hard limiter). Then at time 2 the states of neurons from the second (hidden) layer are updated again according to equation (1.3). In this case, however, the outputs of neurons from the first layer represent the inputs for neurons in the second layer. Similarly, at time 3 the states of neurons in the third layer are updated, etc. Thus, the

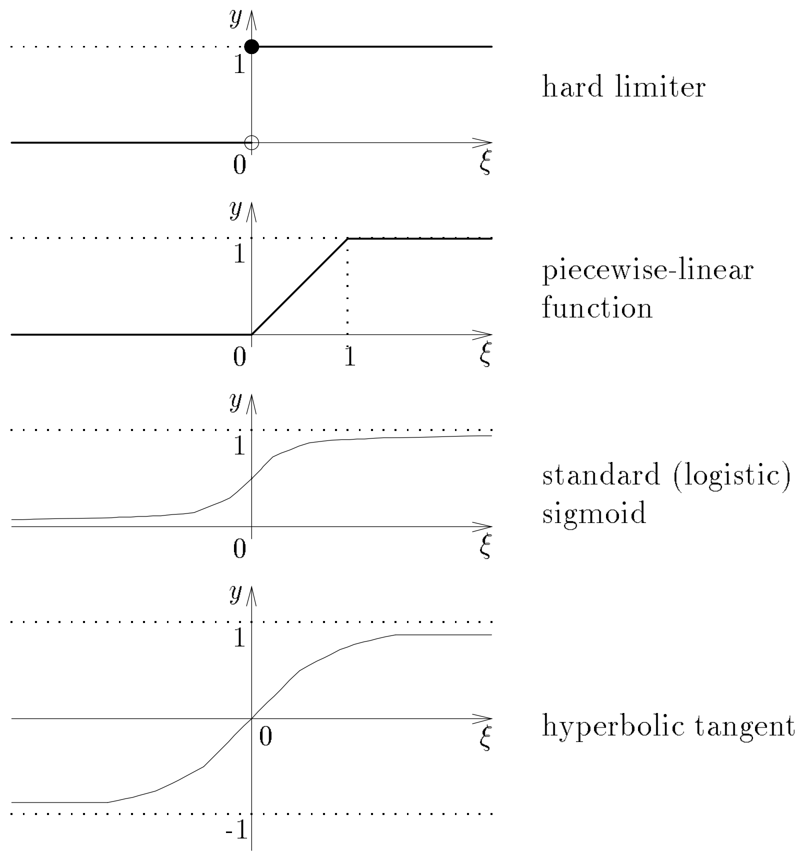

Figure 1.11: Graphs of sigmoid activation function.

computation proceeds in the direction from the input layer to the output one so that at each time step all neurons from the respective layer are updated in parallel on the basis of inputs collected from the preceding layer. Finally, the states of neurons in the output layer are determined which form the network output, and the computation of multilayered neural network terminates.

We will try to generalize our geometrical interpretation of neuron function (see Figure 1.4) for the function of three-layered neural network. Clearly, the neurons in the first (hidden) layer disjoin the network input space by corresponding hyperplanes into various halfspaces similarly as in Paragraph 1.3.1. Hence, the number of these halfspaces equals the number of neurons in the first layer. Then, the neurons in the second layer can, for example, classify the intersections of some of these halfspaces, i. e. they can represent convex regions in the input space. This means that a neuron from the second layer is active if and only if the network input corresponds to a point in the input space that is located simultaneously in all halfspaces which are classified by selected neurons from the first layer. Although the remaining neurons from the first layer are formally connected to this neuron in the topology of multilayered network, however, the corresponding weights are zero and hence, they do not influence the underlying neuron. In this case, the neurons in the second layer realize logical conjunction ($AND$) of relevant inputs, i. e. each of them is active if and only if all its inputs are 1. In Figure 1.12 the partition of input space into four halfspaces $P_1$, $P_2$, $P_3$, $P_4$ by four

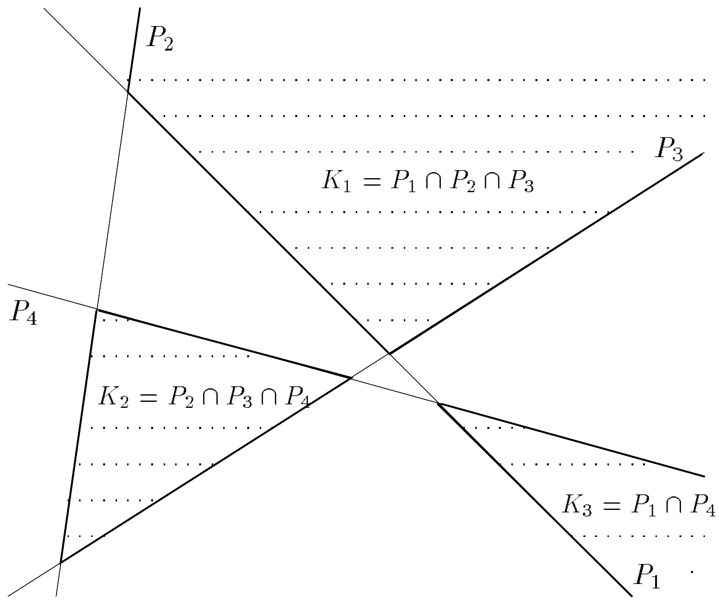

Figure 1.12: Example of convex region separation.

neurons from the first layer is depicted (compare with the example of multilayered architecture 3–4–3–2 in Figure 1.10). Three convex regions are marked here that are the intersections of halfspaces $K_1 = P_1 \cap P_2 \cap P_3$, $K_2 = P_2 \cap P_3 \cap P_4$, and $K_3 = P_1 \cap P_4$. They correspond to three neurons from the second layer, each of them being active if and only if the network input is from the respective region.

The partition of the input space into convex regions can be exploited for the pattern recognition of more characters (compare with the dichotomy of two character images from the motivational example in Figure 1.5) where each character is associated with one convex region. Sometimes the part of the input space corresponding to a particular image cannot be closed into a convex region. However, a non-convex area can be obtained by a union of convex regions which is implemented by a neuron from the third (output) layer. This means that the output neuron is active if and only if the network input represents a point in the input space that is located in at least one of the selected convex regions which are classified by neurons from the second layer. In this case, the neurons in the output layer realize logical disjunction ($OR$) of relevant inputs, i. e. each of them is active if and only if at least one of its inputs is 1. For example, in Figure 1.12 the output neuron is active if and only if the network input belongs to the region $K_1$ or $K_2$ (i. e. $K_1 \cup K_2$). It is clear that the non-convex regions can generally be classified by three-layered neural networks.

In the preceding considerations the fact that the logical conjunction ($AND$) and disjunction ($OR$), respectively, are computable by one neuron with the computational dynamics (1.3), has been exploited. In Figure 1.13 the neuron implementation of these

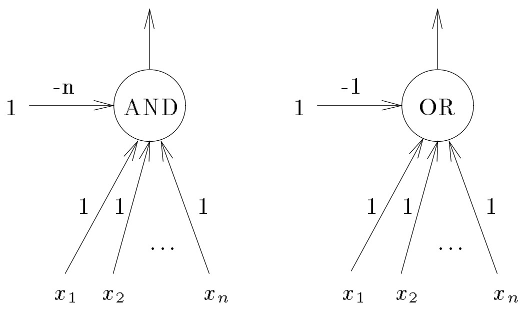

Figure 1.13: Neuron implementation of $AND$ and $OR$.

functions is depicted. The unit weights (excluding bias) ensure the weighted sum of actual binary inputs (taken from the set $\{0, 1\}$) to equal the number of 1's in the input. The threshold (the bias with the opposite sign) $n$ for $AND$ and 1 for $OR$ function causes the neuron to be active if and only if this number is at least $n$ or 1, respectively. Of course, the neurons in the upper layers of a multilayered network may generally compute other functions than only a logical conjunction or disjunction. Therefore, the class of functions computable by multilayered neural networks is richer than it has been assumed in our example.

In addition, the geometrical interpretation of the multilayered neural network will be illustrated by the above-mentioned important example of logical function, the

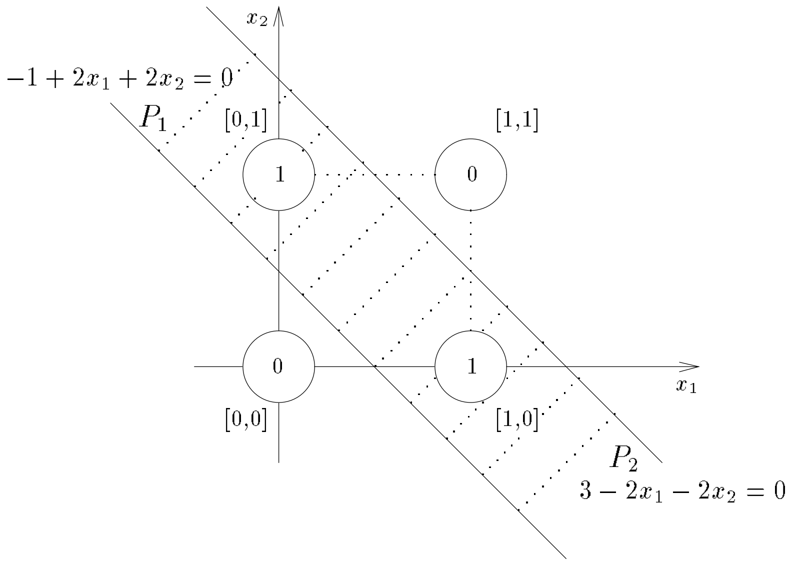

Figure 1.14: Geometrical interpretation of $XOR$ computation by two-layered network.

exclusive disjunction ($XOR$). We know from Paragraph 1.3.1 that this function is not computable by one neuron with the computational dynamics (1.3). As it can be seen in Figure 1.14 (compare with the note to Figure 1.7), the (two-dimensional) network inputs for which the output value of $XOR$ function is 1, can be closed by the intersection of two halfspaces (half-planes) $P_1$, $P_2$ bounded by hyperplanes (straight lines), into a convex region. Therefore, the $XOR$ function can be implemented by the two-layered neural network 2–2–1 (with one hidden layer) which is depicted in Figure 1.15.

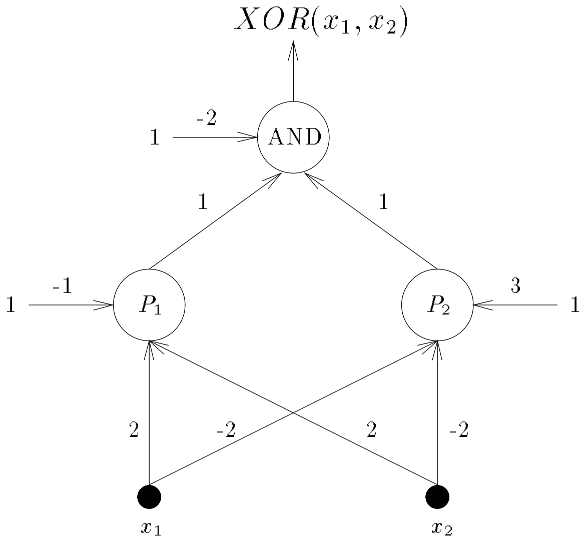

Figure 1.15: Two-layered network for $XOR$ function computation.

## Adaptive dynamics

The adaptive dynamics specifies the network *initial configuration* and the way how the weights in the network are being adapted in time. All potential network configurations form the so-called *weight space* of neural network. At the beginning of adaptive mode the weights of all network connections are assigned to the initial configuration (e. g. randomly). After initializing the network configuration, the proper adaptation is performed. Similarly as for the computational dynamics, a model with a continuous-time evolution of neural network weights when the configuration is a (continuous) function of time usually described by a differential equation, may generally be considered. However, in most cases a discrete adaptation time is assumed (compare with computational dynamics).

As we know the network function in the computational mode depends on configuration. The aim of adaptation is to find such a network configuration in the weight space that realizes a desired function in the computational mode. The computational mode is exploited for the respective network function computations while the adaptive mode serves for *learning* ('programming') this function. There exist hundreds of successful *learning algorithms* for various neural network models. For example, the most well-known and widely-applied learning algorithm is the backpropagation heuristics for multilayered neural network whose significance has been explained in Section 1.1 and its description can be found in Section 2.2. The neural network learning represents mostly a complex non-linear optimization problem whose solving can be very time-consuming (tens of hours or days of PC computation) even for small tasks.

The desired network function is usually specified by a *training set* (*sequence*) of pairs composed of the network sample input and the corresponding desired output which are called *training patterns*. This is illustrated by the above-mentioned motivational example (see Paragraph 1.3.1) in which the sample images of characters (i. e. arrays of $n = k \times k$ pixels) may represent the inputs of training patterns to which the corresponding letters as the desired outputs are assigned. For example, the state of the output neuron that classifies character "A" is required to be 1 if and only if a sample of this character image is at the input, and the remaining output neurons are desired to be passive in this case. The way how the network function is described by a training set, models a teacher (supervisor) who informs the adaptive mechanism about the correct network output corresponding to a given sample network input. Therefore, this type of adaptation is called *supervised learning*. Sometimes, instead of giving the desired network output value associated with a given sample stimulus (input), a teacher evaluates the quality of actual current responses (outputs) by a mark. This is called *graded* (*reinforcement*) *learning*. The examples of models that employ supervised learning will be described in Chapter 2 (e. g. a typical representative of this class is already mentioned backpropagation algorithm).

A different type of adaptation is a so-called *self-organization* that models the situation when a teacher is not available. In this case the training set contains only sample inputs and the neural network itself organizes the training patterns (e. g. into clusters) and discovers their global features.

## 1.4 Position of neural networks in computer science

### 1.4.1 Neural networks and von Neumann computer architecture

It may not be clear from the preceding exposition what the proper contribution of neural networks to computer science is. First consider that a neural network can, in principle, implement any desired function in the computational mode. In a certain sense neural networks represent a universal computational device (for a more exact formulation see the second part of this book) and thus, they have the same computational power as the classical von Neumann computer architecture (e. g. everything computable on PC can, in principle, be computed by a neural network, and vice versa). Considering that there exist hundreds of various (including very exotic) universal computational models, this property of neural networks is not exceptional. In addition, the network function is specified by a great number of weight parameters and it is not clear how a desired task would be programmed under this model.

The main advantage and the distinctness of neural networks from the classical von Neumann computer architecture is their capability to learn. The desired network function is not programmed in such a way that an exact procedure for its evaluation is described, but the network itself abstracts and generalizes the functional characteristics from training examples in the adaptive mode during the learning process. In this sense, the network evokes the intelligence of a man who acquires a lot of his knowledge and skills from experience and, in most cases, he is unable to formulate it analytically by using exact rules or algorithms. In the sequel, several motivational (slightly exaggerated) examples will be outlined to help us to better understand this phenomenon.

Consider a bricklayer who wants to train his apprentice to plaster a wall. You, who have experienced plastering your own house, probably know that first attempts are not too successful (a half of mortar usually ends up on the ground). How grotesque a theoretical schooling would be if the bricklayer would write differential equations describing an ideal trajectory and speed of hand moves during plastering, on a blackboard in front of the startled apprentice. Even if he had a background in differential calculus he would not learn how to plaster a wall this way. This skill can be acquired by observing an experienced bricklayer during plastering and by his own attempts supervised by a teacher.

Also in the above-mentioned example of a scholar who learns how to read (see Section 1.3), the analytical description of ideal letter shapes would not help him too much. It is important for him to read as many examples as possible of simple sentences from a spelling-book under suitable supervision and the sentences written by a teacher with different hand-writing on a blackboard. In addition, the correct letter recognition does not guarantee the correct word reading as it is illustrated through an example of two English words *THE* and *TEA* in Figure 1.16 where the deformed shape of letter

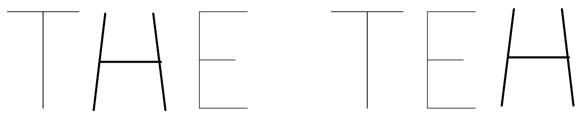

Figure 1.16: Differentiating between letters "H" and "A" by context.

"H" is identical with the distorted shape of letter "A". The context of particular letters or even the word meaning is essential for correct reading (compare with the system NETtalk mentioned in Section 1.1 and Paragraph 1.4.2).

The next demonstrational example which is described in literature [77] is a broomstick balancer. A neural network has been constructed to control the broomstick in a vertical position. A special cart was used during the experiment to which the broomstick was freely attached so that it would fall (for the simplicity in one vertical plane) without catching it. The neural network was trained to determine the cart moves (left or right) while keeping the broomstick from falling. At first the broomstick deviation (angle) from the vertical position was used and later even a filter image of broomstick was scanned by a camera for this purpose. The training patterns for the network adaptation consisted of the sample input corresponding to a filter image of broomstick and the desired output determining the correct cart move for the respective broomstick position. These patterns were obtained from a demonstrator (during a slow-motion computer simulation) who had been moving the cart for some time to keep the broomstick in the vertical position. After the network had been trained, it successfully took on his task to control already the real cart. Also here the differential equations for cart moves can theoretically be formed, however, before a classical computer of von Neumann architecture would solve them, the broomstick might probably fall down. On the other hand, in this simple case (the variant with the input angle) there exists a successful system based on the classical control theory.

A similar example described in literature is the control of a raw material inflow in a complex manufacturing process where an analytical model is practically impossible to be formed. In practice this is done by an experienced worker who regulates the inflow of particular materials by several control sticks using the information read from various measuring clock-faces. This well-paid expert, besides he would not be willing to share his long-time acquired experience in his own interests, he would not even be able to express it by means of exact rules for the control stick moves. Also here a neural network was attached that has learned to regulate the material inflow using examples of measuring clock-face states and the corresponding expert's responses. Finally, the expert lost his job.

It follows from the above-mentioned illustrations that a neural network models the human ability to learn the knowledge and skills from examples which cannot be managed algorithmically using classical computers of von Neumann architecture since the respective analytical descriptions are not known or their analysis is too complicated. This corresponds to the application areas of neural networks (see Paragraph 1.4.2) where the conventional computers fail. Obviously, it is insufficient to memorize all sample examples (training set), e. g. as a table stored in the conventional computer memory. In addition, the relevant rules should be *generalized* in order to solve similar cases which have not been met by a neural network during learning. For example, in the case of character recognition it is impossible to store all potential shapes of letter images.

The importance of generalization capability of human intelligence can be illustrated by student's preparation for an exam in mathematics. It is clear that memorizing all sample exercises from a textbook by heart without understanding the methods to solve them, does not guarantee the success. In this case, the student probably would not pass the exam unless the test includes only the identical exercises from the textbook because he would be unable to derive a solution even for similar tasks. Obviously, it is insufficient to memorize the sample exercises by heart, however, it is necessary to know how to generalize the rules for their solving.

The ability to learn and generalize is a typical feature of human intelligence. However, the evaluation of the generalization capability of a neural network represents a great problem since it is not clear how to define the correct generalization. This issue can be demonstrated by a simple task from an IQ test in which a subsequent member of numerical sequence $1, 2, 3, \ldots$ should be added. Most people would probably add the next number which is 4. However, consider a mathematician who has noticed that the number 3 can be viewed as a sum of two preceding numbers 1 and 2. Thus, according to this more complicated coherency he supplies with 5 which is again a sum of two preceding numbers 2 and 3. Besides that some 'normal' people would consider him to be strange, it is not obvious at all which of these two completions is a correct generalization of the rule in this sequence. Clearly, there are infinitely many such completions that can somehow be justified.

Since we are unable to define (formalize), and thus to measure the generalization capability of neural networks, a basic criterion is missing (except practical experiments) that would decide which neural network models are good or better than others in a particular case, etc. The generalization capabilities of designed neural network models are often illustrated by isolated examples which exhibit good properties (perhaps thanks to their appropriate choice). However, these properties cannot be formally verified (proved). In our opinion this situation is one of the reasons for the crisis in the theory of neural networks.

On the other hand, the neural networks have been successfully applied to solve important practical problems where the conventional computers failed. Moreover, the simulations of (very simplified) models of biological neural networks exhibit phenomena similar to human intelligence. This probably means that these models share certain features that are important to resemble the intelligent human activities and which are missing in von Neumann computer architecture. From the computational point of view, the biological nervous system can basically be characterized as a densely interconnected network of a great number of simple computational units (neurons) which compute only simple functions. Within mathematical models of neural networks, this framework probably creates a computational paradigm sufficient to emulate an intelligent behavior.

The systematic logic and accuracy of classical computers is replaced by association and vagueness in neural networks when an associated (similar) sample pattern is 'recalled' to a new problem to derive its analogous solution from it. Also instead of explicit data representation employed in conventional computer memory, the information in neural networks is encoded implicitly. This means that, besides the inputs and outputs, the exact purpose for particular numerical parameters in neural networks cannot be easily identified. While the classical computers are sensitive to errors and a one-bit change may cause the global system crash, the neural networks are robust. For example, it is known that after a neurosurgical operation when a part of the patient's brain tissue is taken away, the patient temporarily forgets some abilities (e. g. to speak) or partially loses the dexterity in them (e. g. he stammers). Later on, these abilities are revived or improved after additional training because other neurons take on the task of the original ones. This phenomenon can also be observed in neural network models when the network may not necessarily lose its functionality after removing several neurons but, perhaps, only the accuracy of resulting responses is affected. Furthermore, in von Neumann computer architecture the sequential run of a program is localized, e. g. by a program counter. In contrast to this, the computation is distributed throughout the neural network and it is naturally parallel.

When comparing the models of neural networks to the classical von Neumann computer architecture the clash of two different intelligences can be observed, namely biological and silicon ones. The way out of this situation which can find a wider application in current overtechnologized world is the symbiosis of both approaches. The idea to create a computer in the image of man is still attractive.

### 1.4.2 Application of Neural Networks

The comparison of neural network models to the von Neumann computer architecture suggests potential areas of their application where the conventional computers fail. In particular, this is the question of problems for which no algorithms are known or their analytical description is too complicated for computer processing. Typically, neural networks may be employed in cases where example data is available which sufficiently covers the underlying problem domain. Of course, the advantages of neural networks over the classical computers do not mean that neural networks can be substituted for current computers. Particularly, in the case of mechanical computations (e. g. multiplication) which can be algorithmically described, neural networks (similarly as people) cannot compete with conventional computers for speed and accuracy. Only neural networks in the form of specialized modules will probably enrich the von Neumann computer architecture. In the following exposition several potential fields of neural network applications will be mentioned.

The neural networks can naturally be exploited for *pattern recognition*. A special case of this task is the recognition of (scanned) hand-written or printed characters (digits, letters, etc.) which has accompanied us as a motivational example throughout the preceding exposition. In this case a character image is first separated (e. g. by using a conventional computer) from the neighbouring text (e. g. the limit points of image are determined) and then, it is normalized, i. e. projected on the standardized array of (e. g. $15 \times 10 = 150$) pixels. These pixels correspond to the inputs of a neural network so that a particular input neuron is active if and only if a line from the character image crosses the associated pixel. Each output neuron in the network represents a potential character and it is active if and only if this character is being recognized at the input. This means that only one output neuron at each moment should be active for an unambiguous recognition. For example, a training set can be created for this purpose by rewriting some text that has already been available in a computer, in the same way which will be used during the respective recognition process, e. g. by a particular type of hand-writing. The rewritten character images represent the training sample inputs while the original computer text is used to label them with the corresponding desired network outputs, i. e. with the identified characters. Then, the neural networks can be adapted to this training task so that it is able to recognize the underlying characters. This way the 95% character recognition reliability can be achieved in a relatively short time. A similar approach can be exploited e. g. in robotics for image information processing, or for satellite picture evaluation etc.

Another potential neural network application field is the *control* of complex devices in a dynamically varying environment. In Paragraph 1.4.1 two motivational examples from this area have been introduced, namely a broomstick balancer and the regulation of the raw material inflow in a complex manufacturing process. Another demonstrational example of a control system which is described in literature [72] is an automobile autopilot. A car in a computer simulation was driven by a neural network on a two-lane freeway roadbed together with other cars moving in the same direction. The neural network controlled the speed and the lane change of the car on the basis of the distance and speed of neighbouring cars in both lanes. In addition, the neural network handled the steering wheel according to a road curvature and its current direction angle. It is interesting that the neural network, besides successful car control (without collisions) including overtaking other cars, learned even different driving habits and style (e. g. risking fast drive and frequent passing or on the contrary, careful slow drive) according to training drivers from whom the training patterns were collected.

Yet another important application area of neural networks is *prediction* and as the case may be, the subsequent *decision-making*. Typical examples from this field include weather forecast, the share index development at the stock exchange, electrical energy consummation load, etc. For instance, in a meteorological application the basic parameter readings in time (e. g. temperature, pressure, etc.) represent the neural network input and the actual successive weather development serves as a teacher. There were some experiments done in which the weather forecast by a neural network for several days was better than that by meteorologists in some cases.

Further possible neural network exploitation is *data compression*, e. g. in television signal transmission, telecommunication, etc. For this purpose, an interesting technique [14] has been developed in which a two-layered neural network $n$–$n/4$–$n$ depicted in Figure 1.17 is employed. The number of hidden neurons in the intermediate

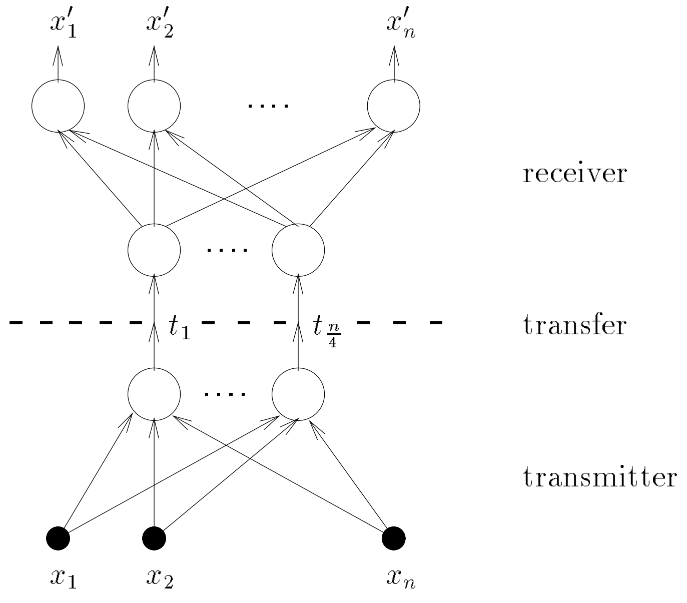

Figure 1.17: Transmitting signal compression by two-layered network $n$–$n/4$–$n$.

layer of this network is considerably (e. g. four times) less than the number of inputs or outputs which represent the same image signal. The two-layered neural network has learned the image patterns so that the training sample input and the respective desired output correspond to an identical image and the network responds with the output approximately equal to a given image input. During the proper transmission the hidden neuron states $t_1, \ldots, t_{\frac{n}{4}}$ are first computed from a given input image signal $x_1, \ldots, x_n$ at the transmitter. Then this four times compressed image is transferred via an information channel to a receiver which decodes it by computing the output neuron states $x'_1, \ldots, x'_n$. In this way, almost an identical original image is obtained. It has been shown during the respective experiment that the transmission quality (comparable with other data compression methods) depends on whether the transmitting images are similar to the training patterns to which the network had been adapted.

*Signal transformation* represents a further application domain of neural networks which includes the well-known NETtalk system [71] mentioned in Section 1.1 which transforms the English written text into spoken English. This system is again based on two-layered network 203–80–26 with $7 \times 29$ input neurons for encoding the seven-letter context of written text. For each of these seven characters, 29 input neurons are reserved that correspond to 26 letters from the English alphabet, comma, period, and space, from which exactly one neuron is active when the respective character occurs. The intermediate layer consists of 80 hidden neurons and 26 output neurons represent phonemes of the spoken signal. The network operation is demonstrated in Figure 1.18 where the input text is, letter by letter, moved from right to left while only such an output neuron is active at each moment that is associated with a phoneme

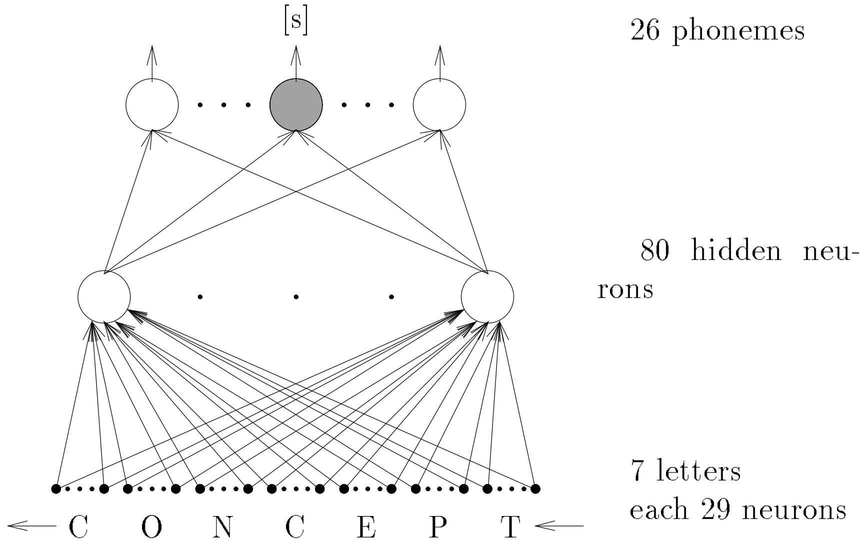

Figure 1.18: NETtalk.

corresponding to the middle letter of the seven-character input context. In our example, the middle letter "C" in the English word *CONCEPT* with pronunciation [konsept] is being read and the respective phoneme is [s]. However, the same letter "C" at the beginning of this word was assigned to phoneme [k] within the previous context. The successful NETtalk implementation lead to an effort to construct a neural network system with the inverse function transforming the spoken language into a written form (phonetic typewriter).

Another example of the neural network usage is *signal analysis*, e. g. ECG, EEG, etc. A continuous-time signal is sampled in equidistant time intervals and several last discrete values of signal level serve as an input, e. g. for a two-layered neural network. The neural network, after being trained, is able to identify a specific signal pattern that is important for a diagnostics. For example, a neural network 40–17–1 has been used to classify EEG signals with specific $\alpha$-rhythms [59].

*Expert systems* illustrate a further application area of neural networks. The main problem of the classical expert systems based on rules is creating a knowledge base which is a very hard, time-consuming human task with an uncertain result. Neural networks represent an alternative solution to this problem when the knowledge base representation is produced automatically by learning from example inferences. In this case, the computational mode of the neural network stands for inference engine operation. However, the implicit knowledge representation in neural networks complicates incomplete information processing and it does not provide an explanation of inference results, which are the necessary properties for any applicable expert system in practice. This problem is partially solved in a universal neural expert system *EXPSYS* [74]. This system enriches a multilayered network with interval arithmetic to cope with imprecise information as well as it is supplied with heuristics that analyzes the network function to provide a simple inference explanation. The system EXPSYS has been successfully applied in medicine and energetics. For example, in a medical application, the encoded symptoms and test results serve as a neural network input while the corresponding diagnosis and recommended treatment represent its output. In this case, a training set can be formed from the clinical record about patients.

### 1.4.3 Neural Network Implementation and Neurocomputers

The different neural network architecture requires a special hardware implementation. In this context, we speak about so-called *neurocomputers*. However, since the conventional computers are widely spread and there are some problems concerning the hardware realization of neural networks, meanwhile, the simplest and most frequent neural network implementation is a so-called *netware*. The netware is a software for classical computers (e. g. PC) simulating a neural network operation. This includes mostly demonstrational programs with impressive user-interface that simulate the well-known neural network models on simple examples. In some more elaborated programs, a user can select his own computational and adaptive dynamics which enable the accommodation of a neural network to a given practical problem or to justify the applicability of a new proposed model. There exist even programming languages (and their compilers) that support the program implementation of neural algorithms. An example of such programming language is *AXON* [26] based on the *C* language syntax. More advanced netware usually supports the exploitation of specialized coprocessors which, e. g. being plugged into a PC, implement the neuron operations effectively and speed up the time-consuming learning.

The proper neurocomputers do not mostly run independently but they are connected to host conventional computers which may realize e. g. a user-interface. The reason for this is that the neurocomputers are not used as universal computers but they usually operate as specialized devices to solve specific tasks (see Paragraph 1.4.2). Small neurocomputers are directly attached to a host-computer memory bus while bigger ones can be connected to a local area network. According to the way how neural network parameters are updated in time, the *continuous* and *discrete* neurocomputers are distinguished. Similarly, the *analog*, *digital*, and *hybrid* (the combination of analog and digital) neurocomputers are differentiated with respect to a numerical parameter representation. Rarely, one neuron in the implemented network corresponds to one processor of a neurocomputer, i. e. the so-called fully-implemented neurocomputer, which may be exploited for very fast real-time computations. Mostly, the so-called *virtual* neurocomputers, in which one processor performs operations of hundreds or even thousands of neurons of an implemented neural network part, are constructed.

From a technological point of view, most neurocomputers are based on classical microelectronics (e. g. VLSI technology) where neurons correspond to gates (e. g. specialized transistors) and the weights of synaptic connections are represented by resistor couplings. This approach brings about technical problems such as great density of inter-neuron connections (increasing quadratically with the number of neurons) and the weight adaptability of all of these connections. Therefore, the neural network adaptive mode is sometimes realized separately on the conventional computer by using available netware beforehand and then, the resulting network configuration is wired into a respective neurocomputer circuit. Also optoelectronics is more often employed here and long-term prospects envisage completely different technologies, e. g. molecular electronics, hybrid biochips, etc.

# Chapter 2

# Classical Models of Neural Networks

## 2.1 The Network of Perceptrons

The first successful model of neural network in the history of neurocomputing (see Section 1.1) was the network of *perceptrons* [66]. The **architectural** dynamics of this network specifies a fixed architecture of a one–layered network $n$-$m$ at the beginning. This means that the network consists of $n$ input neurons, each being an input for all $m$ output neurons as it is depicted in Figure 2.1. Denote by $x_1, \ldots, x_n$ the real states

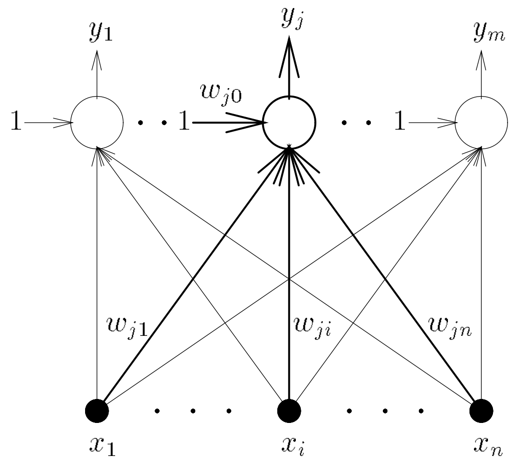

Figure 2.1: The Architecture of the Network of Perceptrons.

of input neurons, i. e. $\mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$ is the network input, and denote by $y_1, \ldots, y_m$ the binary states of output neurons, i. e. $\mathbf{y} = (y_1, \ldots, y_m) \in \{0, 1\}^m$ is the network output. Furthermore, $w_{ji}$ represents a real synaptic weight associated with the connection leading from the $i$th input neuron ($i = 1, \ldots, n$) to the $j$th output neuron

($j = 1, \ldots, m$), and $w_{j0} = -h_j$ is a bias (a threshold $h_j$ with the opposite sign) of the $j$th output neuron corresponding to a formal unit input $x_0 = 1$.

The **computational** dynamics of the network of perceptrons determines how the network function is computed. In this case, the real states of neurons in the input layer are assigned to the network input and the output neurons compute their binary states determining the network output in the same way as the formal neuron does (see equation (1.3)). This means that every perceptron computes its excitation level first as the corresponding *affine combination* of inputs:

$$\xi_j = \sum_{i=0}^{n} w_{ji} x_i \quad j = 1, \ldots, m\,. \tag{2.1}$$

The coefficients $\mathbf{w} = (w_{10}, \ldots, w_{1n}, \ldots, w_{m0}, \ldots, w_{mn})$ create the network configuration. Then the perceptron state is determined from its excitation level by applying an activation function $\sigma : \mathbb{R} \longrightarrow \{0, 1\}$ such as the hard limiter (1.7). This means, that the function $\mathbf{y}(\mathbf{w}) : \mathbb{R}^n \longrightarrow \{0, 1\}^m$ of the network of perceptrons, which depends on the configuration $\mathbf{w}$, is given by the following equation:

$$y_j = \sigma(\xi_j) \;\; j = 1, \ldots, m \quad \text{where } \sigma(\xi) = \begin{cases} 1 & \xi \geq 0 \\ 0 & \xi < 0\,. \end{cases} \tag{2.2}$$

In an **adaptive** mode the desired function of the network of perceptrons is given by a training set:

$$\mathcal{T} = \left\{ (\mathbf{x}_k, \mathbf{d}_k) \;\middle|\; \begin{array}{l} \mathbf{x}_k = (x_{k1}, \ldots, x_{kn}) \in \mathbb{R}^n \\ \mathbf{d}_k = (d_{k1}, \ldots, d_{km}) \in \{0, 1\}^m \end{array} \; k = 1, \ldots, p \right\} \tag{2.3}$$

where $\mathbf{x}_k$ is a real input of the $k$th training pattern, and $\mathbf{d}_k$ is the corresponding desired binary output (given by a teacher). The aim of adaptation is to find a configuration $\mathbf{w}$ such that for every input $\mathbf{x}_k$ ($k = 1, \ldots, p$) from training set $\mathcal{T}$, the network responds with the desired output $\mathbf{d}_k$ in computational mode, i. e. it holds:

$$\mathbf{y}(\mathbf{w}, \mathbf{x}_k) = \mathbf{d}_k \quad k = 1, \ldots, p\,. \tag{2.4}$$

Of course, the condition (2.4) cannot always be fulfilled because not every function can be computed by a single perceptron (e. g., see Figure 1.7 of the XOR function) or the training set may not even be a function (i. e., two different outputs are required for one input). In this case, we make an effort to adapt the network to as many patterns as possible. In practice, sometimes it is better when the network does not learn the training set one hundredpercent, since the example patterns may not be completely precise (a teacher may not be perfect).

At the beginning of adaptation, at time 0, the weights of configuration $\mathbf{w}^{(0)}$ are initialized randomly close to zero, e. g. $w_{ji}^{(0)} \in \langle -1, 1 \rangle$ ($j = 1, \ldots, m$, $i = 0, \ldots, n$). The network of perceptrons has discrete adaptive dynamics. At each adaptation time step $t = 1, 2, 3, \ldots$ one pattern from the training set is presented to the network which attempts to learn it, i. e. the network adapts its weights with respect to this pattern. The order of patterns during the learning process is prescribed by the so-called *training* *strategy* and it can be, for example, arranged on the analogy of human learning. One student reads through a textbook several times to prepare for an examination, another one learns everything properly during the first reading, and as the case may be, at the end both revise the parts which are not correctly answered by them. Usually, the adaptation of the network of perceptrons is performed in the so-called *training epoch* in which all patterns from the training set are systematically presented (some of them even several times over). For example, at time $t = (c-1)p + k$ (where $1 \leq k \leq p$) corresponding to the $c$th training epoch the network learns the $k$th training pattern.

The adaptive dynamics of the network of perceptrons determines the update of the configuration $\mathbf{w}^{(t)}$ at time $t > 0$ after the $k$th training pattern is presented, and is defined by the following *perceptron learning rule*:

$$w_{ji}^{(t)} = w_{ji}^{(t-1)} - \varepsilon x_{ki} \left( y_j(\mathbf{w}^{(t-1)}, \mathbf{x}_k) - d_{kj} \right) \qquad \begin{array}{l} j = 1, \ldots, m \\ i = 0, \ldots, n\,. \end{array} \tag{2.5}$$

The so-called *learning rate* $0 < \varepsilon \leq 1$ measures the influence of a pattern on adaptation (the 'motivation' for learning). This parameter is usually initialized with a small value which is being increased later during the adaptation. By the above-mentioned simplified analogy with students' preparation for an examination, this corresponds to a first cursory acquaintance with the learning subject and to a subsequent detailed training.

The expression $y_j(\mathbf{w}^{(t-1)}, \mathbf{x}_k) - d_{kj}$ in formula (2.5) is the discrepancy between the actual $j$th network output for the $k$th pattern input and the corresponding desired output of this pattern. Hence, this determines the error of the $j$-th network output with respect to the $k$th training pattern. Clearly, if this error is zero the underlying weights are not modified. Otherwise this error may be either 1 or $-1$ since only the binary outputs are considered. In the geometrical interpretation (see Paragraph 1.3.1), the adaptation according to (2.5) means that the hyperplane with coefficients $w_{j0}, \ldots, w_{jn}$ associated with the $j$th neuron is moved in the input space in the direction of an incorrectly classified pattern $\mathbf{x}_k$ to include it into the correct halfspace. The inventor of the perceptron, Rosenblatt showed that the adaptive dynamics (2.5) guarantees that the network finds its configuration (providing that it exists) in the adaptive mode after a finite number of steps, for which it classifies all training patterns correctly (i. e., the network error with respect to the training set is zero), and thus the condition (2.4) is met.

Considering the network of perceptrons is capable of computing only a restricted class of functions, the significance of this model is rather theoretical. In addition, the above-outlined convergence theorem for the adaptive mode does not guarantee the learning efficiency which has been confirmed by time consuming experiments. The generalization capability of this model is also not revolutionary because the network of perceptrons can only be used in cases when the classified objects are separable by hyperplanes in the input space (e. g., special tasks in pattern recognition, etc.). However, this simple model is a basis of more complex paradigms like a general feedforward network with the backpropagation learning algorithm.

## 2.2 Feedforward Network and Backpropagation

The most prominent and widely applied model of neural networks is the *multilayered neural network* (*feedforward network*) with the backpropagation learning algorithm [68] which is used approximately in 80% of all neural network applications. The importance of this algorithm for the progress in neurocomputing has been described in Section 1.1. This model generalizes the network of perceptrons for the architectures with hidden layers (the so-called *multilayered perceptron*), and therefore we refer to its principles (see Section 2.1) in the following exposition. Since the multilayered perceptron is widely spread and has certain drawbacks there are many variants of this model which attempt to improve its properties. At first, we will describe a basic variant and then we will outline some possible modifications.

### 2.2.1 Architectural and Computational Dynamics

At the beginning, the **architectural** dynamics of a multilayered perceptron commonly specifies a fixed architecture of feedforward network. Usually, a two-layered or three-layered network (e. g., see Figure 1.10) is exploited since the explicit purpose for hidden neurons and their bindings which is necessary for a specific architecture design, is unknown. In the sequel, the model with a general (acyclic) feedforward architecture will be described. For this purpose, the following notation is introduced. The set of $n$ input neurons is denoted by $X$ and $Y$ is the set of $m$ output neurons. The neurons are indicated by indices $i$, $j$, etc., and thus, $\xi_j$ represents the real excitation level of neuron $j$, as well as $y_j$ is the real state, i. e. the output of $j$. Similarly as in the network of perceptrons, the connection leading from neuron $i$ to the non-input neuron $j$ is labeled with a real synaptic weight $w_{ji}$, and $w_{j0} = -h_j$ is a bias (a threshold $h_j$ with the opposite sign) of $j$, corresponding to a formal unit input $y_0 = 1$. Furthermore, $j_{\leftarrow}$ denotes the set of all neurons, from which a connection to neuron $j$ leads, and hence they represent the inputs for neuron $j$ (including the formal unit input $0 \in j_{\leftarrow}$), and $j^{\rightarrow}$ is the set of neurons, to which a connection from neuron $j$ leads, and hence the neuron $j$ is their input.

In a **computational** mode the feedforward network evaluates a function $\mathbf{y}(\mathbf{w}) : \mathbb{R}^n \longrightarrow (0,1)^m$ for a given input, which is determined by a configuration $\mathbf{w}$. The computation proceeds according to the following computational dynamics. At time 0, the corresponding states of input neurons $y_i$ ($i \in X$) are assigned to the network input, and the remaining neurons have their states undetermined. At time $t > 0$ the real values of excitation levels

$$\xi_j = \sum_{i \in j_{\leftarrow}} w_{ji} y_i \tag{2.6}$$

for all neurons $j$, whose inputs (from $j_{\leftarrow}$) have already had their states determined, are computed. This means that the neurons in the $t$th layer are updated at time $t$. Then the excitation level (2.6) is used to determine the real state $y_j = \sigma(\xi_j)$ of neuron $j$ by applying a differentiable activation function $\sigma : \mathbb{R} \longrightarrow (0,1)$ which is the standard sigmoid (1.9) approximating continuously the hard limiter (1.7):

$$\sigma(\xi) = \frac{1}{1+e^{-\lambda\xi}} \,. \tag{2.7}$$

The differentiability of the exploited transfer function (2.7) implying the differentiability of the network function is essential for the backpropagation learning algorithm. The real parameter $\lambda$ called *gain* determines a nonlinear grow (or a fall for $\lambda < 0$) of the standard sigmoid in a neighborhood of zero which measures the neuron 'determination' (e. g., for $\lambda \to \infty$ the hard limiter is obtained). A graph of the standard sigmoid for $\lambda > 0$ is drawn in Figure 1.11 (it is symmetric about the $y$-axis for $\lambda < 0$). In the basic model $\lambda = 1$ is usually considered (see equation (1.9)), however, in general the gain $\lambda_j$ (and hence even the activation function $\sigma_j$) may be different for each (non-input) neuron $j$. Then the neuron state is computed as follows:

$$y_j = \sigma_j(\xi_j) \quad \text{where } \sigma_j(\xi) = \frac{1}{1+e^{-\lambda_j\xi}} \,. \tag{2.8}$$

This way, layer by layer, the outputs of all neurons are being computed in the computational mode since the underlying architecture is an acyclic (connected) graph. The computational mode is accomplished when the states of all neurons in the network are determined, especially the output neuron states represent the network output, i. e. the network function value for a given input.

### 2.2.2 Adaptive Dynamics

The **adaptive** mode of the feedforward network is performed similarly as in the network of perceptrons. The desired function is again specified by the training set (2.3). The *network error* $E(\mathbf{w})$ with respect to this training set is defined as the sum of partial network errors $E_k(\mathbf{w})$ regarding particular training patterns, and it depends on the network configuration $\mathbf{w}$:

$$E(\mathbf{w}) = \sum_{k=1}^{p} E_k(\mathbf{w}) \,. \tag{2.9}$$

Further, the partial network error $E_k(\mathbf{w})$ with respect to the $k$th training pattern is proportional to the sum of squared discrepancies between the actual values of network outputs for the $k$th training pattern input and the relevant desired output values:

$$E_k(\mathbf{w}) = \frac{1}{2} \sum_{j \in Y} \left(y_j(\mathbf{w}, \mathbf{x}_k) - d_{kj}\right)^2 \,. \tag{2.10}$$

The aim of adaptation is to minimize the network error (2.9) in the weight space. Since the network error directly depends on the complicated nonlinear composite function of feedforward network this goal represents a non-trivial optimization task. In the basic model the simplest variant of the gradient method is employed for its solving which requires the error function to be differentiable.

At the beginning of adaptation, at time 0, the weights of configuration $\mathbf{w}^{(0)}$ are initialized randomly within a neighborhood of zero, e. g. $w_{ji}^{(0)} \in \langle -1, 1\rangle$ (or more ingeniously with values of order $1/\sqrt{|j_{\leftarrow}|}$ where $|j_{\leftarrow}|$ is the number of inputs $i \in j_{\leftarrow}$ of neuron $j$). The adaptation proceeds at the discrete time steps which correspond to the training epochs. The new configuration $\mathbf{w}^{(t)}$ at time $t > 0$ is computed as follows:

$$w_{ji}^{(t)} = w_{ji}^{(t-1)} + \Delta w_{ji}^{(t)} \tag{2.11}$$

where the increment of weights $\Delta\mathbf{w}^{(t)}$ at time $t > 0$ is proportional to the negative gradient of error function $E(\mathbf{w})$ at the point $\mathbf{w}^{(t-1)}$:

$$\Delta w_{ji}^{(t)} = -\varepsilon \frac{\partial E}{\partial w_{ji}} \left(\mathbf{w}^{(t-1)}\right) \tag{2.12}$$

The learning rate $0 < \varepsilon < 1$ has a similar meaning here as in the network of perceptrons.

Once more, the geometrical interpretation is helpful for a better understanding of the above-introduced gradient method. In Figure 2.2 the error function $E(\mathbf{w})$ is

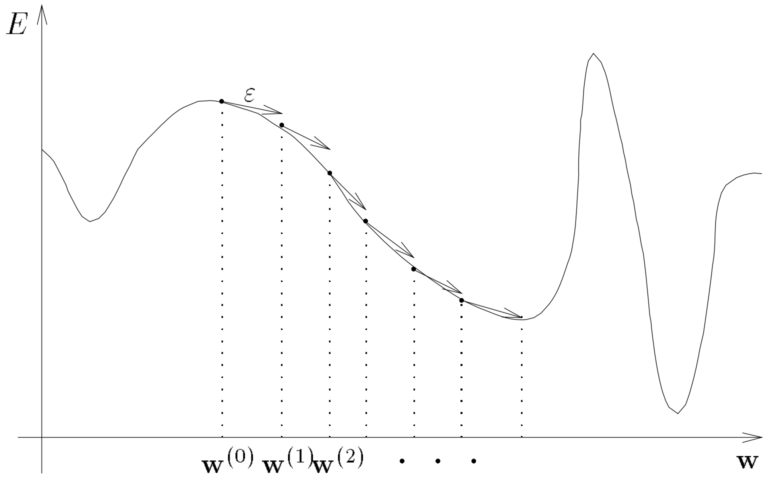

Figure 2.2: Gradient method.

schematically depicted in such a way that the configuration representing (typically) a many-dimensional weight vector $\mathbf{w}$ is projected to the $x$-axis. The error function depending on the configuration determines the network error with respect to a fixed training set. In the adaptive mode such configuration is searched which minimizes the error function. We start with a randomly chosen configuration $\mathbf{w}^{(0)}$ for which the corresponding network error is probably large with respect to the desired function. On the very simplified analogy of human learning this corresponds to the initial setting of synaptic weights in a baby's brain who instead of a desired behavior like walking, speaking, etc. performs non-coordinating moves and gives out indeterminate sounds.

During the adaptation, the tangent vector (gradient) $\frac{\partial E}{\partial \mathbf{w}}(\mathbf{w}^{(0)})$ to the network function graph at the point $\mathbf{w}^{(0)}$ is constructed and the searching position in the weight space is shifted by $\varepsilon$ in the direction of this vector downward. For a sufficiently small $\varepsilon$ a new configuration $\mathbf{w}^{(1)} = \mathbf{w}^{(0)} + \Delta\mathbf{w}^{(1)}$ is obtained for which the error function value is less than that for the original configuration $\mathbf{w}^{(0)}$, i. e. $E(\mathbf{w}^{(0)}) \geq E(\mathbf{w}^{(1)})$. The whole procedure of the tangent vector construction is repeated for $\mathbf{w}^{(1)}$ and thus, $\mathbf{w}^{(2)}$ is produced such that $E(\mathbf{w}^{(1)}) \geq E(\mathbf{w}^{(2)})$, etc. Finally, a local minimum of the error function is reached in the limit. In a multidimensional space this process exceeds our imagination. Despite that the gradient method always converges from an arbitrary initial configuration to some local minimum (providing that it exists) for an appropriate choice of learning rate $\varepsilon$ it is not guaranteed that this will happen in a real time. Usually, this process is very time consuming (several days of PC computations) even for small architectures of multilayered networks (hundreds of neurons).

The main problem concerning the gradient method is that even if it finds a local minimum, then this minimum may not be a global one (see Figure 2.2). The above-introduced adaptation procedure will terminate at this local minimum (zero gradient) and the network error is further not decreased. The motivation analogy with human learning can be developed here in a fanciful way. The initial setting of configuration in a neighbourhood of some error function minimum can be interpreted as to determine a learning capacity of an individual or even the intelligence. From this point of view more intelligent creatures start their adaptation nearby deeper minima. However, the error function is defined even here relatively with respect to a desired 'intelligent' behavior (training set) which may not be universally valid. The human value can probably not be measured by any error function. Electric shocks applied in mental hospitals evoke some methods of neural network adaptation. In a case when the learning process stagnates in a shallow local minimum of the error function, a random noise is carried into the network configuration to get the network out of the attraction area associated with this local minimum and hopefully, to converge to a deeper minimum.

### 2.2.3 The Backpropagation Strategy

To implement the adaptive dynamics (2.11), the gradient of the error function in formula (2.12) must be computed and this represents a non-trivial task, due to the complexity of the error function. At first, by applying the rule for the derivative of the sum in (2.9) this gradient is reduced to the sum of gradients of partial error functions:

$$\frac{\partial E}{\partial w_{ji}} = \sum_{k=1}^{p} \frac{\partial E_k}{\partial w_{ji}}. \tag{2.13}$$

Since the network is composed of single neurons the rule for a composite function derivative is naturally used for the gradient calculation:

$$\frac{\partial E_k}{\partial w_{ji}} = \frac{\partial E_k}{\partial y_j} \frac{\partial y_j}{\partial \xi_j} \frac{\partial \xi_j}{\partial w_{ji}}. \tag{2.14}$$

The partial derivative $\frac{\partial \xi_j}{\partial w_{ji}}$ in equation (2.14) can be calculated explicitly from formula (2.6):

$$\frac{\partial \xi_j}{\partial w_{ji}} = y_i \tag{2.15}$$

and the partial derivative $\frac{\partial y_j}{\partial \xi_j}$ is obtained from the formula (2.8) for the standard sigmoid whose derivative can be expressed by its function value as follows:

$$\begin{aligned}\frac{\partial y_j}{\partial \xi_j} &= \frac{\lambda_j e^{-\lambda_j \xi_j}}{(1+e^{-\lambda_j \xi_j})^2} = \\ &= \frac{\lambda_j}{1+e^{-\lambda_j \xi_j}}\left(1-\frac{1}{1+e^{-\lambda_j \xi_j}}\right) = \lambda_j y_j (1-y_j)\,. \qquad (2.16)\end{aligned}$$

Substituting (2.15) and (2.16) into (2.14) we get:

$$\frac{\partial E_k}{\partial w_{ji}} = \frac{\partial E_k}{\partial y_j}\lambda_j y_j (1-y_j) y_i\,. \qquad (2.17)$$

The computation of the remaining partial derivatives $\frac{\partial E_k}{\partial y_j}$ in formula (2.17) starts in the output layer and it is propagated back to the input layer as the name backpropagation suggests. Providing that $j \in Y$ is an output neuron, then the mentioned derivative can be calculated directly from the formula (2.10) for the partial error function:

$$\frac{\partial E_k}{\partial y_j} = y_j - d_{kj} \quad j \in Y \qquad (2.18)$$

and its value corresponds to the error of output neuron $j$ for the $k$th training pattern. For a hidden neuron $j \notin X \cup Y$ the rule for a composite function derivative is again employed for $\frac{\partial E_k}{\partial y_j}$. In addition, the derivatives which can be obtained by direct formal differentiating are calculated:

$$\begin{aligned}\frac{\partial E_k}{\partial y_j} &= \sum_{r \in j^{\rightarrow}} \frac{\partial E_k}{\partial y_r}\frac{\partial y_r}{\partial \xi_r}\frac{\partial \xi_r}{\partial y_j} = \\ &= \sum_{r \in j^{\rightarrow}} \frac{\partial E_k}{\partial y_r}\lambda_r y_r (1-y_r) w_{rj} \quad j \notin X \cup Y\,. \qquad (2.19)\end{aligned}$$

In formula (2.19) the partial derivative $\frac{\partial E_k}{\partial y_j}$ for hidden neuron $j$ has been reduced to partial derivatives $\frac{\partial E_k}{\partial y_r}$ for incident neurons $r \in j^{\rightarrow}$ which have neuron $j$ as their input. Hence, the computation of $\frac{\partial E_k}{\partial y_j}$ can start with output neurons for which this partial derivative is computed according to (2.18). Further, the computation proceeds layer by layer backwards in such a way that $\frac{\partial E_k}{\partial y_j}$ is computed for a hidden neuron $j$ according to (2.19) providing that the corresponding partial derivatives $\frac{\partial E_k}{\partial y_r}$ have already been evaluated for all $r \in j^{\rightarrow}$. The correctness of this procedure follows from the fact that the network architecture is an acyclic graph.

### 2.2.4 The Implementation of Backpropagation

Now, the adaptive dynamics can be additionally summarized in the following transparent algorithm:

1. Initialize the discrete adaptation time $t := 0$.
2. Choose randomly $w_{ji}^{(0)} \in \langle -1, 1 \rangle$.
3. Increment $t := t + 1$.
4. Assign $E'_{ji} := 0$ for every connection from $i$ to $j$.
5. For every training pattern $k = 1, \dots, p$ do:
   (a) Evaluate the network function output $\mathbf{y}(\mathbf{w}^{(t-1)}, \mathbf{x}_k)$, i. e. compute the states and the excitation levels of all neurons (computational mode) for the $k$th training pattern input $\mathbf{x}_k$ by using formulas (2.6) and (2.8).
   (b) By means of the backpropagation strategy, for every non-input neuron $j \notin X$ evaluate the partial derivative $\frac{\partial E_k}{\partial y_j}(\mathbf{w}^{(t-1)})$ of the $k$th partial error function by the state of neuron $j$ at the point $\mathbf{w}^{(t-1)}$ by applying formulas (2.18) and (2.19) (for this purpose exploit the values of neuron states which have been computed in step 5a).
   (c) Compute the gradient $\frac{\partial E_k}{\partial w_{ji}}(\mathbf{w}^{(t-1)})$ of the $k$th partial error function at point $\mathbf{w}^{(t-1)}$ according to formula (2.17) by using the partial derivatives $\frac{\partial E_k}{\partial y_j}(\mathbf{w}^{(t-1)})$ which have been evaluated in step 5b.
   (d) Add $E'_{ji} := E'_{ji} + \frac{\partial E_k}{\partial w_{ji}}(\mathbf{w}^{(t-1)})$.
6. { It holds $E'_{ji} = \frac{\partial E}{\partial w_{ji}}(\mathbf{w}^{(t-1)})$, due to (2.13). }
   According to (2.12) set $\Delta w_{ji}^{(t)} := -\varepsilon E'_{ji}$.
7. According to (2.11) update $w_{ji}^{(t)} := w_{ji}^{(t-1)} + \Delta w_{ji}^{(t)}$.
8. Evaluate the network error $E(\mathbf{w}^{(t)})$ for the configuration $\mathbf{w}^{(t)}$ according to formulas (2.9) and (2.10).
9. If the error $E(\mathbf{w}^{(t)})$ is sufficiently small, then terminate or else continue with step 3.

Although the proper description of the backpropagation learning algorithm is formulated for the classical von Neumann computer architecture it can obviously be implemented distributively. For each training pattern the following procedure is performed. At first, the computational mode is applied for its sample input and hence, the information is spread within the network from the input towards the output. After an external information regarding the desired output (i. e. particular output errors) is provided by a teacher, the partial derivatives $\frac{\partial E_k}{\partial y_j}$ (particularly $\frac{\partial E_k}{\partial w_{ji}}$) are computed by propagating the signal from the network output backwards to the input. This backward run reminds a reverse 'computational mode' when the 'input' corresponds to the desired output, the partial derivative $\frac{\partial E_k}{\partial y_j}$ represents the 'state' of neuron $j$ and the 'computational dynamics' is given by formulas (2.18) for the 'input' neuron and (2.19) for the hidden one. During the backward run the network computation proceeds sequentially layer by layer while within one layer it can be performed in parallel.

Moreover, we add several comments on the implementation for those who might want to program the backpropagation learning algorithm on a conventional computer. For each neuron $j$ in a general feedforward network the set $j_{\leftarrow}$ of incident neurons (including the relevant synaptic weights) which represent the actual inputs for neuron $j$ must be explicitly stored in a memory (in the multilayered topology this is given implicitly). This is especially important during the computational mode (namely, for the excitation level computation according to formula (2.6)) which is also embodied in the backpropagation learning algorithm (step 5a). On the other hand, for the backward run of this algorithm the set $j^{\rightarrow}$ of incident neurons which are the actual outputs of neuron $j$, does not need to be explicitly stored. Although in step 5b the partial derivative $\frac{\partial E_k}{\partial y_j}$ is computed for each neuron $j$ according to formula (2.19) in which one sums over the neurons $r \in j^{\rightarrow}$, this can be avoided in such a way that the sum for $\frac{\partial E_k}{\partial y_j}$ is being evaluated term by term in advance as follows. Suppose that during the backward run when proceeding from the network output to the input, a neuron $r \in j^{\rightarrow}$ is being traversed and assume that $\frac{\partial E_k}{\partial y_r}$ has been computed. Then the terms $\frac{\partial E_k}{\partial y_r}\lambda_r y_r(1-y_r)w_{ri}$ (see formula (2.19)) are added to the variables for $\frac{\partial E_k}{\partial y_i}$ for all neurons $i \in r_{\leftarrow}$ which represent the inputs for $r$ (especially, the neuron $j$ is included). At the moment when neuron $j$ is traversed all neurons $r \in j^{\rightarrow}$ have already been processed and hence, the value of $\frac{\partial E_k}{\partial y_j}$ has been correctly computed. Moreover, the computation of $\frac{\partial E_k}{\partial w_{ji}}$ in step 5c can be realized during this procedure and the steps 5b, 5c, and 5d proceeds simultaneously.

In the introduced algorithm the network configuration is always updated after each training epoch and thus, the complete training set is taken into account. The partial derivatives of the partial error functions are added term by term (i. e., are accumulated) in step 5d and hence, the order of training patterns does not matter during the learning phase. This variant is called *accumulated learning* (*off-line backpropagation, true gradient method*). In this case, for each connection in the network the relevant values $E'_{ji}$ of accumulated partial derivatives must be stored besides proper weights. Another less memory consuming alternative is to update the configuration for every training pattern. This is called *on-line backpropagation.* In this case a complete training set does not need to be given beforehand and pattern after pattern may be generated (even randomly and without repeating) in the order determined by a chosen training strategy. After the computation of gradient $\frac{\partial E_k}{\partial w_{ji}}(\mathbf{w}^{(t-1)})$ for the $k$th partial error function is accomplished in step 5c, the weights are immediately updated with respect to the $k$th pattern instead of accumulating the derivative in step 5d:

$$w_{ji}^{(t)} := w_{ji}^{(t-1)} + \Delta w_{ji}^{(t)} \quad \text{where } \Delta w_{ji}^{(t)} := -\varepsilon \frac{\partial E_k}{\partial w_{ji}}(\mathbf{w}^{(t-1)}) . \tag{2.20}$$

This replaces the corresponding weight modification in steps 6 and 7. However, the gradient of the partial error function for the next training pattern is evaluated by using previously updated weights. Therefore, the global network error is not minimized with respect to the whole training set exactly under the gradient method (2.11), (2.12), and on our experience this process decelerates the convergence.

The current value of the network error provides a user with the information on the course of learning (e. g., whether the algorithm converges or whether it finds itself in a local minimum, etc.) as well as it represents the basis for a halting criterion in step 9. The error computation in step 8 is very time consuming (it involves the computational mode for every training pattern) and therefore, it is not reasonable to compute the error in each training epoch, but it is sufficient to perform it once in a multiple of these epochs. The learning rate $\varepsilon$ can be tuned according to the error evolution and therefore, it is appropriate to display (e. g. on the screen) the current information about the error. It is also necessary to enable the program to update the learning rate interactively during the adaptive mode. Since the adaptive mode is typically a long-term process it is recommended to store (archive) the network configuration on the disc for security reasons once in a while (providing that the error has decreased in the meantime).

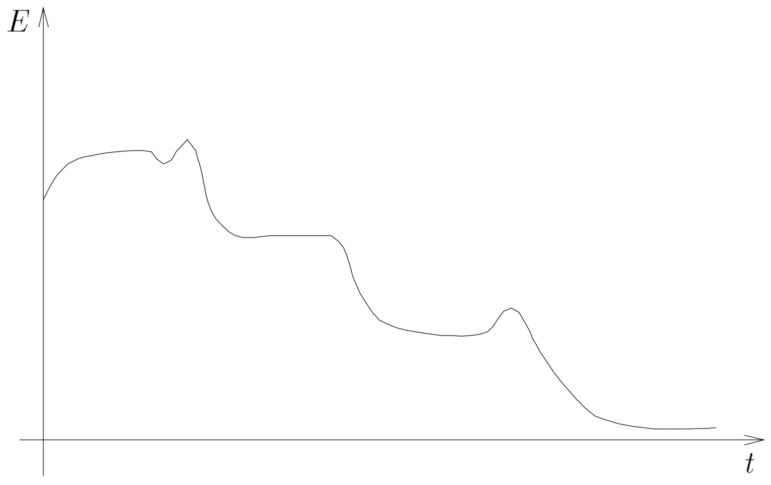

Figure 2.3: A typical error evolution during the backpropagation learning.

It appears that the objective of the adaptive mode based on the backpropagation learning algorithm is to minimize the network error function $E(\mathbf{w})$ as much as possible (e. g. to find its global minimum) by choosing an appropriate initial configuration and by controlling the learning rate properly. For a small $\varepsilon$ the method converges very slowly (the error is slightly decreasing) and on the other hand, for a greater $\varepsilon$ it diverges (the error is increasing). The way in which the parameter $\varepsilon$ is tuned to control the adaptive mode is usually learned by experience. The following recommendation can be formulated only in a very inexact way. For larger architectures it is better to start with a smaller $\varepsilon$ (e. g. of the order of thousandths or tenthousandths). During the successful convergence, $\varepsilon$ may be slightly increased and during the apparent divergence or oscillation the value of $\varepsilon$ must be decreased (e. g. exponentially). Even for a sudden great error increase after a preceding successful convergence with a long-term constant

$\varepsilon$, it is worth to continue in the adaptation with the same $\varepsilon$. The method might jump within the weight space into the area of a better convergence which is either confirmed or disproved in the next training epochs. Sometimes it is better to restart the learning process for a new initial configuration when the method fails. The graph of a typical error evolution in the course of adaptation is sketched in Figure 2.3. At the beginning of learning the error may slightly increase and later on it alternates the phase of its rapid (approximately exponential) decrease with its long-term stagnation when the error tends very slowly (the error decrease is approximately exponentially decreasing) upper to some non-zero value. Usually, under certain conditions the method succeeds to find a global minimum (zero error) after a long-term learning.

### 2.2.5 The Backpropagation Variants

Various variants of the basic backpropagation model make an effort to solve the problems regarding the minimization of the error function $E(\mathbf{w})$ in a real time. For example, the selection issue of the gain values $\lambda_j$ within the activation function (2.8) can be solved for particular neurons $j$ as follows. Besides synaptic weights $\mathbf{w}$, the network error $E(\mathbf{w}, \boldsymbol{\lambda})$ is also a function of gains $\boldsymbol{\lambda}$. This means that the configuration is determined by the vector $\mathbf{w}$ of all weights as well as by the vector $\boldsymbol{\lambda}$ of all gain parameters. Thus, during the learning phase this configuration (i. e., including gains) is adapted so that the network error is minimized by the gradient method in the weight and gain space. This way the degree of freedom for adaptation is extended when the shapes of activation functions (i. e., the measures of neuron 'determination') are adjusted to the training set and hopefully, a global minimum of the network error function may be found. On the other hand, by increasing the number of updated parameters the number of arithmetic operations is increased as well and hence, the learning process is decelerated. From the technical point of view, the gain parameter adaptation is implemented similarly as the weight updates (2.11), (2.12). At the beginning of adaptation, at time 0, the gains $\boldsymbol{\lambda}^{(0)}$ are initialized randomly, e. g. close to 1. In the next training epochs, at time $t > 0$, the new values of gain parameters $\boldsymbol{\lambda}^{(t)}$ are computed as follows:

$$\lambda_j^{(t)} = \lambda_j^{(t-1)} + \Delta\lambda_j^{(t)} \quad \text{where } \Delta\lambda_j^{(t)} = -\varepsilon' \frac{\partial E}{\partial \lambda_j}\left(\mathbf{w}^{(t-1)}, \boldsymbol{\lambda}^{(t-1)}\right) \tag{2.21}$$

and $0 < \varepsilon' < 1$ is the learning rate for gains with the same meaning as for synaptic weights. The evaluation of the partial derivatives $\frac{\partial E}{\partial \lambda_j}$ is again analogous (see formulas (2.13), (2.14) and (2.16)):

$$\frac{\partial E}{\partial \lambda_j} = \sum_{k=1}^{p} \frac{\partial E_k}{\partial \lambda_j} \quad \text{where } \frac{\partial E_k}{\partial \lambda_j} = \frac{\partial E_k}{\partial y_j}\frac{\partial y_j}{\partial \lambda_j} = \frac{\partial E_k}{\partial y_j}\xi_j y_j (1 - y_j)\,. \tag{2.22}$$

The partial derivatives $\frac{\partial E_k}{\partial y_j}$ are computed by using the backpropagation strategy (2.18), (2.19). During the adaptation the gain parameters of some neurons may even become negative which causes the corresponding activation functions to be decreasing.

The above-described basic variant of the gradient method (2.11), (2.12) is often exploited due to its simplicity although it is not very efficient. With a small learning rate $\varepsilon$ this method converges very slowly, however, for a greater $\varepsilon$ the method diverges. A simple and quite frequent modification which makes effort to avoid this drawback takes still into account the preceding weight correction for the new weight update. This is called a *momentum* term [64]:

$$\Delta w_{ji}^{(t)} = -\varepsilon \frac{\partial E}{\partial w_{ji}} \left( \mathbf{w}^{(t-1)} \right) + \alpha \Delta w_{ji}^{(t-1)} \tag{2.23}$$

where $0 < \alpha < 1$ is the *momentum rate* which measures the influence of the previous weight increment (e. g., its common value is $\alpha = 0.9$). By using the momentum term the gradient method follows the shape of the error function $E(\mathbf{w})$ better because it takes into account the previous gradient.

The gradient method can be accelerated properly by tuning the learning and momentum rates $\varepsilon$ and $\alpha$, respectively. Therefore, several heuristics have been proposed to automate the selection of these parameters during the learning process [12, 17, 38, 78]. For example, a separate learning rate $\varepsilon_{ji}$ is introduced for each weight $w_{ji}$ in the network. The value of $\varepsilon_{ji}$ can be selected e. g. of order $1/|j_{\leftarrow}|$ where $|j_{\leftarrow}|$ is the number of inputs for the neuron $j$ [64]. Another possibility is to increase the learning rate $\varepsilon_{ji}^{(t)} = K\varepsilon_{ji}^{(t-1)}$ linearly for $K > 1$ (e. g., $K = 1.01$) at time $t > 1$ if the increment sign for the relevant weight is not changed, i. e. $\Delta w_{ji}^{(t)} \Delta w_{ji}^{(t-1)} > 0$. Otherwise, $\varepsilon_{ji}^{(t)} = \varepsilon_{ji}^{(t-1)}/2$ is exponentially decreased.

In practical applications, the well-known, elaborated and more efficient methods of nonlinear optimization [47] are often omitted for neural network learning. For example, *Newton method* converges rapidly in a sufficiently near neighborhood of an error function minimum. Nevertheless, this method requires a second-order derivative computation, it is computationally very time consuming (e. g., at each iterative step the matrix inversion is computed) as well as numerically non-stable. A more suitable candidate for the minimization of neural network error function is the *conjugate gradient method* [44, 48] which exploits first-order derivatives only.

We conclude with a note that the introduced backpropagation learning algorithm for feedforward neural networks can also be generalized for cyclic architectures providing that the recurrent network converges to a stable state under a computational mode. This variant of the algorithm is called *recurrent backpropagation* [3, 2, 61, 62, 63, 65].

### 2.2.6 The Choice of Topology and Generalization

One of the main problems with the feedforward neural networks trained by the backpropagation learning algorithm (besides the error function minimization) is the choice of an adequate topology for solving a particular practical task. The relations between inputs and outputs are rarely known in details to be exploited for a special architecture design. As usual, a multilayered topology with one or two hidden layers is employed. One expects the backpropagation learning algorithm to generalize the underlying relations from the training set and to project them into the weights associated with particular connections among neurons. However, even in this case the numbers of neurons in hidden layers must be selected. It appears that this issue of the architectural dynamics is related to the adaptation and generalization of a neural network.

The architecture of a multilayered network, i. e. the numbers of hidden neurons, should correspond to the complexity of a currently solved problem, i. e. to the number of training patterns, to the numbers of their inputs and outputs, and to a relation structure which is described by them. Clearly, poor architectures are probably too weak to solve complex tasks. In the backpropagation learning algorithm, too small networks usually terminate in shallow local minima and the topology needs to be enriched with additional hidden neurons to provide the adaptation with a greater degree of freedom. On the other hand, rich architectures commonly allow to find the global minimum during the learning process although the computational time for adaptation increases with a greater number of weight parameters. However, in this case the resulting network configuration is usually adjusted too much to the training patterns including their inaccuracy and errors. Hence, the network responds incorrectly to previously unseen inputs, i. e. its generalization capability is poor. This situation in which the network memorizes perfectly the training patterns without generalizing the laws covered by the training set is called *overfitting*. The graphs of two network functions including the training patterns (points) to which they both were adapted, are depicted in Figure 2.4. The bold line represents the overfitted network whose function

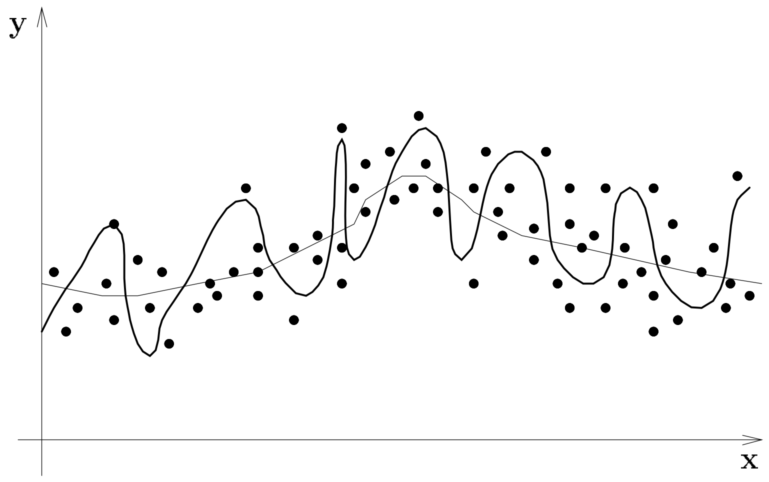

Figure 2.4: The function graphs of the overfitted network (bold line) and of the network with a 'correct' generalization.

is adjusted to imprecise training patterns while the thin line corresponds to the network function which has 'correctly' generalized rules from the training set. It appears that, for a particular task, there exists an 'optimal' architecture that is rich enough to solve the problem at hand and, at the same time is not too large to generalize correctly the relevant relations between inputs and outputs.

There exist theoretical results concerning the upper bound for the number of hidden neurons which are sufficient to implement an arbitrary function from a given class (see the third part of this book), however, they are too overvalued for practical needs and hence useless for architectural design. In practice, the topology is usually searched using heuristics, e. g. the number of neurons in the first hidden layer is approximately proportional to the number of inputs while the average of the numbers of inputs and outputs is used for the second hidden layer. After the adaption is accomplished, several neurons are either added because of the great network error or removed due to the poor generalization and the entire adaptive mode is repeated for this new architecture. To examine the generalization capability, the network error is computed with respect to the so-called *test set* that is a part of the training set which has not been intentionally exploited for learning.

More elaborated methods modify the architecture automatically during the adaptation when it is needed. This represents the combined *architecturally–adaptive* network mode. The following two different approaches are possible. In the so-called *constructive algorithms* [15, 18, 49, 52] one starts with a small topology and the new neurons are added when the error function value cannot be further decreased. On the contrary, in the so-called *pruning algorithms* [23, 28, 37, 44, 70] one issues from a sufficiently rich topology and the connections which happen to have small absolute values of weights during the learning phase, are removed (including the relevant hidden neurons). The architecture modification is feasible in adaptive mode since the network function is robust. The elimination of small weights can be even included in the definition of error function by adding a term which penalizes all weights in the network proportionally to their magnitudes:

$$E'(\mathbf{w}) = E(\mathbf{w}) + \frac{1}{2}\gamma \sum_{j,i} \frac{w_{ji}^2}{1 + w_{ji}^2} \tag{2.24}$$

where parameter $\gamma > 0$ measures the influence of this term on the network error and $E(\mathbf{w})$ is the original error function (2.9).

## 2.3 MADALINE

Another historically important model of neural network is *MADALINE* (**M**ultiple ADALINE) which was invented by Widrow and Hoff [83, 82] (see Section 1.1). The basic element of this model is a neuron called *ADALINE* (**ADA**ptive **LIN**ear **E**lement) which is very similar to the perceptron. Therefore, MADALINE is formally almost identical to the network of perceptrons which has been described in Section 2.1 although it originated from different principles.

The **architectural** dynamics of MADALINE (i. e., the topology) and the notation of network parameters is the same as in the network of perceptrons (see Figure 2.1) except that the perceptron is substituted for ADALINE. The **computational** dynamics of this model differs in the network outputs which are generally real numbers since particular ADALINE's compute only linear functions, i. e. the nonlinear activation function is omitted. Thus, the function $\mathbf{y}(\mathbf{w}) : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ of MADALINE which depends on configuration $\mathbf{w} = (\mathbf{w}_1, \ldots, \mathbf{w}_m)$ where $\mathbf{w}_1 = (w_{10}, \ldots, w_{1n})$,. . ., $\mathbf{w}_m = (w_{m0}, \ldots, w_{mn})$, is defined as the affine combinations of its inputs:

$$y_j = \sum_{i=0}^{n} w_{ji} x_i \quad j = 1, \ldots, m\,. \tag{2.25}$$

Similarly, the geometrical interpretation of the $j$th ADALINE function differs slightly from that of the perceptron. Consider an input $\mathbf{x} = (x_1, \ldots, x_n)$, i. e. a point $[x_1, \ldots, x_n]$ in the $n$-dimensional input space. The hyperplane with coefficients $\mathbf{w}_j$ associated with the $j$th ADALINE is determined by the equation:

$$w_{j0} + \sum_{i=1}^{n} w_{ji} x_i = 0 \,. \tag{2.26}$$

This hyperplane disjoins the input space into two halfspaces in which the corresponding output values $y_j$ (2.25) have different signs. Especially, the output value is zero for the points on this hyperplane. The distance $\varrho_j$ from the point $[x_1, \ldots, x_n]$ to the hyperplane (2.26) is given by the following formula:

$$\varrho_j = \frac{|w_{j0} + \sum_{i=1}^{n} w_{ji} x_i|}{\sqrt{\sum_{i=1}^{n} w_{ji}^2}} = \frac{|y_j|}{\sqrt{\sum_{i=1}^{n} w_{ji}^2}} \,. \tag{2.27}$$

Hence, the absolute value $|y_j|$ of the $j$th ADALINE output depends linearly on the distance from the relevant point to the hyperplane:

$$|y_j| = \sqrt{\sum_{i=1}^{n} w_{ji}^2} \cdot \varrho_j \,. \tag{2.28}$$

The points in the input space corresponding to the same output create a hyperplane that is parallel to the hyperplane (2.26) and which is located at the distance $\varrho_j$ in the direction determined by the sign of $y_j$. This situation is depicted in Figure 2.5 where the hyperplane corresponding to the same output is represented by a dashed line.

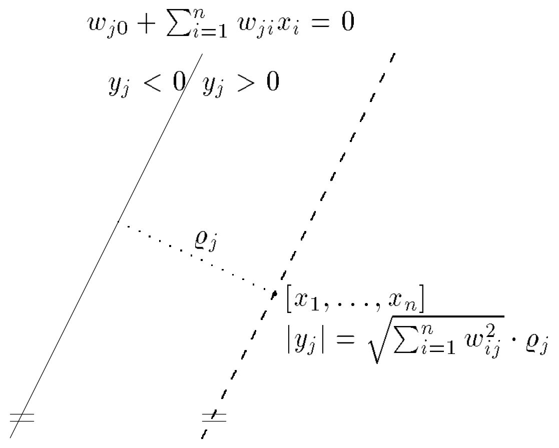
$w_{j0} + \sum_{i=1}^{n} w_{ji} x_i = 0$
$y_j < 0$
$y_j > 0$
$\varrho_j$
$[x_1, \ldots, x_n]$
$|y_j| = \sqrt{\sum_{i=1}^{n} w_{ij}^2} \cdot \varrho_j$

Figure 2.5: Geometric interpretation of ADALINE function.

In **adaptive** mode, the desired MADALINE function is specified by a training sequence where the real inputs $\mathbf{x}_k$ of training patterns are generated randomly with a given probability distribution and each $\mathbf{x}_k$ is labeled with a desired real output $\mathbf{d}_k$:

$$(\mathbf{x}_k, \mathbf{d}_k)_{k=1,2,\ldots} \quad \text{where} \quad \begin{array}{l} \mathbf{x}_k = (x_{k1}, \ldots, x_{kn}) \in \mathbb{R}^n \\ \mathbf{d}_k = (d_{k1}, \ldots, d_{km}) \in \mathbb{R}^m \,. \end{array} \tag{2.29}$$

The error of the $j$th ADALINE with respect to the training sequence (2.29) depends on the partial configuration $\mathbf{w}_j$. It is defined as follows:

$$E_j(\mathbf{w}_j) = \lim_{p\to\infty} \frac{\frac{1}{2}\sum_{k=1}^{p} \left(y_j(\mathbf{w}_j, \mathbf{x}_k) - d_{kj}\right)^2}{p} \qquad j = 1, \ldots, m \tag{2.30}$$

which is a mean value (the expectation operator is denoted by bold $\mathbf{E}$) of a half squared difference between the actual state of the $j$th ADALINE and the corresponding desired output:

$$E_j(\mathbf{w}_j) = \mathbf{E}\left[\frac{1}{2}\left(y_j(\mathbf{w}_j, \mathbf{x}_k) - d_{kj}\right)^2\right] \qquad j = 1, \ldots, m \,. \tag{2.31}$$

The aim of adaptation is to minimize each error function $E_j(\mathbf{w}_j)$ ($j = 1, \ldots, m$) which determines a paraboloid (see (2.25)) in the weight space. For this purpose, the gradient of the error function $E_j(\mathbf{w}_j)$ is computed first from (2.30) by commuting the limit and derivative, by using the rule for composite function derivative and by substituting for the derivative of (2.25):

$$\frac{\partial E_j}{\partial w_{ji}} = \lim_{p\to\infty} \frac{1}{p} \sum_{k=1}^{p} x_{ki} \left(y_j(\mathbf{w}_j, \mathbf{x}_k) - d_{kj}\right) \qquad i = 0, \ldots, n \,. \tag{2.32}$$

The gradient (2.32) can be again expressed as a mean value:

$$\frac{\partial E_j}{\partial w_{ji}} = \mathbf{E}\left[x_{ki}\left(y_j(\mathbf{w}_j, \mathbf{x}_k) - d_{kj}\right)\right] \qquad i = 0, \ldots, n \,. \tag{2.33}$$

After substituting (2.25) into (2.33) the rule for expectation operator of linear function is applied:

$$\frac{\partial E_j}{\partial w_{ji}} = -\mathbf{E}[d_{kj}x_{ki}] + \sum_{r=0}^{n} w_{jr}\mathbf{E}[x_{kr}x_{ki}] \qquad i = 0, \ldots, n \,. \tag{2.34}$$

One of the possible approaches to minimize the error function $E_j(\mathbf{w}_j)$ is to set the partial derivative $\frac{\partial E_j}{\partial w_{ji}} = 0$ equal zero. Providing that the expectation values $\mathbf{E}[d_{kj}x_{ki}]$, $\mathbf{E}[x_{kr}x_{ki}]$ in (2.34) are statistically estimated, the system of linear equations is obtained:

$$\sum_{r=0}^{n} \mathbf{E}[x_{kr}x_{ki}]w_{jr} = -\mathbf{E}[d_{kj}x_{ki}] \qquad i = 0, \ldots, n \,. \tag{2.35}$$

Its solution is the configuration $\mathbf{w}_j^\star$ of the $j$th ADALINE which minimizes the error function $E_j(\mathbf{w}_j)$.

Usually, the gradient method is employed to minimize the error $E_j(\mathbf{w}_j)$ similarly as in the backpropagation algorithm (see Section 2.2). In this case, however, there are no problems with local minima due to the shape of error function (paraboloid). At the beginning of adaptation, at time 0, the weights of configuration $\mathbf{w}^{(0)}$ are initialized randomly in a neighborhood of zero, e. g. $w_{ji}^{(0)} \in \langle -1, 1 \rangle$ ($j = 1, \ldots, m$, $i = 0, \ldots, n$). At the discrete adaptation time $t > 0$, the configuration $\mathbf{w}^{(t)}$ is updated according to the gradient method:

$$w_{ji}^{(t)} = w_{ji}^{(t-1)} + \Delta w_{ji}^{(t)} \quad \text{where} \;\; \Delta w_{ji}^{(t)} = -\varepsilon \frac{\partial E_j}{\partial w_{ji}}(\mathbf{w}^{(t-1)}) \tag{2.36}$$

and $0 < \varepsilon < 1$ is the learning rate.

By the analogy with the backpropagation algorithm the partial derivatives $\frac{\partial E_j}{\partial w_{ji}}$ in (2.36) should be accumulated for the complete training sequence, i. e. the limit (2.32) of their averages should be computed. For MADALINE, however, Widrow and Hoff proposed a learning rule known as *Widrow* or *LMS* (**L**east–**M**ean–**S**quare) *rule* under which the weights are updated after each training pattern is presented (compare with the on-line backpropagation implementation in Paragraph 2.2.4). Thus, the MADALINE model has the discrete adaptive dynamics, i. e. at each time step $t = 1, 2, \ldots$ the $k$th pattern ($k = t$) from the training sequence (2.29) is presented to the network and the weights are adapted with respect to this pattern. Under the Widrow law the increment of configuration $\mathbf{w}^{(t)}$ at time $t > 0$ is defined by the following equation (compare with (2.36) and (2.32)):

$$w_{ji}^{(t)} = w_{ji}^{(t-1)} - \varepsilon x_{ki} \left( y_j(\mathbf{w}_j^{(t-1)}, \mathbf{x}_k) - d_{kj} \right) \qquad \begin{array}{l} j = 1, \ldots, m \\ i = 0, \ldots, n \,. \end{array} \tag{2.37}$$

It is interesting that the MADALINE adaptive dynamics (2.37) is formally identical with the perceptron learning rule (2.5). Widrow and Hoff proved that the adaptive process under (2.37) converges from an arbitrary initial configuration $\mathbf{w}^{(0)}$ to the configuration $\mathbf{w}^\star$ which minimizes the error functions $E_j(\mathbf{w}_j)$ ($j = 1, \ldots, m$).

There exist various variants of the MADALINE adaptive dynamics. For example, the weight adaptation may be performed according to the gradient method (2.36) so that the limiting value of partial derivative $\frac{\partial E_j}{\partial w_{ji}}$ in (2.32) is approximated by the average over $p$ training patterns:

$$\frac{\partial E_j}{\partial w_{ji}} \doteq \frac{1}{p} \sum_{k=1}^{p} x_{ki} \left( y_j(\mathbf{w}_j, \mathbf{x}_k) - d_{kj} \right) . \tag{2.38}$$

The Widrow rule may also be extended with a momentum term (see formula (2.23)), i. e. for $t > 1$ we obtain

$$w_{ji}^{(t)} = w_{ji}^{(t-1)} - \varepsilon x_{ki} \left( y_j(\mathbf{w}_j^{(t-1)}, \mathbf{x}_k) - d_{kj} \right) + \alpha \Delta w_{ji}^{(t-1)} \tag{2.39}$$

where $\Delta w_{ji}^{(t-1)} = w_{ji}^{(t-1)} - w_{ji}^{(t-2)}$ denotes the preceding weight correction and $0 < \alpha < 1$ is the momentum rate.

# Chapter 3

# Associative Neural Networks

## 3.1 Linear Associator Neural Network

A *linear associator neural network*, introduced by Anderson [7] and further refined by him and Kohonen [8, 9, 10, 40, 41], is an example of the neural network model that is exploited as an *associative* (*data-addressed*) *memory*. In contrast with a classical computer memory where the key to search an item is its address, in a data-addressed memory the required information is recalled on the basis of its partial knowledge (association). For example, in database applications the information about some of the items in a record is sufficient to find the corresponding complete record. Similarly, in the human analogy a man is able to recall his friend's eyes colour or name when seeing his black-and-white photograph. Basically, we will distinguish two types of data-addressed memories: *autoassociative* and *heteroassociative* memories. In the autoassociative memory the input information is completed or even reconstructed while in the heteroassociative memory a certain data, associated with the input, is recalled. In the above-mentioned motivational example with a black-and-white photograph, this corresponds to reconstructing the coloured picture (autoassociation) and recalling the name of a person in the photograph (heteroassociation), respectively.

The **architectural** and **computational** dynamics of linear associator is almost identical with that of MADALINE which has been described in Section 2.3. The only difference consists in the computational mode of linear associator network which computes linear combinations of inputs instead of affine combinations, i. e. the respective biases are zero and the formal unit input is omitted. Formally, the network function $\mathbf{y}(\mathbf{w}) : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ can be expressed as follows (compare with (2.25)):

$$y_j = \sum_{i=1}^{n} w_{ji} x_i \quad j = 1, \ldots, m\,. \tag{3.1}$$

In the geometrical interpretation, this means that the respective hyperplanes (see Figure 2.5) corresponding to the output neurons in the network pass through the origin.

For the need of further exposition, the computational dynamics (3.1) is formally rewritten into matrix notation. The network input and output are regarded as column vectors $\mathbf{x} = (x_1, \ldots, x_n)$ of size $n \times 1$ and $\mathbf{y} = (y_1, \ldots, y_m)$ of length $m \times 1$, respectively. Similarly, the network configuration is given by a $m \times n$ weight matrix $\mathbf{W}$ whose rows

$\mathbf{w}_j = (w_{j1}, \ldots, w_{jn})$ correspond to the input synaptic weights of (output) neurons $j$ $(j = 1, \ldots, m)$:

$$\mathbf{W} = \begin{pmatrix} w_{11} & \ldots & w_{1n} \\ \vdots & \ddots & \vdots \\ w_{m1} & \ldots & w_{mn} \end{pmatrix} . \tag{3.2}$$

In this notation the computational dynamics (3.1) of linear associator can formally be described as a matrix product:

$$\mathbf{y} = \mathbf{W}\mathbf{x} . \tag{3.3}$$

In the **adaptive** mode the desired function of the linear associator network is again given by a training set:

$$\mathcal{T} = \left\{ (\mathbf{x}_k, \mathbf{d}_k) \,\middle|\, \begin{array}{l} \mathbf{x}_k = (x_{k1}, \ldots, x_{kn}) \in \mathbb{R}^n \\ \mathbf{d}_k = (d_{k1}, \ldots, d_{km}) \in \mathbb{R}^m \end{array} \; k = 1, \ldots, p \right\} . \tag{3.4}$$

Especially, in the case of autoassociative memory the desired output equals the input pattern, i. e. $m = n$ and $\mathbf{x}_k = \mathbf{d}_k$ for $k = 1, \ldots, p$. In following Paragraphs 3.1.1 and 3.1.2, two possible adaptive dynamics of the linear associator will be introduced.

### 3.1.1 Adaptation under Hebb Law

In this paragraph the adaptive dynamics of the linear associator will generally be described for the case of heteroassociative memory which includes the autoassociative network as a special case. One of the possible ways of linear associator adaptation is motivated by the neurophysiological *Hebb law* saying that the change of synaptic weight associated with a connection between two neurons is proportional to their consonant activities expressed by the product of their states. In this way, Donald Hebb tried to explain the rise of the conditioned reflex [25] (see Section 1.1) when the almost simultaneous activity (or passivity) of the first neuron corresponding to a condition (premise) and of the second neuron compelling the reflex, strengthens the synaptic link in the direction from the first neuron to the second one. On the contrary, the antithetical activity of these two neurons weakens the respective connection.

Hebb law can formally be summarized in the following adaptive dynamics of the linear associator network. At the beginning of adaptation, at time 0, all weights in the configuration are zero, i. e. $w_{ji}^{(0)} = 0$ $(j = 1, \ldots, m,\ i = 1, \ldots, n)$. At the discrete adaptation time $t = 1, \ldots, p$, the $k$th training pattern $(k = t)$ is presented to the network and the weights are updated according to Hebb law:

$$w_{ji}^{(t)} = w_{ji}^{(t-1)} + d_{kj} x_{ki} \qquad \begin{array}{l} j = 1, \ldots, m \\ i = 1, \ldots, n . \end{array} \tag{3.5}$$

Since the adaptive phase terminates in this case after $p$ steps when all training patterns are learnt, the resulting configuration can be expressed as a finite sum:

$$w_{ji} = \sum_{k=1}^{p} d_{kj} x_{ki} \qquad \begin{array}{l} j = 1, \ldots, m \\ i = 1, \ldots, n . \end{array} \tag{3.6}$$

The adaptive dynamics (3.5) of the linear associator can also lucidly be written in the matrix notation:

$$\mathbf{W}^{(0)} = \mathbf{0}, \quad \mathbf{W}^{(k)} = \mathbf{W}^{(k-1)} + \mathbf{d}_k \mathbf{x}_k^{\mathsf{T}}, \quad k = 1, \ldots, p \tag{3.7}$$

where $^{\mathsf{T}}$ denotes matrix transposition, $\mathbf{0}$ is the null matrix (i. e. all its items are zero) and the weight matrix $\mathbf{W}^{(k)}$ determines the network configuration at adaptation time $t = k$. The resulting configuration can again be described as a matrix product (compare with (3.6)):

$$\mathbf{W} = \mathbf{W}^{(p)} = \sum_{k=1}^{p} \mathbf{d}_k \mathbf{x}_k^{\mathsf{T}} = \mathbf{D}\mathbf{X}^{\mathsf{T}} \tag{3.8}$$

where the columns of a $n \times p$ matrix $\mathbf{X}$ and of a $m \times p$ matrix $\mathbf{D}$ are sample inputs $\mathbf{x}_k$ and desired outputs $\mathbf{d}_k$ ($k = 1, \ldots, p$), respectively, from training patterns (3.4), i. e.

$$\mathbf{X} = \begin{pmatrix} x_{11} & \ldots & x_{p1} \\ \vdots & \ddots & \vdots \\ x_{1n} & \ldots & x_{pn} \end{pmatrix}, \quad \mathbf{D} = \begin{pmatrix} d_{11} & \ldots & d_{p1} \\ \vdots & \ddots & \vdots \\ d_{1m} & \ldots & d_{pm} \end{pmatrix}. \tag{3.9}$$

Especially, the adaptive dynamics for the case of autoassociative memory where $\mathbf{D} = \mathbf{X}$ can be rewritten:

$$\mathbf{W} = \mathbf{X}\mathbf{X}^{\mathsf{T}}. \tag{3.10}$$

We will further assume that the input vectors $\{\mathbf{x}_1, \ldots, \mathbf{x}_p\}$ of training patterns (3.4) are orthonormal (necessarily $p \leq n$). This means that these vectors are mutually orthogonal, i. e. $\mathbf{x}_r^{\mathsf{T}} \mathbf{x}_s = 0$ for $r \neq s$ ($1 \leq r, s \leq p$), and, at the same time, they are of the unit length (normalized), i. e. $\mathbf{x}_r^{\mathsf{T}} \mathbf{x}_r = 1$ ($r = 1, \ldots, p$). This assumption can be interpreted in such a way that the particular input patterns differ substantially from each other due to their orthogonality and they are comparable thanks to their unit length. Under this condition the linear associator has the property of a so-called *reproduction*, i. e. the network responds with the respective desired output $\mathbf{d}_r$ when the corresponding stimulus $\mathbf{x}_r$ ($1 \leq r \leq p$) from the training set appears at the input. This can be shown by substituting (3.8) for $\mathbf{W}$ into the computational dynamics (3.3) and, in addition, the associativeness and distributivity of matrix multiplication (inner product) is exploited together with the orthonormality assumption:

$$\mathbf{y}(\mathbf{x}_r) = \mathbf{W}\mathbf{x}_r = \left( \sum_{k=1}^{p} \mathbf{d}_k \mathbf{x}_k^{\mathsf{T}} \right) \mathbf{x}_r = \sum_{k=1}^{p} \mathbf{d}_k \left( \mathbf{x}_k^{\mathsf{T}} \mathbf{x}_r \right) = \mathbf{d}_r. \tag{3.11}$$

The reproduction property (3.11) can be considered as a necessary condition for associative memories. Moreover, the linear associator network should also respond with the desired output $\mathbf{d}_r$ even for a stimulus $\mathbf{x}_r + \boldsymbol{\delta}$ that is nearby the respective training pattern input $\mathbf{x}_r$ (i. e. the norm $\|\boldsymbol{\delta}\| = \sqrt{\sum_{i=1}^{n} \delta_i^2} = \delta$ is sufficiently small). The corresponding error with respect to this requirement can be expressed as the norm of the difference between the actual output for input $\mathbf{x}_r + \boldsymbol{\delta}$ and the desired output $\mathbf{d}_r$:

$$E_r(\boldsymbol{\delta}) = \|\mathbf{y}(\mathbf{x}_r + \boldsymbol{\delta}) - \mathbf{d}_r\| = \|\mathbf{W}\mathbf{x}_r + \mathbf{W}\boldsymbol{\delta} - \mathbf{d}_r\| = \|\mathbf{W}\boldsymbol{\delta}\|. \tag{3.12}$$

In addition, suppose that the desired outputs $\mathbf{d}_r$ of training patterns are normalized as well, i. e. $\mathbf{d}_r^\mathsf{T}\mathbf{d}_r = 1$ $(r = 1, \ldots, p)$, which has already been true for the autoassociative memory since $\mathbf{x}_r = \mathbf{d}_r$ are assumed to be orthonormal. Then the error $E_r(\boldsymbol{\delta})$ from (3.12) can be upper bounded by using the triangular ($\|\mathbf{x}+\mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$) and Cauchy-Schwarz ($|\mathbf{x}^\mathsf{T}\mathbf{y}| \leq \|\mathbf{x}\| \cdot \|\mathbf{y}\|$) inequalities:

$$E_r(\boldsymbol{\delta}) = \|\mathbf{W}\boldsymbol{\delta}\| \leq \sum_{k=1}^{p} \|\mathbf{d}_k\mathbf{x}_k^\mathsf{T}\boldsymbol{\delta}\| \leq \sum_{k=1}^{p} \|\mathbf{d}_k\| \cdot \|\mathbf{x}_k\| \cdot \|\boldsymbol{\delta}\| = p\delta \leq n\delta\,. \tag{3.13}$$

Hence, $E_r(\boldsymbol{\delta}) \to 0$ for $\delta \to 0$ and the linear associator neural network responds approximately with the desired output for the stimuli close to the sample input, i. e. it can be used as an associative memory.

### 3.1.2 Pseudohebbian Adaptation

The reproduction property of the linear associator that has been adapted under Hebb law in Paragraph 3.1.1 requires an additional assumption regarding the orthonormality of training pattern inputs. Therefore, a modification of Hebb rule, so-called *pseudohebbian adaptation* [79, 43] was proposed to weaken this assumption by means of mathematical manipulations. However, this modified approach does not correspond to neurophysiological reality anymore. For the simplicity reasons, this alternative adaptive dynamics of the linear associator neural network will first be described for the autoassociative memory and then, it will be generalized for the heteroassociative case.

Suppose that the set of input vectors $\{\mathbf{x}_1, \ldots, \mathbf{x}_p\}$ from the training set (3.4) (coinciding with the desired outputs in the autoassociative case) is linearly independent (necessarily $p \leq n$) and hence, it forms a basis of vector space $V_p$ which is a subspace of $\mathbb{R}^n$. Because of the reproduction property an orthogonal basis $\{\mathbf{z}_1, \ldots, \mathbf{z}_p\}$ of vector space $V_p$ is being created by Gram-Schmidt orthogonalization process during the pseudohebbian adaptation. Thus, the adaptive pseudohebbian dynamics of the linear associator neural network proceeds in the following way. At the beginning of adaptation, at time 0, the weight matrix is zero, i. e. $\mathbf{W}^{(0)} = \mathbf{0}$. At the discrete adaptation time $t = 1, \ldots, p$ when the $k$th training pattern $(k = t)$ is presented to the network, first, an $n \times 1$ column vector $\mathbf{z}_k$ is determined:

$$\mathbf{z}_k = \mathbf{x}_k - \mathbf{W}^{(k-1)}\mathbf{x}_k \tag{3.14}$$

which then serves for the weight matrix update:

$$\mathbf{W}^{(k)} = \mathbf{W}^{(k-1)} + \frac{\mathbf{z}_k\mathbf{z}_k^\mathsf{T}}{\mathbf{z}_k^\mathsf{T}\mathbf{z}_k}\,. \tag{3.15}$$

Again, the resulting weight matrix can formally be expressed as a finite sum of matrices from (3.15) which can be reduced to a matrix product (compare with (3.10)):

$$\mathbf{W} = \mathbf{W}^{(p)} = \sum_{k=1}^{p} \frac{\mathbf{z}_k\mathbf{z}_k^\mathsf{T}}{\mathbf{z}_k^\mathsf{T}\mathbf{z}_k} = \mathbf{X}\mathbf{X}^+ \tag{3.16}$$

where $\mathbf{X}$ is the matrix from (3.9) and

$$\mathbf{X}^{+} = (\mathbf{X}^{\mathsf{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathsf{T}} \tag{3.17}$$

is its *pseudoinverse.*

The geometrical interpretation of pseudohebbian adaptation (3.14), (3.15) at the $k$th step is depicted in Figure 3.1. We will prove by mathematical induction on $k$

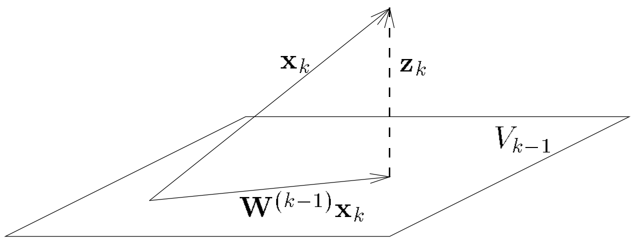

Figure 3.1: Geometrical interpretation of pseudohebbian strategy.

that the vector $\mathbf{W}^{(k-1)}\mathbf{x}_k$ is the orthogonal projection of $\mathbf{x}_k$ into the vector space $V_{k-1}$ that is determined by its basis $\{\mathbf{x}_1, \ldots, \mathbf{x}_{k-1}\}$ or equivalently, by its orthogonal basis $\{\mathbf{z}_1, \ldots, \mathbf{z}_{k-1}\}$. Notice that the vector $\mathbf{x}_k$ is not included in the space $V_{k-1}$ since the vectors $\{\mathbf{x}_1, \ldots, \mathbf{x}_p\}$ are linearly independent. Thus, we want to verify that the vector $\mathbf{z}_k = \mathbf{x}_k - \mathbf{W}^{(k-1)}\mathbf{x}_k$ is perpendicular to the basis vectors $\mathbf{z}_r$ ($r = 1, \ldots, k-1$). This means, we want to prove $\mathbf{z}_k^{\mathsf{T}}\mathbf{z}_r = 0$ for $r = 1, \ldots, k-1$. After transposing, $\sum_{s=1}^{k-1} \frac{\mathbf{z}_s\mathbf{z}_s^{\mathsf{T}}}{\mathbf{z}_s^{\mathsf{T}}\mathbf{z}_s}$ from (3.15) is substituted for $(\mathbf{W}^{(k-1)})^{\mathsf{T}}$ in $\mathbf{z}_k^{\mathsf{T}} = \mathbf{x}_k^{\mathsf{T}} - \mathbf{x}_k^{\mathsf{T}}(\mathbf{W}^{(k-1)})^{\mathsf{T}}$. Furthermore, we know that $\{\mathbf{z}_1, \ldots, \mathbf{z}_{k-1}\}$ is orthogonal according to the inductive hypothesis and hence, we obtain:

$$\mathbf{z}_k^{\mathsf{T}}\mathbf{z}_r = \mathbf{x}_k^{\mathsf{T}}\mathbf{z}_r - \sum_{s=1}^{k-1} \frac{\mathbf{x}_k^{\mathsf{T}}\left(\mathbf{z}_s\mathbf{z}_s^{\mathsf{T}}\right)\mathbf{z}_r}{\mathbf{z}_s^{\mathsf{T}}\mathbf{z}_s} = \mathbf{x}_k^{\mathsf{T}}\mathbf{z}_r - \frac{\mathbf{x}_k^{\mathsf{T}}\mathbf{z}_r\left(\mathbf{z}_r^{\mathsf{T}}\mathbf{z}_r\right)}{\mathbf{z}_r^{\mathsf{T}}\mathbf{z}_r} = 0\,. \tag{3.18}$$

It follows from above that the vector $\mathbf{x}$ coincides with its orthogonal projection $\mathbf{W}\mathbf{x} = \mathbf{x}$ when it lies in the space $V_p$. Especially, $\mathbf{y}(\mathbf{x}_r) = \mathbf{W}\mathbf{x}_r = \mathbf{x}_r$ for input basis vectors $\{\mathbf{x}_1, \ldots, \mathbf{x}_p\}$. This means that the linear autoassociator neural network arisen from the pseudohebbian adaptation (3.14), (3.15) has the reproduction property. Moreover, the network output $\mathbf{y}(\mathbf{x}_r + \boldsymbol{\delta}) = \mathbf{W}(\mathbf{x}_r + \boldsymbol{\delta})$ for the input $\mathbf{x}_r + \boldsymbol{\delta}$ nearby the $r$th training pattern $\mathbf{x}_r$, is its orthogonal projection into the space $V_p$, i. e. its best approximation in the space $V_p$. After substituting (3.16) for $\mathbf{W}$, the error $E_r(\boldsymbol{\delta}) = \|\mathbf{y}(\mathbf{x}_r + \boldsymbol{\delta}) - x_r\| = \|\mathbf{W}\mathbf{x}_r + \mathbf{W}\boldsymbol{\delta} - x_r\| = \|\mathbf{W}\boldsymbol{\delta}\|$ can be upper bounded similarly as in (3.13):

$$E_r(\boldsymbol{\delta}) = \|\mathbf{W}\boldsymbol{\delta}\| \leq \sum_{k=1}^{p} \frac{\|\mathbf{z}_k\mathbf{z}_k^{\mathsf{T}}\boldsymbol{\delta}\|}{\|\mathbf{z}_k\|^2} \leq p\delta \leq n\delta \tag{3.19}$$

where $\delta = \|\delta\|$, and hence, $E_r(\boldsymbol{\delta}) \to 0$ for $\delta \to 0$.

In conclusion, the pseudohebbian adaptive dynamics (3.16) of the linear associator is generalized for heteroassociative memory:

$$\mathbf{W} = \mathbf{D}\mathbf{X}^{+} = \mathbf{D}(\mathbf{X}^{\mathsf{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathsf{T}} \tag{3.20}$$

where $\mathbf{D}, \mathbf{X}$ are matrices from (3.9). The above-mentioned matrix notation of adaptive dynamics (3.20) is not suitable for a distributed adaptive mode of the heteroassociative network. The analogy of recursive notation (3.14), (3.15) in the heteroassociative case issues from Greville's theorem [22, 42]:

$$\mathbf{W}^{(k)} = \mathbf{W}^{(k-1)} + \frac{\left(\mathbf{d}_k - \mathbf{W}^{(k-1)}\mathbf{x}_k\right)\mathbf{z}_k^{\mathsf{T}}}{\mathbf{z}_k^{\mathsf{T}}\mathbf{z}_k} \tag{3.21}$$

where $\mathbf{z}_k$ is the same $n \times 1$ column vector as in the autoassociative case, i. e. in its definition (3.14) the matrix $\mathbf{W}^{(k-1)}$ is computed according to (3.15) (not by (3.21)). By using the pseudoinverse (3.17), $\mathbf{z}_k$ can alternatively be expressed as follows:

$$\mathbf{z}_k = \mathbf{x}_k - \mathbf{X}^{(k-1)}\left(\mathbf{X}^{(k-1)}\right)^{+}\mathbf{x}_k \tag{3.22}$$

where $\mathbf{X}^{(k-1)}$ is an $n \times (k-1)$ matrix, whose columns are the input vectors $\mathbf{x}_1, \ldots, \mathbf{x}_{k-1}$ of the first $k-1$ training patterns (3.4):

$$\mathbf{X}^{(k-1)} = \begin{pmatrix} x_{11} & \ldots & x_{k-1,1} \\ \vdots & \ddots & \vdots \\ x_{1n} & \ldots & x_{k-1,n} \end{pmatrix}. \tag{3.23}$$

Besides the proper heteroassociative network adapted by (3.21), the computation of $\mathbf{z}_k$ according to (3.14) requires an extra autoassociative network model with the weight matrix from (3.15). Therefore, the pseudohebbian adaptive dynamics (3.20) for heteroassociative memory (i. e. the computation of $\mathbf{W} = \mathbf{D}\mathbf{X}^{+}$) is sometimes approximated by Widrow rule (2.37).

Also the pseudohebbian adaptive dynamics (3.20) ensures the capability of linear heteroassociator to reproduce the training patterns (3.4). For the reproduction property verification, $[\mathbf{X}]_r$ denotes the $r$th column of matrix $\mathbf{X}$:

$$\mathbf{y}(\mathbf{x}_r) = \mathbf{W}\mathbf{x}_r = \mathbf{D}\mathbf{X}^{+}[\mathbf{X}]_r = \mathbf{D}[(\mathbf{X}^{\mathsf{T}}\mathbf{X})^{-1}(\mathbf{X}^{\mathsf{T}}\mathbf{X})]_r = \mathbf{d}_r\,. \tag{3.24}$$

## 3.2 Hopfield Network

Another important model of the autoassociative neural network which is subjected to researchers' great interest is a *Hopfield network*. This model had already been introduced by McCulloch and Pitts [50] and, later on, analyzed by Amari [4], W. A. Little, and G. L. Shaw [46]. However, only thanks to Hopfield [31] who exploited a lucid analogy with physical theories of magnetic materials to analyze the stability of this network, this model became widely known (see Section 1.1) and that is why it also bears his name now. The Hopfield network can be exploited as an autoassociative memory (see Section 3.1). At present times there exist many theoretical results and variants of this model that make an effort to improve its properties. In this section we confine ourselves to a description and discussion concerning a basic model of Hopfield network.

### 3.2.1 Basic Model

At the beginning, the **architectural** dynamics of Hopfield network specifies a fixed complete topology of the cyclic network with $n$ neurons. In this architecture, each neuron is connected with all neurons in the network which represent its inputs. In addition, all neurons in the network serve simultaneously as input and output neurons. The topology of Hopfield network is depicted in Figure 3.2. Furthermore, denote by

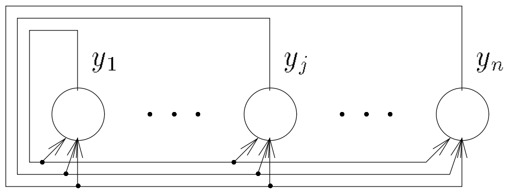

Figure 3.2: The topology of Hopfield network.

$\xi_1, \ldots, \xi_n \in \mathbb{Z}$ the integer excitation levels and $y_1, \ldots, y_n \in \{-1, 1\}$ are the bipolar states of all neurons. Each network connection from neuron $i$ ($i = 1, \ldots, n$) to neuron $j$ ($j = 1, \ldots, n$) is labeled with an integer synaptic weight $w_{ji} \in \mathbb{Z}$. In the basic model, the biases are omitted, i. e. all thresholds are zero. Similarly, no neuron is linked to itself, i. e. the respective weights $w_{jj} = 0$ ($j = 1, \ldots, n$) are zero as well.

For the Hopfield network, the **adaptive** dynamics which is based on Hebb law (see Paragraph 3.1.1) will be described at first. The desired network function is again specified by a training set of $p$ patterns (3.25), each being given by a vector of $n$ bipolar states of the input and output neurons that coincide in autoassociative memories.

$$\mathcal{T} = \{\mathbf{x}_k \mid \mathbf{x}_k = (x_{k1}, \ldots, x_{kn}) \in \{-1,1\}^n,\ k = 1, \ldots, p\} \tag{3.25}$$

The adaptive phase under Hebb law proceeds in $p$ discrete steps in which the training patterns are, one by one, presented to the network whose synaptic weights are adapted with respect to them. The resulting configuration can be described as follows (compare with (3.6)):

$$w_{ji} = \sum_{k=1}^{p} x_{kj} x_{ki} \quad 1 \leq j \neq i \leq n\,. \tag{3.26}$$

First notice that $w_{ji} = w_{ij}$ ($1 \leq i, j \leq n$) since the positions of neurons $i, j$ in formula (3.26) are symmetric. Therefore, the Hopfield network is also sometimes called a *symmetric neural network* in which two oppositely oriented connections between two neurons may be considered as one undirected link. The Hopfield network adaptation according to Hebb law (3.26) can be interpreted as a voting of patterns on mutual neuron bindings. Namely, the weight $w_{ji} = w_{ij}$ represents the difference between the number of agreeing states $x_{kj} = x_{ki}$ (i. e. $x_{kj}x_{ki} = 1$) of neurons $i$ and $j$ in the training patterns which strengthen the respective connection, and the number of disagreeing states $x_{kj} \neq x_{ki}$ (i. e. $x_{kj}x_{ki} = -1$) which weaken this link. The result of voting is expressed by the sign of weight $w_{ij}$ which is positive if the number of consonant states of neurons $i, j$ in the training set (3.25) outweighs the number of differing states, and it is negative in the opposite case. The absolute value of this weight determines by how many votes the winning side exceeded the loosing one. Obviously, the training patterns are not stored directly and explicitly in the Hopfield network but they are represented rather by means of relations among neuron states.

The **computational** dynamics of Hopfield network will be described for the case of sequential synchronous computations. At the beginning of computational mode, at time 0, the neuron states are assigned to the network input $\mathbf{x} = (x_1, \ldots, x_n)$, i. e. $y_i^{(0)} = x_i$ $(i = 1, \ldots, n)$. At the discrete computational time $t > 0$, only the state of one neuron $j$ is updated while the remaining neurons do not change their states. For example, this neuron is selected systematically so that $t = \tau n + j$ where $\tau$ is a so-called *macroscopic time* counting the number of epochs in which all neurons are updated. At first, the integer excitation level of neuron $j$ is computed:

$$\xi_j^{(t-1)} = \sum_{i=1}^{n} w_{ji} y_i^{(t-1)} \qquad (3.27)$$

whose sign determines its new bipolar state:

$$y_j^{(t)} = \begin{cases} 1 & \xi_j^{(t-1)} \geq 0 \\ -1 & \xi_j^{(t-1)} < 0 \,. \end{cases} \qquad (3.28)$$

Alternatively, the rule (3.28) can be modified so that the state $y_j^{(t)} = y_j^{(t-1)}$ of neuron $j$ remains unchanged for $\xi_j^{(t-1)} = 0$ instead of preferring the unit state $y_j^{(t)} = 1$ in this boundary case as the dynamics (3.28) suggests. The neuron output determined from the excitation level (3.27) according to (3.28) may, in fact, be interpreted as the application of an activation function that is a bipolar version of hard limiter (1.7). The computation of Hopfield network terminates at time $t^\star$ when the network finds itself in a so-called *stable state*, which means that the neuron states do not change anymore: $y_j^{(t^\star+n)} = y_j^{(t^\star)}$ $(j = 1, \ldots, n)$. At that time the (output) neuron states represent the network output $\mathbf{y} = (y_1, \ldots, y_n)$ where $y_j = y_j^{(t^\star)}$ $(j = 1, \ldots, n)$. It can be proved that under the weight symmetry assumption, the sequential computation of the Hopfield network controlled by the computational dynamics (3.27), (3.28) terminates for any input. Thus, in the computational mode the Hopfield network computes a function $\mathbf{y}(\mathbf{w}) : \{-1, 1\}^n \longrightarrow \{-1, 1\}^n$ in the input space. Besides the configuration $\mathbf{w}$, this function depends on the order of neuron updates.

Also a parallel computation of the Hopfield network may be considered when, at one computational step, the states of more neurons are simultaneously updated according to (3.27), (3.28). However, in this case the computation may generally not terminate and the network may alter two different states after some time. Or in the asynchronous model of the Hopfield network, the individual neurons are independently updated in a random order instead of the systematic (sequential or parallel) neuron updates.

### 3.2.2 Energy Function

As it has been already mentioned in the beginning of this section, the Hopfield network has a natural physical analogy. Some simple models of magnetic materials in statistical mechanics (e. g. spin glasses) can be viewed as the Hopfield network. A magnetic material in these models can be described as a set of atomic magnets, so-called *spins* which correspond to neurons in the Hopfield network. These spins are arranged on a regular lattice representing the crystal structure of the materials as it is depicted in Figure 3.3. The simplest case of atoms is considered in which each spin may have two

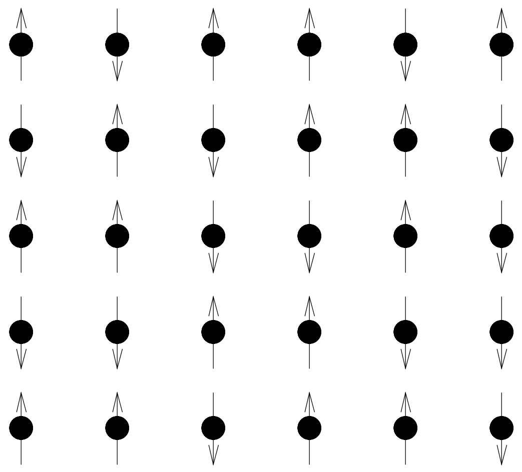

Figure 3.3: A simplified model of magnetic material.

distinct magnetic orientations which are modelled by the bipolar neuron states 1 and $-1$ in the Hopfield network. The physical model is further described by interactions and dynamics of spins. Each spin is influenced by a surrounding magnetic field which can be divided into an external field that corresponds to the input of the Hopfield network, and to the internal one produced by the other spins. The contributions of each atom to the surrounding internal field are proportional to its spin. Thus, the influence of the magnetic field on a given spin is determined by the sum of respective contributions which corresponds to the computational dynamics (3.27), (3.28) of the Hopfield network. The synaptic weights in formula (3.27) measure the strengths of mutual interactions between spins. These coefficients are necessarily symmetric in the physical model and they may even be distinct, positive or negative depending on the macroscopic properties of the material. Obviously, the physical model corresponds to the asynchronous Hopfield network.

For a better understanding of computational dynamics (3.27), (3.28), Hopfield defined a so-called *energy function* $E(\mathbf{y})$ by the analogy with physical processes. This function associates each network state $\mathbf{y} \in \{-1,1\}^n$ with its potential *energy* according to the following quadratic form:

$$E(\mathbf{y}) = -\frac{1}{2}\sum_{j=1}^{n}\sum_{i=1}^{n} w_{ji}y_jy_i\,. \tag{3.29}$$

It follows from the definition of energy function $E(\mathbf{y})$ that the network states with a low energy (i. e. the respective addition terms $w_{ji}y_jy_i$ in the sum (3.29) are sufficiently large, e. g. positive) have a greater stability since the sign of the weight $w_{ji}$ associated with the connection between neurons $j$ and $i$ is consistent with the mutual relation between their states $y_j$ a $y_i$. Namely, we know that the positive weight compels the agreeing neuron states while the negative one forces these neuron states to disagree. On the contrary, the states with a high energy are unstable from the same reason. Thanks to this property, the energy function with the opposite sign is sometimes called the *stability* or *harmony*.

For illustrational reasons, we will assume in the following exposition that the energy function is continuous although only discrete values are considered in the above-introduced basic model of the Hopfield network. At the beginning of the computational mode, the Hopfield network is supplied with energy $E(\mathbf{y}^{(0)})$ by means of the initial network state $\mathbf{y}^{(0)} = \mathbf{x}$ which is the network input. In the course of computation this energy is consumed, i. e. the energy function $E(\mathbf{y}^{(t)}) \geq E(\mathbf{y}^{(t+1)})$ is decreasing in time until the network reaches a stable state $\mathbf{y}^{(t^\star)}$ at time $t^\star$, corresponding to a local minimum of energy function $E(\mathbf{y}^{(t^\star)})$. It is interesting that the decrease of energy function (3.29) according to the computational dynamics (3.27), (3.28) is similar to a minimization of 'error' $E(\mathbf{y})$ by a gradient method. In particular, at computational time $t > 0$, the new state $y_j^{(t)}$ of a selected neuron $j$ corresponds to the sign of negative gradient (3.30) at the point $y_j^{(t-1)}$.

$$-\frac{\partial E}{\partial y_j}(y_j^{(t-1)}) = \sum_{i=1}^{n} w_{ji}y_i^{(t-1)} \tag{3.30}$$

In comparison with the multilayered neural network adapted by the backpropagation learning algorithm (see section 2.2), the Hopfield network has an opposite character of the computational and adaptive dynamics and therefore, we have described them in the opposite order. While the Hopfield network adaptation under Hebb law (3.26) represents an unrepeated procedure whose lasting depends only on the number of training patterns, the backpropagation learning algorithm (see Paragraph 2.2.4) realizes an iterative process to minimize the network error by the gradient method without a convergence guarantee. On the other hand, the time consumed for the computational phase of a multilayered network is given only by the number of layers (see Paragraph 2.2.1) while the computational mode of Hopfield network according to (3.27), (3.28) is an iterative process minimizing the network energy by a discrete variant of the gradient method which may generally (e. g. parallel computations) not converge.

The objective of the Hopfield network adaptation under Hebb law (3.26) is to find a network configuration such that the respective network function implements an autoassociative memory during the computational mode. Thus, the Hopfield network should output a training pattern for all inputs that are close to this pattern. From the energy point of view, each training pattern in (3.25) should be a local minimum of energy function, i. e. a stable state of the network. All inputs that are closed to the respective pattern are situated in a near neighborhood of this stable state and they form a so-called *attraction area*. The attraction area of a stable state is a set of the initial network states (inputs) such that starting with them, the Hopfield network reaches this stable state (output) in the computational phase, which corresponds to a local minimum representing the underlying training pattern. In the geometrical interpretation, the energy surface is decomposed into the attraction areas of particular local minima and the Hopfield network function maps each input from an attraction area of some local minimum exactly on this minimum. The energy surface is graphically depicted in Figure 3.4 where the local minima of particular attraction areas are marked.

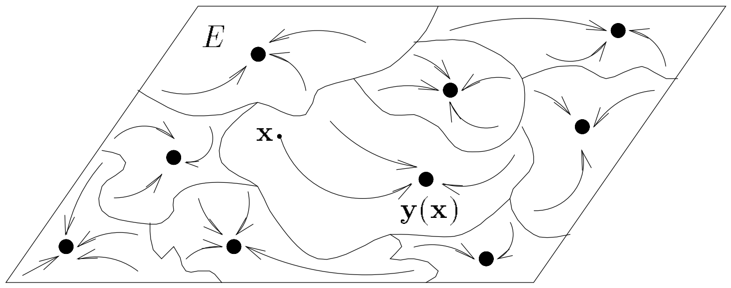

Figure 3.4: Energy surface.

However, as a result of the Hebbian learning (3.26), additional local minima, so-called stable *spurious states* spontaneously arise on the energy surface of the Hopfield network that do not correspond to any training patterns. The output that is produced for the input sufficiently close to such a spurious state, does not represent any pattern and thus, it does not make any sense. There exist variants of the Hopfield network adaptive dynamics in which the spurious states are being additionally unlearned. For example, in the Hopfield network with a configuration $w_{ji}$ $(1 \leq j, i \leq n)$, a spurious state $\mathbf{x}' = (x'_1, \ldots, x'_n) \in \{-1, 1\}^n$ can be unlearned by a modification of Hebb law [33] that produces new weights $w'_{ji}$ as follows:

$$w'_{ji} = w_{ji} - x'_j x'_i \quad 1 \leq j \neq i \leq n\,. \tag{3.31}$$

In the neurophysiological analogy, delivering the neural network of spurious states might evoke a neurosis treatment. There are even hypotheses assuming that a man improves his memory by unlearning the spurious patterns in a dream.

### 3.2.3 Capacity of Hopfield Memory

We will first deal with the reproduction property of the Hopfield network which represents a necessary condition for autoassociative memories (see Paragraph 3.1.1). In this case, the training patterns should be local minima of the energy function, i. e. the network stable states. The reproduction property depends on the ratio $p/n$ of the number $p$ of training patterns to the number $n$ of neurons. This fraction also determines a so-called *capacity* of the Hopfield autoassociative memory provided that all patterns are stored without unacceptable errors. Supposing that the neuron states prescribed by random training patterns are independently chosen with equal probability for both bipolar values, then for a sufficiently large numbers $n$, $p$ of neurons and training patterns, respectively, a probability $P$ that a state of a given neuron in the pattern is stable, can be estimated [27]:

$$P = \frac{1}{2} - \frac{1}{\sqrt{\pi}} \int_{0}^{\sqrt{n/2p}} e^{-x^2} dx \,. \tag{3.32}$$

For example, it is expected for $p = 0.185n$ that the number of unstable neuron states in training patterns do not exceed 1%. However, this result does not say anything about a subsequent network computation when the respective unstable state is updated, e. g. it does not guarantee that the network reaches a stable state which is close to the underlying training pattern. A similar analysis [51, 80] concerning the reproduction of most or all, complete patterns (unlike the reproduction of only particular neuron states in training patterns) shows that the maximum number of patterns that can be stored into the Hopfield autoassociative memory with $n$ neurons, is asymptotically proportional to $n/\log n$.

However, the autoassociative memory should possess more than the reproduction property which ensures the perfect storage of all training patterns, but it should also recall patterns only by using their partial (imprecise) knowledge. A more detailed analysis [5, 6, 20, 60] shows for the number $p \leq 0.138n$ of training patterns (i. e. for the memory capacity of at most 0.138) that these patterns correspond to local minima of the energy function in the Hopfield network that has been adapted according to Hebb law (3.26) and hence, this network can, in principle, be exploited as an autoassociative memory in this case. On the contrary, for $p > 0.138n$, the local minima corresponding to training patterns disappear. Furthermore, for $p < 0.05n$ (i. e. for the memory capacity less than 0.05), the training patterns correspond to global minima of the energy function that are deeper than the local minima related to spurious states. This means in a first-quality Hopfield autoassociative memory, 200 neurons are needed to store 10 training patterns which includes 40000 connections labeled with (integer) synaptic weights in the complete topology of cyclic network. Although it appears in practice that the above-mentioned theoretical estimates are rather overvalued, yet the basic model of the Hopfield autoassociative memory is only of theoretical significance thanks to its low capacity. There are many modifications of the Hopfield model described in the literature that make an effort to avoid this drawback.

### 3.2.4 An Application Example of Hopfield Network

The above-introduced general principles will be illustrated through an example of the Hopfield network exploitation for the noise reduction in recognition of characters, namely digits. The image of a digit is projected on the array of $12 \times 10$ black-and-white pixels that correspond to neurons of a Hopfield network so that their states $1, -1$ represent black and white colors, respectively (compare with the motivational example of scholar's character recognition in Paragraph 1.3.1). Furthermore, sample images of eight digits were created and used for the adaptation of Hopfield network by Hebbian learning (3.26). Then an input that arose from the sample image of the digit three by introducing a 25% noise was presented to the learned network. In Figure 3.5 the

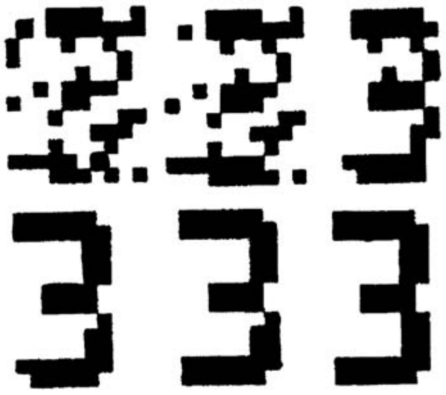

Figure 3.5: An application example of Hopfield network.

course of the computational mode for this input can be traced which is monitored at the macroscopic time steps. The Hopfield network, step by step, removes a noise from the digit image, i. e. it functions as an autoassociative memory when it reconstructs the original image of the digit three corresponding to a network stable state.

## 3.3 Continuous Hopfield Network

In this section, a continuous variant of an analog Hopfield network [13, 32] and its application to heuristic solving the traveling salesman problem will be described. In contrast with the analog models of neural networks that have been considered so far (e. g. the multilayered neural network with the backpropagation learning algorithm in Section 2.2) where the real neuron state is a continuous function of the excitation level, this network is an example of the model in which, in addition, the real state evolution in the computational mode is a continuous function of time. In such cases, the computational (or even adaptive) dynamics is usually given by differential equations whose solution cannot be expressed in an explicit form. Therefore, these models are not suitable for simulations on conventional computers unless their discrete version is used (in our case, this is the discrete-time Hopfield network from Section 3.2). On the other hand, they are advantageous to analog hardware implementations by electric circuits.

### 3.3.1 Continuous Computational Dynamics

Suppose that a Hopfield network model has the same **architectural** dynamics and, as the case may be, the same **adaptive** dynamics as it has been described in Paragraph 3.2.1. In addition, assume that the symmetric network weights may be reals and $w_{j0}$ $(1 \leq j \leq n)$ represents a generally non-zero, real bias (a threshold with the opposite sign) of the $j$th neuron corresponding to a formal unit input $y_0 = 1$. Remember that $w_{jj} = 0$ $(j = 1, \ldots, n)$. Then, the following **computational** dynamics is considered for the continuous Hopfield network. At the beginning of the computational mode, at time 0, the neuron states are assigned to the network input $\mathbf{x} = (x_1, \ldots, x_n)$, i. e. $y_i^{(0)} = x_i$ $(i = 1, \ldots, n)$. During the computational phase, the evolution of real network state $\mathbf{y}(t)$ is a continuous function of time $t > 0$ that is given by the following system of differential equations (for the lucidity reasons the time parameter $t$ is omitted in the following formulas):

$$\tau_j \frac{dy_j}{dt} = -y_j + \sigma(\xi_j) = -y_j + \sigma\left(\sum_{i=0}^{n} w_{ji} y_i\right) \quad j = 1, \ldots, n \tag{3.33}$$

where $\tau_j > 0$ $(j = 1, \ldots, n)$ are suitable time constants, $\xi_j(t)$ is the real excitation level of neuron $j$ which also depends on time $t$, and $\sigma$ is a continuous activation function, e. g. the standard sigmoid (1.9) with the range $(0, 1)$:

$$\sigma(\xi) = \frac{1}{1 + e^{-\lambda\xi}} \tag{3.34}$$

or its bipolar form — the hyperbolic tangent (1.10) with the range $(-1, 1)$:

$$\sigma(\xi) = \mathrm{tgh}\left(\frac{1}{2}\lambda\xi\right) = \frac{1 - e^{-\lambda\xi}}{1 + e^{-\lambda\xi}} \tag{3.35}$$

where $\lambda > 0$ is the gain parameter (compare with (2.7)). For $\lambda \to \infty$, the discrete neuron outputs $-1$ or $1$ are obtained similarly as in the discrete Hopfield network (3.28).

The computation of the continuous Hopfield network terminates at time $t^\star$ when the network finds itself in a stable state $\mathbf{y}(t^\star)$ whose change in time is zero, i. e. $\frac{dy_j}{dt}(t^\star) = 0$ for $j = 1, \ldots, n$. After substituting for $\frac{dy_j}{dt}(t^\star) = 0$ into the system (3.33) we get:

$$y_j = \sigma(\xi_j) = \sigma\left(\sum_{i=0}^{n} w_{ji} y_i\right) \quad j = 1, \ldots, n\,. \tag{3.36}$$

The equations (3.36) remind the stable state of the discrete version of the Hopfield network in the computational mode under (3.27), (3.28).

Moreover, the computational dynamics (3.33) of the continuous Hopfield network can also be described by a similar system of differential equations for the excitation level evolutions $\xi_j(t)$ $(j = 1, \ldots, n)$ in time:

$$\tau_j \frac{d\xi_j}{dt} = -\xi_j + \sum_{i=0}^{n} w_{ji} y_i = -\xi_j + \sum_{i=0}^{n} w_{ji} \sigma(\xi_i) \quad j = 1, \ldots, n\,. \tag{3.37}$$

Under certain conditions [62], the system (3.37) has the same equilibrium solutions (3.36) as the system (3.33).

Analogically to the discrete version (3.29), the energy of the continuous Hopfield network is defined which depends on the network state and therefore, its value evolves continuously in time:

$$E(t) = -\frac{1}{2}\sum_{j=1}^{n}\sum_{i=1}^{n} w_{ji} y_j y_i - \sum_{j=1}^{n} w_{j0} y_j + \sum_{j=1}^{n} \int_0^{y_j} \sigma^{-1}(y) dy \, . \tag{3.38}$$

For the activation function (3.34) (similarly for (3.35)), the integrals in formula (3.38) can explicitly be written as follows:

$$\int_0^{y_j} \sigma^{-1}(y) dy = \frac{1}{\lambda} \ln y_j^{y_j} \left(1 - y_j\right)^{1-y_j} \qquad j = 1, \dots, n \, . \tag{3.39}$$

In what follows it will be shown that the energy function $E(t)$ is non-increasing in the computational mode (3.33) (or (3.37)), i. e. $\frac{dE}{dt} \leq 0$:

$$\begin{aligned} \frac{dE}{dt} \quad &= \quad -\frac{1}{2}\sum_{j=1}^{n}\sum_{i=1}^{n} w_{ji} \frac{dy_j}{dt} y_i - \frac{1}{2}\sum_{j=1}^{n}\sum_{i=1}^{n} w_{ji} y_j \frac{dy_i}{dt} - \\ &\quad -\sum_{j=1}^{n} w_{j0} \frac{dy_j}{dt} + \sum_{j=1}^{n} \sigma^{-1}(y_j) \frac{dy_j}{dt} \, . \end{aligned} \tag{3.40}$$

From the fact that the weights are symmetric, i. e. $w_{ji} = w_{ij}$ $(1 \leq i,j \leq n)$ and by (3.36), (3.37) we obtain:

$$\frac{dE}{dt} = -\sum_{j=1}^{n} \frac{dy_j}{dt} \left( \sum_{i=0}^{n} w_{ji} y_i - \xi_j \right) = -\sum_{j=1}^{n} \tau_j \frac{dy_j}{dt} \frac{d\xi_j}{dt} \, . \tag{3.41}$$

The derivative $\frac{dE}{dt}$ can be expressed by (3.36) in such a form that we see it is non-positive:

$$\frac{dE}{dt} = -\sum_{j=1}^{n} \tau_j \sigma'(\xi_j) \left( \frac{d\xi_j}{dt} \right)^2 \leq 0 \tag{3.42}$$

since $\tau_j > 0$ $(j = 1, \dots, n)$ and the activation function $\sigma(\xi)$ is increasing for $\lambda > 0$, i. e. $\sigma'(\xi_j) > 0$. Thus, the energy $E(t)$ decreases during the computational phase (3.33), i. e. $\frac{dE}{dt} < 0$, until the network reaches its local minimum where $\frac{dE}{dt} = 0$, because the energy function (3.38) is bounded (the real states are within $(0, 1)$, or $(-1, 1)$, and the respective integrals (3.39) are bounded). According to (3.42), this corresponds to a stable network state $\frac{d\xi_j}{dt} = \frac{dy_j}{dt} = 0$. Thus, the continuous Hopfield network is proved to always terminate its computation in a stable state.

On the analogy of discrete version, the continuous Hopfield network that has been adapted under Hebb law, can be used as an autoassociative memory. In the following exposition, we will deal with yet another, non-standard exploitation of this model.

### 3.3.2 Traveling Salesman Problem

We have shown that the continuous Hopfield network minimizes the energy function $E$ in the state space during the computational phase. This can be exploited for heuristic solving of such an optimization problem whose objective function together with the underlying constraints can be expressed as a 'quadratic form' (3.38). By comparing the respective objective function with the energy function, the synaptic weights of a continuous Hopfield network are extracted ('adaptive' mode). Then, the optimal and feasible solution of a given problem is searched in the computational mode. This approach will be illustrated through the well-known traveling salesman problem [34].

Let $N > 2$ be a number of cities and for each pair $1 \leq r, s \leq N$, let a function $d : \{1, \ldots, N\}^2 \longrightarrow \mathbb{R}$ determine the distance $d(r, s)$ from the city $r$ to $s$. In addition, assume that $d(r, r) = 0$ and $d(r, s) = d(s, r)$ for every $1 \leq r, s \leq N$. Then, the traveling salesman problem consists in finding such a city ordering that forms the shortest circular tour in which each city is visited exactly once except the salesman ceases his tour at the same city from which he has started. This problem is represented by a continuous Hopfield network with $n = N \times N$ neurons whose states $y_{ru}$ are indexed by the pairs: a city number $r$ ($1 \leq r \leq N$) and its potential position $u$ ($1 \leq u \leq N$) in the city ordering on the traveling salesman tour. The matrix neuron arrangement in the network topology is outlined in Figure 3.6 where particular rows correspond to the

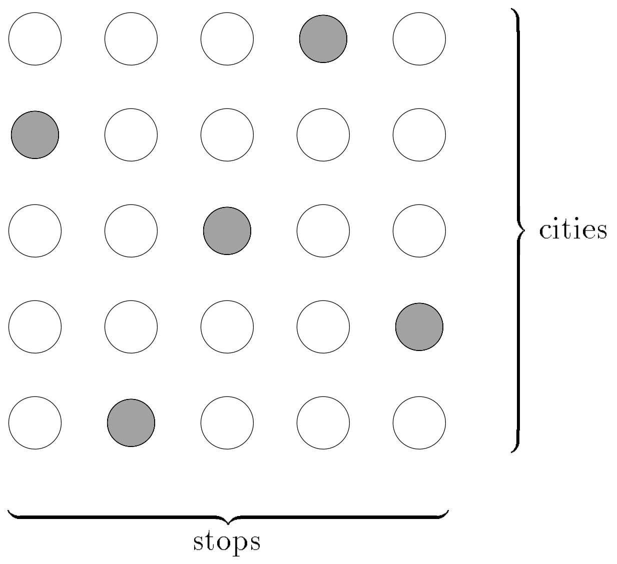

Figure 3.6: The neural representation of traveling salesman problem.

cities while columns determine the ordering of traveling salesman stops at these cities. For example, the standard sigmoid (3.34) is taken here as the activation function, i. e. the neuron states $y_{ru} \in (0, 1)$ ($1 \leq r, u \leq N$). When minimizing the energy function $E$ in the computational mode, the integral term in formula (3.38) compels the neurons to saturate in binary states, i. e. either $y_{ru} \doteq 1$ or $y_{ru} \doteq 0$, since the respective integrals (3.39) are zero for these values. Therefore, we can further confine ourselves only to the remaining terms in the energy definition (3.38) while assuming the binary states. The state $y_{ru} \doteq 1$ (the respective neuron is said to be active) is interpreted in such a way that the $r$th city finds in the $u$th position on the circular tour, and similarly, $y_{ru} \doteq 0$ (the neuron is passive) if this is not the case.

Obviously, in the above-introduced neural representation of the traveling salesman problem, not all network states correspond to the actual circular tours. Therefore, it is necessary to take the feasibility of the solution into account within the objective function being minimized. First, the traveling salesman is required to visit each city at most once, i. e. in each row at most one neuron should be active as it is depicted in Figure 3.6. This corresponds to the minimization of the following expression:

$$\begin{aligned} E_A &= \frac{A}{2}\sum_{r=1}^{N}\sum_{u=1}^{N}\sum_{\substack{v=1\\ v\neq u}}^{N} y_{ru}y_{rv} = \\ &= -\frac{1}{2}\sum_{\substack{j=1\\ j=(r,u)}}^{n}\sum_{\substack{i=1\\ i=(s,v)}}^{n} -A\delta_{rs}(1-\delta_{uv})y_j y_i \end{aligned} \tag{3.43}$$

where a parameter $A > 0$ measures the influence of $E_A$ during the minimization process, and $\delta_{rs}$ is the Kronecker delta, i. e. $\delta_{rs} = 1$ if $r = s$ and $\delta_{rs} = 0$ if $r \neq s$. The purpose of the formal manipulation in (3.43) is to transform $E_A$ to a form that is comparable with the first term of the energy function (3.38). Similarly, the traveling salesman is able to visit at most one city at each stop on the circular tour, i. e. in each column at most one neuron should be active as it is depicted in Figure 3.6. This corresponds to minimizing the following expression:

$$\begin{aligned} E_B &= \frac{B}{2}\sum_{u=1}^{N}\sum_{r=1}^{N}\sum_{\substack{s=1\\ s\neq r}}^{N} y_{ru}y_{su} = \\ &= -\frac{1}{2}\sum_{\substack{j=1\\ j=(r,u)}}^{n}\sum_{\substack{i=1\\ i=(s,v)}}^{n} -B\delta_{uv}(1-\delta_{rs})y_j y_i \end{aligned} \tag{3.44}$$

where a parameter $B > 0$ measures the influence of $E_B$ during the minimization process. Again, the aim of the formal manipulation in (3.44) is to transform $E_B$ to a form that is comparable with the first term in (3.38). Finally, the traveling salesman is required to visit exactly $N$ cities on his tour, i. e. exactly $N$ neurons should be active in the network (see Figure 3.6). This corresponds to the minimization of the following expression:

$$\begin{aligned} E_C &= \frac{C}{2}\left(N - \sum_{r=1}^{N}\sum_{u=1}^{N} y_{ru}\right)^2 = \frac{CN^2}{2} - \\ &-\frac{1}{2}\sum_{\substack{j=1\\ j=(r,u)}}^{n}\sum_{\substack{i=1\\ i=(s,v)}}^{n} -C\left(1-\delta_{ji}\right)y_j y_i - \sum_{j=1}^{n}\left(CN - \frac{C}{2}\right)y_j\,, \end{aligned} \tag{3.45}$$

where a parameter $C > 0$ measures the influence of $E_C$ during the minimization process. The second line in (3.45) is of the form that is comparable to the first two terms in (3.38) while the term $\frac{CN^2}{2}$ is constant and does not influence the minimization process and therefore, it will further be omitted in $E_C$. Thus, the simultaneous minimization of (3.43), (3.44), (3.45) in the state space compels the network states to represent feasible solutions of the traveling salesman problem.

Furthermore, the condition that the traveling salesman tour is as short as possible, i. e. that the respective feasible solution is optimal as well, is taken into account in the objective function. This corresponds to minimizing the following expression:

$$\begin{aligned} E_D &= \frac{D}{2}\sum_{r=1}^{N}\sum_{s=1}^{N}\sum_{u=1}^{N} d(r,s)y_{ru}\left(y_{s,u-1}+y_{s,u+1}\right) = \qquad (3.46)\\ &= -\frac{1}{2}\sum_{\substack{j=1\\ j=(r,u)}}^{n}\;\sum_{\substack{i=1\\ i=(s,v)}}^{n} -Dd(r,s)\left(\delta_{u,v-1}+\delta_{u,v+1}\right)y_j y_i \end{aligned}$$

where a parameter $D > 0$ measures the influence of $E_D$ during the minimization process. Again, the purpose of the formal manipulation in (3.46) is to transform $E_D$ to a form that is comparable with the first term in (3.38). For the notational simplicity the indices $u-1$ for $u=1$ and $u+1$ for $u=N$ in (3.46) are interpreted as $u=N$ and $u=1$, respectively.

Thus, the resulting objective function $E_{TS}$ is the sum of (3.43), (3.44), (3.45), and (3.46) whose minimum gives an optimal feasible solution of the respective traveling salesman problem, i. e. the city ordering of the shortest circular tour:

$$\begin{aligned} E_{TS} &= E_A + E_B + E_C + E_D = \\ = -\frac{1}{2}\sum_{\substack{j=1\\ j=(r,u)}}^{n}\;\sum_{\substack{i=1\\ i=(s,v)}}^{n} &\Big(-A\delta_{rs}(1-\delta_{uv}) - B\delta_{uv}(1-\delta_{rs}) - \\ &-C(1-\delta_{ji}) - Dd(r,s)(\delta_{u,v-1}+\delta_{u,v+1})\Big)y_j y_i \\ &-\sum_{j=1}^{n}\left(CN-\frac{C}{2}\right)y_j\,. \qquad (3.47) \end{aligned}$$

By comparing the traveling salesman objective function (3.47) with the first two terms of energy function (3.38), the weights of a continuous Hopfield network are extracted that ensure the $E_{TS}$ minimization during the computational phase:

$$\begin{aligned} w_{ji} &= -A\delta_{rs}(1-\delta_{uv}) - B\delta_{uv}(1-\delta_{rs}) - C(1-\delta_{ji}) \\ &\quad -Dd(r,s)(\delta_{u,v-1}+\delta_{u,v+1}) \qquad (3.48)\\ w_{j0} &= CN - \frac{C}{2} \qquad j=(r,u),\;\; i=(s,v),\;\; 1\le r,s,u,v\le N\,. \end{aligned}$$

It follows from the weight definition (3.48) that $w_{ji} = w_{ij}$ and $w_{jj} = 0$ for $1 \le i,j \le n$.

For heuristic solving of the traveling salesman problem, the weights of the continuous Hopfield network are first assigned according to formula (3.48). Here, the suitable real parameters $A$, $B$, $C$, $D$ are chosen to determine the degree of minimization with respect to the particular terms in $E_{TS} = E_A + E_B + E_C + E_D$ (e. g. $A = B = D = 500$, $C = 200$ [34]). The computational phase starts at a random initial network state nearby the zero vector, e. g. $y_j^{(0)} \in (0, 0.001)$ $(j = 1, \ldots, n)$. The proper computation of the continuous Hopfield network is being performed according to (3.33) until the network reaches a stable state corresponding to a local minimum of energy function (3.38) which coincides with the objective function $E_{TS}$ of the traveling salesman problem. Finally, the respective result is read from the network, i. e. $y_{ru} \doteq 1$ is interpreted as the traveling salesman visits the $r$th city at the $u$th stop on his circular tour.

However, since the achieved minimum of the energy function may not be a global one, the resulting solution of the traveling salesman problem obtained in the computational mode of the continuous Hopfield network, may generally not be optimal and it may not even be feasible. By a suitable choice of parameters $A$, $B$, $C$, $D$ and by repeating the computation with different initial network states, a better approximation of the optimum can be accomplished. Also an appropriate selection strategy of the gain parameter $\lambda > 0$ within the activation function (3.34) may help here. At the beginning, $\lambda$ is chosen sufficiently small in order to provide the particular neuron states with a greater degree of freedom. Then, during the computational phase when the network finds itself in the state space close to a deeper minimum of the energy function, this parameter may be increased to accelerate the convergence. However, there are no general directions for an effective selection of the above-mentioned parameters that would ensure the convergence to a global minimum of the energy function in order to achieve the actual optimum of the traveling salesman problem since this problem is $NP$-complete. The above-introduced procedure is only the heuristics which, in addition, cannot compete with the well-known, classical approximation algorithms for the traveling salesman problem. Rather than that, it illustrates a possible non-standard application of neural networks.

## 3.4 Boltzmann Machine

A *Boltzmann machine* is the neural network model introduced by Hinton and Sejnowski [1, 29, 30], which is a stochastic variant of the Hopfield network with hidden neurons. The name of this model originated in statistical mechanics because the probability of its states in a so-called thermal equilibrium is controlled by the Boltzmann distribution. For example, the Boltzmann machine can be used as a heteroassociative memory (see Section 3.1).

### 3.4.1 Stochastic Computational Dynamics

At the beginning, the **architectural** dynamics of the Boltzmann machine specifies a fixed topology of the cyclic neural network with symmetric connections, i. e. the architecture is described by an undirected graph. The set $V = A \cup B$ of $s$ neurons in this network is disjointly split into a set $B$ of $b$ hidden neurons and a set $A = X \cup Y$ of $a = n + m$, so-called *visible* neurons which are further categorized into a set $X$ of $n$ input neurons and a set $Y$ of $m$ output neurons. An example of the Boltzmann machine

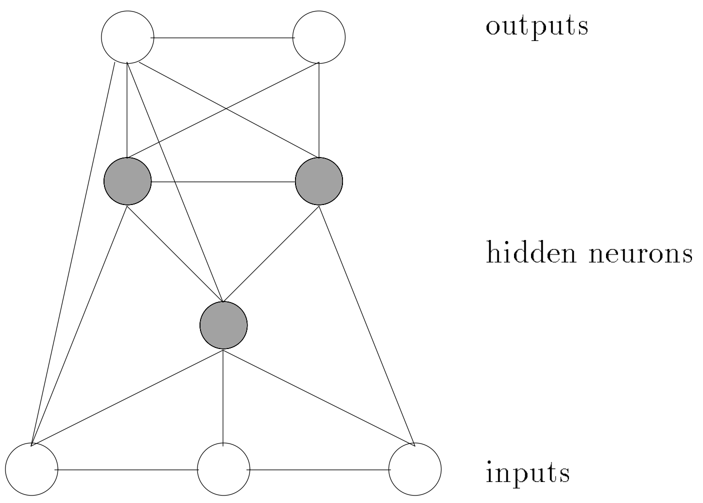

Figure 3.7: An example of Boltzmann machine topology.

architecture is depicted in Figure 3.7. The neurons are denoted by indices $i$, $j$ etc., $\xi_j$ represents a real excitation level and $y_j \in \{-1, 1\}$ is a bipolar state (output) of neuron $j$. The state of visible neurons is denoted by a vector $\boldsymbol{\alpha} \in \{-1,1\}^a$ and similarly, $\boldsymbol{\beta} \in \{-1,1\}^b$ is the state of hidden neurons. Then $\mathbf{y} = (\boldsymbol{\alpha}, \boldsymbol{\beta}) \in \{-1,1\}^s$ denotes the state of the Boltzmann machine. The connection between $i$ and $j$ is labeled with a real symmetric weight $w_{ji} = w_{ij}$. For simplistic reasons, we again assume the thresholds of all neurons to be zero and $w_{jj} = 0$ for every $j \in V$. We also use a notation $\overleftarrow{j}$ for a set of all neurons that are connected with neuron $j$.

The **computational** dynamics of Boltzmann machine will be described for the case of sequential computations. At the beginning of the computational mode, at time 0, the states of input neurons $i \in X$ are assigned to a bipolar network input $\mathbf{x} = (x_1, \ldots, x_n) \in \{-1,1\}^n$ and these are further clamped (fixed) during the whole computation, i. e. $y_i^{(t)} = x_i$ for $i \in X$, $t \geq 0$. The states $y_j^{(0)} \in \{-1,1\}$ of the remaining non-input neurons $j \in V \setminus X$ are initialized randomly. Then, at the discrete computational time $t > 0$, one non-input neuron $j \in V \setminus X$ is randomly selected whose state is updated while the remaining neurons do not change their states. Similarly as in the Hopfield network (see Paragraph 3.2.1), the macroscopic time $\tau$ can again be considered here, i. e. $\tau s' < t \leq (\tau + 1)s'$ where $s' = s - n$ is the number of updated (non-input) neurons during one time period. At first, a real excitation level of neuron $j$ is computed:

$$\xi_j^{(t-1)} = \sum_{i \in \overleftarrow{j}} w_{ji} y_i^{(t-1)} . \tag{3.49}$$

Then, the output of neuron $j$ at time $t$ is stochastically determined so that the neuron $j$ is active with probability

$$\mathbf{P}\left\{y_j^{(t)} = 1\right\} = \sigma(\xi_j^{(t-1)}) \tag{3.50}$$

and thus, it is passive with probability

$$\mathbf{P}\left\{y_j^{(t)} = -1\right\} = 1 - \mathbf{P}\left\{y_j^{(t)} = 1\right\} = \sigma(-\xi_j^{(t-1)}) \tag{3.51}$$

where $\sigma$ is a *stochastic activation function*:

$$\sigma(\xi) = \frac{1}{1 + e^{-2\xi/T^{(\tau)}}} \tag{3.52}$$

whose shape corresponds to the standard sigmoid (1.9).

### 3.4.2 Simulated Annealing

The parameter $T^{(\tau)} > 0$ in formula (3.52) which may evolve in macroscopic time $\tau$, is called *temperature* due to the physical analogy. It is inversely proportional to the gain parameter of the stochastic activation function (compare with (3.34)). Clearly, for the temperature $T \to \infty$, the probabilities (3.50), (3.51) of both bipolar states are coincidently 1/2 and the Boltzmann machine behaves completely randomly during the computational mode. On the contrary, for $T \to 0$ the computation of Boltzmann machine under (3.49), (3.50) is deterministic and coincides with the computational phase of the Hopfield network described by (3.27), (3.28). Usually, a sufficiently high temperature $T^{(0)}$ is chosen at the beginning so that the probability (3.50) is a little bit greater than 1/2 and the computation has a great degree of freedom. Then, because of the convergence, the whole system is gradually 'cooled', i. e. the temperature is slowly being decreased. This procedure is called *simulated annealing* [39] thanks to the physical analogy. For example, the temperature $T^{(\tau)} = \eta^c T^{(0)}$ for $cq \leq \tau < (c+1)q$ $(c = 0, 1, 2, \ldots)$ is decreased after each $q > 0$ macroscopic steps where $0 < \eta < 1$ is selected to be close to 1 and inversely proportional to a choice of $q$ (e. g. $\eta = 0.98$ for a less $q$ and $\eta = 0.9$ for a greater $q$). Or the temperature evolution is controlled by the following heuristics [19]:

$$T^{(\tau)} = \frac{T^{(0)}}{\log(1+\tau)} \qquad \tau > 0\,. \tag{3.53}$$

The graph of the stochastic activation function for different temperature examples is drawn in Figure 3.8.

The above-introduced computational dynamics of Boltzmann machine further develops the physical analogy of the Hopfield network from Paragraph 3.2.2 so that, in addition, the temperature is taken into account. Namely, the spins in magnetic materials are affected by random, so-called *thermal fluctuations* that tend to frequently and randomly flip the magnetic orientation of spins. Thus, these fluctuations restrict the influence of the surrounding magnetic field while the ratio of both effects depends on the temperature. The thermal fluctuations are being slipped off with decreasing temperature until the absolute zero 0K (i. e. $-273°$C) is reached when they disappear. On the contrary, the thermal fluctuations dominate under high temperatures and the magnetic orientation of a spin is independent on the magnetic field, i. e. it is nearly as often opposite to its field as aligned with it. This phenomenon can formally be described by the so-called *Glauber dynamics* [21] which corresponds to the computational

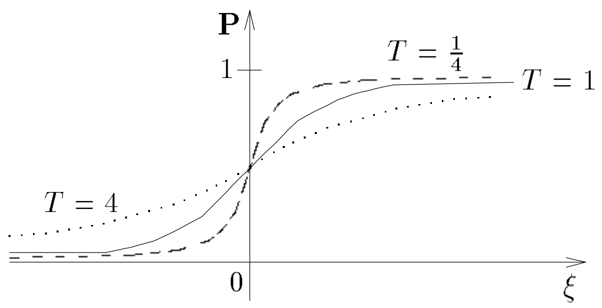

Figure 3.8: The graph of the stochastic activation function for different temperatures $T$.

dynamics of Boltzmann machine (3.49), (3.50) except the actual absolute temperature $T_K$ in Kelvin is still adjusted: $T = k_B T_K$ where $k_B = 1.38 \times 10^{-23} J/K$ is *Boltzmann's constant*. The temperature decrease in the computational mode evokes the actual material annealing, e. g. in the production of a hard metal when a gradual cooling is applied in order to avoid anomalies in the material crystalline structure.

The computational dynamics of Boltzmann machine may be interpreted neurophysiologically as well. Besides proper electric signals from dendrites (see Section 1.2), the biological neuron is influenced by other circumstances. For example, the impulse generated in an axon has a different intensity at each time, they are delays in synapses, random fluctuations, etc. These interfering effects can be thought of as noise that can be represented by thermal fluctuations. In this case, the parameter $T$ does not correspond to the actual temperature but it determines the noise level.

### 3.4.3 Equilibrium State

Similarly as in the Hopfield network (see Paragraph 3.2.2) the energy function $E(\mathbf{y})$ of Boltzmann machine is defined which depends on its state $\mathbf{y}$:

$$E(\mathbf{y}) = -\frac{1}{2} \sum_{j \in V} \sum_{i \in \overleftarrow{j}} w_{ji} y_j y_i \,. \tag{3.54}$$

It can be proved that on average, the energy (3.54) is decreasing together with the temperature parameter $T$ (simulated annealing) during the computational phase until, at time $t^\star$, the Boltzmann machine reaches a so-called *thermal equilibrium* with the temperature $T^\star$. In this equilibrium, the state of Boltzmann machine is not constant due to the stochastic computation, but it fluctuates locally around the constant average 'stable' state $\overline{\mathbf{y}^\star} \in [-1,1]^s$ corresponding to a local minimum of $E$. A fundamental result from statistical mechanics says that in thermal equilibrium, at the temperature $T^\star$, each of the possible state $\mathbf{y}^\star \in \{-1,1\}^s$ of Boltzmann machine occurs with probability (Boltzmann distribution):

$$p_B(\mathbf{y}^\star) = \frac{e^{-E(\mathbf{y}^\star)/T^\star}}{\sum\limits_{\mathbf{y} \in \{-1,1\}^s} e^{-E(\mathbf{y})/T^\star}} \,. \tag{3.55}$$

By formula (3.55), the stable *equilibrium state* $\overline{\mathbf{y}^\star} \in [-1,1]^s$ which is an average over all states in thermal equilibrium, weighted by the respective probabilities, can be computed as follows:

$$\overline{\mathbf{y}_j^\star} = \sum_{\mathbf{y}\in\{-1,1\}^s} p_B(\mathbf{y})y_j \quad j \in V\,. \tag{3.56}$$

Then, the average output neuron states $\overline{y_j^\star} \in [-1,1]$ $(j \in Y)$ within the equilibrium state can determine the *average output* for a given input.

Besides the heteroassociative memory, the Boltzmann machine can be exploited for heuristic solving the optimization tasks providing that an optimization problem with the respective objective function (optimality) and constraints (feasibility) can be reduced to the minimization of a quadratic form (3.54) (compare with the continuous Hopfield network application to the traveling salesman problem in Paragraph 3.3.2). In this case, the weights of Boltzmann machine are obtained by comparing the respective quadratic form with the network energy function $E$. A feasible and optimal solution of the given problem is searched during the computational phase by minimizing the energy function using a 'stochastic gradient method' (3.49), (3.50) (compare with (3.30)). The main problem in non-linear optimization that has already been discussed in the context of the backpropagation learning algorithm (see Paragraph 2.2.2) is how to avoid the local minima in order to reach a global minimum. The advantage of the Boltzmann machine over the (continuous) Hopfield network consists in the possibility, under a higher temperature, to move from a local minimum of $E$ to a state with a higher energy (thanks to the stochastic computation) from which, hopefully, a path in the state space to a deeper minimum leads. By using an appropriate strategy in the simulated annealing, a better optimum than that by the deterministic Hopfield network, can be found. Also in the case of associative memory when the global minima of the energy function represent the training patterns while the local minima correspond to spurious patterns (see Paragraph 3.2.2) these admissible states can be avoided under the stochastic computational mode.

### 3.4.4 Boltzmann Learning

Now, we will describe the **adaptive** dynamics of the Boltzmann machine for the case of heteroassociative memory. In the adaptive mode, the desired network function is prescribed by a discrete probability distribution of visible neuron states so that a desired non-zero probability $p_d(\mathbf{x},\mathbf{d}) > 0$ is assigned to each possible state $\boldsymbol{\alpha} = (\mathbf{x},\mathbf{d}) \in \{-1,1\}^{n+m}$ of input and output (visible) neurons. However, in practice such a distribution is usually explicitly unknown and also its description would require exponentially many ($2^a$, where a=n+m) non-zero probability values. Therefore instead, a training set is considered:

$$\mathcal{T} = \left\{ (\mathbf{x}_k,\mathbf{d}_k) \;\middle|\; \begin{array}{l} \mathbf{x}_k = (x_{k1},\ldots,x_{kn}) \in \{-1,1\}^n \\ \mathbf{d}_k = (d_{k1},\ldots,d_{km}) \in \{-1,1\}^m \end{array} \quad k = 1,\ldots,p \right\} \tag{3.57}$$

that contains only the relevant inputs $\mathbf{x}_k$ $(k = 1,\ldots,p)$ paired with the desired outputs $\mathbf{d}_k$ so that the number $0 < p \ll 2^a$ of patterns is typically much less than the number of all possible states of visible neurons. Furthermore, a uniform distribution of training patterns is usually assumed, i. e. $p_d(\mathbf{x}_k, \mathbf{d}_k) = \frac{1}{p} > 0$ $(k = 1, \ldots, p)$. In addition, a random noise with a small probability is carried into the training set $\mathcal{T}$ which ensures that the remaining visible neuron states which are not included in the training set, occur with non-zero probabilities.

The aim of the Boltzmann machine adaptation is to find such a network configuration $\mathbf{w}$ that the probability distribution $p_A(\boldsymbol{\alpha})$ of states $\boldsymbol{\alpha} \in \{-1,1\}^a$ of visible neurons from $A = X \cup Y$ in thermal equilibrium coincides with the desired distribution $p_d(\boldsymbol{\alpha})$ of states $\boldsymbol{\alpha} = (\mathbf{x}, \mathbf{d})$ in the training patterns (3.57). By means of (3.55) the probability $p_A(\alpha)$ of a visible neuron state $\boldsymbol{\alpha} \in \{-1,1\}^a$ can be expressed independently on the hidden neuron states $\boldsymbol{\beta} \in \{-1,1\}^b$ as follows:

$$p_A(\boldsymbol{\alpha}) = \sum_{\boldsymbol{\beta} \in \{-1,1\}^b} p_B(\boldsymbol{\alpha}, \boldsymbol{\beta}) . \tag{3.58}$$

A relative entropy weighted by the probabilities of training pattern occurrences measures adequately the discrepancy between the probability distributions $p_A$ and $p_d$, i. e. it represents an error $\mathcal{E}$:

$$\mathcal{E}(\mathbf{w}) = \sum_{\boldsymbol{\alpha} \in \{-1,1\}^a} p_d(\boldsymbol{\alpha}) \log \frac{p_d(\boldsymbol{\alpha})}{p_A(\boldsymbol{\alpha})} . \tag{3.59}$$

It can be shown that the error $\mathcal{E}(\mathbf{w})$ is always nonnegative and it equals zero if and only if $p_A = p_d$. In addition, $\mathcal{E}(\mathbf{w})$ is a function of the Boltzmann machine configuration $\mathbf{w}$ since $p_A$ depends on synaptic weights according to (3.58), (3.55), and (3.54). Thus, the error function $\mathcal{E}(\mathbf{w})$ can be minimized in the weight space by using a gradient method.

At the beginning of the Boltzmann machine adaptive mode, at time 0, the weights in the configuration $\mathbf{w}^{(0)}$ are uniformly distributed around zero, e. g. $w_{ji}^{(0)} = w_{ij}^{(0)} \in \langle -1, 1 \rangle$ $(j \in V, i \in \bar{j})$. The underlying adaptation proceeds at the discrete time steps that correspond to training epochs in which each training pattern from $\mathcal{T}$ is presented to the network more times. The new configuration $\mathbf{w}^{(t)}$ is computed at time $t > 0$:

$$w_{ji}^{(t)} = w_{ji}^{(t-1)} + \Delta w_{ji}^{(t)} \quad j \in V, i \in \bar{j} \tag{3.60}$$

where the weight increment $\Delta \mathbf{w}^{(t)}$ at time $t > 0$ is proportional to the negative gradient of the error function (3.59) at the point $\mathbf{w}^{(t-1)}$:

$$\Delta w_{ji}^{(t)} = -\varepsilon \frac{\partial \mathcal{E}}{\partial w_{ji}} \left( \mathbf{w}^{(t-1)} \right) \quad j \in V, i \in \bar{j} \tag{3.61}$$

where $0 < \varepsilon < 1$ is the learning rate.

To implement the adaptive dynamics (3.60), the gradient of the error function in formula (3.61) must be computed. By formal differentiating the error function (3.59) while employing formulas (3.58), (3.55), (3.54) and by interpreting the respective probabilities we get the following expression (a detailed derivation can be found e. g. in [27]):

$$\frac{\partial \mathcal{E}}{\partial w_{ji}} = -\frac{1}{T^\star} \left( \overline{y_j^\star y_i^\star}(A) - \overline{y_j^\star y_i^\star} \right) \quad j \in V, i \in \bar{j} . \tag{3.62}$$

The term $\overline{y_j^\star y_i^\star}(A)$ in formula (3.62) is an average value of $y_j^\star y_i^\star$ in thermal equilibrium supposing that the states $\boldsymbol{\alpha} = (\mathbf{x}, \mathbf{d})$ of input and output (visible) neurons from $A = X \cup Y$ are clamped:

$$\overline{y_j^\star y_i^\star}(A) = \sum_{\boldsymbol{\alpha} \in \{-1,1\}^a} p_d(\boldsymbol{\alpha}) \sum_{\boldsymbol{\beta} \in \{-1,1\}^b} \frac{p_B(\boldsymbol{\alpha}, \boldsymbol{\beta})}{p_A(\boldsymbol{\alpha})} y_j^\star(\boldsymbol{\alpha}) y_i^\star(\boldsymbol{\alpha}) \tag{3.63}$$

where $y_j^\star(\boldsymbol{\alpha})$ is an output of neuron $j \in V$ in the thermal equilibrium of the Boltzmann machine provided that the states of visible neurons are clamped with a vector $\boldsymbol{\alpha}$. Similarly, the term $\overline{y_j^\star y_i^\star}$ in (3.62) is an average value of $y_j^\star y_i^\star$ in the thermal equilibrium of the Boltzmann machine without clamping the visible neurons:

$$\overline{y_j^\star y_i^\star} = \sum_{\mathbf{y} \in \{-1,1\}^s} p_B(\mathbf{y}) y_j^\star y_i^\star . \tag{3.64}$$

After substituting the partial derivative for (3.62) into (3.61) the *Boltzmann learning rule* is obtained:

$$\Delta w_{ji} = \frac{\varepsilon}{T^\star} \left( \overline{y_j^\star y_i^\star}(A) - \overline{y_j^\star y_i^\star} \right) \qquad j \in V, i \in \overleftarrow{j} \, . \tag{3.65}$$

The first term $\overline{y_j^\star y_i^\star}(A)$ (see (3.63)) in formula (3.65) can be interpreted as Hebbian learning (compare with (3.5)) with clamping the input and output (visible) neurons while the second term $-\overline{y_j^\star y_i^\star}$ (see (3.64)) corresponds to Hebbian unlearning (compare with (3.31)) with free visible neurons. The adaptive mode converges if these terms match for all $j \in V, i \in \overleftarrow{j}$.

### 3.4.5 The Learning Algorithm

The above-introduced adaptive dynamics of the Boltzmann machine will be summarized into a transparent algorithm that can be realized distributively by means of a specialized hardware [76] or it can be simulated on a conventional computer by using the Monte Carlo method:

1. Initialize the discrete adaptation time $t := 0$.
2. Choose randomly $w_{ji}^{(0)} = w_{ij}^{(0)} \in \langle -1, 1 \rangle$ with the uniform probability distribution.
3. Increment $t := t + 1$.
4. Assign $\varrho_{ji}^{+(t-1)} := 0$ for each $j, i$.
5. For every training pattern $k = 1, \ldots, p$ perform the following actions $q$ times:
   (a) Fix the states $\boldsymbol{\alpha} = (\mathbf{x}'_k, \mathbf{d}'_k) \in \{-1, 1\}^{n+m}$ of the visible neurons so that the probability that one visible neuron state agrees with the corresponding desired state prescribed by the $k$th training pattern equals e. g. $\mathbf{P}\{x'_{ki} = x_{ki}\} = 0.9$ ($i = 1, \ldots, n$) and $\mathbf{P}\{d'_{kj} = d_{kj}\} = 0.9$ ($j = 1, \ldots, m$). Further follow the computational dynamics (3.49), (3.50) (only the hidden neurons

are being updated while the visible neurons are clamped) by using simulated annealing until the thermal equilibrium is reached at the state $\mathbf{y}^\star$ with a final temperature $T^\star$. The state $\mathbf{y}^\star$ will serve as an initial state of the subsequent Boltzmann machine computation starting at macroscopic computational time $\tau = 0$.

   (b) At macroscopic computational time $\tau = 1, \ldots, r$ do the following actions:

      i. Perform one macroscopic computational step of the Boltzmann machine under temperature $T^\star$, i. e. in a random order update the states of all hidden neurons according to (3.49), (3.50).

      ii. Update the statistics for $\overline{y_j^\star y_i^\star}(A)$ with respect to the current network state, i. e. $\varrho_{ji}^{+(t-1)} := \varrho_{ji}^{+(t-1)} + y_j^{(\tau)} y_i^{(\tau)}$.

6. Ascertain the average of $\varrho_{ji}^{+(t-1)} := \frac{\varrho_{ji}^{+(t-1)}}{pqr}$.
   { $\varrho_{ji}^{+(t-1)}$ is an estimate of the current $\overline{y_j^\star y_i^\star}(A)$. }

7. Assign $\varrho_{ji}^{-(t-1)} := 0$ for each $j, i$.

8. Perform the following actions $pq$ times:

   (a) Without clamping the visible neurons, i. e. for a random initial state of the Boltzmann machine follow the computational dynamics (3.49), (3.50) (all visible and hidden neurons are being updated) by using simulated annealing until the thermal equilibrium is reached at the state $\mathbf{y}^\star$ with a final temperature $T^\star$. The state $\mathbf{y}^\star$ will serve as an initial state of the subsequent Boltzmann machine computation starting at macroscopic computational time $\tau = 0$.

   (b) At macroscopic computational time $\tau = 1, \ldots, r$ do the following actions:

      i. Perform one macroscopic computational step of the Boltzmann machine under temperature $T^\star$, i. e. in a random order update the states of all hidden neurons according to (3.49), (3.50).

      ii. Update the statistics for $\overline{y_j^\star y_i^\star}$ with respect to the current network state, i. e. $\varrho_{ji}^{-(t-1)} := \varrho_{ji}^{-(t-1)} + y_j^{(\tau)} y_i^{(\tau)}$.

9. Ascertain the average of $\varrho_{ji}^{-(t-1)} := \frac{\varrho_{ji}^{-(t-1)}}{pqr}$.
   { $\varrho_{ji}^{-(t-1)}$ is an estimate of current $\overline{y_j^\star y_i^\star}$. }

10. According to (3.65) compute $\Delta w_{ji}^{(t)} := \frac{\varepsilon}{T^\star} \left( \varrho_{ji}^{+(t-1)} - \varrho_{ji}^{-(t-1)} \right)$.

11. According to (3.60) update the configuration $w_{ji}^{(t)} := w_{ji}^{(t-1)} + \Delta w_{ji}^{(t)}$.

12. If $\Delta w_{ji}^{(t)}$ is sufficiently small, then terminate otherwise continue with step 3.

It is clear from the above-introduced algorithm that the adaptive phase of the Boltzmann machine is very time-consuming. In spite of that, the results achieved by the Boltzmann machine are very good [27] in practical applications.

# Bibliography

[1] D. H. Ackley, G. E. Hinton, and T. J. Sejnowksi. A learning algorithm for Boltzmann machines. *Cognitive Science*, 9:147–169, 1985.

[2] L. B. Almeida. Backpropagation in perceptrons with feedback. In R. Eckmiller and Ch. von der Malsburg, editors, *Neural Computers*, pages 199–208, Neuss, 1987. Berlin: Springer-Verlag.

[3] L. B. Almeida. A learning rule for asynchronous perceptrons with feedback in a combinatorial environment. In M. Caudill and C. Butler, editors, *Proceedings of the IEEE First International Conference on Neural Networks*, volume II, pages 609–618, San Diego, 1987. New York: IEEE Press.

[4] S.-I. Amari. Neural theory of association and concept formation. *Biological Cybernetics*, 26:175–185, 1977.

[5] D. J. Amit, H. Gutfreund, and H. Sompolinsky. Storing infinite numbers of patterns in a spin-glass model of neural networks. *Physical Review Letters*, 55:1530–1533, 1985.

[6] D. J. Amit, H. Gutfreund, and H. Sompolinsky. Information storage in neural networks with low levels of activity. *Physical Review*, A 35:2293–2303, 1987.

[7] J. A. Anderson. A memory storage model utilizing spatial correlation functions. *Kybernetik*, 5:113–119, 1968.

[8] J. A. Anderson. Two models for memory organization using interacting traces. *Mathematical Biosciences*, 8:137–160, 1970.

[9] J. A. Anderson. A simple neural network generating an interactive memory. *Mathematical Biosciences*, 14:197–220, 1972.

[10] J. A. Anderson. A theory for the recognition of items from short memorized lists. *Psychological Review*, 80:417–438, 1973.

[11] A. E. Bryson and Y.-C. Ho. *Applied Optimal Control*. Blaisdell, New York, 1969.

[12] J. P. Cater. Successfully using peak learning rates of 10 (and greater) in back-propagation networks with the heuristic learning algorithm. In M. Caudill and C. Butler, editors, *Proceedings of the IEEE First International Conference on Neural Networks*, volume II, pages 645–651, San Diego, 1987. New York: IEEE Press.

[13] M. A. Cohen and S. Grossberg. Absolute stability of global pattern formation and parallel memory storage by competitive neural networks. *IEEE Transactions on Systems, Man, and Cybernetics*, 13:815–826, 1983.

[14] G. W. Cottrell, P. Munro, and D. Zipser. Image compression by back propagation: An example of extensional programming. Technical Report 8702, University of California at San Diego Institute for Cognitive Science, 1987.

[15] S. E. Fahlman and C. Lebiere. The cascade-correlation learning architecture. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems II*, pages 524–532, Denver, 1989. San Mateo: Morgan Kaufmann, 1990.

[16] L. V. Fausett. *Fundamentals of Neural Networks: Architectures, Algorithms, and Applications.* Prentice–Hall, New Jersey, 1994.

[17] M. A. Franzini. Speech recognition with back propagation. In *Proceedings of the Ninth Annual Conference of the IEEE Engineering in Medicine and Biology Society*, pages 1702–1703, Boston, 1987. New York: IEEE Press.

[18] M. Frean. The upstart algorithm: A method for constructing and training feedforward neural networks. *Neural Computation*, 2:198–209, 1990.

[19] S. Geman and D. Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6:721–741, 1984.

[20] T. Geszti. *Physical Models of Neural Networks.* World Scientific, Singapore, 1990.

[21] R. J. Glauber. Time-dependent statistics of the Ising model. *Journal of Mathematical Physics*, 4:294–307, 1963.

[22] T. Greville. Some applications of the pseudoinverse of a matrix. *SIAM Review*, 2:15–22, 1960.

[23] S. J. Hanson and L. Pratt. A comparison of different biases for minimal network construction with back-propagation. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems I*, pages 177–185, Denver, 1988. San Mateo: Morgan Kaufmann, 1989.

[24] S. Haykin. *Neural Networks.* Macmillan College Publishing Company, New York, 1994.

[25] D. O. Hebb. *The Organization of Behavior: A Neuropsychological Theory.* Wiley, New York, 1949.

[26] R. Hecht-Nielsen. *Neurocomputing.* Addison-Wesley, California, 1990.

[27] J. Hertz, A. Krogh, and R. G. Palmer. *Introduction to the Theory of Neural Computation*, volume I of *Lecture Notes, Santa Fe Institute Studies in the Sciences of Complexity.* Addison-Wesley, California, 1991.

[28] G. E. Hinton. Learning distributed representations of concepts. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, pages 1–12, Amherst, 1986. Hillsdale: Erlbaum.

[29] G. E. Hinton and T. J. Sejnowksi. Optimal perceptual inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 448–453, Washington, 1983. New York: IEEE Press.

[30] G. E. Hinton and T. J. Sejnowksi. Learning and relearning in Boltzmann machines. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume I, pages 282–317. MIT Press, Cambridge MA, 1986.

[31] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. In *Proceedings of the National Academy of Sciences*, volume 79, pages 2554–2558, 1982.

[32] J. J. Hopfield. Neurons with graded response have collective computational properties like those of two-state neurons. In *Proceedings of the National Academy of Sciences*, volume 81, pages 3088–3092, 1984.

[33] J. J. Hopfield, D. I. Feinstein, and R. G. Palmer. "Unlearning" has a stabilizing effect in collective memories. *Nature*, 304:158–159, 1983.

[34] J. J. Hopfield and D. W. Tank. "Neural" computation of decisions in optimization problems. *Biological Cybernetics*, 52:141–152, 1985.

[35] J. Hořejš. A view on neural networks paradigms development. *An ongoing survey starting in Neural Network World*, 1:61–64, 1991.

[36] J. Hořejš and O. Kufudaki. Počítače a mozek (neuropočítače). In *Sborník semináře SOFSEM*, Beskydy, 1988.

[37] Y. Chauvin. A back-propagation algorithm with optimal use of hidden units. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems I*, pages 519–526, Denver, 1988. San Mateo: Morgan Kaufmann, 1989.

[38] R. A. Jacobs. Increased rates of convergence through learning rate adaptation. *Neural Networks*, 1:295–307, 1988.

[39] S. Kirkpatrick, C. D. Jr. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[40] T. Kohonen. Correlation associative memory. *IEEE Transactions on Computers*, C-21:353–359, 1972.

[41] T. Kohonen. *Associative Memory — A System Theoretical Approach.* Springer-Verlag, New York, 1977.

[42] T. Kohonen. *Self-Organization and Associative Memory.* Springer-Verlag, Berlin, 1984.

[43] T. Kohonen and K. Ruohonen. Representation of associated data by matrix operations. *IEEE Transactions on Computers*, C-22:701–702, 1973.

[44] A. H. Kramer and A. Sangiovanni-Vincentelli. Efficient parallel learning algorithms for neural networks. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems I*, pages 40–48, Denver, 1988. San Mateo: Morgan Kaufmann, 1989.

[45] R. P. Lippmann. An introduction to computing with neural nets. *IEEE ASSP Magazine*, 4:4–22, 1987.

[46] W. A. Little and G. L. Shaw. Analytical study of the memory storage capacity of a neural network. *Mathematical Biosciences*, 39:281–290, 1978.

[47] L. Lukšan. *Metody s proměnnou metrikou.* Academia, Praha, 1990.

[48] S. Makram-Ebeid, J.-A. Sirat, and J.-R. Viala. A rationalized back–propagation learning algorithm. In *Proceedings of the International Joint Conference on Neural Networks*, volume II, pages 373–380, Washington, 1989. New York: IEEE Press.

[49] M. Marchand, M. Golea, and P. Ruján. A convergence theorem for sequential learning in two-layer perceptrons. *Europhysics Letters*, 11:487–492, 1990.

[50] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[51] R. J. McEliece, E. C. Posner, E. R. Rodemich, and S. S. Venkatesh. The capacity of the Hopfield associative memory. *IEEE Transactions on Information Theory*, 33:461–482, 1987.

[52] M. Mézard and J.-P. Nadal. Learning in feedforward layered networks: The tiling algorithm. *Journal of Physics A*, 22:2191–2204, 1989.

[53] M. L. Minsky. *Theory of Neural-Analog Reinforcement Systems and its Application to the Brain-Model Problem.* Ph. D. thesis, Princeton University, Princeton NJ, 1954.

[54] M. L. Minsky and S. A. Papert. *Perceptrons.* MIT Press, Cambridge MA, 1969.

[55] J. von Neumann. The general and logical theory of automata. In L. A. Jeffress, editor, *Cerebral Mechanisms in Behavior*, pages 1–41. Wiley, New York, 1951.

[56] J. von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, volume 34 of *Annals of Mathematics Studies*, pages 43–98. Princeton University Press, Princeton NJ, 1956.

[57] N. J. Nilsson. *Learning Machines.* McGraw-Hill, New York, 1965.

[58] D. B. Parker. Learning-logic: Casting the cortex of the human brain in silicon. Technical Report TR–47, Center for Computational Research in Economics and Management Science, MIT, Cambridge, 1985.

[59] E. Pelikán. On a neural network approach to the detection of characteristic events in signals. In *Proceedings of the 13th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 1251–1252, Orlando, Florida, 1991.

[60] P. Peretto. On learning rules and memory storage abilities of asymmetrical neural networks. *Journal de Physique Lettres*, 49:711–726, Paris, 1988.

[61] F. J. Pineda. Generalization of back-propagation to recurrent neural networks. *Physical Review Letters*, 59:2229–2232, 1987.

[62] F. J. Pineda. Dynamics and architecture for neural computation. *Journal of Complexity*, 4:216–245, 1988.

[63] F. J. Pineda. Recurrent back-propagation and the dynamical approach to adaptive neural computation. *Neural Computation*, 1:161–172, 1989.

[64] D. Plaut, S. Nowlan, and G. Hinton. Experiments on learning by back propagation. Technical Report CMU–CS–86–126, Department of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1986.

[65] R. Rohwer and B. Forrest. Training time-dependence in neural networks. In M. Caudill and C. Butler, editors, *Proceedings of the IEEE First International Conference on Neural Networks*, volume II, pages 701–708, San Diego, 1987. New York: IEEE Press.

[66] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386–408, 1958.

[67] F. Rosenblatt. *Principles of Neurodynamics*. Spartan Books, Washington DC, 1962.

[68] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume I, pages 318–362. MIT Press, Cambridge MA, 1986.

[69] D. E. Rumelhart and J. L. McClelland, editors. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition I & II*. MIT Press, Cambridge MA, 1986.

[70] R. Scalettar and A. Zee. Emergence of grandmother memory in feedforward networks: Learning with noise and forgetfulness. In D. Waltz and J. A. Feldman, editors, *Connectionist Models and Their Implications: Readings from Cognitive Science*, pages 309–332. Ablex, Norwood, 1988.

[71] T. J. Sejnowski and C. R. Rosenberg. Parallel networks that learn to pronounce English text. *Complex Systems*, 1:145–168, 1987.

[72] J. F. Shepanski and S. A. Macy. Teaching artificial neural systems to drive: Manual training techniques for autonomous systems. In D. Z. Anderson, editor, *Proceedings of the Neural Information Processing Systems Conference*, pages 693–700. New York: American Institute of Physics, 1988, 1987.

[73] P. K. Simpson. *Artificial Neural Systems: Foundations, Paradigms, Applications, and Implementations*. Pergamon Press, New York, 1990.

[74] J. Šíma. Neural expert systems. *Neural Networks*, 8:261–271, 1995.

[75] J. Šíma and R. Neruda. *Theoretical Issues of Neural Networks*. MATFYZPRESS, Prague, 1996.

[76] A. J. Ticknor and H. Barrett. Optical implementations of Boltzmann machines. *Optical Engineering*, 26:16–21, 1987.

[77] V. V. Tolat and B. Widrow. An adaptive 'broom balancer' with visual inputs. In *Proceedings of the International Conference on Neural Networks*, volume II, pages 641–647. New York: IEEE Press, 1988.

[78] T. P. Vogl, J. K. Mangis, A. K. Rigler, W. T. Zink, and D. L. Alkon. Accelerating the convergence of the back-propagation method. *Biological Cybernetics*, 59:257–263, 1988.

[79] W. Wee. Generalized inverse approach to adaptive multiclass pattern classification. *IEEE Transactions on Computers*, C-17:1157–1164, 1968.

[80] G. Weisbuch and F. Fogelman-Soulié. Scaling laws for the attractors of Hopfield networks. *Journal de Physique Lettres*, 46:623–630, Paris, 1985.

[81] P. J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.* Ph. D. thesis, Applied Mathematics, Harvard University, Cambridge, MA, 1974.

[82] B. Widrow. Generalization and information storage in networks of ADALINE "neurons". In M. C. Yovits, G. T. Jacobi, and G. D. Goldstein, editors, *Self-Organizing Systems*, pages 435–461. Spartan Books, Washington DC, 1962.

[83] B. Widrow and M. E. Hoff. Adaptive switching circuits. In *IRE WESCON Convention Record*, volume 4, pages 96–104. IRE, New York, 1960.